EVALUACION DEL PERSONAL DOCENTE

CURSO : Microsoft Visual C++

Del 19 al 30 Junio, 1995

Conferencista : Ing. Noe Alvarez Martínez

Marque con una "X" , su respuesta.

Los conocimientos del profesor sobre el curso son:

□ Excelentes □ Buenos □ Regulares □ Malos

Las preguntas de los alumnos las contestan con :

□ Mucha seguridad □ Seguridad □ Inseguridad

La clase se desarrolla en forma :

□ Muy interesante □ Interesante □ Aburrida

El método de enseñanza del profesor conduce a un aprendizaje :

□ Excelente □ Bueno □ Regular

La organización y desarrollo del curso es :

□ Adecuada □ Malo

La calidad del material utilizado es :

□ Excelente □ Bueno □ Regular □ Malo

Le agrado su estancia en la División de Educación Continua :

□ Si □ No, Diga porque!

Recomendaría el curso a otras personas :

□ Si □ No, Diga porque!

Medio del cual se entero de este curso ____________________

____

Centro de Información

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MICROSOFT VISUAL C ++

DIRECTORIO DE PROFESORES

19 de julio al 3 de julio de 1995

ING. NOE ALVAREZ MARTINEZ
ADMINISTRADOR DE LA RED
UNIDAD DE COMPUTO ACADEMICO
FACULTAD DE INGENIERIA, U N A M
CIUDAD UNIVERSITARIA
C.P. 04510 MEXICO, D.F.
TEL: 622 09 51

'pmc.

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

MATERIAL DIDACTICO

MICROSOFT VISUAL C++

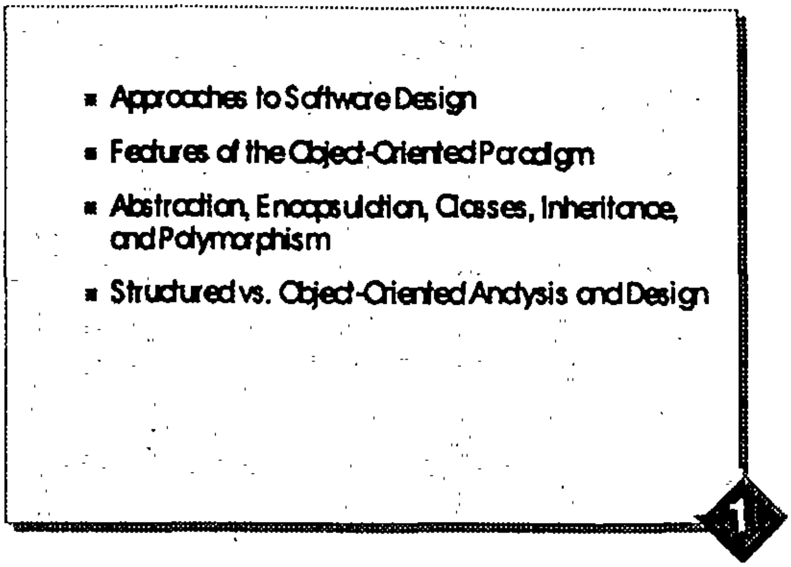
JUNIO 1995

Contents

Module 1: What Is Object-Oriented Analysis?

∑ Overview

Slide Objective
Provide an overview of the module contents. Establish the importance of understanding the new problem approach and new terminology.



- Approaches to Software Design
- Features of the Object-Oriented Paradigm
- Abstraction, Encapsulation, Classes, Inheritance, and Polymorphism
- Structured vs. Object-Oriented Analysis and Design

This is the first of two introductory modules. In this module and the next, you will examine the general concepts that are the framework for object-oriented software design and implementation.

These concepts serve to clarify the content of the course and help you determine your expectations. At the same time, the modules will provide examples and activities that contribute to your understanding of the overall picture. Once this foundation is laid, you will learn to actually read and use object-oriented code.

Module Summary

This module offers a description of object-oriented analysis and design (OOAD). The next one presents a general approach to OOAD. In both cases, the stage is being set for subsequent modules, in which you will develop and apply your skills.

As you go through this module, be thinking of an application design problem. As you begin to get a feel for what objects are, try to apply an object-oriented perspective to that design.

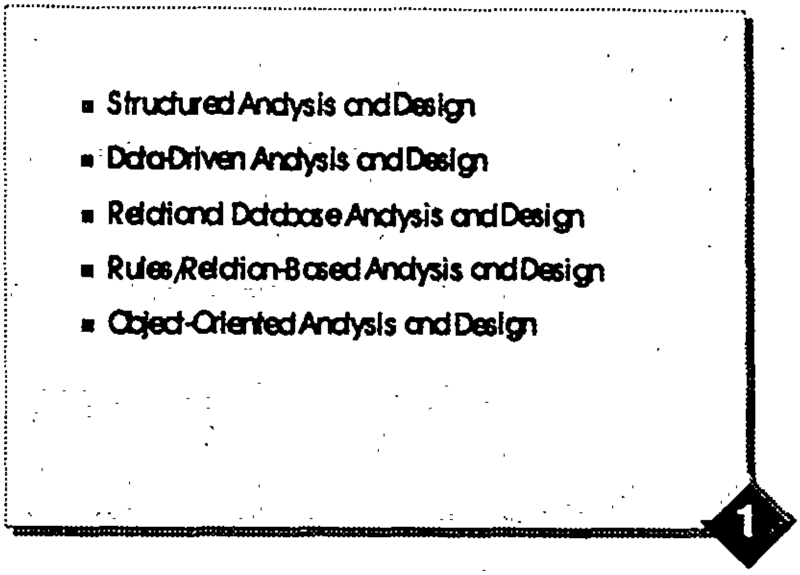
Objectives

At the end of this module, you will be able to:

- Discuss key software design approaches and issues.
- List methods for achieving software design goals.
- Discuss essential object-oriented analysis and design.
- Differentiate between the attributes of an object and its behaviors.
- Contrast procedural and object-oriented analysis.

Approaches to Software Design

Slide Objective
Briefly cover various approaches to software design and analysis noting that each is valid for various types of problems.



- Structured Analysis and Design
- Data-Driven Analysis and Design
- Relational Database Analysis and Design
- Rules/Relation-Based Analysis and Design
- Object-Oriented Analysis and Design

Analysis and Design (A/D)

Before any coding occurs, the first phase of software construction should be an analysis and design phase. This phase defines the logical problem domain—the problem that must be solved or the service that must be performed. The problem must be defined (analyzed) and modeled (designed) in terms that are transferable to a program coding style.

There are a number of generally accepted broad approaches or methodologies for analysis and design. Each is suited to a particular class of problem:

Structured A/D uses functional decomposition to arrive at a procedure-oriented approach to solving a problem. This is probably the most commonly used and flexible of all methodologies.

Data-driven A/D centers on records as they originate, change, and pass through a system. This approach is often used to model record-keeping, inventory, and material control systems. It is the other side of the coin to the structured approach.

Relation database A/D seeks to apply relations between attributes in a system to form a multi-dimensional table of values and connections.

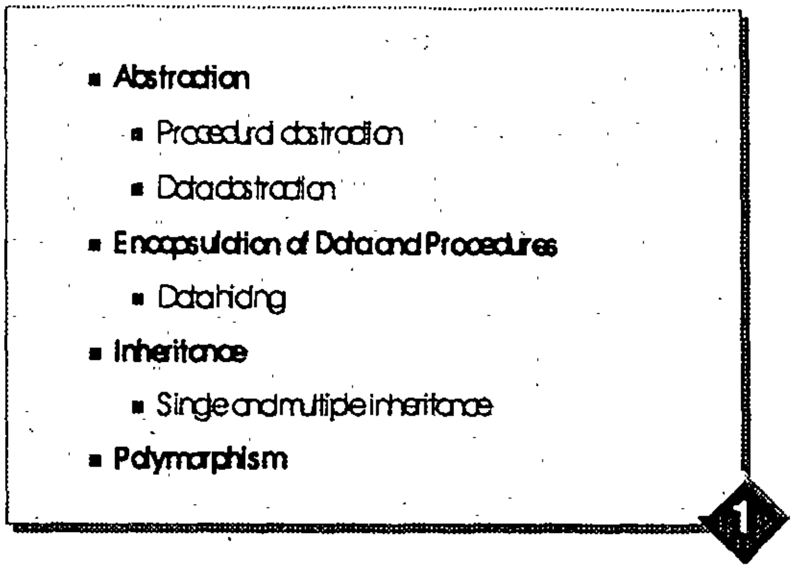
Rules- and relation-based A/D seeks to set up a series of logical relationships or rules to govern or describe a system behavior or structure. This is most commonly used in artificial intelligence (AI) and expert system applications.

Object-oriented A/D (abbreviated OOAD) identifies "actors" in the problem domain, the abilities or responsibilities of each actor, the relationships between the actors, and finally, the main script for the actors.

Computer languages are often designed (and better suited) for use with only one or a few of these A/D methodologies. Microsoft® Visual C++™ is a very flexible language, but it is best suited to the structured (procedural) and object-oriented approaches.

Features of the Object-Oriented Paradigm

Slide Objective
To approach O-O programming, introduce these 4 high-level features as characteristics of the O-O programming paradigm.



- **Abstraction**
  - Procedural abstraction
  - Data abstraction
- **Encapsulation of Data and Procedures**
  - Data hiding
- **Inheritance**
  - Single and multiple inheritance
- **Polymorphism**

What Are Objects?

As the phrase implies, objects are the basis for object-oriented programming. The notion of an object is familiar to all of us, and it translates well to the world of programming.

For our purposes, an object has an identity. It is defined by its attributes (data elements) and behaviors (functions). An object's attributes and behaviors make it distinct from other objects. In the language of object-oriented programming, objects represent things such as rectangles, ellipses, and triangles, as well as money, part numbers, and items in inventory.

The Object-Oriented Paradigm

Although there is no hard definition of what the object-oriented paradigm entails, most people agree that it encompasses at least four general concepts:

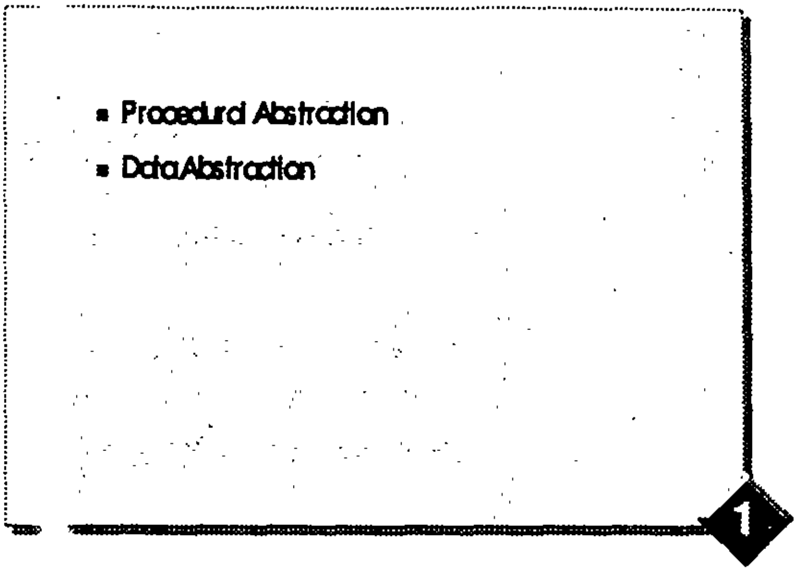
Delivery Tips
Introduce each item briefly to set terminology. Each will be covered on a following page.

- Abstraction allows users to ignore the implementation details and concentrate on a higher-level view of an entity. That is, object-oriented programming encourages the programmer to design in abstract terms.
- Encapsulation provides a grouping mechanism that describes the bundling of data and functions together within an object so that access to the data is permitted only through the object's own functions.
- Inheritance is a mechanism for automatically sharing functions and data among classes, subclasses, and objects.
- Polymorphism allows related objects to respond differently (but appropriately) when responding to the same message.

Important This course does not attempt to cover multiple inheritance or polymorphism as supported by Visual C++.

Abstraction

Slide Objective
Define two types of abstraction: Procedural provides behaviors while Data provides attributes for objects.



- Procedural Abstraction
- DataAbstraction

Abstraction is the capability to represent, denote and handle information at a higher level than is inherent to a computer or base language. For example, it is easier to work with records and processes than it is to work with a collection of integers, floating point numbers, and executable instructions. All high-level modem, languages support abstraction.

Procedural abstraction provides us with the *behaviors* of a system or entity. Global functions and member functions provide for procedural abstraction in C++.

Delivery Tips
Challenge students to achieve abstraction within all the problem domains presented during the week. Abstraction is a paradigm shift for procedural programmers.

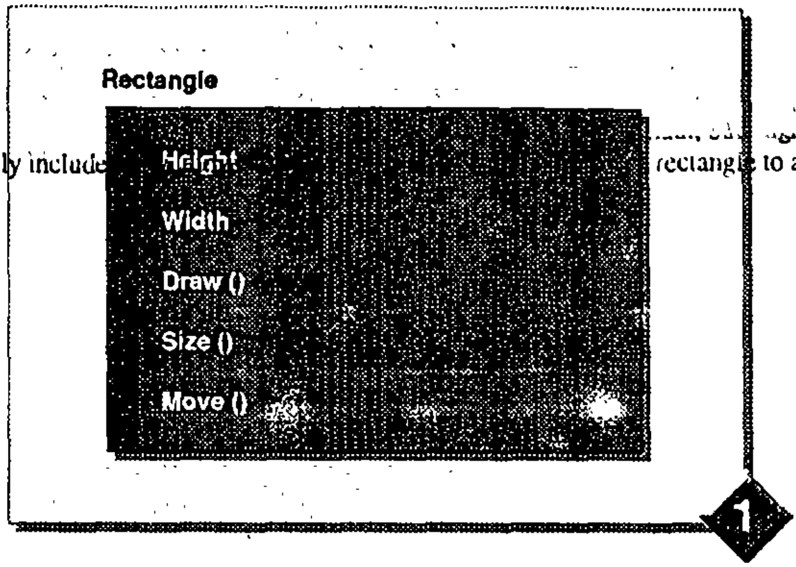
Data abstraction provides us with the *attributes* of an entity. The higher-level data types challenge students to work toward achieving abstraction with all the problem domains presented in the course. Abstraction is a major shift for procedural programmers.rays, pointers, structures, and classes particularly support data abstraction in C++.

Reference

Refer to "Fundamentals of Object-Oriented Design" in the *C++ Tutorial.*

Encapsulation

Slide Objective · Staying high-level, encapsulation groups related "information & processes" into a unit. Introduce a rectangle, describing its attributes and behaviors.



Rectangle

Height

Width

Draw ()

Size ()

Move ()

likely include

rectangle to another

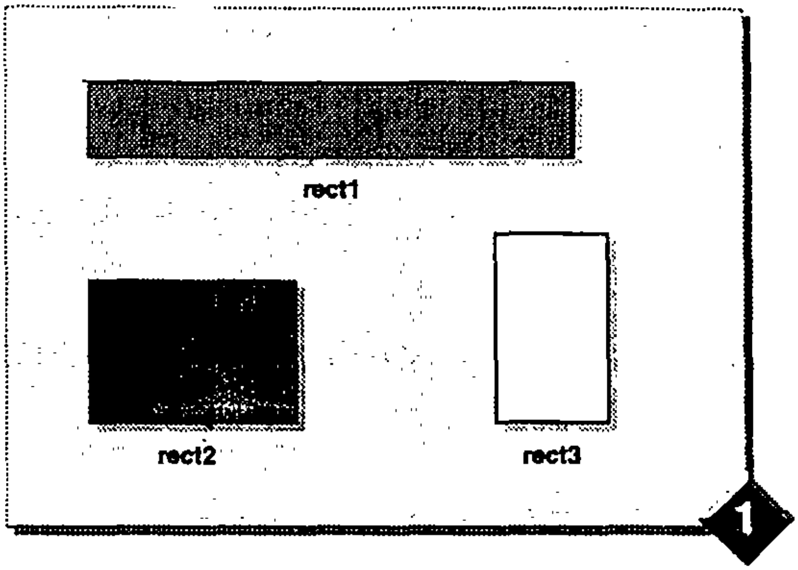
Encapsulation is the ability to group related pieces of information and processes into a self-contained unit. In many cases, it also allows data-implementation details to be hidden. (The software industry has learned the costly lesson that dependence on specific data-implementation schemes often hampers maintenance.) Encapsulation groups information and processes in the form of attributes and behaviors.

The *attributes* of a rectangle include its width, height, and location, and perhaps its color. Notice that other attributes, such as the perimeter and area, are redundant because they can be calculated by knowing the height and width, and knowledge of the fundamental nature of rectangles.

The *behaviors* of a rectangle largely depend on the problem domain, but might likely include draw, move, resize, rotate, reflect, and compare a rectangle to another shape.

Classes

Slide Objective
Interject the definition of a "class" to describe a category of related entities. Define an "instance" as one object from the category Rectangle.



What Are Classes?

A class names a category of related entities or objects. Each of those entities is called an object or instance of that class. Each object in a class is a particular example of a more general category.

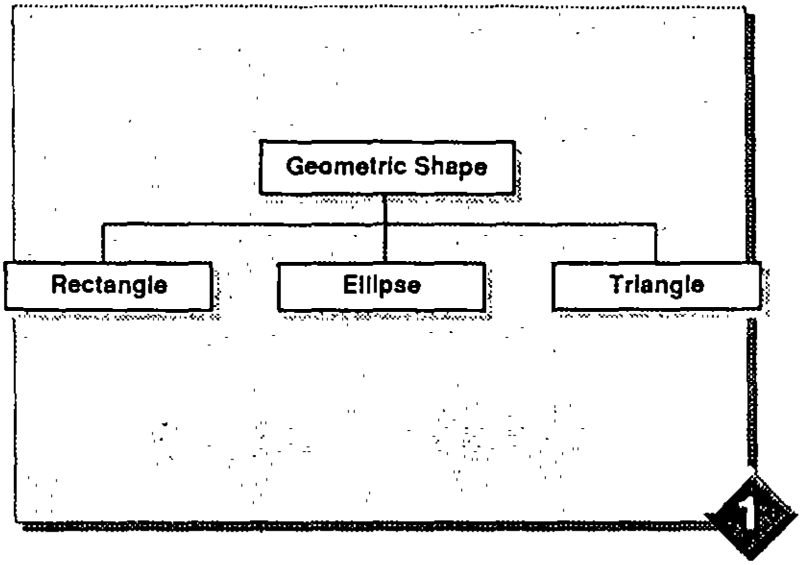
The class Rectangle includes any object that exactly meets the basic requirements of the rectangle category. The illustration shows three different rectangles. In object-oriented terms, rect1, rect2, and rect3 are objects of the class Rectangle.

Classes are recognized as a useful and widely used construct, even though they are not strictly required for OOAD or object-oriented language implementation. C++ directly uses the class construct for abstraction, encapsulation, inheritance and polymorphism.

Delivery Tips
OOA and OOD typically don't use the "Class" terminology—the implementation of the design does!

Inheritance

Slide Objective
The third major characteristic of OOP, "inheritance," allows a generalized grouping to show "**Is a type of**" relationships.



Geometric Shape

Rectangle

Ellipse

Triangle

1

What Is Inheritance?

Inheritance is a means for creating a new, more specific type from an existing, more general type. This is done by stating the difference between the two types. Inheritance defines one type as a subcategory of another.

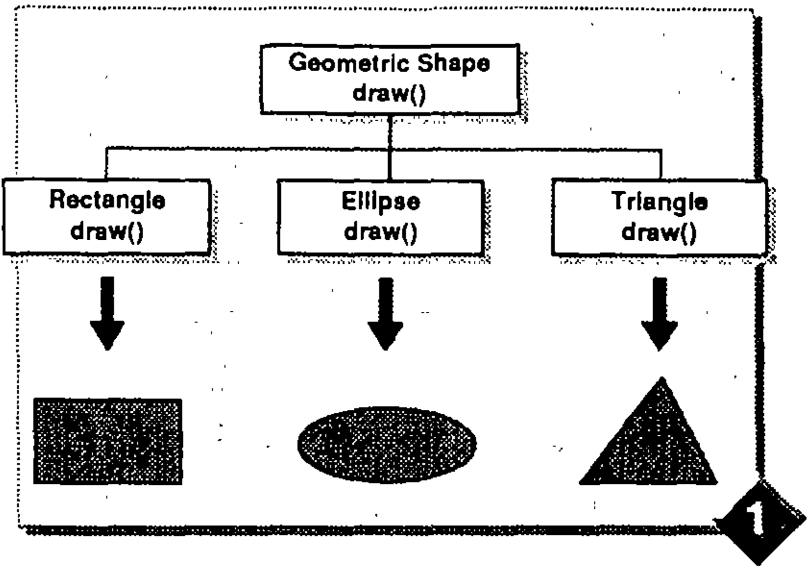
The general class is referred to as the *base* or *parent* class. A more specific class is referred to as the *derived* or *child* class.

The derived classes gain or inherit both attributes and behaviors from the base class.

The exact mechanism for inheritance will be covered in a later module.

Polymorphism

Slide Objective
The fourth characteristic of OOP is Polymorphism. Define only — course does not cover the topic.



Geometric Shape
draw()

Rectangle
draw()

Ellipse
draw()

Triangle
draw()

What Is Polymorphism?

For our purposes, polymorphism may be defined as the ability of related objects to respond to the same message with different, but appropriate, actions.

In the example above, each shape class has its own version of the **draw** function that provides the appropriate action for an object of that class. A Rectangle object's **draw** function displays a rectangle, an Ellipse **draw** function displays an ellipse, and so on.

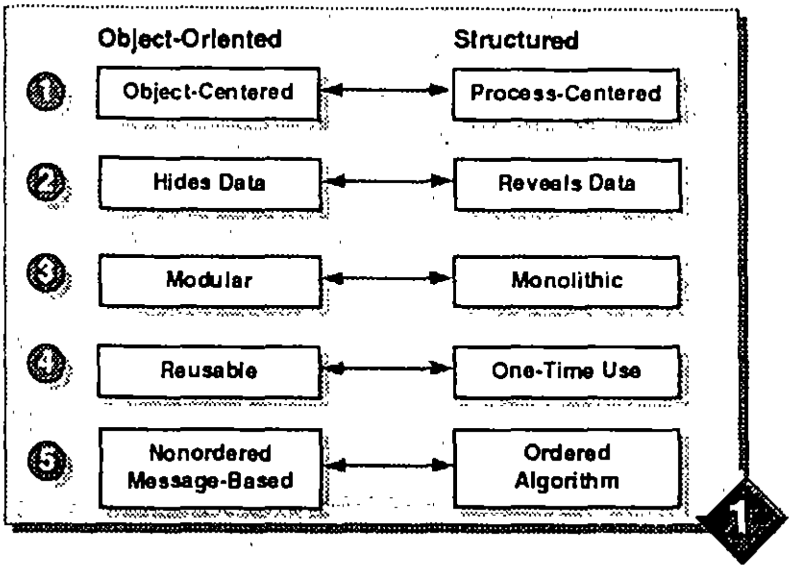
Delivery Tips
As a shift for procedural programmers, the abstract viewpoint says "each object knows how to draw itself."

What this means to the programmer is a simpler, more flexible interface to a group of related objects.

Polymorphism is implemented in C++ through virtual functions. (An explanation of virtual functions falls outside the scope of this course.)

Structured vs. Object-Oriented A/D

Slide Objective
Contrast Procedural or Structured Design techniques vs. O-O Analysis and Design. Both are valid.



	Object-Oriented		Structured
1	Object-Centered	↔	Process-Centered
2	Hides Data	↔	Reveals Data
3	Modular	↔	Monolithic
4	Reusable	↔	One-Time Use
5	Nonordered Message-Based	↔	Ordered Algorithm

Structured vs. Object-Oriented Design

Since most programmers are trained in the structured, procedural approach, it behooves us to compare objected-oriented approaches with structured approaches.

The first point is that OOAD focuses on objects that have certain behaviors and attributes; structured A/D focuses on a hierarchy of processes.

Secondly, object-oriented implementations hide data, showing only behaviors. The structured approach leaves this decision up to the implementor.

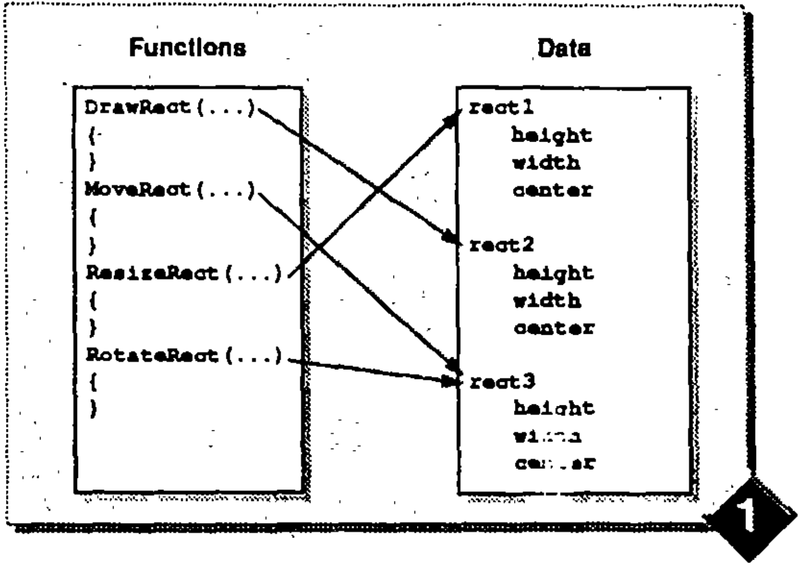
Delivery Tips
Relate the third and fourth points. Objects are modular and reusable and they send messages to other objects.

The next two points are closely related. Since objects are by definition modular in their construction (that is, they are complete in and of themselves), they tend to be highly reusable. Structured processes may or may not be reusable, again depending on the implementation.

Finally, object-oriented applications are constructed on a message-based or event-driven paradigm where objects send messages to other objects. Structured approaches with processes tend to result in linear, algorithm-based implementations.

Structured Approach to Design

Slide Objective
Contrast approaches: Part 1 of 2.
Tends to disassociate processes from data.
Leads to increasing complexity.



Functions

```
DrawRect(...)
{
}
MoveRect(...)
{
}
ResizeRect(...)
{
}
RotateRect(...)
{
}
```

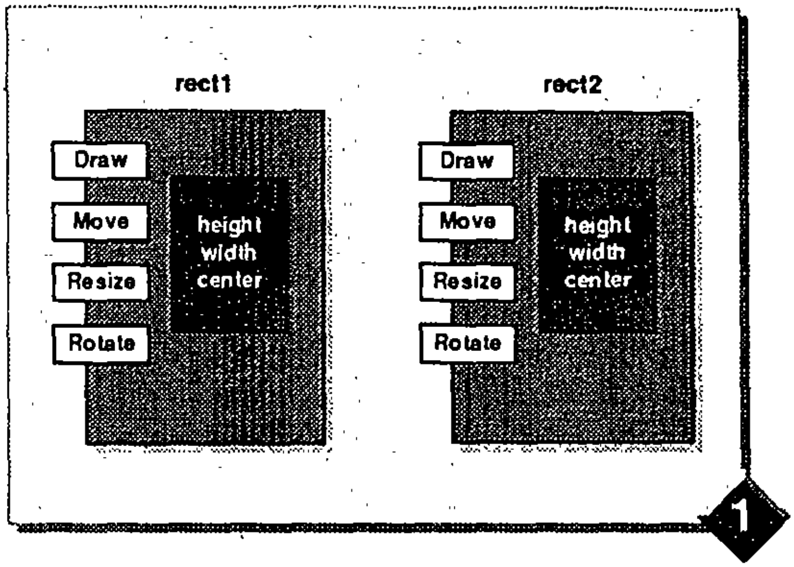
Data

```
rect1
    height
    width
    center
rect2
    height
    width
    center
rect3
    height
    width
    center
```

The traditional structured approach to design tends to disassociate logical processes (functions) from the information (data) they work on. As the number and complexity of the processes and information increase, a very real danger exists that the pictured relationship network becomes too complex to be managed by mere mortals.

Object-Oriented Approach to Design

Slide Objective
Contrast approaches: Part 2 of 2. Tightly joins processes with data. Reduces complexity, increases modularity.



The object-oriented paradigm groups processes and information together as a unit (classes and their objects). The information in these units is typically hidden, being revealed by an interface or set of behaviors.

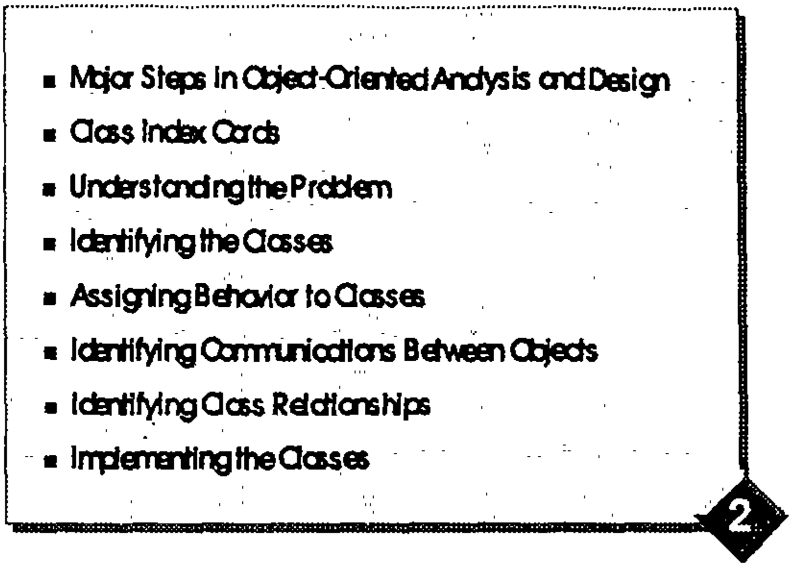
A Final Word

After some practice, most people find the OOAD approach much more natural than other methodologies. This is because it meshes very well with the way people naturally interpret the world. Human understanding largely rests on identification and generalization (objects and classes), finding relationships between groups (containment and inheritance), and interacting through the normal interface of an entity (behaviors).

Module 2: A General Approach to Object-Oriented Analysis and Design

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Major Steps in Object-Oriented Analysis and Design
- Class Index Cards
- Understanding the Problem
- Identifying the Classes
- Assigning Behavior to Classes
- Identifying Communications Between Objects
- Identifying Class Relationships
- Implementing the Classes

Module Summary

In this module, we examine a general approach to OOAD by looking at many of its elements. You will be introduced to basic steps and methodologies, as well as the concepts of class behaviors and relationships.

Much of this information is presented in parallel with a class activity: implementing a simple graphics program. As you go through this module, remember that designing and implementing classes is really creating user-defined abstract data types.

Objectives

At the end of this module, you will be able to:

- Characterize objects in design terminology.
- Describe the object-oriented design process.
- Describe messaging between objects.
- Define inheritance.

Lab

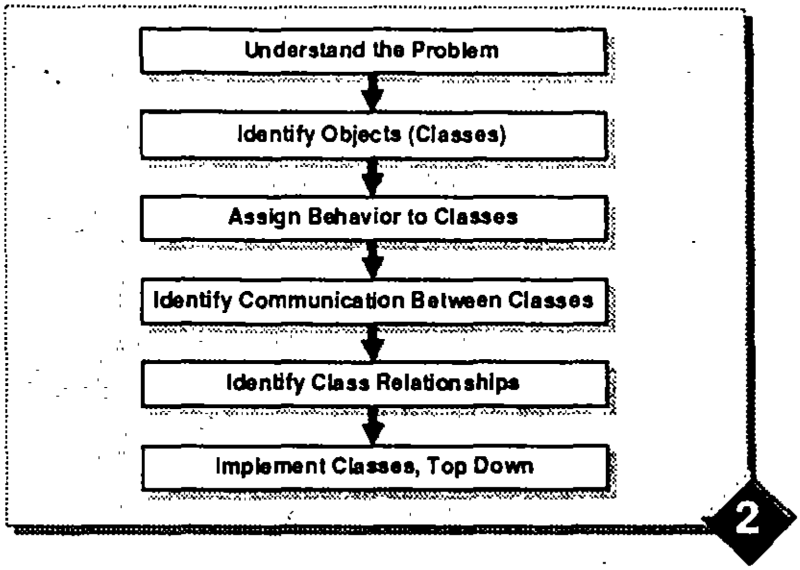
Fundamentals of Object-Oriented Design

Reference

Refer to "Fundamentals of Object-Oriented Design" in the *C++ Tutorial.*

Major Steps in OO Analysis and Design

Slide Objective
Provide an overview of the general steps in the OOAD process.



Understand the Problem

Identify Objects (Classes)

Assign Behavior to Classes

Identify Communication Between Classes

Identify Class Relationships

Implement Classes, Top Down

Delivery Tips
Don't go too deep into each area. An upcoming page covers each block.

Key Points
Design is an iterative process; it is not linear steps.

Although there are a number of formal object-oriented analysis and design methodologies being developed, most share a common flavor in their approach to OOAD. In this module, we will follow a general, high-level approach. (In the standard development cycle of software, implementation is not a part of A/D.) The last phase of the cycle, testing, is not shown or considered in this course.

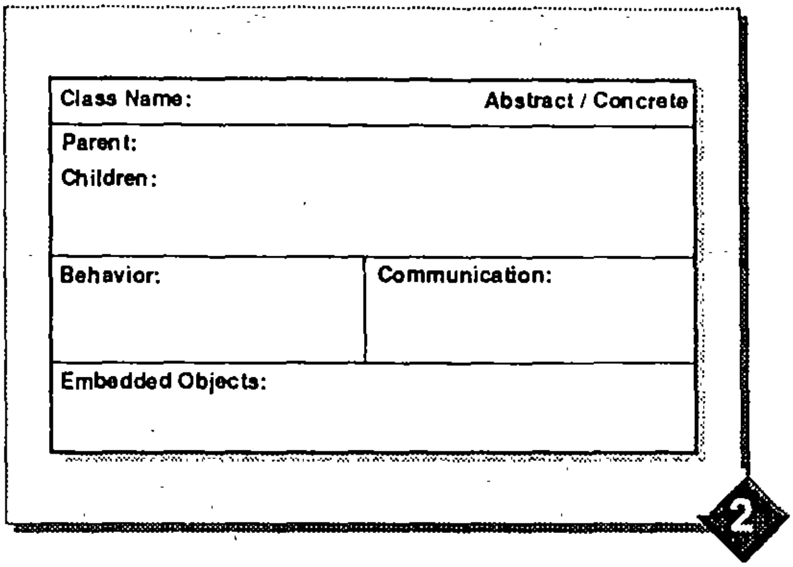
Although the steps are shown in a linear order, they represent an iterative, overlapping process of constant refinement. In contrast to the structured approach, the analysis and design phases of OOAD tend to consume a greater portion of the development cycle.

When Visual C++ is used, the result of this process, should be a set of classes that describe the actors or objects in the original problem domain. Since each class should completely encapsulate an actor, the ideal is for each to "stand on its own," and thus be portable.

Classes should be internally cohesive, and have narrow, loosely coupled external interfaces.

Class Index Cards

Slide Objective
Introduce a variation of Wirfs-Brock, *et al.* CRC cards as a tool used during the design process.



Class Name:	**Abstract / Concrete**
Parent: **Children:**	
Behavior:	**Communication:**
Embedded Objects:	

Class index cards are a useful device for aiding the A/D process. They have slots for the following information:

Class name: the name of the class. By convention the first letter of each word is capitalized.

Abstract/Concrete option: If objects of a class are to be created, a class is said to be *concrete*; if no objects of that type are to be created, the class is *abstract*. Base classes are sometimes abstract.

Parent: the name of the parent class, if any

Children: the name of child classes, if any

Behavior: a list of interface functions

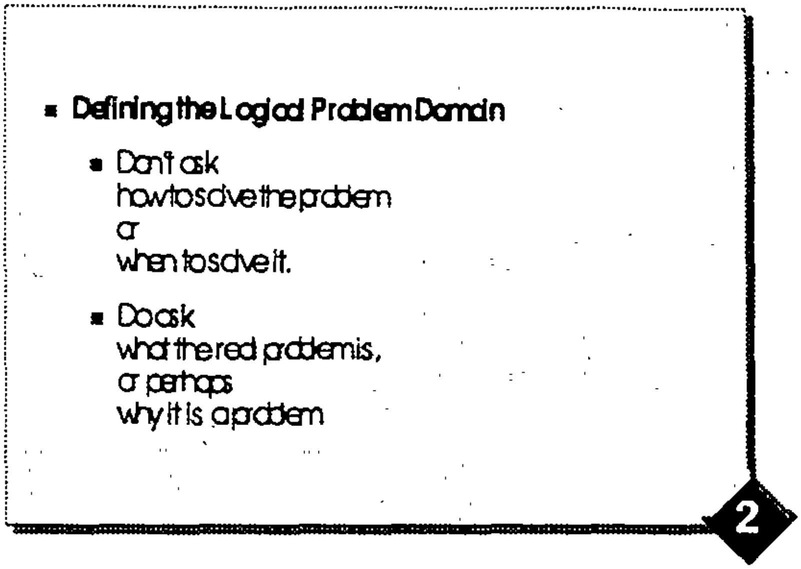
Communication: a list of all other classes on whose behaviors this class relies

Embedded objects: a list of all user-defined objects that are contained in objects of this class

The concept of class index cards is a slightly altered form of CRC cards, championed by Wirfs-Brock, Wilkerson, and Wiener in *Designing Object-Oriented Software*.

Understanding the Problem

Slide Objective
Major Step #1: Define the problem domain in terms of what (and perhaps why).



- **Defining the Logical Problem Domain**
  - Don't ask
    how to solve the problem
    or
    when to solve it.
  - Do ask
    what the real problem is,
    or perhaps
    why it is a problem

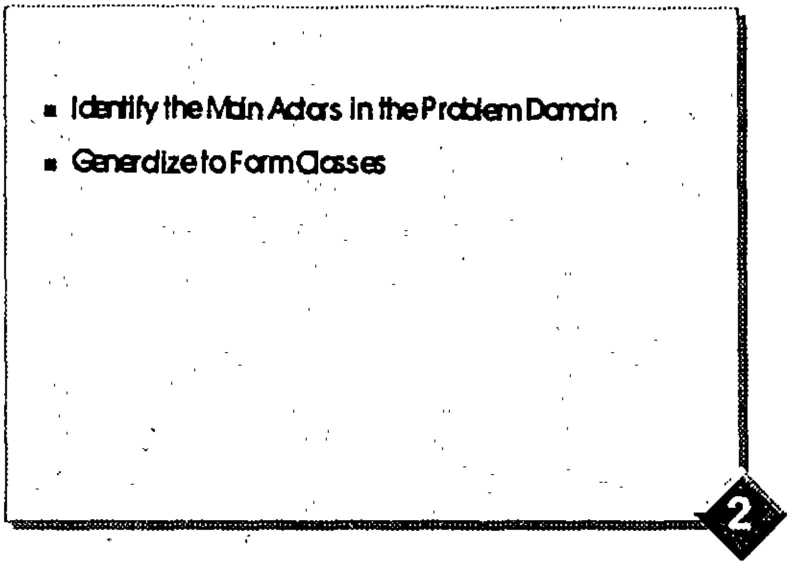
The first and foremost step in any analysis process is to identify the problem that must be solved or the service that is needed. The problem should be conceptualized in logical space, since its solution will be implemented on a computer. The question is not yet how or when to solve the problem. Instead, ask what the real problem is, or perhaps why it is a problem.

Improper definition is the first step on the road to ruin, regardless of whether it is caused by defining a problem too narrowly, too broadly, or missing the target altogether.

If there is more than one person on a development team, all must agree on the problem definition.

Identifying the Classes

Slide Objective
Step #2: Identify the actors or items in the problem domain using nouns.



- Identify the Main Actors in the Problem Domain
- Generalize to Form Classes

Basic Steps in Identifying Classes

Once the dimensions of a problem are understood, the next step is to identify what important actors (objects) are involved. Good candidates usually have the following characteristics:

- Noun (or verb that can be made into a noun—spooler, for example)
- They serve several useful purposes in the problem domain.
- They represent a discrete, stand-alone concept.

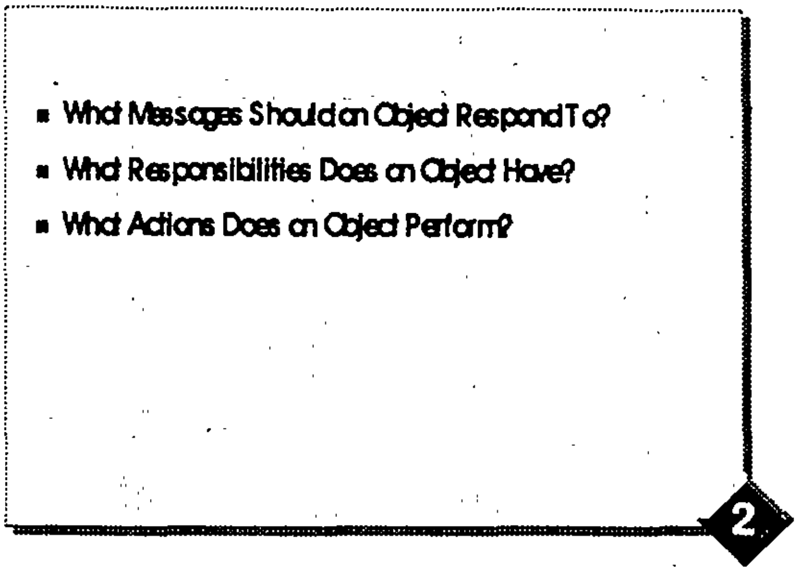
For Your Information
This process is Part 1 in the upcoming exercises.

Perhaps the best way to start this process is to list all likely nouns on a blank sheet of paper. Then use the criteria above to qualify likely candidates.

Even though the process described above is for specific actors or instance objects, it is normally a short trip to identify the general classes these actors belong to. For example, if a problem domain calls for a small pink rectangle, a large blue rectangle, and a medium gray rectangle, obviously the class Rectangle is required.

Assigning Behavior to Classes

Slide Objective
Major Step #3: Answer these three questions to define an object's behavior.



- What Messages Should an Object Respond To?
- What Responsibilities Does an Object Have?
- What Actions Does an Object Perform?

2

Delivery Tips
A possible approach: Imagine holding the object in your hand and having a conversation about its behavior in the problem domain.

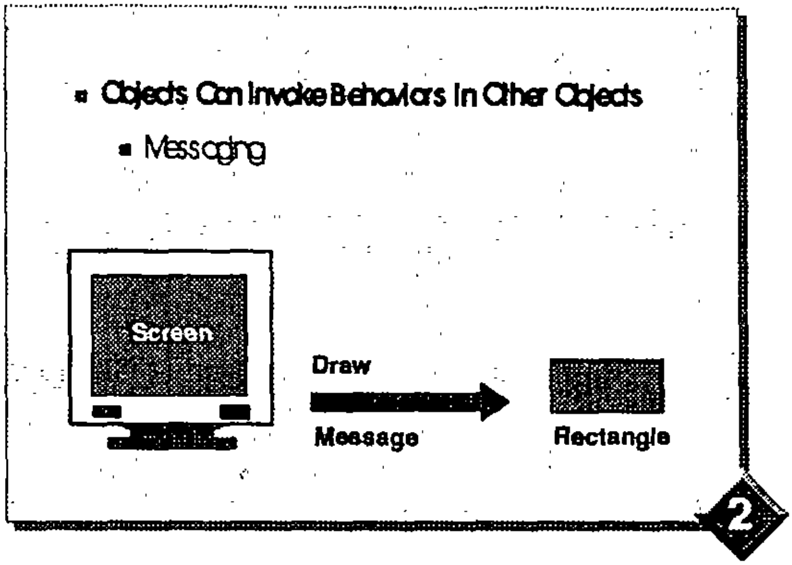
The answers to all three of these questions contribute equally to the assignment of class behaviors. (All objects of the same class have the same possible behaviors.) Normally all object behavior is directed at maintaining itself.

For example, what behaviors does a rectangle have? Again it depends on the problem domain, but assuming that we are working on a graphics display application, a rectangle would probably be expected to perform the following actions on itself: to **draw**, **move**, **resize**, **rotate**, **reflect**, or to **compare** itself to another object.

There may be many processes that affect the object that are not direct behaviors of that object. For example, although video mode certainly affects the way a rectangle is displayed, this behavior more properly belongs to the class (video) Screen.

Identifying Communications Between Objects

Slide Objective
Major Step #4: Identify the requests an object might receive from (or make to) another object.



- Objects Can Invoke Behaviors In Other Objects
  - Messaging

Communication

Key Points
A communication is a request for an object to perform a behavior.

In an object-oriented application, objects commonly invoke behaviors in other objects. The request for action that is directed at an object is called "sending a message." In C++, it is also called "invoking a member function."

As part of doing so, it might send a message to an on screen rectangle (by invoking the **Draw** function) so that the rectangle redraws itself.

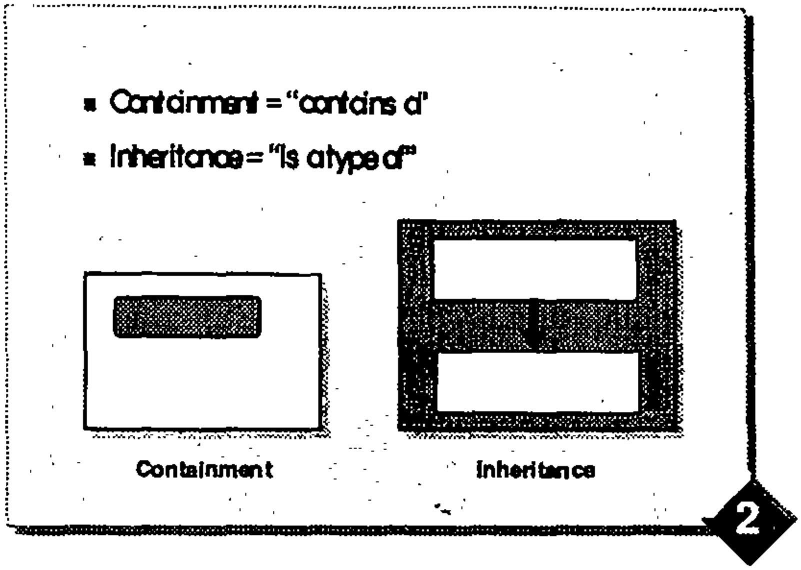
For example, in our graphics application, the screen object might be required to refresh itself. As part of doing so, it might send the **Draw** message to an on-screen rectangle to draw itself.

For Your Information
Spoof: Ask not what you can do for your object, but what your object can do for you.

Note again that although objects are the actual actors in a C++ application, this message-passing association is actually encoded into the respective classes.

Identifying Class Relationships

Slide Objective
Major Step #5: Use these phrases to determine relationships between objects.



- Containment = "contains a"
- Inheritance = "is a type of"

Containment and Inheritance

Containment is also called composition or embedding. Containment is where one object contains, is composed of, or owns an object of another class. For example, each rectangle contains a center point.

By contrast, inheritance is where one class is a type of or a kind of another class. For example, a rectangle is a type of geometric shape.

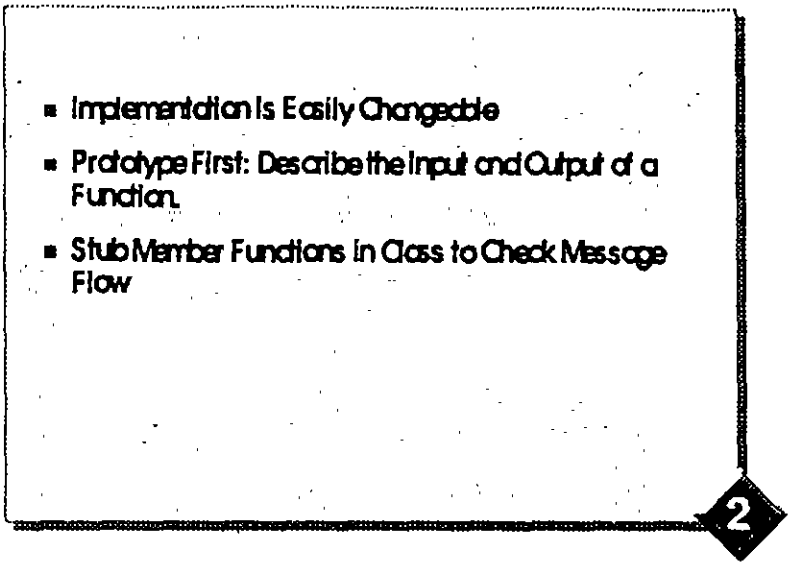
A class hierarchy may form a tree of relationships. In the previous module, we saw that Rectangle had a parent (Geometric Shape) and two siblings (Ellipse and Triangle).

As you will see, one (or more than one) level of containment or inheritance is possible. For example, a square is a type of rectangle, which is a type of geometric shape.

Tip It is a common mistake for beginners to confuse these two relationships, thereby creating interface problems later in the design and implementation phases.

Implementing Classes

Slide Objective
Major Step #6: Leaving the OOAD arena, implementation enters the OOP field.
Use these steps to get started, top-down.



- **Implementation Is Easily Changeable**
- **Prototype First: Describe the Input and Output of a Function**
- **Stub Member Functions in Class to Check Message Flow**

The last concern in OOAD is choosing an implementation for the various classes, including a data representation for each class. It is possible to delay implementation choices because the object-oriented approach concentrates on behaviors while hiding data. Therefore, as long as the interface does not change, implementation remains flexible and mutable. Another way of stating this is to say that each actor represents a black box: its behavior is known, but its internal workings (perhaps including state) remain a mystery.

Key Points
Design is an iterative process.
Behaviors might be tuned.
New communication needs are frequently added.

Often at this phase (or any previous phase), shortcomings will be noted from previous phases, and the OOAD cycle will repeat itself. This is natural and should be expected and encouraged. Rarely is a complete and elegant design accomplished on the first pass.

Prototyping the interfaces for a class involves writing the prototypes for each member function. This entails naming and defining each one, and specifying the type of data it takes and returns. (This topic will be dealt with more fully in the modules on functions and classes.) Next, in order to check the message flow between classes, it is useful to stub each function. This entails adding a simple "message out" statement for the body of each member function.

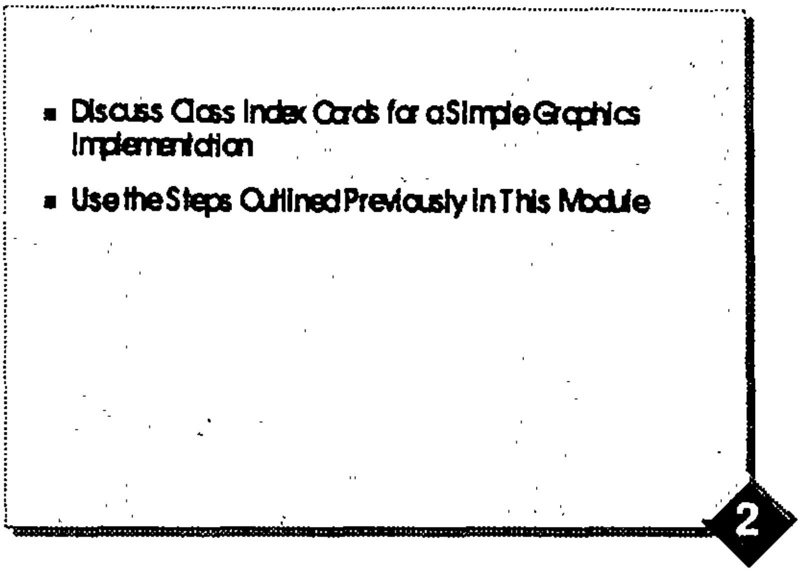
After an acceptable class design is conceptualized, the following phases must still be completed:

- Full class implementation
- Overall program implementation (scripting for actors)
- Testing and documentation

Note that these phases may be carried out in overlap or in parallel.

Class Activity

Slide Objective
Instructor Lead Walkthrough: Describe use of cards to define GeoShape classes.



- Discuss Class Index Cards for a Simple Graphics Implementation
- Use the Steps Outlined Previously In This Module

Class Activity

This activity applies the steps you have learned. You will solve a problem by developing the elements of a simple object-oriented design.

Step 1: Understand the Problem

You will develop a set of classes to implement a simple graphics program. The program must be able to display three different kinds of geometric objects on the video screen: ellipses, rectangles, and triangles. Also, it must allow the objects to be moved, resized, and have their color changed.

In addition, objects need to be managed somehow. For example, objects may be partially or fully moved off the physical video screen and may need to be clipped. At a later date, it might be desirable to change the video mode resolution and other screen attributes. For that purpose, we suggest a video screen class.

Use the class index cards on the following pages to design a set of classes that will meet the requirements stated above.

Step 2: Identify the Classes

To identify the actors in the problem domain, it is often helpful to start with a blank sheet and quickly write down the likely candidates:

Triangle	Keyboard	Ellipse	Rectangle	Line
Point	Screen	Array	Color	Draw

From this potential list, eliminate unlikely candidates and promote likely ones. Here Keyboard and Array can be eliminated from the initial design because they represent physical and data type implementation classes. They are implementation details. Draw is actually a behavior or function of a group of objects, and is not a class. Line, Point, and Color are attributes of the geometric shapes. At the moment, it is hard to say which of these are useful enough and complex enough to qualify as classes. For now, we think of Point as a likely candidate.

Based on the problem, it seems that the remaining four—Triangle, Ellipse, Rectangle, and Screen—are strong class candidates.

Step 3: Assign Behavior to Classes

Our problem description prescribes most of the required behaviors for the geometric shapes: **Draw**, **Move**, **Size**, **SetColor**, and so on.

The Screen class is useful for several purposes. First, shapes must be drawn on some surface, and this surface itself might have attributes and behavior: color, dimensions, ratio, et cetera. Also, a common problem associated with drawing individual objects is keeping track of interactions between shapes. For example, when one shape moves, it might uncover another that will have to be redrawn. You might choose to put this knowledge at the Screen class level.

The Point class has a very simple interface composed of get and set functions.

Steps 4 and 5: Identify Communication Between Objects and Identify Class Relationships

Which objects of which classes need cooperation from other classes? Well, each shape has a center (contains Point), so when a shape moves, that center must be changed (communication). And if the screen is to manage shapes, it must be informed when a shape is created or when it changes position or size. If that is the case, it would be beneficial to be able to update the view by having the Screen class send a message to all current shape objects so that they draw themselves.

Note that for all communications, the corresponding class must have that invoked behavior.

At this point, you also factor out the common behaviors and attributes of Triangle, Ellipse, and Rectangle, and place them in a common base class, Geometric Shape.

Step 6: Implement the Classes.

Using the approach outlined above, the cards might look like this:

Class Name: Geometric Shape	[✓ Abstract]/Concrete
Parent: Children: Rectangle, Ellipse, Triangle	

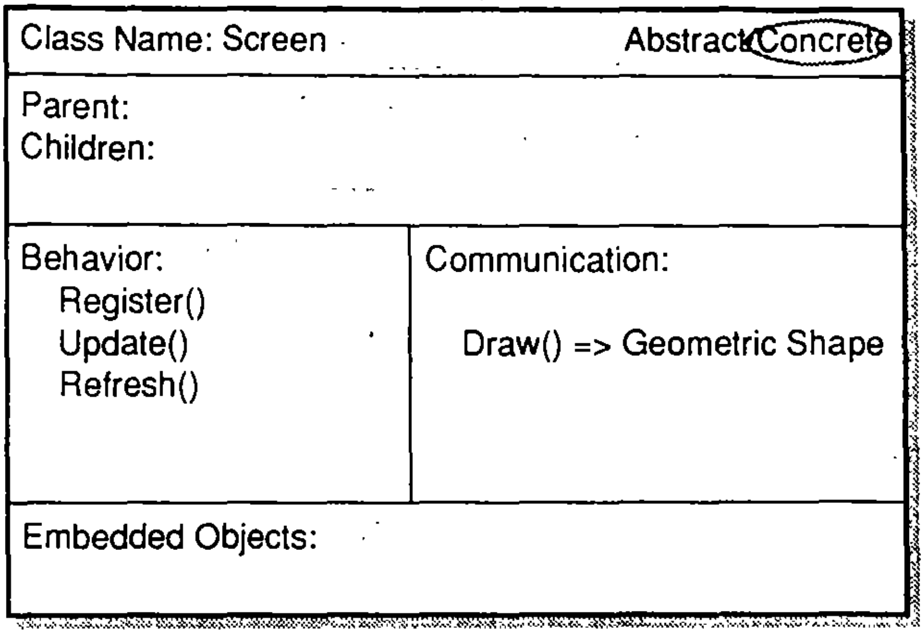
Behavior:
Draw()
Move()
Size()
SetColor()
(etc.)
	Communication:
	Setx() => Point
	Sety() => Point
	Register() => Screen
	UpDate() => Screen

Embedded Objects: Center Point (for object center)	

Class Name: Rectangle	Abstract/[✓ Concrete]
Parent: Geometric Shape Children:	

Behavior:
(see Geometric Shape)
SetHeight()
SetWidth()
	Communication:
	(see Geometric Shape)

Embedded Objects: (see Geometric Shape)	



Class Name: Screen Abstract/[✓ Concrete]

Parent:
Children:

Behavior:
Register()
Update()
Refresh()

Communication:
Draw() => Geometric Shape

Embedded Objects:

Class Name: Ellipse	Abstract/[✓ Concrete]
Parent: Geometric Shape Children:	

Behavior:
(see Geometric Shape)
SetMajorDia()
SetMinorDia()
	Communication:
	(see Geometric Shape)

Embedded Objects: (see Geometric Shape)	

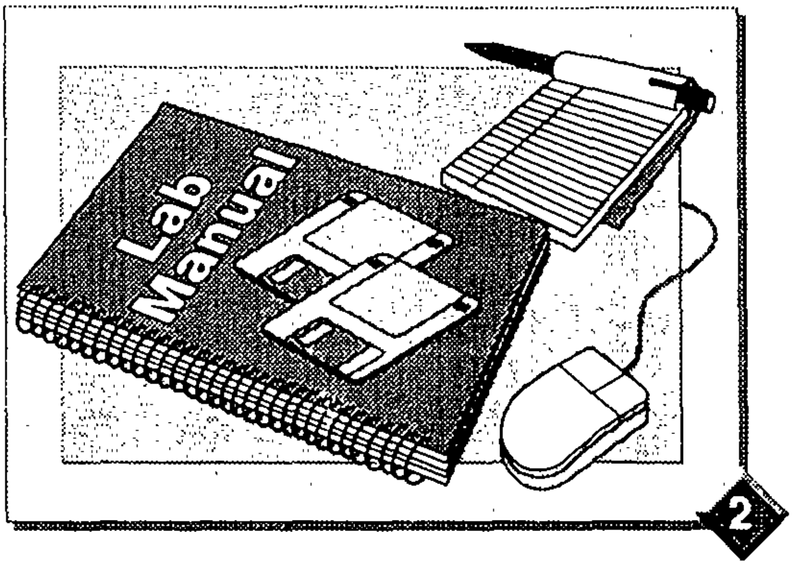
Class Name: Point	Abstract/[✓ Concrete]
Parent: Children:	

Behavior:
Setx(), Getx()
Sety(), Gety()
Delta()
	Communication:

Embedded Objects:	

Lab 1: Fundamentals of Object-Oriented Design

Slide Objective
Introduce the practice exercises. Query the students to find experience with Inventory, MRP, Purchasing or Sales Order. Depending on responses, group students into small design teams.

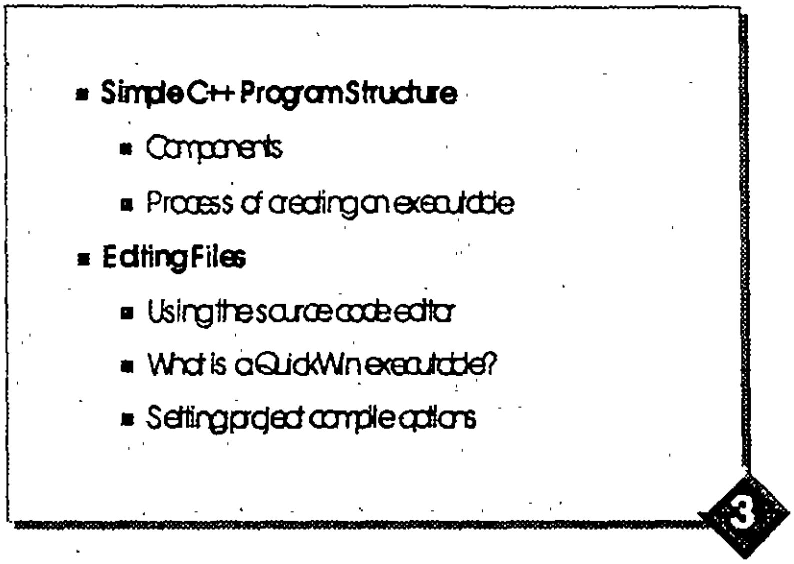


Module 3: The Basics

∑ Overview

Slide Objective
Provide an overview of the module contents.

Move quickly, remain high-level.



- **Simple C++ Program Structure**
  - Components
  - Process of creating an executable
- **Editing Files**
  - Using the source code editor
  - What is a QuickWin executable?
  - Setting project compile options

This is the first of four modules that explain the fundamentals of the Visual C++ language.

Module Summary

In this module you'll build your first program. This module will form the foundation for most of the rest of this course, as well as all the Visual C++ programming you will do from this point forward.

Key Points
Cover objectives to level-set student expectations.

Objectives

At the end of the module, you will be able to:

- Edit source code.
- Build a simple QuickWin executable.
- Use context-sensitive Help to obtain information about the C++ language.
- Write preprocessor directives.
- Create a main function.

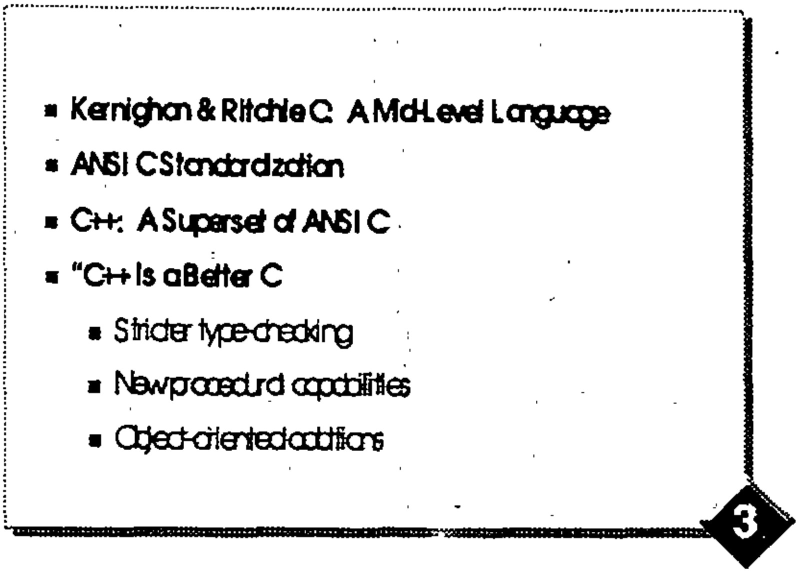
Delivery Tips
Module covers three major areas:
• Anatomy of a C++ source file
• VC++ Development Environment at a high level
• C++ Statements and keywords

Lab

The Basics

The Roots of C/C++

Slide Objective
Cover quickly to set history of language.



- Kernighan & Ritchie C: A Mid-Level Language
- ANSI C Standardization
- C++: A Superset of ANSI C
- "C++ Is a Better C
    - Stricter type-checking
    - New procedural capabilities
    - Object-oriented additions

The C language was developed by Brian Kernighan and Dennis Ritchie at AT&T Bell Labs in the early 1970s. Their goal was to produce a portable, efficient, flexible language, that would maintain the capabilities of a high-level, procedural language like Pascal, but still allow some of the "close to the machine" capabilities of assembly language. This original version, now known as K&R C, was later standardized, with slight modification by the American National Standards Institute (ANSI) Committee X3J11. C was first used as a systems language—UNIX®, Microsoft Windows, Windows NT™, OS/2®, and the Mac® operating system are largely written in C—but it later became popular as an applications language also. Today it is the most portable of all computer languages.

In the early 1980s, Bjarne Stroustrup at AT&T Bell Labs used C as the bedrock of a new language that came to be known as C++. C++ is largely a superset of ANSI C, with additional features at both the procedural and object-oriented level:

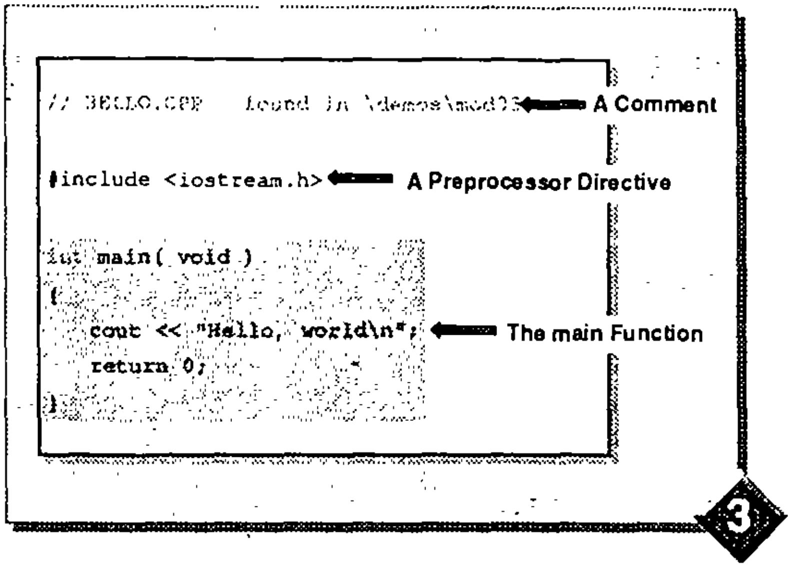
Key Points
Cover language features briefly —no detail. Most terminology is new to students.

- Stricter type-checking guards against inadvertent errors caused by badly mismatched data types. C++ is stricter than ANSI C.
- C++ adds powerful new procedural capabilities such as inline functions, function overloading, and default argument values.
- C++ supports the OO paradigm mainly through the *class* construct, which is an extension of the *structure* construct in C.

C++ is still a new language. While there is a standing International Standards Organizations ANSI committee (X3J16) in the process of standardizing C++, the current reference work on C++ is *The Annotated Reference Manual*, by Bjarne Stroustrup and Margaret Ellis. As of this writing, the newest version of the language is AT&T release 3.0.

Anatomy of a Simple C++ Program

Slide Objective
Identify major characteristics of program code.



```
// HELLO.CPP    found in \demos\mod3

#include <iostream.h>

int main( void )
{
    cout << "Hello, world\n";
    return 0;
}
```

Comments `//HELLO.CPP  found in \demos\mod3`

In C++, code is annotated with comments like this one. Two styles can be used. Comments that occupy multiple lines are typically enclosed within forward slashes and asterisks: /* <comment> */. Single-line comments begin with double slashes and continue to the end of the physical line: //<comment>.

```
/* This is a comment! */
. . .; //This is a comment, too!
```

Tip Comment your code liberally.

Preprocessor Directives `#include <iostream.h>`

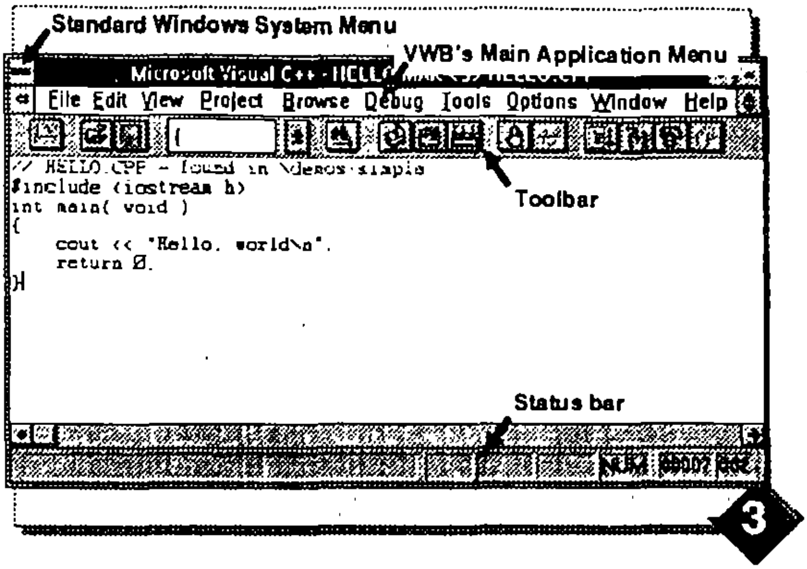
These are instructions for the preprocessor, which reads all of the source code before the compiler starts to create binary code. It performs a number of editorial tasks, such as stripping out comments, searching and replacing tokens, and adding code from other files. In the **#include** statement above, the preprocessor is adding information about the cout object used in the body of the main function. (This module will cover preprocessor directives in more detail.)

The main Function

The main function is the entry point in a C++ program. It is the first section of code to be executed. When the main function returns, your program terminates execution and control passes back to the operating system. Every C++ program must have one and only one main function. In this program, the main function requires no arguments (void) and returns an integer. For that reason, the last line in the program is return 0.

Fundamentals of Editing Source Files

Slide Objective
Cover basic interface of Visual Workbench. Depth depends upon student experience with Windows interface.


Standard Windows System Menu
VWB's Main Application Menu
Microsoft Visual C++ - HELLO
File Edit View Project Browse Debug Tools Options Window Help
Toolbar
Status bar
NUM 00001 001
3

```
// HELLO.CPP - found in \demos\simple
#include <iostream.h>
int main( void )
{
    cout << "Hello, world\n";
    return 0;
}
```

The Visual Workbench is an integrated source editor, compiler, and debugger. It is a Windows™-hosted application that behaves according to the Microsoft Windows Application User Interface Guidelines. It uses the multiple-document interface, which means that more than one source file can be open at a time.

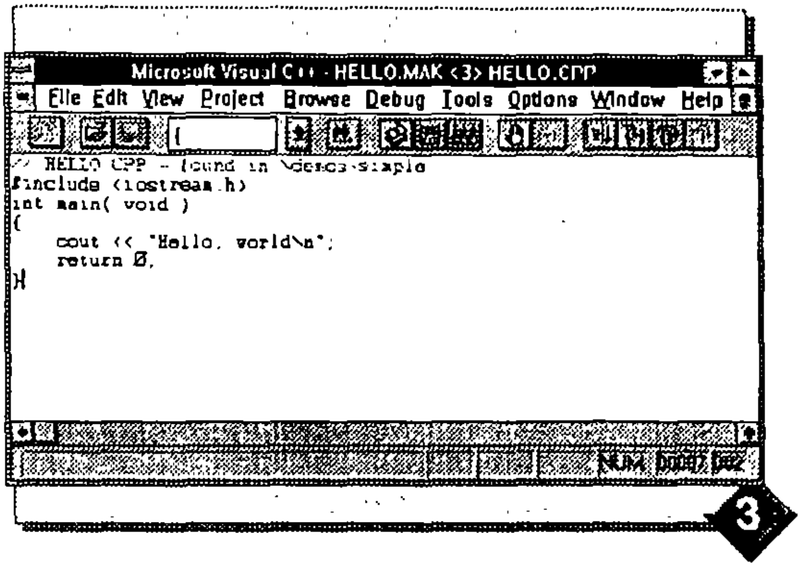
The Visual Workbench main application menu encompasses the entire functionality of the editor, compiler and debugger.

The Visual Workbench toolbar provides shortcuts to commonly used features.

The Visual Workbench status bar provides messages and information, including compiler and linker errors, process status, and so forth.

Fundamentals of Editing Files

Slide Objective
Assure students that VWB has the standard editing features they are accustomed to using.


Microsoft Visual C++ - HELLO.MAK <3> HELLO.CPP
File Edit View Project Browse Debug Tools Options Window Help
NUM 00007 002
3

```
// HELLO.CPP - found in \demos\simple
#include <iostream.h>
int main( void )
{
    cout << "Hello, world\n";
    return 0;
}
```

Use the File Menu in Visual C++ to:

1. Start a New source file.
2. Open (and locate) an existing source file.
3. Save and rename (Save As) an existing source file.
4. Print out a source file.

Use the Edit menu to:

1. Cut, Copy and Paste portions of source code. You can also use the "shortcut keys".
2. Find and replace text.

Delivery Tips
Watch students and assist any inactive students immediately — before they ask for help.

Student Activity

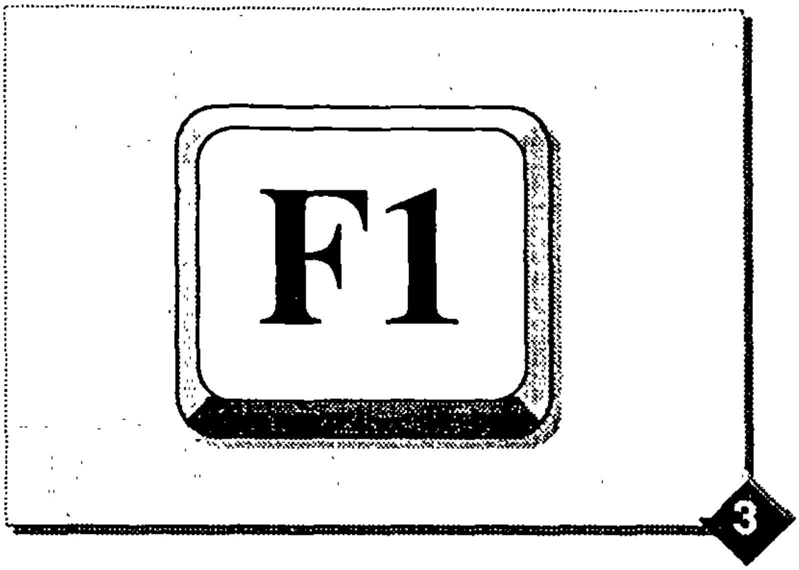
Enter, but *do not* compile, build, or execute HELLO.CPP.

Reference

Refer to "Using the Editor," in the *Visual Workbench User's Guide*

Context-Sensitive Help

Slide Objective
All of the language, library, and tools documentation is cross-referenced and available online via the Help system.



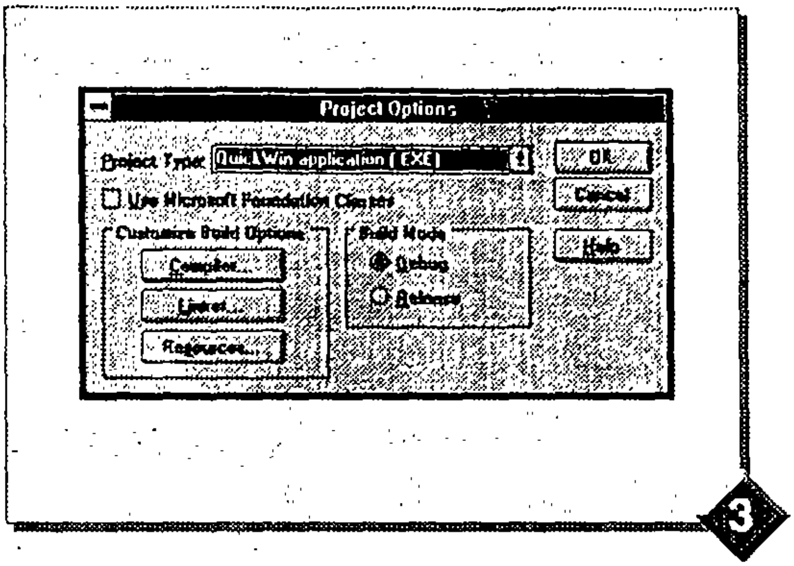
For Your Information
"cout" is an object. Help is available for C/C++ keywords, data types, classes, syntax, and more — not objects.

Context-Sensitive Help

Whenever you have question about a portion of the Visual C++ product, you need only press F1 to get Help on the topic. Not only does the F1 key invoke Help, but it is context-sensitive as well. Suppose you don't remember what **#include** does. You can look it up in the paper-based documentation, or you could place the cursor over the word **#include** and press F1. A second overlapped window would appear on your display with **#include** information from the Visual Workbench Help system. Try it.

Setting Compile Options

Slide Objective
Simple, quick activity to set VWB to create a QuickWin Application (.EXE)



Project Options

Project Type: QuickWin application (.EXE)

☐ Use Microsoft Foundation Classes

Build Mode
● Debug
○ Release

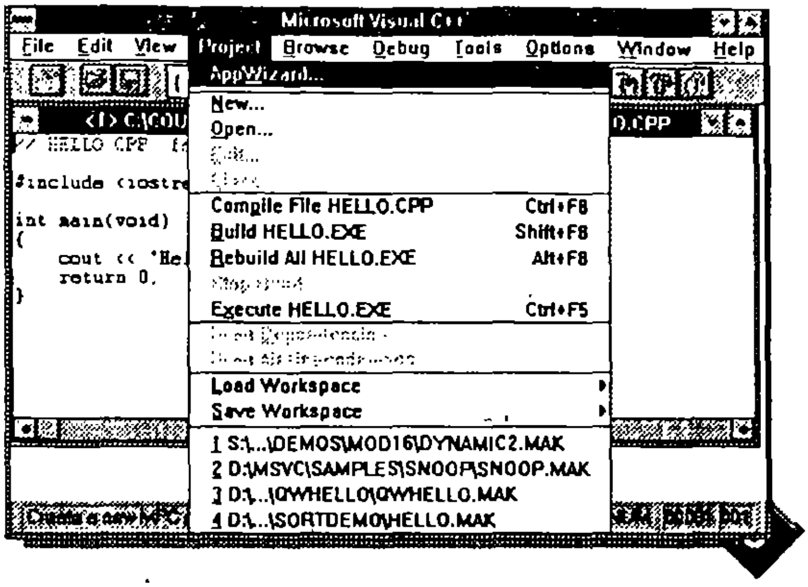
Demo

Set basic compiler options by following these steps:

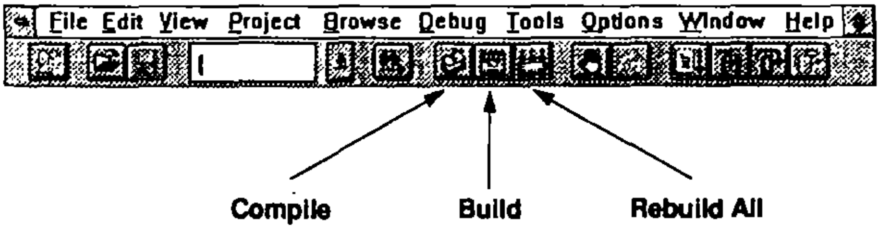
1. From the Visual C++ window, choose the Options menu.
2. Choose Project.
3. The Project Options dialog box appears.
4. In the Project Type list box, select QuickWin Application (.EXE).
5. Move to the Customize Build Options field and choose the Compiler button.
6. This displays the Compiler Options dialog box.
7. In the Category list box, select the Custom Options option and change the Warning Level from 3 to 4. Then select the Listing Files option. Uncheck the Browser Information option by clicking it. Verify that the X is removed.
8. Choose the OK button to dismiss the Compiler Options dialog box.
9. Choose the OK button to dismiss the Project Options dialog box.

Compiling Building, and Rebuilding Programs

Slide Objective
Introductory tour of the Project menu. Only need to cover Build, Rebuild and Execute.


Microsoft Visual C++
File Edit View Project Browse Debug Tools Options Window Help
AppWizard...
New...
Open...
Compile File HELLO.CPP Ctrl+F8
Build HELLO.EXE Shift+F8
Rebuild All HELLO.EXE Alt+F8
Execute HELLO.EXE Ctrl+F5
Load Workspace
Save Workspace
1 S:\...\DEMOS\MOD16\DYNAMIC2.MAK
2 D:\MSVC\SAMPLES\SNOOP\SNOOP.MAK
3 D:\...\QWHELLO\QWHELLO.MAK
4 D:\...\SORTDEMO\HELLO.MAK
// HELLO.CPP
#include <iostr
int main(void)
{
cout << "He
return 0;
}

You can compile, build, and rebuild all source files in your application from either the menus or from three buttons on the toolbar.


File Edit View Project Browse Debug Tools Options Window Help
Compile
Build
Rebuild All

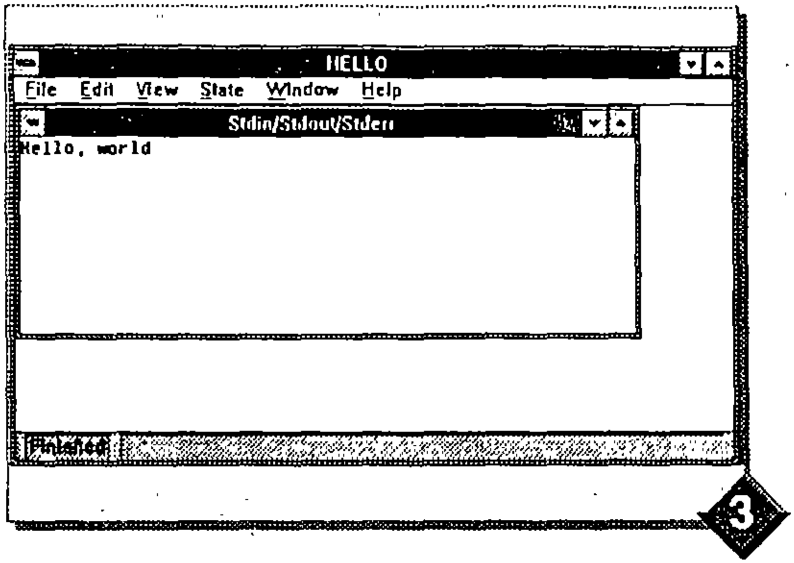
Delivery Tips
Students are familiar with Compile, Link, and Execute. Defer questions on those topics for the second page (following).

- Compiling a source file results only in an .OBJ file.
- Build attempts to generate an .EXE file by compiling and linking. This operation only occurs when changes have been made to the source file.
- Rebuild All forces a compile and link that generates an .EXE file.

These topics will be covered more completely.

What Is a QuickWin Executable?

Slide Objective
Quickly define QuickWin as a character-mode application that receives a typical, Windows application interface. No coding is required to receive the menus, windows, etc.



```
Hello, world
```

Purpose for QuickWin Executables

QuickWin offers a set of translation libraries and compiler options that allow you to produce a Windows program with a minimum of Windows coding.

QuickWin User Interface

Key Points
This course only uses the File menu Exit command or CTRL+C to close.

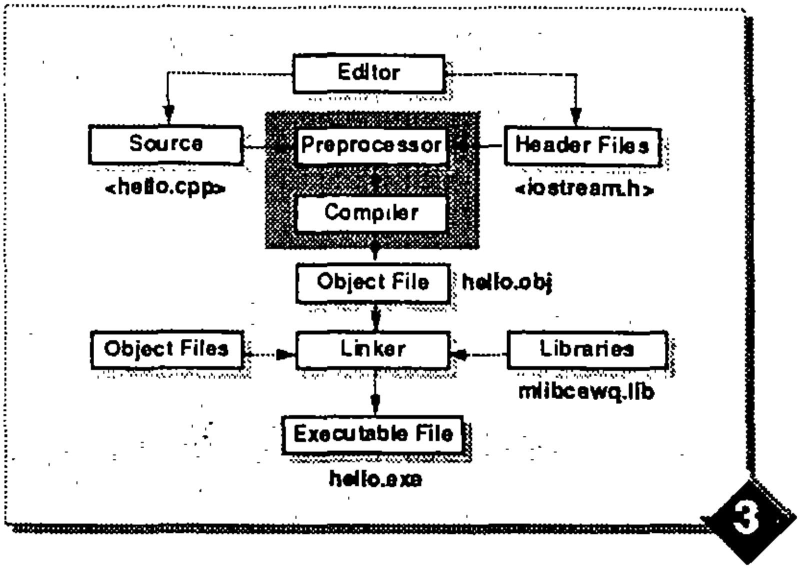
- File: Exit
- Edit: Mark, Paste, Copy Tabs, Copy, Select All
- View: Size to Fit, Full Screen
- State: Pause, Resume
- Window: Cascade, Tile, Arrange Icons, Input, Clear Paste, Status Bar
- Help: Index, Using Help, About

Reference

Refer to "QuickWin Programs," in the *Programming Techniques* manual.

What Does the Build Process Do?

Slide Objective
Familiar to programmers in all languages. Cover quickly. Ask for questions.



Editor

Source

<hello.cpp>

Preprocessor

Compiler

Header Files

<iostream.h>

Object File

hello.obj

Object Files

Linker

Libraries

mlibcewq.lib

Executable File

hello.exe

3

The Process of Building a Program

The first step in the process is creating the C++ source files. When you invoke the compiler, the preprocessor runs; then the compiler runs, creating an object (binary code). Finally, the linker supplies all the statically linked code that your program has asked for.

What Does the Preprocessor Do?

The C/C++ preprocessor makes the first pass through the source code. As it does this, it strips out comments, adds in the .H header files, and makes replacements as defined.

What Does the Compiler Do?

Delivery Tips
Ask for questions. End of big-picture focus. Moving to statement focus.

The compiler takes the preprocessed file and converts the source code into an object module that contains machine-language instructions. In order to be compilable and linkable, a C++ program must have a function called **main**, which serves as the program's entry point. Typically, **main** serves as a "driver" function—the real work is done by the functions that are called by **main**. While **main** isn't technically a reserved word in the C++ language, it should never be used anywhere but as the name of the entry-point function.

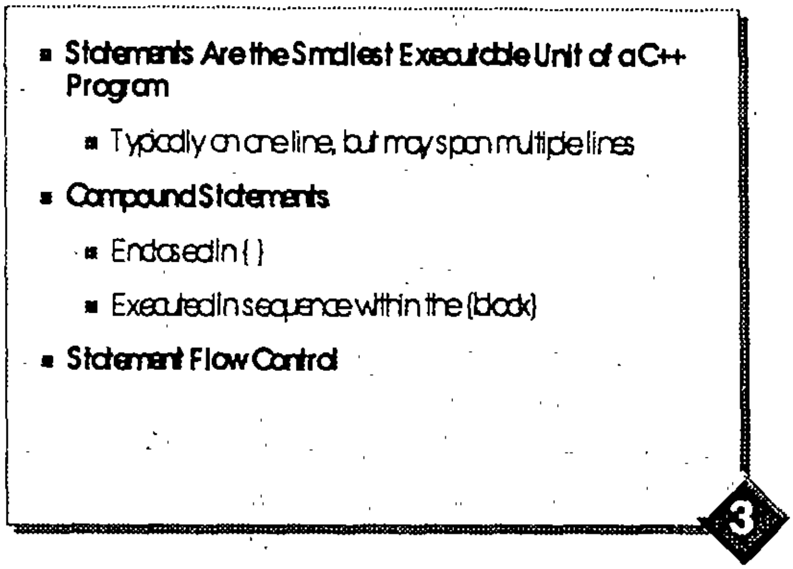
A program's actual code must be placed between a function's braces. If the example above were coded, it would show only one function: `main()`.

What Does the Linker Do?

The linker forms .EXE files by combining object files. The linker can locate these files from compiled modules, existing object files, and from within libraries.

Statements

Slide Objective
Change focus to statement level. Define statements with ; syntax.



- **Statements Are the Smallest Executable Unit of a C++ Program**
  - Typically on one line, but may span multiple lines
- **Compound Statements**
  - Enclosed in { }
  - Executed in sequence within the {block}
- **Statement Flow Control**

Statements are terminated by a semicolon.

A null statement

```
;
```

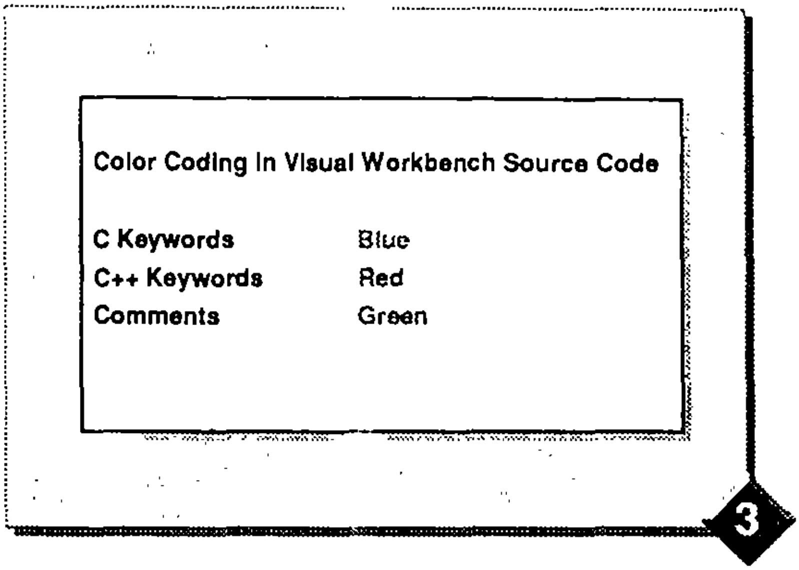
is permissible in C++. The presence of unnecessary statements will not cause compile-time errors. You will return to the study of statements in the next module.

Statements, by default, are executed sequentially within the body of a function. There are flow control statements (such as **if**, **if...else**, and **while**) that cause execution of statements to follow other rules. This subject will be revisited in an upcoming module.

Note Compound statements are similar to a COBOL paragraph.

C++ Keywords

Slide Objective
Looking at a lower level, many statements will use a C++ keyword.



Color Coding In Visual Workbench Source Code

C Keywords	Blue
C++ Keywords	Red
Comments	Green

C++ Keywords

Delivery Tips
HELLO.CPP used two C++ keywords:
int main

The following keywords are reserved for C++:

asm
auto
break
case
catch
char
class
const
continue
default
delete
do
double
else
enum
extern
float
for
friend
goto
if
inline
int
long
new
operator
private
protected
public
register
return
short
signed
sizeof
static
struct
switch
template
this
throw
try
typedef
union
unsigned
virtual
void
volatile
while

The following keywords are reserved for both 16- and 32 bit Microsoft compilers:

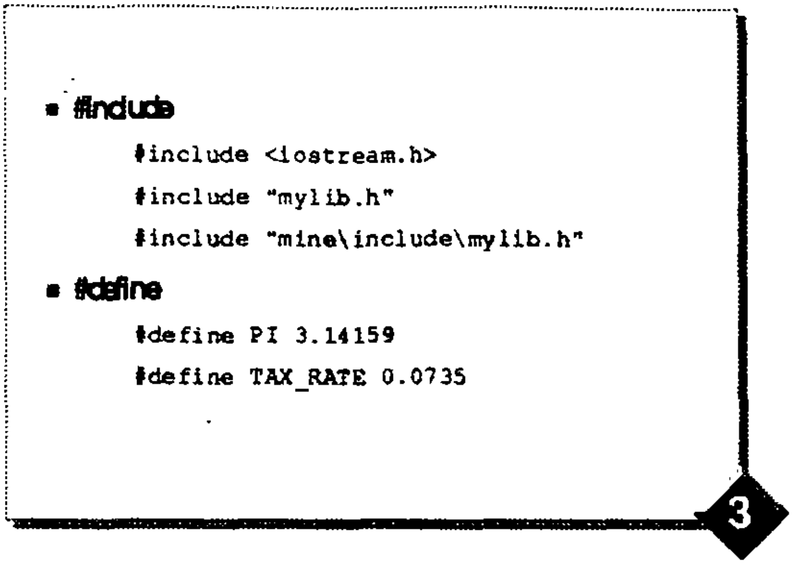
__asm
__based
__cdecl
__export
__fastcall
__loadds
__near
__segname

The following keywords are legal for only 16-bit targets:

__far
__fortran
__huge
__interrupt
__pascal
__saveregs
__segment
__self

Preprocessor Directives

Slide Objective
These are not C++ language statements. They are standard instructions for the C/C++ compilers.



- #include

```
#include <iostream.h>
#include "mylib.h"
#include "mine\include\mylib.h"
```

- #define

```
#define PI 3.14159
#define TAX_RATE 0.0735
```

What Are #includes?

An include directive tells the preprocessor to include the contents of the specified file at that point in the program. Path names must either be enclosed by double quotes or angle brackets.

In the first example above, the <> tell the preprocessor to search for the included file in a special known \INCLUDE directory or directories. From the command line, this directory is specified by the INCLUDE= environment string (usually set in AUTOEXEC.BAT). In the C++ environment, this directory is specified in an Include Files Path text box. (You gain access to that text box from the Options menu. Choose Directories to display the appropriate dialog box.)

In the second example, the double quotes ("") indicate that the current directory should be checked for the header file first. If it is not found, the special directory (or directories) should be checked, as detailed above. The third example is similar, but the named relative directory \MINE\INCLUDE is checked for the header file MYLIB.H.

Relative paths can also be preceded by the .\ or ..\ notation; absolute paths always begin with a \

Header Files (.H)

Header files contain declaration information for functions or constants that are referred to in programs. They are used to keep source-file size to a minimum and to reduce the amount of redundant information that must be coded.

Delivery Tips
Ask for questions. Prepare to move to next topic within module:

What Are Manifest Constants?

The **#define** directive is used to tell the preprocessor to perform a search-and-replace operation. In the first example above, the preprocessor will search through the source file and replace every instance of the token PI with 3.14159.

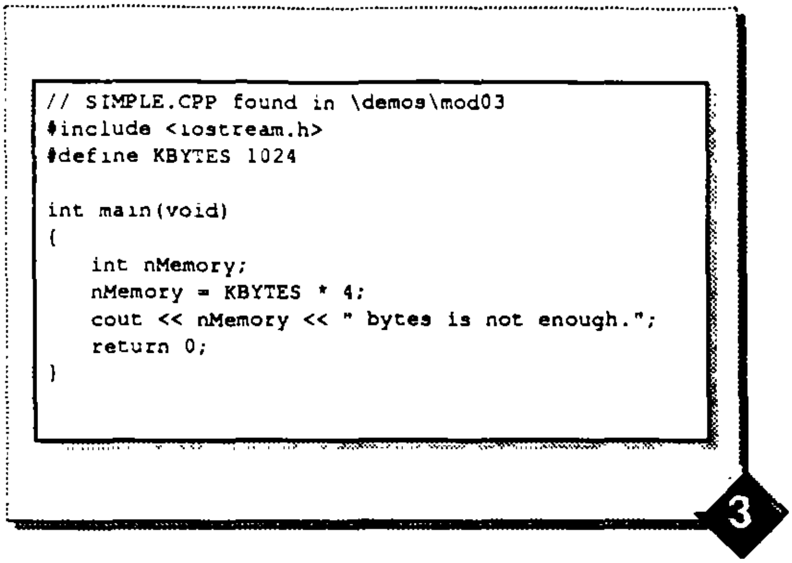
After performing the search-and-replace operation, the preprocessor removes the **#define** line.

There are two purposes for defining and using manifest constants:

- They improve source-code readability.
- They facilitate program maintenance.

SIMPLE.CPP

Slide Objective
Examine code and identify elements listed.



```
// SIMPLE.CPP found in \demos\mod03
#include <iostream.h>
#define KBYTES 1024

int main(void)
{
    int nMemory;
    nMemory = KBYTES * 4;
    cout << nMemory << " bytes is not enough.";
    return 0;
}
```

Delivery Tips
In addition to the definition of main()... students should be able to locate these items.

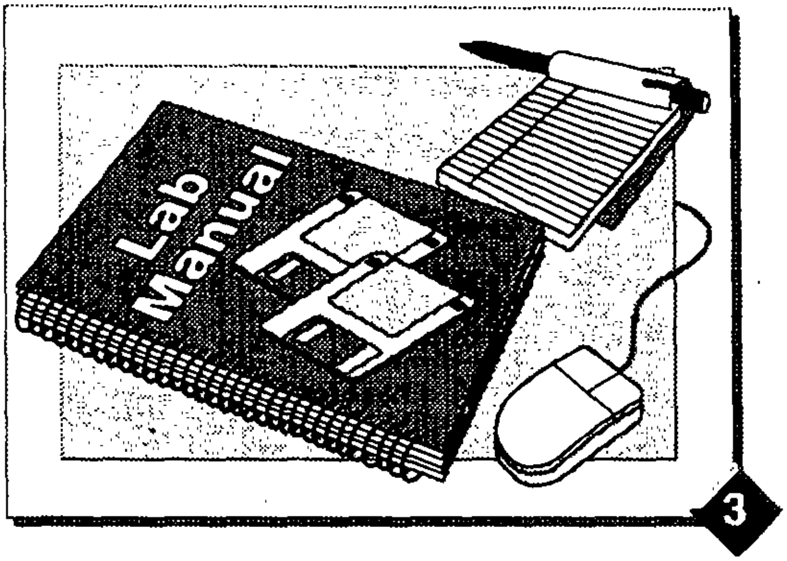
Use VWB to open the file, build, and execute.

The sample application on the slide contains the following elements:

- a comment
- an include
- a manifest constant
- a variable
- four statements

Lab 2: The Basics

Slide Objective
Introduce the lab instructions. Run the executable in the \student directory. Have student read the Scenario and lab introductions.

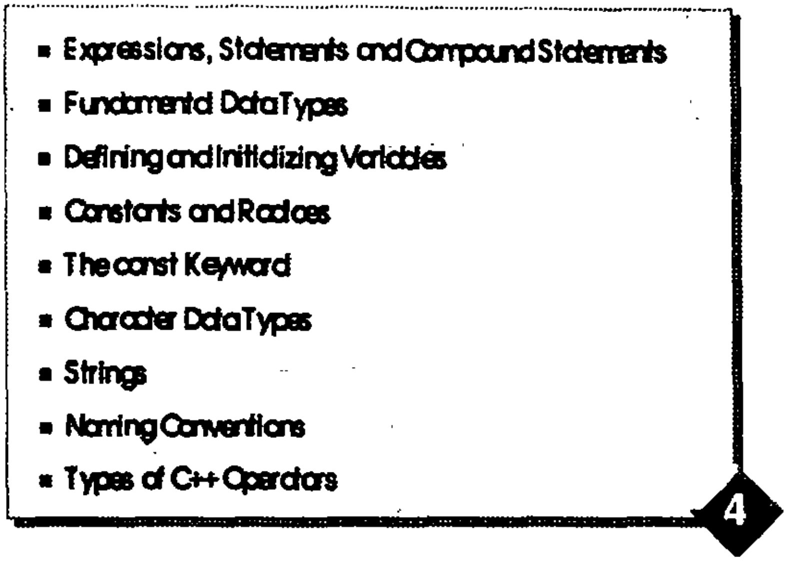


Delivery Tips
Be proactive. Don't wait for questions. Help any student that appears apprehensive.

Module 4: Basic C++ Syntax, Data Types, and Operators

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Expressions, Statements and Compound Statements
- Fundamental Data Types
- Defining and Initializing Variables
- Constants and Radices
- The const Keyword
- Character Data Types
- Strings
- Naming Conventions
- Types of C++ Operators

Module Summary

In Module 3, you created a simple program without much knowledge of its parts. In this module you'll explore the fundamental program unit, expressions.

Though the compact syntax of the C++ language may be a bit different from what you are used to, you will find that the underlying logic of expressions is similar to what you have seen before in other languages. All the data types and operators that the C++ language supports will be listed, but, you will be focusing on only a few that will be important for the programs you'll code in upcoming modules. You may want to mark the data types and operator precedence pages for future reference.

You will need to be able to write expressions in order to implement functions, the subject of the next module.

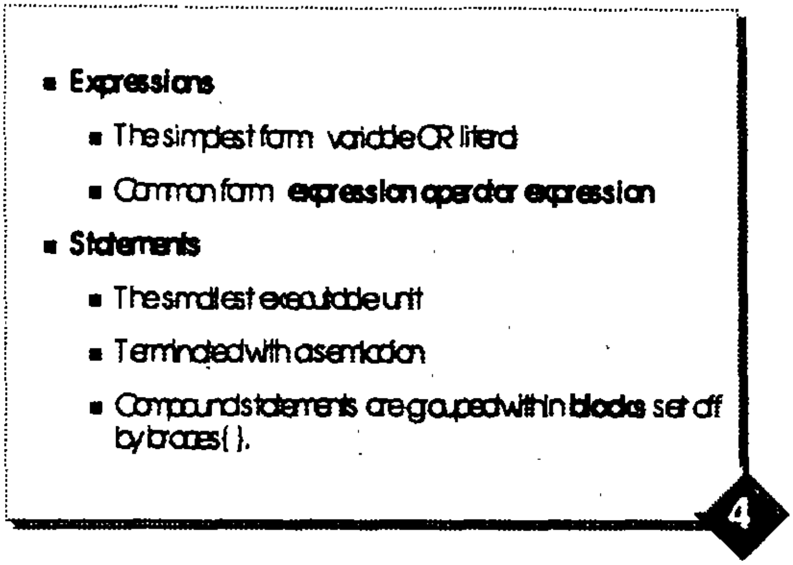
Objectives

Upon completion of this module you will be able to:

- Write simple expressions.
- Create and use variables to hold data.
- Use some operators to manipulate variables within expressions.
- Use literals to initialize variables.

Expressions, Statements, and Compound Statements

Slide Objective
The audience knows expression and statements. Introduce the C++ differences.



- **Expressions**
  - The simplest form variable OR literal
  - Common form **expression operator expression**
- **Statements**
  - The smallest executable unit
  - Terminated with a semicolon
  - Compound statements are grouped within **blocks** set off by braces { }.

Expressions and Statements

To relate these two concepts to the English language, expressions are like clauses and statements are like sentences. Expressions are not executable on their own; statements are. Statements can be made up of expressions. They are terminated by semicolons.

Many expressions are data manipulations.

The simplest expressions are just a variable or literal. They involve no manipulation:

```
nUpperLimit

5
```

All expressions result in a value (including the simple examples cited above).

More commonly, however, expressions are made up of operands and operators. Operands are the data, represented either by variables or literals. (You will examine the predefined C++ data types in the next few foils.) Operators can be unary, binary, or ternary. A unary operator requires only one operand, a binary operator two, and a ternary operator three. You can form complex, nested expressions.

```
(nLowerLimit + 10)*(nUpperLimit - 20)
```

You can find a list of all the C++ operators and the precedence with which they are evaluated in Appendix B.

Statements, as mentioned earlier, are the smallest unit of execution in C++ programs.

Null statements are allowed.

```
; //Null statement
```

"Do-nothing" statements will not generate compile-time errors.

```
5; //do-nothing
```

Statements serve a number of different purposes in C++ programs, for example:

```
nUpperLimit = 200; //assignment
return 0; //return statement
```

You will examine a number of other types of statements in later modules.

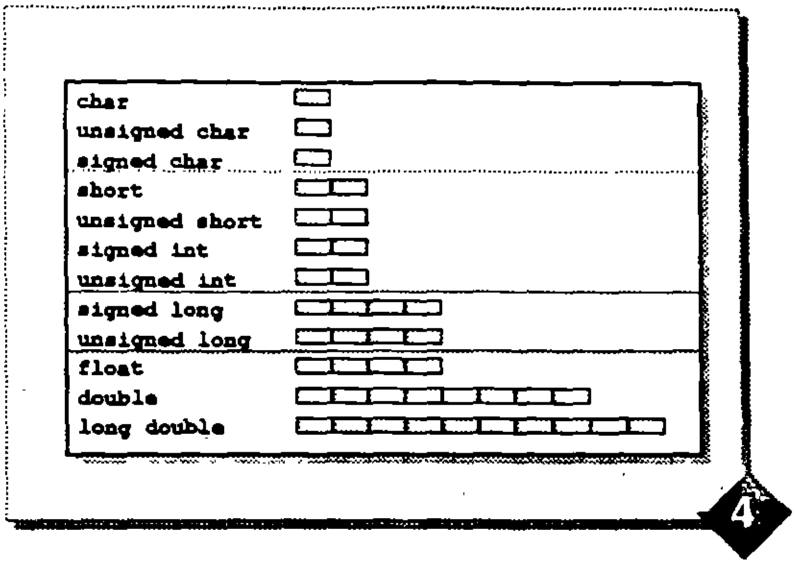
Statements can be grouped into sequences using curly braces. These are called *compound statements* or *blocks*. A compound statement can be used in place of a simple statement.

C++ is a block-structured language, meaning that groups of statements are executed as an indivisible unit. In fact, the body of a function like **main** is nothing more than a block. This important concept forms the cornerstone of the next few modules.

Key Points
Use of braces to denote a statement block is a new concept. Similar to COBOL paragraph (just somewhat like a COBOL function).

Fundamental Data Types

Slide Objective
Explain the (inverted) hierarchy of data types offered by C++. Students only need char, int, and long to get started!



```
char
unsigned char
signed char
short
unsigned short
signed int
unsigned int
signed long
unsigned long
float
double
long double
```

Delivery Tips
Note: 16-bit target. For other machine targets, you can determine ranges by examining the contents of the include files: LIMIT.H and FLOAT.H

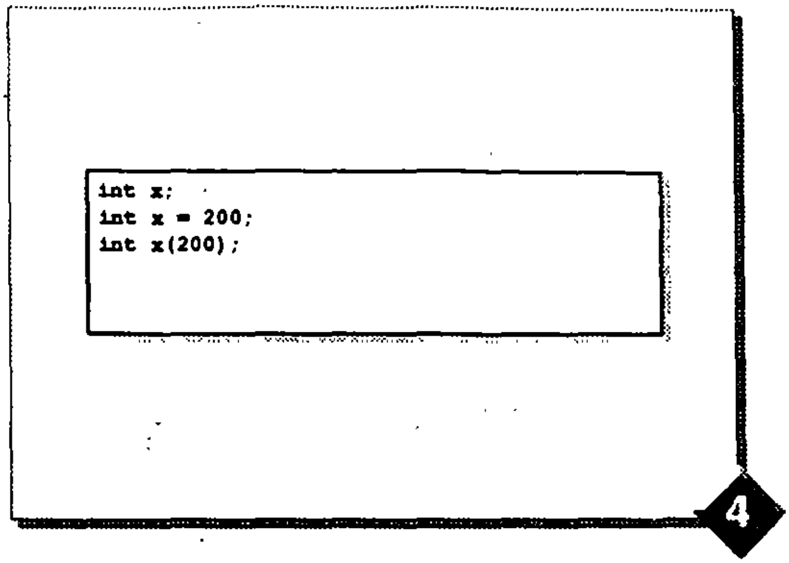
16 bit implementation

Type	Size	Range
char	1 byte	–128 to 127
unsigned char	1 byte	0 to 255
signed char	1 byte	–128 to 127
short	2 bytes	–32,768 to 32,767
unsigned short	2 bytes	0 to 65535
int	2 bytes	–32,768 to 32,767
unsigned int	2 bytes	0 to 65535
long	4 bytes	± 2.1 billion
unsigned long	4 bytes	0 to 4.2 billion
float	4 bytes	± 3.4 x 10^ ± 23
double	8 bytes	± 1.7 x 10^ ± 308
long double	10 bytes	± 3.4 x 10^–4932 to 1.2 x 10^4932

Currently the three char data types are guaranteed to be 1 byte in length, but the other data types are machine-architecture-dependent.

Defining and Initializing Variables

Slide Objective
Declaring a variable orders the compiler to create space at run-time. Declaring and initializing a variable defines a value for that space. C++ supports three styles:



```
int x;
int x = 200;
int x(200);
```

Key Points
Example 1: declares space.
Examples 2 and 3: declare space and set the value.

Example 3 is analogous to using a constructor on an int.

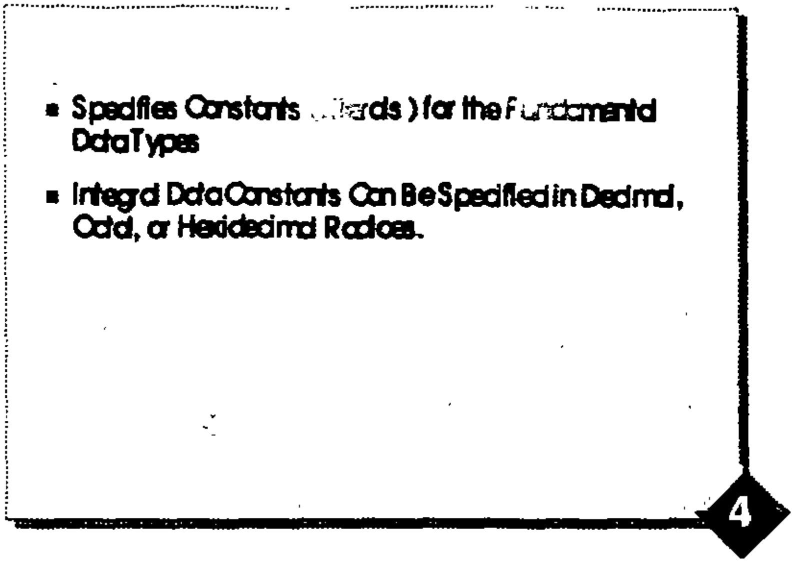
Before a variable can be used in a program, it has to be defined. A definition is a nonexecutable statement that consists of the following parts:

- A data type
- A variable name
- An optional initializer
- The semicolon

As is shown in the foil, the initial value can be coded in two different ways.

Constants and Radices

Slide Objective
The initialization values and other constants use a prefix to denote the radices (base-8, base-10, etc.) and a suffix to denote data type (default is int).



- Specifies Constants (Literals) for the Fundamental Data Types
- Integral Data Constants Can Be Specified in Decimal, Octal, or Hexidecimal Radices.

Integral constants (or literals) may be represented in decimal (base 10), hexidecimal (16), or octal (8) radices.

The 0x or 0X prefix specifies a hexidecimal constant.

17 decimal is 0x11

The zero prefix specifies an octal constant.

17 decimal is 021

By default, an integral numeric constant is of type **signed integer**.

The l or L suffix forces an **int** to a type **long**.

```
0xA49C0L
```

The u or U suffix forces an **int** to type **unsigned**.

```
50000U
```

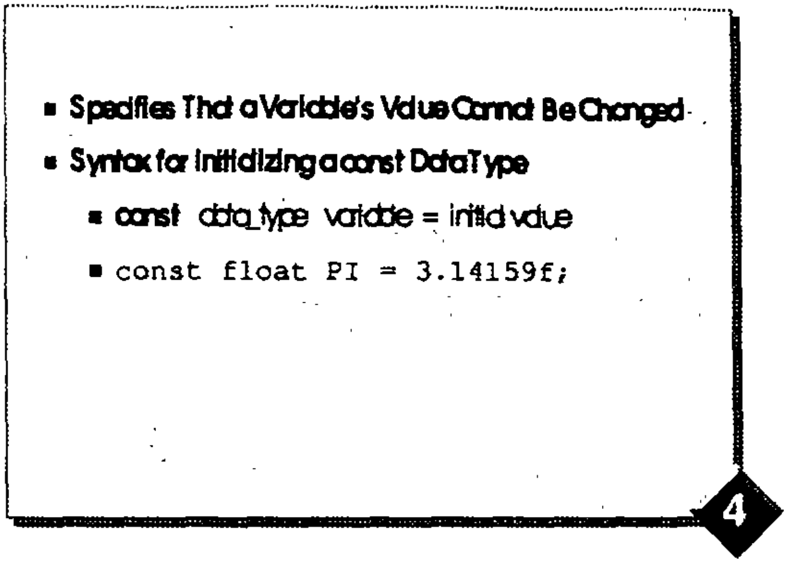
Any constant containing a decimal point or an exponent is a **double floating point** type by default. Floating point numbers may only be represented in base 10.

The f or F suffix forces a value to type **float**.

```
3.2345e3F
```

The const Keyword

Slide Objective
Explain the "const" modifier in the arena of read-only variables (not the same as a manifest constant).



- Specifies That a Variable's Value Cannot Be Changed
- Syntax for Initializing a const Data Type
  - const data_type variable = initial value
  - `const float PI = 3.14159f;`

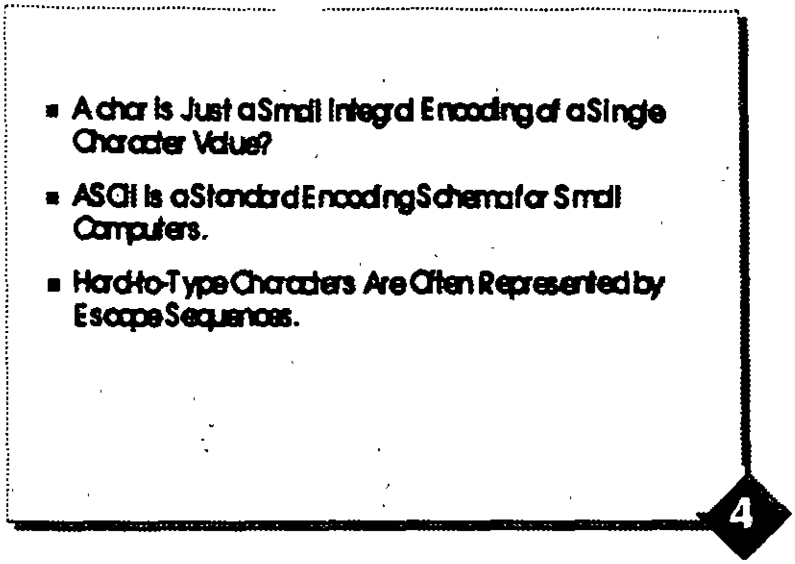
Delivery Tips
const is a type modifier (like unsigned and long).

The **const** keyword provides a way to provide data to your program symbolically without allowing your program to change it. In the example above, you may want to provide the universal value PI to functions making geometric calculations. It is cumbersome to have to use the literal value if there are lots of places that it is needed. Further, if another programmer looks at your code, the symbol PI is immediately identifiable.

Recall from the last module that you can use a #define preprocessor directive to create a manifest constant—or an unchanging value. The difference between a **const** variable and a manifest constant is that the #define causes the preprocessor to do a search-and-replace operation throughout your code. This sprinkles the literal (specified in the #define) throughout your code wherever it is used. On the other hand, a **const** variable allows the compiler to optimize its use. (Compiler optimization is outside the scope of this course.) This makes your code run faster.

Character Data Types

Slide Objective
Typically half the size of an integer, the char data type represents a character (or byte or word) of information.



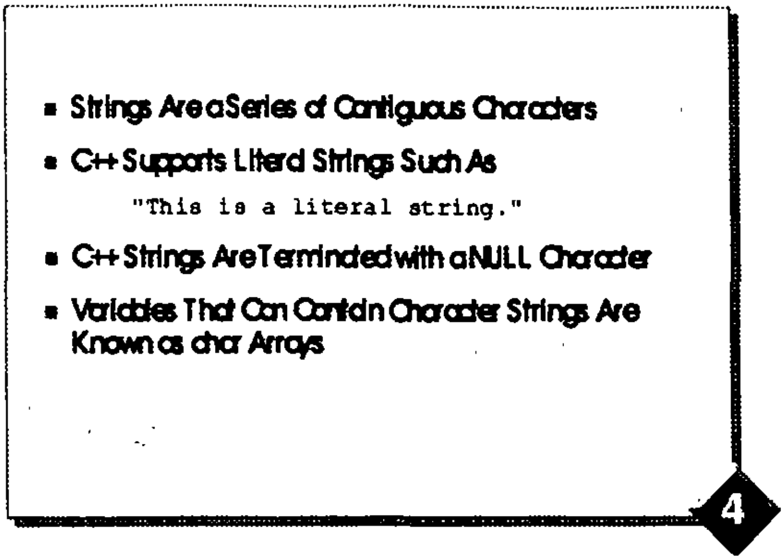
- A char Is Just a Small Integral Encoding of a Single Character Value?
- ASCII Is a Standard Encoding Schema for Small Computers.
- Hard-to-Type Characters Are Often Represented by Escape Sequences.

Check the documentation for the ASCII table.

Escape Sequence	Character	ASCII Value
\n	newline	10
\t	horizontal tab	9
\v	vertical tab	11
\b	backspace	8
\r	carriage return	13
\f	formfeed	12
\a	alert	7
\\	backslash	92
\?	question mark	63
\'	single quote	39
\"	double quote	34
\ooo	octal number	any
\xhh	hexidecimal number	any
\0	null character	0

Strings

Slide Objective
Cover string literals and NULL characters.
C++ does not have a string data type.



- Strings Are a Series of Contiguous Characters
- C++ Supports Literal Strings Such As

```
"This is a literal string."
```

- C++ Strings Are Terminated with a NULL Character
- Variables That Can Contain Character Strings Are Known as char Arrays

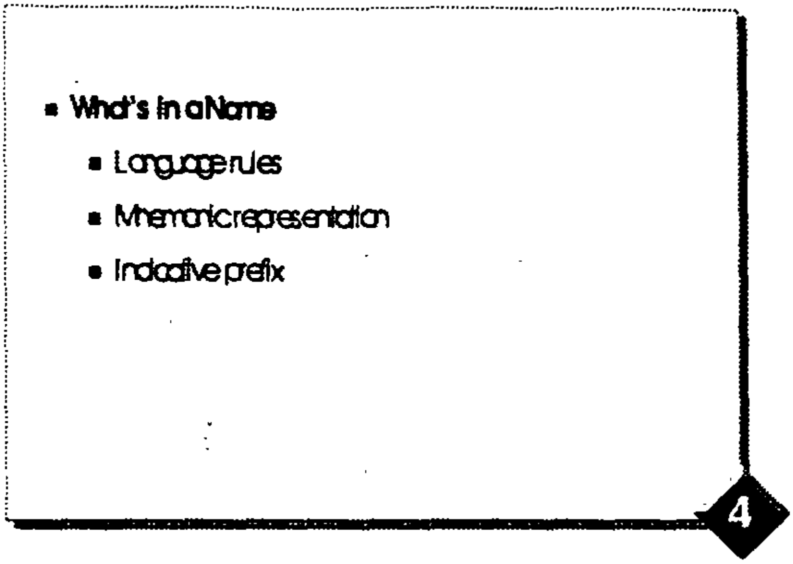
Delivery Tips
Eschew arrays topic.

Arrays of strings are an advanced topic.

The data type of a string literal is a **char** pointer. You will explore arrays and pointers in a later module.

Naming Conventions

Slide Objective
Explain the benefits of encoding variable names with a char or two that denote the data type or major usage of the variable.



- **What's in a Name**
  - Language rules
  - Mnemonic representation
  - Indicative prefix

Naming Conventions

There are a few rules that you should keep in mind when naming variables:

1. You can't use reserved words.
2. The first character must be a letter or an underscore.
3. Other characters can be letters, numbers, or underscores.
4. Only the first 31 characters are significant.

Naming conventions exist for all identifiers in the language: variables, functions, structs, and classes.

For information about Hungarian notation, refer to Appendix A. It is a naming convention that Microsoft supports and encourages.

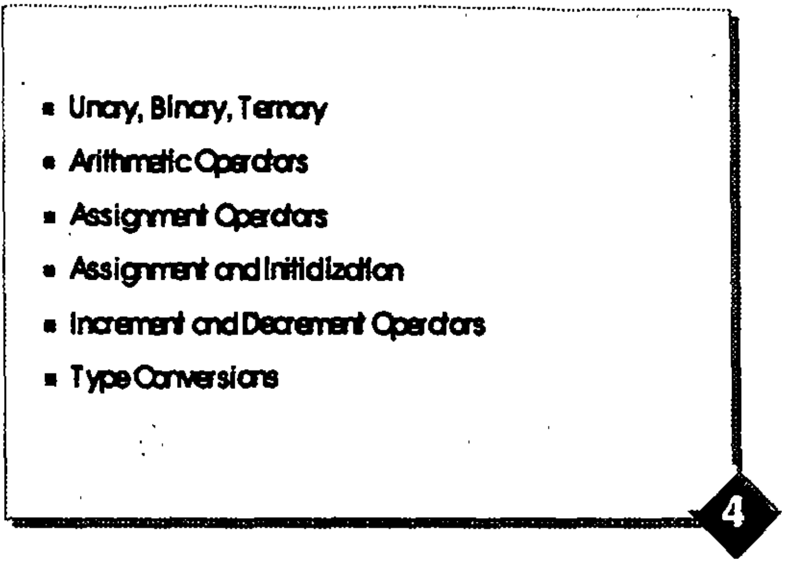
Key Points
Takes very little time to code. Saves hours of maintenance time looking back over pages and pages to look up a variable's definition. Self-documenting variables!

Typical prefixes include:

Prefix	Meaning
f	flag
ch	character
sz	zero-terminated string
i	index
n	number (usually an integer)
l	long
u	unsigned long
p	pointer

Types of C++ Operators: An Overview

Slide Objective
Introduce final topics of the module. Explain unary, binary, and ternary in terms of their operands.



- Unary, Binary, Ternary
- Arithmetic Operators
- Assignment Operators
- Assignment and Initialization
- Increment and Decrement Operators
- Type Conversions

Definitions

Unary operators take one operand.

Binary operators take two operands.

Ternary operators take three operands.

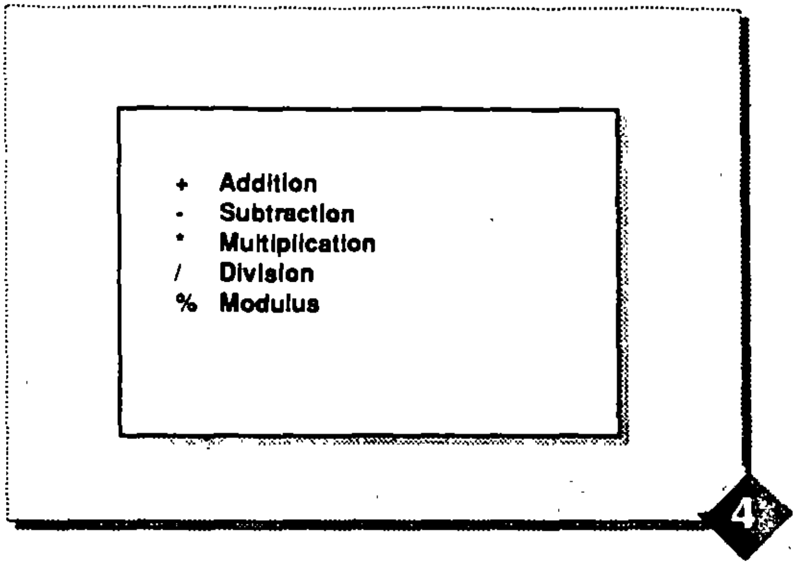
Delivery Tips
Have students locate the Operator Precedence Chart in Appendix A or in the documentation.

Several of the operators in C++ are covered in this module. The relational and logical operators are covered in the next module. Bitwise operators are not covered at all. They are an advanced topic.

Note See Appendix B for the Operator Precedence chart.

Arithmetic Operators

Slide Objective
Quickly explain these binary, arithmetic operators.



+	**Addition**
-	**Subtraction**
*	**Multiplication**
/	**Division**
%	**Modulus**

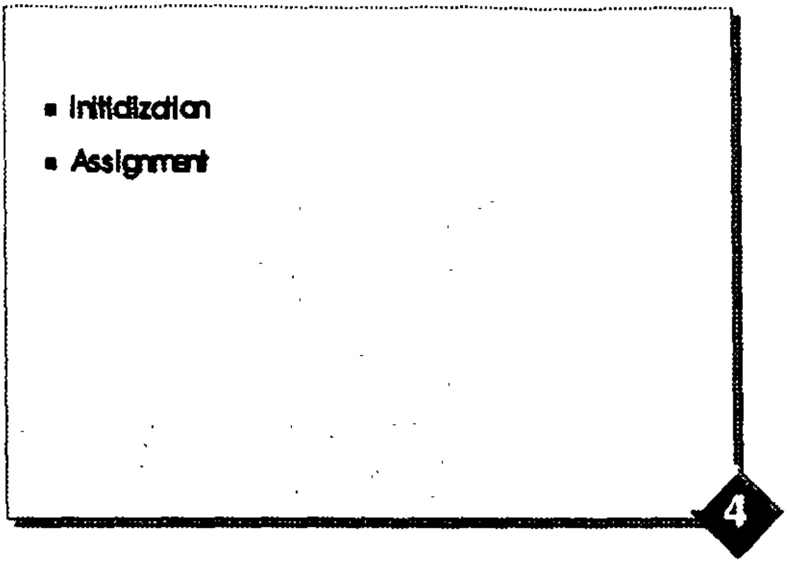
Key Points
C++ arithmetic operators are NOT the same as COBOL. The exception is the FROM verb used in subtraction and division.
Example in COBOL:
SUBTRACT A FROM B GIVING C.
In C++ code:
C = B - A;
C is assigned the value of B less A.

In C++, arithmetic operations are consistent with the way they are performed mathematically: multiplication and division take precedence over addition and subtraction, and so on. Expressions enclosed in parentheses are evaluated first. The rules for associativity and commutivity are maintained.

It is possible to generate numbers that overflow the size of the data types to which they are assigned. Errors of this sort do not generate run-time errors. C++ will not round off values.

The compiler will reconcile mismatched data types automatically through promotion and truncation. These two concepts will be covered in a later foil.

Assignment and Initialization



- Initialization
- Assignment

4

The following code fragment shows you a couple of methods for declaring and initializing variables.

```
#include <iostream.h>

int main(void)
{
int x;
int y = 25;
int z(26);
x = 24;
return 0;
}
```

When a variable is created, it can be given an initial value:

```
int x = 3;
```

This is not considered an executable statement; it is a definition.

Once a variable has been created, it can be assigned a value as an executable instruction in your program:

```
x = 5;
```

Delivery Tips
rvalue and lvalue are defined next page.

The left side of the assignment operator must be a variable or other modifiable entity, known collectively as lvalues.

An rvalue is any expression that resolves to a value.

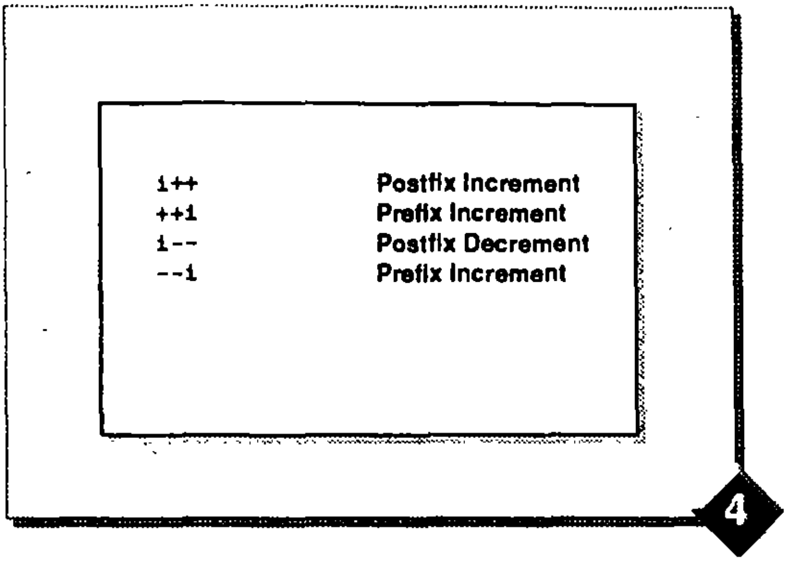
Assignment Operators

- Simple Assignment =
- L-Values and R-Values
- Compound Assignment
  - Multiply and assign *=
  - Divide and assign /=
  - Modulus and assign %=
  - Add and assign +=
  - Subtract and assign -=

An assignment operation writes the value of the right-hand expression or operand to the storage location named by the left-hand operand—an L-value. After the assignment occurs, the assignment expression has the value of the left operand.

A common programming practice is to add a value to a variable, as in x= x + 3. A shortcut notation, compound assignment, allows this statement to be expressed as x += 3. Any operations that use the L-value and R-value properties of a variable written as *<L-value>* = *<R-value>* *<operator>* *<variable>* may be rewritten as *<L-value>* *<operator>* = *<variable>*.

Increment and Decrement Operators



`i++`	**Postfix Increment**
`++i`	**Prefix Increment**
`i--`	**Postfix Decrement**
`--i`	**Prefix Increment**

Prefix and postfix operators increment and decrement their operands according to these rules:

- They obey the rules of unary operators.
- Prefixed increment and decrement operators add or subtract 1 from their operands prior to the operand being used. The R-value of the expression is the result.
- Postfixed increment and decrement operators add or subtract 1 from their operands only after the value of the operand has been used within the expression.

For example, given

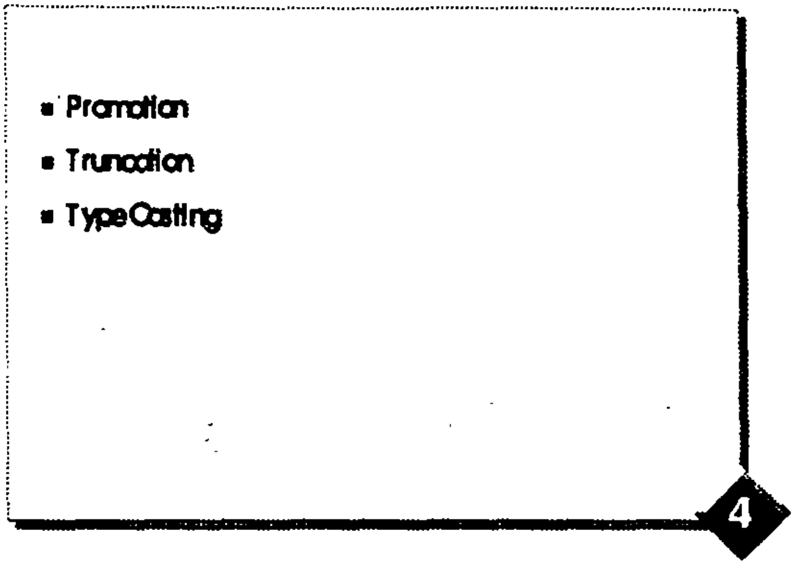
```
int y, x = 10;     // y is undefined and x is 10
y = ++x;   // with prefix increment
y = x++;   // with postfix increment
```

y is 11 and *x* is 12.

Key Point
y is assigned 11 before the postfix makes x a 12.

Type Conversions

Slide Objective
Explain automatic translations of standard data types through truncation and promotion. Contrast to user- controlled use of type-casting.



- Promotion
- Truncation
- TypeCasting

In C++, most binary operators require that operands be of the same data type. If they are not, the compiler implicitly changes the data type of one operand to match the other.

Normally the compiler seeks to promote the smaller data type operand to the same data type as the larger operand. For example:

```
3.14 + 7 / 'p'
```

Key Points
Use Operator Precedence Chart to explain why (int / char) occurs first.

This is seen by the compiler as:

double + (int / char)

It resolves the expression within the parentheses by promoting the **char** to an **int** (an int / an int = an int):

double + (int)

To resolve the **double** + an **int**, the compiler must promote the **int** to a **double**.

Occasionally the compiler will need to specify truncation. During assignment, the rvalue must be the same data type as the lvalue (variable). If there is a mismatch, the rvalue will be truncated:

```
int x;

x = 3.14;
```

If you were to display *x*, you would find it has the value 3!

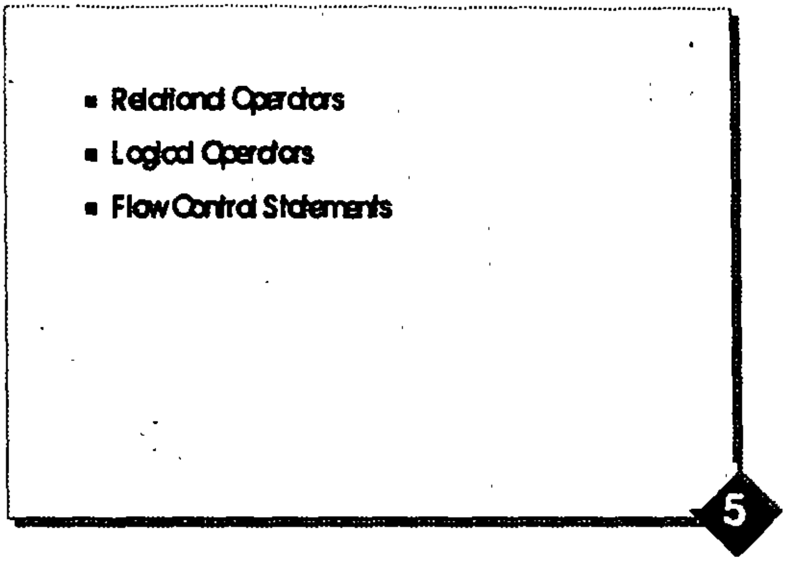
Truncation and promotion occur without generating run-time error messages.

Type casting variables to another type is the most effective way to control the effects of promotion and truncation.

Module 5: Relational and Logical Operators and Flow Control

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Relational Operators
- Logical Operators
- Flow Control Statements

5

Module Summary

By default, C++ statements within a function are executed in a sequential manner. There are a number of ways to alter this flow. As we have seen, a **return** statement executed by **main** will pass control back to the operating system. In this module, you will learn how to code conditional and looping statements.

Objectives

At the end of this module, you will be able to:

Delivery Tips
Cover the objectives to set expectations for the module. COBOL programmers already know 90% of this, go very fast!

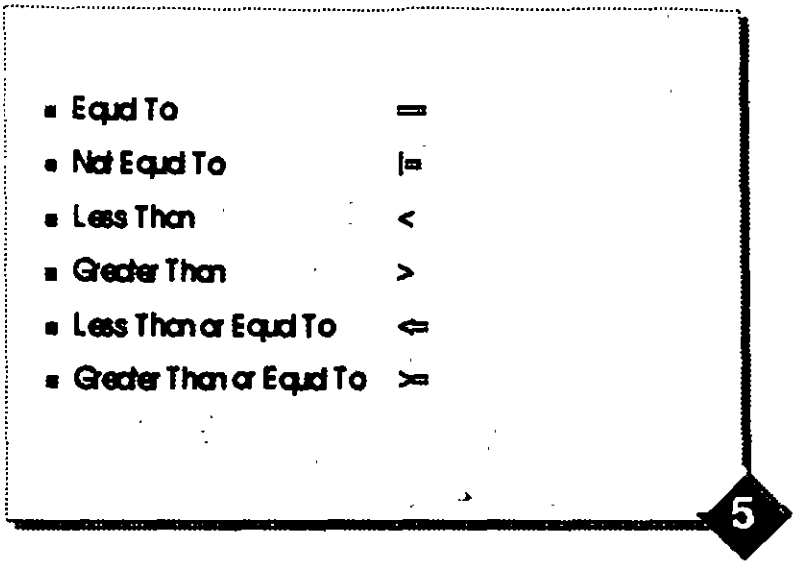
- Use logical and comparison operators.
- . Use relational and equality operators.
- Use **if...else** statements.
- Use **while** and **do...while** loops.
- Use **for** loops.
- Use **switch**, **continue**, and **break** statements.

Lab

Using Statements and Expressions

Relational Operators

Slide Objective
COBOL supports all these relation condition operators.
Spoof: C++ does not support the word equivalents of GREATER THAN, etc.



▪ Equal To	==
▪ Not Equal To	!=
▪ Less Than	<
▪ Greater Than	>
▪ Less Than or Equal To	<=
▪ Greater Than or Equal To	>=

Features of Relational Operators

Associativity is from left to right. The left and right operands are evaluated, and then the operator is applied to give a result.

If the expression is determined to be *false*, the resolved value of the expression is 0 (zero) of data type int. A *true* expression resolves to some non-zero value, typically 1. As you will see in a moment, relational expressions are often used as conditional or looping test expressions.

Delivery Tips
Quickly explain that given x, a two-way test is not logical.
The next page explains the use of logical operators to join relation conditions.

How would the compiler evaluate the following:

```
int x = 20;
10 < x < 5
```

It is evaluated from left to right, testing the first logical pair (10<x) to determine an outcome. In this case, the compiler returns TRUE (most compilers value TRUE as a 1 or some other non-zero number). Next, it evaluates that result against the next operand (TRUE < 5). Illogically, given x=20, the two-way test 10<20<5 would be TRUE. The next page shows how to implement this test correctly.

Warning

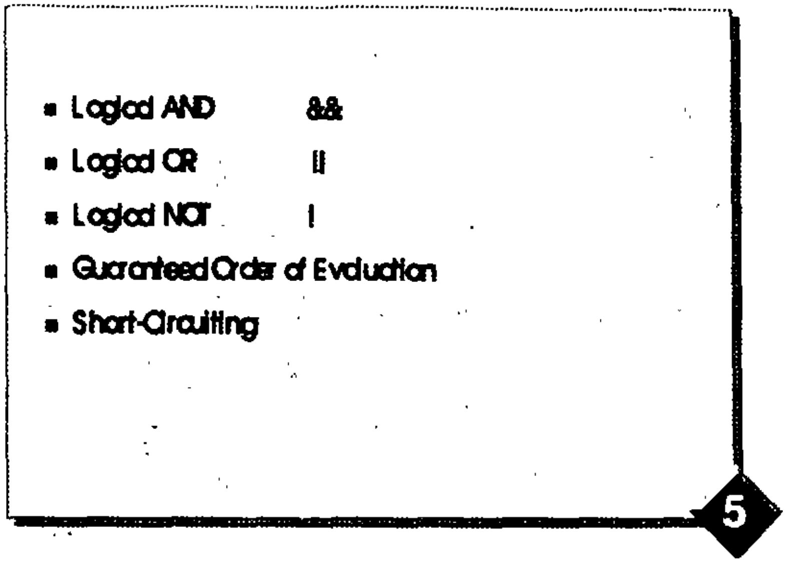
Typographical errors happen frequently when these operators are used:

Delivery Tips
A third warning concerns inserting a space between the characters. "= =" is a syntax error.

- *Equal to* is represented by the operator == (two equal signs). Equal to is easily confused with assignment = (a single equal sign).
- Inequality is represented by the operator != (an exclamation point followed by an equal sign). It is easily transposed to =!, which is an invalid character sequence.

Logical Operators

Slide Objective
Use these logical operators to join logical expressions in a meaningful way.



- Logical AND &&
- Logical OR ||
- Logical NOT !
- Guaranteed Order of Evaluation
- Short-Circuiting

Features of Logical AND, OR, and NOT

The first two operators are used to combine multiple relational expressions to form a compound test.

```
x > 10  &&  x < 5
```

The logical NOT is a unary operator that returns the inverse logical value of its operand—from *true* to *false* or from *false* to *true*.

Compound logical expressions using && and || are guaranteed to be evaluated from left to right. Furthermore, the compiler will construct your code so that at the time when the value of the entire compound expression is known, the appropriate action is taken and part of the expression may not be evaluated. This is known as short-circuiting. For example,

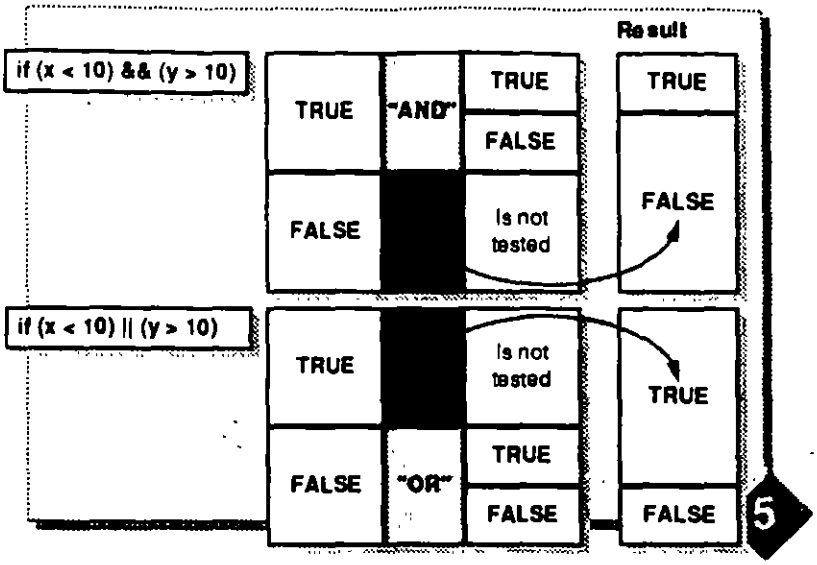
```
int x = 0;
if (x != 0 && x < 100)
do something;
```

Since the first expression evaluates to *false*, the rest of the expression is not evaluated since *false* AND any other value always resolves to *false*. The dependent expression is skipped.

Conversely, in a compound that uses the OR operator, when the first expression evaluates as *true*, the result of the entire compound expression must be *true*. For that reason, the trailing expressions are not evaluated, but the dependent expression is evaluated.

AND and OR Operators

Slide Objective
Walk through the various true-table paths showing the difference between logical AND and logical OR.



Result

if (x < 10) && (y > 10)

TRUE "AND" TRUE → TRUE

TRUE "AND" FALSE → FALSE

FALSE — Is not tested → FALSE

if (x < 10) || (y > 10)

TRUE — Is not tested → TRUE

FALSE "OR" TRUE → TRUE

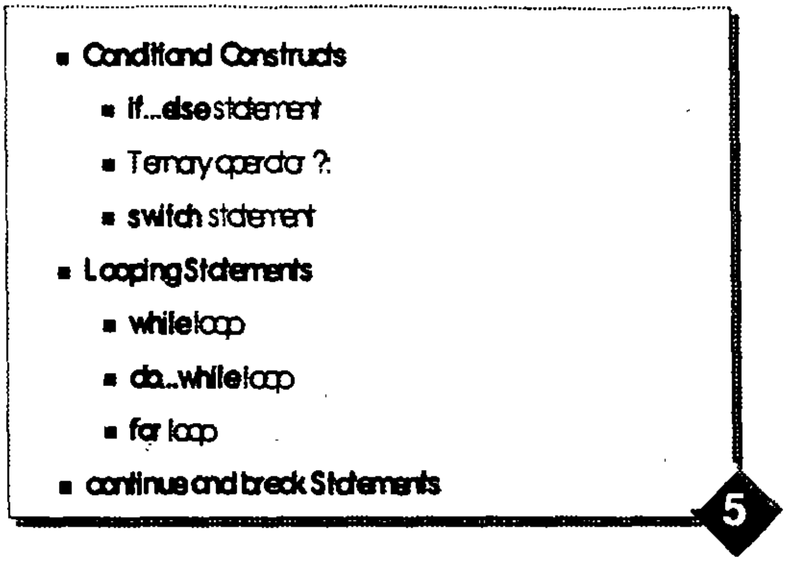
FALSE "OR" FALSE → FALSE

Key Points
Logical AND: Requires both sides of the && to be True to return an overall True. If the left side of AND evaluates to **False**, the right side is not evaluated.
Logical OR: Only requires either side of the | | to be True to return an overall True. If the left side of OR evaluates to **True**, the right side is not evaluated.

Delivery Tips
End of operators. Moving to Flow Control subsection.

Flow Control: Overview

Slide Objective
Name the various constructs and prepare to move quickly through the rest of the module.



- **Conditional Constructs**
  - **if...else** statement
  - Ternary operator ?:
  - **switch** statement
- **Looping Statements**
  - **while** loop
  - **do...while** loop
  - **for** loop
- **continue** and **break** Statements

Now that you are familiar with writing simple and compound conditional test expressions, you are ready to examine the conditional and looping constructions available in C++. Many of these should already be familiar from your past experience with other modern languages.

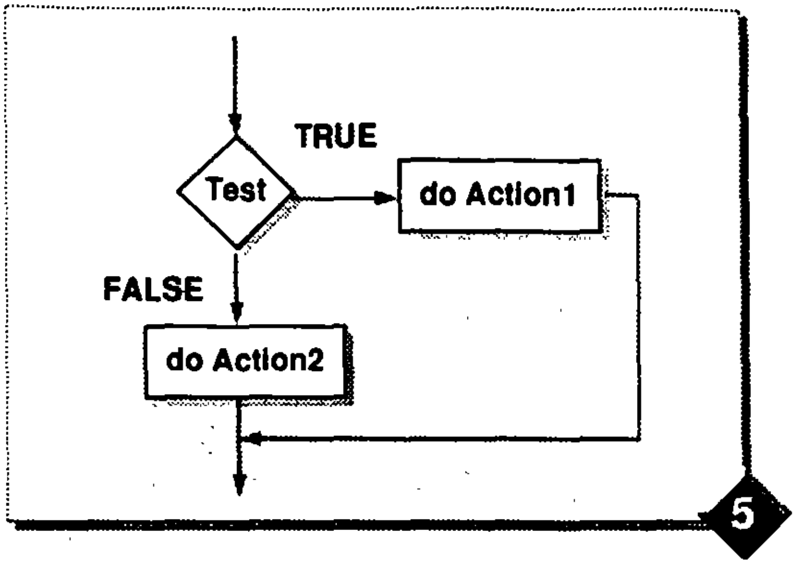
Delivery Tips
Prepare students to look at demo code (online or in their books). Get them set to move quickly. COBOL programmers already know these constructs.

In the following discussions, wherever a statement is required in the syntax, it can be either a null statement, a simple statement, or a block of code (a compound statement).

Conditional and looping statements can be nested to an arbitrary depth in C++.

C++ also has a **goto** statement. Because its use encourages nonstructured coding—also known as spaghetti coding—it will not be covered in this course.

if...else Statements



Syntax

```
if (expression)
    statement; // Action1
else
    statement; // Action2
Given integer variables x, y, and max:
if (x >= y)
    max = x;
else
    max = y;
cout << "maximum value is " << max;
```

The entire else portion of the statement is optional.

For Your Information
COBOL differences:
- Expression in ( )'s
- No THEN clause
- "else statement;" is optional.

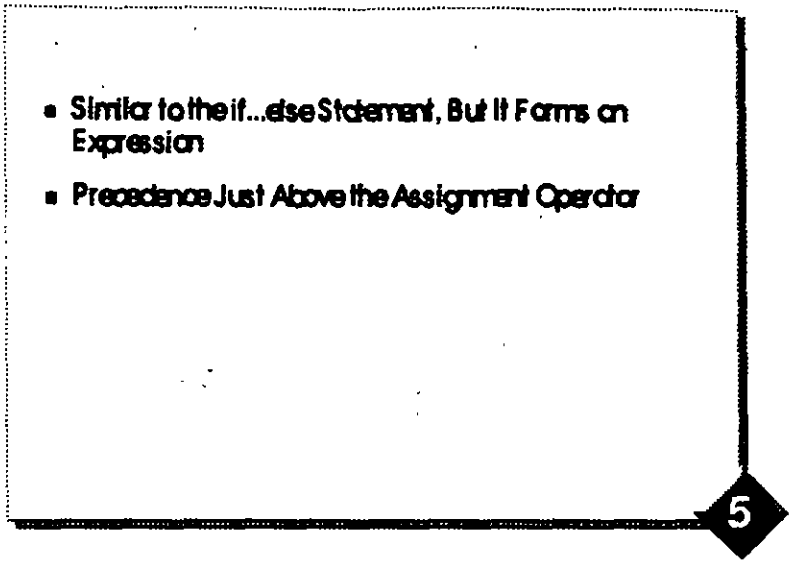
Demc

ELSE.CP' , found in \DEMOS\MOD05. This demonstration shows use of the if...else construct.

```
// ELSE.CPP      found in \demos\mod05
/'/ Demonstrate if and if-else conditional flow.
// The expression should be encased by parentheses.
                                // preprocessor directive
#include <iostream.h>
                                // manifest constants
#cefine B_KEY 'b'
#define CAPITAL_B 'B'

int main(void)
{
    char ch;
    cout <<."Enter the 'b' key for a beep: ";
    cin >> ch;
    if (ch == B_KEY) // test equivalence char vs char
        cout << "Beep!"; //    true
    else                //    false
        if (ch == CAPITAL_B)    // another test
            cout << "BEEP!!";   // true
        else
            cout << "Bye bye";  // false again
    return 0;             // Regardless of the input,
}                         // return success (0 errors)
```

Ternary Operator ?:

Slide Objective
Quickly explain conditional operator within if-else terminology.



- Similar to the if...else Statement, But It Forms an Expression
- Precedence Just Above the Assignment Operator

The ternary or conditional operator closely mimics the function of the **if...else** statement in C++. Its main advantage is that it forms an expression, and expressions can be used in many places where statements are not allowed.

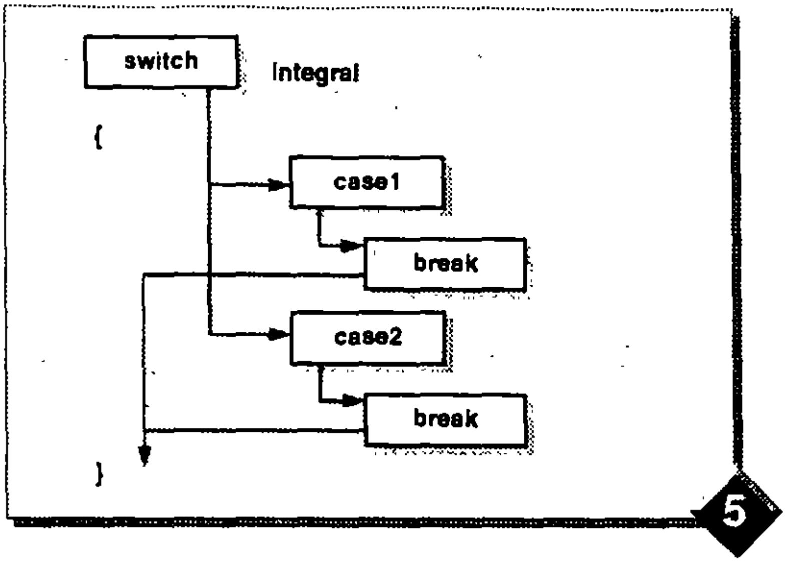
Key Points
Ternary operator of:
(exp) ? s1 : s2;
is analogous to:
(exp) THEN s1
ELSE s2;

```
cout <<  "maximum value is " <<  (x >= y ? x : y);
```

Tip Avoid the temptation of over-using the ternary operator. Use it only where C++ syntax forces or suggests the use of an expression.

switch Statements

Slide Objective
Explain SWITCH statement processing.



Syntax

Delivery Tips
This is a new construct, slow down for this page.

```
switch (integral expression)
{
   case IVAL1:
   statement; // case 1
   break;
   case IVAL2:
   statement; // case 2
   break;
   . . .
   default:
   statement;
   break;
}
```

C++ **switch** statements, also called **case** statements, have the following limitations and considerations:

- Only integral expressions may be tested.
- Each **case** statement may only test against a compile-time integral constant.
- Without the **break** at the end of each **case** portion, fall-through execution will occur.

The **switch** statement should be used in preference to a nested **if...else** whenever these conditions can be met.

Tip Case logic is more efficient than nested **if...else**. This construct works well for setting up a decision framework.

Demos

POWER1.CPP is located in \DEMOS\MOD05. It demonstrates use of **switch** statements with breaks.

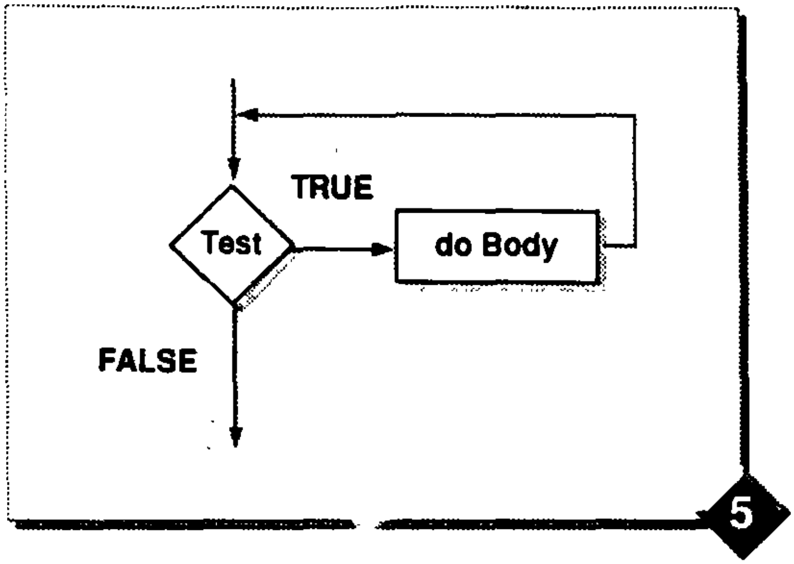
```
// POWER1.CPP    found in \demos\mod05
// A typical use for the switch statement.
#include <iostream.h>

int main()               // definition for main
{
    long lNumber, lResult;
    int iPower;

    cout << "Enter a number: ";
    cin >> lNumber;
    cout << "What power do you want it raised "
         << "to? (1-5) ";
    cin >> iPower;       // based on the user's input,
    switch (iPower)      // perform a case section
    {
        case 5:          // only if user entered '5'
            lResult = lNumber * lNumber * lNumber *
                      lNumber * lNumber;
            break;
        case 4:          // statement(s) for '4'
            lResult = lNumber * lNumber * lNumber *
                      lNumber;
            break;       // break jumps flow out of switch
        case 3:
            lResult = lNumber * lNumber * lNumber;
            break;
        case 2:          // notice ":" for each case
            lResult = lNumber * lNumber;
            break;
        case 1:          // Any number raised to first
            lResult = lNumber;  // power is itself.
            break;
        default:     // "default" catches all other cases
            cout << "Only powers of 1 to 5 are "
                 << "valid.\n"; // Show error to user.
            return 1;    // Premature return from program!
    }
    cout << lNumber << "raised to the power"
         << iPower << "is" << lResult << ".\n";
    return 0;            // normal return from program
}
```

POWER2.CPP is located in \DEMOS\MOD05. It demonstrates use of switch statements with fall-through execution.

```
// POWER2.CPP    found in \demos\mod05
// A non-standard use for the switch statement
// allows cases to fall through to the next case
#include <iostream.h>

int main()
{
    long lNumber, lResult;
    int iPower;

    cout << "Enter a number:";
    cin >> lNumber;
    cout << "What power do you want it raised"
         << "to? (1-5) ";
    cin >> iPower;
                       // optimistically, set lResult
    lResult = lNumber; // to "first power"

    switch (iPower)    // depending on user's input...
    {                  // enter at the appropriate
        case 5:        // case location in the switch...
            lResult *= lNumber;
        case 4:        // and fall from one case...
            lResult *= lNumber;
        case 3:        // into the next...
            lResult *= lNumber;
        case 2:        // again...
            lResult *= lNumber;
        case 1:        // finally, ...
            break;     // a break! 1-5 all break here.
        default:
            cout << "Only powers of 1 to 5 are"
                 << "valid.\n";
            return 1;  // Error return (still no break)
    }                  // but the program is done.

    cout << lNumber << "raised to the power"
         << iPower << "is" << lResult << ".\n";
    return 0;
}
```

while Loops

Slide Objective
A C++ while statement is COBOL's "PERFORM n1 WITH TEST BEFORE..."



Syntax

```
while (expression)
    statement; // loop body
```

Note that there is no semicolon at the end of the test expression line. The possible number of iterations of a while loop is between zero and infinity.

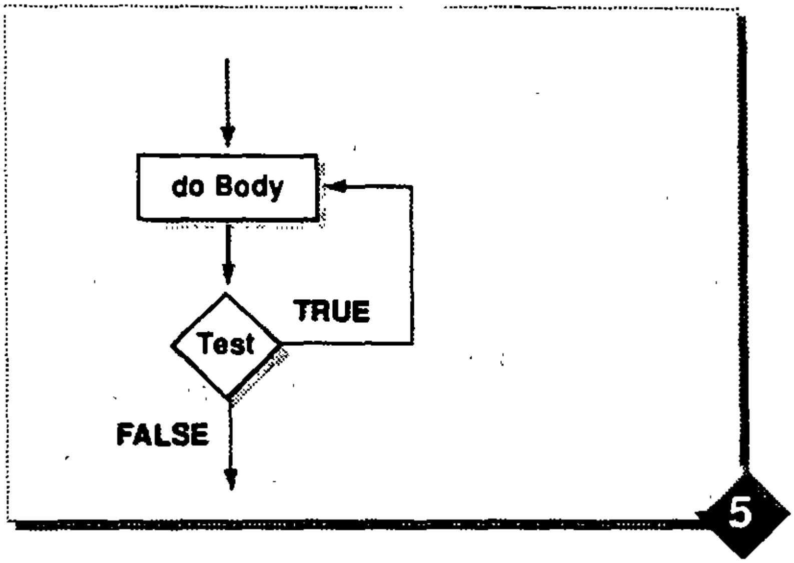
Demo

WHILE.CPP is located in \DEMOS\MOD05. It shows the use of the **while** loop construct.

```
// WHILE.CPP   found in \demos\mod05
// A while loop is processed zero or more times because
// the test happens first - before the body of the loop.
#include <iostream.h>

#define B_KEY 'b'

void main()
{                  // Local variables (undefined contents)
    char ch = ' '; // must be initialized or preset with
                   // a value before entering the "while."
    cout << "Enter a 'b' for a beep: ";
    while (ch != B_KEY) // while loop (conditional)
    {                   // Body of the loop
        cin >> ch;            // get input
        if (ch == B_KEY)      // another test (expression)
            cout << "Beep!";  // true
        else                  // false
            cout << "Please, enter the 'b' key.";
    }   // End of loop.  Loop continues while the expression
        // is True (non-zero), but stop at False...
}       // Notice the test used in the while is a != test.
```

do...while Loop

Slide Objective
A C++ while statement is COBOL's "PERFORM n1 WITH TEST AFTER..." The body exec's one or more times.



Syntax

```
do
statement; // loop body
while (expression);
```

Note that there is a semicolon at the end of the test expression line. The possible number of iterations of a **do...while** loop is between one and infinity.

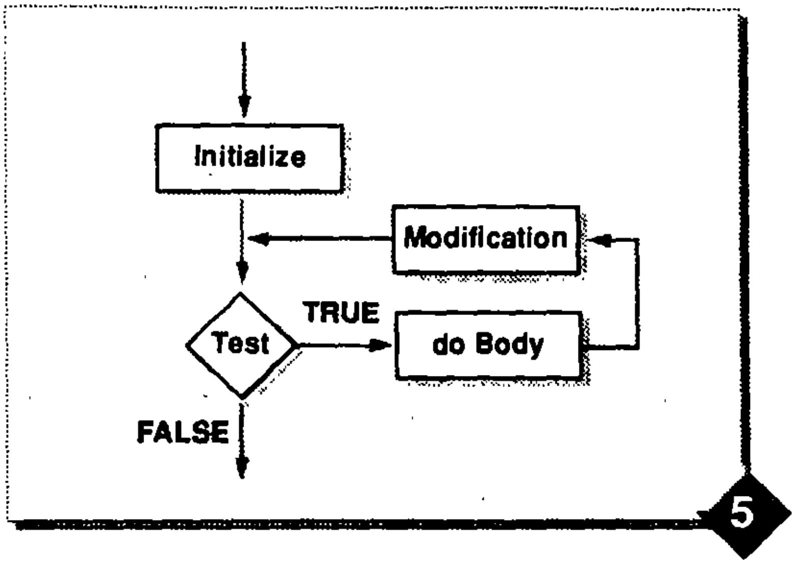
Demo

DOWHILE.CPP is located in \DEMOS\MOD05. It shows the use of the do...while loop construct.

```
// DOWHILE.CPP   found in \demos\mod05
// The body of a do-while is processed one or
// more times.
#include <iostream.h>
                         // manifest constant
#define B_KEY 'b'

int main(void)           // definition for main func
{
    char ch;             // ch has undefined contents

    cout << "Enter tne 'b' key for a beep:";
    do {
       cin >> ch;        // ch has user's character
       if (ch == B_KEY)
           cout << "Beep!";
       else
           cout << "Please, enter the 'b' key.";
    } while (ch != B_KEY); // loop reiterates while
                           // user's ch != 'b'
    return 0;              // (Note: single quotes)
}
```

for Loop

Slide Objective
A C++ "for" statement is COBOL's PERFORM using all the options.



Syntax

Key Points
"initialization" is VARYING v1 FROM

"test" is WITH TEST BEFORE

"statement" is PROCEDURE-NAME

"modification" is VARYING v1 UNTIL

```
for (initialization; test; modification)
statement;
```

Note that exactly two semicolons are needed inside the **for**'s parentheses. The possible number of iterations of a **for** loop is between zero and infinity.

A **for** loop is equivalent to the following **while** loop:

```
initialization;
while (expression)
{
   statement;
   modification;
}
```

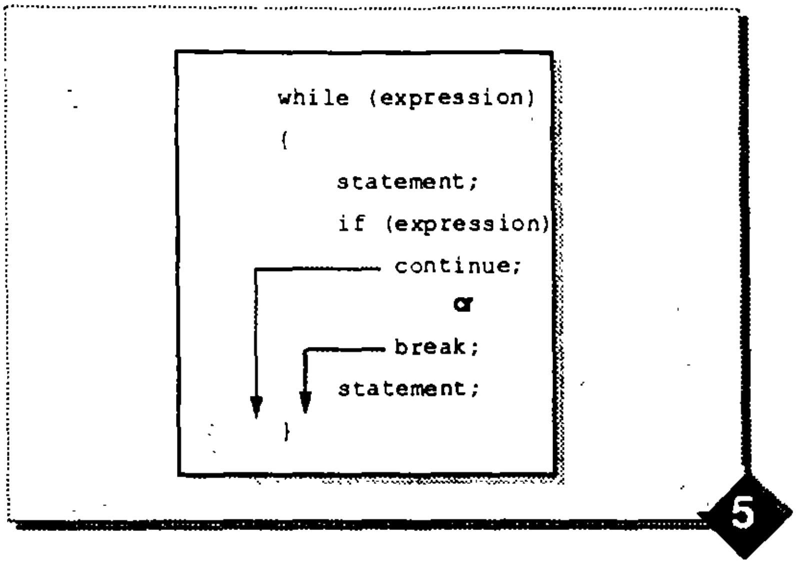
Demo

FORLOOP.CPP is located in \DEMOS\MOD05. It shows the use of the **for** loop construct.

```
// FORLOOP.CPP    found in \demos\mod05
// A for loop has four phases of execution.
#include <iostream.h>

void main()               // definition for main func
{
    int iLCV;             // integer Loop Control Value

    cout << "The factors of 72 are: \n";
                          // initialization; test; increment
    for (iLCV = 1; iLCV <= 72; iLCV++)
    {                                  // body
        if ((72 % iLCV) == 0)          // of
            cout << iLCV << endl;      // the
    }                                  // loop
}
```

...continue and ...break Statements

Slide Objective
Explain continue and break in context of the looping constructs just covered.



```
while (expression)
{
    statement;
    if (expression)
        continue;
           or
        break;
    statement;
}
```

You have seen that the **break** statement is used in a **switch** construction to prevent fall-through execution of the **case** portions.

The flow of loops in C++ can also be modified with **break** and **continue** statements. When executed, **break** causes control to pass immediately after the loop; **continue** causes flow to pass to just after the last dependent statement in the loop body.

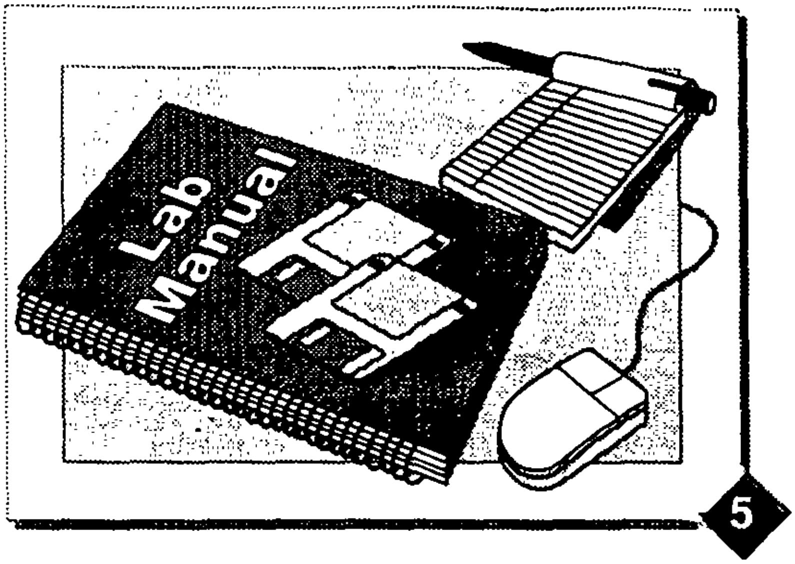
Demo

CONTBRK.CPP is located in \DEMOS\MOD05. It shows the use of **continue** and **break** statements.

```
// CONTBRK.CPP   found in \demos\mod05
// Contrast flow control differences:
//     continue vs. break
#include <iostream.h>

void main(void)          // definition of main func
{
    int nNumber;         // the following "while" is
                         // an infinite loop -- cout
                         // always is a positive value
    while (cout << "Enter an even number:")
    {
        cin >> nNumber;
        if ((nNumber % 2) == 1)
        {
            cout << "I said, ";
            continue;    // "continue" restart loop!
        }
        break;           // "break" exits loop!
    }
    cout << "Thanks. I needed that!\n";
}   // Note: A "void" main cannot return a value.
```

Lab 3: Using Statements and Expressions

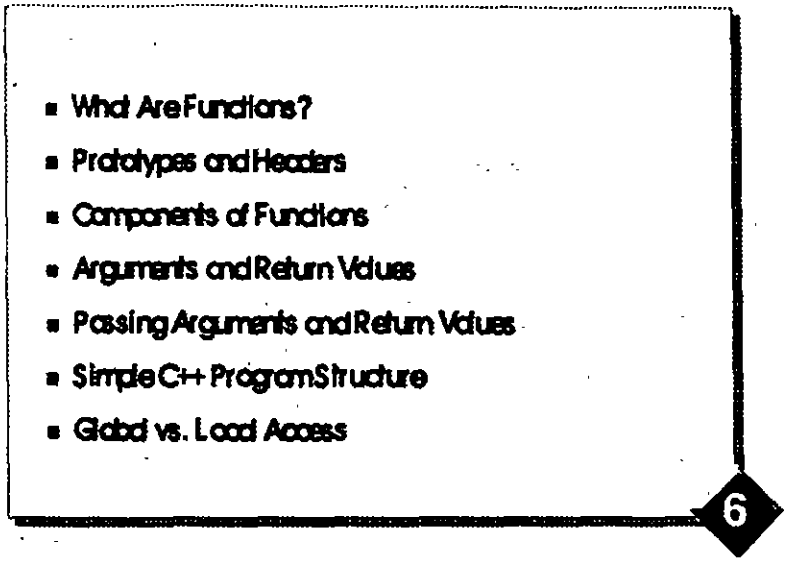
Slide Objective
Execute the lab solution.
Explain the purpose of the lab.
Ask students to read the scenario.



Module 6: Implementing a Simple Function

Σ Overview

Slide Objective
Provide an overview of the module contents.



- What Are Functions?
- Prototypes and Headers
- Components of Functions
- Arguments and Return Values
- Passing Arguments and Return Values
- Simple C++ Program Structure
- Global vs. Local Access

Module Summary

In the last few modules, you learned how to create a program by using variables and basic operators to form simple statements. You also used looping and conditional statements. As you will see, these statements, are also used to form the body of functions other than **main**. That is the subject of this module.

Remember that Visual C++ is a hybrid language that supports both the procedural and object-oriented approaches. In fact, most C++ programs are not strictly object-oriented. They must contain the global function **main**, and they normally contain other functions that exist outside of classes.

Objectives

Upon completion of this module, you will be able to:

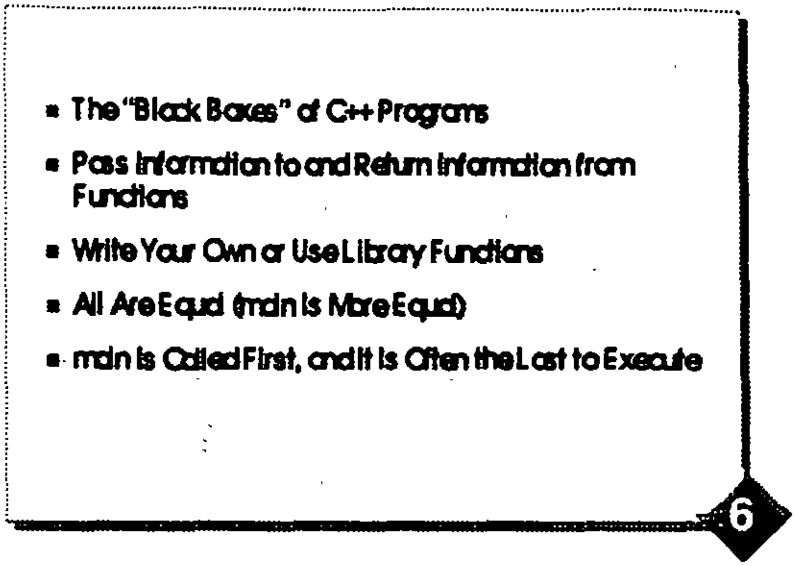
- Create prototypes for simple functions.
- Implement functions.
- Specify the visibility of a program's variables.

Lab

Implementing Simple Functions

What Are Functions?

Slide Objective
Describe the purposes, features and sources for functions. Remind students that a function will perform an O-O "behavior."



- The "Black Boxes" of C++ Programs
- Pass Information to and Return Information from Functions
- Write Your Own or Use Library Functions
- All Are Equal (main Is More Equal)
- main Is Called First, and It Is Often the Last to Execute

Essential Features of Functions

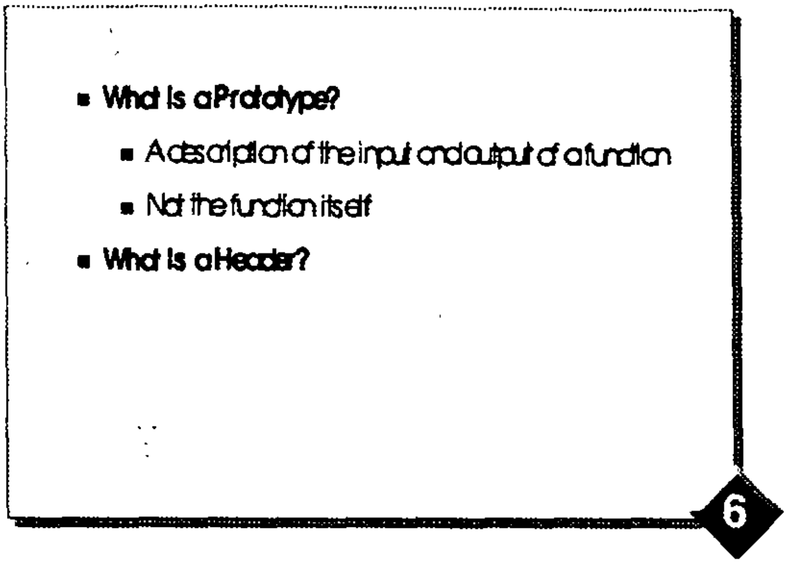
Functions represent the standard procedural black boxes of a C++ program. (From the object-oriented perspective, classes represent the major black boxes.) From a user's perspective, the important characteristics of a function are the information that a function receives (the arguments), the information returned (the returned value), and any side effects the function may cause.

Functions originate from two sources: either the user explicitly creates them, or they are "borrowed" from commercially written libraries. The **main** function is an example of the former, whereas the ANSI-standard C and iostream libraries are examples of the latter.

Though all functions are structurally and mechanically equivalent, the **main** function happens to be a little more equal than user-written functions. It is the first function called from the operating system, and often the last one executing when your program terminates. The **main** function also acts as the highest-level function, directing logic flow by calling other functions, prescribing the important test and looping conditions, and creating and sending messages to objects.

Prototypes and Headers

Slide Objective
Define prototypes and their purposes.



- **What Is a Prototype?**
  - A description of the input and output of a function
  - Not the function itself
- **What Is a Header?**

Prototypes and Header Files

Before each function is used or defined in C++, the compiler must see a description or *declaration* of each function. Declarations do not allocate any storage or produce code. A function declaration is also called a *prototype*.

Caution In older pre-ANSI C programs, prototypes were not supported.

Delivery Tips
Don't add too much detail about arguments or return values. Wait a few pages.

Defer explanation of "if any" for 3 pages.

In C++, functions take arguments and return values of very specific data types. An important part of designing a function is specifying this interface. A prototype describes this interface by providing three pieces of information:

- The function name
- The data types of any arguments
- The return data type, if any

Prototypes for commercially written functions in libraries are supplied in header files that are then included in programs.

More Facts About Prototypes

- They allow you to place functions in any order in the program.
- Prototypes don't make the program bigger.
- They permit checking for argument and return-type consistency at compile time.
- They don't place source code or define variables in headers.

Demos

RECTVOL1.CPP is found in \DEMOS\MOD06.

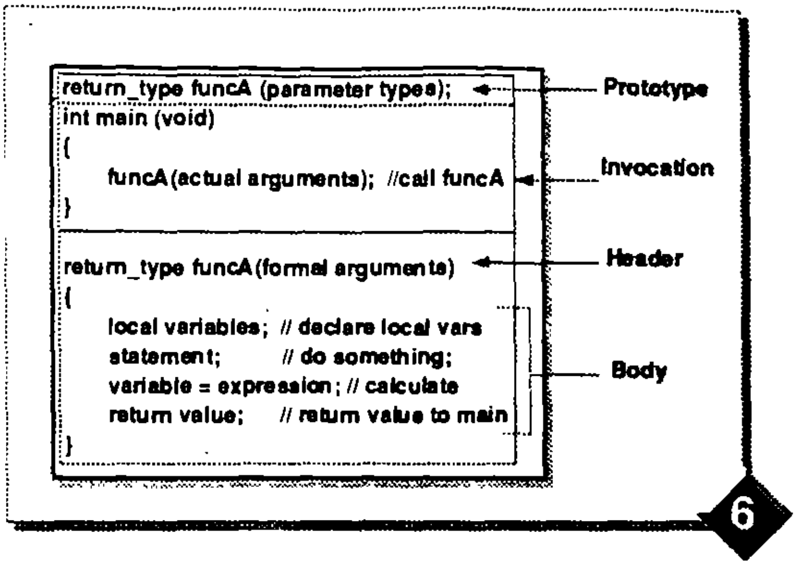
```
// RECTVOL1.CPP  found in \demos\mod06
// Shows use of user-supplied functions
                      // Preprocessor directive to include
                      // library-supplied func prototypes
#include <iostream.h>
                         // Prototype user-supplied func
long rectVol(int, int);  // denotes return-type, func-name
                         // and data-type of arguments.

int main(void)           // main func is special - "void"
{                        // denotes lack of arguments
    int nWidth, nHeight;
    cout << "Enter the width, in inches, of rectangle: ";
    cin >> nWidth;
    cout << "Enter the height, in inches, of rectangle: ";
    cin >> nHeight;
    cout << "\nThe volume is " // within a cout statement,
         << rectVol(nWidth, nHeight) // embedded func call
         << " square inches.";
    return 0;
}

/* rectVol function definition.
   Note: cast to long required to avoid truncation. */

long rectVol(int nW, int nH)
{
    return ((long) nW * (long) nH);
}
```

RECTVOL2.CPP is found in \DEMOS\MOD06.

```
// RECTVOL2.CPP  found in \demos\mod06
// Shows use of user-supplied functions
#include <iostream.h>

// coarse conversion from inches to millimeters
#define MM_PER_INCH  25

// prototypes user-supplied func
int convert(int);
long rectVol(int, int);

int main()
{
    int nWidth, nHeight;

    cout << "Enter the width, in inches, of rectangle: ";
    cin >> nWidth;
    cout << "Enter the height, in inches, of rectangle: ";
    cin >> nHeight;

    cout << "\nThe volume is "
        << rectVol(nWidth, nHeight)
        << " square inches.";
    cout << "\n    or about "
        << rectVol(convert(nWidth), convert(nHeight))
        << " square millimeters.";
    return 0;
}

int convert(int nInches)
{
    return nInches * MM_PER_INCH;
}

long rectVol(int nW, int nH)
{
    return ((long) nW * (long) nH);
}
```

Function Implementation

Slide Objective
Summarize use of prototypes, introduce more detail with args and return values.



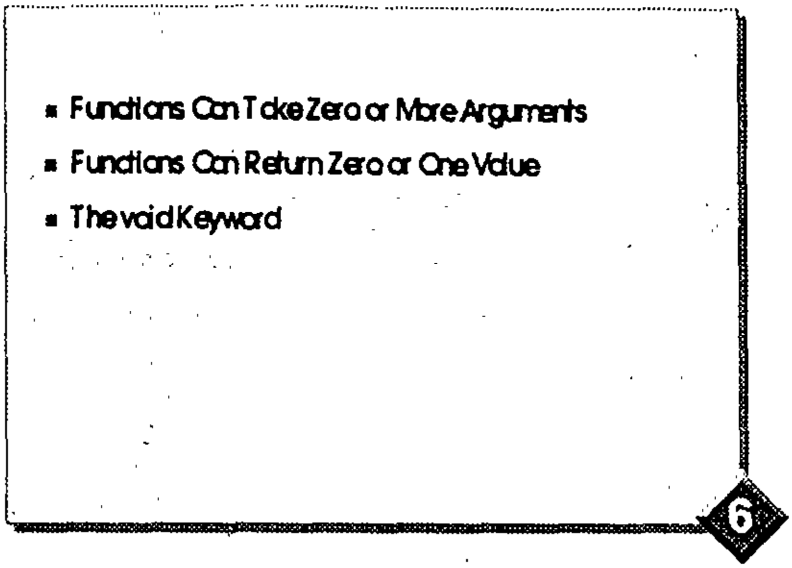
```
return_type funcA (parameter types);
int main (void)
{
    funcA(actual arguments); //call funcA
}

return_type funcA(formal arguments)
{
    local variables; // declare local vars
    statement;       // do something;
    variable = expression; // calculate
    return value;    // return value to main
}
```

A function represents a general logical process. Its implementation requires four general steps:

1. Design the interface. Choose a name, the parameter types, and the type of the return value.
2. Implement the function. First, write the header of the function from the information generated in step 1. Then write the body of the function as required to perform the logical process. Keep the following in mind:
   - The function body is delimited by a pair of curly braces.
   - Most functions will probably define local variables and contain a number of assignment and flow-control statements.
   - Normally a function will also contain at least one statement that calculates and returns a value.
   - Most statements are terminated by a semicolon.
3. Prototype the function. Create a declaration statement for your new function at the top of your source file. The easiest way to do this is to cut and paste the header, then add a terminating semicolon.
4. Test the function by using typical and limiting values for actual arguments.

Arguments and Return Values

Slide Objective
Detail the "use of" and "lack of" arguments and/or return type.



- Functions Can Take Zero or More Arguments
- Functions Can Return Zero or One Value
- The void Keyword

6

In C++, you can create functions that take zero or more arguments, and return zero or one value.

Key Points
Explain "if any" within context of the two uses for "void":
1: void func();
No return type
2: int func(void);
No arguments

The **void** keyword in a function prototype can be interpreted as "nothing"; either no arguments are required, or no returned value is generated.

Tip The **void** keyword was added in ANSI C. In K&R, all functions were required to return a value.

Examples

Here is a **sqrt** function that takes a **double** as an argument and returns a value of type **double**.

```
double sqrt(double);
```

The **srand** function takes an **unsigned int** as an argument and returns no value.

```
void srand(unsigned int);
```

The **rand** function takes no arguments and returns a value of type **int**.

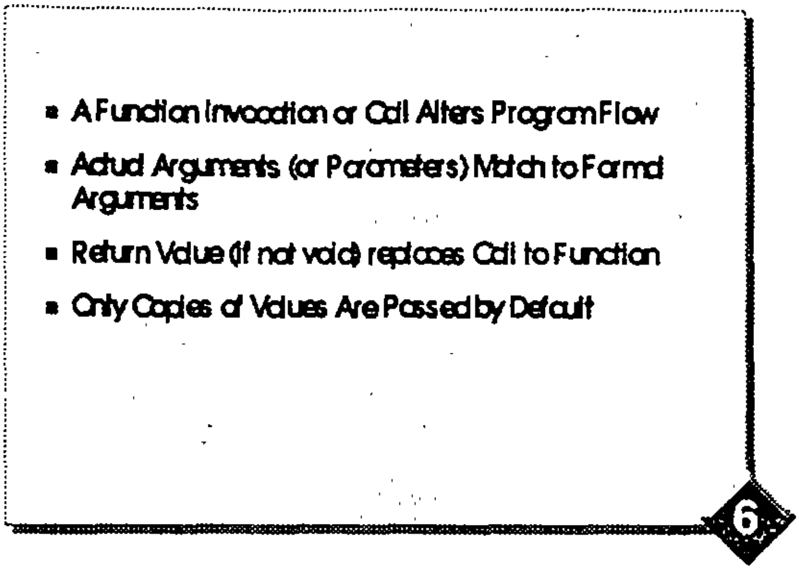
```
int rand (void);
```

The **tzset** (time zone set) function takes no arguments and returns no values.

```
void tzset(void);
```

Passing Arguments and Return Values

Slide Objective
Complete argument and return values detail. Summarize use of functions.



- A Function Invocation or Call Alters Program Flow
- Actual Arguments (or Parameters) Match to Formal Arguments
- Return Value (if not void) replaces Call to Function
- Only Copies of Values Are Passed by Default

A function invocation or call is an expression that drastically alters the normal linear program flow. When a call is executed, two important events occur:

- The values of the actual arguments in the function call are copied into the formal arguments.
- Control passes to the first executable line in the function.

Tip The function call operator is in Appendix B, the Operator Precedence chart.

The statements inside a function continue to execute until one of the following occurs:.

- A **return** statement is executed.
- The ending curly brace of the function is encountered. This is equivalent to returning no value.

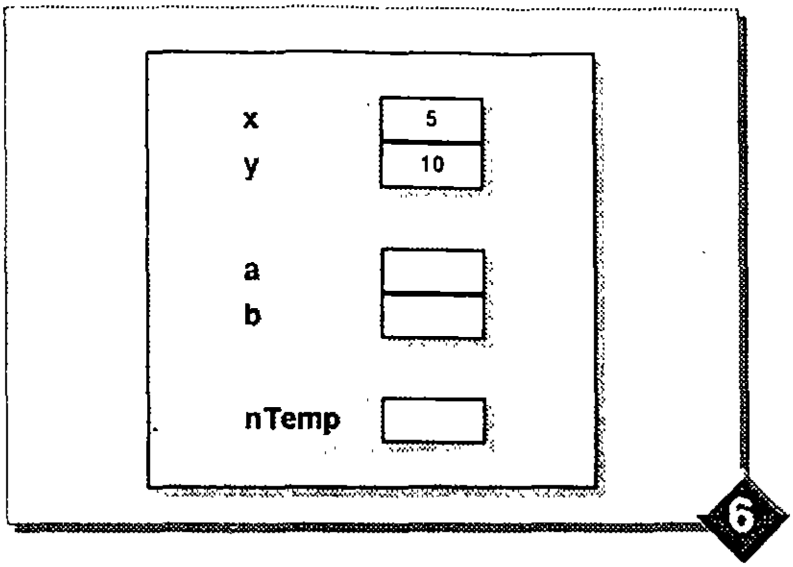
At this point, control passes back to the call that invoked the function. If a value is returned, that value replaces the entire function-call expression. The function call is said to *resolve to that value*. Program execution continues from that point.

Tip Calls to functions that return void are the only expressions in C++ that do not resolve to a value.

The default mechanism whereby values are passed to and from functions is termed *call by value*. With this mechanism, only copies of values are passed around. Each function still only has access to its formal parameters, local variables, and global variables.

Stack Architecture

Slide Objective
Begin a new subtopic detailing how arguments are passed to a function on the stack. Sub-topic includes a contrast of auto variables and globals.



x 5
y 10
a
b
nTemp

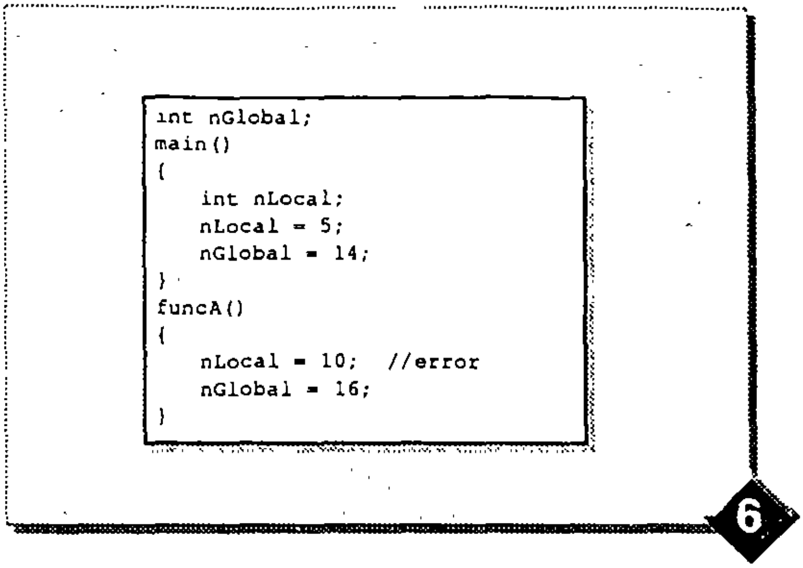
Demo

SWAP.CPP is found in \DEMOS\MOD06.

```
// SWAP.CPP      found in \demos\mod06
// Demonstrates the default calling conventions for
// functions.
#include <iostream.h>
                         // function prototype
void swap(int, int);     // swap is a function that
                         // takes two arguments
void main()
{                        // two local variables x and y
    int x (5), y (10);   // Note: equivalent to:
                         //       int x = 5, y = 10;
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
    swap (x, y);          // function call
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
}

void swap(int a, int b) // function definition
{
    int nTemp;

    nTemp = a;           // nTemp assigned the 5
    a = b;               // a assigned the 10 from b
    b = nTemp;           // b assigned the 5 from nTemp
}
```

Global vs. Local Access

Slide Objective
Complete the sub-topic on variables by dealing differences between auto (local) and global variables.



```
int nGlobal;
main()
{
   int nLocal;
   nLocal = 5;
   nGlobal = 14;
}
funcA()
{
   nLocal = 10;  //error
   nGlobal = 16;
}
```

Delivery Tips
COBOL programmers are used to "all global" variables. Be sure they understand the concept of locals and limited scope visibility.

Facts About Local and Global Variables

- Globals are typically defined at the top of the program.
- Globals come into existence before `main` and exist for the duration of the entire program.
- Globals can be used by any function in the program.
- Locals can be defined anywhere within a function, but are typically defined at the beginning of a function.
- Locals exist for the duration of the function invocation only, then they die or *go out of scope*.
- Locals can only be used within the function in which they are defined.
- In the absence of an explicit initializer, global variables are initialized to zero. By default, local variables are initialized to an unknown value—often referred to as "garbage."

As a rule of thumb, you should minimize the use of global variables to aid program modularity.

The topic of storage class and lifetime will be revisited in a future module.

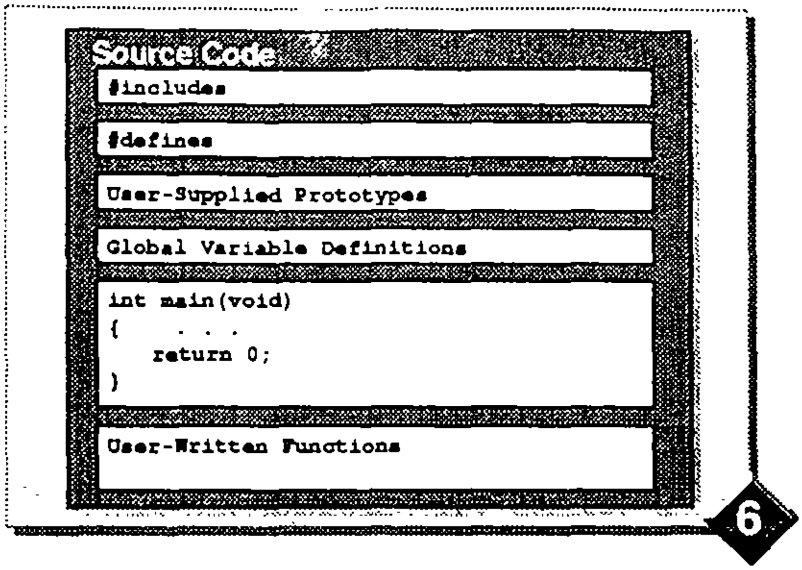
Demo

SCOPE.CPP is found in \DEMOS\MOD06. It demonstrates the local and global scope of variables.

```
// SCOPE.CPP     found in \demos\mod06
// This program demonstrates variable scope:
// Two identically named variables are declared
// and used in this program. This is legal because
// the variables have different scope.

#include <iostream.h>

// user-supplied function prototypes.  Read prototypes as:
                        // funcA is a function that takes
int funcA(void);        // no arguments and returns an int
int funcB(void);

// global variables
int nTemp = 5;          // nTemp has global scope

int main()
{
    cout << "Calling funcA..." << endl;
    cout << funcA() << endl;
    cout << funcA() << endl;
    cout << funcA() << endl;
    cout << funcA() << endl;
    cout << funcA() << endl;
    cout << endl;
    cout << "Calling funcB..." << endl;
    cout << funcB() << endl;
    cout << funcB() << endl;
    cout << funcB() << endl;
    cout << funcB() << endl;
    cout << funcB() << endl;
    return 0;
}

int funcA()
{   // The return value from funcA is the global nTemp.
    // nTemp is incremented by 5 each time funcA is called.
    nTemp += 5;
    return nTemp;
}

int funcB()
{   // The return value from funcB is a local called nTemp.
    // nTemp is created each time funcB is called
    int nTemp = 5; // and initialized with a value of 5.
    nTemp += 5;    // nTemp is incremented to 10.  Due to
    return nTemp;  // local scope the value is not retained.
}   // A local scope value may be returned-not retained.
```

Simple C++ Program Structure

Slide Objective
Summarize the use of functions and recommend that global, user-written functions be placed alphabetically after main().



Source Code

```
#includes

#defines

User-Supplied Prototypes

Global Variable Definitions

int main(void)
{    . . .
   return 0;
}

User-Written Functions
```

A nontrivial C++ application typically has six general portions to it:

#includes to declare commercially written functions. Header files also typically contain other declarations and preprocessor directives not yet covered in this course.

#defines to create manifest constants.

User-Supplied Prototypes declare the user-written functions actually defined later in the source file.

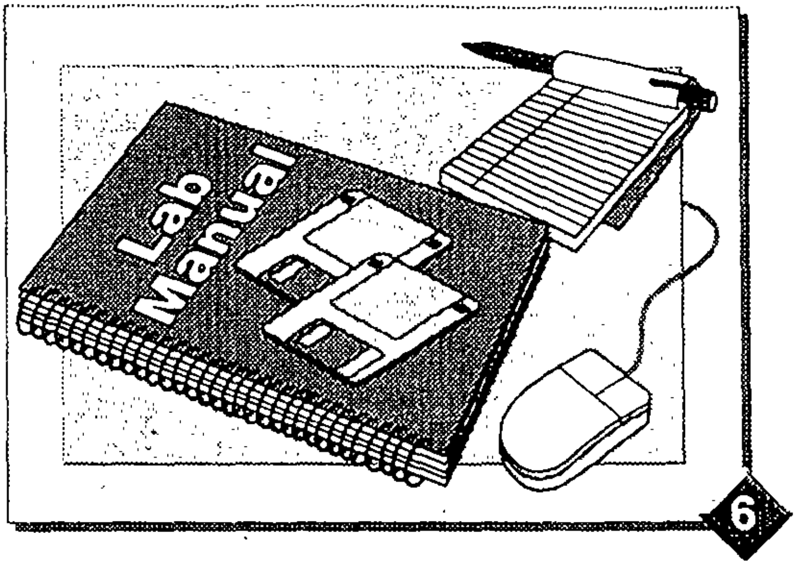
Global Variable Definitions create global variables.

The main Function: Every application has one and only one. It serves as the entry point to the application. By convention, it is before all other functions in the source file.

User-Written Functions: Divide the application into logical procedural units and factor out commonly used code to eliminate repetition.

Lab 4: Implementing Simple Functions

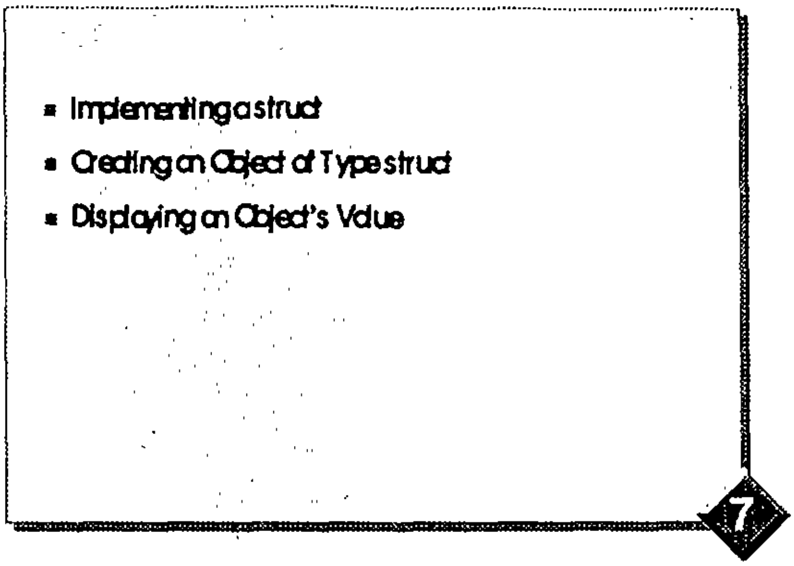
Slide Objective
Execute the lab solution.
Explain the purpose of the lab.
Ask students to read the scenario.



Module 7: Using Structures to Encapsulate Data

∑ Overview

Slide Objective
Provide an overview of the module contents.



- Implementing a struct
- Creating an Object of Type struct
- Displaying an Object's Value

Delivery Tips
COBOL programmers are familiar with DATA DIVISION and WORKING STORAGE contructs that are very similar to "structs". Expect to move quickly.

Module Summary

At this point you have explored the fundamental concepts of coding. In this module, you will integrate what you know about variables, datatypes, and functions to create your own custom data—structures.

Objectives

Upon completion of this module, you will be able to:

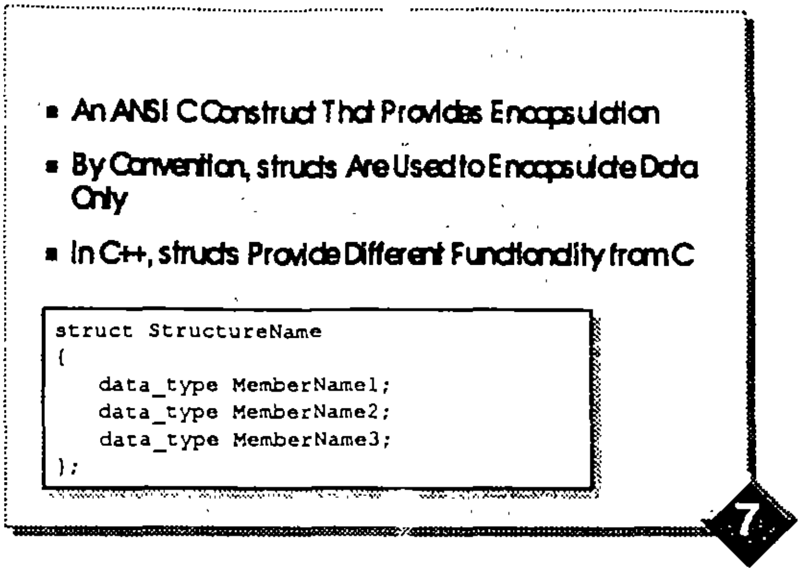
- Implement a **struct** (a custom data structure).
- Create objects of your data structure's type.
- Access the values contained in your data structure.

Lab

Using Structures to Encapsulate Data

What Is a struct?

Slide Objective
Provide a simple definition for structures.



- An ANSI C Construct That Provides Encapsulation
- By Convention, structs Are Used to Encapsulate Data Only
- In C++, structs Provide Different Functionality from C

```
struct StructureName
{
   data_type MemberName1;
   data_type MemberName2;
   data_type MemberName3;
};
```

What Is a struct?

The keyword **struct** is used to create a data structure. A data structure is created by the programmer and combines existing heterogeneous data types (integers, floating point numbers, characters, and so on) into an indivisible unit. The individual data fields in a **struct** are called members. A **struct** in C++ is similar to a record in other languages.

Operationally, to use a **struct** in a program, you must first declare the new **struct** data type. By this declaration, you are effectively making a new variable type. Like all declarations, a **struct** declaration provides information to the compiler, but does not allocate memory for data or code.

Key Points
This structure definition is analogous to a function prototype; it has no cost and takes no space.

Creating YourRect defines a memory area for the Rectangle variable.

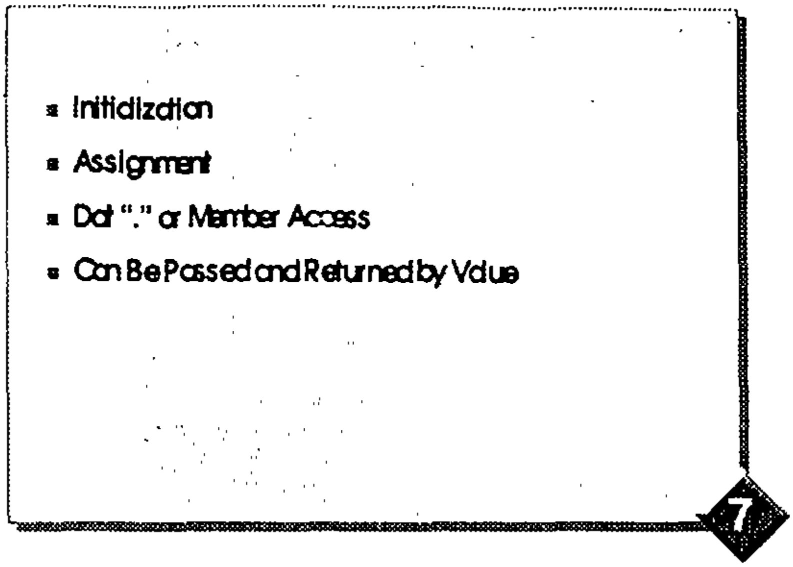
```
struct Rectangle
{
int nLength;
int nWidth;
short int Color;
};
```

Once a **struct** is declared as above, variables of type Rectangle can be defined.

```
Rectangle YourRect;
```

struct Operations

Slide Objective
Using terminology similar to the initialization and assignment of standard data type variables, explain initialization and assignment of structs. Cover ways the two are identical.



- Initialization
- Assignment
- Dot "." or Member Access
- Can Be Passed and Returned by Value

Initialization and Assignment

Recall from an earlier module that there are two ways to provide actual values for variables: initialization and assignment. There is a subtle difference between initialization and assignment. Initialization is done when a variable is defined. Your program does not consider this an executable statement:

```
Rectangle YourRect = {3,4};
```

Notice that a literal initializer is provided for every data member (the 3 and the 4 above).

Assignment can only be performed on existing variables. It is an executable statement. Assignment can also be used to provide values to your data members.

```
MyBox = YourRect;
```

Member Access

Key Points
Using the dot operator to access struct members.

To return or assign values of individual data members, use the "." operator as follows:

```
YourRect.nLength = 3;
YourRect.nWidth = 4;
```

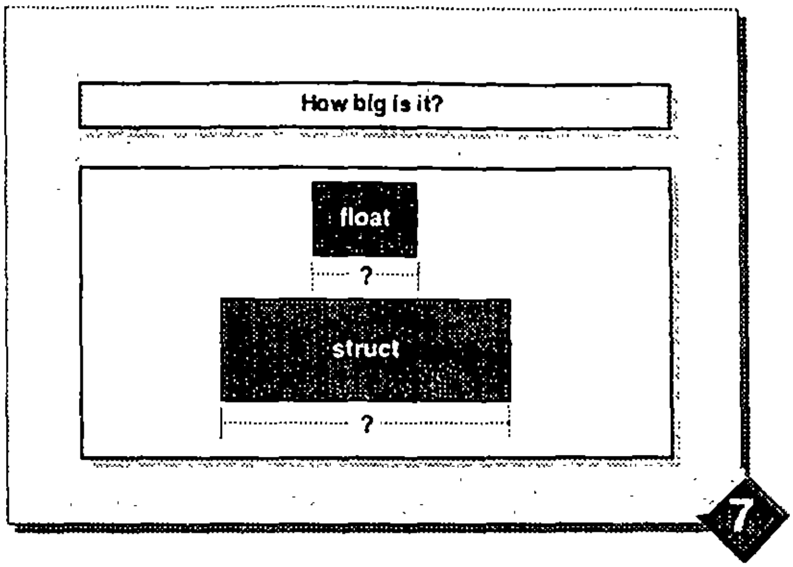
Demo

STRUCT.CPP is found in \DEMOS\MOD07.

```
// STRUCT.CPP   found in \demos\mod07
// This program demonstrates how to create and use a
// user-defined data structure using the struct keyword.
#include <iostream.h>
        // A user-defined data structure for Rectangle
struct Rectangle
{
    int x, y;             // x and y denote the center point
    int nHeight;
    int nWidth;
};

        // function prototype for GetArea function:
        // that takes a Rectangle argument and returns
        // a long data-type value
long GetArea(Rectangle r);

int main()
{
    long lArea;
        // An instance of a struct can get data through
        // initialization.   r1's is initialized below:
    Rectangle r1 = {0, 0, 100, 200};

        // An instance of a struct can get data through
        // assignment.  r2's members get assigned below:
    Rectangle r2;
    r2.x = 100;
    r2.y = 100;
    r2.nHeight = 300;
    r2.nWidth = 300;
                        // Call GetArea passing r1
    lArea = GetArea(r1);
    cout << "r1's area is " << lArea << endl;
                        // Call GetArea passing r2
    lArea = GetArea(r2);
    cout << "r2's area is " << lArea << endl;

    return 0;
}

                            // GetArea function definition
long GetArea(Rectangle r)   // takes a Rectangle struct as an
{                           // arg, calc's area (cast as a
    return ((long) r.nHeight * r.nWidth);  // long to avoid
}                                          // truncation)
```

Introduction to the sizeof Operator

Slide Objective
Familiarize students with use of the sizeof operator to determine space requirements for structs.


How big is it?

Key Points
Always let the compiler count the space needed.
Adds to portability of source code across platforms.
Padding may be changed by compiler options.
Compiler never miscounts.

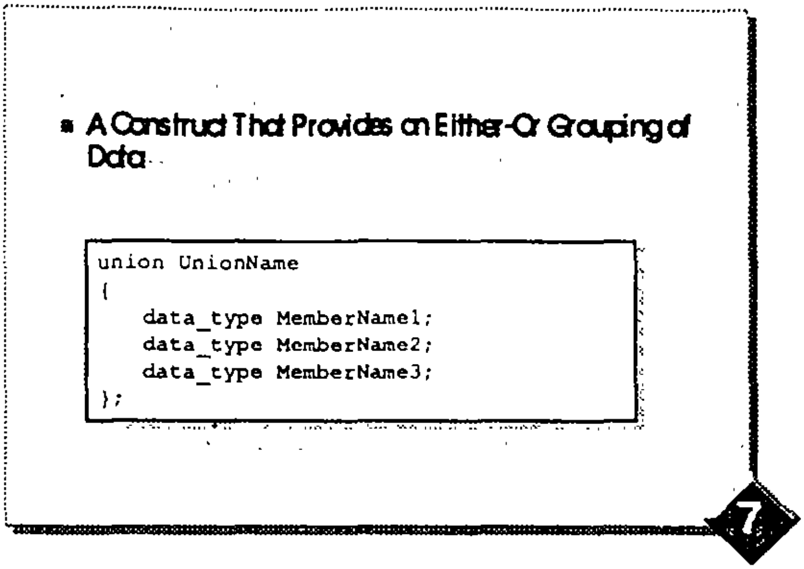
The **sizeof** operator yields the size of its operand in bytes. This operand can be either a type name (in which case the name must be enclosed in parentheses), or an expression. When the **sizeof** operator is applied to an object of type **char**, it yields 1 (byte). When it is applied to a **struct**, it yields the total number of bytes in that **struct**. This size is the sum of the size of all of the members plus any padding. Unlike other operators, **sizeof** is a compile-time operator; the compiler resolves the expression, replacing it with an integral constant.

Example

```
Rectangle yourRect;
int nBytes = sizeof(float);
. . .
nBytes = sizeof(yourRect);
```

What Is a Union?

Slide Objective
Unions are included for completeness. Analogous to the COBOL 'redefines' clause. Major problem: Unions are contrary to OOD views of black-box pgmg. Requires outside code with knowledge of some variable to tell what type of data is inside.



- A Construct That Provides an Either-Or Grouping of Data

```
union UnionName
{
    data_type MemberName1;
    data_type MemberName2;
    data_type MemberName3;
};
```

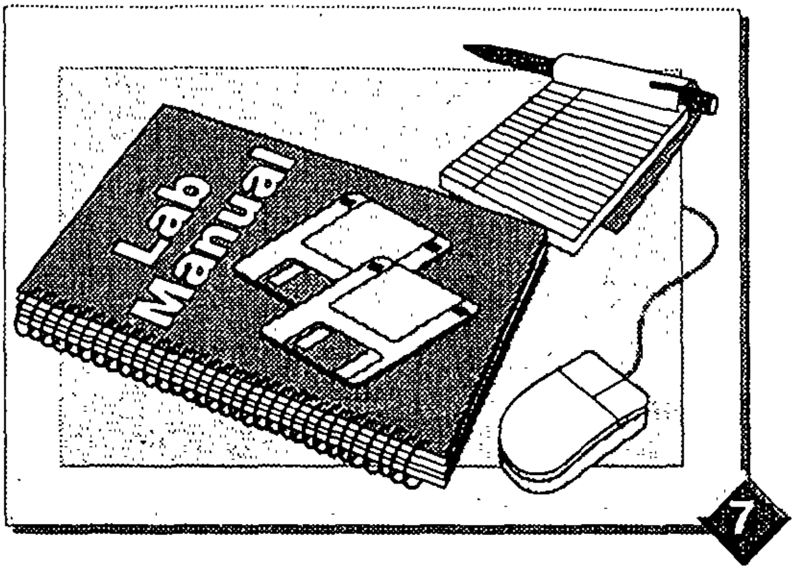
What Are Unions?

A **union** populates only one of its members at a time. You might want to use a **union** in lieu of a **struct** if the struct is very large and you only need access to a small portion of its data members. In a **union**, data members overlap, saving memory, but only one data member is populated with valid data at any given instant. A union can also be used to provide a generalized approach to some problems.

```
union Salary
{
float fHourly;
unsigned long ulSalary;
};
```

Lab 5: Using Structures to Encapsulate Data

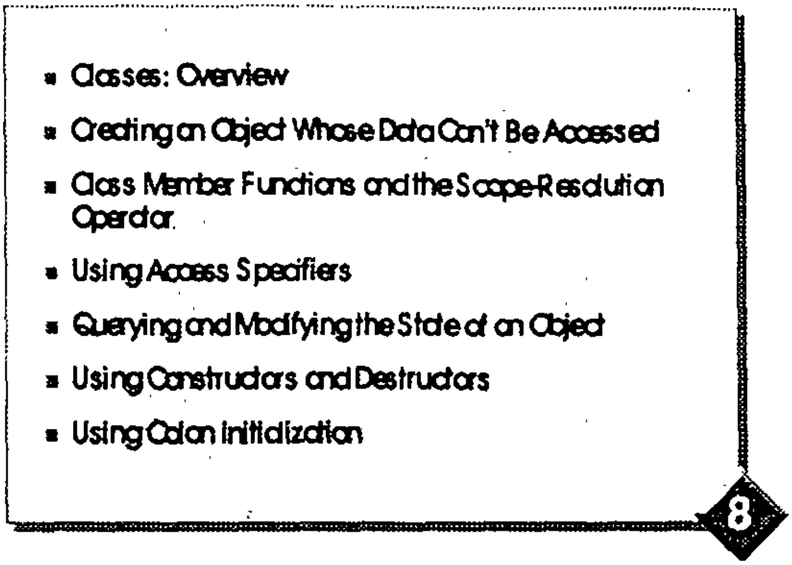
Slide Objective
Execute the lab solution.
Explain the purpose of the lab.
Ask students to read the scenario.



Module 8: Writing a Simple Class

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Classes: Overview
- Creating an Object Whose Data Can't Be Accessed
- Class Member Functions and the Scope-Resolution Operator
- Using Access Specifiers
- Querying and Modifying the State of an Object
- Using Constructors and Destructors
- Using Colon Initialization

This is the first of five modules on classes. The features of classes that you learn in this module will be extended in the next four modules, culminating in your ability to derive new classes through inheritance.

Module Summary

You are about to see that structs and classes are intimately related. In this module, you'll actually create a class using the same information contained in the **struct**.

A class is the central OO construct that you will be programming with in this course. You will explore the entire process—from declaring the class to creating an object of that class type in a program.

Objectives

Upon completion of this module, you will be able to:

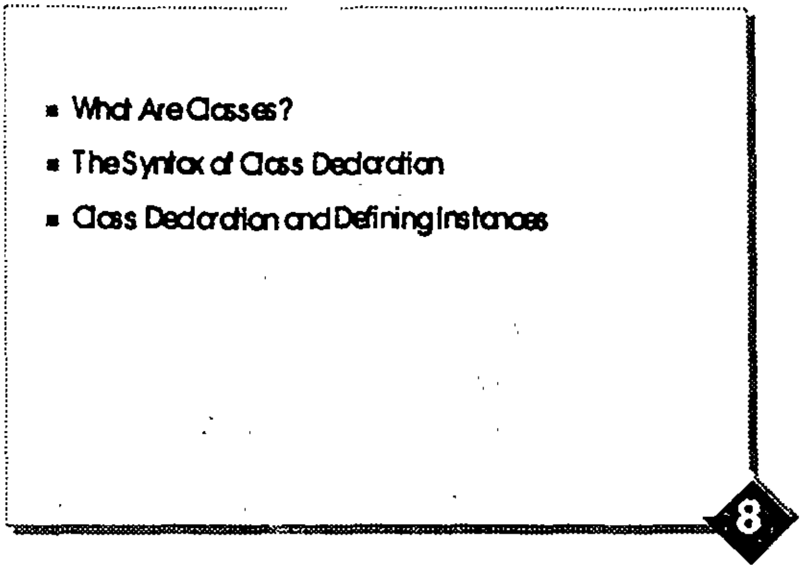
- Declare a class.
- Create data members for your class.
- Create member functions for your class.
- Use access specifiers to protect data.
- Create constructors and destructors.
- Use colon initialization.

Lab

Creating Classes and Member Functions

Classes: Overview

Slide Objective
Introduce the topic of C++ classes. The following pages have the details.



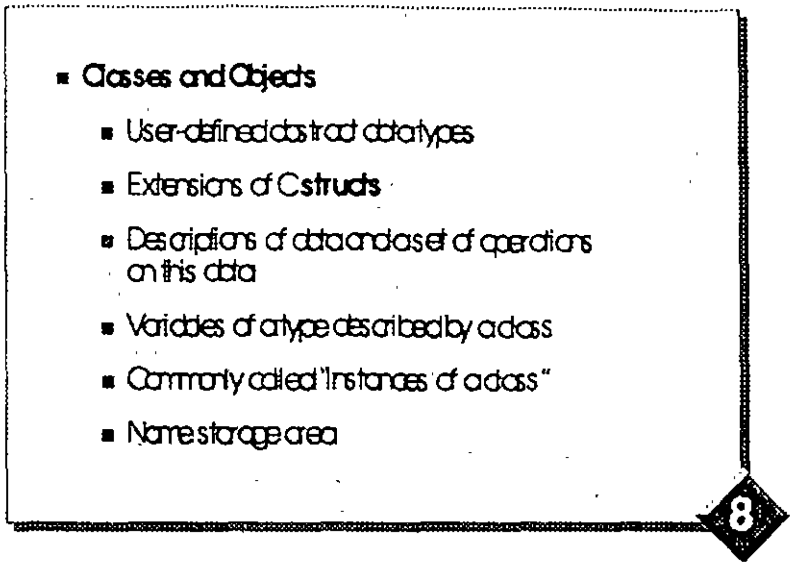
- What Are Classes?
- The Syntax of Class Declaration
- Class Declaration and Defining Instances

Delivery Tips
Cover the next 4 pages, detailing to students how much they already know about classes.

The next couple of pages cover the fundamentals of classes.

What Are Classes?

Slide Objective
Remind students of facts they already know about classes to put all the details in order.



- Classes and Objects
  - User-defined abstract data types
  - Extensions of C structs
  - Descriptions of data and a set of operations on this data
  - Variables of a type described by a class
  - Commonly called "Instances of a class"
  - Name storage area

Objects

Without reviewing the earlier discussion of OO programming, here's a review of the important points about objects. OO programs are designed in terms of objects rather than functions. This has the helpful side effect of making your programs more closely resemble real-world systems, thus making them easier to design. Objects contain data and functions. Classes of objects are related by the types of data and functions they contain, though each object (being an individual instance of a class) has its own data. In fact, the relationship between an object and a class is much the same as between a variable and a data type.

Classes

Classes, like structs, provide user-defined data structures to your programs. Classes specify both data members and the functions that manipulate the data members. Once a class has been declared, your program can instantiate many objects that class type. Classes are generally declared at file scope.

Access to Class Members

Data and functions can be hidden from the rest of your program by the use of keywords. This is an important feature of classes, the details of which will be discussed later in this module.

Typical Member Functions

Every class has at least one constructor function used to instantiate its objects.

Every class has a destructor function used to destroy its objects.

Typically a class will also have one or more member functions to get and set data members, display information to the user, and manipulate its data according to the needs of the program.

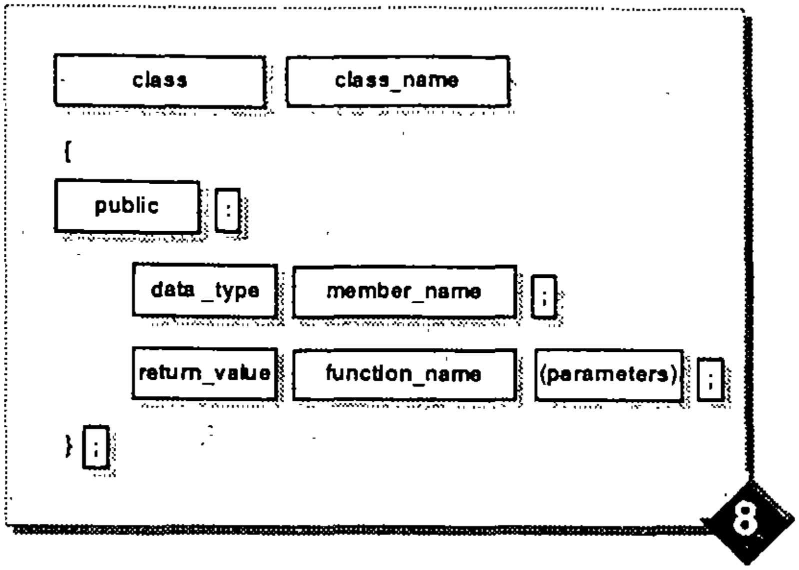
The Syntax of Class Declaration

Slide Objective
Detail similarities: The class depicted has 3 changes from a struct:

Uses "class" instead of "struct"

Uses "public:" which is default for a struct

Uses a member function (which is legal in a C++ struct).



```
class class_name
{
public :
    data_type member_name ;
    return_value function_name (parameters) ;
} ;
```

Class Declarations

A class declaration begins with the class keyword, followed by the class name, followed by an open curly brace. Within the curly braces, data members are declared and member functions are prototyped. Though the body of member functions can be defined within a class declaration, the convention is to define the body of member functions outside the class declaration. You will examine member-function definitions later in this module.

After the open curly brace of a class declaration, and prior to declaring any data members or functions, an access-specifier keyword followed by a colon must appear:

```
public:
```

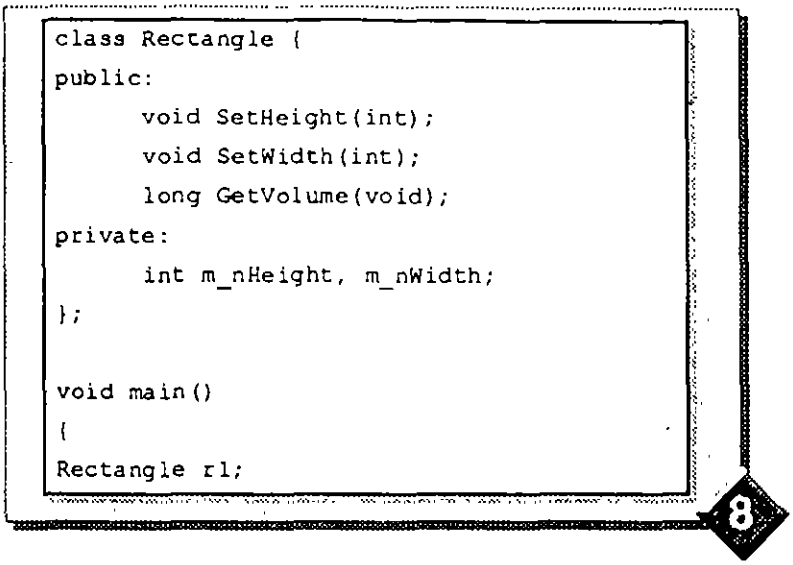
There are three types of access that can be specified: public, private, and protected. Access limitations that these keywords provide will be discussed later in this module. Access specifiers can appear in any order, or as often as you like (one keyword per member if you wish).

Following the access specifier, data members or function prototypes are listed. For data members, variable names and their data type are added much the same as you saw in earlier programs. Remember to terminate the declaration with a semicolon. Member functions are also prototyped similarly to functions that appear in the body of a program (outside a class declaration). The function's **return** type appears to the left. The function's name and a list of its arguments enclosed in parentheses appear to the right. The statement is terminated with a semicolon.

The class declaration is ended with a closing curly brace followed by a semicolon.

Class Declaration and Definition

Slide Objective
Using the same terminology as the Structures module, describe the declaration and definition of Classes.



```
class Rectangle {
public:
      void SetHeight(int);
      void SetWidth(int);
      long GetVolume(void);
private:
      int m_nHeight, m_nWidth;
};

void main()
{
Rectangle rl;
```

The code fragment shown in the foil is from a demo program that you will examine in a moment. Notice the last line:

```
Rectangle rl;
```

This is a definition for an object of type Rectangle. It creates an instance of a rectangle for your program to use.

Delivery Tips
Watch usage of terminology:
Don't declare classes.
DO "instantiate" objects.
Don't initialize classes.
DO initialize objects.
DO access member data.
DO refer to "data members."
DO refer to "member functions."

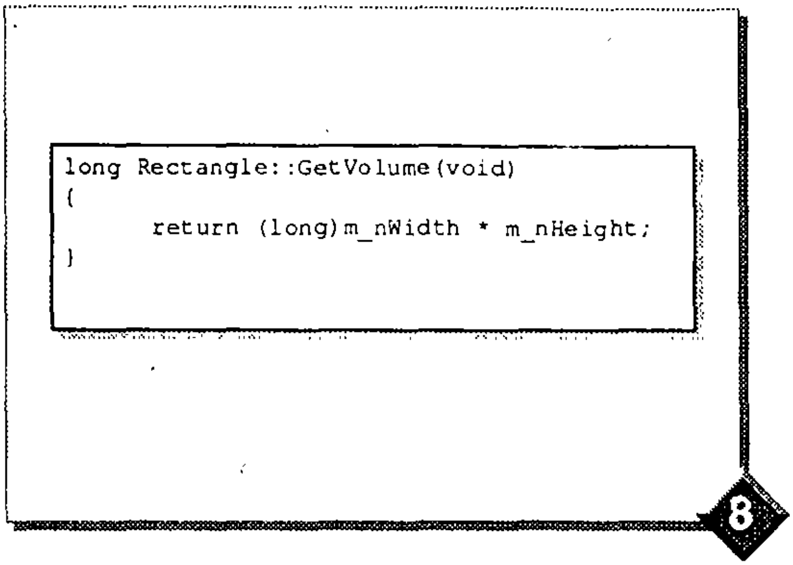
Demo

MEMBER.CPP is located in \DEMOS\MOD08.

```
// MEMBER.CPP found in \DEMOS\MOD08
// Using access specifiers and accessor member functions
#include <iostream.h>

/************ Rectangle Class Declaration ***************/
// Interface to x and y coordinates not yet implemented.
class Rectangle
{         // Interface is public
public:                    // Sometimes called mutators,
    void SetHeight(int); // Set and Get func's allow class
    void SetWidth(int);  // users to access attributes
    long GetVolume(void);// of an object
private:  // Data members are private
    int m_nHeight, m_nWidth;
};

/************** Rectangle Member Functions **************/
void Rectangle::SetHeight(int h)
{
    m_nHeight = h;
}

void Rectangle::SetWidth(int w)
{
    m_nWidth = w;
}

long Rectangle::GetVolume(void)
{
    return (long)m_nWidth * m_nHeight;
}

/****************** Small Test Program *****************/

int main()
{
    Rectangle r1;        // Declare a Rectangle object, r1
    r1.SetHeight(15);
    r1.SetWidth(10);
    // Note: Un-comment the following line to reveal
    // an error message concerning private access!
    // cout << "width is " << r1.m_nWidth;
    cout << "The volume of rectangle r1 is "
         << r1.GetVolume() << '.' << endl;
    return 0;
}
```

Class Member Functions and the Scope-Resolution Operator

Slide Objective
Explain the use of the scope resolution operator, "::", used in previous Demo example.



```
long Rectangle::GetVolume(void)
{
      return (long)m_nWidth * m_nHeight;
}
```

By convention you will define the body of your member functions outside the class declaration. This is done to enhance the readability of class declarations. Following the declaration, you define the member funcions as shown on the foil.

The Scope Resolution Operator

As usual, the function's return value appears to the left followed by the name of the class to which the function is a member. The :: which follows the class name tells the compiler that the function's scope is at the level of that particular class. The actual code that forms the body of the function is defined within curly braces. In the example above, the **GetVolume** function merely returns the value of the data member **m_nWidth**. Notice that there is no terminating semicolon following a member function definition as there was following a class declaration.

In short, the scope-resolution operator takes a classname to its left and a member of that class to its right.

Key Points
Scope resolution operator is: class::member. Dot operator is: object.member.

So "scope" is used for the class, "dot" for the object.

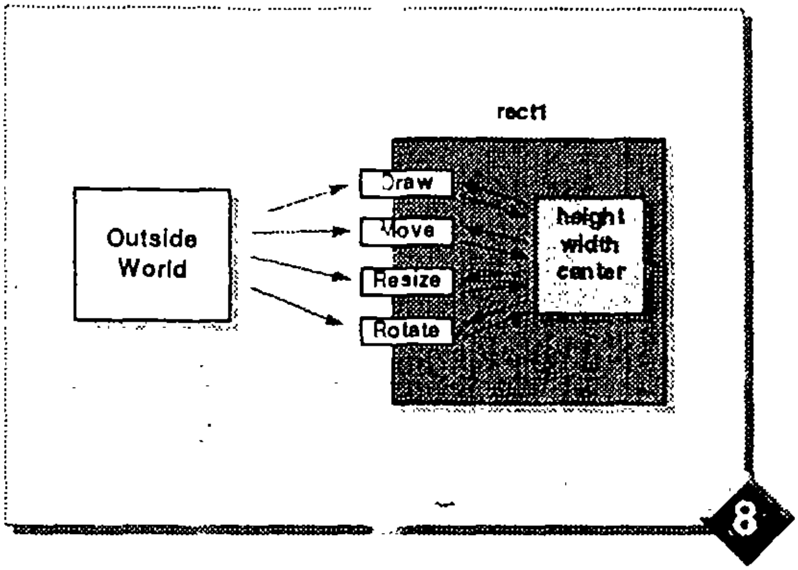
The Dot Operator

To access a member (usually a function) for an object, you use the dot operator. In the following example, the dot operator precedes the **GetVolume** function.

```
cout << "Volume is : << r1.GetVolume( ) << \n";
```

Using Access Specifiers

Slide Objective
Explain the uses for Access Specifiers based on OOD terminology:
data-hiding
black-box
hidden implementation
class-defined interface.



Delivery Tips
This graphic was presented in the OOD module.

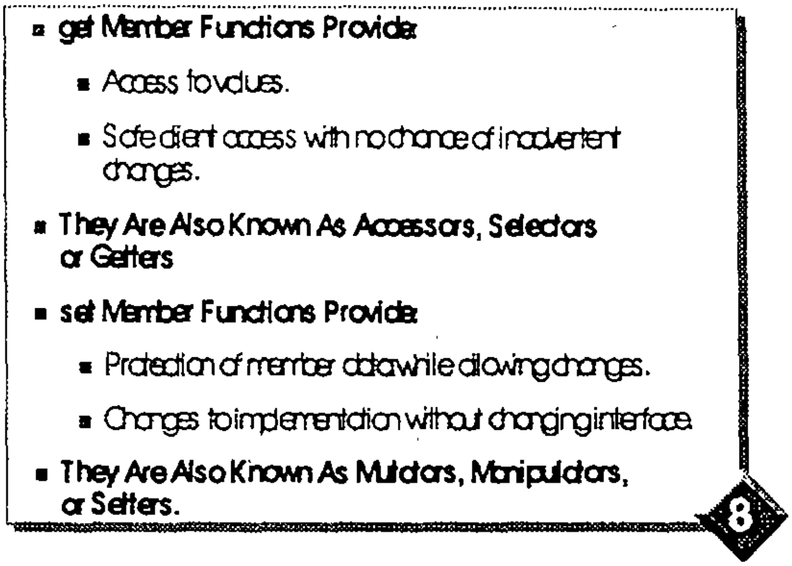
Public members are accessible to everything in your program. **Private** members are accessible only to class member functions. (There are exceptions to this rule which fall outside the scope of this course. See a C++ reference manual for a description of **friends.**) **Protected** members are accessible to class member functions and member functions of classes related through inheritance. (Inheritance will be examined in an upcoming module.)

Tip The following general advice General advice applies to access specifiers.

- Declare member functions as public.
- Declare data members as private.
- Provide access member functions to set and retrieve values for data.

Querying and Modifying the State of an Object

Slide Objective
Explain benefits of controlling user access to the data members through "accessor" and "mutator" functions.



- **get Member Functions Provide:**
  - Access to values.
  - Safe client access with no chance of inadvertent changes.
- **They Are Also Known As Accessors, Selectors or Getters**
- **set Member Functions Provide:**
  - Protection of member data while allowing changes.
  - Changes to implementation without changing interface.
- **They Are Also Known As Mutators, Manipulators, or Setters.**

Disadvantages of set and get Functions

If there are a lot of data members, the interface can become cumbersome because of a large number of functions. In a case like this, it might be wise to mark the data members as **public** and allow direct access.

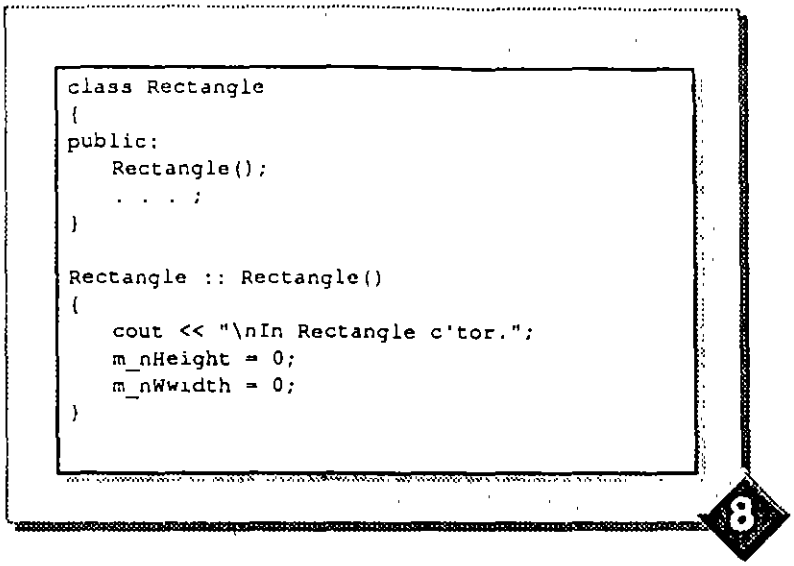
Demo

SETGET.CPP is found in \DEMOS\MOD08.

```
// SETGET.CPP found in \DEMOS\MOD08
// Demonstration of accessor/manipulator pairs.
// Note: Many commercial class packages refer to these
// as functions that access object attributes.
#include <iostream.h>

/************* Rectangle Class Declaration *************/
// Interface to x and y coordinates not yet implemented.
class Rectangle
{
public:
    void SetHeight(int); // Set member functions:
    void SetWidth(int);  // take an arg as a new value
    int GetHeight(void); // Get member functions:
    int GetWidth(void);  // take no args, return a value
private:
    int m_nHeight, m_nWidth;
};

/************** Rectangle Member Functions **************/
void Rectangle::SetHeight(int h)
{
    m_nHeight = h;
}

void Rectangle::SetWidth(int w)
{
    m_nWidth = w;
}

int Rectangle::GetHeight(void)
{
    return m_nHeight;
}

int Rectangle::GetWidth(void)
{
    return m_nWidth;
}

/***************** Small Test Program *******************/

int main()
{
    Rectangle r1;         // Declare a rectangle object, r1
    r1.SetHeight(15);     // Set height attribute
    r1.SetWidth(10);      // Set width attribute
    // cout << "width is " << r1.m_nWidth; // access!!
    cout << "The volume of rectangle r1 is "
         << (long)r1.GetHeight() * r1.GetWidth() << ".\n";
    return 0;
}
```

Constructors

Slide Objective
We have intentionally avoided the topic of initialization. Introduce "construction" as the method for building objects.



```
class Rectangle
{
public:
   Rectangle();
   . . . ;
}

Rectangle :: Rectangle()
{
   cout << "\nIn Rectangle c'tor.";
   m_nHeight = 0;
   m_nWwidth = 0;
}
```

Constructors

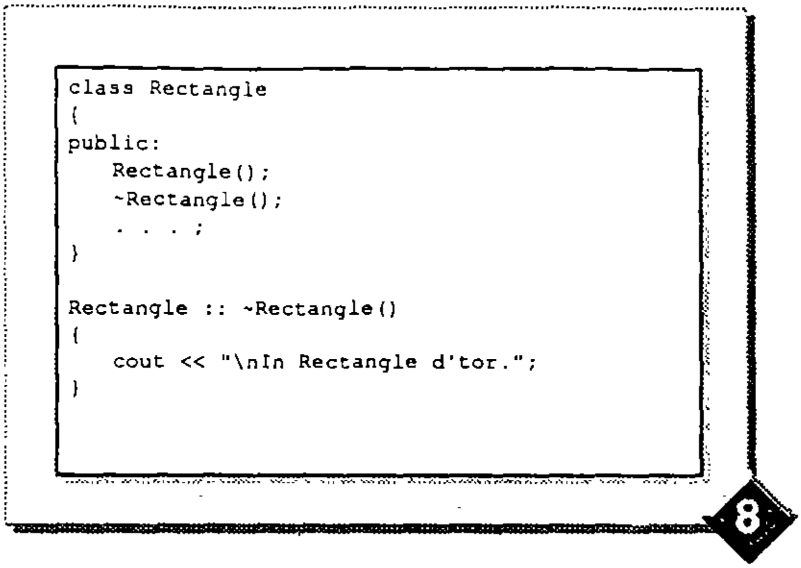
A constructor is called at the point the object is created. The purpose of a constructor is to set the initial state of an object—that is, to assign appropriate values to an object's data members (and perhaps other related values).

Every class has at least one member function called a constructor. It is not mandatory that you create a constructor. If you do not supply one, the compiler will create one for you. A constructor always has the same name as the class. Default constructors must be called with no arguments.

A constructor executes any code provided in its body, but cannot return a value. Constructors must be prototyped as returning no value; void is not allowed. A constructor is sometimes abbreviated as c'tor.

Destructors

Slide Objective
Introduce topic of object destruction. Don't go too deep — students won't know any valid reasons or features for a destructor for some time.



```
class Rectangle
{
public:
   Rectangle();
   ~Rectangle();
   . . . ;
}

Rectangle :: ~Rectangle()
{
   cout << "\nIn Rectangle d'tor.";
}
```

Destructors

For Your Information
Stuck for an example?
If pushec 'or an example of a valid C'tor and D'tor, propose a database object where the C'tor handles login and dbopen, the D'tor does signoff and dbclose.

Every class has exactly one destructor. Its purpose is to do any "clean-up" work. A destructor always has the same name as the class, but it is distinguished from the constructor by a tilda (~) prefix:

```
Rectangle :: ~Rectangle()
```

It is not mandatory to supply a destructor; the compiler will do it for you. Destructors cannot return a value. They are called at the point the object is destroyed. A destructor is sometimes abbreviated as d'tor.

Destructors are called when a local object with block scope goes out of scope, or when a program ends and global objects exist.

Demo

CTORDTOR.CPP is located in \DEMOS\MOD08. It shows the use of a constructor and a destructor.

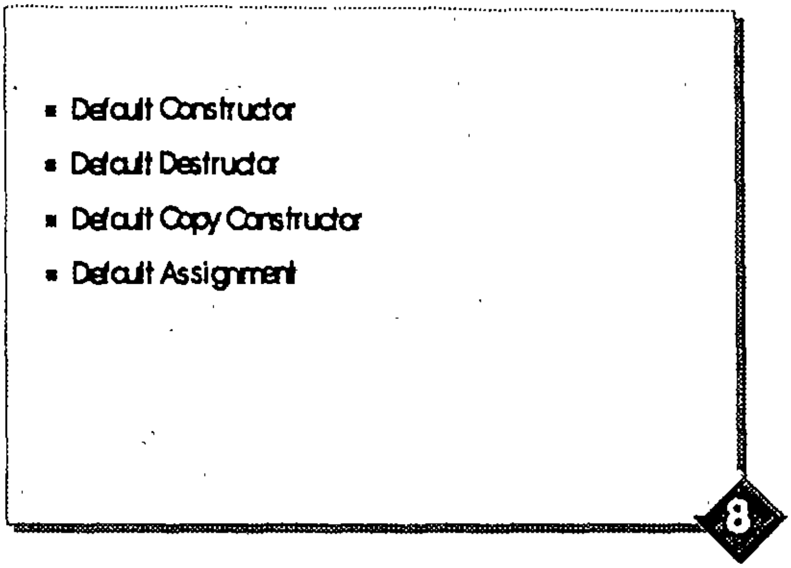
```
// CTORDTOR.CPP found in \DEMOS\MOD08
// Includes default constructor and destructor
#include <iostream.h>

/************* Rectangle Class Declaration **************/
// Interface to x and y coordinates not yet implemented.
class Rectangle
{
public:                    // Construction section:
    Rectangle();           // constructor (no return value)
    ~Rectangle();          // destructor (no args, no ret)
    void SetHeight(int);   // Attributes section:
    void SetWidth(int);
    long GetVolume(void);
private:                   // Implementation section:
    int m_nHeight, m_nWidth;
};

/************* Rectangle Member Functions ***************/
Rectangle::Rectangle()     // Definition of constructor
{                          //   name matches class name
    cout << "Rectangle c'tor.\n";
    m_nHeight = 0;         // free access to data members
    m_nWidth = 0;
}                          // never return a value!

Rectangle::~Rectangle()    // Definition of destructor
{                          //   ~ and class name
    cout << "Rectangle d'tor.\n";
}

void Rectangle::SetHeight(int h)
{
    m_nHeight = h;
}

void Rectangle::SetWidth(int w)
{
    m_nWidth = w;
}

long Rectangle::GetVolume(void)
{
    return (long)m_nWidth * m_nHeight;
}

```

(continued)

```
/****************** Small Test Program ******************/

int main()
{
    Rectangle r1;           // Declaring a class object (the
                            // constructor is called)
    // Rectangle r2();      // This is a function prototype!

    cout << "The initial volume of rectangle r1 is "
         << r1.GetVolume() << endl;
    r1.SetHeight(15);       // Set attributes for r1
    r1.SetWidth(10);
    cout << "The volume of rectangle r1 is "
         << r1.GetVolume() << endl;
    return 0;               // Note: A call to the d'tor
}                           // is not coded!
```

Default Class Operations

Slide Objective
Staying very high-level, explain the 'defaults' given to each class.



- Default Constructor
- Default Destructor
- Default Copy Constructor
- Default Assignment

In the absence of user-supplied versions of the following member functions, the compiler will supply a simple built-in default version.

Delivery Tips
Cover default c'tor and d'tor.

Simply define the default copy c'tor and assignment operator, but stay clear of details!

A default constructor is a constructor that takes no arguments. The compiler will supply a default c'tor *only if no constructor is supplied for the class.* The default c'tor supplies the same functionality as for standard types like **int**, giving global objects an initial value of zero and local objects and unknown (garbage) value. Note that the default constructor is essentially what you used when you built struct data instances.

If no destructor is supplied for a class, the compiler supplies a default destructor, which, from the user's perspective, does nothing.

As with a **struct**, objects can be created from an existing object of the same type:

```
Rectangle rect1;
rect1.SetHeight(15);
rect1.SetWidth(20);
Rectangle rect2(rect1);  //copy c'tor
```

This operation is technically known as a *copy construction*; here it is provided automatically by the compiler. In the module on conversions, you will see how to supply your own version.

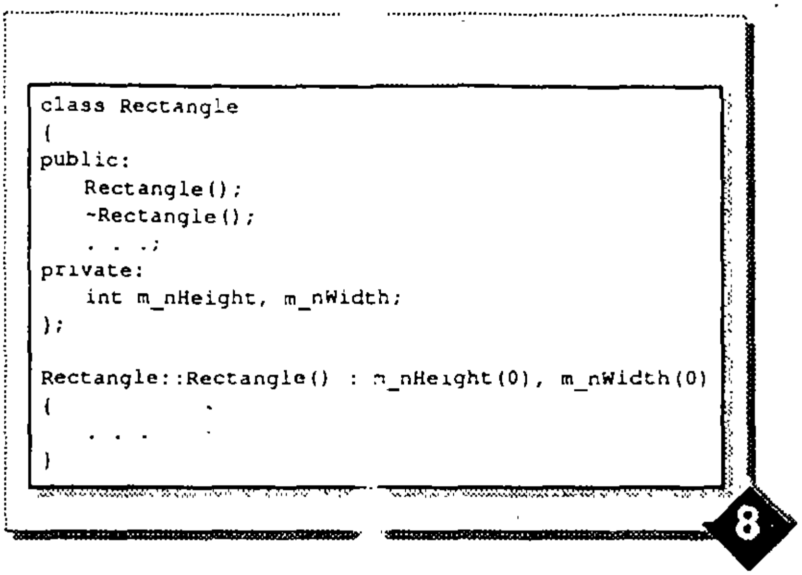
Assignment from one object to another object of the same type is inherently supported by a default assignment operator:

```
rect1 = rect2;
```

Supplying your own version by using the operator-overloading capability of C++ is beyond the scope of this course.

Colon Initialization

Slide Objective
Explain colon initialization syntax. Defer discussion of why it is a preferred method to initialize member data in c'tors until later.



```
class Rectangle
{
public:
   Rectangle();
   ~Rectangle();
   . . .;
private:
   int m_nHeight, m_nWidth;
};

Rectangle::Rectangle() : m_nHeight(0), m_nWidth(0)
{
   . . .
}
```

In an earlier module, a distinction was drawn between initialization and assignment. Initialization happens when an object is created and assignment takes place during its normal life. Since neither of these conditions is true at the time a class declaration is made, initialization and assignment are illegal within class declarations. Data members, therefore, are initialized by constructors, using the colon syntax shown above.

A discussion of why colon initialization is preferred will be put off until a later module. As a rule of thumb, though use the colon-initialization syntax in preference to assignment of data members in the constructor whenever possible.

Demo

COLONINI.CPP is found in \DEMOS\MOD08.

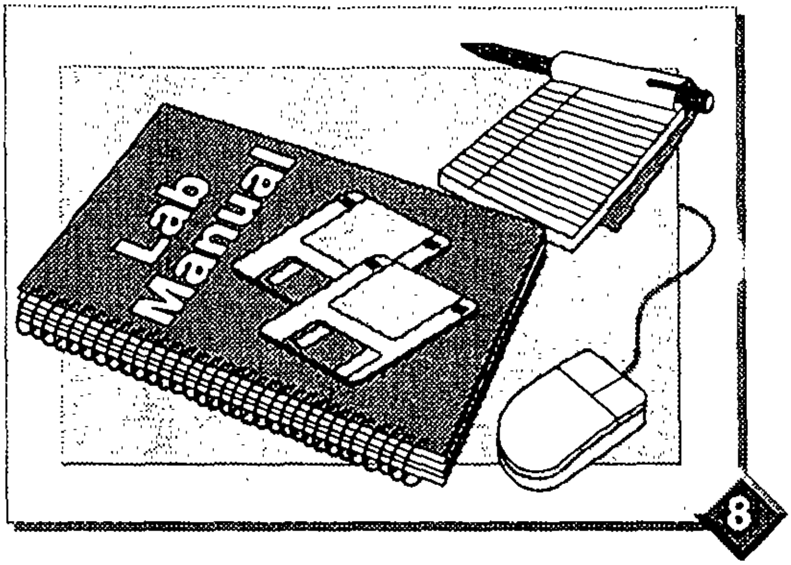
```
// COLONINI.CPP found in \DEMOS\MOD08
// Shows a constructor using colon initialization.
#include <iostream.h>

/************ Rectangle Class Declaration *************/
// Interface to x and y coordinates not yet implemented.
class Rectangle
{
public:
    Rectangle();          // construction
    ~Rectangle();
    void SetHeight(int); // attributes
    void SetWidth(int);
    long GetVolume(void);
private:                  // implementation
    int m_nHeight, m_nWidth;
};

/************** Rectangle Member Functions **************/
Rectangle::Rectangle()            // Constructors may use
    : m_nHeight(0), m_nWidth(0)   // colon initialization.
{   // Data members are set before the c'tor body runs.
    cout << "Rectangle c'tor.\n";
}
Rectangle::~Rectangle()
{
    cout << "Rectangle d'tor.\n";
}

void Rectangle::SetHeight(int h)
{
    m_nHeight = h;
}

void Rectangle::SetWidth(int w)
{
    m_nWidth = w;
}

long Rectangle::GetVolume(void)
{
    return (long)m_nWidth * m_nHeight;
}

```

(continued)

```
/*** ************** Small Test Program ******************/
int main()
{
    Rectangle r1;         // The contructor assigns values
                          // to avoid undefined contents
    cout << "The initial volume of rectangle r1 is "
         << r1.GetVolume() << endl;
    r1.SetHeight(15);     // Set attributes for r1
    r1.SetWidth(10);
    cout << "The set volume of rectangle r1 is "
         << r1.GetVolume() << endl;
    return 0;
}
```

Lab 6: Creating Classes and Member Functions

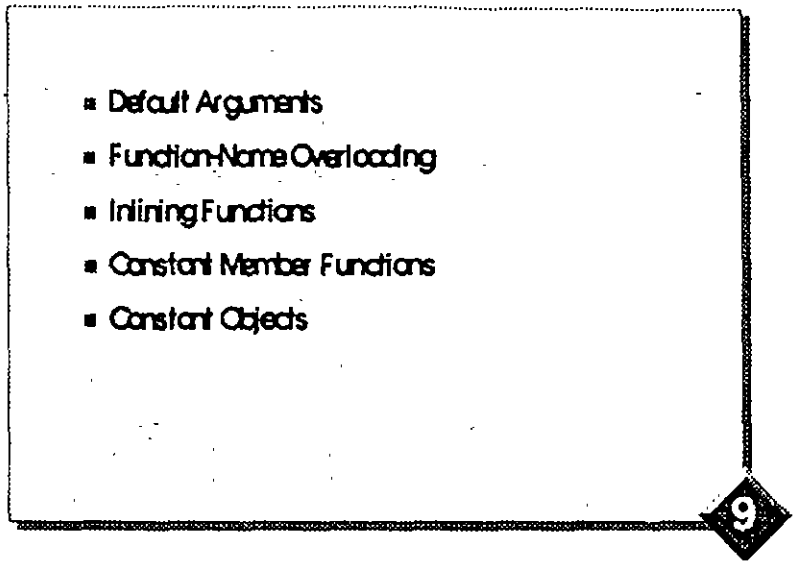
Slide Objective
Execute the lab solution.
Explain the purpose of the lab.
Ask the students to read the scenario.



Module 9: Tuning Member and Global Functions

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Default Arguments
- Function-Name Overloading
- Inlining Functions
- Constant Member Functions
- Constant Objects

Module Summary

In the last module you created a simple class—the most important thing you've done so far. In this module you will explore ways to add efficiency to your class's member functions.

You will be introduced to some new class features that will allow you to reduce the number of instructions a PC executes to employ your functions. You will also be streamlining the way in which arguments are passed.

Though these concepts are not direct building blocks for following modules, they will nonetheless be important as you return to the workplace and use these new coding skills.

Objectives

Upon completion of this module, you will be able to:

- Use default arguments.
- Overload function names.
- Create inline function bodies.
- Create constant member functions and constant objects.

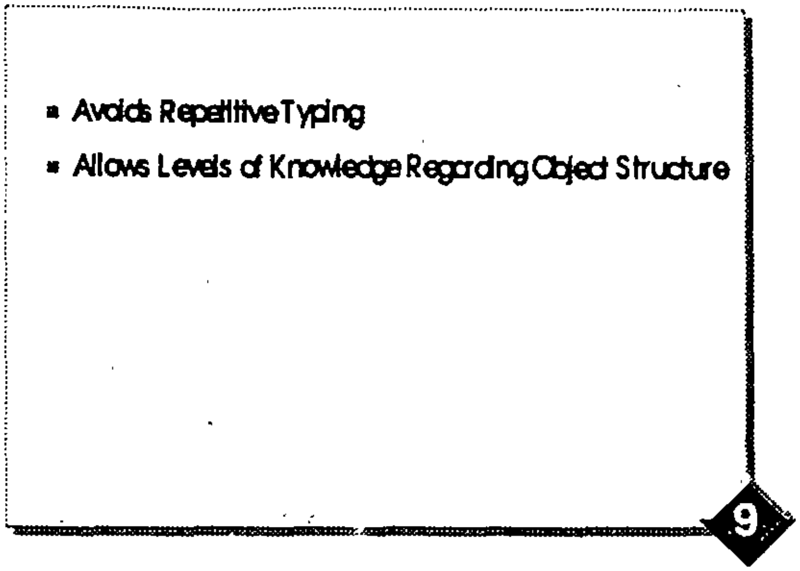
For Your Information
The lab for this module builds upon the previous solution. It's not pretty, but you may want to execute the lab solution here to show students where we're going.

Lab

Tuning Your Member Functions

Default Arguments

Slide Objective
Define the uses for default arguments.



- Avoids Repetitive Typing
- Allows Levels of Knowledge Regarding Object Structure

Key Point
Default arguments simplify programming for the class users, those programmers that are using a well-defined class.

Many functions that take multiple actual arguments may have default values for one to all parameters. A function that accepts Month, Day, and Year arguments would expect to be called hundreds of times with the same year value. A function to open files might expect various filenames, but most text files will probably be opened in read-write mode.

Functions may specify a default value for one or more arguments using a special assignment syntax within the signature. Always beginning with the rightmost argument, the default value is specified following an equal sign. In a prototype, it might appear like this:

```
void funcB( int, char, int = 94 );
```

Key Point
Defaults are specified in the prototype! Never in the formal definition.

Default arguments are specified in the prototype rather than in the function definition.

```
void funcB( int nC, char chA, int nD = 94 );
```

Typically, you will be creating header files for your classes and prototypes. Given the preceding prototype example, a source file that includes that function declaration could extend default values for that function as long as the function has not yet been defined.

Delivery Tip
Defining additional default argument(s) for a function is an advanced topic. Rules: Never redefine. Always right to left.

Given the following header file,

```
void funcB( int, char, int = 94 );
```

a source file that intends to use function **funcB** in a specific manner may redeclare the function as

```
void funcB( int, char = 't'  int);
```

Important Using the rule of rightmost definition first, the third argument was assigned a default value of 94. It is illegal to redefine that assignment (or to respecify the same value). The third argument retains the original assignment and the second argument gains the default.

Demo

DEFAULT.CPP is found in \DEMOS\MOD09.

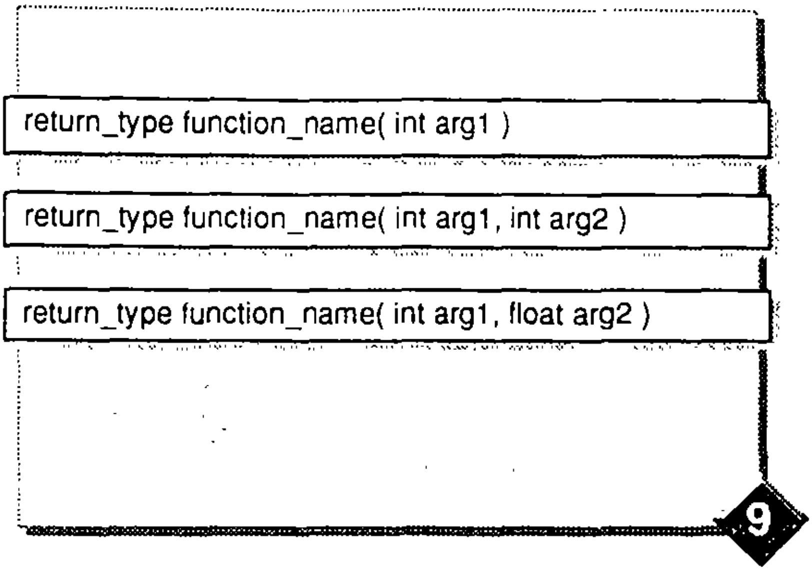
```
// DEFAULT.CPP    found in \demos\mod09
// Functions that define default values for selected
// arguments streamline the interface and allow
// class users multiple variations
#include <iostream.h>

/************ Rectangle Class Declaration ***************/
class Rectangle
{
public:
    // This c'tor is equivalent to three c'tors
    Rectangle(int h, int w, int x=0, int y=0);
    ~Rectangle();
    void SetCenter(int, int);
    void Size(int, int);
    void Draw();
private:
    int m_x, m_y;
    int m_nHeight, m_nWidth;
};

/********* Rectangle Member Function Definitions ********/
Rectangle::Rectangle(int h, int w, int x, int y)
    : m_nHeight (h), m_nWidth (w), m_x (x), m_y (y)
{
    cout << "Rect c'tor\n";
}

Rectangle::~Rectangle()
{
    cout << "Rect d'tor\n";
}

void Rectangle::SetCenter(int x, int y)
{
    m_x = x;
    m_y = y;
}

void Rectangle::Size(int nh, int nw)
{
    m_nHeight = nh;
    m_nWidth = nw;
}

void Rectangle::Draw(void)
{            // Currently just a display function
    cout << "Rectangle at x:" << m_x << " y:" << m_y;
    cout << " height:" << m_nHeight << " width:" <<
m_nWidth;
}

```

(continued)

```
/**************** Small Test Function *******************/
int main()
{
    Rectangle r1 (1, 2),          // default x and y as 0
              r2 (5, 6, 8),             // default y as 0
              r3 (10, 10, 100, 100);        // no defaults

//  Rectangle r4;                // Error: no default c'tor
//  Rectangle r5 (9, 9, , 40);   // Error: improper syntax

    cout << "Displaying r1:\n";
    r1.Draw();
    cout << endl;
    r1.Size(11, 12);
    r1.SetCenter(-10, -10);
    cout << "Displaying r1 after manipulation:\n";
    r1.Draw();
    cout << endl;

    cout << "Displaying r2:\n";
    r2.Draw();
    cout << endl;

    cout << "Displaying r3:\n";
    r3.Draw();
    cout << endl;
    return 0;
}
```

Function-Name Overloading

Slide Objective
Explain function name overloading as "a variation on argument type or number." Note: Expect to contrast between default arguments.



```
return_type function_name( int arg1 )

return_type function_name( int arg1, int arg2 )

return_type function_name( int arg1, float arg2 )
```

Key Points
Overloaded functic-- may differ o:
# of arguments and data type of args.

Not due to function return-type.

Features

Function overloading occurs when there are two or more functions in the same scope that have the same name. C++ allows this when the prototypes differ in the number and/or types of arguments. (Function-name overloading may vary by *constness.* This topic will be deferred until later.) Overloading is made possible by function-name encoding (also known as name-decoration or name-mangling).

Overloaded functions cannot differ on **return** type only. The compiler knows how to generate promotion and truncation of **return** values, so variations on just **return** type would be ambiguous.

Function-name encoding is implemented by appending class-name and argument-type information. The encoding scheme is implementation-dependent.

Although any global functions can also be overloaded, multiple constructors are the most common example of function-name overloading.

Reference

Refer to "Overloading," in the *C++ Language Reference*.

Demo

OVERLOAD.CPP is located in \DEMOS\MOD09.

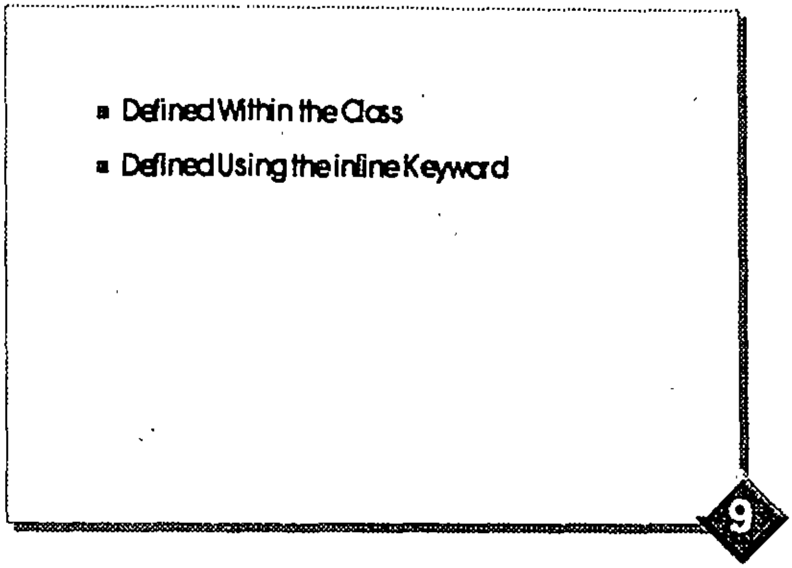
```
// OVERLOAD.CPP  found in \demos\mod09
// Functions with the same name and different argument
// data-types and/or argument counts are overloaded.
#include <iostream.h>

/************* Rectangle Class Declaration **************/
class Rectangle
{
public:
        // The following c'tors are overloaded
    Rectangle();
    Rectangle(int h, int w, int x=0, int y=0);
    ~Rectangle();
    void SetCenter(int, int);
    void Size(int,int);
    void Draw(void);
private:
    int m_x, m_y;
    int m_nHeight, m_nWidth;
};

/******** Rectangle Member Function Definitions *********/
Rectangle::Rectangle()
    : m_nHeight(0), m_nWidth(0), m_x(0), m_y(0)
{
    cout << "Rect default c'tor\n";
}

Rectangle::Rectangle(int h, int w, int x, int y)
    : m_nHeight(h), m_nWidth(w), m_x(x), m_y(y)
{
    cout << "Rect(int,int,int,int) c'tor\n";
}

Rectangle::~Rectangle()
{
    cout << "Rect d'tor\n";
}
```

(continued)

```
void Rectangle::SetCenter(int x, int y)
{
    m_x = x;
    m_y = y;
}

void Rectangle::Size(int nh, int nw)
{
    m_nHeight = nh;
    m_nWidth = nw;
}

// Currently just a display function
void Rectangle::Draw(void)
{
    cout << "Rectangle at x:" << m_x << " y:" << m_y;
    cout << " height:" << m_nHeight << " width:" <<
m_nWidth;
}

/*************** Small Test Function ********************/
                         // function prototypes
void Goodbye(int x = 1); // Goodbye with default, int arg
void Goodbye(Rectangle); // Goodbye with Rectangle arg

int main()               // Cannot overload main function!
{
    Rectangle r1 (1, 2),
              r2 (5, 6, 8),
              r3 (10, 10, 100, 100);
    Rectangle r4;        // legal with default c'tor

    cout << "Displaying r1:\n";
    r1.Draw();
    cout << "\nDisplaying r2:\n";
    r2.Draw();
    cout << "\nDisplaying r3:\n";
    r3.Draw();
    cout << "\nDisplaying r4:\n";
    r4.Draw();
    cout << endl;
    Goodbye();
    //Note destruction of temporary Rectangle object
    Goodbye(r4);
    cout << endl;
    return 0;
}

void Goodbye(int x)
{
    cout << "Hello from Goodbye(int x = "
         << x << "\n";
}

void Goodbye(Rectangle r)
{
    cout << "Hello from Goodbye(Rectangle)\n";
}
```

Inlining Functions

Slide Objective
Explain the benefits of inlining functions. The syntax is covered in the Demo program.



- Defined Within the Class
- Defined Using the inline Keyword

Inline Member Functions

It has already been established that manifest constants can be useful to the document values your program uses. The compiler would substitute the value specified in the `#define` line before generating code. The second use of the `#define` is to create a code fragment (typically an equation) called a macro. Although macros add to program readability and are treated like **inline** functions, the arguments to a macro do not benefit from type-checking, and therefore suffer side effects.

The **inline** keyword is a suggestion to the compiler that the body of the following function should be substituted at the location where the function is invoked. A function can be labeled as **inline** in either its definition or declaration. The **inline** and **static** keywords have similar effects on a function's visibility—both limit linkage to the local file or class (translation unit). Also, the compiler needs the C++ code of an **inline** function to expand a call to it. Therefore, **inline** functions that are used in multiple files should be defined in .H files.

Delivery Tip
Remind students of the overhead associated with a function call (recall the graphic depicting the stack frame for the SWAP program).

Inline functions avoid the overhead associated with a function call. Data hidden through private keywords, but accessible through **Get** functions, is readily available. The tradeoff is repeating the function body within program code. This can increase code size.

A class member function may be implicitly defined as **inline** by including the body of the function within the class. Accessor functions, such as the **Get** and **Set** members discussed in the class module are good candidates for **inline** functions. A good rule is short functions of five statements or less.

Demos

IMPLICIT.CPP is located in \DEMOS\MOD09. It demonstrates a member function defined within a class.

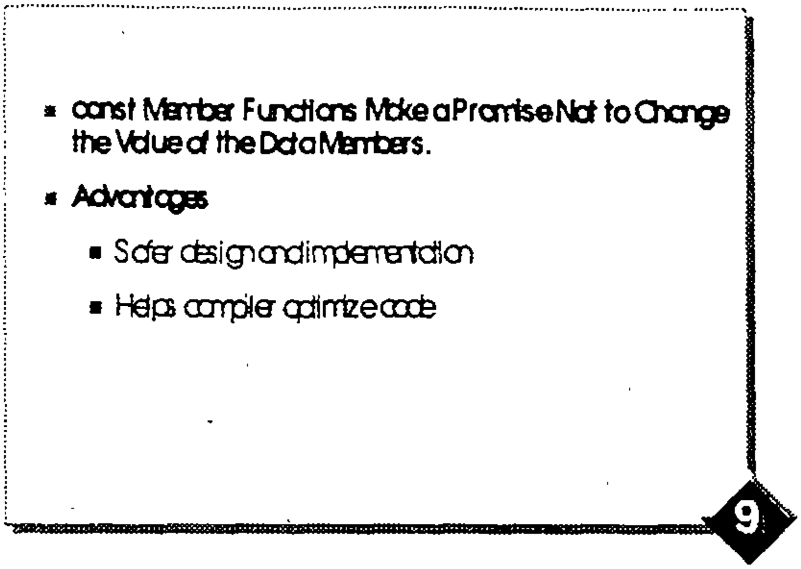
```
// IMPLICIT.CPP  found in  \demos\mod09
// Implicitly "inline" functions have the function body
// defined within the class definition.
#include <iostream.h>

/************** Money Class Definition ******************/
class Money
{
public:
    Money(long lD, int nC)
        : lDollars (lD), nCents (nC)
    { }
    void Display() { cout << "$" << lDollars << "." <<
nCents; }
private:
    long lDollars;
    int nCents;
};

/*************** Small Test Function *******************/
int main()
{
    Money PocketChange (1, 50);
    Money MoneyClip (12, 0);
    PocketChange.Display();
    cout << endl;
    MoneyClip.Display();
    cout << endl;
    return 0;
}
```

EXPLICIT.CPP is located in \DEMOS\MOD09. It demonstrates inline implementation of a class member function.

```
// EXPLICIT.CPP   found in \demos\mod09
// Using the "inline" keyword, functions are suggested
// for inlining regardless of the location of body.
#include <iostream.h>

/*************** Money Class Definition *****************/
class Money
{
public:
    inline Money(long lD, int nC);
    inline void Display();
private:
    signed long m_lDollars;
    int m_nCents;
};

/************ Money Class Member Functions **************/
Money::Money(long lD, int nC)
    : m_lDollars (lD), m_nCents (nC)
{ }

void Money::Display()
{
    cout << "$" << m_lDollars << "." << m_nCents;
}

/**************** Small Test Function *******************/
int main()
{
    Money PocketChange (1, 50);
    Money MoneyClip (12, 0);
    PocketChange.Display();
    cout << endl;
    MoneyClip.Display();
    cout << endl;
    return 0;
}
```

Constant Member Functions

Slide Objective
Inlining may add efficiencies to the program code; "const" member functions may also.



- **const Member Functions Make a Promise Not to Change the Value of the Data Members.**
- **Advantages**
  - Safer design and implementation
  - Helps compiler optimize code

Member functions often do not change any of the values of the data members; that is, they do not change the state of the current object. For example, you have seen this constant behavior in accessor and display member functions. C++ supports this concept by marking a member function as **const** in both its declaration and definition:

```
class Rectangle {
public:
void Display(void) const;
. . .
};

void Rectangle::Display(void) const
{
    . . .
```

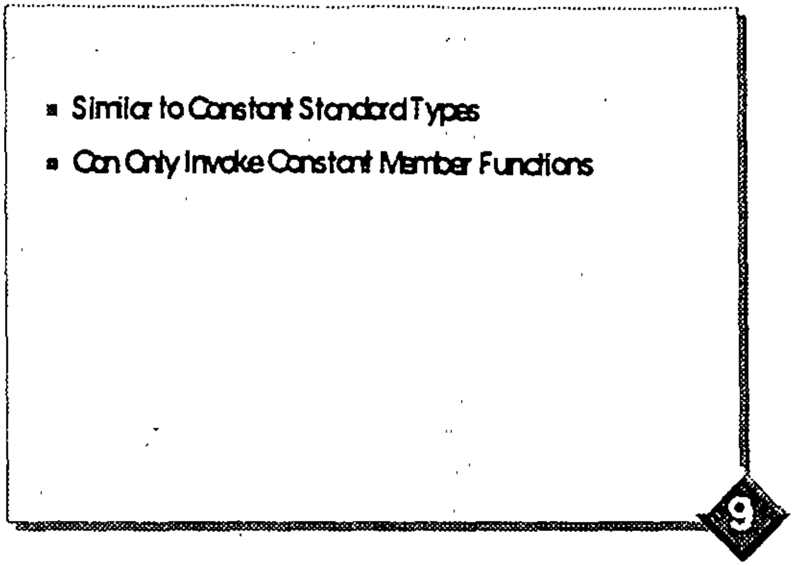
Key Points
Func doesn't change data.
Func doesn't call another member func to change data.
Func is not c'tor or d'tor.

Now if **Display** tries to change one of the data members, the compiler will issue an error. The compiler also tracks calls that **Display** makes, even disallowing **Display** to indirectly change a data member. *Therefore, a* **const** *member function cannot call non-***const** *member functions within the same class.*

Constructors and destructors should not be labeled **const**.

Constant Objects

Slide Objective
"const" may also be used as a type-modifier in the declaration of an object.
Rule: Object must be initialized at declaration.



- Similar to Constant Standard Types
- Can Only Invoke Constant Member Functions

Constant objects can be created:

```
const Rectangle rectunit(1,1,0,0);
```

When a constant object is created, it must be assigned correct values by invoking the logically proper constructor. After creation, a constant object may not be changed. According to this rule, both of the following statements are illegal:

```
rectunit = rectl;          // error!
rectunit.SetWidth(10);     // error!
```

Only constant member functions may be invoked for a **const** object. Assuming that **Display** is now constant, you could code as follows:

```
rectunit.Display();     // okay
```

This introduces a third reason to use constant member functions: to allow class users to create and properly manipulate constant objects of that type.

Demo

CONST.CPP is found in \DEMOS\MOD09.

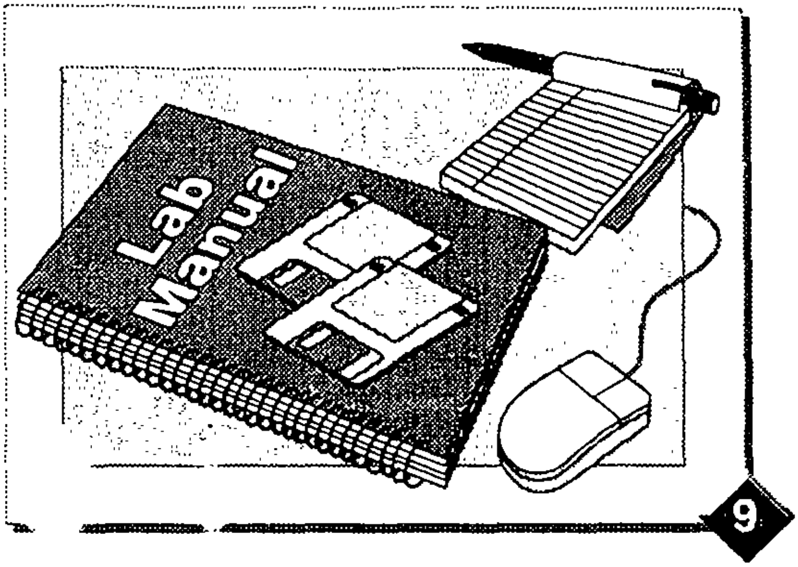
```
// CONST.CPP    found in \demos\mod09
// Demonstrates const member functions and
// const Rectangle objects.
#include <iostream.h>

/************ Rectangle Class Declaration ***************/
class Rectangle
{
public:                     // construction
    Rectangle(int h, int w, int x=0, int y=0);
    ~Rectangle();
                            // operations
    void SetCenter(int, int);
    void Size(int, int);
    void Draw() const;      // "const" member function
private:                    // implementation
    int m_x, m_y;
    int m_nHeight, m_nWidth;
};

/********* Rectangle Member Function Definitions ********/
Rectangle::Rectangle(int h, int w, int x, int y)
    : m_nHeight (h), m_nWidth (w), m_x (x), m_y (y)
{
    cout << "Rect c'tor\n";
}

Rectangle::~Rectangle()
{
    cout << "Rect d'tor\n";
}

void Rectangle::SetCenter(int x, int y)
{
    m_x = x;
    m_y = y;
}

void Rectangle::Size(int nh, int nw)
{
    m_nHeight = nh;
    m_nWidth = nw;
}

         // Function definition must also be "const"!
void Rectangle::Draw(void) const
{
 // m_nHeight = 0;   //illegal
 // SetCenter (0,0); //illegal
    cout << "Rectangle at x:" << m_x << " y:" << m_y;
    cout << " height:" << m_nHeight
         << " width:" << m_nWidth;
}

```

(continued)

```
/**************** Small Test Function *******************/
int main()
{                        // modifiable object
    Rectangle r1 (1, 2, 3, 4);
                         // constant objects
    const Rectangle rc1 (10, 10), rc2 (r1);
    cout << "\nDisplaying rc1:\n";
    rc1.Draw();
    cout << endl;
    cout << "Displaying rc2:\n";
    rc2.Draw();
    cout << "\n\n";

    r1 = rc2;            // ok to modify r1
 // rc2 = r1;            // error: using rc2 as lvalue
 // rc1.Size (20, 20);   // error: const arg mismatch
    return 0;
}
```

Lab 7: Tuning Your Member Functions

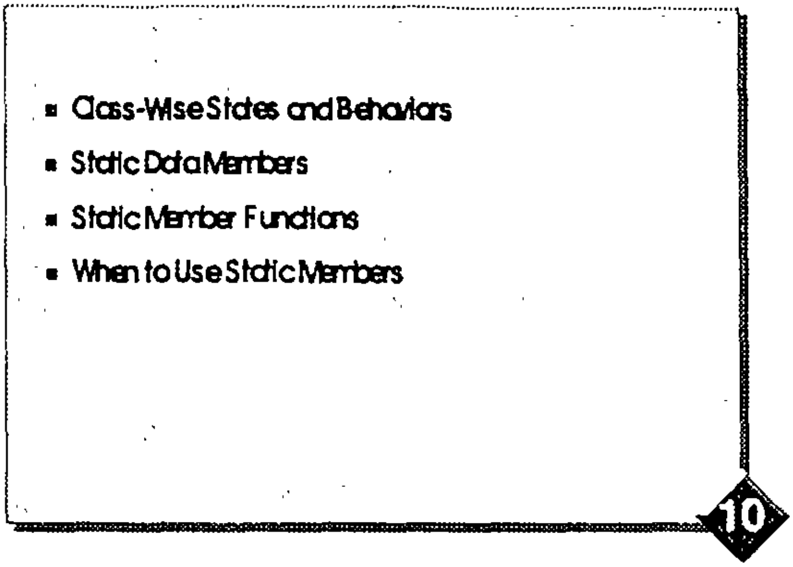
Slide Objective
Execute the lab solution.
Explain the purpose of the lab.
Ask students to read the scenario.



Module 10: Static Members

Σ Overview

Slide Objective
Provide an overview of the module contents.
It's recommended that you run the lab solution prior to delivering the module. The topical area is the same (Date), but the program automatically determines today's date.



- Class-Wise States and Behaviors
- Static Data Members
- Static Member Functions
- When to Use Static Members

Delivery Tip
Static members were not discussed in the OOAD modules. They can be viewed as either 1) representing invariant attributes and behavior for all class objects, or 2) alternately as attributes and behaviors of entire classes. (That extends the class as a limited actor in its own right.)

Module Summary

A **static** member supports the concept of class-wise or object-invariant behaviors or states. When used properly, static members help create more robust and efficient class implementations. They eliminate unnecessary duplication in every object, while still ensuring proper encapsulation.

Objectives

Upon completion of this module, you will be able to:

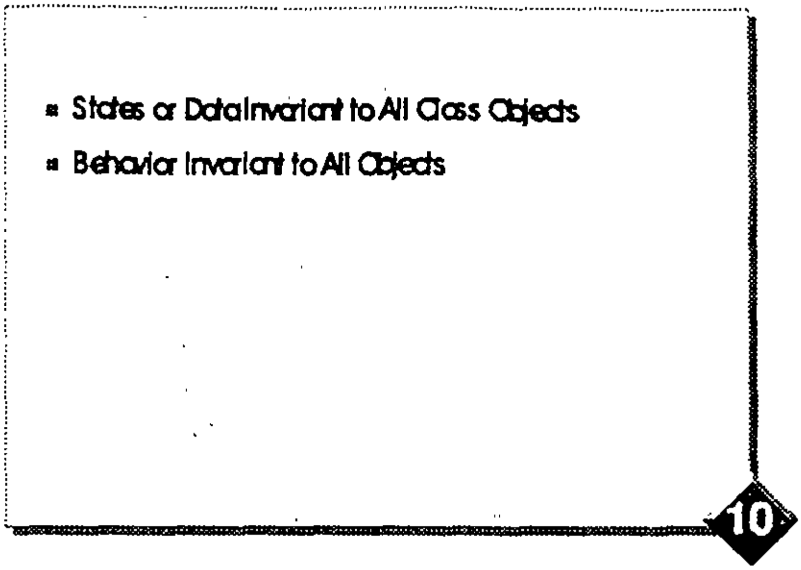
- Create and initialize static data members;
- Create and invoke static member functions;
- Understand the limitations and benefits of static members;

Lab

Using Static Data and Members

Class-Wise States and Behaviors

Slide Objective
Define the purpose, features and benefits of the "static" type-modifier for the class from 1) data member and 2) member function viewpoints.



- States or Data Invariant to All Class Objects
- Behavior Invariant to All Objects

The **static** keyword may be used with a local variable to implement persistence of an assigned value, or used with a global variable to hide the variable from functions in other source files. Similar use with a **static** global function sets the function's visibility to be callable only from other functions in the same source file. Within C++ classes, the **static** keyword may be used to modify the attributes of either a data member of a member function.

Key Point
From class view

The **static** attribute indicates that a member generally acts at the class level and is not different for each object of that class's type.

Key Point
From member data view

Sometimes a class will have an attribute that must have the same value for all of its objects. For example, a Character class might have an ASCII/EBCDIC/ Unicode translation table. Although it is possible to allocate a new instance of this table for each Character object created, it would be very inefficient to do so. Such a table would be a prime candidate for becoming a **static** data member. As such, only one copy is created for the entire class.

Key Point
From member function view

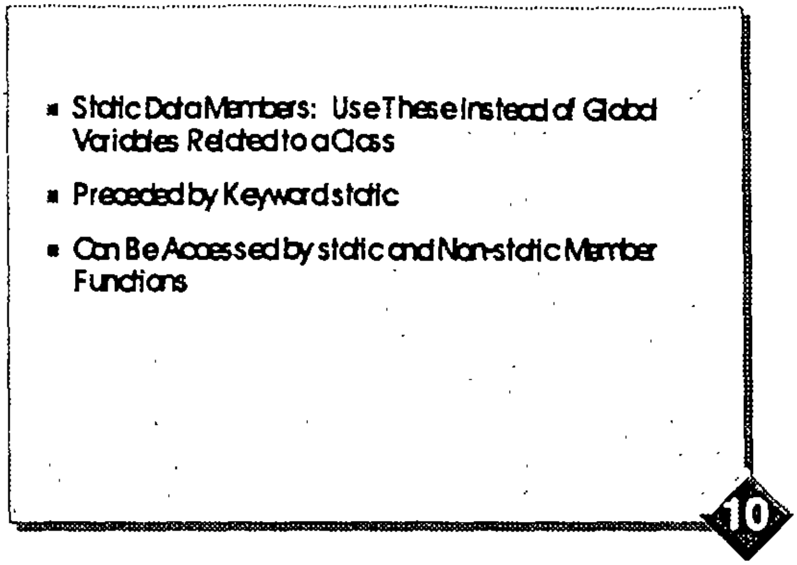
Member functions can also be **static**. These functions do not manipulate any of the object's data members—rather, they act at the class level, often manipulating **static** data member(s). For example, an **ASCIItoEBCDIC** function would probably be **static**. **Static** member functions are also often used to perform high-level actions connected with a class.

Our Screen class also contains **static** members. If we assume that although there may be multiple logical display spaces there will be just one actual hardware monitor displaying the objects, then the members concerned with the monitor will be **static** because there is just one-per-class instance of it.

Tip Do not confuse **static** members with constant members.

Static Data Members

Slide Objective
Concentrate on static data member needs: "static" keyword, initialization, and access.



- Static Data Members: Use These Instead of Global Variables Related to a Class
- Preceded by Keyword static
- Can Be Accessed by static and Non-static Member Functions

Key Points
Use "static" keyword when defining the data member. Initialize variable at file scope to some benign value (outside any class or func definition.)

Delivery Tip
The initialization syntax does not actually break the private access of the class member.

Static data members can be an improvement over global variables. A static data member has the same lifetime as a global variable (the entire program) and there is, only one instance of the variable—but its use is restricted to (encapsulated in) the class.

Static data members are declared by prepending their declaration with the keyword **static** as in:

```
static int bVidState;
```

Both non-static and static member functions can access static data members.

Each static data member must be initialized once and only once before the **main** function, for example:

```
int Screen::bVidState = OFF;
```

The static keyword must not be repeated in the initialization statement. The initialization statement must be outside the class definition and at file scope. It causes the storage space to be allocated.

Demo

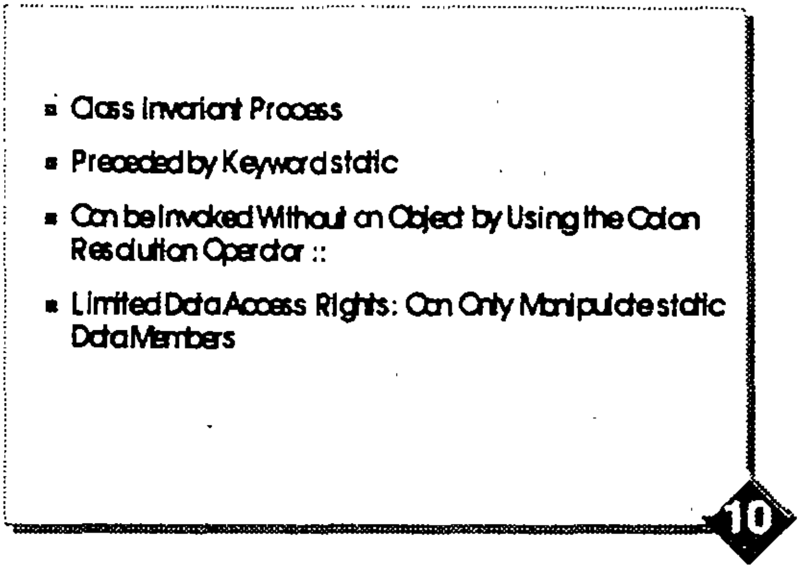
STATIC1.CPP is found in \DEMOS\MOD10.

```
// STATIC1.CPP   found in \demo\mod10
// Demonstrates use of static data member.  Note: fgc is
// ForeGround Color, brc is BackGround Color.
#include <iostream.h>

#define BLACK   1
#define WHITE   2
#define RED     4
#define GREEN   8
#define BLUE    16

#define ON      1
#define OFF     0

/********************** Screen Class *********************
    Maps the logical display space onto the video
    monitor.  The class allows multiple logical screen
    objects to be created.  It only supports one
    physical video monitor through static members.
*********************************************************/
class Screen
{
public:                     // construction
    Screen(short fgc=WHITE, short brc=BLACK)
        : m_FGC(fgc), m_BRC(brc)
        {;}
    void Graphics(int bstate)
    {
        bVidState = bstate;
    }
    int Update(void);       // implementation
private: // one instance of static data shared by objects
    static int bVidState; // video OFF=0, ON=1
    short m_BRC;          // background color
    short m_FGC;          // foreground color
};

/************** Screen Member Functions *****************/
int Screen::Update(void)
{
    if (bVidState == OFF)
    {
        cerr << "Error: monitor is not in video mode.";
        return 0;
    }
    cout << "Monitor updated: FGC is "
         << m_FGC << ", BRC is " << m_BRC << "\n";
    return 1;
}
    // NOTE: Static data members must be initialized to a
    // value at file scope prior to any execution.
int Screen::bVidState = OFF;  // Assume initial state: OFF

(continued)
```

```
/*************** Small Test Function *********************/
int main()
{

    Screen s1(BLUE);
    s1.Update();         // fails because mode is OFF
    cout << endl;
    s1.Graphics(ON);
    s1.Update();         // succeeds now
    return 0;
}
```

Static Member Functions

Slide Objective
Rhetorically: How could static members help initialize the screen? Answer: "static" member functions have special properties.



- Class Invariant Process
- Preceded by Keyword static
- Can be Invoked Without an Object by Using the Colon Resolution Operator ::
- Limited Data Access Rights: Can Only Manipulate static Data Members

Static member functions can be an improvement over global (non member) functions. A **static** member function can be invoked in the absence of an object, but it is still encapsulated within a class.

Static member functions are declared by prepending their declaration (but not the definition) with the keyword **static**, as in:

```
static int InitVideo(void);
```

Access to a **static** member function can be achieved through two mechanisms:

Key Points
Static member functions may be invoked by 1) an object using the "." dot operator, or 2) the class using the "::" scope resolution operator (regardless of whether any objects exist.)

1. Using the standard dot operator on an object:

```
s1.InitVideo();
```

2. Using the class name and the colon resolution operator:

```
Screen::InitVideo();
```

Static member functions may be invoked, even if there is no current object of that class, by using the class name and :: operator.

However, **static** member functions are limited in that they cannot access non-**static** member data. That is because this information is contained within objects, and **static** member functions work at the class level. Therefore, most programmers prefer to use the class name and :: operator syntax, because it is more suggestive.

Demo

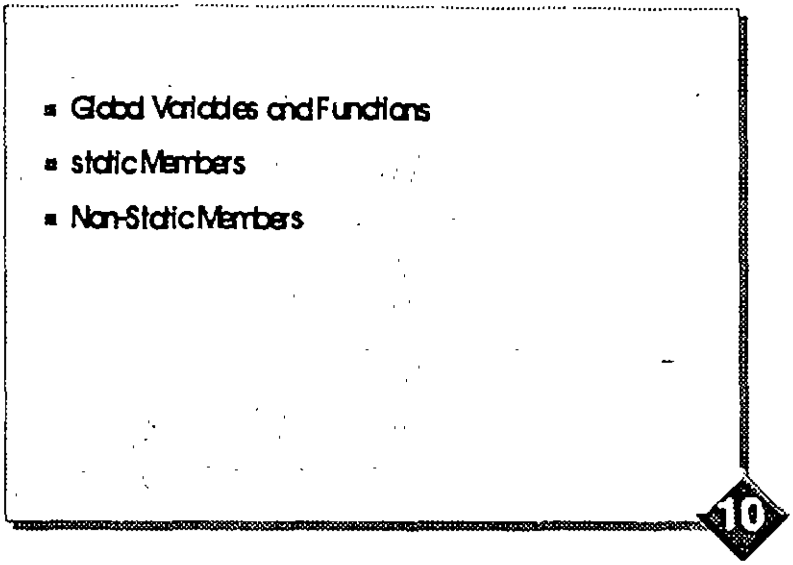
STATIC2.CPP is found in \DEMO\MOD10.

```
// STATIC2.CPP   Found in \demo\mod10
// Demonstrates use of static data and function. Note:
// fgc is ForeGround Color, brc is BackGround Color.

#include <iostream.h>

#define BLACK   1
#define WHITE   2
#define RED     4
#define GREEN   8
#define BLUE    16

#define ON      1
#define OFF     0
#define TRUE    1
#define FALSE   0

/********************* Screen Class **********************
    Maps the logical display space onto the video
    monitor.  The class allows multiple logical screen
    objects to be created.  It only supports one
    physical video monitor through static members.
***********************************************************/
class Screen
{
public:                      // construction
    Screen(short fgc=WHITE, short brc=BLACK)
        : m_FGC(fgc), m_BRC(brc)
        {;}
    void Graphics(int bstate)
    {
        bVidState = bstate;
    }
    int Update(void);        // implementation
         // "static" member function has normal scope
    static int InitVideo(void);
private: // one instance of static data shared by objects
    static int bVidState; // video OFF=0, ON=1
    short m_BRC;             // background color
    short m_FGC;             // foreground color
};

```

(continued)

```
/************** Screen Member Functions *****************/
int Screen::Update(void)
{
    if (bVidState == OFF)
    {
        cerr << "Error: monitor is not in video mode.\n";
        return 0;
    }
    cout << "Monitor updated: FGC is "
         << m_FGC << ", BRC is " << m_BRC << endl;
    return 1;
}
                         // static member function
int Screen::InitVideo(void)
{
    int success = TRUE;
    cout << "(Re)Initializing Monitor: ";
 //
 // Magic here: try to initialize monitor to graphics mode.
 //
    if (success)
    {
        cout << "succeeded.\n";
     // cout << " in BR color " << m_BRC;  // Illegal:
     // attempting to display member data before any
     // object exists!  Typically static funcs only modify
     // static data!
        bVidState = ON;  // Only "static" data may be set.
        return TRUE;
    }
    cout << "failed.\n";
    return FALSE;
}
     // NOTE: Static data members must be initialized to a
     // value at file scope prior to any execution.
int Screen::bVidState = OFF;   // Assume initial state: OFF

/*************** Small Test Function ********************/
int main()
{                        // Static function may be accessed
    Screen::InitVideo(); // without an object (using ::)

    Screen s1 (BLUE);
    s1.InitVideo();      // access via object, success
    s1.Graphics(ON);
    s1.Update();
    return 0;
}
```

When to Use Static Members



- Global Variables and Functions
- static Members
- Non-Static Members

When you want to access information or implement a behavior with respect to an object or a class, you really have three choices: global functions and variables, **static** class members, and non-**static** class members.

Global variables and functions should be used when information or processes must be shared throughout an entire program, but they do not logically belong in any of the recognized classes. Remember two points: 1) that the number of global variables should be kept at a minimum, and 2) as a program develops, new candidate classes are often discovered.

Non-**static** members represent the state of each object and the behaviors that affect those states.

Static members represent class invariant states and processes that affect those invariant states. Sometimes, **static** member functions also perform global actions not directly affecting **static** data members. We can see that **static** members represent a nice middle ground between standard members and globals.

Note that each global and member function can also contain local variables that are encapsulated within that function. These variables are important when implementing a function, but like data members, they should be mostly invisible to the user.

Lab 8: Using Static Data and Members

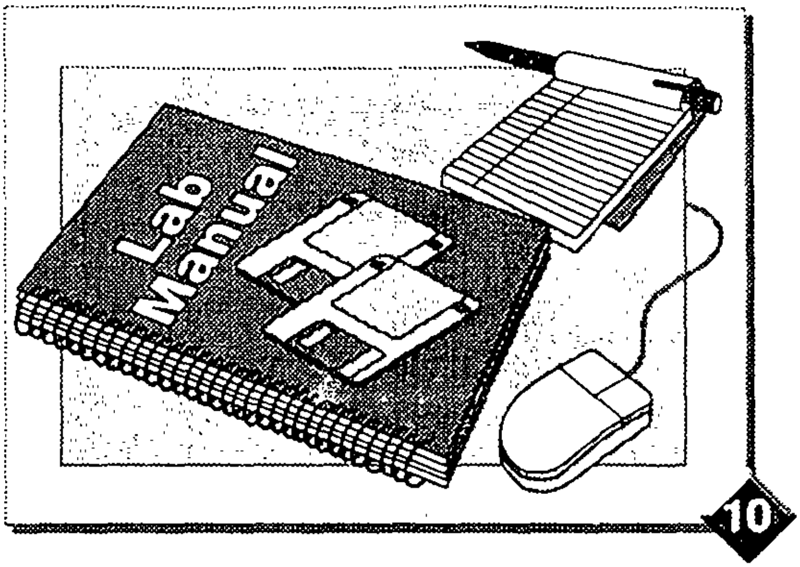
Slide Objective

Execute the lab solution.

Explain the purpose of the lab.

Explain MFC AFX.H (see Del Tip).

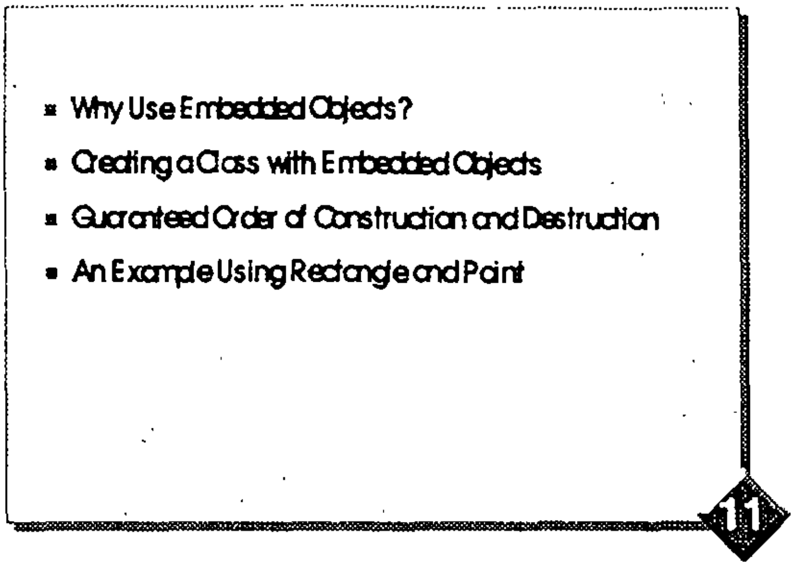
Ask the students to read the scenario.



Module 11: Embedded Objects

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Why Use Embedded Objects?
- Creating a Class with Embedded Objects
- Guaranteed Order of Construction and Destruction
- An Example Using Rectangle and Point

11

Module Summary

In the last two modules, you created and performed some optimization on simple classes. In this module, you will learn how to create classes that contain objects or instances of other classes.

Embedding objects is an important technique for extending your class. In effect, you use code that other programmers have written. Remember, code reuse is an important reason why you are making the shift to OO programming in the first place.

The mechanism for embedding an object is straightforward. In the surrounding class's declaration, simply declare an object of another class as a data member. The C++ language guarantees that the embedded objects within a class will be constructed and destroyed at the appropriate times.

In this module, you will transform the simple Rectangle class to contain a Point object that is a center point.

You will use embedded objects throughout the rest of this course.

Objectives

Upon completion of the module, you will be able to:

Key Points
Explain the module objectives in OOD terms.
Execute the lab solution to show a problem domain.
Sight examples: Inventory "contains a" PartID.
A Sales Order "contains" Inventory.

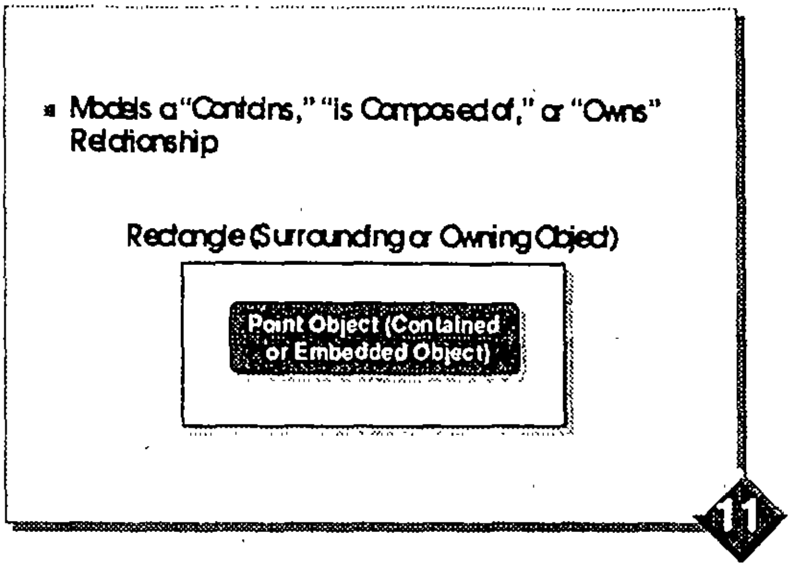
- Add an object of a different class as a data member of a new class.
- Test your class by creating a program to instantiate objects.

Lab

Containment and Embedded Objects

Why Use Embedded Objects?

Slide Objective
Provide an overview of the features and benefits of using containment.



- Models a "Contains," "Is Composed of," or "Owns" Relationship

Rectangle (Surrounding or Owning Object)

Remember from the first two modules on OOAD that containment or embedding represents a "contains," "is composed of," or "owns" relationship. In this example, every rectangle contains a center point.

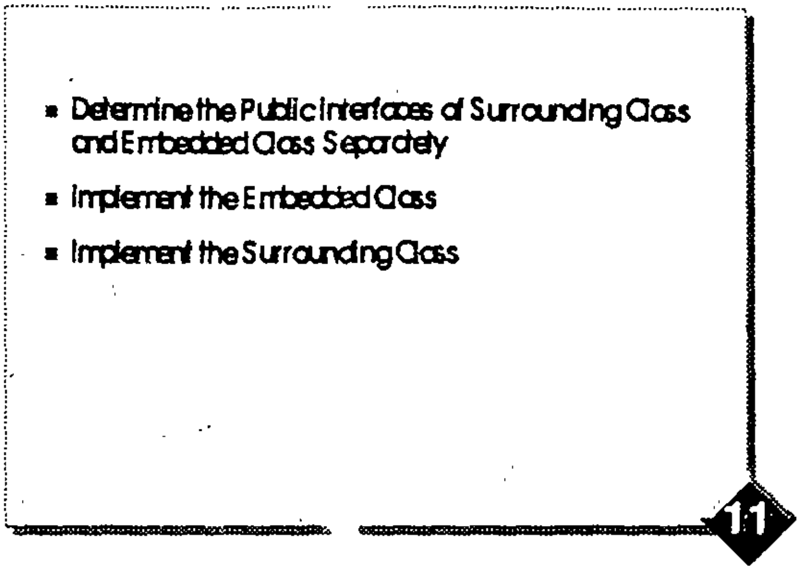
It is important to contrast containment with inheritance; the latter implies a "is a type of" relationship. Inheritance will be discussed in the next module.

Class relationships are initially determined during the A/D phase. During this phase, it may be noticed that some more complicated classes actually are composed of other logical entities—an assembly, so to speak. These component portions may be rich enough in their own right to deserve being modeled by classes. This is especially true if the components will be reused or replaced in future projects.

Since embedded objects are data members, they normally have private access specification. Because of this, users of a class with embedded objects in it may be unaware of that fact because they only use the public interface for the surrounding class. For example, as a user of the Rectangle class, you may not be able to tell (without looking at the class source code) if the location of a rectangle is implemented as a center point, as center *x* and *y* coordinates stored as integers, or as a pair of upper-right/lower-left coordinates. Nor should you care.

Creating a Class with Embedded Objects

Slide Objective
Propose the following high-level steps to implement classes where a surrounding class "contains" objects of another class.



- Determine the Public Interfaces of Surrounding Class and Embedded Class Separately
- Implement the Embedded Class
- Implement the Surrounding Class

Key Point
Make an effort to create a full, useful interface for both classes.

After the need for an embedded object has been determined, the next step is to specify the required interface for its class. Since it is embedded, that interface is largely determined by the surrounding class. But since an embedded object may have future use in other projects, some effort should be made to implement it as a complete, self-supporting class.

The surrounding class's interface must also be fleshed out. After these two interfaces have been specified, it should become apparent if the original containment relationship is still valid.

Next, separately implement both classes to at least initial level:

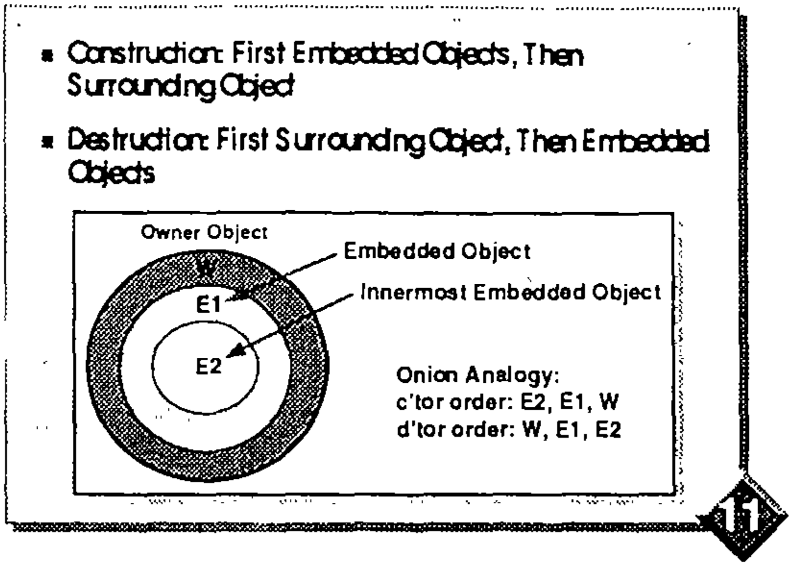
- Create stub member functions.
- Embed an object into the surrounding class.
- Make initial connections between the containing class's member functions and the embedded object.
- Test implementation.

Key Point
Only the surrounding class "knows" it contains another object.

Typically the communication between them will be one-way from the surrounding class to the embedded object.

Guaranteed Order of Construction and Destruction

Slide Objective
The compiler automatically handles c'tor and d'tor execution in the order depicted



- Construction: First Embedded Objects, Then Surrounding Object
- Destruction: First Surrounding Object, Then Embedded Objects

Onion Analogy:
c'tor order: E2, E1, W
d'tor order: W, E1, E2

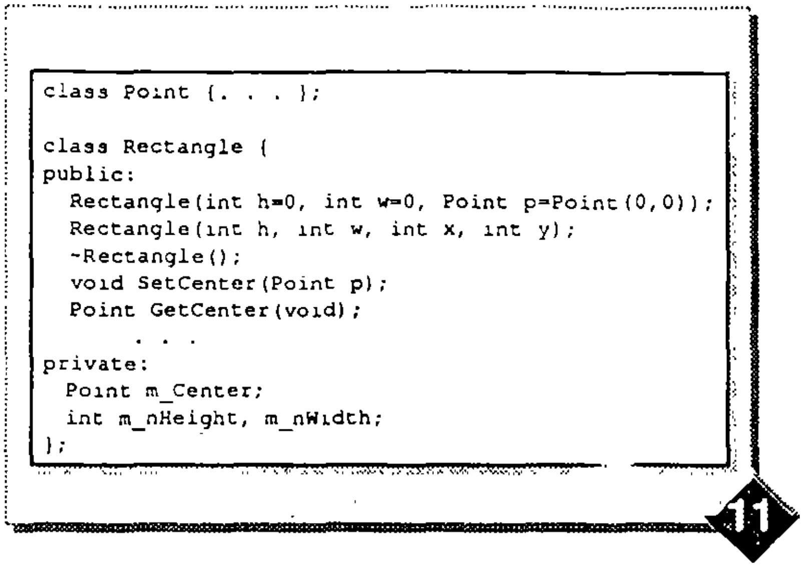
Delivery Tips
Present the c'tor/d'tor process as easy, effortless, and automatic.

The C++ language guarantees that when an object is instantiated, all embedded portions of that object will be built first, followed by the surrounding object. Conversely, when an object is destroyed, the surrounding or owning object is destroyed first, then the embedded objects are destroyed.

Embedding can be nested to any level. The order of construction and destruction is extended, and is analogous to building and ripping apart an onion.

An Example Using Rectangle and Point

Slide Objective
Highlight the private "Point" data member, then show how the c'tor builds it and SetCenter mutates the m_Center.



```
class Point {. . . };

class Rectangle {
public:
  Rectangle(int h=0, int w=0, Point p=Point(0,0));
  Rectangle(int h, int w, int x, int y);
  ~Rectangle();
  void SetCenter(Point p);
  Point GetCenter(void);
      . . .
private:
  Point m_Center;
  int m_nHeight, m_nWidth;
};
```

In the demo program, we have replaced the x and y integer data members with an embedded object of the class Point. Note the following lines in the source:

- Declaration of member `m_Center` within the class Rectangle
- The use of the colon initialization syntax in the constructor for Rectangle
- Implementation of the `GetCenter` and `SetCenter` member functions.

Because we have factored out a concise entity from our original Rectangle implementation, we now have a very usable, modular second class called Point.

Key Point
Rectangle "knows" about Point and implements Point as "m_Center."

Also note that the interface to our Rectangle class is now at a slightly higher level, having moved away from x and y integer coordinates to Point coordinates. Although it is often true that the surrounding class's interface "matures" after embedding objects, from an implementation standpoint, Rectangle's interface does not depend on how we implement coordinates as data. We maintain data independence.

Demo

CONTAIN.CPP is found in \DEMOS\MOD11.

```
// CONTAIN.CPP  found in \demos\mod11
// Classes that contain classes use embedding.
#include <iostream.h>

/********************** Point Class ************************
 Declaration and definition since the Point class has only
 implicitly inline member functions.
************************************************************/
class Point
{
public:                  // construction
    Point(int x=0, int y=0)
        : m_x(x), m_y(y)
        { cout << "Point c'tor\n"; }
    ~Point()
        { cout << "Point d'tor\n"; }
                         // attributes
    int Getx(void) { return m_x;}
    int Gety(void) { return m_y;}
    void Setx(int x) { m_x = x; }
    void Sety(int y) { m_y = y; }
private:                 // implementation
    int m_x, m_y;
};

/************* Rectangle Class Declaration **************/
class Rectangle
{
public:                  // construction
        // Default c'tor creates "point" rectangles at 0,0
    Rectangle();
        // 3-arg c'tor (default arg) may invoke Point
        // c'tor (and its default copy c'tor) to build
        // a Point object at 50,50
    Rectangle(int h, int w, Point p=Point(50,50));
    Rectangle(int h, int w, int x, int y);
    ~Rectangle();
                         // attributes
    void SetCenter(Point p);
    Point GetCenter(void);
                         // implementation
    void Size(int nh, int nw);
    void Draw(void);
private:
    Point m_Center;
    int m_nHeight, m_nWidth;
};
```

(continued)

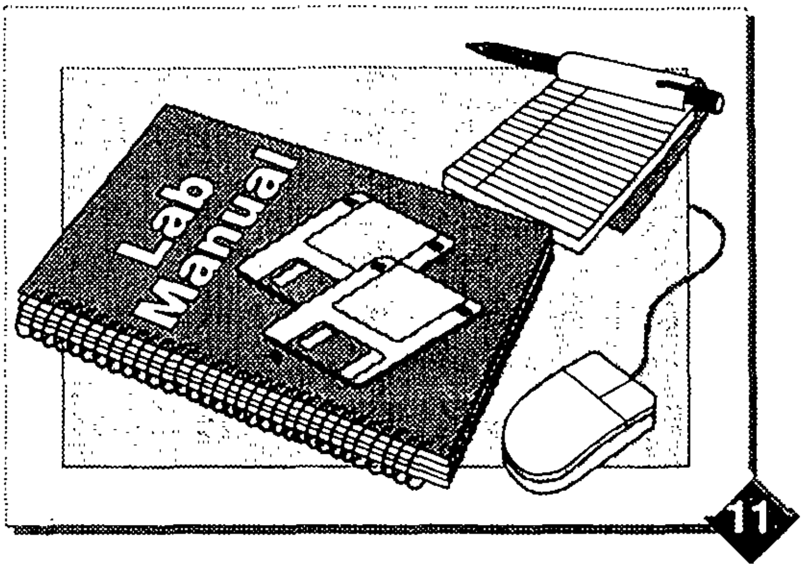
```
/********* Rectangle Member Function Definitions ********/
inline Rectangle::Rectangle()
    : m_nHeight(0), m_nWidth(0), m_Center(0,0)
{
    cout << "Rectangle default c'tor\n";
}

inline Rectangle::Rectangle(int h, int w, Point p)
    : m_nHeight(h), m_nWidth(w), m_Center(p)
{
    cout << "Rectangle c'tor: 3 args (int,int,point)\n";
}

inline Rectangle::Rectangle(int h, int w, int x, int y)
    : m_nHeight(h), m_nWidth(w), m_Center(x,y)
{
    cout << "Rectangle c'tor: 4 args (int,int,int,int)\n";
}

inline Rectangle::~Rectangle()
{
    cout << "Rectangle d'tor\n";
}

inline void Rectangle::SetCenter(Point p)
{
    m_Center = p;
}

inline Point Rectangle::GetCenter(void)
{
    return m_Center;
}

void Rectangle::Size(int nh, int nw)
{
    m_nHeight = nh;
    m_nWidth = nw;
}

// Currently just a display function
void Rectangle::Draw(void)
{
    cout << "Rectangle at x:" << m_Center.Getx()
         << " y:" << m_Center.Gety();
    cout << " height:" << m_nHeight
         << " width:" << m_nWidth;
}
```

(continued)

```
/****************** Simple Test Function ***************/
int main()
{
    cout << "Create p1:";  // Create a Point, p1, at
    Point p1 (25, 35);     // coordinates 25,35
    cout << endl;
    cout << "Create r1:";  // Creating r1 creates a Point
    Rectangle r1;          // with default center 0,0
    cout << endl;
    cout << "Create r2:";      // Create r2 using p1 obj
    Rectangle r2 (1, 2, p1);   // for center at 25,35
    cout << endl;
    cout << "Create r3:";      // Create r3.  Rectangle
    Rectangle r3 (8, 8, 9, 9); // c'tor creates Point(9,9)
    cout <<"\nNow leaving main():";

    //Note: destruction order of non-embedded objects
    //with respect to each other is not guaranteed.
    return 0;
}
```

Lab 9: Containment and Embedded Objects

Slide Objective
Execute the lab solution.
Set the lab objectives.
Ask stucents to read the scenario.



For Your Information
This version of the Inventory class has private data including:

int m_nQuantity

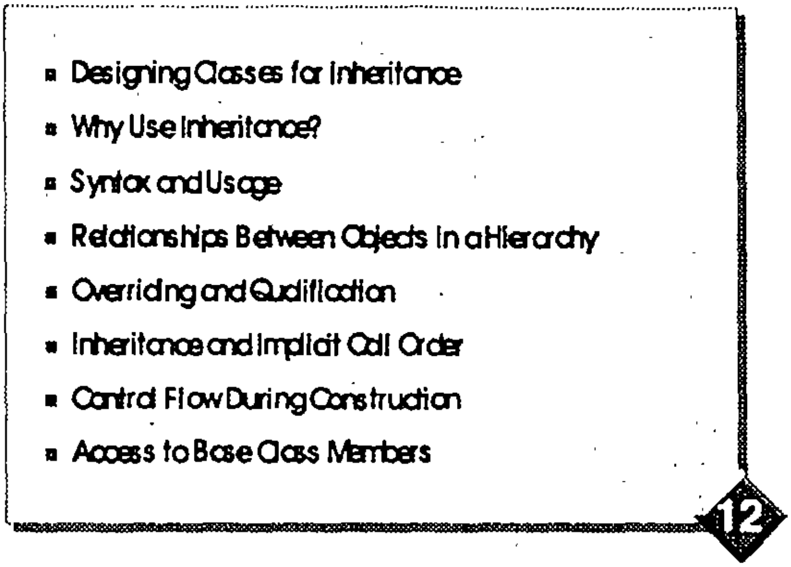
and three objects:

PartID pPartNbr
Money mCost
Date dOrig

Module 12: Using Inheritance

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Designing Classes for Inheritance
- Why Use Inheritance?
- Syntax and Usage
- Relationships Between Objects in a Hierarchy
- Overriding and Qualification
- Inheritance and Implicit Call Order
- Control Flow During Construction
- Access to Base Class Members

This is the last of five modules on implementing simple classes.

Module Summary

In the last module, you studied one possible relationship between classes and their objects—containment. In this module you will study another important relationship: inheritance. Remember that inheritance implies "a type of" relationship. (A third relationship, templates or parameterized types, is beyond the scope of this course.)

A more formal definition for inheritance is the capacity to define new types by stating the differences from a more general type. Inheritance is the mechanism for developing class hierarchies. Class hierarchy is an important concept that underlies commercial class libraries.

Objectives

Key Points
Cover the objectives and propose a problem to be solved in the lab. ISM does business internationally and receives inventory from various places. Rather than create a class for every country and currency, encapsulate the problem into a single class that "knows" how to handle exchange rates.

Upon completion of this module, you will be able to:

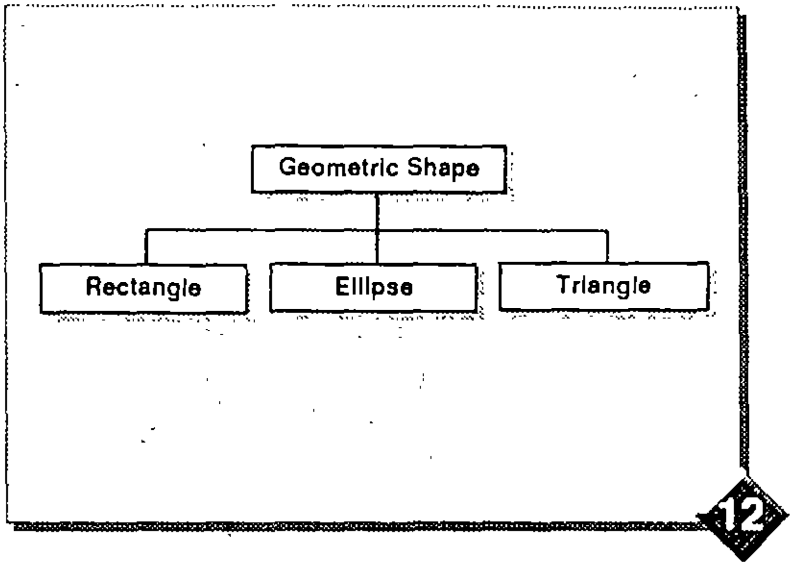
- Create a base class.
- Create a derived class.
- Add a member function to a derived class.
- Properly pass initializers along the construction chain.
- Test a derived class by instantiating objects from it.

Lab

Inheritance

Designing Classes for Inheritance

Slide Objective
Explain the graphical representation of inheritance. Show the big-picture concept. Note: Arrows go UP to the base class.



Geometric Shape

Rectangle

Ellipse

Triangle

Delivery Tips
Keep it simple.

Remember that in the original class design from the first two modules, geometric shapes formed a natural hierarchy, as depicted above. This hierarchy has the following features:

Key Points
Stress terminology:
Base
Derived
General -> specific
"kind of"

- A base class: Geometric Shape
- Three derived classes: Rectangle, Ellipse, and Triangle
- Progression from general to specific, where the derived classes have a "kind of" relationship to the base class.

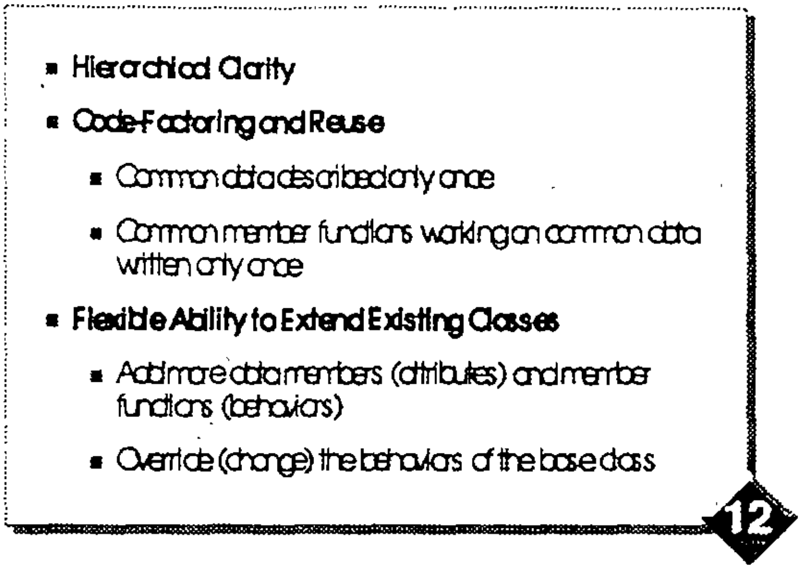
As noted in an earlier module, the base class is also called the parent class or sometimes the superclass; the derived classes are also called child classes or subclasses (super/subclass terminology is from Small Talk®).

Reference

Refer to "Derived Classes," in the *C++ Language Reference*.

Why Use Inheritance?

Slide Objective
Explain the purpose and benefits of inheritance using OOD terminology.



- **Hierarchical Clarity**
- **Code Factoring and Reuse**
  - Common data described only once
  - Common member functions working on common data written only once
- **Flexible Ability to Extend Existing Classes**
  - Add more data members (attributes) and member functions (behaviors)
  - Override (change) the behaviors of the base class

As noted before, a language support of inheritance is important to model real-world relationships. You will see that since C++ syntax denotes inheritance concisely, the design intention is conveyed with authority.

Because derived classes are a type of the base class, derived class objects automatically gain most of the member functions and data members of the base class. This alleviates much of the repetitive coding or data-type tricks necessary to mimic an inheritance relationship in a procedural language like C.

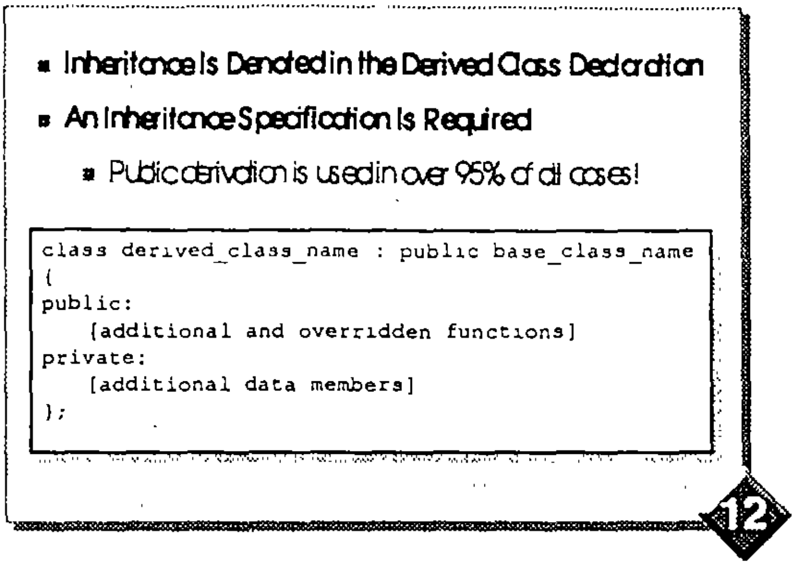
However, a derived class (object) is obviously different from its parent. Therefore, C++ allows you to extend the derived class by two means:

- Creating additional members in the derived class.
- Changing the meaning of an interface inherited from the base class by *overriding* it.

When applied properly, these features make inheritance a very powerful concept.

Syntax and Usage

Slide Objective
Show the syntax and detail public inheritance. Eschew protected and private.



- Inheritance Is Denoted in the Derived Class Declaration
- An Inheritance Specification Is Required
  - Public derivation is used in over 95% of all cases!

```
class derived_class_name : public base_class_name
{
public:
    [additional and overridden functions]
private:
    [additional data members]
};
```

The class declaration syntax for showing inheritance is straightforward. For example:

```
class Rectangle : public GeoShape
{
public:
. . .
};
```

In the foil, note the use of the keyword **public**. In the first line, it denotes inheritance specification. In the third, it denotes access specification (which you should be familiar with).

The vast majority of designs in C++ use **public** derivation. The use of **private** and **protected** derivation is beyond the scope of this course.

Demo

INHERIT.CPP is in \DEMOS\MOD12.

```
// INHERIT.CPP found in \demos\mod12
// GeoShape has an embedded Point.  Rectangle inherits
// from GeoShape and calls base member functions.
#include <iostream.h>

/******* Declaration and Definiton of Point Class ******/
class Point
{
public:                           // construction
    Point(int x=0, int y=0)
        : m_x(x), m_y(y)
        { cout << "Point c'tor\n"; }
    ~Point()
        { cout << "Point d'tor\n"; }
    int Getx(void) { return m_x;} // attributes
    int Gety(void) { return m_y;}
    void Setx(int x) { m_x = x; }
    void Sety(int y) { m_y = y; }
private:                          // implementation
    int m_x, m_y;
};

/************ GeoShape Class Declaration ****************
 * Base class for the 2-D geometrical classes Rectangle, *
 * Ellipse, and Triangle.  Dimensions do not make sense  *
 * for a generic shape, but a center point does.         *
 *********************************************************/
class GeoShape
{
public:                          // construction
    GeoShape(Point p=Point(0,0));
    GeoShape(int x, int y);
    ~GeoShape();
    void SetCenter(Point p);     // attributes
    Point GetCenter(void);
    void Draw(void);             // operations
private:                         // implementation
    Point m_Center; // Point is "embedded" in GeoShape
};

/************* Rectangle Class Declaration **************/
class Rectangle : public GeoShape   // public inheritance
{
public:                          // construction
    Rectangle();
    Rectangle(int h, int w, Point p=Point(50,50));
    Rectangle(int h, int w, int x, int y);
    ~Rectangle();
    void Size(int nh, int nw);   // operations
    void Draw(void);
private:                         // implementation
    int m_nHeight, m_nWidth;
};

```

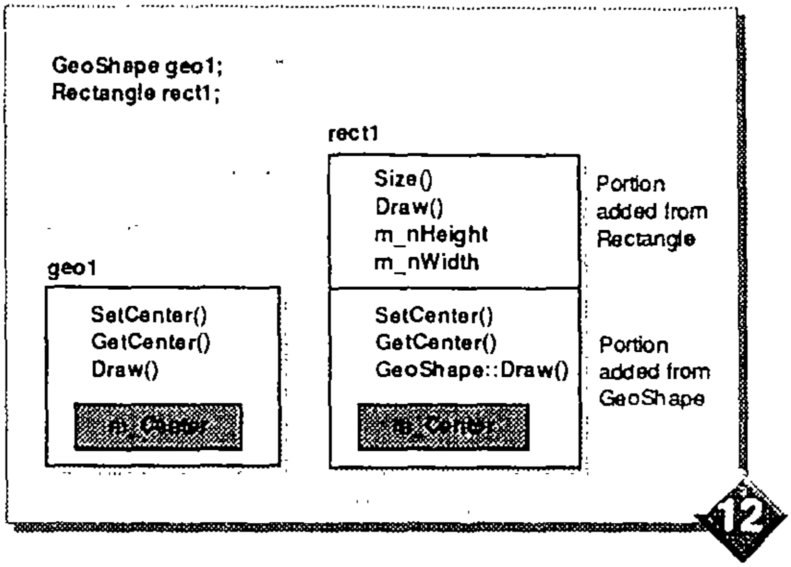
(continued)

```
/******** GeoShape Member Function Definitions **********/
inline GeoShape::GeoShape(Point p)
    : m_Center(p)
{
    cout << "GeoShape c'tor: 1 arg\n";
}

inline GeoShape::GeoShape(int x, int y)
    : m_Center(x,y)
{
    cout << "GeoShape c'tor: 2 arg\n";
}

inline GeoShape::~GeoShape()
{
    cout << "GeoShape d'tor\n";
}

inline void GeoShape::SetCenter(Point p)
{
    m_Center = p;
}

inline Point GeoShape::GetCenter(void)
{
    return m_Center;
}

/* Currently just a display function */
void GeoShape::Draw(void)
{
    cout << "Center at x:" << m_Center.Getx()
         << " y:" << m_Center.Gety() << endl;
}

/******** Rectangle Member Function Definitions *********/
inline Rectangle::Rectangle()
    : m_nHeight(0), m_nWidth(0), GeoShape(0,0)
{
    cout << "Rectangle default c'tor\n";
}

inline Rectangle::Rectangle(int h, int w, Point p)
    : m_nHeight(h), m_nWidth(w), GeoShape(p)
{
    cout << "Rectangle c'tor: 3 arg (int,int,Point)\n";
}

inline Rectangle::Rectangle(int h, int w, int x, int y)
    : m_nHeight(h), m_nWidth(w), GeoShape(x,y)
{
    cout << "Rectangle c'tor: 4 arg (int,int,int,int)\n";
}

(continued)
```

```
inline Rectangle::~Rectangle()
{
    cout << "Rectangle d'tor\n";
}

void Rectangle::Size(int nh, int nw)
{
    m_nHeight = nh;
    m_nWidth = nw;
}

/* Currently just a display function */
void Rectangle::Draw(void)
{
    GeoShape::Draw();     // :: used for qualification
    cout << " height:" << m_nHeight
         << " width:" << m_nWidth;
}

/************* Small Test Program ***********************/
void main()
{
    cout << "Create p:";
    Point p (55, -55);
// Although it's possible to tag a class to
// enforce its abstractness, the method is
// beyond the scope of this course.
    cout << "Creating two generic objects:\n";
    GeoShape g1, g2 (12, -12);
    cout << "Creating three rectangles:\n";
    Rectangle r1 (2, 4, 150, 150),
              r2 (10, 10, p),
              r3 (55, 55);

    cout<<"\n\"Draw\" two objects:\n";
    cout <<"g1 draws  s: \n";
    g1.Draw();
    cout <<"r2 draws  s: \n";
    r2.Draw();
    cout << "\nEnding main()" << endl;
}
```

Relationships Between Objects in a Hierarchy

Slide Objective
Quickly, confirm student understanding that Rectangle inherits the base functionality of GeoShape and adds its own behaviors.



GeoShape geo1;
Rectangle rect1;

In inheritance, it is critically important to differentiate between objects and classes and how they are related.

The base class shown here, GeoShape, declares a set of member functions and data members. An object of this type, such as **geo1**, contains those data members and has access to the member functions.

Tip Each object, of course, does not contain member functions.

Although the derived class, Rectangle, does not explicitly declare the members **Draw**, **GetCenter**, **SetCenter**, and **m_Center**, it gains these members from the base class, GeoShape. It declares three new members, **Size**, **m_nHeight**, and **m_nWidth**, and overrides the **Draw** function.

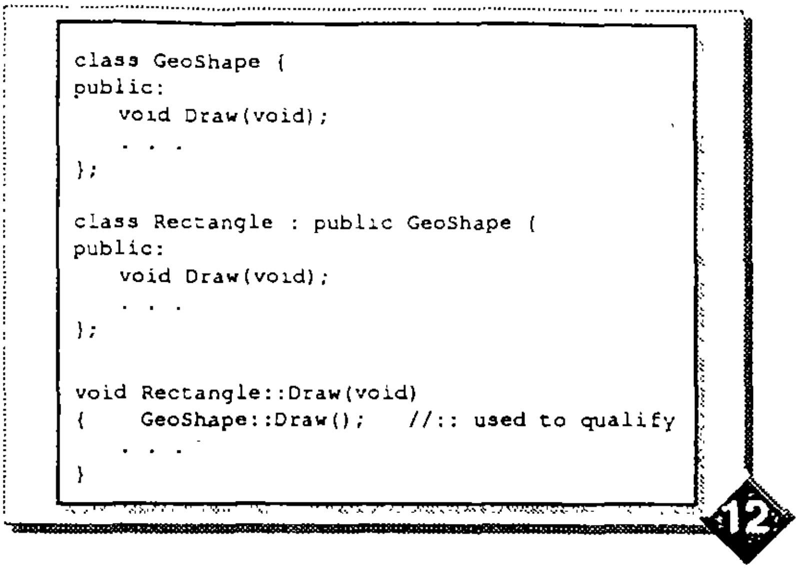
Delivery Tips
Don't explain details concerning the Draw functions. Save for next page.

Therefore, an object of type Rectangle, such as rect1, contains all the mentioned members of the base class as well as those declared in the derived class.

If we look at an object from each class, such as geo1 and rect1, there is a strong resemblance. To beginners, this is sometimes misinterpreted. Although their classes are related, the objects geo1 and rect1 are not related, in the sense that manipulating one will not have an effect on the other.

Overriding and Qualification

Slide Objective
Complete the details concerning the derived class, Rectangle, "overriding" the Draw function in the base class, GeoShape.



```
class GeoShape {
public:
    void Draw(void);
    . . .
};

class Rectangle : public GeoShape {
public:
    void Draw(void);
    . . .
};

void Rectangle::Draw(void)
{    GeoShape::Draw();    //:: used to qualify
    . . .
}
```

Although the **Draw** function is inherited by Rectangle, its base implementation is inadequate—we want a rectangle object to display dimensional information also. C++ allows us to supply a new definition for a function in a derived class; this is called *overriding*.

To override a function in the derived class, it must only have the same name. Overridden functions generally have the same prototype also. When you invoke the function using a derived object, for example,

```
rect1.Draw();
```

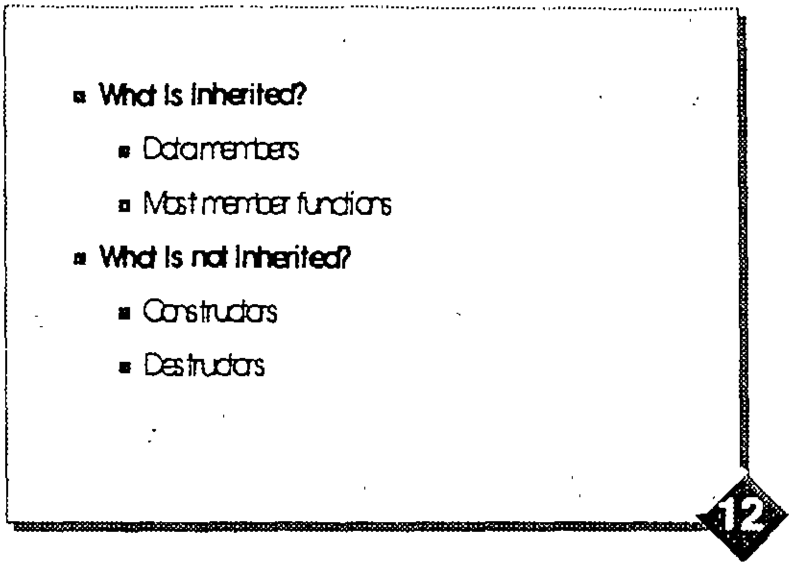
the derived class's version of **Draw** is invoked by default. If you wish to invoke the base class's version, qualification can be used:

```
rect1.GeoShape::Draw();
```

Note that in INHERIT.CPP, the definition of **Draw** for Rectangle uses qualification to invoke its parent's version. Then it does some additional work.

Tip Overriding should not to be confused with *overloading*. Overloading occurs in the same scope, and the compiler differentiates functions by argument type and number. Overriding occurs across inheritance scopes, and the base function is normally hidden in the derived class.

Inheritance and Implicit Call Order



- What is inherited?
  - Data members
  - Most member functions
- What is not Inherited?
  - Constructors
  - Destructors

12

In this module, the subject of constructors and destructors has been avoided until now. Because they are special member functions that relate to the life and death of class objects, they are not inherited as other members are.

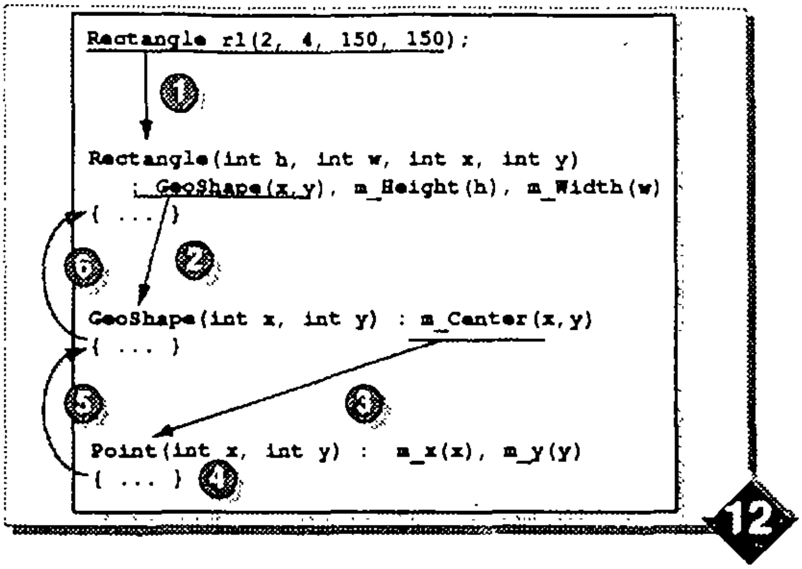
The convenience of constructors and destructors is not forfeited, however. Since a derived object has a portion that it gains from the base class, C++ automatically invokes the base class constructor and destructor for that portion. And as with embedded objects, C++ guarantees an order of construction and destruction.

Construction Graphic is NEXT PAGE

That order is presented on the next page.

Control Flow During Construction

Slide Objective
Trace through the diagram to build an understanding of the c'tor order.



```
Rectangle r1(2, 4, 150, 150);

Rectangle(int h, int w, int x, int y)
    : GeoShape(x,y), m_Height(h), m_Width(w)
{ ... }

GeoShape(int x, int y) : m_Center(x,y)
{ ... }

Point(int x, int y) :  m_x(x), m_y(y)
{ ... }
```

Construction call order: 1. Base class portion

1a. Embedded objects, if any

1b. Surrounding portion

2. Derived portion

2a. Embedded objects, if any

2b. Surrounding portion

Destructors are called in reverse order.

When the Rectangle object `rect1` in INHERIT.CPP is defined, the following occurs:

1. The `Rectangle` constructor is invoked when `rect1` is defined.
2. Since the base class portion of `rect1` must be built first, the constructor for the base class is called and passed `x` and `y`.
3. The `GeoShape` constructor invokes the embedded object `m_Center` constructor.
4. The body of the `Point` constructor is executed.
5. The body of the `GeoShape` constructor is executed.
6. The body of the `Rectangle` constructor is executed.

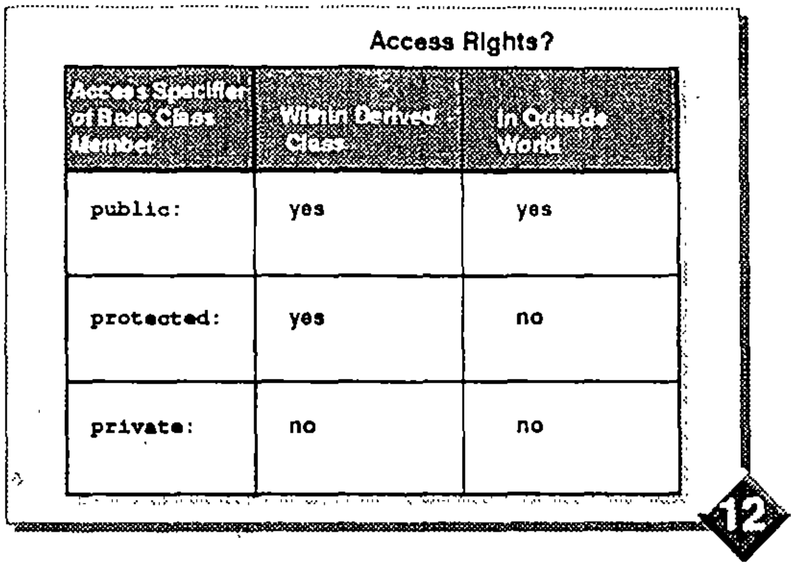
Remember that before the body of a constructor function is entered, C++ guarantees that the colon-initialized data members will have their proper values. For the standard data type members, this has not been explicitly shown in the diagram above.

During destruction of an object, the order of destructor calls is reversed. It is considerably simpler because there are no arguments being passed around.

Proper use of colon initialization is especially important within classes that have inheritance or contained objects.

Access to Base Class Members

Slide Objective
Present this table as a summary of inheritance, detailing ways to access the base class.



Access Rights?

Access Specifier of Base Class Member	Within Derived Class	In Outside World
`public:`	yes	yes
`protected:`	yes	no
`private:`	no	no

Under **public** derivation, there are strict rules of access to base class members, both with respect to the derived class member functions, and with respect to the outside world (global functions and other, unrelated classes).

The **public** members of a base class can be accessed anywhere.

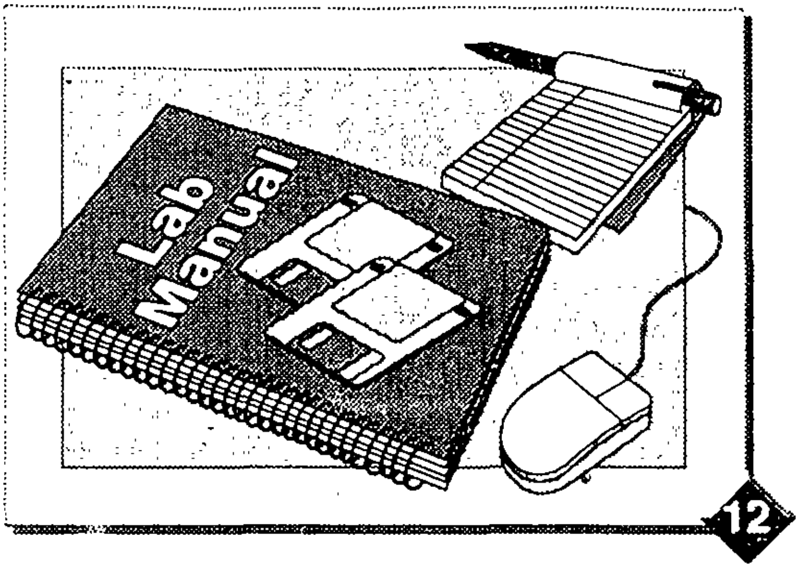
The **private** members can only be directly accessed by member functions of the current (base) class. *Even its child class cannot access these directly!* This is analogous to your internal organs; they are a part of you, but can only be accessed indirectly.

A base class's **protected** members are midway between **public** and **private**. They are inaccessible outside the class hierarchy, but are accessible to any child classes.

Lab 10: Inheritance

Slide Objective

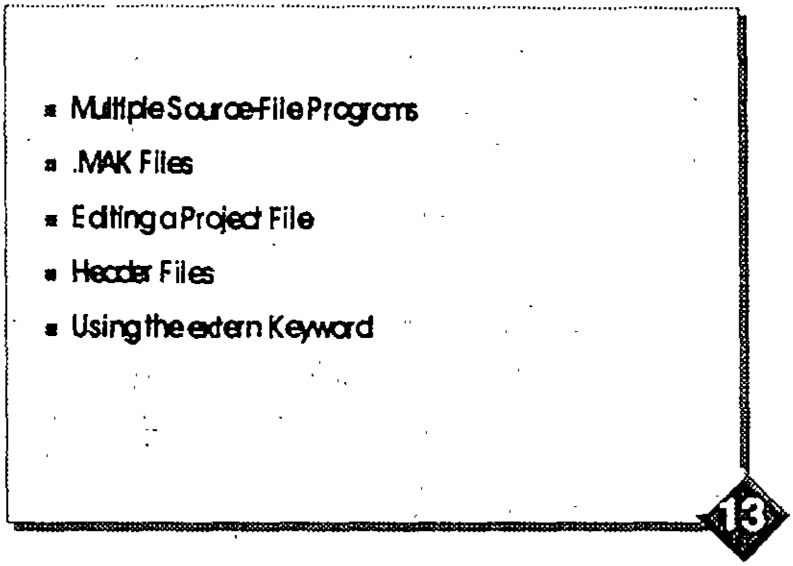
Execute the lab solution (again).
Set the lab objectives.
Ask students to read the lab scenario.



Module 13: Managing Complex Projects Using the Integrated Development Environment

∑ Overview

Slide Objective
Provide an overview of the module contents.



- Multiple Source-File Programs
- .MAK Files
- Editing a Project File
- Header Files
- Using the extern Keyword

Module Summary

Up to now, your programs existed in a single file. It is common, however, for real-world projects to extend over many source files. You'll create a ***project*** to manage the various dependencies that multiple files entail. Project information is maintained in ***make*** files (.MAK extension).

Visual Workbench provides important tools for managing projects. In this module, you'll explore the process of creating and maintaining a project file.

Objectives

Upon completion of this module, you will be able to:

- Use the Project Manager to specify options.
- Create header files.
- Use the **extern** keyword to provide cross-module data access.

Delivery Tips
Present objectives for the module to set the direction.

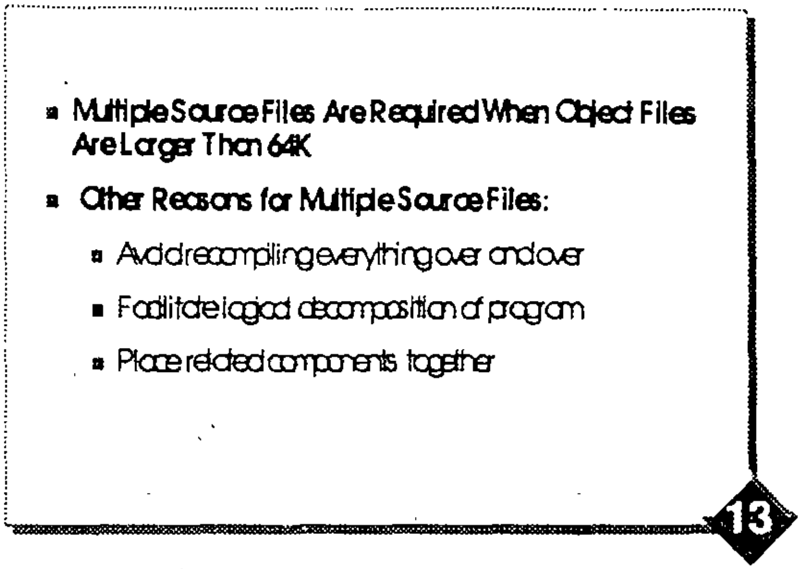
Don't bother to execute any lab solutions. No changes are evident.

Lab

Managing Projects

Mulitple Source-File Programs

Slide Objective
Set a real-world expectation for the processes that are encountered developing large applications.



- Multiple Source Files Are Required When Object Files Are Larger Than 64K
- Other Reasons for Multiple Source Files:
  - Avoid recompiling everything over and over
  - Facilitate logical decomposition of program
  - Place related components together

Apart from this 16-bit limitation, you will commonly encounter other situations where multiple source files are efficient and practical.

Visual Workbench supports an incremental build feature that allows you to rebuild only those source files that you have changed since the last build. If all of your source code is in one big file, you will always rebuild everything. But if you split things up as you work on various parts of the program, the compiler only has to touch a few files, and the build process is sped up significantly.

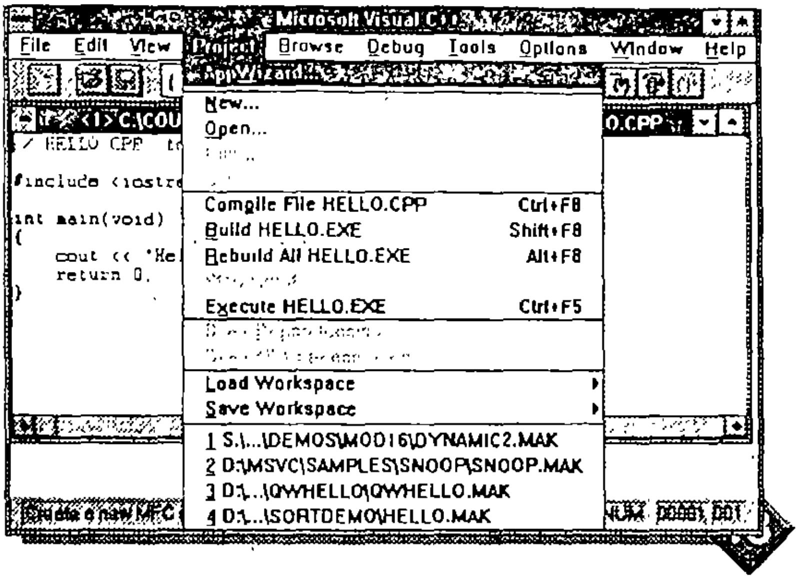
Splitting files as they grow in complexity also enhances their readability. There are conventions for splitting monolithic source files. As you have seen in earlier modules, C++ programs have a definite structure to them. Preprocessor directives, declarations, and function prototypes are placed in header (.H) files. Associated function definitions are segregated into their own source files (.CPP). Depending upon the type of program you are creating (MS-DOS®, Windows, QuickWin, and so on), there will be other files as well.

In the lab for this module you will split up a single source file and create a project.

.MAK Files

Slide Objective
Present a high-level overview of the purpose and benefits of Project .MAK files.

Cover "What and Why" — later pages cover How.



Make Files (.MAK)

When you build a program, the Make utility invokes the compiler and linker with specific instructions you want. Make files contain other important information about your project too: its path, the type of executable that you are building (Windows, QuickWin, MS-DOS, and so on), whether it uses MFC libraries, and a list of the source files to include. It also controls the libraries that your program will link to for the code that is needed to execute run-time functions.

Tip Under Visual Workbench, make files are transparent.

Project information has been set for you in the examples you've seen up to now. You will, however, need to know how to set options for future programming projects as you return to your Workplace. You'll go through the process in the next few foils.

Opening Projects

You have three choices for opening a project using a .MAK file. From the Project menu, you can:

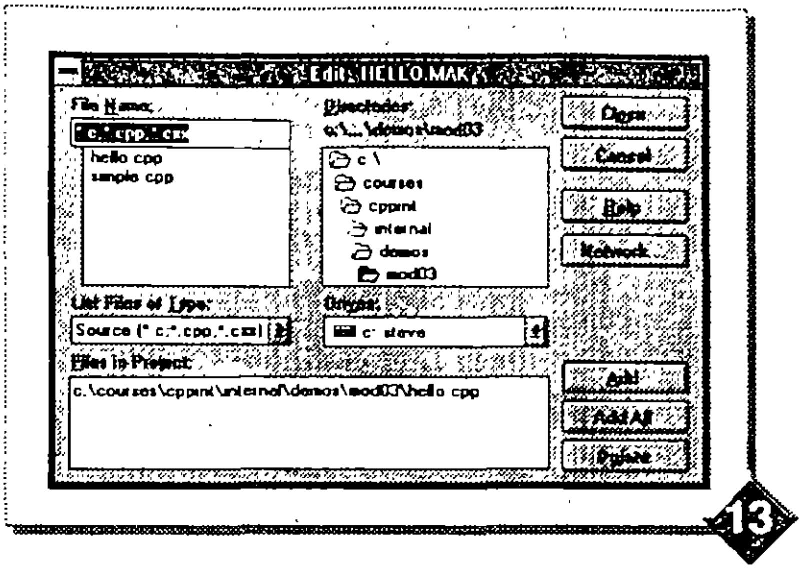
1. Use the New command to create a new project.
2. Use the Open command to browse for an already created project.
3. Select from the last four projects you worked on listed at the bottom of the menu.

Opening Files Within a Project

No matter what method you use to start a project, the easiest way to navigate among the files in the project is using the Project Files button on the extreme left end of the toolbar

Editing a Project File

Slide Objective
Continue to provide a high-level overview of the options to deal with .MAK files and Projects.



File Name: *.c;*.cpp;*.cxx
hello cpp
sample cpp

Directories: c:\...\demos\mod03
c:\
courses
cppint
internal
demos
mod03

List Files of Type: Source (*.c;*.cpp;*.cxx)

Drives: c: steve

Files in Project:
c:\courses\cppint\internal\demos\mod03\hello cpp

Delivery Tips
Move quickly! The lab instructions contain step-by-step instructions for these procedures.

Editing a Project

Whether you use the New command or the Open command from the Project menu, you end up at the Edit dialog box. This dialog allows you to edit the .MAK file. It is from this dialog that you can either add or delete files from your project.

Editing an Existing Project File

Open Visual Workbench. From the Project menu, choose Edit. This displays the Edit dialog box.

Use the Drives and Directories boxes to find the files you want to add to your project.

Select the individual files from File Name dialog box and choose the Add button.

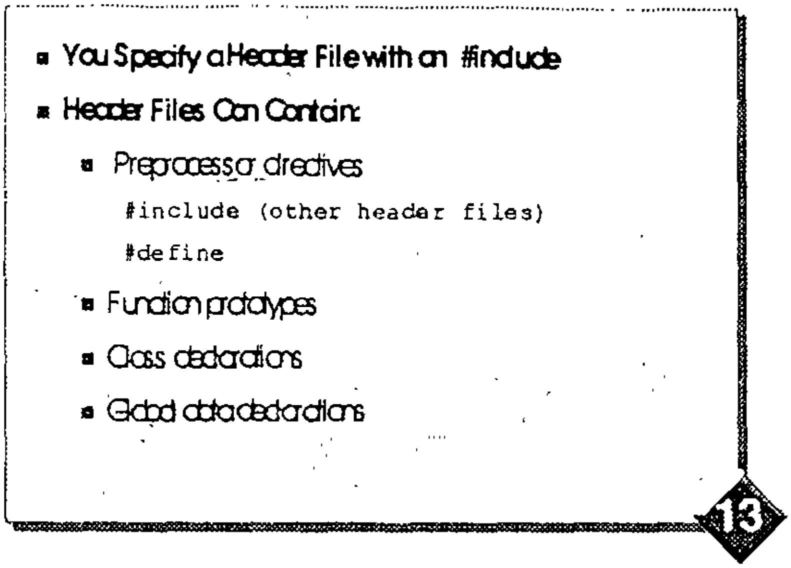
When you're finished, choose Close.

Dependencies

During the discussion of preprocessor directives, you learned that you can specify dependencies with #includes. Visual Workbench automatically scans for all these dependencies when you edit your project file. As you include new source files into your project you should force a rescan of dependencies. The Scan All Dependencies option on the Project menu regenerates the dependency list for the entire project. The Scan Dependencies *ActiveFilename* will scan just the active file.

Header Files

Slide Objective
Add details to the purpose and use of header files.



- You Specify a Header File with an #include
- Header Files Can Contain:
  - Preprocessor directives

    #include (other header files)

    #define
  - Function prototypes
  - Class declarations
  - Global data declarations

Header files (extension .H) contain information that must be available globally. In your earlier programs, you included IOSTREAM.H, which contained information about cin and cout. You specified the streams header file with an **#include**:

```
#include<iostream.h>
```

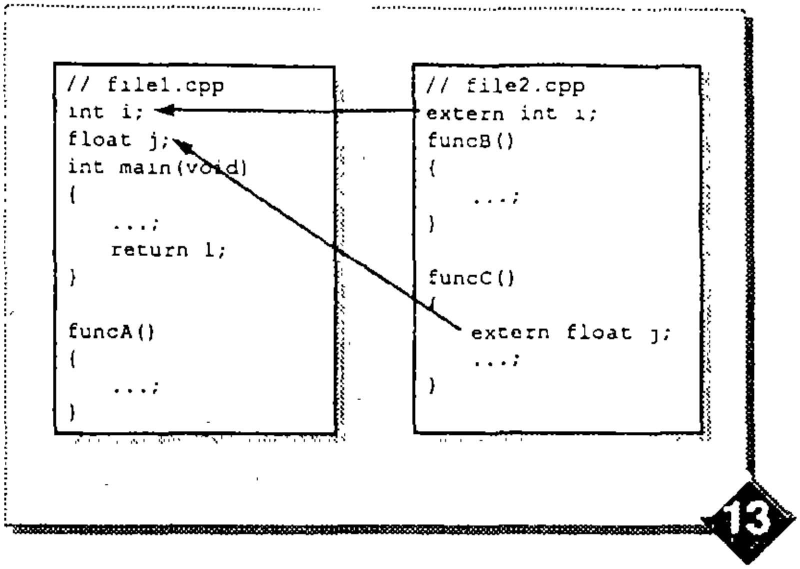
Now that you are setting up multiple source-file projects, you should extract any information that you want all the files to see into a header file. Then include it. One nice feature of Visual Workbench is that it will recursively scan all the source files that have been added to your project file for include dependencies. If, however, you create any #includes in your source files after the files are added to your project, you must force a scan. You'll see how to do this later in the module.

Declarations and prototypes usually go in header files. For example, function prototypes should go in header files but, in general, their definitions do not. Class declarations definitely go in header files, but their member function definitions belong in a separate source file (.CPP).

Recall from an earlier discussion that an **#include** tells the preprocessor to go out and find a file and place its contents at this point in the code. This is a shorthand way to place the same information at the top of each of your source files. Why is this important? In C++, all functions must be prototyped before they are called. If a function is used in more than one of your source files, it must be prototyped at the start of each file. An **#include** statement at the top of the file takes care of this.

Using the extern Keywor

Slide Objective
Introduce the "extern" keyword as a type-modifier in the declaration of variables.



```
// file1.cpp
int i;
float j;
int main(void)
{
   ...;
   return 1;
}

funcA()
{
   ...;
}
```

```
// file2.cpp
extern int i;
funcB()
{
   ...;
}

funcC()
{
   extern float j;
   ...;
}
```

What the extern Keyword Does

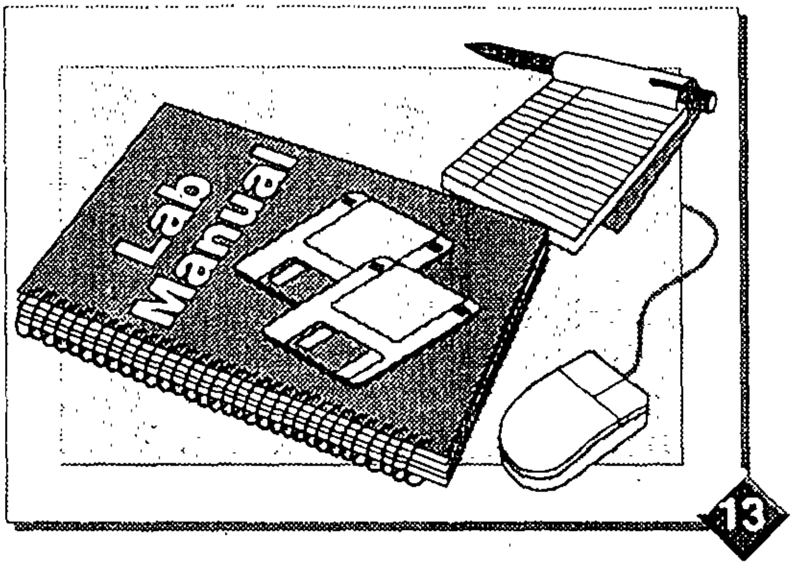
The **extern** keyword is a storage-class specifier. It makes another file's global variables visible to one or all functions in a source file. In essence, it says to the compiler that storage will be found for the variable at link time.

In the foil, the `extern int i` statement in `file2` references the `int` declared in `file1` and makes that variable available to all functions in `file2`. The `extern float j` statement makes the variable defined in `file1` visible only to the statements within `funcC`.

Tip In some computer languages, all data is global. One of the advantages of C++ is that data can be encapsulated within objects. This adds modularity to your programs—it makes them easier to reuse and maintain. As a programmer, you should begin taking more advantage of this feature of the language by reducing your dependence on global data.

Lab 11: Managing Projects

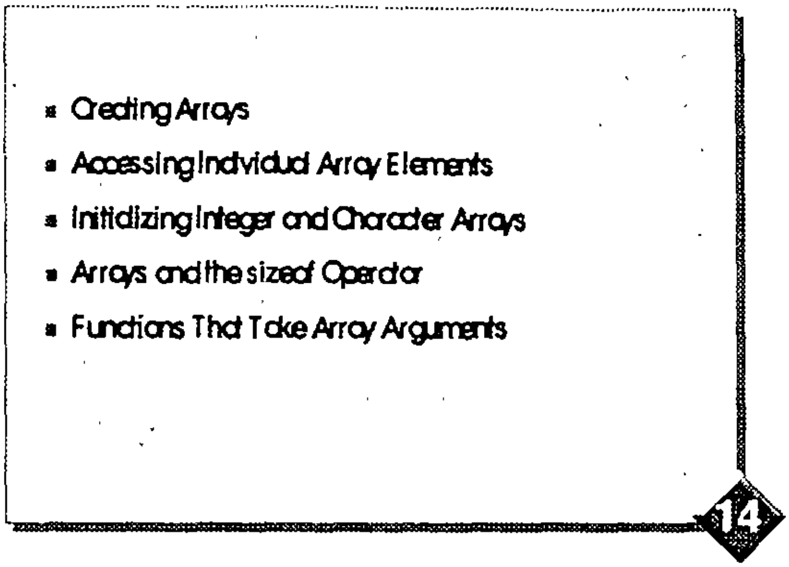
Slide Objective
Provide an overview of the labs.



Module 14: Using Arrays

∑ Overview

Slide Objective
Provide an overview of the module contents.



- Creating Arrays
- Accessing Individual Array Elements
- Initializing Integer and Character Arrays
- Arrays and the sizeof Operator
- Functions That Take Array Arguments

Module Summary

This module begins a three-module sequence on arrays, pointers, references, and objects that contain arrays of data—that is, strings of characters. From the first module on, you have been using data in your programs. Without exception, however, your variables have contained single values. From your experience, you already know that it is important to create variables that contain more than one data element. It is also important to be able to index and examine them individually, and be able to manipulate them as a whole. In C++, such a variable is declared as an array.

Although arrays (particularly strings) will be used throughout the remainder of this course, the primary value of an array will be realized once you've returned to your workplace. It is hard to imagine solving many real-world problems without arrays and strings.

In the next modules, you will learn to manipulate arrays using pointers, and you will see how objects of a commercial string class can be used to simplify the manipulations you learned in this module.

Objectives

Upon completion of this module, you will be able to:

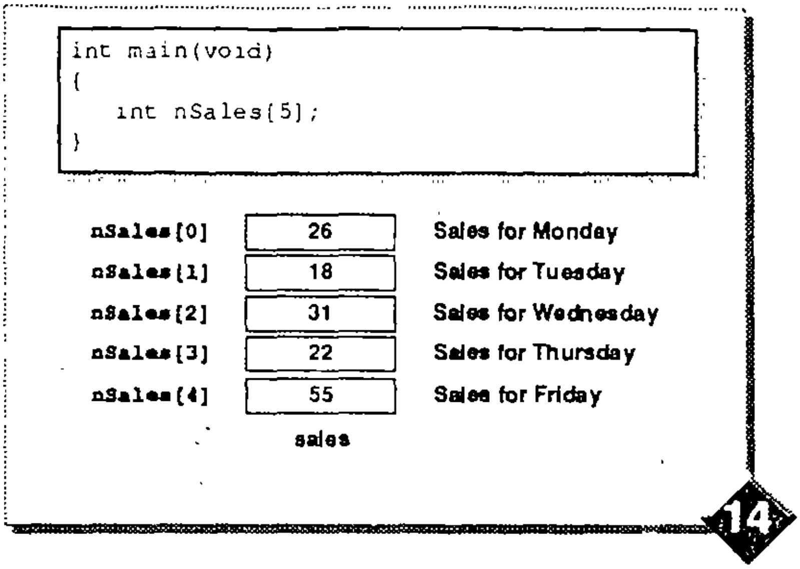
- Create an array.
- Manipulate an array using subscript notation.
- Create a character array as a string.
- Manipulate a string.

Lab

Manipulating Arrays

Creating an Integer Array

Slide Objective
Present a simple overview dealing with a local integer array. Present the purpose and uses for arrays.



```
int main(void)
{
   int nSales[5];
}
```

nSales[0]	26	Sales for Monday
nSales[1]	18	Sales for Tuesday
nSales[2]	31	Sales for Wednesday
nSales[3]	22	Sales for Thursday
nSales[4]	55	Sales for Friday
	sales	

What Is an Array?

An array is a collection of contiguous data, all of the same data type. An integer array is an array of 2-byte elements.

Single-Dimension Arrays

In the example on the slide, you see an integer array being declared. It uses the name **nSales**, and it allocates five bytes of storage.

This array is declared as a local variable, so it has the same scoping and storage class rules as ordinary variables do. Note that global arrays are initialized to 0 by the compiler, and that auto arrays can easily exhaust the stack. Also, because it is a stack-based (auto) array, its contents are undefined at this point. Finally, note that the total size of each array or the range of the subscripts must be known at compile time.

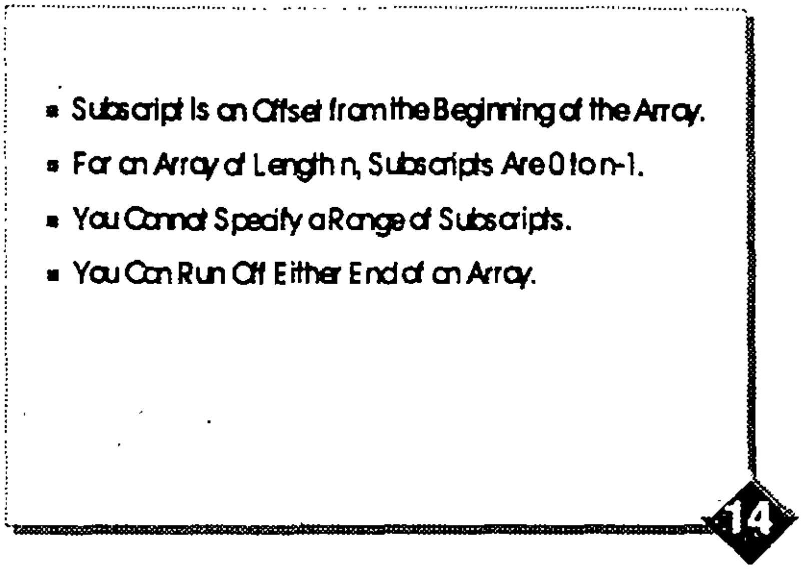
Demo

ARRAY.CPP is located in \DEMOS\MOD14. It shows how to create an array and access elements.

```
// ARRAY.CPP    Found in \demos\mod14
// Creating arrays follows the scoping, initialization and
// assignment rules as standard data types but adds a
// subscript notation to address individual array elements.
#include <iostream.h>

int main(void)            // test function
{
    // Declare an integer array will space for 5 integers
    int nSales[5];        // nSales has undefined contents
    // Assign values to each element using subscripts
    // starting at ZERO counting up to array size-1.
    nSales[0] = 26;       // Monday sales total
    nSales[1] = 18;       // Tuesday
    nSales[2] = 31;       // etc.
    nSales[3] = 22;
    nSales[4] = 55;

    cout << "    I.S.M. Inc.\nWeekly Sales Report\n";
    cout << "\nMonday    $" << nSales[0];
    cout << "\nTuesday    " << nSales[1];
    cout << "\nWednesday  " << nSales[2];
    cout << "\nThursday   " << nSales[3];
    cout << "\nFriday     " << nSales[4];
                          // Total daily sales
    long sales = nSales[0] + nSales[1] +
                 nSales[2] + nSales[3] + nSales[4];
    cout << "\n   Total $" << sales << endl;
    return 0;
}
```

Accessing Individual Array Elements

Slide Objective
Define subscripting as an addressing mechanism — simple address addition.



- Subscript Is an Offset from the Beginning of the Array.
- For an Array of Length n, Subscripts Are 0 to n-1.
- You Cannot Specify a Range of Subscripts.
- You Can Run Off Either End of an Array.

Think of an array as being like the houses on a block. What is the distance from the beginning of the block to the first house on the block? It's 0, and this provides a clue as to what subscripts are to the compiler. They are a measure of the displacement or offset of an array element from the beginning of the array. Element #1 in an array is at an offset of 0 from the beginning of the array.

Key Points
C++ programmers count from zero!

Actually, this is true for all arrays in a computer. Compilers for languages that permit subscripts starting at 1 make an adjustment to reflect this fact. The C++ compiler doesn't have to make an adjustment. The programmer coming to C/C++ from another language makes the adjustment mentally.

Demo

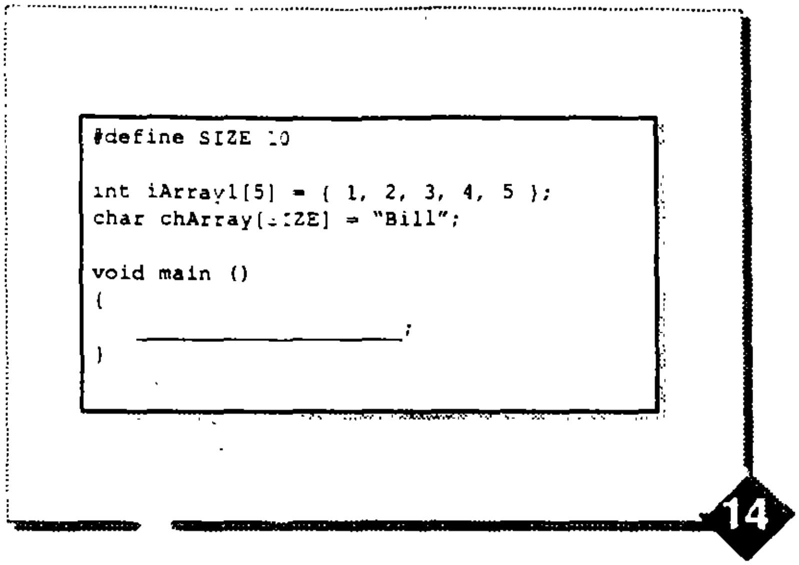
ACCESS.CPP is located in \DEMOS\MOD14. It shows how to use subscript notation to access array elements.

Accessing array elements using subscript notation

```
// ACCESS.CPP   Found in \demos\mod14
// Array elements are typically accessed using a variable
// within the subscript notation.
#include <iostream.h>

int main(void)           // test function
{
    int i = 0;  // Use an integer to index array elements
    int nSales[5];       // nSales has undefined contents
    // Assign values to each element using subscripts
    // starting at ZERO
    nSales[0] = 26;      // Monday sales total
    nSales[1] = 18;      // Tuesday
    nSales[2] = 31;      // etc.
    nSales[3] = 22;
    nSales[4] = 55;
//  This is not a language error, it is a logic error.
//  nSales[5] = 7; // #1 common programming error-Trouble!

    cout << "    I.S.M. Inc.\nWeekly Sales Report\n";
    for (long lSales = 0L; i < 5; i++)
    {                    // "i" indexes the array
        cout << "\nDay " << i << "     $" << nSales[i];
        lSales += nSales[i];
    }
    cout << "\n   Total $" << lSales << endl;
    return 0;
}
```

Initializing Integer and Character Arrays

Slide Objective
Each of previous examples used multiple lines to set values into the array elements. Introduce ways to efficiently initialize arrays.



```
#define SIZE 10

int iArray1[5] = { 1, 2, 3, 4, 5 };
char chArray[SIZE] = "Bill";

void main ()
{
    ____________________;
}
```

Key Point
Let the compiler count.

The size of an array must be known at compile time. Generally, you provide this size by means of the number in brackets in the array declaration. If the array is being initialized, however, the compiler can count the elements between the curly braces to derive the size of the array.

For instance, both of the following produce the same results:

```
static int nPowersOf2[5] = { 1, 2, 4, 8, 16 };
```

or

```
static int nPowersOf2[] = { 1, 2, 4, 8, 16 };
```

Delivery Tip
Don't get off topic talking about character arrays!

There are several advantages to letting the compiler derive the size of an initialized array. When you are initializing an array, you often want to change it by adding or removing an element. If you specify the size, you have to change it. There's always a chance you'll forget, or that you'll miscount the elements in the array set. The compiler never miscounts.

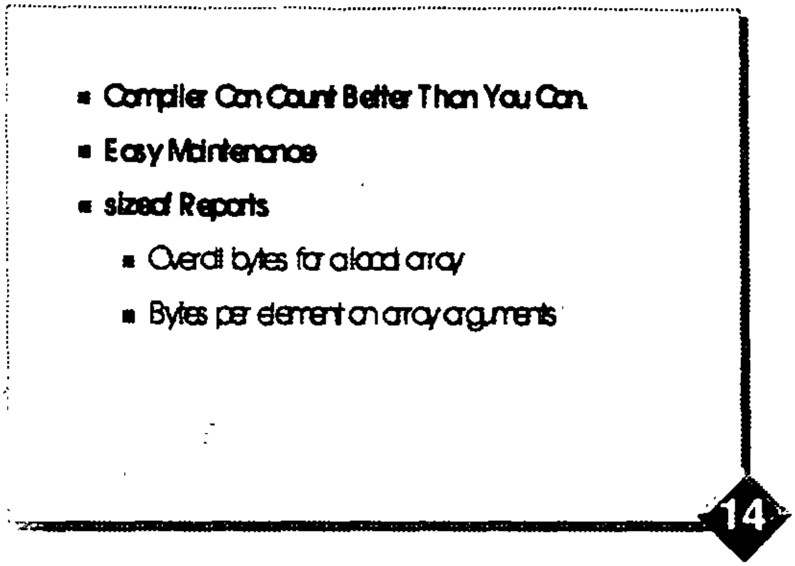
Demo

INITARY.CPP is located in \DEMOS\MOD14. It shows the initialization of integer and character arrays.

```
// INITARY.CPP   Found in \demos\mod14
// Alternate ways to initialize elements in an array.
#include <iostream.h>
                          // manifest constant
#define NBR_OF_INTS 5

void main()               // simple test function
{
    int iCount, iPO2Sum = 0;
                          // Explicitly sized using manifest
                          // constant (for maintainability)
    int iPowersOf2[NBR_OF_INTS]
       = { 1, 2, 4, 8, 16 };
                          // Implicitly sized, compiler
                          // will count elements and size
    int iNbrSeries[]      // the array to match the list.
       = { 1, 2, 4, 8, 16 };
                          // Loop to total the array
    for (iCount = 0; iCount < NBR_OF_INTS; iCount++)
        iPO2Sum += iPowersOf2[iCount];

    // Below are three ways to initialize character arrays.
    // Output is: "The sum of the 1st 5 powers of 2 is "
                          // Init to size with string literal
    char szMsg1[16] = "The sum of the ";
                          // Init letting compiler count
chars
    char szMsg2[] = "1st 5 powers of 2 ";
                          // Init by programmer with too much
                          // free time (Note: NULL is '\0').
    char szMsg3[] = {'i', 's', ' ', '\0'};
    cout << szMsg1 << szMsg2 << szMsg3 << iPO2Sum << endl;
}
```

Arrays and the sizeof Operator

Slide Objective
The compiler can count elements for programmers. Does the programmer need to know how many elements exist? Use sizeof operator.



- **Compiler Can Count Better Than You Can.**
- **Easy Maintenance**
- **sizeof Reports**
  - Overall bytes for a local array
  - Bytes per element on array arguments

Key Points
The sizeof operator is resolved at compilation time.
Aids portability in source code.
Works great on standard and user-defined data types.
Works great on arrays of local or global scope.

"sizeof" returns just the size of an element for arrays passed as arguments! See the demo.

When you are writing loops, how do you know how big the array is? The **sizeof** operator comes to your rescue. You were introduced to the **sizeof** operator in an earlier module.

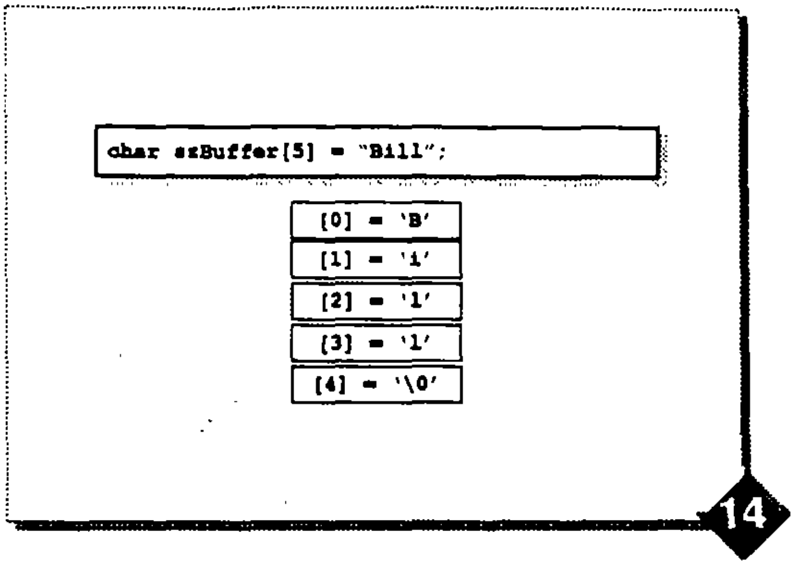
Demo

INITARY2.CPP is in \DEMOS\MOD14. It shows how to initialize arrays and pass them to a function. Note the difference from the sizeof operator.

```
// INITARY2.CPP  Found in \demos\mod14
// The compiler can determine the number of elements in an
// array.  The sizeof operator allows programs to discover
// that length at runtime without a maintenance problem.
#include <iostream.h>
                        // function prototype
void IntArrayTotal(int[], int);
                        // manifest constant
#define NBR_OF_INTS 5

/**************** Simple Test Function ****************/
void main()
{                       // Explicitly sized
    int nPowersOf2[NBR_OF_INTS] = { 1, 2, 4, 8, 16 };
                        // Implicitly sized
    int nDays[] = { 1, 2, 3, 4, 5 };

    cout << "Within main...\nnPowersOf2 is an array of "
         << NBR_OF_INTS << " integers.\n";
    cout << "nPowersOf2's sizeof shows "
         << sizeof(nPowersOf2) << "-bytes of storage.\n";
    cout << "A "
         << sizeof(nPowersOf2) << "-byte array of "
         << sizeof(int) << "-byte integers is "
         << sizeof(nPowersOf2) / sizeof(int) << " ints.\n";
    IntArrayTotal(nPowersOf2, NBR_OF_INTS);
    cout << "Within main...\nnDays is an array of "
         << "unspecified ([]) integers.\n";
    cout << "Fortunately, sizeof shows nDays as "
         << sizeof(nPowersOf2) << "-bytes of storage\n";
    cout << "allowing the function to be called with a "
         << "second argument of \n";
    cout << "sizeof(nDays) / sizeof(int) or "
         <<  sizeof(nDays) / sizeof(int) << ".\n";
    IntArrayTotal(nDays, sizeof(nDays) / sizeof(int));
}

void IntArrayTotal(int iArray[], int iSize)
{
    int iCount, iSum = 0;
    cout << "Within a function receiving the array...\n";
    cout << "iArray's sizeof shows "
         << sizeof(iArray) << "-bytes of storage.\n";
    cout << "A "
         << sizeof(iArray) << "-byte array of "
         << sizeof(int) << "-byte integers is "
         << sizeof(iArray) / sizeof(int) << " ints.\n";
                        // Loop to total the array
    for (iCount = 0; iCount < iSize; iCount++)
        iSum += iArray[iCount];
    cout << "The sum of the array is " << iSum << endl;
}
```

Differences with Character Arrays

Slide Objective
Begin the explanation of character arrays with '\0' character implied in literal strings and required within char array processing.



```
char szBuffer[5] = "Bill";
```

```
[0] = 'B'
[1] = 'i'
[2] = 'l'
[3] = 'l'
[4] = '\0'
```

In the example on the foil, a character array is being declared. It uses the name **szBuffer**, and it allocates five bytes of storage. Note that the sz prefix indicates that this is a zero-terminated string, so the fifth character should be NULL. All literals within double quotation marks have a NULL character.

If you changed the example removing the 5, **szBuffer** would still be assigned five locations and be initialized with the characters depicted.

If you changed it again by increasing the 5 to 50, **szBuffer** would contain 45 more NULL characters.

Demo

CHARRAY.CPP is found in \DEMOS\MOD14. It examines functions that input to character arrays.

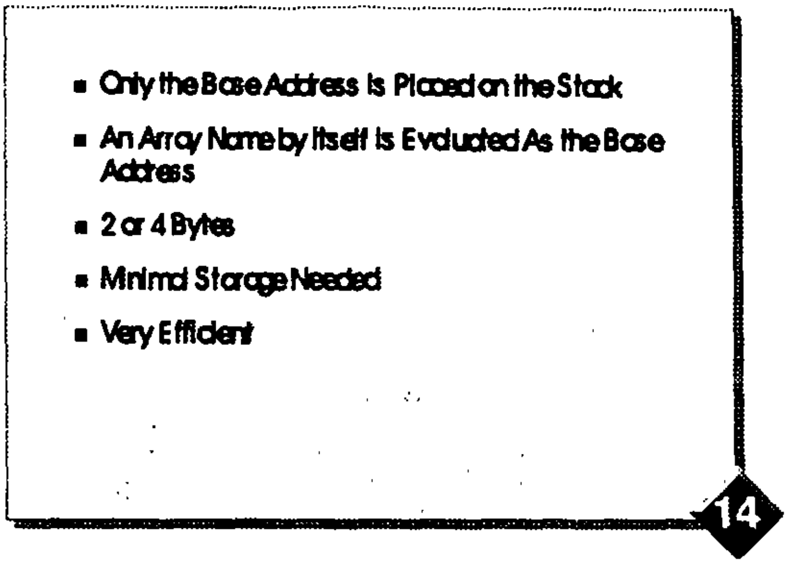
```
// CHARRAY.CPP  Found in \demos\mod14
// Managing character arrays using various iostream
// operators and functions.
#include <iostream.h>
                                // manifest constant
#define SIZE 30

/**************** Array Class Declaration ****************/
class Arrays
{
public:                         // operations
    void ByCharCinOperator();
    void ByWordCinOperator();
    void ByCinGet();
    void ByCinGetline();
    void Display()
    {
        cout << "\"" << m_chArray << "\"\n";
        cout << "           Extras \""
             << m_chExtras << "\"\n";
        m_chExtras[0] = '\0';
    }
private:                        // implementation
    char m_chArray[SIZE];
    char m_chExtras[SIZE];
};

/********** Array Member Function Definitions ***********/
void Arrays::ByCharCinOperator()
{
    cin >> m_chArray[0];
    // remove rest of chars and the newline
    cin.getline(m_chExtras, SIZE);
}
void Arrays::ByWordCinOperator()
{
    cin >> m_chArray;
    // remove rest of m_chars and the newline
    cin.getline(m_chExtras, SIZE);
}
void Arrays::ByCinGet()
{
    cin.get(m_chArray, SIZE);
    // remove rest of chars and the newline
    cin.getline(m_chExtras, SIZE);
}
void Arrays::ByCinGetline()
{
    cin.getline(m_chArray, SIZE);
}

```

(continued)

```
/****************** Simple Test Program *****************/
void main()
{
    Arrays aNames;       // default C'tor
    cout << "Enter your name (cin >> chArray[0]).\n";
    aNames.ByCharCinOperator();
    aNames.Display();
    cout << "Enter your name (cin >> chArray).\n";
    aNames.ByWordCinOperator();
    aNames.Display();
    cout << "Enter your name (cin.get(chArray, SIZE).\n";
    aNames.ByCinGet();
    aNames.Display();
    cout << "Enter your name (cin.getline(chArray, SIZE)."
        "\n";
    aNames.ByCinGetline();
    aNames.Display();
}
```

Character Arrays As Function Arguments

Slide Objective
Describe character arrays (and string literal) as arguments to functions.



- Only the Base Address Is Placed on the Stack
- An Array Name by Itself Is Evaluated As the Base Address
- 2 or 4 Bytes
- Minimal Storage Needed
- Very Efficient

Features of Functions That Take Array Arguments

Remember—the prototype specifies that an argument is an array, and that only the base address is on the stack. When you think about it, it wouldn't make much sense to physically place an entire array on the stack. The stack size is finite and limited to 2K. If an array were placed on the stack in a pass to a function, you'd quickly exhaust your stack.

Tip Except for **char** arrays (which are NULL terminated), length cannot be determined.

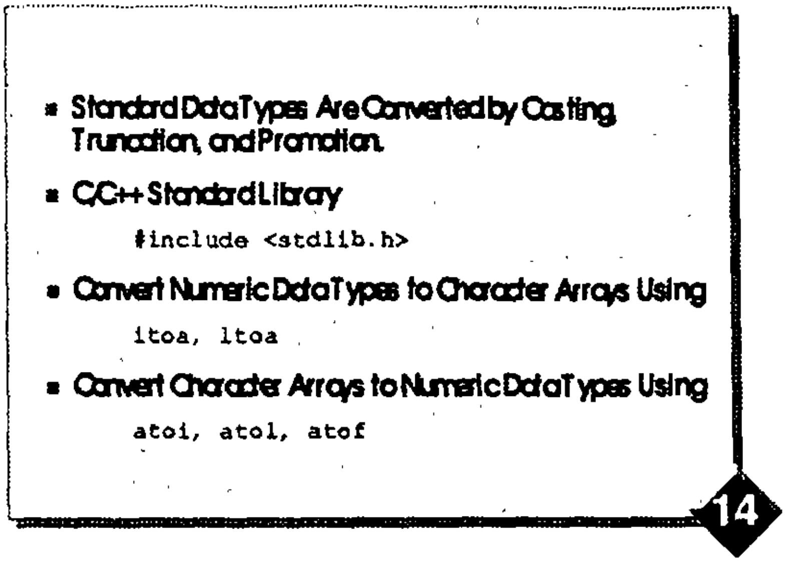
Demos

SEARCH.CPP is located in \DEMOS\MOD14. It passes an array and a character to a function that returns the number of occurrences of the character in the array.

```
// SEARCH.CPP   Found in \demos\mod14
// Passing character arrays as function arguments.
#include <iostream.h>

#define MAXLENGTH 30

int CharCount (char[], char);

void main()
{                           // an array and a char
    char chBuffer[30], chInput;
    int iLetterCount;

    cout <<   nter a line of text.\n";
    cin.ge    e(ch  uffer, MAXLENGTH);
    cout <    nter a search character: ";
    cin >>    nput
                            // array name and char name
    iLetterCount = CharCount (chBuffer, chInput);
                            // Array passed as address
                            // char passed as value
    cout << chInput << " occurred "
         << iLetterCount << " times in '"
         << chBuffer << "'." << endl;
}
int CharCount (char chSearchString[], char chLookup)
{
    int iCount = 0, nSum = 0;
    while (chSearchString[iCount] != '\0')
        if (chSearchString[iCount++] == chLookup)
            nSum++;
    return nSum;
}
```

Functions That Convert to and from Strings

Slide Objective
The numeric data types are automatically truncated or promoted to different types. Present library functions that perform those translations for character arrays.



- **Standard Data Types Are Converted by Casting, Truncation, and Promotion.**
- **C/C++ Standard Library**

  `#include <stdlib.h>`
- **Convert Numeric Data Types to Character Arrays Using**

  `itoa, ltoa`
- **Convert Character Arrays to Numeric Data Types Using**

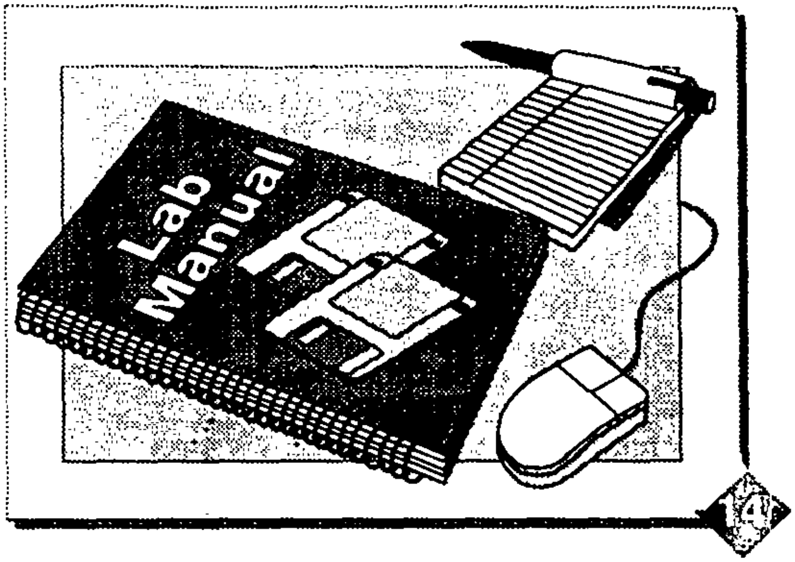
  `atoi, atol, atof`

Delivery Tips
Two functions: **itoa** and **ltoa** are needed in the lab.

To locate details on any of these functions, open any C++ file, type in any of the function names, and press F1.

Lab 12: Manipulating Arrays

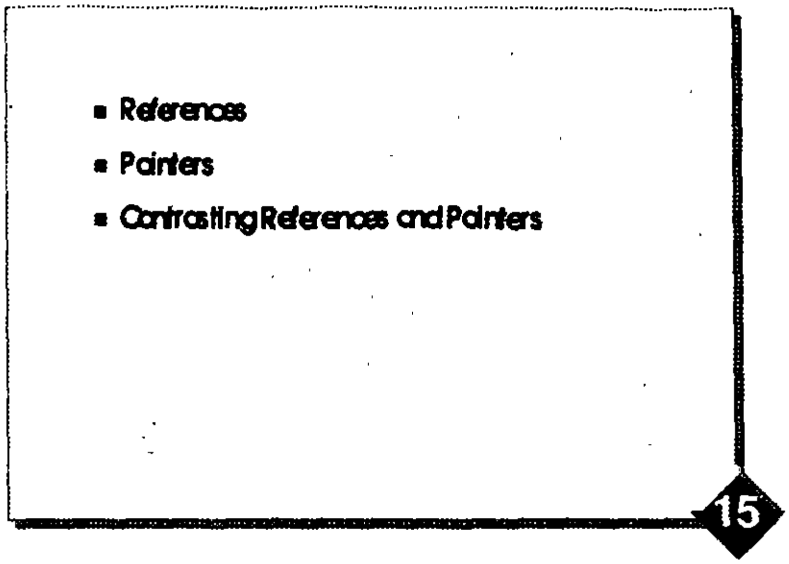
Slide Objective
Execute the lab solution.
Set the lab objectives.
Ask students to read the lab scenario.



Module 15: Working with References and Pointers

Σ Overview

Slide Objective
Provide an overview of the module contents.



- References
- Pointers
- Contrasting References and Pointers

Module Summary

In the last module, you learned to create and manipulate arrays. That makes for a good introduction to references and pointers. References are extremely easy to work with, and they add power to your applications. Though pointers are useful for manipulating the elements in an array, their value transcends simple array-manipulation. In fact, pointers are one of the most useful constructs of the C++ language.

In later modules, you will see that it is easier to use strings when you know how to encapsulate the pointer manipulations you learn in this module.

Objectives

Key Points
Present the learning objectives and set the expectation that two different (but similar and related) topics are presented.

Upon completion of this module, you will be able to:

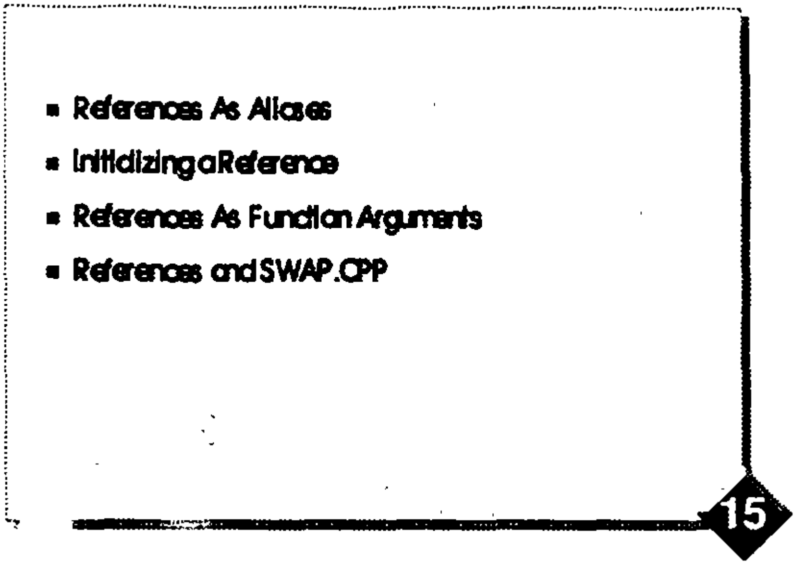
- Use references.
- Understand reference syntax.
- Understand pointer syntax.
- Pass references and pointers as function arguments.
- Manipulate strings with reference and pointer notation.

Lab

Using Pointers to Manipulate Strings

References: An Overview

Slide Objective
Loosely define references (eschew address terminology) and cite "why" programs might use them.



- References As Aliases
- Initializing a Reference
- References As Function Arguments
- References and SWAP.CPP

Key Points
References can be an alternate name for a variable or object.
References are similar to type-modifiers but do not create another variable.
Used as function arguments, references are more efficient than the default pass-by-value.

What Are References?

References are aliases for objects—that is, they are nicknames for objects. Once you have initialized a reference to an object, you can refer to the object by its alias.

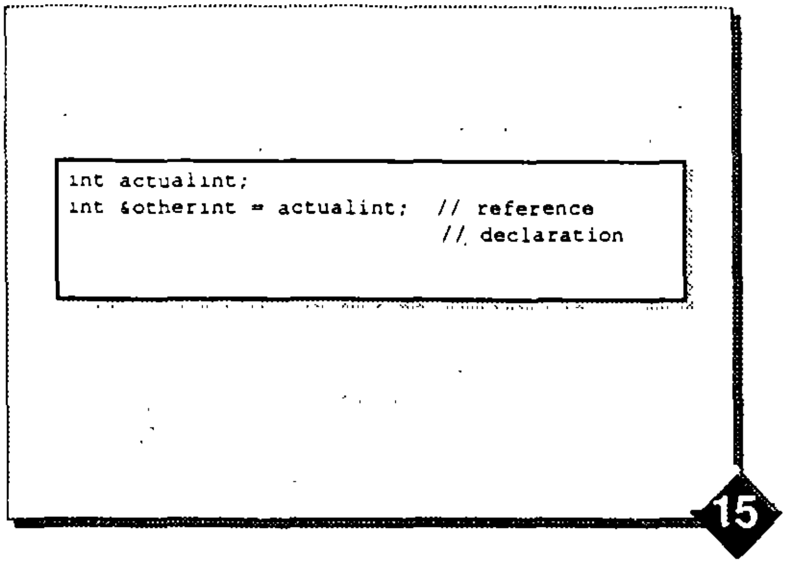
How Are References Used?

References are used primarily to pass parameters to functions and to return values back from functions. The syntax is the same for objects.

References are semantically identical to constant pointers, and they can be assigned only one value at a time. Since references can only be initialized once, there is only one way to initialize a class data member which has a reference. That is to initialize it in the constructor, using colon syntax.

References as Aliases

Slide Objective
Loosely define reference as another name for an existing variable or object.



```
int actualint;
int &otherint = actualint;  // reference
                            // declaration
```

Key Points
The "&" symbol is NOT the address-of operator. It is not any operator—it is a type-declarator. Students have not seen the "address operator" yet.

What Is a Reference?

A reference is a type declaration that creates an alias for an existing variable. Usually, a reference is initialized explicitly, giving it something to refer to when you declare it. As the foil title suggests, a reference is an alternate name for a variable—not a copy of the variable. The declaration with initialization associates the two names. What that means for you is that operations on either name have the same result. The reference becomes a synonym for the variable.

Remember that when you declare an array—such as **szBuff[100]**—the bracket characters are not operators. They are declarators that have a special meaning. The ampersand character, &, used in the declaration of a reference is not an operator. (nor is it the **address-of** operator or the bitwise-AND operator listed in the Operator Precedence chart.) References use the ampersand to identify the variable as a reference to the compiler.

References may be used any time you want to permanently associate names for a variable.

Reference

See "References" in the *C++ Tutorial.*

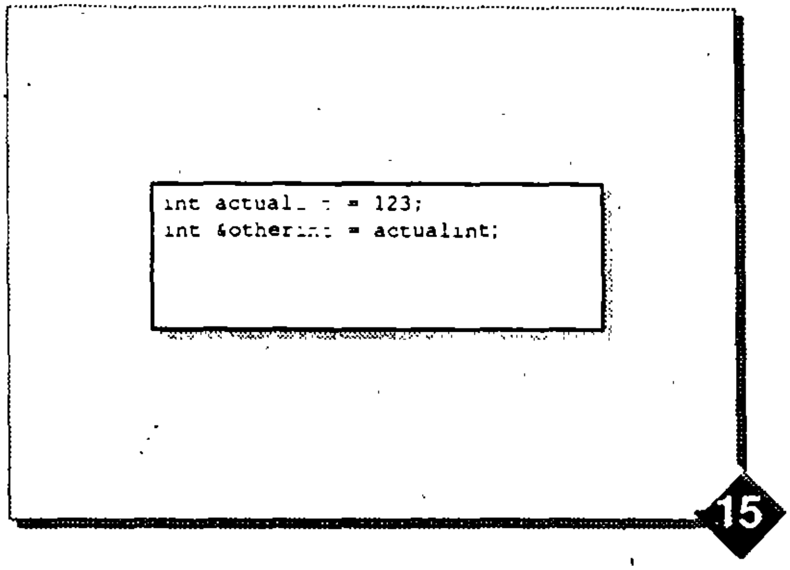
Demo

REFDEMO.CPP is found in \DEMOS\MOD15. It creates an alias and proves that it is identical to the original object.

```
// REFDEMO.CPP  found in \\demos\mod15
// Using reference notation to create an alias for
// an integer.  Usage after declaration is identical.
#include <iostream.h>

void main()
{
    int actualint = 123;       // the actual integer
    int &otherint = actualint; // the alias

    cout << actualint << endl;
    cout << otherint  << endl;
    otherint++;                // increment alias
    cout << actualint << endl;
    cout << otherint  << endl;
    actualint++;               // increment actual
    cout << actualint << endl;
    cout << otherint  << endl;
}
```

Initializing a Reference

Slide Objective
Detail how references are initialized. Note the exceptions where initialization is not required.



```
int actualInt = 123;
int &otherInt = actualInt;
```

Delivery Tips
Don't try to explain details of what the compiler does with a reference or how they work. The implementation may vary between various compilers. References are easy and they work.

Creating References

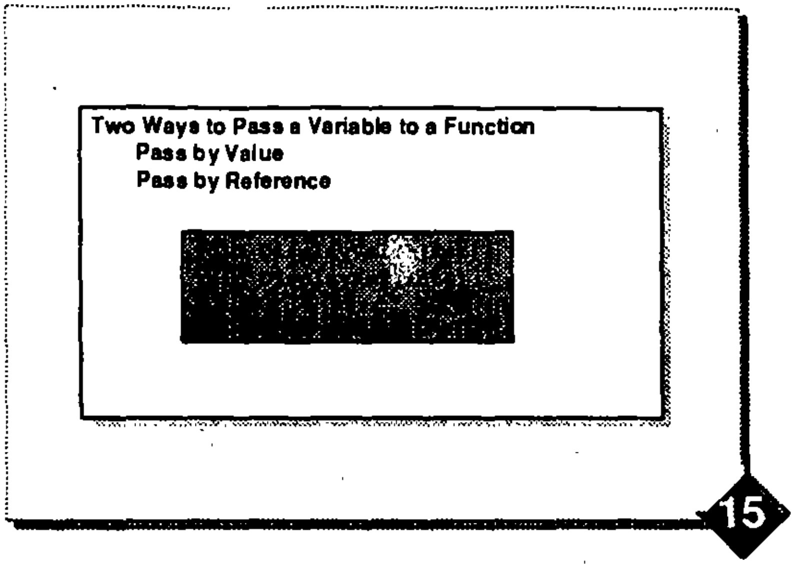
References rarely exist without a variable to which they can refer—and they cannot be manipulated as a separate entity. Once the association between a reference and a variable is set, it cannot be changed.

Not all cases require the initialization to be set at declaration. Here are some exceptions:

1. There is no need to initialize a reference if it is declared **extern** and initialized elsewhere. An **extern** reference typically would be initialized in the source file where the declaration was made.
2. If the reference is a member of a class and is initialized in a constructor.
3. If the reference is declared as a parameter and its value is established when the function is called.
4. If the reference is declared as a **return** type and is established when the function returns.

References as Functi n Arguments

Slide Objective
Describe the changes between a function that takes an integer and one that takes a reference to an integer.



Two Ways to Pass a Variable to a Function

Pass by Value

Pass by Reference

Delivery Tips
Students may be bothered by the notation: (int& a) versus (int &a).
C++ ignores whitespace so the compiler doesn't care.
The convention is:
int& a;

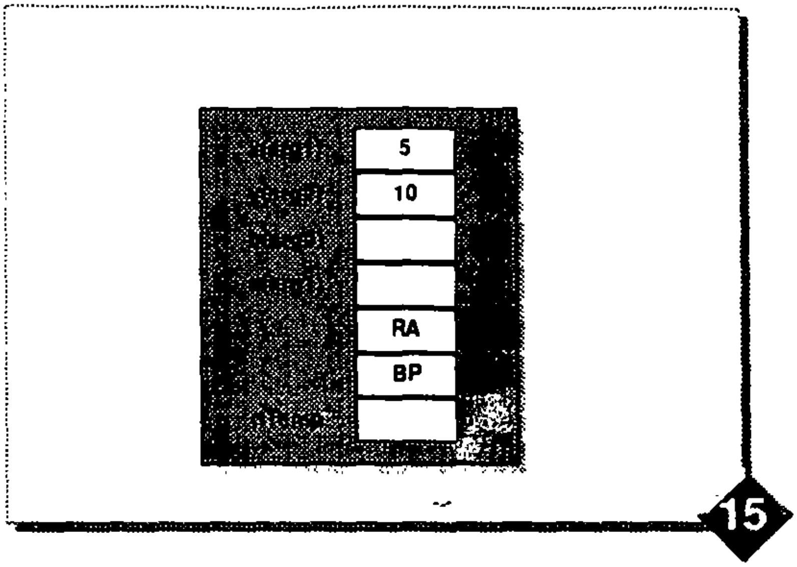
Demo

REFADDR.CPP is found in \DEMOS\MOD15. It details the declaration and initialization for references. Contrast the usage of the actual integer versus the reference both in statements and as arguments to functions.

```
// REFADDR.CPP  found in \demos\mod15
// Initializing references uses a simple variation
// on syntax.  After that, everything is easy.
#include <iostream.h>
                         // function prototype
int Add1(int&);          // call by reference
void Disp(const int&);   // call by const reference

void main()
{                        // a variable must exist
    int actualint = 123; // before the reference
                               // a reference must
    int &otherint = actualint; // be initialized
                               // to the target

    // compare standard usage of the variables
    cout << "\nComparing actualint and otherint...\n";
    cout << "  Value: " << actualint
                 << ' ' << otherint << endl;
    cout << "Address: " << &actualint
                 << ' ' << &otherint << endl;

    // compare usage as function arguments
    cout << "\nTesting Add1(int&) function...\n";
    cout << "Before call actual " << actualint << endl;
    Add1(actualint);
    cout << " After call actual " << actualint << endl;
    cout << "Before call other " << otherint << endl;
    Add1(otherint);
    cout << " After call other " << otherint << endl;

    cout << "\nTesting Disp(const int&) function...\n";
    cout << "What is the difference between\n"
          << "actualint ";
    Disp(actualint);
    cout << " and otherint ";
    Disp(otherint);
    cout << "?" << endl;
}

int Add1(int& n)         // call by reference
{   // a reference argument can be changed
    n++;
    return n;
}

void Disp(const int& n)  // call by const reference
{   // a const argument can't be changed
    cout << n;
}
```

References and SWAP.CPP

Slide Objective
Detail the activities that occur when a reference is passed to a function.



References are frequently used to pass arguments to a function or to return a value from a function. Passing by reference is much more efficient than passing by value.

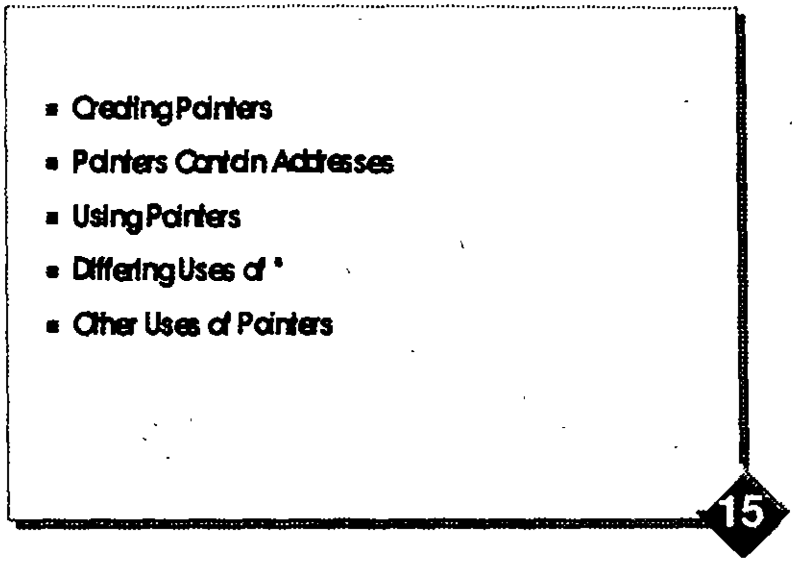
Demo

SWAPREF.CPP is found in \DEMOS\MOD15.

```
// SWAPREF.CPP   Found in \demos\mod15
// Functions that take reference arguments have
// access to the caller's data.
#include <iostream.h>
     // CHANGE #1          // function prototype
void swap(int&, int&);    // reference to integer

                          // Identical to swap.cpp
void main()
{                         // two local variables x and y
    int x (5), y (10);   // Note: equivalent to:
                         //       int x = 5, y = 10;
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
    swap (x, y);          // function call
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
}
     // CHANGE #2
void swap(int &a, int &b)// Now takes references
{                         // as arguments
    int temp;
                          // same as before!!
    temp = a;
    a = b;
    b = temp;
}
```

Pointers: An Overview

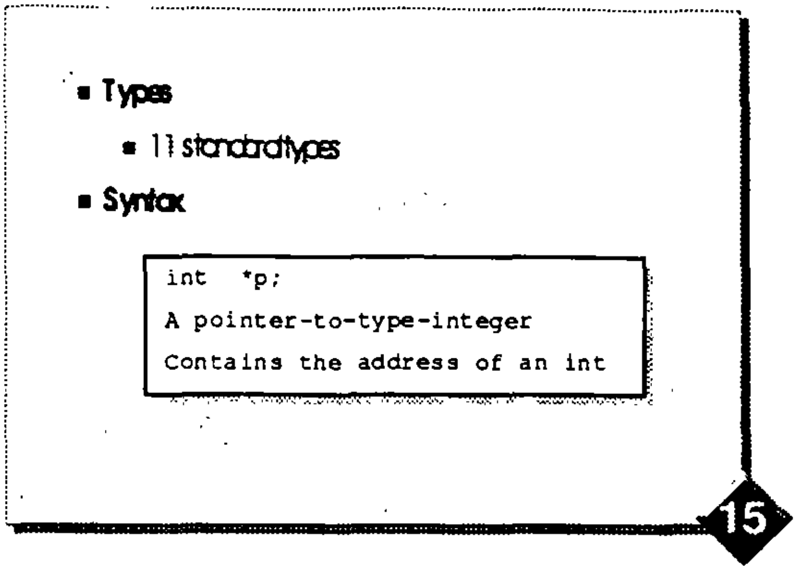
Slide Objective
Provide an overview of pointers with an introductory definition of addresses. Cover "why" you would use pointers, including features and benefits.
The following pages add details to the points listed.



- Creating Pointers
- Pointers Contain Addresses
- Using Pointers
- Differing Uses of *
- Other Uses of Pointers

Creating Pointe

Slide Objective
Cover pointers to standard data types. Each of the standard types has a pointer type associated with it.



- Types
  - 11 standard types
- Syntax

```
int  *p;
A pointer-to-type-integer
Contains the address of an int
```

Key Points
There are int pointers, float pointers, etc.

There are no generic pointers. A void pointer can only serve as a bucket to hold something of unspecified type; they can't be directly used.

Types of Pointers

There is a pointer type for each of the C/C++ standard data types. Thus, you will create and use an **int** pointer for working with integers, a **char** pointer for working with characters, and so on.

What Isn't Covered Here

C supports a special, generic type of pointer called a **void** pointer. The uses and implications of these are discussed later in this module. In another module, you learned how to define your own data types. User-defined types can also have their own pointers. (This issue is covered in another module.) Finally, you can have pointers that point to functions. That is an advanced topic that is not covered in this course.

Features of Pointers

Pointer variables have to be created, just like other variables.

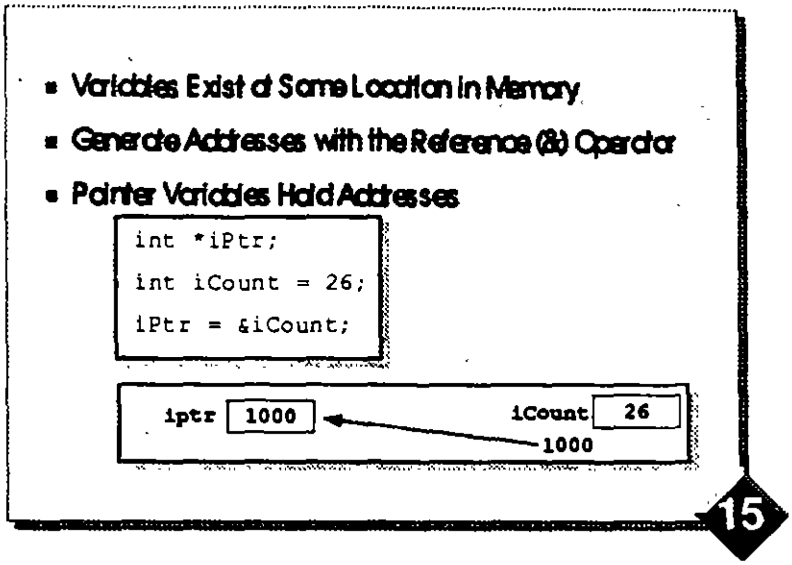
The asterisk in a declaration statement makes the variable that follows it a pointer. The * does not have the same meaning as the multiplication or the dereferencing operator. The example in the foil creates an integer pointer. You might say that p is a variable that is capable of pointing to an integer.

It's important to recognize that in the declaration above, the pointer does not currently point to anything. As you learned earlier with the built-in data types, creating space doesn't mean that anything is assigned to that space yet. It is important to stress that even though the pointer is capable of pointing, it doesn't point to anything yet.

Pointers, like other variables in C programs, can be automatic local, static local, or global in scope.

Pointers Contain Addresses

Slide Objective
Cover the use of the ampersand character, the address-of operator, to set a pointer.



- Variables Exist at Some Location in Memory
- Generate Addresses with the Reference (&) Operator
- Pointer Variables Hold Addresses

```
int *iPtr;
int iCount = 26;
iPtr = &iCount;
```

Key Points
Cover the three statements above in the sequence presented. Use pointer and address terminology.

Sequence

The three lines of code in the foil are interpreted as follows:

- `iPtr` is a pointer to a type integer.
- `iCount` is an integer initialized to 26.
- Assign the address of `iCount` to the `int` pointer `iPtr`.

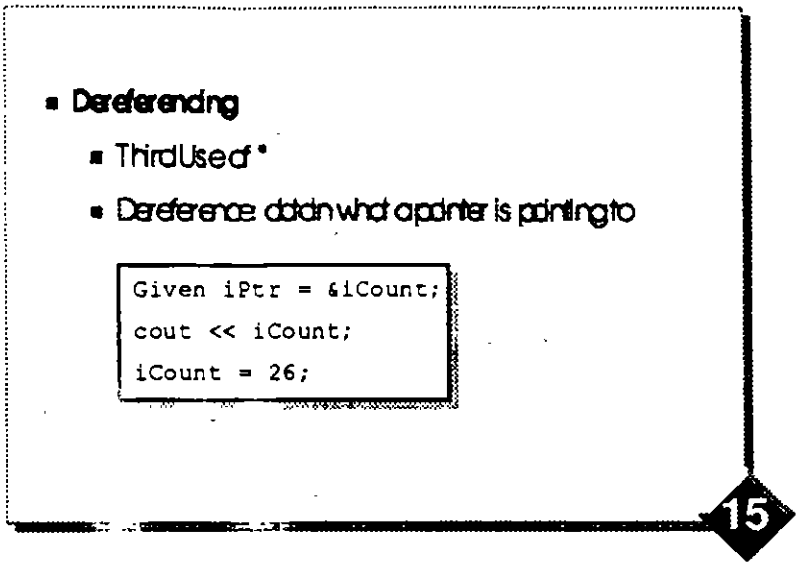
In algebra, the equal sign(=) is much like a balance scale: the two sides of an equation must balance. For instance, 8 + 8 = 16. The same is true, generally, of computer languages like C. The type on the left must be same as the type on the right. In the statement iPtr = &iCount, this is true. On the left is a pointer variable that can hold an address of an **int**. On the right, the **& operator** generates the address of an integer. The two sides balance.

We have seen that there are two uses for the asterisk as a token in the C language: as the multiplication operator, and as the pointer-creation operator in a declaration statement.

There's a third use of the asterisk, as you'll see next.

Using Pointers

Slide Objective
Dereferencing a pointer gives the variable pointed to by the pointer.



- **Dereferencing**
  - Third Use of *
  - Dereference: obtain what a pointer is pointing to

```
Given iPtr = &iCount;
cout << iCount;
iCount = 26;
```

Dereferencing

An asterisk is a dereferencing operator if it is placed before a pointer variable in executable code.

What Is a Dereferencing Operator?

When placed before a pointer variable in an executable statement, the asterisk generates an instruction to look (through the pointer) to the address that the pointer contains. Dereferencing an integer pointer obtains an integer; dereferencing a double pointer obtains a double, and so on. Use of a pointer is called "indirection."

Key Points
The processing depicted uses the dereference operator to assign 26 "where the pointer, iPtr, points to."

In the foil example, you see that a dereferenced pointer variable can be used as both an rvalue and an lvalue. When you use a dereferenced pointer as an lvalue, the original value is changed like this:

```
*iPtr = 26;
cout << iCount;
```

This prints out 26.

`*iPtr` is translated as "the contents stored at the address `iPtr` holds"

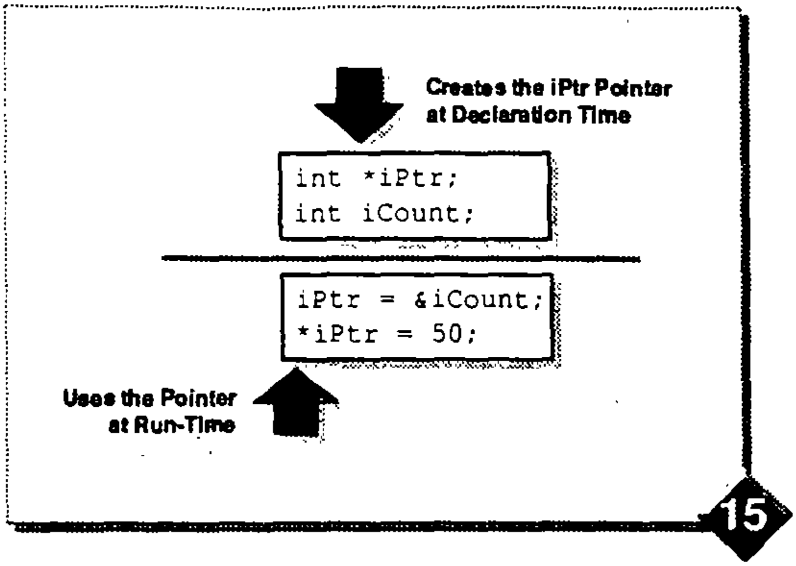
Demo

POINT1.CPP is located in \DEMOS\MOD15. This demo ties a pointer to an integer and compares the syntax for variables and addresses to that of pointers and dereferences.

```
// POINT1.CPP   Found in \demos\mod15
// Creating pointers and working with pointer notation.
#include <iostream.h>

void main()
{   // '*' used in a declaration denotes a pointer variable
    // (This * is not multiplication and not dereferencing.)
    int *iPtr;   // iPtr is a pointer to data-type integer
    int iCount = 26;
    // set the pointer to point to a variable
    iPtr = &iCount;      // address-of '&' assigns address

                         // iCount == *iPtr
    cout << "  iCount = " << iCount << endl;
    cout << "   *iPtr = " << *iPtr  << endl;

                         // &iCount == iPtr
    cout << " &iCount = " << &iCount << endl;
    cout << "    iPtr = " << iPtr    << endl;

                         // just for fun...
    cout << "   &iPtr = " << &iPtr          << endl;
    cout << " *iCount = " << *(int *)iCount << endl;
}
```

Differing Uses of *

Slide Objective
Explain the distinction between the use of "*" in a definition statement where the pointer is created versus an executable statement where the pointer is dereferenced. Only "dereference" is an operator.



Creates the iPtr Pointer at Declaration Time

```
int *iPtr;
int iCount;
```

```
iPtr = &iCount;
*iPtr = 50;
```

Uses the Pointer at Run-Time

Delivery Tips
Draw a similarity that the "&" used to declare a reference is like the "*" for a pointer.
NEITHER IS AN OPERATORS.

```
int *iPtr;
int iCount = 26;
*iPtr = &iCount;    // wrong
Ptr = &iCount;      // right
iPtr = 50;          // right
```

Demo

POINT2.CPP is located in \DEMOS\MOD15. This demo compares the syntax for variables and addresses to that of pointers and dereferencing. It also shows various ways a pointer can be used to manipulate an array of integers.

```
// POINT2.CPP  Found in \demos\mod15
// Contrast 5 different methods to total an array
// of integers.  The last 3 use an integer pointer.
#include <iostream.h>

int iSum1, iSum2, iSum3, iSum4, iSum5;
int nSales[] = { 26, 18, 31, 22, 35 };

void main()
{
    int *iPtr, iIndex;
                                // calculate the size of the
                                // array (portable src code)
    int iSize = sizeof (nSales) / sizeof (*nSales);

    // Method 1: traditional array notation
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum1 += nSales[iIndex];

    // Method 2: use the array name as a pointer
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum5 += *(nSales + iIndex);

    // Method 3: "scale" off the pointer
    iPtr = nSales;              // equivalent to = &nSales[0]
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum3 += *(iPtr + iIndex);

    // Method 4: subscript off the pointer
    iPtr = nSales;
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum4 += iPtr[iIndex];

    // Method 5: "walk" the pointer
    iPtr = nSales;
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum2 += *iPtr++;

    cout << "Any way you look at it, the sum of the "
         << iSize << " weekly\n";
    cout << "sales numbers is: " << iSum1 << ", "
         << iSum2 << "," << iSum3 << ", " << iSum4
         << ", and " << iSum5 << endl;
}
```

Demo

POINT2.CPP is located in \DEMOS\MOD15. This demo compares the syntax for variables and addresses to that of pointers and dereferencing. It also shows various ways a pointer can be used to manipulate an array of integers.

```
// POINT2.CPP  Found in \demos\mod15
// Contrast 5 different methods to total an array
// of integers.  The last 3 use an integer pointer.
#include <iostream.h>

int iSum1, iSum2, iSum3, iSum4, iSum5;
int nSales[] = { 26, 18, 31, 22, 35 };

void main()
{
    int *iPtr, iIndex;
                                // calculate the size of the
                                // array (portable src code)
    int iSize = sizeof (nSales) / sizeof (*nSales);

    // Method 1: traditional array notation
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum1 += nSales[iIndex];

    // Method 2: use the array name as a pointer
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum5 += *(nSales + iIndex);

    // Method 3: "scale" off the pointer
    iPtr = nSales;          // equivalent to = &nSales[0]
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum3 += *(iPtr + iIndex);

    // Method 4: subscript off the pointer
    iPtr = nSales;
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum4 += iPtr[iIndex];

    // Method 5: "walk" the pointer
    iPtr = nSales;
    for (iIndex = 0; iIndex < iSize; iIndex++)
        iSum2 += *iPtr++;

    cout << "Any way you look at it, the sum of the "
         << iSize << " weekly\n";
    cout << "sales numbers is: " << iSum1 << ", "
         << iSum2 << "," << iSum3 << ", " << iSum4
         << ", and " << iSum5 << endl;
}
```

Demos

POINT3.CPP is located in \DEMOS\MOD15. It shows three versions of a string copy routine. This is where pointers to character arrays are most efficient.

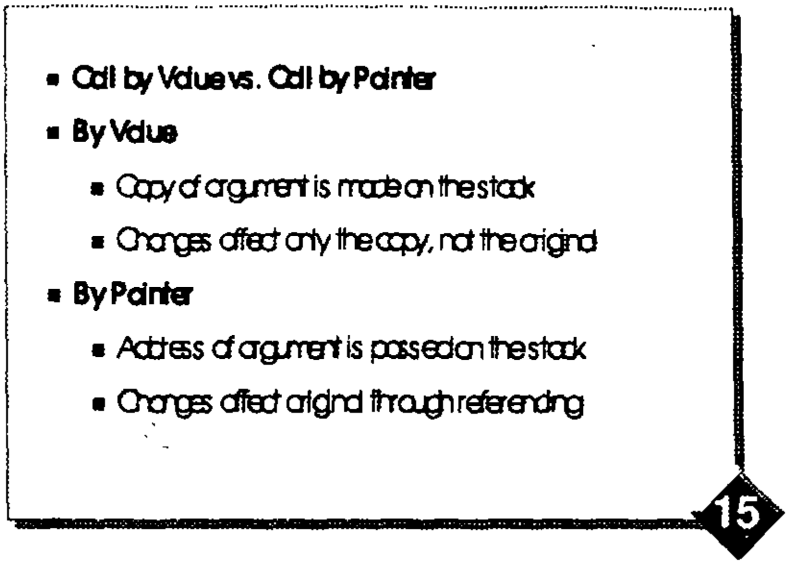
```
// POINT3.CPP  Found in \demos\mod15
// Contrast three ways to pass arrays of characters
// to functions.
#include <iostream.h>

// Use [] or *, it's all the same in a prototype
void my_strcpy1(char [], char []);
void my_strcpy2(char *, char *);
void my_strcpy3(char *, char *);

char szBuff[] = "An array is always passed"
                " by reference.\n";

void main()
{
    char szBuff1[100], szBuff2[100], szBuff3[100];

    my_strcpy1(szBuff1, szBuff);
    cout << szBuff1;
    my_strcpy2(szBuff2, szBuff1);
    cout << szBuff2;
    my_strcpy3(szBuff3, szBuff2);
    cout << szBuff3 << endl;
}

// Method 1: traditional array notation.
void my_strcpy1 (char szDest[], char szSource[])
{
    int i;
    for (i = 0; szSource[i] != '\0'; i++)
        szDest[i] = szSource[i];
    szDest[i] = '\0';
}

// Method 2: shrink the code
void my_strcpy2 (char *szDest, char *szSource)
{
    int i = 0;
        // loop stops after NULL assignment occurs
    while (szDest[i] = szSource[i])
        i++;
}

// Version 3, increment the pointers
void my_strcpy3 (char *szDest, char *szSource)
{       // loop stops after NULL assignment occurs
    while (*szDest++ = *szSource++);
}

// Note: The "while" loops in Methods 2 and 3 may //
// generate warning messages from your compiler.  //
// That's good.  I'd want to be warned about that //
// unexpected location of an assignment.   - Ed   //
```

SWAI R.CPP is located in \DEMOS\MOD15. It shows how to make the sw functic .wap by passing addresses and using pointers.

```
// SWAPPTR.CPP   Found in \demos\mod15
// Functions that take pointer arguments have
// access to the caller's data.
#include <iostream.h>
     // CHANGE #1          // function prototype
void swap(int *, int *);// swap is a function that
                           // takes int ptr arguments
void main()
{                          // two local variables x and y
    int x (5)     (10);    // Note: equivalent to:
                           //       int x = 5, y = 10;
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
    swap(&x, &y); // CHANGE #2 &address of integers
    cout << "X is " << x;
    cout << " and Y is " << y << endl;
}
     // CHANGE #3
void swap(int *a, int *b) // Now takes pointers
{                          // as arguments
    int temp;
     // CHANGE #4 Must dereference ptrs to get values
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Contrasting References and Pointers

Slide Objective
Starting with call by value, begin a contrast of Pointers vs. References. The graphic on the following page will assist the contrast.



- Call by Value vs. Call by Pointer
- By Value
  - Copy of argument is made on the stack
  - Changes affect only the copy, not the original
- By Pointer
  - Address of argument is passed on the stack
  - Changes affect original through referencing

When to Call by Pointer

You should call by pointer when a function argument must be modified in the function and/or it takes up a lot of space. Space is an issue because an argument passed by value will be pushed onto the stack. Suppose you have a 1000-byte structure. Every time you pass it by value to a function, 1000 bytes will copied over to the stack. This will be time-consuming.

Demo

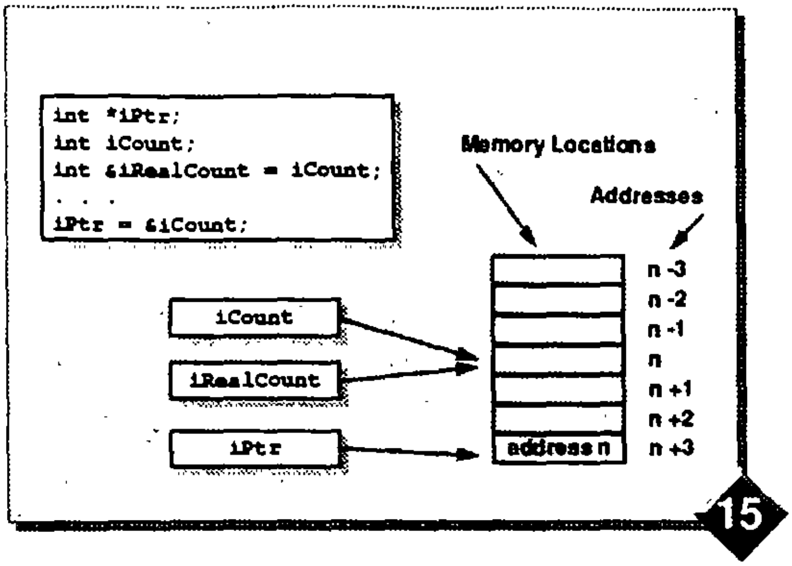
REFPARAM.CPP is found in \DEMOS\MOD15.

Note the use of the asterisk and the ampersand as well as the use of the **const** keyword in the prototypes.

```
// REFPARAM.CPP   found in \demos\mod15
// Contrast three ways to pass arguments to functions.
// (Note: Pointers will be covered next.)
#include <iostream.h>
        // structure definition and declaration, bo
struct bigone
{
    int nbr;
    char text[1000];  // space for a lots of char's
} bo = {123, "This is a big structure" };

                            // function prototypes
void valfunc(big   );             // call by value
void reffunc(con   bigone&);     // call by reference
void ptrfunc(con   bigone *);    // call by pointer

/****************  Small Test Program ******************/
void main()
{
    valfunc(bo);       // passing the bo values
    reffunc(bo);       // passing a reference to bo
    ptrfunc(&bo);      // passing the address of bo
    cout << endl;
}

void valfunc(bigone v1)          // pass by value
{
    cout << '\n' << v1.nbr;      // "." dot operator is
    cout << '\n'    v1.text;     // member of notation
}

void reffunc(con   bigone& r1)  // pass by reference
{
    cout << '\n'   r1.nbr;      // reference notation
    cout << '\n'  < r1.text;     // same as member of
}

void ptrfunc(const bigone *p1)  // pass by pointer
{
    cout << '\n' << p1->nbr;    // "->" pointer to
    cout << '\n' << p1->text;   // struct member notation
}
```

References and Pointers

Slide Objective
Contrast the processing that occurs during the declaration and assignment of Refs and Ptrs.
Note the opportunities for errors or typos.
Note the additional storage for a pointer.

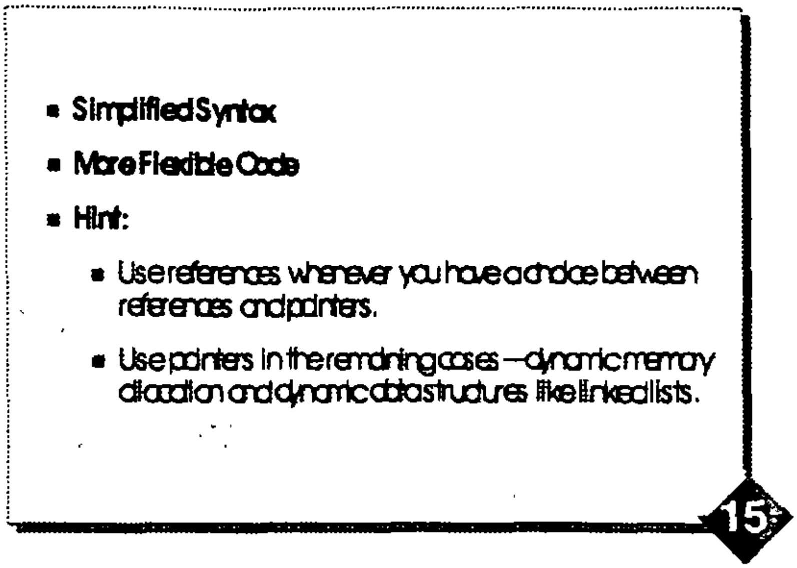


```
int *iPtr;
int iCount;
int &iRealCount = iCount;
. . .
iPtr = &iCount;
```

Put graphically, the contrast of pointers to references would look like the above.

Advantages of References Over Pointers

Slide Objective
Summarize the Refs versus Ptrs contrast.

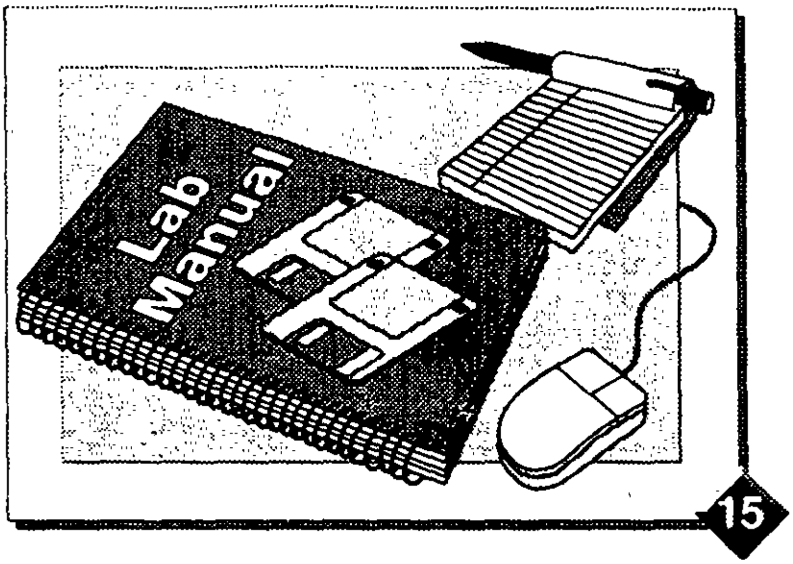


- **Simplified Syntax**
- **More Flexible Code**
- **Hint:**
  - Use references whenever you have a choice between references and pointers.
  - Use pointers in the remaining cases—dynamic memory allocation and dynamic data structures like linked lists.

References give you more flexibility because you can easily change back and forth between passing and returning by value and by reference. Only the function prototype and header must be touched. By contrast, when you use pointers, you must also touch the function call and the function body.

Lab 13: Using Pointers to Manipulate Strings

Slide Objective
Execute the lab solution.
Set the lab objectives.
Ask students to read the lab scenario.

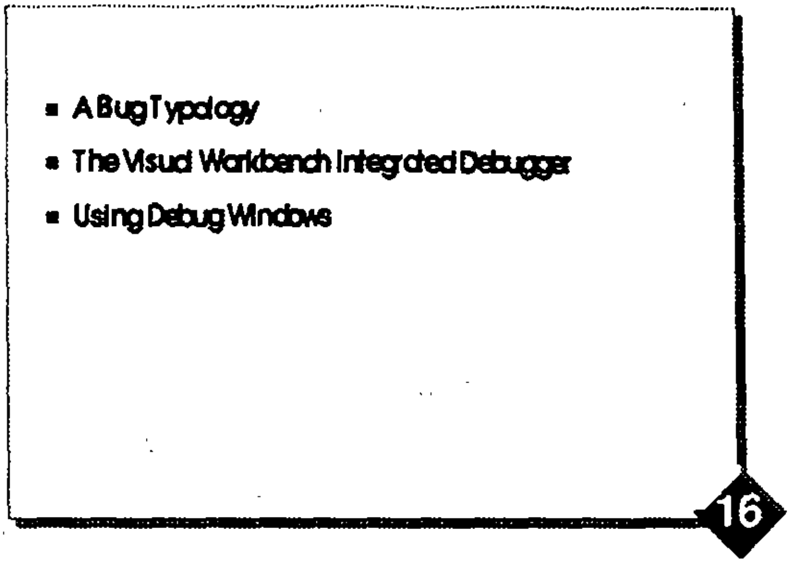


Key Points
The loop to copy characters from one string to another would appear not to work if:
1) later code places the NULL character in the wrong location,
or
2) there is an "off by one" error starting the copy loop.

Module 16: Using the Debugger

∑ Overview

Slide Objective
Provide an overview of the module contents.



- A Bug Typology
- The Visual Workbench Integrated Debugger
- Using Debug Windows

Some people define a bug as any shortcoming that a program might have. Others define a bug as incorrect operation. There's room for interpretation between these two definitions. For example, would you say a program that runs too slow has a bug?

In this module we'll restrict our scope to those bugs which arise either from incorrect use of the language or some flaw in the basic logic of the program.

Module Summary

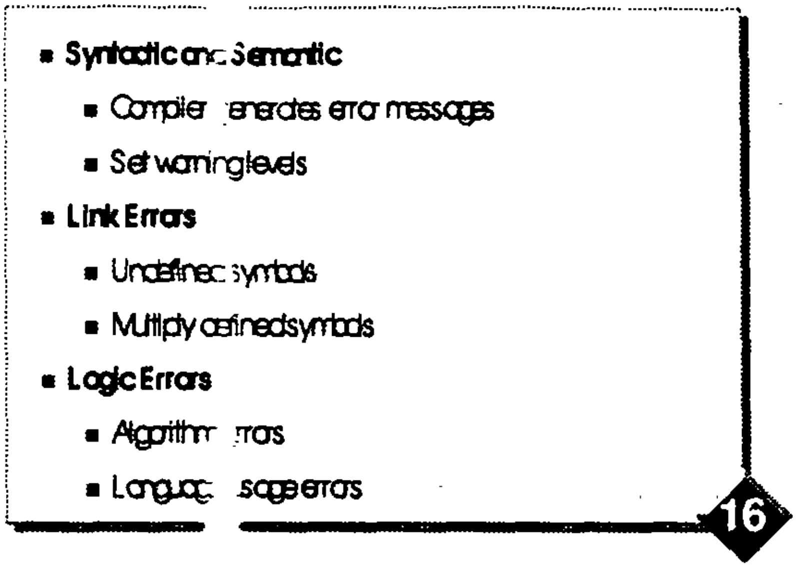
Continuing on the theme of important programming skills, you will now learn to use the debugger. In the demo you will be given a sample program that has a number of errors embedded into its code. You will use the features of the Visual Workbench debugger to find and eradicate them. And while we strive to provide you with non-trivial examples, you will still need to gain real-world experience before you can fully appreciate how and when to apply the debugger.

Objective

Upon completion of the module, you will be able to use the features of the Visual Workbench integrated debugger.

A Bug Typology

Slide Objective
Loosely, a bug includes all these errors. Developers need to get through the first 2 areas and have the .EXE in order to use the debugging tools.



- **Syntactic and Semantic**
  - Compiler generates error messages
  - Set warning levels
- **Link Errors**
  - Undefined symbols
  - Multiply defined symbols
- **Logic Errors**
  - Algorithm errors
  - Language usage errors

Errors Caught by the Compiler

A syntax error is caused by miscoding a statement. You've probably encountered a number of them by now: a missing semicolon, a parenthesis out of place, a misspelling, and so on. The compiler finds these and alerts you quickly. Semantic errors, on the other hand, are a little more complex. They occur when you have obeyed the grammatical rules of the C++ language, but have done something nonsensical—multiplied a pointer by an integer, for example. On the surface, this looks like one variable multiplied by another, but the compiler knows that a pointer can't be multiplied by a number meaningfully. The compiler would generate a compile-time error message, and you would have to remedy the situation before the program would build.

Errors Caught by the Linker

The linker's job is to find and incorporate all the external references your program makes. It generates an error message if it either can't find a symbol (function name, class name, or global variable) it needs to resolve, or if the symbol is defined more than once. Again, you would receive some sort of message stating the problem.

Logic Errors

Logic errors can be very tricky. Let's say you have created utterly intelligible code. It compiles and links without incident, but it doesn't do what you want it to. The culprit is generally found in two types of logic error: 1) either you've used the wrong algorithm—or miscoded it, or 2) you have inadvertently composed an entity that destroys itself (or something else important). For example, you might have accidentally indexed off the end of an array. You might have created a *wild pointer* that is happily corrupting things it shouldn't touch. You might be dividing by zero —either through a truncation or a convoluted calculation. Remember, run-time errors (generally logic errors) may or may not be accompanied by error messages. This is compiler-dependent. The C++ language does not require run-time errors to be scouted out by the compiler.

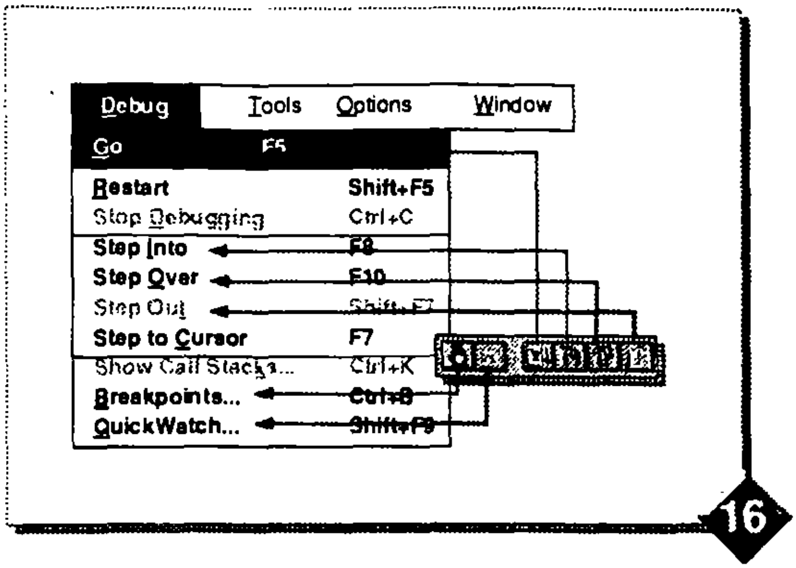
Key Points
Remind students to use F4 to match code-lines with error and warning msgs. Also use F1 for additional assistance.

Tip The first two categories of bugs are dealt with in a very straightforward way: The compiler points to the offending syntax and you search down the problem.

Logic errors are not like this. Often you want to jump immediately into the debugger to solve logic errors. Don't. Take a moment to carefully read over your code and see if the problem isn't apparent. If the problem's not apparent, you may be able to at least formulate a hypothesis that you can test by using the debugger. You will probably want to invoke the debugger, however, if you have pointer or dynamic memory errors.

The Visual Workbench Integrated Debugger

Slide Objective
Present high-level interface for starting the VWB debugger.



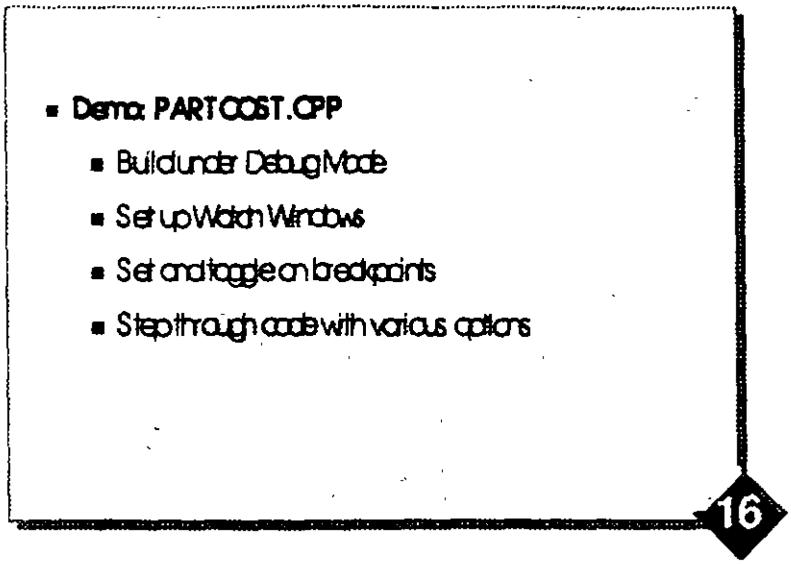
Visual Workbench has an integrated debugger that is accessible from either the Debug menu or the toolbar. (The control mapping is shown above.) If you need more information about how the debugger is controlled, go to the Help menu and chose the Visual Workbench option. Visual Workbench Help provides information on the toolbar and shortcut keys, a narrative introduction to debugging your application, and a discussion about to provide build information to the debugger.

With the debugger, you can step through your program's statements a variety of ways. You can place breakpoints in your code and toggle them on and off. You can see how the values of variables change as your program executes. You can also see the values placed in the CPU's registers (though this is a bit outside the scope of this course).

Note The Visual C++ Professional Edition also includes the Microsoft Code View debugger if you prefer to use it.

Using Debug Windows

Slide Objective
Depending on student experience with Windows and debugging tools,
1) lead students through the exercise
2) get them started, or
3) turn them loose to complete the exercise.



- Demo: PARTCOST.CPP
  - Build under Debug Mode
  - Set up Watch Windows
  - Set and toggle on breakpoints
  - Step through code with various options

Working with the Debugger: A Walkthrough

Preface Concerning Conventions

As you progress through this exercise, you'll discover that the Microsoft Visual Workbench offers multiple methods for controlling the debug session. The instructions listed below progress through three different methods: using menu options, using function or control keys, and using the toolbar buttons. (This exercise generally ignores most accelerator keys.) After completing the exercise, take time to practice whichever method is most comfortable and efficient for you.

Instructions

Before you start this exercise, you should understand what the application does. It is very similar to the inheritance lab you completed earlier.

Σ **To open the file PARTCOST.CPP**

1. Start MS Visual C++ and make sure any open projects or files are closed.

   To close a file, choose Close from the File menu.

   To close a project, choose Close from the Project menu.
2. From the File menu, choose Open.

   The Open File dialog box appears.
3. In the Directory box, select the \DEMOS\DEBUG subdirectory.

   PARTCOST.CPP will appear in the File box.
4. In the File box, select the filename PARTCOST.CPP.
5. Choose the OK button.

Σ **To set Visual Workbench to build a non-debug .EXE file**

Run this to see what the application does.

1. From the Options menu, choose Project.

   The Project Options dialog box appears.
2. In the Project Type box, select QuickWin application (.EXE).
3. Under Build Mode, select the Release option button.
4. Choose the OK button.

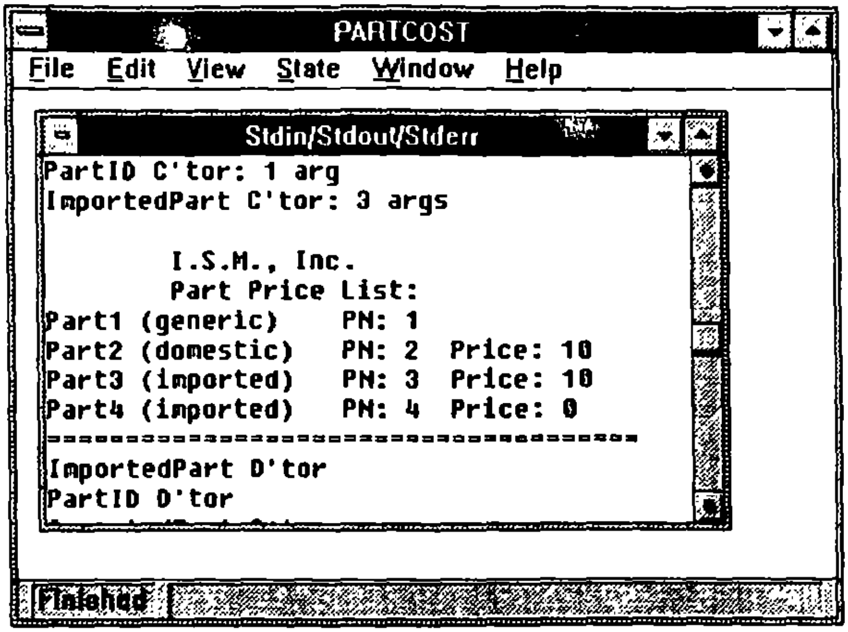
Σ **To build PARTCOST.EXE**

1. From the Project menu, choose Build PARTCOST.EXE.
2. Assuming PARTCOST compiled and linked with no warnings or errors, use CTRL+F4 to close the compiler output window.

Σ To start PARTCOST from Visual Workbench

1. From the Project menu, choose Execute PARTCOST.EXE.

   You'll see this output:



```
PartID C'tor: 1 arg
ImportedPart C'tor: 3 args

       I.S.M., Inc.
       Part Price List:
Part1 (generic)    PN: 1
Part2 (domestic)   PN: 2  Price: 10
Part3 (imported)   PN: 3  Price: 10
Part4 (imported)   PN: 4  Price: 0
======================================
ImportedPart D'tor
PartID D'tor
```

   PARTCOST.EXE created three PartID objects and displayed their values.

2. Use CTRL+C to close the PARTCOST output window.

   The current build has *not* been compiled for debugging.

   Note its size here: __________.

Compiling for MS Visual Workbench

Σ To recompile PARTCOST.CPP for Visual Workbench debugging

1. From the Options menu, choose Project.

   The Project Options dialog box appears. Do not change the Project Type; leave it as QuickWin application (.EXE).

2. Under Build Mode, select the Debug option button.
3. Choose the OK button.

Σ To build PARTCOST.EXE

1. From the Project menu, choose Build PARTCOST.EXE.

   A dialog box appears, asking you to confirm that you wish to build the affected files.

2. Choose the Yes button.

   Note the new size of PARTCOST.EXE here: ___________.

Two or more important compiler options were changed for this build. The /Od option suppresses optimization and the /Zi option inserts debugging information into the .EXE file.

3. Assuming PARTCOST compiled and linked with no warnings or errors, use CTRL+F4 to close the output window.

Starting Debugging in MS Visual Workbench

Σ To start a debug session with Go

1. From the Debug menu, choose Go.

   PARTCOST runs to completion. Note that the output is identical to the execution results you have already seen.

   Use the Control menu (the icon in the upper-left corner of the PARTCOST window that looks like a miniature spacebar) as follows.

2. From the Control menu, choose Close.
3. Close the process-termination message box by choosing the OK button.

Σ To Restart the debug session

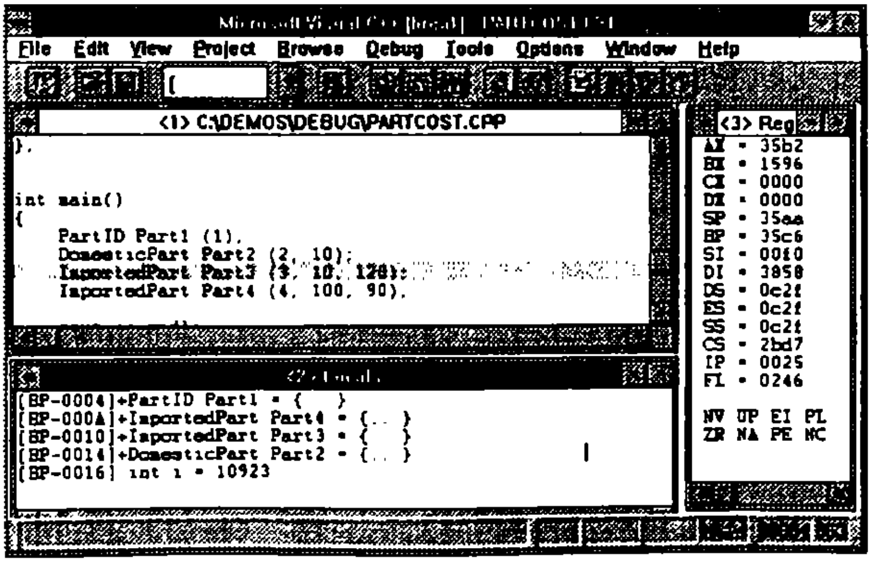
- From the Debug menu, choose Restart.

Controlling Multiple Windows in Visual Workbench

As MS Visual Workbench restarts, the Source window appears. Many other windows are available to view the execution of the application. One of the most useful is the Locals window.

Σ To open the Locals window

1. From the Window menu, choose Locals.
2. Arrange the two windows so that both are visible. (Choose Tile from the Window menu, or select, size and move them yourself.)



```
}.


int main()
{
    PartID Part1 (1),
    DomesticPart Part2 (2, 10);
    ImportedPart Part3 (3, 10, 120);
    ImportedPart Part4 (4, 100, 90),
```

```
[BP-0004]+PartID Part1 = {   }
[BP-000A]+ImportedPart Part4 = {...}
[BP-0010]+ImportedPart Part3 = {   }
[BP-0014]+DomesticPart Part2 = {...}
[BP-0016] int i = 10923
```

```
AX = 35b2
BX = 1596
CX = 0000
DX = 0000
SP = 35aa
BP = 35c6
SI = 0010
DI = 3858
DS = 0c2f
ES = 0c2f
SS = 0c2f
CS = 2bd7
IP = 0025
FL = 0246

NV UP EI PL
ZR NA PE NC
```

Using Function Keys

Σ **To single-step through a procedure using function keys**

1. From the Debug menu, choose Step Into to get through the startup code and into the `main` function.

   The first executable line of `main` is highlighted, and the function's local variables appear in the Locals window. The variables displayed in the Locals window change every time you move from one function to another. The incoming parameters to a function and auto variables are shown in the Locals window.

Everything you'll need to do in MS Visual Workbench can be done with the function keys, the mouse, keystroke combinations, or the toolbar (below the menu bar). You'll explore all of them in this exercise.

Here's what the function keys do:

F1	Help
F2	N/A
F3	Find
F4	Next error
F5	GO! Execute to end of program or next breakpoint
F6	Switch windows
F7	Execute up to the line the cursor is on
F8	Single-step and trace into user-written functions
F9	Toggle breakpoint on the current line
F10	Single-step, but don't trace into user-written functions (They are executed, however.)

Using mouse options

- The left mouse button makes the current window the active window. It's thus similar to F6, but faster. It also chooses menu items in the normal fashion.
- Double-clicking the left mouse button in a line selects the closest word to the mouse pointer. (It does not toggle a breakpoint, as in MS CodeView.) This is useful when selecting a variable for a Watch window.

Stepping Through a Program

Σ **To step or trace through a program**

1. From the Debug menu, choose Step Into.

   MS Visual Workbench has executed one line of the code listed in the Source window. Execution goes to the 1-argument constructor for the `PartID` class.

2. Press F8.

   MS Visual Workbench has executed one more line. Which step was easier for you?

3. Continue pressing and watch the program trace.

   MS Visual Workbench is executing one line of code in the Source window. The selected line is the next line to execute. Notice that the variables in the Locals windows are updated as they are assigned new values and as execution enters various functions.

4. Restart the program by pressing SHIFT+F5. (Compare this method to that of using the mouse or menu items.)

5. Perform the following steps:

   a. Press F8 five times. The cursor should be on the declaration of the `DomesticPart` object, `Part3`.

   b. Press F8 five times more. Execution has created the base object, `PartID` with a value of 2, and execution is back to the two-argument constructor for the `DomesticPart`. Note that there's a new set of variables in the Locals window.

   c. Continue pressing F8 until the cursor is on the curly brace at the end of the 2-argument constructor.

   d. Press F8 once more to return from the constructor.

   e. Execution has advanced to the declaration of `Part3` in `main`.

6. Press F10 two times.

   The construction of the `Part3` and `Part4` objects is complete. The `ImportedPart` 3 argument constructor was called, the base `PartID` was built, and both constructors were completed. You didn't have to trace through it. This is useful for when you're tracing through a program and you hit a function that works correctly or that you're not interested in.

   Note that F8 only traces into all **inline** and all user-written functions. When you're looking at source code, if you use Step Into on a call to `cin` or `cout`, for instance, F8 will jump from your source code window into the source code window for IOSTREAM.H at the statement definition for the **inline** function. This may not be what you want. Plan to use F10 for all **inline** functions.

Examining Variables in the Locals Window

Σ **To explode the display of objects, structs, and variables**

1. Click anywhere in the Locals window to give it the focus. Then place the mouse cursor on a corner of the window and drag the edge around as needed to see the four objects.

2. Restart the program by pressing SHIFT+F5. Now start pressing F10 a few times (it doesn't matter many times, but five or six will do).

   The objective here is to watch the variables change. In particular, the four objects which hold member data. No changes are visible.

   Any time an object, structure, or array appears, you can expand or collapse the display to include or exclude members by double-clicking on a variable. Try this on `Part1` and `Part3` in the Locals window. Note that the + on the extreme left converts to a -. Double-clicking the first line of the object again collapses the display.

3. Restart the program by pressing SHIFT+F5. The Locals window will retain the settings you established.

Setting and Clearing Breakpoints

Σ **To set and clear breakpoints**

1. Click somewhere on the Source window to give it the focus, and use cursor-movement keys to place the cursor on line 101. (The line number is the next-to-last field on the status bar at the bottom of the Visual C++ window.)
2. Press F9.

   This selects line 101 and establishes it as a breakpoint. The F9 key is also used to remove a breakpoint.

   Press F9 twice, leaving a breakpoint set on line 101.
3. Place the cursor on line 103. Press F9. This will establish line 103 as another breakpoint.
4. Press F5.

   MS Visual Workbench executes the program up to the first breakpoint. Line 101 is the next line to execute. Press F9 to remove the breakpoint on line 101.

   Press F5 to execute to line 103. Press F9 to remove the breakpoint on line 103.

Viewing Assembly Code

Σ **To see PARTCOST in Assembly**

1. Press CTRL+F7.

   The source code window now shows a mixture of C/C++ statements and assembly-language statements.

   Move around in the Source window using the PAGE DOWN and PAGE UP keys to examine this feature.

C/C++ programmers sometimes find it necessary to see what the compiler generated from a given expression. This is also a valuable learning tool. You can see how a compiler builds a program, how a function is called, and many other useful bits of information. You are encouraged to use the debugger and this display mode to examine programs this way.

At this course's level of programming, you probably won't use the CTRL+F7 keys when doing actual debugging. Still, in advanced programming, a mixed view of source code can be a useful debugging tool.

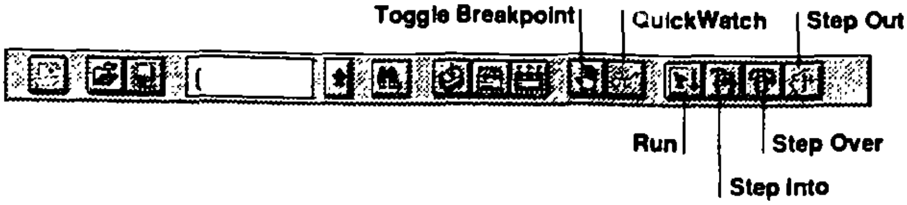
2. Press CTRL+F7 again and you're back to just source code.

There is another use for F7. It is the equivalent of setting a breakpoint with F9 and then pressing F5.

3. Use the cursor-movement keys to position the cursor on line 118. Press F7.

   MS Visual Workbench executes up to line 108 and stops.

The Visual Workbench Debugging Toolbar



From your expenence in the class, you might already be familiar with the leftmost buttons on the Visual Workbench toolbar. Those buttons are used when you write your applications. From left to right they are Project Files, Open, Save, Find (and the dropdown), and Find Next. The middle three buttons are Compile File, Build, and Rebuild All. The six toolbar buttons we'll examine in this debugging exercise are as follows.

- **Toggle Breakpoint** sets or clears a breakpoint at the current location in the Source window.
- **QuickWatch** works with the QuickWatch dialog box to add and display a vanable in the Watch window.
- **Run** starts execution from the current location until a breakpoint is reached or the application terminates. (It is equivalent to the Go menu option or the F5 key.)
- **Step Into** executes one line stepping into a local function call if appropriate. (It is equivalent to the Step Into menu option or the F8 key.)
- **Step Over** executes one line or function call without stepping into the function. (It is equivalent to the Step Over menu option or the F10 key.)
- **Step Out** executes out of the current function call and stops immediately following the call to the function. (It is equivalent to the Step Out menu option or the SHIFT+F10 keys.)

Σ **To practice using the debugging buttons on the toolbar**

1. Place the cursor on line 121 in the Source window. Click the Toggle Breakpoint button on the toolbar. It will be highlighted.
2. Restart the program by pressing SHIFT+F5.
3. Click the Run button on the toolbar several times.

   Notice how the program stops each time it hits the breakpoint. Watch the value of `i` in the Locals window as it changes. You may have to juggle the positions and sizes of the Locals and Source windows to see all this.
4. Click the Step Over button on the toolbar once to advance to `for` loop line above the breakpoint. Move the cursor to line 121 and click the Toggle Breakpoint button on the toolbar. (That deselects the line.)

   Note The apostrophes in here aren't true. (They should be.) Remove parentheses.
5. Click the Run button on the toolbar again.

   The program runs to completion. You should see the QuickWin output screen.
6. Use ALT+F5 to stop debugging. (There is no toolbar equivalent.)
7. Close the process-termination status box by choosing the OK button.

Σ **To restart the program**

1. From the Debug menu, choose Restart.
2. Make the last line of `main` (line 123) a breakpoint.

   Use the scroll bar on the Source window, the mouse, and cursor-movement keys to get the cursor to line 123.
3. Click the Toggle Breakpoint button on the toolbar.

   Make a breakpoint at the end of `main` whenever you begin a debugging session. Since you're never interested in anything after `main`, this is a good and typical practice when debugging applications.
4. Click the Run button on the toolbar.

Σ **To stop and restart the program**

1. Press ALT+F5 to stop debugging. (There is no toolbar equivalent.)
2. From the Debug menu, choose Restart.
3. Click the Step Into button on the toolbar.

The Registers Window

Σ **To examine values in the registers**

1. From the Window menu, choose Registers.

   A new window opens, showing the machine's registers in two-column format. You can resize the window as taller and less wide; the display will change to a single column. Ordinarily this isn't of much interest to a novice programmer.
2. Start pressing F10 and watch the registers change.

One register that is of interest to a programmer is the AX register. All functions with a **return** statement pass the **return** value in the AX register. If you're calling a function and your program isn't written to check the **return** value, you can examine the **return** value this way.

3. Press ALT+1 to change focus to the Source window. Similarly, press ALT+2 and ALT+3 to cycle through the Locals and Registers windows.
4. Press ALT+1 again to return to the Source window.

The QuickWatch Dialog Box

Σ **To display the QuickWatch dialog box**

The Locals window shows all the variables visible by scope to this function. When debugging, you should closely track the values in just a few variables. The QuickWatch box allows you to check the current contents of any variable.

1. In the Source window, place the mouse pointer over the object name, `Part1`.
2. Double-click the left mouse button. (The variable is selected.)
3. Click the QuickWatch button on the toolbar.

   The QuickWatch box appears, listing the variable and its current value.
4. Press the ESC key to close the QuickWatch box.

The Watch Window

Some of the important variables in PARTCOST are the arguments received for the `ImportedPart` object `Part4`. The program display indicates an error in that object. The value listed on the screen for the Price is incorrect.

It's easier to track important variables in a separate Watch window.

Σ **To watch the values of your program's variables change during execution**

1. In the Source window, place the mouse pointer over the variable `Part4`.
2. Double-click the left mouse button. (The word is selected.)
3. Click the QuickWatch button on the toolbar (or use SHIFT+F9).
4. Choose the Add To Watch Window button.

   A Watch window appears. It displays variable details in a window. The Watch window is handy for examining global variables. You usually won't place local variables in the Watch window unless you want to alter how they're displayed.
5. Press F10 several times to see the variable in the Watch window change.

Other Visual Workbench Features

Here are some other MS Visual Workbench features you might find handy:

- Any time a structure or array appears, you can expand or collapse the display to include or exclude structure members. This done by double-clicking on a variable. Try this on `Part4` in the Locals window. Note that the + on the extreme left converts to a -. Double-clicking the first line of the struct collapses it again.
- You can work with all of your breakpoints at once by displaying the Debug menu and choosing Breakpoints. (Breakpoints are a complicated subject in MS Visual Workbench.) In addition to just making a particular line a breakpoint, you can do the following:
  - Break on a line if an expression is true.
  - Break on a line if an expression changes.
  - Break anywhere if an expression is true.
  - Break anywhere if an expression changes.

  The latter two options drastically slow down the Go, Run, and Step options of the MS Visual Workbench debugger. This is because the debugger has to interrupt your program after every machine instruction to see if it should stop.
- If you can find a variable in a window, you can change its contents. Try this on the `n_mPartNbr` variable in the Locals window. (You can even change registers in the Registers window, including IP, the instruction pointer. Be sure you know what you're doing if you attempt this.)

On Your Own

For the remainder of this exercise, experiment with MS Visual Workbench. Try to locate the processing error that causes the `Price` of `Part4` to be zero. (We expect to see a numeric `Price` of 90.)

Be sure you're comfortable with the features covered so far. All debuggers are the same in that they all:

- Allow you to single-step through a program.
- Examine variables.

Everything else is just an enhancement. Be sure you can do those two things with MS Visual Workbench.

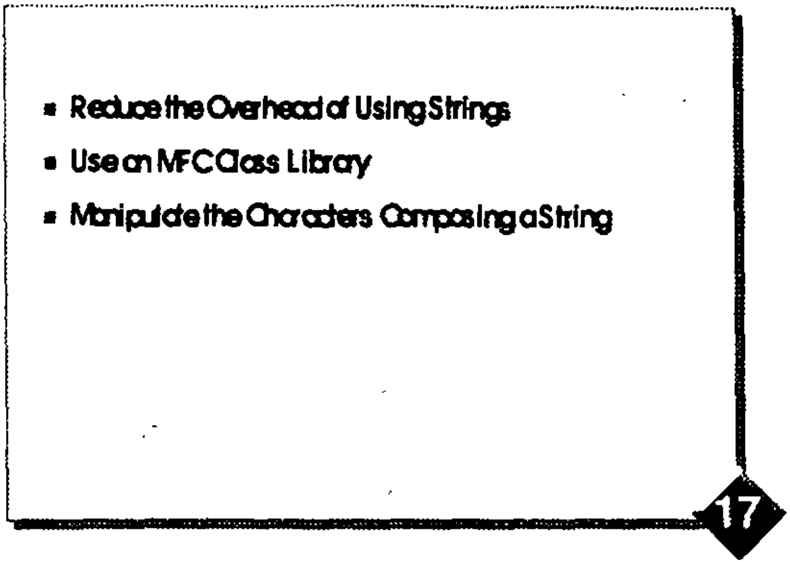
A complete mastery of MS Visual Workbench takes considerable time. This exercise has just touched on the highlights and most essential features. You are encouraged to consult the documentation and to experiment a lot. There's also considerable help available in the helpfiles. You can press F1 to get Help in MS Visual Workbench.

The very best programmers are often those who have mastered a good debugger.

Module 17: Using CString

∑ Overview

Slide Objective
Provide an overview of the module contents.



- Reduce the Overhead of Using Strings
- Use an MFC Class Library
- Manipulate the Characters Composing a String

Module Summary

In the last few modules, you have explored character arrays, pointers to character arrays, and strings. In this module, you'll see how using string objects can significantly reduce the overhead associated with manipulating the character array that composes a string.

The point that is being made in this module can be extended beyond mere character arrays. Using commercially available class libraries can significantly reduce the amount of programming you need to do in general. In fact the whole point of this course is to provide you with the skills you need to be a competent class library user. Microsoft's Foundation Class library is by no means your only option. Since it is included with the Visual C++ development environment, it will be used as an example of how you can incorporate and reuse code from commercially available class libraries.

This module concludes the three-module set on arrays, pointers, references, and strings. Recall from the lectures in these modules that pointers and references can be used to refer to either the value contained within a variable, or its address. This brings us to an important subject: how does a program utilize the computer's memory? That is the topic of the next module.

Objectives

Upon completion of the module, you will be able to:

Key Points
Cover the objectives to set the direction for the module.
The lab solution output is identical to the previous lab, but is much smarter about string-handling.

- Include the MFC CString class declarations.
- Instantiate objects of type String.
- Manipulate the characters composing a string.

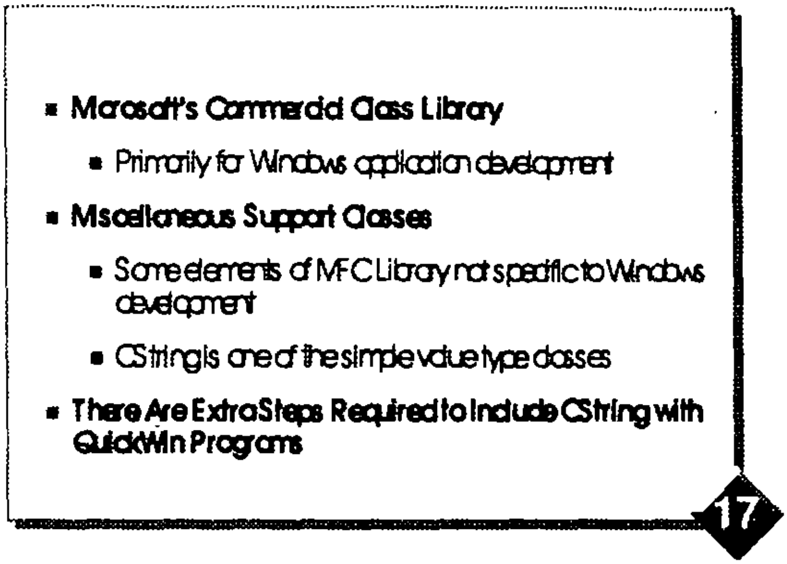
Lab

Using Commercially Available Classes

CString: A Microsoft Foundation Class

Slide Objective

Disclaimer: MFC libraries are C++ classes and objects created for the MS-DOS and MS-Windows platforms. The QuickWin apps we're building are closer to character-mode DOS apps than graphical Windows apps. The applications framework (afx) must be told NOT to include all the Windows app classes. The preprocessor directives below make that distinction.



- **Microsoft's Commercial Class Library**
  - Primarily for Windows application development
- **Miscellaneous Support Classes**
  - Some elements of MFC Library not specific to Windows development
  - CString is one of the simple value type classes
- **There Are Extra Steps Required to Include CString with QuickWin Programs**

MFC libraries are primarily for Windows application development (which is outside the scope of this course). Using CString objects in QuickWin applications requires a modification to the **include** statements. QuickWin applications are a hybrid between an MS-DOS and a Windows application.

The MFC libraries are not built for the QuickWin applications.

To use CString objects, you must make sure that you have taken the following steps:

1. From the Options menu, choose Project. This invokes the Project Options dialog box. Select QuickWin as the Project Type.
2. In the Project Options dialog box, clear the Use Microsoft Foundation Classes checkbox.
3. In the Project Options dialog box, click on the Linker command button. This invokes the Linker Options dialog box. Select the Prevent Use Of Extended Dictionary checkbox.
4. Manually add the library **mafxcr** (or **mafxcrd** if you are building under debug mode) to the Libraries text box in the Linker Options dialog box. If you still get "unresolved external" link errors after you have added it, make sure that MAFXCR.LIB exists in the \MSVC\MFC\LIB directory.
5. Finally, you must define _DOS before you include AFX.H. Place the above preprocessor directives at the beginning of your source file.

This set of preprocessor directives brings in the MS-DOS version of the function prototypes found in the class declarations of AFX.H.

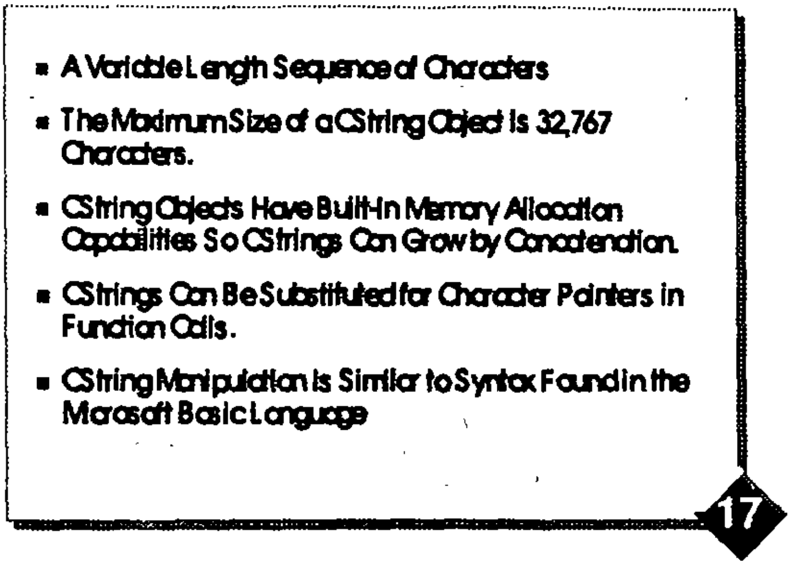
To make sure it all works correctly, try building the following sample program.

Key Points
These statements are not necessary in MFC Windows apps. Only the #include <afx.h> is needed for MS-DOS targets.

```
//*******Test CString with QuickWin EXE*********
#ifdef _WINDOWS
    #undef _WINDOWS
    #define _DOS
    #include <afx.h>
    #undef _DOS
    #define _WINDOWS
#endif
#include <iostream.h>
int main()
{
    CString strHello("Hello World Of Objects");
    cout << strHello <<endl;
    return 0;
}
```

What Is a CString Object?

Slide Objective
A CString object is made from one of the simplest stand-alone classes from MFC.
It is fully self-contained, self-managed, and extremely flexible.



- A Variable Length Sequence of Characters
- The Maximum Size of a CString Object Is 32,767 Characters.
- CString Objects Have Built-In Memory Allocation Capabilities So CStrings Can Grow by Concatenation.
- CStrings Can Be Substituted for Character Pointers in Function Calls.
- CString Manipulation Is Similar to Syntax Found in the Microsoft Basic Language

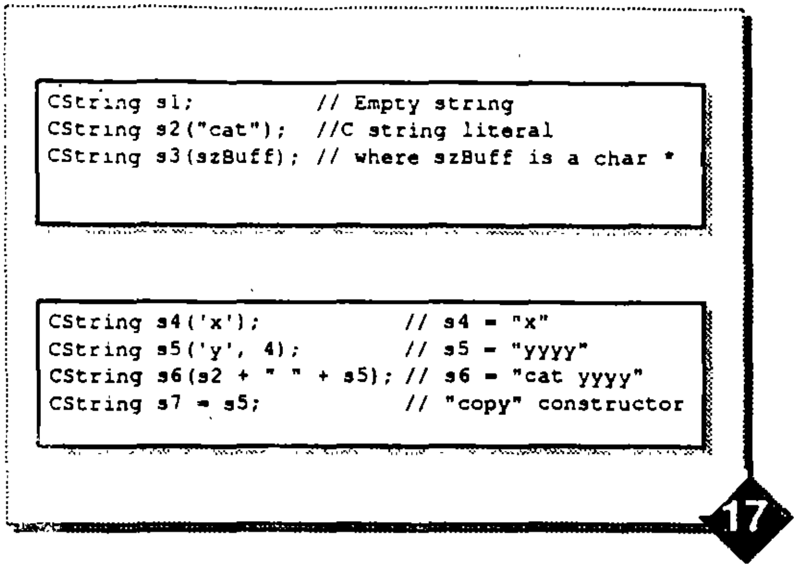
Even though CString objects are similar to arrays and character pointers, they behave like ordinary strings. Like an array, a CString object has member functions to return the number of characters in a CString object and test whether or not it is empty. It can return a character at a given position, and provide access to a character at a given position. Like a pointer, CString objects can be used in place of character pointers as arguments to functions.

But CStrings are objects. You can use them in assignment statements. You can also concatenate them with the + and += operators, compare them, sort them, and extract sequences from them.

Next, you will see how to create CString objects. Following that, you will see how to manipulate data in a CString object.

Creating a CString Object

Slide Objective
Highlight the various overloaded constructors offered by CString.
Note: S7 uses the copy c'tor.



```
CString s1;         // Empty string
CString s2("cat");  //C string literal
CString s3(szBuff); // where szBuff is a char *
```

```
CString s4('x');           // s4 = "x"
CString s5('y', 4);        // s5 = "yyyy"
CString s6(s2 + " " + s5); // s6 = "cat yyyy"
CString s7 = s5;           // "copy" constructor
```

s1 is just instantiated as a CString object. It is empty.

s2 is initialized with a C literal, "cat." CString objects behave like strings, so they can be given literal values.

s3 is constructed from a character pointer.

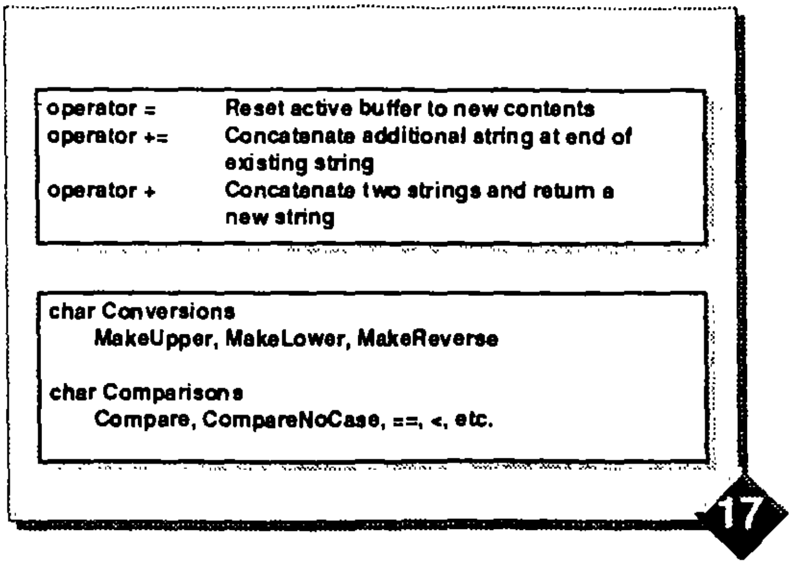
s4 and s5 are constructed from characters.

s6 is constructed by concatenating CString objects with a literal.

s7 might look as if it is getting its data through simple assignment, but this is actually a "copy constructor," which you will examine in a later module.

Manipulating Data in a CString Object

Slide Objective
Besides an expected set of mutator functions (Get & Set), CString offers operators to manipulate strings.



operator =	Reset active buffer to new contents
operator +=	Concatenate additional string at end of existing string
operator +	Concatenate two strings and return a new string

char Conversions
MakeUpper, MakeLower, MakeReverse

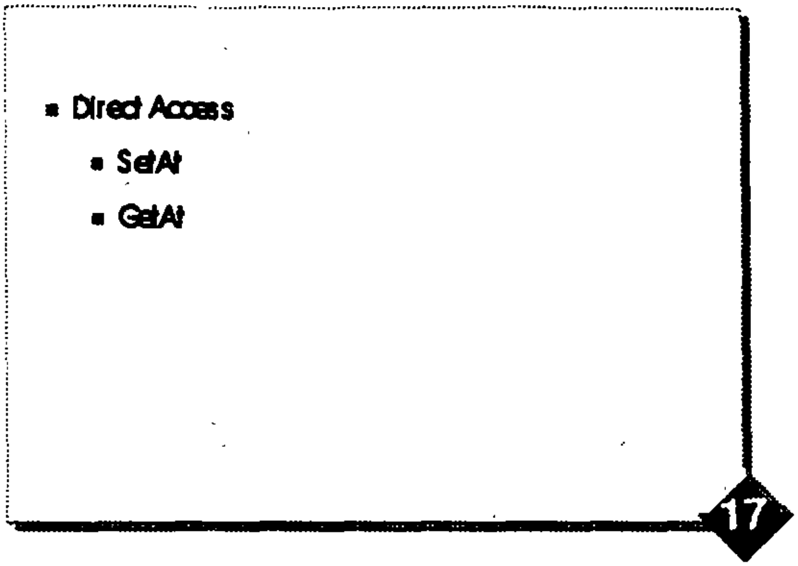
char Comparisons
Compare, CompareNoCase, ==, <, etc.

The CString class has special members that define how standard operators may manipulate CString objects. Those special members, called *overloaded operators*, allow strings to be set and reset (=), expanded or concatenated (+=), and used in string equations with + operators.

CString includes a series of mutator and manipulator functions to massage or modify existing strings in place.

Using a CString Object As an Array

Slide Objective
Introduce direct string access as an alternate to array(subscript) notation.



- Direct Access
  - SetAt
  - GetAt

Key Points
Using the mutator functions, the subscripts are simply function arguments.

With the **SetAt** member function, if you used the following syntax

```
s2.SetAt(2, 'b');
```

`s2` is modified by its member function, `SetAt`, which places the character 'b' at index 2. Given "cat," the result would be "cab."

In contrast, `GetAt(index)` returns the character at a particular index value.

Demo

CSTRING1.CPP is found in \DEMOS\MOD17.

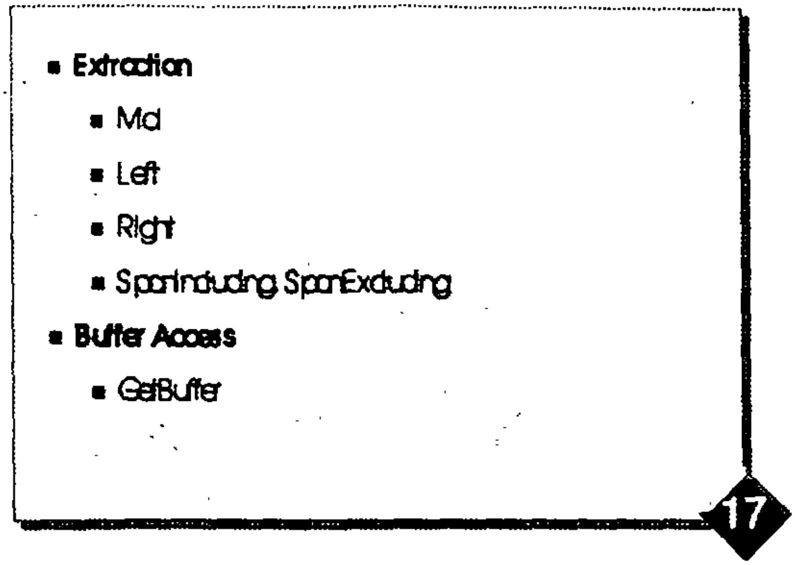
```
// CSTRING1.CPP   Found in \demos\mod17
#ifdef _WINDOWS
   #undef _WINDOWS
   #define _DOS
   #include <afxcoll.h>
   #undef _DOS
   #define _WINDOWS
#endif
#include <iostream.h>

char szBuff[] = "I.S.M. Inc.";

CString s1;                // Empty string
CString s2 ("cat");        // From a string literal
CString s3 (szBuff);       // From a char*    = "I.S.M. Inc."
CString s4 ('$');          // From a char  s4 = "$"
CString s5 ('0', 5);       // Repeat char  s5 = "00000"

                              // From a string expression
CString s6 (s2 + " " + s4); // = "cat 00000"

                            // From a copy constructor, this
CString city = "Redmond"; // is not the assignment operator

void main()
{                          // example for CString::Compare
    cout << "CString object s2 is \"" << s2 << "\".\n";
    if (s2.Compare("bat") == 1) // if cat > bat
    {
        cout << "Cstring Compare showed cat > bat.\n";
        s2.SetAt(0, 'b');           // replace 'c' with 'b'
        if (s2.Compare("bat") == 0) // if 'cat' became 'bat'
            cout << "CString SetAt and Compare worked.\n";
        else
            cout << "CString Compare shows SetAt failed\n";
        cout << "CString CompareNoCase showed \"bat\" is ";
        if (s2.CompareNoCase("BAT") == 0) // bat vs BAT
            cout << "equal.\n";
        else
        {
            cout << "not equal.\n";
            cout << s2 << " can easily be made into ";
            s2.MakeUpper();
            cout << s2 << " using MakeUpper().\n";
        }
    }
    cout << city << " in reverse is ";
    city.MakeReverse();
    cout << city << ".\n";
    city.MakeReverse(); // back to the original city
```

(continued)

```
    // building a string
    city += ',';          // add a char
    city += " WA";        // add a string
    cout << s2 << '\n' << city << ", " << s5 << endl;

    // SetAt and GetAt allow direct access to the
    // current character string
    s5.SetAt(0, '9');   // Set at position 0 char '9'
    s5.SetAt(1, '8');   // Set at position 1 char '8'
    s5.SetAt(3, '7');   // Set at position 3 char '7'
    s5.SetAt(4, '3');   // Set at position 4 char '3'
    cout << s2 << '\n' << city << ", " << s5 << endl;

// Here's trouble!  s5 was initialized to 5 0's and the
// null char is automatically managed by the constructor.
    s5.SetAt(5, 'a');
    cout << "s5.SetAt(5, 'a') sets the 6th element.\n"
         << "s5 is now in an unpredictable state.\n"
         << "Continuing further shows the problem.\n";
    s5.SetAt(6, 'b');
    s5.SetAt(7, 'c');
    cout << "s5 might be '98073abc' but it is '" << s5
         << "'" << endl;
// Don't assume a class member or operator performs extra
// processing (like nulls).  If your CString objects will
// grow, use the += operator.
// SetAt and GetAt may be the best solutions for many cases.
// The class documentation warns about the null character
// condition.
}
```

How You Get Data Out of a CString Object

Slide Objective
The full-featured interface to CString includes functions many will recognize from BASIC.



- **Extraction**
  - Mid
  - Left
  - Right
  - SpanIncluding SpanExcluding
- **Buffer Access**
  - GetBuffer

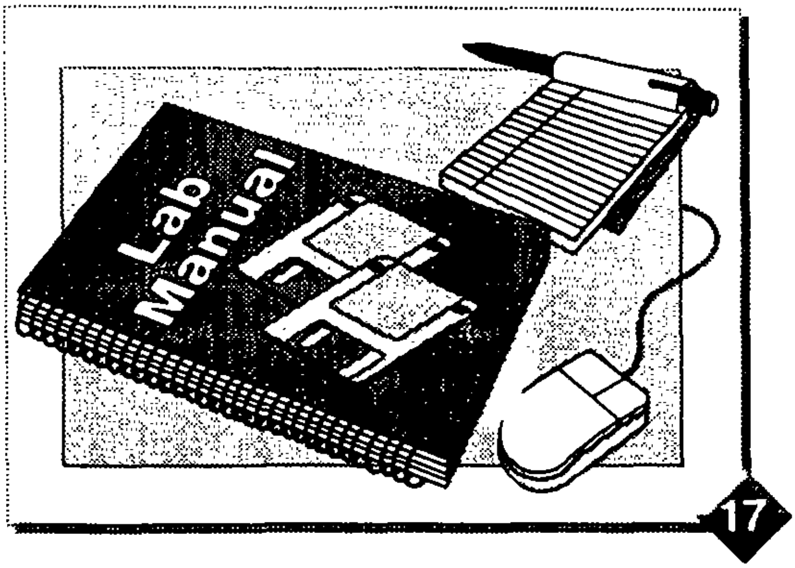
The extraction member functions behave much like those of the Basic language. `Mid(indexFirst, [nCount])` begins with the character in the sequence `indexFirst` and continues either to the end or for `nCount` characters. The `Left` and `Right` member functions behave similarly.

The `GetBuffer` member function returns a character pointer to a buffer where the string's characters exist. Until the buffer is reset, the character pointer has full access to all character locations.

Lab 14: Using Commercially Available Classes

Slide Objective
Execute the lab solution.
Set the lab objectives.
Ask students to read the lab scenario..

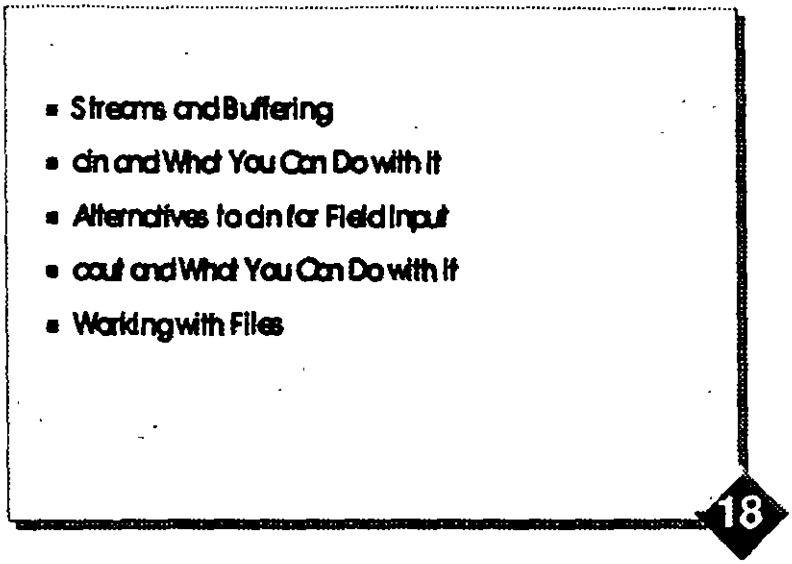
Have students avoid the optional section if time is tight. That section uses SetAt() and GetAt() instead of subscripted arrays. There is no O-O or .EXE benefit.



Module 18: Formatting and File I/O

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Streams and Buffering
- cin and What You Can Do with It
- Alternatives to cin for Field Input
- cout and What You Can Do with It
- Working with Files

Module Summary

This module begins a number of topics that help you add functionality to your programs. You'll start it off with this module on input and output.

C++ stream objects simplify I/O (and particularly file I/O) over the strictly C syntax. And though no other modules rely directly on stream objects, I/O is one of the most important functions of computer programs.

Objectives

Upon completion of this module, you will be able to:

Delivery Tips
Present the learning objects to set the direction for the module. As a variation, this module covers classes that are included in C++ libraries available with all C++ compilers.

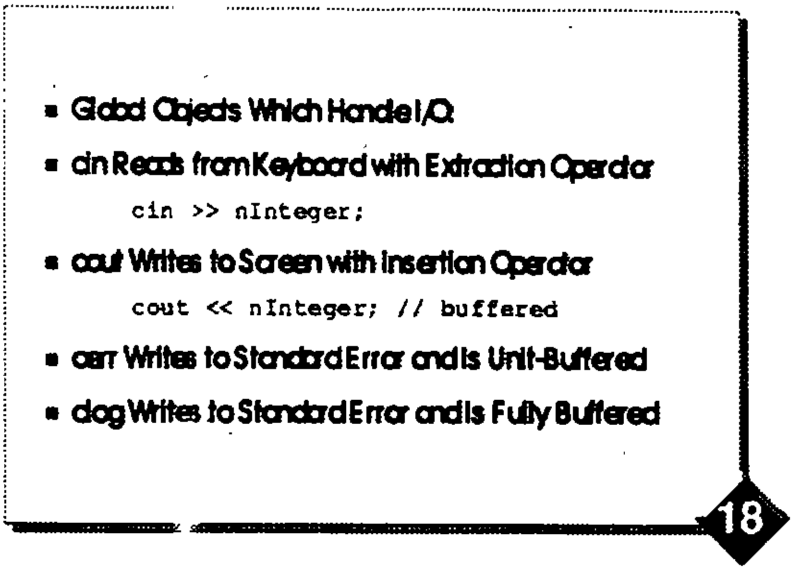
- Create formatted output at the character, word, line, and file levels.
- Open and close files.
- Get data from files and put data into files.

Lab

Formatting and File I/O

Streams and Buffer g

Slide Objective
Set a foundation for i/o streams from a perspective that includes efficient processing for PCs.



- Global Objects Which Handle I/O
- cin Reads from Keyboard with Extraction Operator

```
cin >> nInteger;
```

- cout Writes to Screen with Insertion Operator

```
cout << nInteger; // buffered
```

- cerr Writes to Standard Error and Is Unit-Buffered
- clog Writes to Standard Error and Is Fully Buffered

What Are Streams?

You should think of a stream object as a smart file that acts as a source and a destination for bytes. Although this module cannot cover all devices, these concepts apply when reading from and writing to keyboard, screen, disks, printers, communication ports, memory, and more.

The four stream objects "know" how to input/output **int**, **char**, **char***, and so on. They are objects of classes which overload >> and << such that the input/output of **int**, **char**, **char***, **float**, and others "happens correctly."

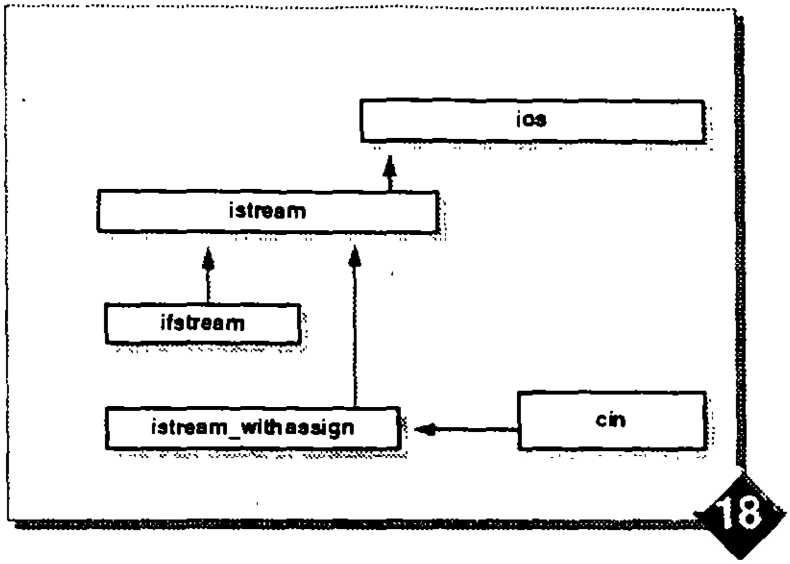
Why Buffers Are Your Friends

Using buffers keeps a PC running at a reasonable pace because buffer access is at RAM speed, not drive speed. The disk and diskette drives in personal computers are block-mode devices. The mechanical operations of moving the read/write heads, waiting for the rotation of the media, and transferring data is hundreds of times slower to a disk drive than to memory chips. Therefore, the disk controller card, device drivers, and operating system work together to buffer information. The device driver will read a sector of information and load it to a buffer. Subsequent requests for the next character are handled from the buffer.

Unit-buffering "packages" characters in a complete line before displaying them on the screen. Fully buffered output packages multiple lines as needed until the stream is explicitly flushed.

cin and What You Can Do with It

Slide Objective
Take the magic out of the "cin" object students have used all week.
Cover "cin" origin as an object from istreamwithassign and member functions, operators, and manipulators inherited from base classes.



Key Points
An object of a well-defined class can be used easily without knowing how it is implemented.

How cin Works

The cin object is a predefined object of class **istream_withassign**. The class **istream_withassign** only allows stream objects to be constructed, destructed, and assigned to replace cin. As depicted in the hierarchy, however, the cin object inherits access to member functions and public data members from **istream**.

How Extraction Works

The extraction operator (>>) matches data from the stream with variables you supply and then returns a reference to the stream. That return allows one line of code to extract multiple variables as follows.

```
cin >> nA >> nB >> nC;
```

The value for integer nA is assigned the first numeric value entered up to the following whitespace (tab, space, newline, and so on). The value for nA is determined and the reference to the stream is passed from the first >> operator to the second >>. From there, input proceeds to extract the value for nB, and so on.

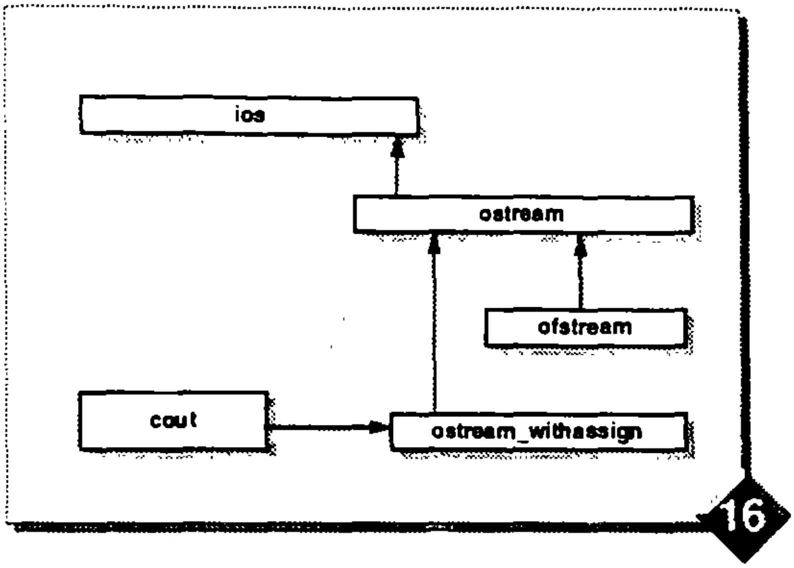
Formatted text input, or extraction, depends upon whitespace to separate values—but data errors or unexpected results can occur and need to be checked for. There are a number of member functions available to help you out.

Error-Handling Member Functions

A failure bit is set when input errors occur. This is the program's clue that cin could not match the input stream to the data types. This bit should be reset for input to continue.

cout and What You Can Do with It

Slide Objective
Explain the origins of the "cout" object. Include details and examples of member functions, operators, and manipulators.



How cout Works

The **ostream_withassign** class is a variant of **ostream** that allows object assignment. This class has the predefined objects **cout**, **cerr**, and **clog**.

Here are some of the many things you can do with **cout** (and **cerr** and **clog**):

You can use the following manipulators. A manipulator is a "packaged" mutator function that modifies the behavior of the stream. Some make permanent changes, and some make temporary changes.

- **endl** inserts a newline character and then flushes the buffer.
- **ends** inserts a null terminator character.
- **flush** flushes the output buffer.

The following member functions are also available:

- **put** inserts a single character into the output stream.
- **write** inserts a specified number of bytes from a buffer into a stream.
- **tellp** gets the position value for the stream.
- **seekp** changes the position value for the stream.

These character escape sequences are used to advance lines down the screen. (You saw them in an earlier module.)

- `'\n'` inserts a newline character.
- `'\l'` inserts a linefeed down.

The following character escape sequences are used to advance columns across the screen:

Spaces or tabs

- `' '` inserts a space character.
- `'\t'` inserts a tab character.
- `'\r'` returns to leftmost column on the same line.

The following can be used to format output with **cout**:

- Setting width:

```
cout.width(10)    // member function
out << setw(10):  // manipulator
```

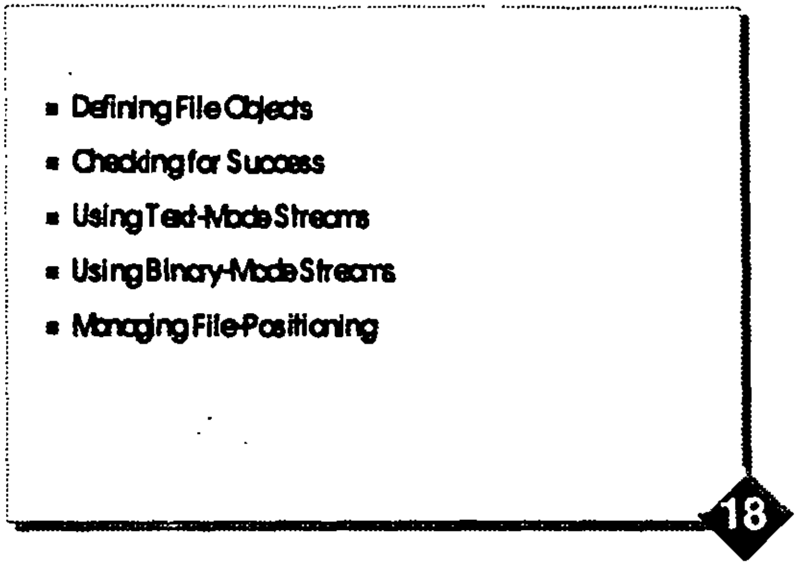
- Filling a field with a user-defined character:

```
cout.width(10);
cout.fill('*');
cout << nCnt:
```

- Flags for formatting

Justify	Float	Example
ios::left	**ios::fixed**	123.4
ios::right	**ios::scientific**	1.2e+002

Working with Files: Overview



- Defining File Objects
- Checking for Success
- Using Text-Mode Streams
- Using Binary-Mode Streams
- Managing File Positioning

Slide Objective
Initiate the topic of File I/O. Begin with the slow, inefficient block-mode devices that are the target meaia.

Let students know that we'll start at ground zero and are going to cover file i/o from the from the C++ library functions up toward user-defined functions.

Quickly cover the simple sample below.

The cin and cout objects are:

- Predefined objects.
- Connected to streams.
- Tools for access to dozens of operators, manipulators, and member functions.

To work with files, you will:

- Define and open objects.
- Connect to data files.
- Have access through dozens of operators, manipulators, and member functions.

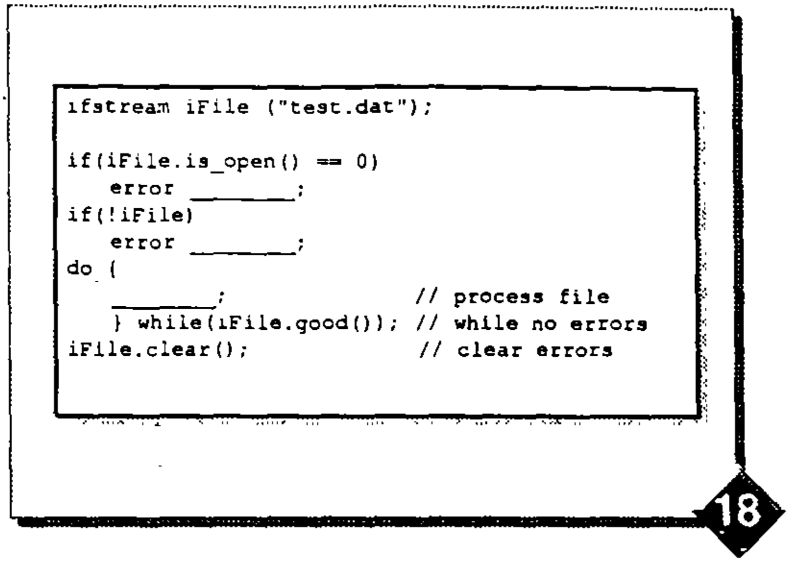
Demo

TFILE.CPP is found in \DEMOS\MOD18.

```
// TFILE.CPP   Found in \demos\mod18
// Create a file, test.dat, and writes the msg:
// "This is test data".  File closed by d'tor.
#include <fstream.h>

void main()
{
    ofstream tfile("test.dat");
    tfile << "This is test data";
}
```

Checking for Success

Slide Objective
Always expect errors dealing with I/O.
Your code may be fine, the disk may be full, or the user may enter letters when you expected an integer.



```
ifstream iFile ("test.dat");

if(iFile.is_open() == 0)
   error ________;
if(!iFile)
   error ________;
do {
   ________;                // process file
   } while(iFile.good()); // while no errors
iFile.clear();             // clear errors
```

Class `ifstream` is specialized for disk file input and output. The constructor (and open) automatically create and attach a file buffer object. The file buffer object holds file-sharing information: either exclusive use, or read-sharing or write-sharing.

The `fstream` class implements a member function, `is_open()`, which returns an integer if the file is not connected.

Both `ofstream` and `ifstream` inherit the NOT operator `!` from class `ios`. This overloaded operator returns a non-zero value if a stream I/O error has occurred. Operator `!` may be used with all stream objects at open or during processing.

Demos

OUT.CPP is found in \DEMOS\MOD18.

```
// OUT.CPP  found in \demos\mod18
// Creates a file, test.txt, and outputs two lines.
#include <iostream.h>
#include <fstream.h>     // For file stream support

void main()
{                        // Create disk file: test.txt
                         // Note: the 2nd arg to the
                         //  c'tor is: ios::out | ios::app
    ofstream outfile("test.txt");
    if (!outfile)        // test for successful open
        cerr << "Cannot open 'test.txt' for output.\n";
    else
        outfile << "This is test data.\n"
                << "File will be closed at termination.\n";
}
```

INOUT.CPP is found in \DEMOS\MOD18.

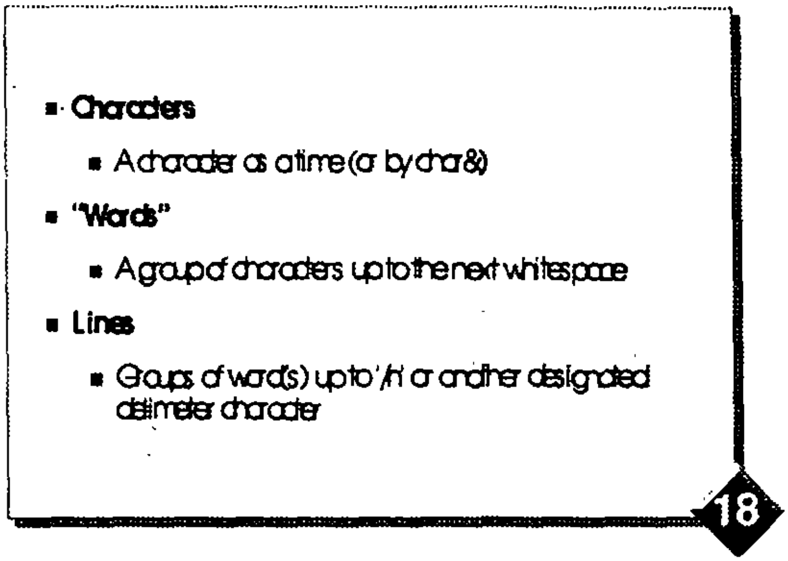
```
// INOUT.CPP   found in \demos\mod18
// Read an input file, test.txt, getting a character
// at a time, appends the files content as all capital
// letters at the end of the original file.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

#define SIZE 100

int iCount = 0;
char data[SIZE];
void main()
{    // fstream inherits input & output
                         // ::in input mode
                         // ::app append additions
    fstream iofile("test.txt", ios::in | ios::app);
    if (!iofile)         // error handling
        cerr << "Trouble opening file 'test.txt'.  "
                "Please run 'out.exe' to create file.\n";
    while (!iofile.eof()) // while data exists, load data
        iofile.get(data[iCount++]); // get 1 char at a time
    iofile.clear();      // clear eof & other error states
    iCount--;            // adjust for 'off by one'
    for (int j = 0; j < iCount; j++)
    {                    // "put" uppercase chars to file
       data[j] = (char) toupper(data[j]);
       iofile.put(data[j]);
    }
}
```

TOFILE.CPP is found in \DEMOS\MOD18.

```
// TOFILE.CPP  found in \demos\mod18
// Takes user input and write characters to file test.out.
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>      // for exit()

void main()
{
    char ch;
    ofstream outfile("test.out", ios::out);
    if (!outfile)        // detect error opening file
    {                    // give user suggestions
        cerr << "Trouble opening file 'test.out'.  "
                "Check disk: file read only? full?\n";
        exit(1);
    }
    cout << "Enter characters.  Use Ctrl-Z to quit.\n";
    while (cin.get(ch))  // while data exists
        outfile.put(ch); // put char to file
}
```

Using Text-Mode File Streams

Slide Objective
The previous examples used various keyboard and file I/O techniques. Summarize those details.



- Characters
  - A character as a time (or by char&)
- "Words"
  - A group of characters up to the next whitespace
- Lines
  - Groups of word(s) up to '/n' or another designated delimeter character

Character-by-character processing with **char ch**:

Member Function	Meaning
`iFile >> ch;`	Extraction operator matches the **char** data type and returns characters.

`iFile.get();`
`iFile.get(ch&);`
	The **get** function has multiple forms. Given a **char** or **char** reference, it extracts one character at a time. **get()** returns whitespace.

Word-by-word processing with **char szBuff[SIZE]**:

Member Function	Meaning
`iFile >> szBuff;`	Again, the extraction operator matches the array of characters and extracts a group of characters into **szBuff**.

Line-by-line processing with **char szBuff[SIZE]**:

Member Function	Meaning

`iFile.get(szBuff, SIZE);`
`iFile.getline(szBuff, SIZE);`
`iFile.getline(szBuff, SIZE, '\t')`
	By default, the **get** and **getline** member functions extract up to SIZE characters. Both accept a third argument to override the default delimiter character, '\n'.

Demo

FTOFNBR.CPP is found in \DEMOS\MOD18.

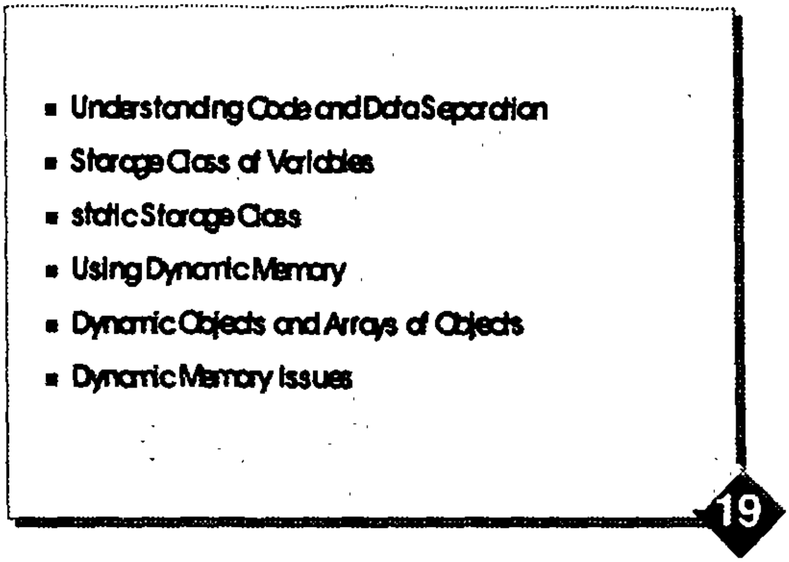
```
// FTOFNBR.CPP found in \demos\mod18
// The application reads text files by char, word, and
// line.  It duplicates the input file, creating a
// line-numbered file with the extension ".NBR".
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>        // for exit()

#define SIZE 256

void main()
{
    int nCntChars, nCntWords, nCntLines;
    char data[SIZE], ch;
    // Create stream objects using constructors:
    ifstream infile("test.txt", ios::in);
    ofstream outfile("test.out", ios::out);
    if (!infile || !outfile)
    {
        cerr << "Error opening file(s)";
        exit(1);
    }
    /************ 'char' pass thru input file ***********/
    for (nCntChars = 0; infile.get(ch); ++nCntChars);

    cout << "Input file contained " << nCntChars
         << " characters, ";
                            // reset infile for 'word' pass
    infile.clear();                   // reset eof state
    infile.seekg(0L, ios::beg);       // seek to 0-byte
    /************ 'word' pass thru input file ***********/
    while (infile >> data)
        ++nCntWords;

    cout << nCntWords << " words, ";
                            // reset infile for 'line' pass
    infile.clear();         // reset eof
    infile.seekg(0L);       // seek (default ios::beg)
    /************ 'word' pass thru input file ***********/
    for (nCntLines = 1; infile.getline(data, SIZE);
        ++nCntLines)
    {
        outfile.width(3);       // set width for line #
        outfile << nCntLines << ". ";  // insert line #
        outfile << data << endl;// insert line to file
    }
    cout << nCntLines << " lines.\n";
    cout << "File-to-file number copy complete.\n\n";
    infile.close();         // close files (disconnect stream)
    outfile.close();        // or the d'tor will (good style!)
 (continued)
```

```
    cout << "***      Brain Teaser      ***\n";
    cout << "    get.(c) reports " << nCntChars
         << " chars.\n";
    cout << "getline.(*) reports " << nCntLines
         << " lines.\n";
    cout << "But, dir cmd shows: "
         << nCntChars + nCntLines << " size.\n";
    cout << "*** Q: Why the difference? ***\n";
}
```

Module 19: Memory Management

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Understanding Code and Data Separation
- Storage Class of Variables
- static Storage Class
- Using Dynamic Memory
- Dynamic Objects and Arrays of Objects
- Dynamic Memory Issues

19

Module Summary

One of the fundamental concepts of modem computer science is the separation of code from data within programs. PC programs place data and executable code in different areas—in the simplest case, in different "segments."

The data area is further divided into the heap, the stack, and the static data areas. Variables in a C++ program live in one of these three subareas. The subarea affects some of the attributes of a variable; it defines the *storage class* for a variable. Selecting the correct storage class can have a profound effect on a program's performance.

This module is only an overview of an extensive and implementation-dependent subject. Appendix C contains additional information on memory issues.

Objectives

Delivery Tips
Cover the learning objectives.

The second lab exercise is a game; it may provide a distraction.

Upon completion of this module, you will be able to:

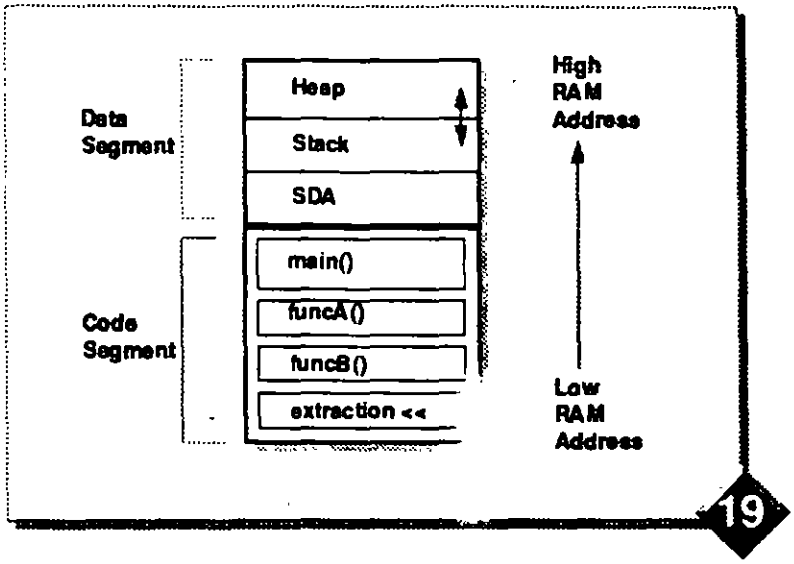
- Draw a distinction between code and data segments and how the data segment is partitioned.
- Create variables of the different storage classes (this includes managing variables dynamically).
- Understand how the storage class of a variable affects it behavior and the performance of your program.

Lab

Dynamic Memory

Understanding Code and Data Separation

Slide Objective
MS Compilers refer the data segment as the DGROUP. Provide a high-level introduction to the type-modifiers, data segment (DGROUP), and code segments.



Data Segment

Heap

Stack

SDA

Code Segment

main()

funcA()

funcB()

extraction <<

High RAM Address

Low RAM Address

19

When a C++ program is loaded in RAM memory, it is divided into two main portions, or segments: the data and the code segments.

The code segment contains all the executable machine code statements, which are grouped into functions. These are just the translations of user-supplied or library C++ statements.

The data segment contains all the variables and literals in the C++ program. It is further divided into three subareas:

- The *SDA* (static data area) contains all global (and static) variables and literal values.
- The *stack* is the data work area for functions. Each currently active function allocates a *stack frame*, where it stores its local variables, arguments, and administrative information.
- The *heap* is the area from which variables are dynamically allocated and deallocated.

The size of the SDA is fixed at link time, and does not change.

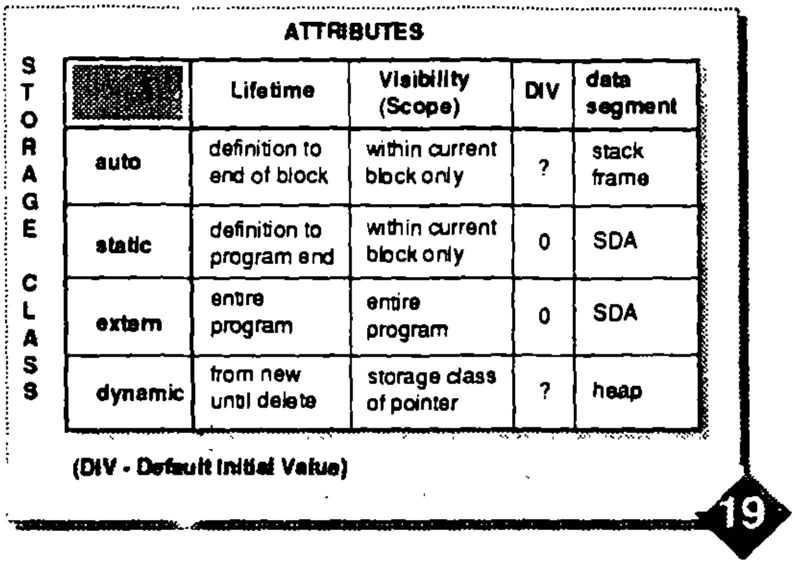
At run-time, the stack grows downward in stack-frame chunks as functions are invoked. It shrinks as functions return.

The heap grows generally in an upward direction as memory is dynamically allocated. It often fragments as memory is deallocated.

Key Points
Cover the load image of an .EXE from the bottom up:
Code seg. is user and library functions.
Data seg. (MS calls DGROUP) contains 3-major subareas:
Static Data area:
NULL seg. Copyright notice marked read-only.
_DATA seg. *Initialized* global data and static local area.
_CONST seg. String literals.
BSS and **_C_COMMON** *Uninitialized* globals and static local (all set to 0)
Stack. Auto variables and parameters are on the stack at run-time.
Heap. Unallocated memory pool for dynamic allocations.

Storage Classes

Slide Objective
With the previous diagram, this summary chart depicts a table of the type-modifier keywords that affect a variable's location in memory. Define dynamic variables in the context of the heap.



ATTRIBUTES

S T O R A G E C L A S S

▨	Lifetime	Visibility (Scope)	DIV	data segment
auto	definition to end of block	within current block only	?	stack frame
static	definition to program end	within current block only	0	SDA
extern	entire program	entire program	0	SDA
dynamic	from new until delete	storage class of pointer	?	heap

(DIV - Default Initial Value)

C++ variables can have four different storage classes that determine their lifetime and visibility within a program. We have used so-called "local," "global," and "static" variables up to this point. Their proper storage class names are **auto** (automatic), **extern** (external), and **static** respectively. (Literal strings have a storage class of **extern.**)

The **static** storage class is an intermediate between **extern** and **auto.** It enjoys the lifetime and default initial value of an **extern**, but the limited visibility of an **auto.**

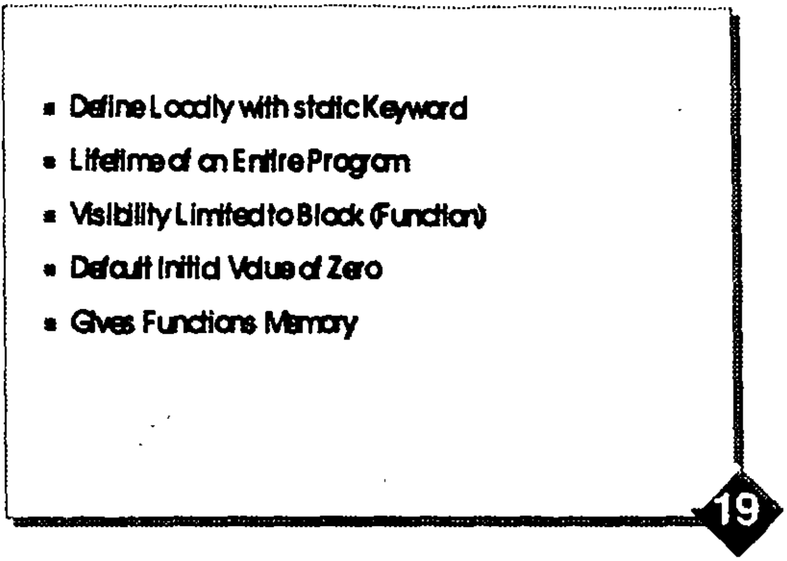
The heap allocates contiguous series of bytes that can be used by the programmer as variables or arrays of variables. Later in this module, you will see how to dynamically allocate and deallocate from the heap subarea.

Delivery Tips
static has three uses in C++: static storage class, static linkage, and static member functions.
void keyword is used for more than one purpose.

static Storage Class

Slide Objective
Cover the use of the static type-modifier for variable declarations. Introduce static in the context of the data variables in the DGroup.

Note: Course has already covered static data members and member functions.



- Define Locally with static Keyword
- Lifetime of an Entire Program
- Visibility Limited to Block (Function)
- Default Initial Value of Zero
- Gives Functions Memory

Key Points
Using "static" in front of an auto variable (in a function) drives the storage from the stack to the static data area. The variable retains values from call to call of the function.

Using "static" in front of a global variable (defined outside a function), affects the global visibility. Only functions defined in the current source file have access to that variable.

Static variables are defined at function scope, much like automatic variables. The difference is that the keyword **static** is placed before the data type keyword:

```
static int nTemp = 5;
```

Static variables live for the entire program; automatic variables are reincarnated each time their function is invoked.

The visibility of statics is limited to the current block, usually a function body. This is also true of automatic variables.

Initialization for statics occurs once, at program load time; the default is zero. Automatic variables are (re)initialized every time their function is invoked, with the default being some unknown value.

Keep in mind that you can assure the default value of abstract variables (regardless of their storage class) by supplying explicit constructors.

The main purpose for static variables is to give functions memory between invocations while still maintaining local function encapsulation.

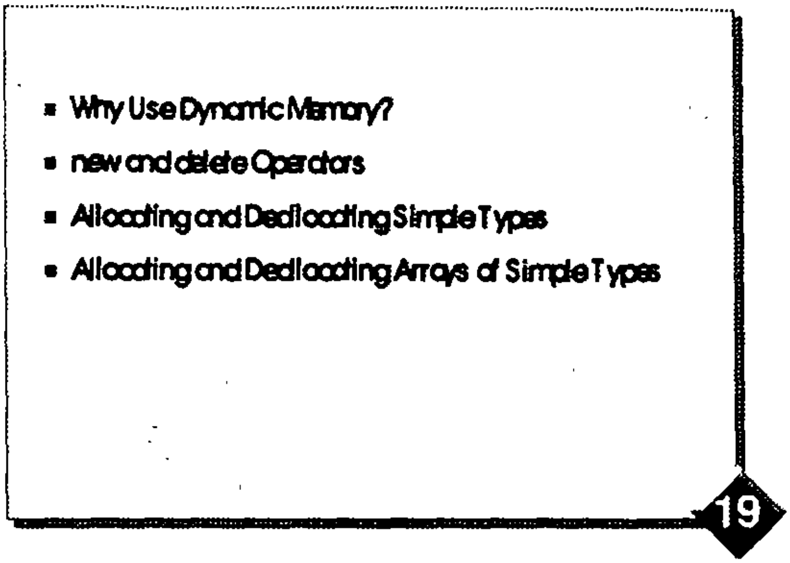
Demo

STATIC.CPP is found in \DEMOS\MOD19.

```
// STATIC.CPP found in \demos\mod19
// Demonstates auto and static storage class.
#include <iostream.h>
                        // function prototypes
int funcA(int);         // un-initialized local
int funcB(int);         // initialized local
int funcC(int);         // static

int nGlobal;            // default initial 0

int main()
{                       // output global to prove 0
    cout << "nGlobal is " << nGlobal << endl;
    cout << "\nCalling funcA...\n";
    cout << funcA(3) << endl;
    cout << funcA(3) << endl;
    cout << funcA(3) << endl;
    cout << "\nCalling funcB...\n";
    cout << funcB(3) << endl;
    cout << funcB(3) << endl;
    cout << funcB(3) << endl;
    cout << "\nCalling funcC...\n";
    cout << funcC(3) << endl;
    cout << funcC(3) << endl;
    cout << funcC(3) << endl;
    return 0;
}

int funcA(int n)
{
    int nTemp;           // nTemp not initialized!
    nTemp += n;
    return nTemp;
}

int funcB(int n)
{
    int nTemp = 1;
    nTemp += n;
    return nTemp;
}

int funcC(int n)
{
    static int nStat;    // default inital 0
    nStat += n;
    return nStat;
}
```

Using Dynamic Memory

Slide Objective
Define dynamic memory and cover its purpose and benefits.



- Why Use Dynamic Memory?
- new and delete Operators
- Allocating and Deallocating Simple Types
- Allocating and Deallocating Arrays of Simple Types

Dynamic memory is useful if a program has no prior knowledge of how much information it must handle, has transient memory needs, or needs to create variably sized objects. Data structure libraries invariably use dynamic memory.

The C++ language allocates heap memory with the **new** operator and deallocates memory with the **delete** operator. For example, to allocate an integer-sized variable on the heap:

```
int *pn = new int;
```

The **new** operator allocates two bytes on the heap and returns a pointer to the beginning of that block. Note that the variable created does not have a name. It can only be accessed through the associated pointer.

Delivery Tips
NULL is defined in stdlib.h as well as several other header files to be 0 in C++ in C it is defined to be ((void*)0).

If **new** fails to allocate this variable for some reason, it will return a pointer with a value of zero, called the *NULL pointer*. When you use **new**, you should always test the return value against NULL.

The initial value of a dynamic variable will be garbage.

The **delete** operator takes a pointer to the beginning of a block of memory, as in

```
delete pn;
```

The heap memory that was used by this variable is now freed.

Key Point
Note the use of [ ] with arrays.

Allocation and deallocation of simple arrays is a straightforward extension:

```
int *pan = new int[100];
. . .
delete [] pan;
```

Demo

DYNAMIC1.CPP is found in \DEMOS\MOD19.

```
// DYNAMIC1.CPP  found in \demos\mod19
// Dynamic allocation and deallocation of standard types.
#include <iostream.h>
#include <stdlib.h>
#include <memory.h>

void CheckNull(void*);

int main()
{
    unsigned int iRange;
    // allocate space for an unsigned long
    unsigned long *pn = new unsigned long;
    CheckNull(pn);          // error checking
    cout << "Enter a positive integer value: ";
    cin >> *pn;             // accept input into alloc space
    cout << "The square of the number is  "
          << *pn * *pn << endl;
    delete pn;              // release the space

    cout << "How many powers of 2 do you want to see?\n";
    cout << "Enter number between 1 and 40 please: ";
    cin >> iRange;
    iRange %= 41;           // trim user input > 40
    // allocate an array of iRange unsigned longs
    pn = new unsigned long[iRange];
    CheckNull(pn);          // error checking
    pn[0] = 1;              // a number to 1st power=itself
    cout << endl;
    cout.width(12);
    cout << pn[0];          // output first element
    for (unsigned int k=1u; k < iRange; k++)
    {
        pn[k] =  pn[k-1u] * 2ul; // calculate next
        cout.width(12);
        cout << pn[k];          // Show results 5-
        if ((k+1u) % 5u == 0)   // wide across the crt
            cout << endl;
    }
    delete [] pn;  // release the array allocation space

    return 0;
}

void CheckNull(void* pv) // Check for new failures
{
    if (pv == NULL)       // NULL ptr indicates error
    {
        cerr << "\nERROR: Heap Allocation Failure!";
        exit(1);
    }
}
```

Demo

Note Close all files and projects. Use DYNAMIC2.MAK (found in \DEMOS\MOD19) to access the following files: DYNARRAY.H, DYNARRAY.CPP, and DYNAMIC2.CPP.

DYNARRAY.H is found in \DEMOS\MOD19.

```
// DYNARRAY.H  found in \demos\mod19
// Demonstates dynamic allocation and deallocation
// of standard types within a class.
#include <iostream.h>
#include <stdlib.h>
#include <memory.h>
#include <limits.h>

/**********************************************************
 Class DynArray - Inefficient but simple implementation
   of dynamic arrays. Only allows adding new element to
   end. Allocation checking performed in c'tor and in
   AddElement and simple range checking done in
   GetElementAt and SetElementAt
**********************************************************/
                         // Uses a manifest data type
#define TYPE int         // value for genericity.
#define SIZE 10          // unit of growth

class DynArray
{
public:
    DynArray(unsigned int size = CHUNKSIZE);
    ~DynArray();
    unsigned int GetSize(void)
        { return m_nSize+1; } // change from 0 to 1-based
    void AddElement(TYPE);
    void SetElementAt(unsigned int index, TYPE val);
    TYPE GetElementAt(unsigned int);
    void Display(unsigned int);
private:
    void CheckNull(void);
    unsigned int m_nSize; // 64K max elements
    unsigned int m_nLast; // last used element
    TYPE *m_pBeg;
};

```

(continued)

```
/******* Class DynArray Inline Member Functions *********/
inline DynArray::~DynArray()
{
    delete [] m_pBeg;
}
        /* Simple allocation checking implemented here. */
inline void DynArray::CheckNull(void)
{
    if (m_pBeg == NULL)
    {
        cerr << "\nError: "
            "Memory Allocation Failure Within DynArray"
            << endl;
        exit(1);
    }
}
```

DYNARRAY.CPP is found in \DEMOS\MOD19.

```
// DYNARRAY.CPP  found in \demos\mod19
// Demonstates dynamic allocation and deallocation
// of standard types within a class.
#include "dynarray.h"
#include <memory.h>

/*********** Class DynArray Member Functions ***********/
        //DynArrays are zero based just like C++ arrrays.
DynArray::DynArray(unsigned int size)
    : m_nSize(size-1), m_nLast(0)
{
    m_pBeg = new TYPE[size];
    CheckNull();
                         // Zero new area out for safety
    memset(m_pBeg, 0, size * sizeof(TYPE));
}

void DynArray::AddElement(TYPE val)
{
    if (m_nLast < m_nSize) // If any unused slots are left
        *(m_pBeg + m_nLast + 1) = val; // use them first
    else                               // else make more.
    {       // This is the horribly inefficient part.
        TYPE *ptemp = m_pBeg;
        m_nSize += CHUNKSIZE;
        m_pBeg = new TYPE[m_nSize];
        CheckNull();
        memcpy(m_pBeg, ptemp, (m_nSize-1)*sizeof(TYPE));
        delete [] ptemp;
        m_pBeg[m_nLast + 1] = val;
    }
    m_nLast++;
}
        // Allow user to access any allocated element.
TYPE DynArray::GetElementAt(unsigned int index)
{
    if (index < 0 || index >= m_nSize)
    {
        cerr << "\nOut of Bounds Error in GetElementAt"
             << endl;
        exit(1);
    }
    return m_pBeg[index];
}
        // Allow user to set any allocated element.
void DynArray::SetElementAt(unsigned int index, TYPE val)
{
    if (index < 0 || index >= m_nSize)
    {
        cerr << "\nOut of Bounds Error in SetElementAt"
             << endl;
        exit(2);
    }
    m_pBeg[index] = val;
}

```

(continued)

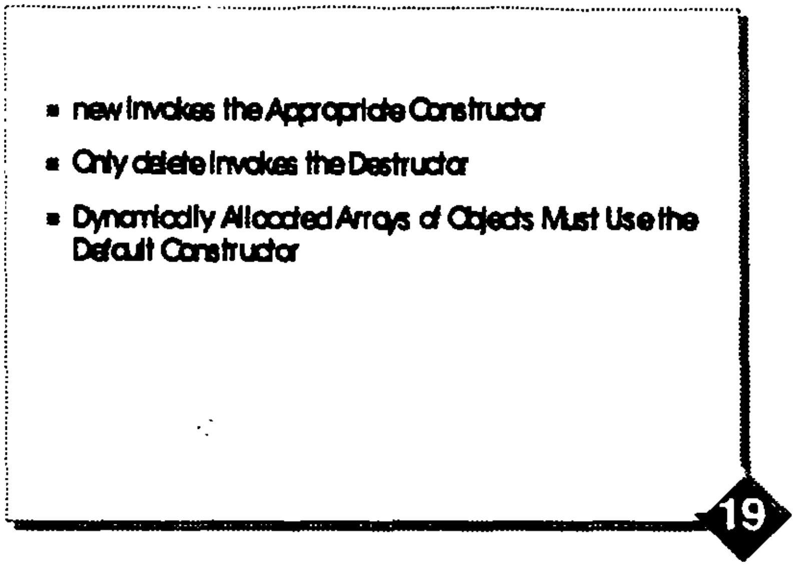
```
void DynArray::Display(unsigned int index)
{
    for (unsigned int i = 0; i <= index; i++)
        cout << m_pBeg[i] << ' ';
}
```

DYNAMIC2.CPP is found in \DEMOS\MOD19.

```
// DYNAMIC2.CPP  found in \demos\mod19
// Project files DYNARRAY.CPP and DYNARRAY.H demonstrate
// allocation and deallocation of standard types within
// the dynamic array class.
#include <iostream.h>
#include "dynarray.h"

int main()
{
    char c;
                          // Create two DynArray objects
    DynArray d1, d2(1000);   // d1 is empty, d2 is 1000
    d1.AddElement(5);     // Add 5-elements to d1
    cout << "The size of d1 is " << d1.GetSize() << endl;
    cout << "The element d2[500] initially is "
          << d2.GetElementAt(500) << endl;
                          // Set number 666 at element 500
    d2.SetElementAt(500, 666);
    cout << "After SetElement, element d2[500] is "
          << d2.GetElementAt(500) << endl;
                          // trip range checking
  // d1.GetElementAt(20);
                          // trip allocation checking
    cout << "\nEnter any key to eat up the heap.";
    cin >> c;
    while(1)
    {
        d1.AddElement(rand());
    }
    cout << "\nEnd of main" << endl;
    return 0;
 }
```

Dynamic Objects and Arrays of Objects

Slide Objective
The syntax for dynamically allocating standard data types was easy either in a function or in a class. The syntax for allocating user-defined data types is consistent and therefore very easy.



- new Invokes the Appropriate Constructor
- Only delete Invokes the Destructor
- Dynamically Allocated Arrays of Objects Must Use the Default Constructor

The **new** and **delete** operators can be used in similar ways to dynamically allocate and deallocate objects:

```
Rectangle *pr1 = new Rectangle;
. . .
delete pr;
```

Since the compiler is not given any initialization information, the default constructor will be used to build the object referenced by **pr**. If you want to initialize this object using a different constructor, arguments can be supplied:

```
Rectangle *pr2 = new Rectangle(2,7,10,-10);
```

Arrays of objects can also be dynamically created, much like you did with standard types:

```
Rectangle *pr3 = new Rectangle[x];
```

Note The default constructor must be used when "newing" an array of objects; no other syntax is permissible. However, to circumvent this limitation, you can declare an array of pointers, then **new** each element separately:

```
Rectangle *apr[10];
apr[0] = new Rectangle(3,3,5,5);
. . .
```

Demo

Note Close all files and projects. Open DYNOBJ.MAK found in \DEMOS\MOD19. You'll use this project to access the file DYNOBJ.CPP.

The Project also uses RECT.H and RECT.CPP. These files are un-modified from earlier demos. No lines were added or modified in either file except to denote their new locations in \demos\mod19. Neither the constructor or destructor nor member functions have been modified to use dynamic memory.

Open the file DYNOBJ.CPP found in \DEMOS\MOD19.

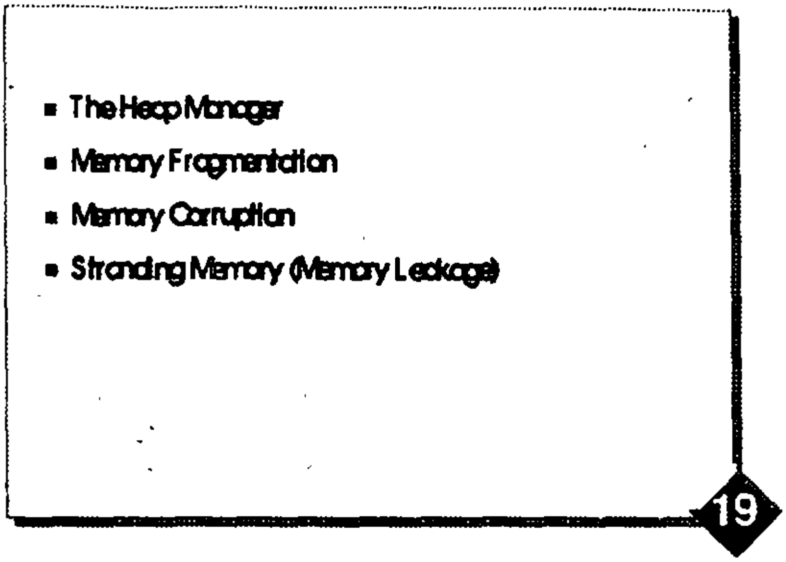
```
// DYNOBJ.CPP found in \demos\mod19
// Dynamically allocates and deallocates objects.
#include <iostream.h>
#include "rect.h"
#include <stdlib.h>
                                // function prototype
void CheckNull(void*);

void main()
{                               // Create a default rectangle
                                // dynmically in the heap
    Rectangle *pr = new Rectangle;
    pr->Draw();
    delete pr;                  // Release the memory
    cout << endl;
                                // Re-use the pointer, pr, to
                                // create another Rectangle
    pr = new Rectangle(4,14,100,-100);
    pr->Draw();
    delete pr;                  // Release the memory
    cout << endl;
    unsigned int nNbrRects; // prompt the user for a number
    cout << "How many Rectangles would you "
            "like in the array? ";
    cin  >> nNbrRects;
                                // Using pr again, allocate an
                                // array of Rectangles with the
    pr = new Rectangle[nNbrRects]; // user's size
    CheckNull(pr);              // error checking
    for (unsigned int i = 0; i < nNbrRects; i++, pr++)
        pr->Draw();             // display each rectangle
    delete [] pr;               // Release the array memory...
                                // Q: Why the [] notation?
    cout << endl;
    pr = new Rectangle;  // Q: When is this one destroyed?
    cout << "\nEnding main()" << endl;
}

```

(continued)

```
void CheckNull(void* pv)
{
    if (pv == NULL)
    {
        cerr << "\nERROR: Heap Allocation Failure!"
             << endl;
        exit(1);
    }
}
```

Dynamic Memory Issues

Slide Objective
Explain cares and concerns when dealing with dynamic memory:
IT IS PREFERRED TO HAVE CLASSES MANAGE ALLOCATIONS.
The last example showed it's not a requirement. The goal is to make programmers aware of the issues, not to scare them away from dynamic memory.



- The Heap Manager
- Memory Fragmentation
- Memory Corruption
- Stranding Memory (Memory Leakage)

Delivery Tips
Although C++ does not provide garbage collection, it is fairly easy to implement such a scheme inside your class.

The heap is managed by a small function that is added to your program by the linker. Implementations of this manager tend to be very simple and efficient. Typically, for every heap block that exists a table entry is made. That entry contains the starting address and size of the block. When a block is deleted, the table is searched for the pointer address. If a match is found, the block of bytes is freed.

The heap generally grows upward in memory, but in a program that allocates and deallocates many different-sized objects, it is very common for small unused areas in the heap to appear after some time. This is called memory fragmentation, and it can result in new returning NULL when enough total memory exists to satisfy an operation. This memory is not, however, contiguous.

The heap is fragile in other ways. For example, it is relatively easy to ruin the operation of the heap manager by doing any of the following:

- Deleting the same non-NULL pointer more than once without newing in between
- Deleting an invalid pointer
- Overwriting the heap manager's data structures

Note that it is safe to delete a NULL pointer; this operation does nothing. After the heap has been corrupted, dynamic memory operations are not guaranteed to work correctly.

Another serious problem can occur in a program if memory is allocated but not deallocated. This is called memory leakage. If a program runs for a sufficient time, this condition will cause a program to run out of heap space. Even though the operating system will release a program's normal resources when it ends, always use proper etiquette and delete outstanding variables.

Lab 16: Dynamic Memory

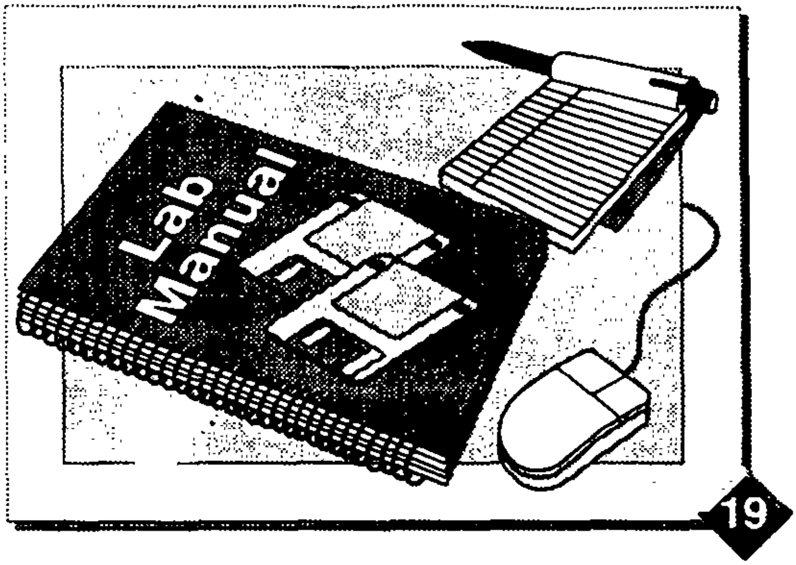
Slide Objective

Execute the lab solution for Exercise 1.

Explain that the string is new in one member function, displayed in another, and deleted in the d'tor.

Set the lab objectives.

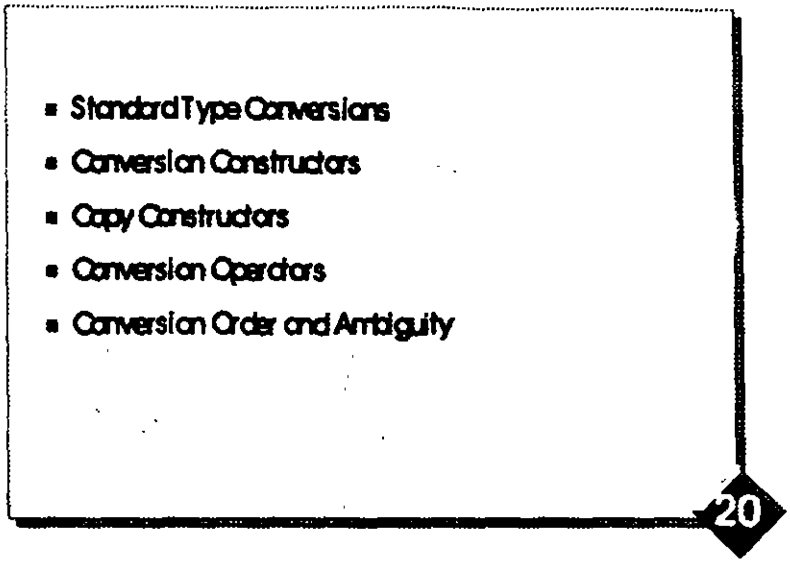
Ask students to read the lab scenario.



Module 20: Conversions

Σ Overview

Slide Objective
Provide an overview of the module contents.



- Standard Type Conversions
- Conversion Constructors
- Copy Constructors
- Conversion Operators
- Conversion Order and Ambiguity

Module Summary

You learned about standard C/C++ data types in the basics module, and a little about how the compiler handles expressions with mixed data types. In the modules on classes, you also saw how to create user-defined data type instances by invoking special member functions called constructors. In this module, you will learn about the possible categories of type conversions one can encounter in C++, namely

standard => standard

standard => abstract

abstract => abstract

abstract => standard

and how we, as class users, can determine when and what conversions will occur.

Objectives

Upon completion of this module, you will be able to:

Delivery Tips
Execute the lab solution to show new conversions with the date class.

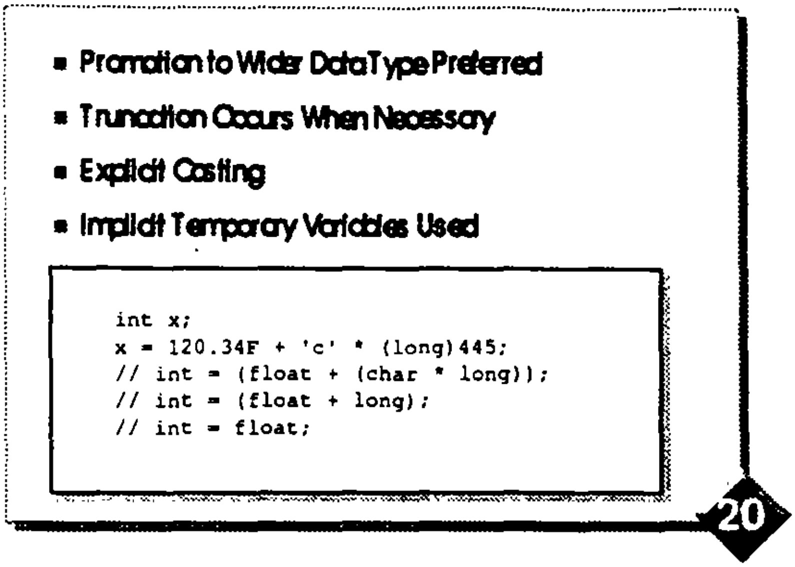
- Explain promotion and truncation.
- Use type casting.
- Use conversion constructors.
- Use copy constructors.
- Use conversion operators.

Lab

Building Streams in the Heap

Standard Type Conversions

Slide Objective
Review promotions and truncations, adding more detail about rvalue and lvalue operations. Explain casting and temporary variables.



- Promotion to Wider Data Type Preferred
- Truncation Occurs When Necessary
- Explicit Casting
- Implicit Temporary Variables Used

```
int x;
x = 120.34F + 'c' * (long)445;
// int = (float + (char * long));
// int = (float + long);
// int = float;
```

Delivery Tips
Use the code example to explain promotion and truncation.

Promotion

You saw in a very early module that when the compiler encounters an expression with mixed data types, it may be forced to promote the narrower data types to wider ones. For example, in the arithmetic expression on the right side of the assignment above, the first subexpression, a multiplication, demands the promotion of the **char** to be a **long**, resulting in a **long** product. Next, the addition demands promotion of this **long** product to a **float**. The result of the right-hand side of the arithmetic expression is of data type **float**.

Assigning a 'truncated' constant expression *always* generates a warning. Using a cast controls and documents the activity (but the warning will remain.)

Truncation

During assignment, and passing and returning function arguments, the compiler may not have the option of promoting; the target data type may be determined. These cases can result in truncation or narrowing of data types. In the foil example, the right-hand side **float** value must be truncated to an **int** value.

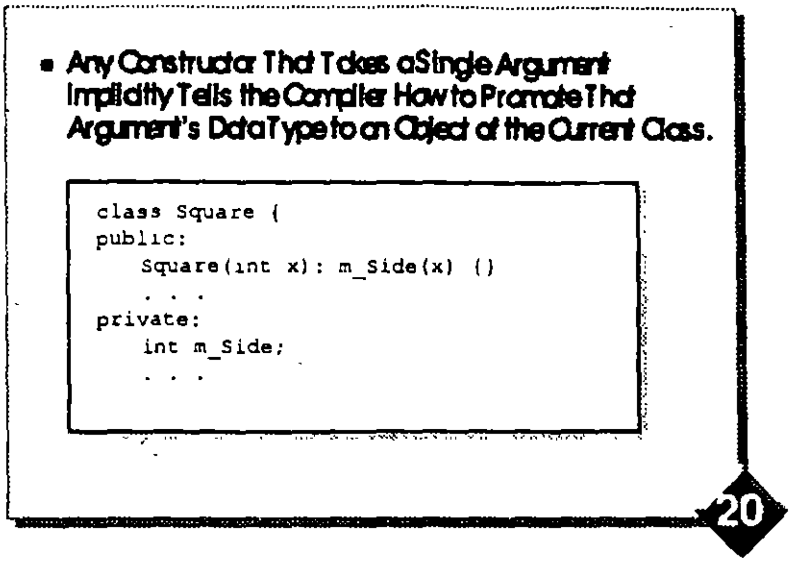
The cast operator can be used to explicitly control this process. It results in an rvalue.

Implicit Temporaries

C++ is a statically typed language. One result of this is that variables do not change data types in a program. When variables or values are promoted or truncated, the compiler often must generate an unnamed variable of the appropriate type for temporary storage.

Conversion Constructors

Slide Objective
Introduce single-arg c'tors as adding promotion features to the class.



- **Any Constructor That Takes a Single Argument Implicitly Tells the Compiler How to Promote That Argument's Data Type to an Object of the Current Class.**

```
class Square {
public:
    Square(int x): m_Side(x) {}
    . . .
private:
    int m_Side;
    . . .
```

Delivery Tips
Use terms:
cast,
temporary object,
and conversion.

Avoid terms:
Copy c'tor and assignment operator.

A conversion constructor is any constructor that takes a single argument. In the example above, the constructor for `Square` takes a single-integer argument. A conversion constructor can be implicitly used by the compiler whenever it needs to do the implied promotion. Examine the following statements:

```
Square s1(10), s2(100);
s1 = s2;    //ok - assignment
s1 = 100;   //ok - implicit conversion via c'tor
```

You might suspect that the third would give you an error message since structures and class instances can normally only be assigned to like objects. However, with the constructor, we have given the compiler the implicit ability to convert an int to a `Square` temporary object. The assignment then occurs, and finally the temporary `Square` object is destroyed.

This conversion can also be forced by invoking the constructor in two explicit ways:

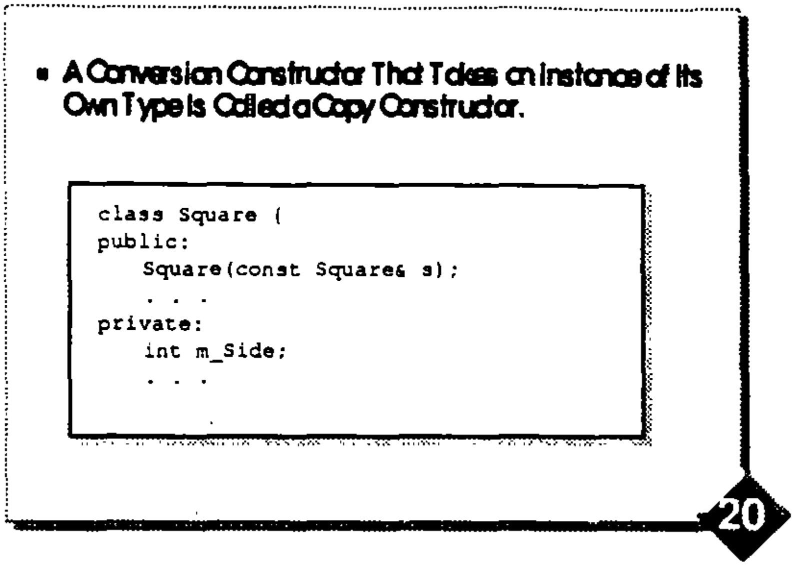
```
s1 = (Square)100;
s1 = Square(100);
```

Key Points
Each example creates a temporary object.
1st example looks like a cast.
2nd example look like a c'tor.

Copy Constructors

Slide Objective
Extend the conversion topic to include those 1-argument c'tors that take an instance of their own type, hence, Copy c'tor.

NOTE: "const" is not required, but reference is typical.



- A Conversion Constructor That Takes an Instance of Its Own Type Is Called a Copy Constructor.

```
class Square (
public:
    Square(const Square& s);
    . . .
private:
    int m_Side;
    . . .
```

A copy constructor tells how to create a new object out of a previously existing object:

```
Square s1(100);
Square s2(s1);  // invoke copy c'tor
```

Key Points
For classes with pointers, a shallow copy simply makes another pointer. Assuming the pointer addresses dynamic memory, trouble begins when the first object is d'tored. The memory is likely to be deleted. The shallow-copied object remains with a pointer to trouble!

The compiler supplies a default copy constructor only if a user-defined one is not provided. The default copy constructor simply does a memberwise copy of values, just as occurs in structure variables.

Even if you do not explicitly use a copy constructor in a program, the compiler may implicitly use it in the following instances:

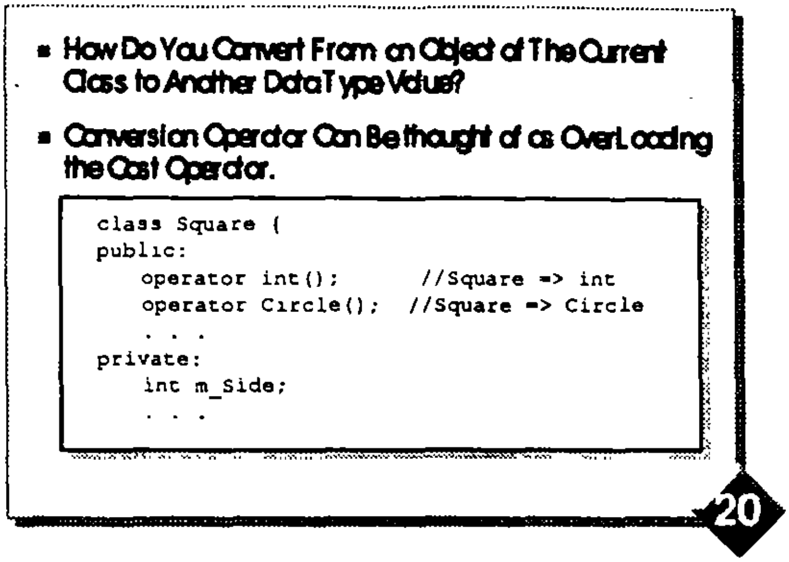
- to pass an object by value
- to return an object by value
- for temporary object creation

For many classes, explicit copy constructors are not needed. However, if a class does dynamic memory allocation within its c'tor, and deallocation within the **d'tor**, as a general rule, it will need an explicit copy **c'tor** (as well as an overloaded assignment operator).

A user-supplied copy **c'tor** always takes a single argument (it meets the criteria for a conversion **c'tor**) that is a constant reference to an object of the same type of the class. Since a copy **c'tor** is invoked implicitly by the compiler when it needs to perform call-by-value, the copy constructor must not use call-by-value, or else an infinite recursion would result.

Conversion Operators

Slide Objective
Introduce the use of the "operator" keyword to create a member function that controls the cast of a class to a standard type or another class type.



- How Do You Convert From an Object of The Current Class to Another Data Type Value?
- Conversion Operator Can Be thought of as OverLoading the Cast Operator.

```
class Square {
public:
    operator int();       //Square => int
    operator Circle();  //Square => Circle
    . . .
private:
    int m_Side;
    . . .
```

Sometimes you want to allow the user to convert an object of the current class to an object of some other class or to a standard type. Constructors only take us the opposite direction—from some other data type to the current class type. C++ allows a special group of member functions, conversion operators, to be defined to do just this.

For example, in the code above the conversion operators tell the compiler how to convert a Square to an int and a Circle object, respectively. These operators can be invoked implicitly:

```
x = 55 + s1 + s2;
```

or explicitly

```
Circle c1(s1), c2((Circle)s1);
```

Delivery Tips
General use of operator overloading is beyond the scope of this course!

Caution Extreme care must be taken when you provide conversion constructors and operators.

Although supplied here as a syntactic example, it is doubtful that the `Square => int` conversion operator in the foil makes good design sense.

Demo

CONVERT.CPP is found in \DEMOS\MOD20.

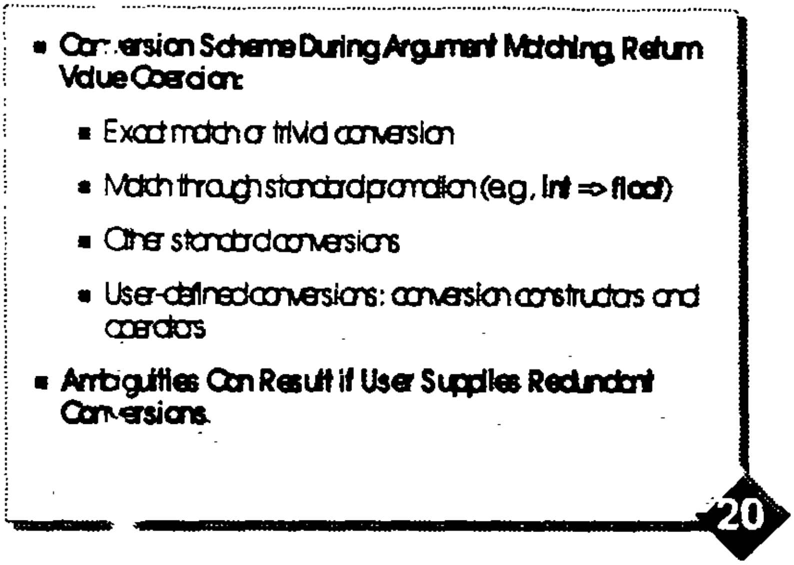
```
/ CONVERT.CPP in \Demos\mod20
// Using conversion c'tors and operators.
#include <iostream.h>

/*************** Class Declarations *********************/
// Circular forward reference needs declaration (pun
// intended). Circle must be predefined for Square.
class Circle;

class Square
{
public:
    Square(int x=0);        // conversion c'tor
    Square(const Square&); // copy c'tor
    Square(const Circle&); // conversion c'tor
    operator Circle () const; //conversion operator
    void Display() const;
private:                    // implementation
    int m_Side;   // Square's have a side dimension
};

class Circle
{
public:
    Circle(int d)           //conversion c'tor
        : m_Dia(d)
        { cout << "Circle Conversion c'tor (int)\n"; }
    int GetDia(void) const { return m_Dia; }
    void Display(void) const;
private:                    // implementation
    int m_Dia;    // Circle's have a diameter dimension
};

/******** Member Functions Definitions ***************/
Square::Square(int x)
    : m_Side(x)
{
    cout << "Square Conversion c'tor (int)\n";
}

Square::Square(const Square& s)
    : m_Side(s.m_Side)
{
    cout << "Square Copy c'tor (Square&)\n";
}

Square::Square(const Circle& c)
{
    m_Side = c.GetDia();
    cout << "Square Conversion c'tor (Circle&)\n";
}

```

(continued)

```
Square::operator Circle () const
{
    cout << "Square => Circle operator\n";
    return Circle(m_Side);    //Invokes Circle(int)
}

void Square::Display(void) const
{
    cout << "Display square of side " << m_Side << endl;
}

void Circle::Display(void) const
{
    cout << "Display circle of diameter " << m_Dia << endl;
}

/****************** Test Function *********************/
int main()
{
    cout << "Construct two circle objects:\n";
    Circle c1 (33),
           c2 (66);
 // Circle cnot;       // error: no default c'tor
    cout << "Construct two square objects:\n";
    Square s1,
           s2 (25);
    cout << "Construct s3 from s2 (25):\n";
    Square s3 (s2);   // copy c'tor
    s3.Display();
    cout << "Construct s4 from c1 (33):\n";
    Square s4 (c1);   // conv c'tor
    s4.Display();
    cout << "Construct c3 from s1 (default):\n";
    Circle c3 (s1);   // how does this work?
    c3.Display();
    cout << "Assign a circle to a square, s1 = c2\n";
    s1 = c2;          // conv c'tor for temp object
    cout << "Assign a square to a circle, c1 = s2\n";
    c1 = s2;          // how does this work?
    return 0;
}
```

Conversion Order and Ambiguity

Slide Objective
Summarize all students know about conversions, then introduce ambiguities.



- **Conversion Scheme During Argument Matching, Return Value Coercion:**
  - Exact match or trivial conversion
  - Match through standard promotion (e.g., **int => float**)
  - Other standard conversions
  - User-defined conversions: conversion constructors and operators
- **Ambiguities Can Result if User Supplies Redundant Conversions.**

Key Points
Detail the 4 areas where conversions occur:
1) Exact or nearly exact
2) Promotion (presented Day 1)
3) Other standard conversions (truncation, specific pointer to non-specific pointer, and from derived-type to base-type.)
4) Through user-defined conversions.

Introduce "ambiguities": multiple ways to perform the same conversion, as an error at compile time.

Where the compiler detects type mismatches, especially in function calls, it attempts to coerce or cast data types to achieve a match. The preferred order is shown above.

Exact matches need no conversions. Trivial conversions are non-**const** to **const**, reference to object, and an array to pointer of the same type.

Standard promotions were covered in an early module; they involve "widening" a data type.

Other standard conversions cover three areas:

- Standard truncation (for example, **float** => **int**)
- Specific pointer type => void*
- Conversion up the public hierarchy (from a derived type to a base type)

Note that the implicit conversions from specific* => void*, and non-**const** => **const** are one-way; the reverse conversions can only be accomplished with an explicit cast operation.

Conversion operators and conversion constructors were featured in the preceding demo.

Ambiguities can occur when a user supplies both conversion constructors and conversion operators for a class. Unfortunately, normally the compiler will only catch these errors when the ambiguous conversion is attempted, not when the offending design is implemented.

Demo

AMBIG.CPP is found in \DEMOS\MOD20.

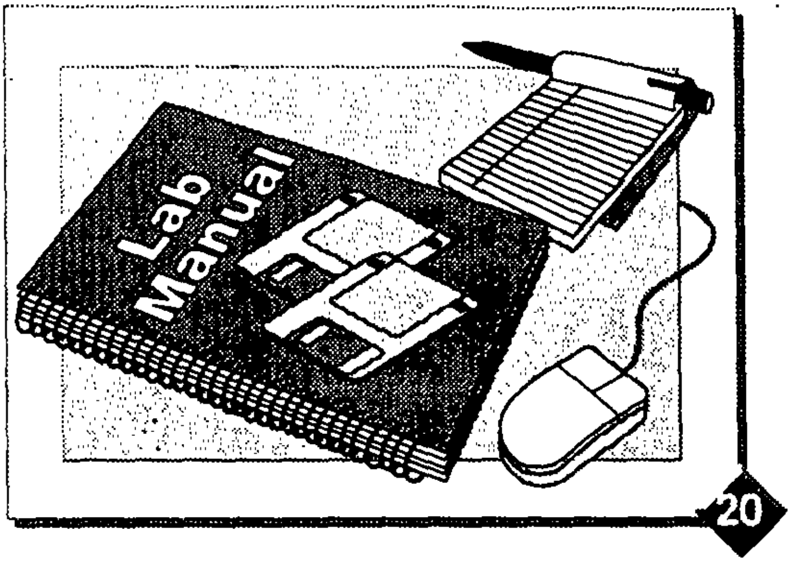
```
// AMBIG.CPP in \demos\mod20
// Demonstrates errors from ambiguous conversions.
/*    The member functions:                       *
 *                Square::operator Circle();      *
 *                Circle::Circle(Square&);        *
 *    do the same thing, and are thus ambiguous.  */
#include <iostream.h>

/***************** Class Declarations ******************/
class Circle;  // Predefine class Circle for use in Square

class Square
{
public:
 // Square();              // Ambiguous Overloading
    Square(int x=0);       // int => Square
    Square(Square&);       // copy c'tor
    Square(Circle&);       // Circle => Square
    operator Circle();     // Square => Circle
    int GetSide(void) { return m_Side; }
private:                   // implementation
    int m_Side;  // Squares have a side dimension
};

class Circle
{
public:
    Circle(int d)
        : m_Dia(d)         // int => Circle
        { cout << "Circle Conversion c'tor (int)\n"; }
    Circle(Square&);       // Square => Circle
    int GetDia(void) { return m_Dia; }
private:                   // implementation
    int m_Dia;  // Circles have a diameter dimension
};

/*********** Member Functions Definitions ***************/
Square::Square(int x)
    : m_Side(x)
{
    cout << "Square Conversion c'tor (int)\n";
}

Square::Square(Square& s)
    : m_Side(s.m_Side)
{
    cout << "Square Copy c'tor (Square&)\n";
}
```

(continued)

```
Square::Square(Circle& c)
{
    m_Side = c.GetDia();
    cout << "Square Conversion c'tor (Circle&)\n";
}

Square::operator Circle()
{
    cout << "Square => Circle operator\n";
    return Circle(m_Side);    //Invokes Circle(int)
}

Circle::Circle(Square& s)
{
    m_Dia = s.GetSide();
    cout << "Circle Conversion c'tor (Square&)\n";
}

/******************* Test Program **********************/
void Func1(Square s);     // function prototypes
void Func2(Circle c);

int main()
{
    cout << "Construct a circle object, c1.\n";
    Circle c1 (33)
    cout << "Const     a square object, s1.\n";
    Square s1 (67)
    cout << endl
         << "Func'     as a Square argument.\n"
         << "Call      uncl() with a square.\n";
    Func1(s1);                // Square => Square (by value)
    cout << "Ca:      'uncl() with a circle.\n";
    Func1(c1);                // Circle => Square
    cout << en:
         << "Fu      kes a Circle argument.\n";
 // UNCOMMENT Th.    LINES
 // cout << "Cal..., Func2() with a square.\n";
 // Func2(s1);                // Square => Circle
    cout << "Calling Func2() with a circle.\n";
    Func2(c1);                // Circle => Circle (by value)
    return 0;
}

void Func1(Square s)
{
    cout << "Func:  calling GetSide()\n";
    s.GetSide();
}

void Func2(Circle
{
    cout << "Func2    ..ing GetDia()\n";
    c.GetDia();
}
```

Lab 17: Building Streams in the Heap

Slide Objective
Execute the lab solution.
Set the lab objectives.
Ask students to read the lab scenario.



Lab Manual

Introduction to Microsoft® Visual C++™ and Object-Oriented Programming

Microsoft Corporation
Course Number: 280
Part Number: 5362A
Master Part Number: 5369A

1a

Course Number: 280
Part Number: 5362A
Master Part Number: 5369A

Contents

Lab 1: Identifying the Components of a Class

Objectives

At the end of this lab, you will be able to:

- Identify the entities and activities of a simple object.
- Identify the state and behavior of a class.
- Determine "is a kind of a" and "is part of a" characteristics of a class.
- Identify "behaviors" and "communication" characteristics of a class.

Scenario

Today is your first day as a Lead Analyst for a small manufacturing corporation called ISM, Inc., which stands for Industrial Smoke and Mirrors. Although the company is small, the domestic and international market demand shows a large sales potential for the products.

Mid-morning news around the coffee area included second-hand reports from an early-morning management meeting. Rumor has it that the CEO clobbered the Purchasing Manager complaining, "Too many unusable parts are stocked in inventory and there are frequent delays getting the right parts to manufacturing." The Finance Manager was the next target: "A lack of purchasing controls has delayed product assembly, and rush orders have increased our cost of goods sold."

Back at your desk, electronic mail has arrived from your boss, the Manager of Information Systems, concerning a meeting with you. After a five-minute meeting with the boss, you're back at your desk, staring at your meeting notes. Although the request sounds simple, you realize that the problem described in your notes may take months to solve.

Your mild-mannered manager has given you until tomorrow morning to answer the following question: "What do we need in an inventory system?"

Estimated time to complete this lab: 30 minutes

Exercise 1
Identifying the Entities and Activities in a Simple Inventory Object

Step 1

Run the completed version of the class application. It is located in the directory \STUDENT\LAB01.

Step 2

Compose a list of items that would be needed in an inventory-control system. Expect that this system will need to interface purchasing (adding new inventory) and both sales and manufacturing (removing existing inventory).

Take a few minutes to compose the list. Soon, we'll review and share ideas with other developers in the group.

Note For all of the code-based labs, answers will be located in a subdirectory on your student disk. For these two exercises, the answers will found at the end of this lab.

Exercise 2
Identifying Objects and Their Behaviors

Scenario

The overall list of items that are needed in an inventory-control system has been approved. The Manager of Information Systems wants to know what the next step is, and wants an estimate for completion of a new system.

You're back at your desk, staring at your meeting notes. You realize the request requires further research.

Note As with the first exercise, there is no clear wrong or right answer. The purpose of this lab is to get you to start thinking about objects and their traits rather than about coding. That will come soon enough!

Step 1

Given the list of items needed in a simple inventory system, you are to develop a set of classes that implement it. The system must keep track of the following:

1. Part number, name, quantity, and cost
2. Inventory adjustments (additions fed from purchase orders, and subtractions as in·.ntory is sold or used in manufacturing)
3. Adjustments in price (including purchases at various prices and various currencies)

4. Bill of materials (built around part numbers to show the explosion of finished goods back to their component parts)

Use this data to identify the items that might become objects in the new system. Keep track of messages or requests that these objects would respond to during interactions with other objects.

Step 2

Use the attached sheets to help shape your ideas. Four classes are identified on the following working cards. Each of the four cards is incomplete. Review the information provided and add other details concerning the information each class will need to be functional.

If you have identified other items that may become classes, you may add those on the subsequent blank cards.

Step 3

The "behavior" and "communication" sections are missing numerous entries that will make the inventory system functional. Add entries to those sections.

As an approach, imagine the conversations that would take place between objects. Try working through various scenarios, such as inventory from a purchase order being received at a loading dock. What information comes in? What behaviors should occur? Don't become burdened with details; view the system abstractly from a mile away.

And, finally, remember that we don't have time to truly design the system this week (or this month)! In design, you won't need any algorithms or accounting rules, just a good imagination. Besides, if you reach a dead end trying to resolve how the Inventory system should interact with another software system, you can always make it the other system's problem! We're trying to build a mind-set that will get you to look at problem from a different perspective.

<table>
<tr><td colspan="2">Class Name: Inventory   Abstract/ [✓ Concrete]</td></tr>
<tr><td colspan="2">Parent:
Children:</td></tr>
<tr><td>Behavior:
Purchase ( )
Sell ( )
TriggerEOQOrder ( )
Load ( )
Store ( )</td><td>Communication:
Quantity In Stock ( )</td></tr>
<tr><td colspan="2">Embedded Objects:
Date, Money, and PartID</td></tr>
</table>

Class Name: Money	Abstract / Concrete
Parent: Children: Dollars, Pounds, Deutsche Marks	
Behavior: Display Display Money Numerically Display Money in Text	Communication:
Embedded Objects:	

Class Name: Date	Abstract / Concrete
Parent: Children:	
Behavior: Display ( )	Communication:
Embedded Objects:	

Class Name: PartID	Abstract / Concrete
Parent: Children:	
Behavior: AdjustPrice ( )	Communication: GetPrice ( )
Embedded Objects:	

Class Name:	Abstract / Concrete
Parent: Children:	
Behavior:	Communication:
Embedded Objects:	

Class Name:	Abstract / Concrete
Parent: Children:	
Behavior:	Communication:
Embedded Objects:	

Class Name:	Abstract / Concrete
Parent: Children:	
Behavior:	Communication:
Embedded Objects:	

Summary

This objective	Was met by...
Identify the entities and activities of a simple object	Exercise 1
Identify the state and behavior of a class	Exercise 2, Step 2
Determine "is a kind of a" and "is part of a" characteristics of a class	Exercise 2, Step 2
Identify "behaviors" and "communication" characteristics of a class	Exercise 2, Step 3

Possible Answer for Exercise One

Even a relatively simple inventory system will have a large number of possible components. For the purposes of this class and this lab, your list of entities for the inventory system should look something like this:

Cost

Price

Quantity

Location or Bin

Raw Material or Finished Good

Current Requirements

Description (size, dimensions)

Purchase Date

Age

Delivery Lead Time

Minimum Amount (also known as EOQ)

Supplier or Vendor

Requestor

Most of these specific entities will show up in later labs.

Possible Answer for Exercise 2

Below is a first pass at a design for the classes in the inventory system. It is only a first pass. You may or may not have some or all of the data we listed. That's not the point. Our goal is to give you a feel for some of the possible data members, inter-class communications and activities that will probably show up in these classes.

Class Name: Inventory Abstract / [✓ Concrete]

Parent:

Children:

Behavior:
ProcessPurchase ( )
ProcessSalesOrder ( )
TriggerEOQOrder ( )
Load ( )
Store ( )

Communication:
QuantityInStock ( ) => quantity
OrderQuantity ( ) => quantity
Price (and cost) => money
Date => date
OrderLeadTime => date range
PurchaseOrders => quantity and cost
Sales Orders ( ) invalid if > Quantity

Embedded Objects:
Date, Money, PartID

Class Name: Money Abstract / [✓ Concrete]

Parent:

Children:
Dollars, Pounds, Deutsche Marks, and so on.

Behavior:
Displays:
as NumericAmount( )
as AlphaTextAount( )
Add Amount(s)
Multiply Amount(s)
Load ( )
Store ( )

Communication:
AdjustAmount( ) => Exchange Rate
CurrencyConversion( )
SetAmount( ) => Money
Display ( )

(See Inventory class.)

Embedded Objects:
Currency symbol, Field Separator Characters

Class Name: Date Abstract / [✓ Concrete]

Parent:

Children:

Behavior:
- Display
  - as Month/Day/Year ( )
  - as Day/Month/Year ( )
  - as AlphaText ( )
- Compare ( ) and Validate ( )
- DateSpan or Range ( )
- GetCurrentDate ( )
- Load ( ) and Store ( )

Communication:
- Display ( )
- JulianValue ( ) => numeric
- SetMonth ( ) => month
- SetDay ( ) => day
- SetYear ( ) => year

(See Inventory class)

Embedded Objects:

Class Name: PartID Abstract / [✓ Concrete]

Parent:

Children:

ImportedPart, and DomesticPart

Behavior:
- GetVendor ( )
- GetPrice ( )
- SetUnitOfMeasure ( )
- Load ( )
- Store ( )

Communication:
- Display ( )

Embedded Objects:

Class Name: ImportedPart	Abstract / [✓ Concrete]
Parent: PartID Children:	
Behavior: CalculatePrice ( )	Communication: GetExchangeRate ( ) => rate SetExchangeRate ( ) <= rate SetPrice ( ) <= money (See PartID class)
Embedded Objects: ExchangeRate	

Lab 2: The Basics

Objectives

At the end of this lab, you will be able to:

- Use **#include** to access precompiled header files.
- Use **#define** to create manifest constants.
- Use **cout** to output to the screen.
- Use the multiple-insertion operations with **cout**.
- Create a **main** function with a return value.

Before You Begin

Before accessing the source file, close any files or projects that may be open. If you're not sure whether Visual Workbench has other files open, display the File menu. If the Close option is available, choose it. If it is unavailable (dimmed), no file is open. Do the same thing from the Project menu.

Scenario

Microsoft® Visual C++™ programs do not have the rigid structure offered in many other languages. As your familiarity with the C++ language grows, you'll discover that most of the conventions used in this module are "required." Through experience, you will learn that other means exist, but all these conventions add to the readability and maintainability of your code.

Estimated time to complete this lab: 20 minutes

Exercise 1
Writing a Simple C++ Program

An empty source file, SIMPLE.CPP, exists in the \STUDENT\LAB02 subdirectory. You will complete the code statements to create a small program that follows the basic program structure described in this module.

Σ **To open a file**

Open the SIMPLE.CPP file by following these steps.

1. From the File menu, choose Open.

   The Open File dialog box appears.
2. In the Directories box, select the \STUDENT subdirectory. (If it is not visible, you may have to first select the root directory, C:\, to find \STUDENT.)
3. Select the \LAB02 subdirectory. A few files should appear in the File Name box.
4. In the File Name box, select SIMPLE.CPP and choose the OK button.

The SIMPLE.CPP file does not contain much of a head-start. The following steps will detail the statements that must be added. Each step is associated with a comment in the source file noted as: `// TO DO #n`.

Step 1

A program that interacts with the user through input or output will typically use the C++ iostreams. Add the preprocessor directive that will cause the compiler to include the header file definitions in IOSTREAM.H within your application.

Step 2

For readability, add a manifest constant, BEGIN_INV, with the value of last year's inventory final balance: $123,500. (Be careful. The $ and , characters can't be mixed with numeric data in C++.)

Step 3

Write the definition line for the **main** function using the standard conventions noted in the lecture.

Step 4

Display the following single line of text after 8 spaces on the screen:

```
I.S.M., Inc.
```

Your display statement should advance to the next line using the `\n` notation that was used in HELLO.CPP.

Step 5

Display a second line of text:

```
1994 Beginning Inventory: $
```

and the amount, using the manifest constant BEGIN_INV. Your display should advance to the next line, although this is the end of the program.

Step 6

The program is complete. Return a 0 to the operating system to indicate success.

Step 7

Build, execute, and test your application.

Summary

This objective	Was met by...
Use **#include** statements to access precompiled header files	Step 1
Use **#define** statements to create manifest constants	Step 2
Create a **main** function with a return value	Step 3
Use **cout** to output to the screen	Step 4 and Step 5
Use the multiple-insertion operations with **cout**	Step 5

Lab 3: Using Statements and Expressions

Objectives

At the end of this lab, you will be able to:

- Declare variables.
- Declare variables with an initial value.
- Write a **do...while** loop that tests for a user's preferences.
- Write a simple **if** statement that tests user input for a range of values.
- Write output statements that inform the user about inventory quantities.
- Write simple arithmetic calculations using C++ syntax.

Scenario

Statements, expressions, and flow control will drive the processing and logic within your applications. To investigate processing and computational calculations, you'll build a small application that simulates inventory-processing and reports final results.

Estimated time to complete this lab: 30 minutes

Exercise 1
Declaring Variables and Using Flow Control

A skeleton source file, FORMULA.CPP exists in the \STUDENT\LAB03 subdirectory. In this file, you will write and exercise several looping, conditional, and computational constructs.

Step 1

Examine the existing preprocessor directives at the top of the source file. A manifest constant is provided: `ECONOMIC_ORDER_QTY` is the value `50`. Within the `main` function, two variables, `nTotalItemsSold` and `nBeginningInv`, are provided and initialized to `0` and `150`, respectively.

Add statements to declare local integer variables, `nBuyQuantity` and `nSellQuantity`, and a local character variable, `chTransType`.

Step 2

The global variable `lInventory` has no initial value, so assign `lIventory` the value of the `nBeginningInv` local variable. To prove the assignment worked, write a statement that displays the following and advances to the next line:

```
Begining inventory: nn items.
```

(where nn is the value of `lInventory`)

Step 3

Most statements within the `main` function are contained within a **do...while** loop that runs `while (chTransType != 'Q')`. Write a short, nested loop that prompts the user for a transaction type, `chTransType`, of Buy (`'B'`) or Sell (`'S'`), and allows the user to Quit (`'Q'`). The body of the loop is provided.

Step 4

The previous line input the user's sell quantity. Test that input value versus the inventory amount. Reject the Sales Order if it exceeds current inventory.

Hint Examine the processing for Buy amounts or purchase orders, if needed.

Step 5

Inventory levels should be maintained at a level supported by sales activity and an item's Economic Order Quantity. A manifest constant, `ECONOMIC_ORDER_QTY`, is provided. Add the conditional construct to test inventory. Display a warning message if the inventory is less than half an item's economic order quantity.

Step 6

Write a statement to calculate inventory rollover and display the value. Your calculation should divide the total items sold by the beginning inventory. The format for the display is

```
"Inventory turnover was nn times."
```

where nn is the result of the calculation.

Step 7

Build, execute, and test your solution.

Summary

This objective	Was met by...
Declare variables	Step 1
Initialize the value of variables	Step 2
Write a simple **do...while** loop that tests for a user's preferences	Step 3
Write a simple **if** statement that tests user input for a range of values	Step 4
Write simple output statements that inform the user about inventory quantities	Step 5
Write simple arithmetic calculations using C++ syntax	Step 6

Lab 4: Implementing Simple Functions

Objectives

At the end of this lab, you will be able to:

- Prototype and define a function.
- Call a function from within another function.
- Return a value from a function.
- Convert a block of statements to a function.

Scenario

Functions will eventually provide the methods, behaviors, and communication message-handling within the inventory-control system. As part of your preliminary research, investigate the implementation of functions in C++. You need to determine whether functions can easily handle various inputs and return values for your business situations.

Estimated time to complete this lab: 30 minutes

Exercise 1
Building Functions and Prototypes

A skeleton source file, FUNCTION.CPP, exists in the \STUDENT\LAB04 subdirectory. You will write and exercise several small functions to test data manipulations within different types of functions.

This program is similar to the formula program in the previous lab. Many of the blocks of statements have been packaged as functions, but others need to be completed. The user-processing of the application has not changed.

Step 1

Examine the existing statements at the top of the source file. A manifest constant is provided. Within the `main` function, several function calls exist.

Add statements to prototype the two functions called within the `main` function: `ProcessBuy` and `ProcessSell`. Those functions are defined below the body of the `main` function. Both functions return an integer to the calling routine.

Step 2

Write a statement to call the `ProcessBuy` function. The function returns an integer value representing the number of items purchased for inventory. Add that return value to update the inventory balance, `lInventory`.

Step 3

Write three statements to handle the processing from the `ProcessSell` function.

1. First, add a statement to call the `ProcessSell` function. It returns an integer value representing the number of items sold. Save that value in the variable `nSold`.
2. Add a statement that updates the inventory balance, `lInventory`.
3. Add a statement that updates the `nTotalItemsSold` variable.

Step 4

1. Locate the function body of the `ProcessBuy` function. Examine how it "returns" the purchase amount to the calling function.
2. Locate the `ProcessSell` function. Portions of this function need to be completed. Use a conditional statement to deny the Sales Order if the quantity exceeds the current inventory amount. You should display a message to the user and return a zero (indicating a rejected order). Alternately, if that quantity is available, return the sell quantity.

Note Your partially completed solution may be compiled and tested at this point.

Step 5

Locate the function body of the main function. Near the end of main, you'll recognize a display statement that calculates inventory turnover. To complete this step, convert that statement to a function: CalcTurnover. You need a statement to prototype the function and a statement to call the function. You also need to "package" that statement from main as a function body. The values of two variables, nTotalItemsSold and nBeginningInv, are needed within the CalcTurnover function.

Step 6

Build, execute, and test your final solution.

Summary

This objective	Was met by...
Prototype and define a function	Step 1
Call a function from within another function	Step 2, Step 3
Return a value from a function	Step 3, Step 4
Convert a block of statements to a function	Step 5

Lab 5: Using Structures to Encapsulate Data

Objectives

At the end of this lab, you will be able to:

- Declare a structure.
- Assign values to structure members.
- Access the contents of a structure's members.

Scenario

Structures are one of the logical frameworks C++ offers to encapsulate or package the data your applications will manage. Your development team will be seeking your guidance as they determine the data needs of the inventory system.

You realize that the inventory system will need to integrate with both Sales and Purchasing groups. Their systems rely heavily on three data items: time, cost, and quantity. C++ offers standard data types that can effectively handle quantity, but there are no data types to handle dates or money. In this lab, you will define a date structure.

Estimated time to complete this lab: 20 minutes

Exercise 1
Declaring and Accessing Data in a Structure

An incomplete source file, DATES.CPP, exists in the \STUDENT\LAB05 subdirectory. You'll write a structure to store date information and create a function to display the date in a format you prefer.

Step 1

Define a Date structure with storage for month, day, and year as data members.

Caution You may be tempted to use the **char** data type to store the day and month variables because they have small ranges. (Calendars typically have 31 or fewer days per month and 12 months per year.) Fight that temptation! In the future, you may want to perform operations that exceed the ranges allowed by **char**.

Step 2

Declare a global instance of the Date structure, named `dSolstice`, that represents this century's last summer solstice: June 21, 1999.

Step 3

Declare a local instance of the Date structure named `dToday` (within `main`, no initialization).

Step 4

Assign values to each member of `dToday` to represent today's date.

Note The answer solution shows today as 9/22/1994.

Step 5

Examine the `DisplayDate` function, looking at the prototype at the top of the source file and the calls inside of `main`. Write the function `DisplayDate` to display the Date structure passed as an argument. Use simple literals to delimit fields (such as "-" or "/") for now. We'll revisit this lab later to improve the display.

Step 6

Build, execute and test your final solution.

Summary

This objective	Was met by...
Declare a structure	Step 1, Step 2, Step 3
Assign values to structure members	Step 4
Access the contents of a structure's members	Step 5

Lab 6: Creating Classes and Member Functions

Objectives

At the end of this lab, you will be able to:

- Create a simple class using access specifiers.
- Write multiple **Get** member functions that retrieve values of class data members.
- Write a **Set** member function that modifies (assigns or mutates) class data members.
- Write a **Display** member function that manages output of data.
- Write a constructor member function to initialize data members.
- Write a destructor member function to perform cleanup.

Scenario

Using classes to encapsulate data members and member functions allows your system to integrate the methods that manage the data's behavior. The access specifiers, **public** and **private**, allow the class designers to control the interface to the class, locking out ill-behaved programs.

Knowing the international nature of your company, you're concerned about the approach your group should take to date-handling. Many operating systems, such as Microsoft® Windows™, offer helper routines for formatting dates, time, currencies, and so on. Eventually, your inventory system will be running on Windows—but in the interim, another solution needs to be devised.

Estimated time to complete this lab: 45 minutes

Exercise 1
Writing a Simple Date Class

An incomplete source file, DATETEST.CPP, exists in the \STUDENT\LAB06 subdirectory. You'll write a Date class with **constructor**, **destructor**, **Get**, **Set**, and **Display** member functions to handle data.

Step 1

Locate the header for the class, `Date`. The definition for the class is incomplete. Overall, this class will have `Display`, `GetMonth`, `GetDay`, `GetYear`, and `Set` member functions. The `Set` function will receive three integer variables and assign values to the data members `m_nMonth`, `m_nDay`, and `m_nYear`, respectively.

Complete the class definition. Prototype all member functions to allow access to the interface, but hide all data members from direct manipulation.

Step 2

The `Display` function should output the three data members in a format that fits your headquarter's date and time reporting standards. If you're unsure about those standards, use an MM/DD/YYYY format.

Step 3

Three member functions, `GetMonth`, `GetDay`, and `GetYear`, are needed to allow controlled access to each data member. A `main` function that invokes these three functions has been provided. (Yes, this interface may be modified in future implementations, but these functions are sufficient for now.)

Step 4

Your `Set` function should accept three values and initialize the three data members: `m_nMonth`, `m_nDay`, and `m_nYear`.

Step 5

Locate the `main` function that has been provided. The statements that follow "TO DO #5" are coded to reference an existing local instance of the Date class: `dMyDate`.

Add a statement to instantiate a Date object named `dMyDate`.

Step 6

In Step 2, you created a `Display` member function. To exercise the three `Get...` functions, write a statement that outputs the three data members in an alternate format. If your `Display` function ordered the member M/D/Y, either D/M/Y or D-M-Y would be acceptable.

Step 7

Build, execute, and test your application before continuing to Exercise 2.

Exercise 2
Adding Constructors and Destructors to a Class

Prerequisites

Exercise 1 should be complete and pass testing.

From the File menu, choose Save As. From the Save As dialog box, edit the filename to DATETST2.CPP. Choose the OK button.

Scenario

What was odd about the output from Exercise 1?

The output from the first `Display` function showed "undefined values" for the uninitialized Date object. Obviously a better solution exists—controlling the creation and deletion of the Date objects.

Step 1

Within the Date class, add a simple, no-argument constructor.

Below the class definition, add the body of the constructor function. It should output the message "`Date C'tor:\n`" and initialize all member data to zeros.

Step 2

Within the Date class, add the prototype of a destructor.

The destructor should output the message "`Date D'tor:\n`".

Step 3

Build, execute, and test your application. Notice the differences in output. Previously, the uninitialized Date displayed undefined results. Does your solution improve that display?

If time permits, continue to Exercise 3.

Exercise 3 (Optional)
Verifying That Your Data Is Secure

Prerequisites

Exercise 2 should be complete and pass testing.

From the File menu, choose Save As. From the Save As dialog box, edit the filename to DATETST3.CPP. Choose the OK button.

Scenario

You have a class that supposedly encapsulates and protects your data. Prove it. Add statements that try to directly manipulate the data.

Step 1

Within main, add a statement to declare another Date structure. Something like this will do:

```
Date ErrorDate;
```

Step 2

At the end of main, add statement(s) to directly change Date data members. They might look like this:

```
ErrorDate.m_nMonth = 10;
ErrorDate.m_nDay += 1 + ErrorDate.m_nYear;
```

Compile your application. Log the error numbers and messages below.

Error Code: Error Message:

__________ ______________________________

__________ ______________________________

__________ ______________________________

Summary

This objective	Was met by...
Create a simple class using access specifiers	Exercise 1, Step 1
Write a Set member function that accesses class data members	Exercise 1, Step 4
Write a **Display** member function that manages output of data	Exercise 1, Step 2
Write a **Get** member function that initializes class data members	Exercise 1, Step 3
Write a constructor member function to initialize data members	Exercise 2, Step 1
Write a destructor member function to perform cleanup	Exercise 2, Step 2

Lab 7: Tuning Your Member Functions

Objectives

At the end of this lab, you will be able to:

- Write overloaded constructors.
- Use default arguments.
- Use inlining to make your code run more efficiently.
- Use colon initialization for efficient object initialization.

Scenario

Based on your inventory system design, numerous small changes have been implemented in other systems that will interface the inventory system (especially the purchasing and sales order systems.)

The new purchase order system was purchased and installed, and it has been well received. The purchasing manager stopped by to thank you for your assistance installing that system—a job well done. "About the only trouble we've encountered has been order-entry errors on purchase-order dates. Sometimes a date field is skipped and unexpected values are filled in by the purchasing system." The purchasing manager left after issuing a teaser:

"I hope the inventory system is smarter about dates . . . "

Back at your desk, you recall that purchase orders may be triggered automatically by the inventory system, but may be held pending approval. Therefore, purchase orders may be cut with the current date, or entered with either a current or a future date.

You'll write a Date class and test application that handles the current date issue and avoids dates with invalid fields. Your Date class will fill in missing fields using today's date whether one, two, or all three fields are missing. If there is no initial value supplied, it should default to today's date. That will also allow order-entry personnel to skip entry on dates if they want today's date for an order.

Estimated time to complete this lab: 45 minutes

Exercise 1
Using Overloaded Functions and Default Arguments

A complete source file, TODAY.CPP. is in the \STUDENT\LAB07 subdirectory. Execute this program so that you are familiar with the issues the purchasing manager raised.

Step 1

At startup, the test application prompts the user to enter today's date. The global function `GetCurrentDate` is invoked. The body of the function consists of the last lines within this source file.

Add the prototype for the `GetCurrentDate` function. It takes no arguments and has no return value.

Step 2

The `GetCurrentDate` function sets three global variables: `nCurrMon`, `nCurrDay`, and `nCurrYear`. Add a statement to declare those global variables.

Step 3

Locate the class `Date` and the four prototypes of overloaded constructors. The no-argument constructor allows a Date object to be created with all zeros. The one- and two-argument constructors allow partial dates with zero fields. (While zero is a reasonable fill-value for an incomplete date, those fields must be correctly completed during Date construction.)

First, determine how those constructors could be overloaded to a single constructor with default arguments of value zero. (Yes, you should still allow zeros—the body of the constructor will replace them with current date values.) A single constructor with three default arguments may be called four different ways.

When you are satisfied with your new constructor prototype, either comment or delete the old prototypes.

Step 4

Locate the definitions for the four Date constructors. The default (no-argument), one-, and two-argument constructors all assigned a zero value to any data member that was not passed a value. The three-argument constructor, `Date::Date(int M, int D, int Y)` assigned the parameters to the data members.

Write the body of your new constructor from Step 3. For each data member, determine whether the value of the parameter is valid. If the passed value is zero, assign the appropriate global variable from Step 2 or accept the user input.

Step 5

The four original constructors for `Date` remain. Either comment or delete those functions.

Step 6

Build, execute, and test your application before continuing to Exercise 2.

Exercise 2
Inlining Functions

Prerequisites

Exercise 1 is complete and passes testing.

From the File menu, choose Save As. From the Save As dialog box, edit the filename to TODAY2.CPP. Choose the OK button.

Scenario

Your test application handles the current date issue and avoids dates with zeros. Your class could be tuned a bit more.

Step 1

Locate the class `Date` and the prototypes of all member functions. Determine which functions are candidates for inlining to avoid the overhead of function-call processing.

Your solution may use either implicit or explicit inlining conventions.

Step 2

Locate the class `Date` and its single constructor. The constructor accepts three values as parameters. Depending on the values, the body of the constructor either assigns the parameter or the **static** data member. The colon initialization syntax is more efficient than the assignment statement.

Your solution should use colon initialization in the constructor.

Since the assignment to the data members occurs prior to the body of the constructor, the body of the constructor can be changed to simply test for zero data members. If a zero value is encountered, assign the appropriate value from the global variables.

Step 3

Build, execute, and test your application.

Summary

This objective	Was met by...
Write overloaded constructors	Exercise 1, Step 3
Use default arguments	Exercise 1, Step 5
Use inlining to make your code run more efficiently	Exercise 2, Step 1
Use colon initialization in constructors	Exercise 2, Step 2

18

Lab 8: Static Class Members

Objectives

At the end of this lab, you will be able to:

- Use and intialize **static** member data.
- Use **static** member functions in classes.

Scenario

The previous Date program solved the invalid data problems—assuming the user entered a correct date when the test program started.

A few additions to the Date class could allow the class to ask the operating system for the current date. Using **static** members, all Date objects could be constructed with current, valid fields on startup.

You'll modify the Date class, and use a **static** member function and member data to handle the current-date issue.

Before You Begin

There's a big-picture issue to consider. Which operating system are you going to ask for today's date? Fortunately, C++ programmers are somewhat protected from the operating system. Libraries of functions that are tuned for various operating-system platforms already exist.

The classroom machines may be running MS®-DOS® version 5.0, 6.0, or above, with either Windows 3.0, 3.1, or above or Windows For Workgroups 3.1 or above. Alternately, this course may be presented without MS-DOS at all. Microsoft Windows NT™ could be used instead.

Two options exist: either call a standard C or C++ language library function, or create an object by using the Microsoft Foundation Class library. Both ways, you'll get accurate date information. If you use the language-library method, you'll code multiple lines using either a pointer to a structure or a binary bit-shifting technique to get the data. If you use the MFC library, you'll need one-line to create and initialize a CTime object.

Welcome to MFC.

Estimated time to complete this lab: 30 minutes

Exercise 1
Using Overloaded Functions and Default Arguments

A complete source file, TODAY3.CPP, is in the \STUDENT\LAB08 subdirectory. It is roughly equivalent to the last date lab program. The .EXE file in this directory conforms to the solution for this lab. You should execute it so that you are familiar with the new program flow.

Step 1

The last version of this application prompts the user to enter today's date by calling the `GetCurrentDate` function. That should change, two different ways.

1. Move the prototype for the `GetCurrentDate` function from the global area to within the class Date.
2. Modify the prototype. The function still takes no arguments and has no return value—but it is only called once for the class, and only modifies static data.

Step 2

The old `GetCurrentDate` function set values for three global variables: `nCurrMon`, `nCurrDay`, and `nCurrYear`. That should change three ways.

1. Move the declaration within the private area of class Date.
2. Modify the declaration so that one copy of each variable exists for the class.
3. Optionally (but still highly recommended), modify the variable names to reflect their new scope as members of class Date.

Step 3

Static data members must be initialized at file scope. Below the definition of class Date, initialize each **static** member to zero. Match the variable names from Step 2.

Step 4

Locate the body of the three-argument Date constructor. The prototype listed default arguments. The definition includes colon initialization. The body of the constructor determines whether the value of the data member is non-zero. That's all fine, except that Step 2 had you change the global names to member names.

With the constructor, match the variable names from Step 2.

Step 5

Locate `GetCurrentTime`. It has been moved above `main` (as of Step 1, it's now part of Date). Rather than asking the user to enter today's date, your program can get the current date from the MFC class CTime. Three changes are needed.

1. Change the definition of the function from file scope to class Date scope.
2. Declare a `CTime` object named `tm`, initialized using the CTime **static** member function **GetCurrentTime**.

Hint Enter CTime and press the F1 key. In the Search dialog box, select the MFC Library and choose the OK button. Use Help to find the `CTime` member `GetCurrentTime` example. You don't get extra credit for original code; copy the example. You deserve extra credit if you can copy and paste the example.

3. Use the `tm` object and `CTime` member functions to assign the current date value to each **static** data member. The `GetDay` example shows the three accessor functions you need.

Step 6

Locate the call to `GetCurrentTime` within `main`. That function may execute before any Date objects are created.

Change the line to call the Date class `GetCurrentTime` function.

Step 7

Build, execute, and test your application. The addition of the MFC includes requires an additional library in the build process. From the Options menu, choose Project. From the Project Options dialog box, choose the Linker button. In the Libraries text box, add the library **mafxcr** for a release mode project.

Summary

This objective	Was met by...
Write overloaded constructors	Exercise 1, Step 1
Use default arguments	Exercise 1, Step 5
Use **static** functions	Exercise 1, Steps 2, 3, and 5
Use inlining to make your code run more efficiently	Exercise 2, Step 1

Lab 9: Containment and Embedded Objects

Objective

At the end of this lab, you will be able to create a class that contains another class.

Scenario

Your development team at ISM has produced a few of the building blocks for an inventory system, specifically a Date class and a Money class. The inventory system will contain those classes and a part-identification class that hasn't been created yet. With these three building-blocks, you decide to create a simple `Inventory` class containing the above classes.

Estimated time to complete this lab: 30 minutes

Exercise 1
Embedding Objects

A complete source file, INVENTRY.CPP, is in the \STUDENT\LAB09 subdirectory. It has two classes, `Date` and `Money`, roughly equivalent to earlier lab and demo programs. Your new version will add a new, simple `PartID` class, and embed all three classes into a new, simple `Inventory` class.

Step 1

Locate the class `Money`. Notice that it has a no-argument and a two-argument constructor (both **int** arguments).

Locate the class `Date`. From a previous lab, you know the constructor for this class accepts 0 to 3 integers and may assign components of the current date to zero fields.

Locate the class `Inventory`. Above this definition, you'll write a new class, `PartID`.

Your class, `PartID`, should be very simple. The class will be revisited in future labs. To avoid data errors (as occurred with Dates), you decide that `PartID` should *not* have a no-argument constructor. Write a one-argument constructor that efficiently initializes the class's private data member, `m_nPartNbr`. The constructor should display a message when it runs.

Step 2

Write a class destructor that displays a message when it runs.

Step 3

Write a `Display` member function that displays the value of the private member `m_nPartNbr` when called.

Step 4

Locate the class `Inventory`. This class is partially complete. The declaration for the constructor is missing. Write the formal definition for the constructor so that it receives seven integers and efficiently initializes the data members.

This version of the `Inventory` class has four data members:

- an integer, `m_nQuantity`
- a PartID object, `pPartNbr`
- a Money object, `mCost`
- a Date object, `dOrig`

Step 5

Locate the `main` function. Declare an `Inventory` object named `iOakMirror` with the following beginning inventory:

■ Quantity	100
■ Part Number:	5
■ Cost:	$50.00
■ Origination:	today's date

Step 6

Build, execute, and test your application. The use of the MFC library for the `CTime` object requires an additional library in the build process. From the Options menu, choose Project. From the Project Options dialog box, choose the Linker button. In the Libraries text box, add the library **`mafxcr`** for a release mode project.

Summary

This objective	Was met by...
Create a class that contains a set of related classes	Exercise 1, Steps 1, 2, and 3

3

Lab 10: Working with Inheritance

Objectives

At the end of this lab, you will be able to:

- Use public inheritance.
- Extend a base class.

Scenario

The international nature of I.S.M., Inc. poses a problem when it comes to purchasing parts through Part Orders. The domestic suppliers provide parts with unit cost information. International suppliers frequently provide cost information based on a foreign currency, and they typically state an exchange rate.

The base class `PartID` maintains the part numbers used for purchasing and receiving. The `PartID` and the unit cost are both used in the inventory system.

Estimated time to complete this lab: 30 minutes

Exercise 1
Extending a Base Class

A skeleton application, PARTCOST.CPP, exists in the \STUDENT\LAB10 subdirectory. The base class, PartID, is complete. There is also an existing derived class, DomesticPart, that is nearly complete. You will finish the DomesticPart derived class and create another derived class: ImportedPart.

Step 1

Open and examine the file PARTCOST.CPP. The PartID base class maintains PartNbr and includes a Display function.

The DomesticPart class inherits from PartID and includes one data member: m_nUnitPrice.

Locate the DomesticPart Display member function. Complete this function. Overall, the output should list

```
PN: nn Price: ppp
```

where nn is the PartID and ppp is the unit price. (It is recommended that you use the DomesticPart Get function). PartID is the private member of the base class. The value is available through the GetID member function, and the first portion of output is provided by the Display function. Either way, you'll be calling the base class.

Step 2

You will complete a new derived class, ImportedPart, that has two data members: m_nUnitPrice and m_nExchangeRatePct.

Examine the constructors and destructor for the DomesticPart class. In a similar fashion, the ImportedPart class should build a base class object.

The ImportedPart Display function also should list

```
PN: nn Price: ppp
```

where nn is the PartID and ppp is the unit price. (It is recommended that you use the GetUnitPrice function rather than access the member data directly.)

Finally, complete the accessor function, GetUnitPrice. It must calculate and return the appropriate part price based on the equation

```
(UnitPrice * ExchangeRatePct / 100)
```

Step 3

Within the main function, declare a DomesticPart object with a PartID of 2 and a unit price of 10. Declare an ImportedPart with a PartID of 3, a unit price of 10, and an exchange rate of 120%.

Step 4

Build, execute, and test your application before continuing to Exercise 2. Exercise 2 is optional. Close all source and header files before continuing.

Exercise 2 (Optional: Complete in open lab time) Extending Another Class

Scenario

Your MIS Manager has offered the use of contract programmers for the short-term need of completing the prototype Inventory System. You realize that the current payroll package includes just salaried employees denoted as permanent. The contractors don't match the job descriptions typically classified as "temporary," due to payroll tax and insurance benefits.

You have time to extend the temporary employee classification to meet the reporting needs for contract programmers. The major variation is hourly pay versus a salary. Contractors, paid monthly, also receive double-time for hours over 160 per month.

A skeleton application, EMPLOYEE.CPP, exists in the \STUDENT\LAB10 subdirectory. The base class, `Employee`, embeds the Date class from previous modules. There is also an existing derived class, `Permanent`.

Step 1

Open and examine the file EMPLOYEE.CPP. The `Date` class occurs first; it is embedded in `Employee`. The `Employee` class maintains the date of hire for each employee. The `Permanent` class inherits from `Employee`, and includes one data member for monthly salary.

You will create a new class, `Contractor`, that has two data members: `m_nHourlyRate` and `m_nHours`. Examine the constructors and destructor for the `Permanent` class. Your new class should include accessor functions for each data member: `GetRate`, `GetHours`, and `SetHours`.

Note Hourly rate is "set" at time of hire (also known as contractor construction.)

Additionally, the member function to generate the contractors' monthly pay, `Paycheck`, must calculate at double-time rates for hours greater than 160.

Step 2

Within the `main` function, declare a contractor object, `cont1`, with a start date of 1/4/1994 and a $12 hourly rate.

Step 3

The contractor worked 180 hours. Set that amount.

Step 4

Examine the lines in `main` where the `Permanent` employee is "paid." In a similar fashion, "pay" the contractor.

Step 5

Build, execute, and test your application. The `CTime` class requires the AFX library in the build process. From the Options menu, choose Project. From the Project Options dialog box, choose the Linker button. In the Libraries text box, add the library **mafxcr** for a release mode project.

5

Summary

This objective	Was met by...
Use public inheritance	Exercises 1, Step 2; Exercise 2, Step 1
Extend a base class	Exercises 1 and 2

Lab 11: Managing Projects

Objectives

At the end of this lab, you will be able to:

- Use various methods to divide header files from source code.
- Use and create project .MAK files to manage multiple files.

Scenario

You will revisit the Inventory application from earlier modules. You will investigate the process of splitting a large source file into logical class components (header files) and test programs (source-code files).

Estimated time to complete this lab: 30 minutes

Exercise 1
Source vs. Header Files

A complete source file, INVENTRY.CPP, is in the \STUDENT\LAB11 subdirectory. It's the solution from a previous lab. It has four classes: `Date`, `Money`, `PartID`, and `Inventory`, plus a `main` function to declare one inventory item. This file *does not* have any TO DO steps listed in the source file.

Note You should close all source and header files (and other windows open in the Visual Workbench) before continuing.

The instructions in Steps 1 through 3 present three distinct ways to copy data from one window to another. Windows experience is not a prerequisite for this course, so these steps spell out some techniques that may already be familiar to you. If you have a preferred way of editing and working with text, feel free to go about it in your own way. If you are unfamiliar with the Windows environment, try each of these methods. Then use the one you prefer in the remaining steps.

As with previous labs, you will go to the File menu and choose Open.

Step 1

This step uses the keyboard to select and manipulate code.

1. In the INVENTRY.CPP source file, locate the class `Money`.
2. Select all of class `Money`, including the blank line after the class definition. To select the code you wish to copy, position the cursor at the blank line above `class Money`. Press and hold the SHIFT key. With the SHIFT key depressed, use the DOWN ARROW key to select line after line in the source file. (Selected text is highlighted on the screen.) Release the SHIFT key.

   The selected text remains highlighted.

3. Copy the highlighted text to the Clipboard. ALT+E displays the Edit menu. The Copy command is chosen with ALT+C.

   The Clipboard temporarily holds data so that it can be pasted (inserted) anywhere in any Windows-based file. When you use the Cut or Copy command to place data in the Clipboard, the Clipboard clears any previous contents and then holds the new data for pasting. (Simply deleting text does not place it in the Clipboard.)

4. Open a new window. (That is where you will paste the text from the Clipboard.) ALT+F displays the File menu. ALT+N chooses the New command, which opens a new window.

   A window labeled <2> UNTITLED.1 appears. The cursor is blinking in the upper-left corner of window 2, which shows that it is the active window.

5. Paste the contents of the Clipboard into the new window. Again, ALT+E displays the Edit menu; ALT+P chooses the Paste command.

   The text should appear in the new window. If the text for the `Money` class does not appear, repeat Step 1 from the beginning. (The following step tells you how to return to the INVENTRY.CPP source window.)

6. To return to the INVENTRY.CPP source window, use ALT+1 (ALT and numeric one—the window number).
7. To delete the `Money` class code from INVENTRY.CPP, verify that it is still selected. Press the DEL key (labeled Delete on some keyboards) to remove the selected code from the file.

Step 2

This step uses the mouse to cut and paste the code for class `Date`.

1. In the INVENTRY.CPP source file, locate the class `Date`.

   Only class `Date` uses the `CTime` functions. Time data and functions are fully encapsulated within `Date`; they are not referenced anywhere else within INVENTRY.CPP.

2. Select the portion you wish to cut and paste: the entire `Date` class. Use the mouse to position the cursor at the start of the `#ifdef _WINDOWS` statement above `class Date`. Click and hold down the left mouse button. Drag the mouse pointer lower and lower in the window. Lines of code are selected as you scroll by. Continue to drag and select all of class `Date`, including the blank line below the `GetTodaysDate` member function.

   Release the mouse button. The area will remain highlighted.

   Scrolling Tip You can control scrolling speed with the mouse. Did you notice that as you approached the bottom of the source window, the window scrolled more quickly? If you want scrolling to slow down or reverse itself, move the mouse to a higher position in the window. The speed with which you move the mouse affects scrolling speed, too.

3. Click the Edit menu and choose Cut.

   The text is cut from this file and held in the Clipboard for pasting.

4. To open a new file, click the File menu. Choose New.

   A window labeled <3> UNTITLED.2 appears. The cursor is blinking in the upper-left corner of window 3. That shows that the new window is the active window.

5. To paste the contents of the Clipboard into the new window, click the Edit menu. Choose Paste.

   If the text does not appear, ask the instructor for assistance.

6. If the text appeared as expected, use ALT+1 to return to the INVENTRY.CPP source window.

   Notice that the `Date` class was deleted from this file by the cut operation.

Step 3

This step performs the cut and paste operations in a combination of mouse and keyboard shortcuts.

Note You can learn any Windows-based shortcuts by looking at the menus. To display a particular menu, press ALT plus the underlined letter in the desired menu. For example, since the F in the File menu is underlined, you know that ALT+F will display the File menu. When you display a menu, you will see that some of the commands have shortcut key combinations to the right of them. Those are the *accelerator* key combinations that will be used in Step 3. Accelerator keys carry out operations without displaying a menu or its commands.

1. In the INVENTRY.CPP source file, locate the class `PartID`.
2. Use the mouse to select the entire `PartID` class. Position the mouse pointer on the blank line just above `class PartID`. Click and hold the left mouse button. As you did in Step 2, drag the mouse pointer down the screen, selecting code as you go. Select all of the `PartID` class, including the blank line after the class definition.

   Release the mouse button. The selection remains highlighted.
3. Use the CTL+X key sequence to cut the selected text and place it in the Clipboard.
4. Use the CTRL+N key sequence to open a new file.

   A window labeled: <4> UNTITLED.3 appears. It is the active window; the pasting operation you're about to do will place the text in the active window.
5. Use the CTRL+V key sequence to paste the text. If the text does not appear, ask the instructor for assistance.
6. To return to the INVENTRY.CPP course window, use the ALT+1 key sequence.

   The `PartID` class was already deleted from this file by the cut operation.

Step 4

Use any of the procedures in Steps 1, 2, or 3 to carry out this step.

1. Locate the class `Inventory`.
2. Select the class `Inventory`.
3. Copy or cut the selection to place it in the Clipboard.
4. Start a new file. It will be <5> UNTITLED.4 if you have performed all of the steps.
5. Paste the contents of the Clipboard to insert the `Inventory` class in the new window.
6. Use ALT+1 to return to the INVENTRY.CPP window. (If you used the copy command to put the text in the Clipboard, you must still delete the selected text from the INVENTRY.CPP file. Use the DEL key to delete it.)

Step 5

1. Use ALT+5 to return to the <5> UNTITLED.4 window.
2. At the top of this file, add a comment describing this header file as INVENTRY.H.

Step 6

Does the `main` function know about `PartId`? or `Money`? or `Date`? The answers are "no," "no," and "a little." The `main` function performs one piece of housekeeping to initialize the static variables used by `Date` (and we'll get rid of that soon.) With most answers as "no," should `main` include these .H files? No.

1. Add statements in INVENTRY.H to include the following:

   MONEY.H

   DATE.H

   PARTID.H

   These files will be in the current directory. Does that change your **include** statements?
2. Save the file by going to the File menu and choosing Save As.

   The Save As dialog box appears.
3. Press the DEL key once to clear the filename extensions. In the File Name text box, enter the name **inventry.h**. (Note that there is no "o" in the filename.)
4. Press ENTER (or choose the OK button).

Step 7

1. Use ALT+2 to change to the `Money` class window.
2. Add a comment at the top of the file describing it as MONEY.H.
3. Use ALT+F and then ALT+A to invoke the Save As command.

   The Save As dialog box appears.
4. Press the DEL key once to clear the filename extensions. In the File Name text box, enter the name **money.h** and press ENTER (or choose the OK button).

Step 8

1. Use ALT+3 to change to the `Date` class definition.
2. Add a comment at the top of the file describing it as DATE.H.
3. Add a second comment line that notes this file's use of AFX.H.
4. Use the CTRL+S key sequence to invoke the Save As dialog box.

Note CTRL+S is usually just Save, but this file has not been named or saved yet. Visual Workbench presents a Save As dialog box in anticipation of your naming the file.

5. Press the DEL key once to clear the filename extensions. In the File Name text box, enter the name **date.h** and press ENTER (or choose the OK button).

Step 9

1. Use ALT+4 to change to the `PartID` class definition.
2. Add a comment at the top of the file describing it as PARTID.H.
3. Save the file as PARTID.H.

8

Step 10

You can save all of the open files at once. From the File menu, choose Save All.

Step 11

1. Use ALT+1 to return to the INVENTRY.CPP file.

Does the `main` function in INVENTRY.CPP know about our class `Inventory`? No. Does it need to know? The answer is easily "yes." It constructs an object and invokes the `Display` member function.

2. Add an **include** statement for INVENTRY.H.

Step 12

Build, execute, and test your application before continuing to Exercise 2. You should also close all source and header files (and other windows open in the Visual Workbench) before continuing.

Exercise 2
Scope in Single Source Files

Scenario

Your return visit to the Inventory application was a good example of project management for source and header files. Building an example with enough code to demand multiple sources would take a long time—and it would take a long time just to present the problem. The two following exercises use small code files, but they present an answer to the overall question of how to protect or share both code and data across multiple source files.

A complete source file, SCOPE1.CPP, is located in the \STUDENT\LAB11 subdirectory. This program displays text concerning the visibility issues within a single source-file application.

Step 1

1. Open the file, rebuild it, and execute the application.
2. Expand the output window for the program. Use either Maximize or Size options for a window. Read the output as a refresher for scoping rules within a single source file.
3. Close this source file (and any other windows that are open in the Visual Workbench) before continuing.

Exercise 3
Scope in Multiple Source Files

Scenario

As was mentioned earlier, this second scope exercise uses small code files as you learn to protect or share code and data across multiple source files.

Two complete source files are located in the \STUDENT\LAB11 subdirectory. Prior to opening the source files, we'll create a project file to control the build process.

Step 1

1. From the Project menu, choose New.

   The New Project dialog box appears.

2. In the Project Name text box, type **scope2.mak**.
3. Press the TAB key twice to advance to the Project Type box. Use the DOWN ARROW key to display the options.
4. Select QuickWin Application (.EXE).

Note Be sure the Use Microsoft Foundation Classes option is cleared—that is, not checked.

5. Choose the OK button.

   The Edit dialog box appears, listing several source candidates in the File Name box.

   You'll be adding two files to this project. There are two ways to do it.

6. Double-click the file named SCOPE2A.CPP.
7. Select the SCOPE2B.CPP file by clicking on it once. Then choose the Add button.
8. Choose the Close button to complete the project. Notice that the title bar for Microsoft Visual C++ now includes the project name, SCOPE2.MAK. No project components are automatically opened.

Step 2

1. From the Project menu, choose Build SCOPE2.EXE.
2. Execute the program. Expand the output.
3. Read the output to confirm concepts for scoping rules within multiple source files.
4. Close any source files and close the project.

9

Summary

This objective	Was met by...
Use the appropriate method for making header files from source code	All three exercises
Use and create project .MAK Files to manage multiple files	Exercise 3, Step 1

Lab 12: Manipulating Arrays

Objectives

At the end of this lab, you will be able to:

- Manage character manipulations using arrays and subscript notation.
- Convert numeric data types to character strings.
- Write a string-handling function.

Scenario

You're very pleased with the status of a number of the sample applications you've created. You should be! Still, it would be nice—and much easier on your eyes—to have nicely formatted output from your applications. A leading currency sign with a string of digits is difficult to decipher. Separators would be a nice addition.

Estimated time to complete this lab: 45 minutes

Exercise 1
Adding Characters to a String

A project file, MONEY.MAK, exists in the \STUDENT\LAB12 subdirectory. It uses a version of the `Money` class that is similar to previous modules. This project uses the files MONEY.CPP and MONEY.H. This version won't compile because `main` is coded to call a missing member function, `DisplayNumeric`.

Get started by going to the Project menu and choosing Open. Select MONEY.MAK. Click the far left button on the toolbar, the Project File button. It displays the list of files that are used in this project. From the list, select a file to open.

Step 1

1. Open the source file MONEY.CPP.
2. Locate the call to invoke the `DisplayNumeric` function within `main`. There is no **return** type, and there are no arguments. `DisplayNumeric` is self-contained.
3. Open the header file MONEY.H.
4. Locate the class `Money`. The class constructors have changed. Both constructors still assign values to the data members. But there is a new statement in each that assigns a NULL character to the data member `szFormatted`.
5. Declare `szFormatted` as a new private data member with room for 20 characters.

Step 2

1 Locate the `DisplayNumeric` member function. It contains simple conditional logic to determine whether `szFormatted` contains information. If it contains no information, the function `BuildNumeric` is called to load the data.

2. Add a prototype for the `BuildNumeric` function.

Overview of Steps 3–9

The steps that follow are a recommendation. There are various ways to achieve the desired output. You may follow these steps, or create your own solution. You are strongly urged to design your solution using a notepad and pencil before starting with the code!

The loop in Step 6 is the most challenging algorithm in this lab. Characters are transferred from `szTemp` and are merged with currency separator characters to load the `szFormatted` string into an array. The logic for the loop could be pseudo-coded as follows:

```
Loop from start of szTemp until the full length of the string is processed.
    Determine if current char in szTemp is an even multiple of 3 from
    the end of the string.
        If true, assign a separator char to the next location in szFormatted
    Assign the next char from szTemp to the next location in szFormatted
End of loop
```

Three integers and a small character array are given within BuildNumeric. iFormat is used to index the szFormatted data member as characters are assigned to that string. iTemp indexes into the **char** array, szTemp. iLen is set to the length of szTemp and used as a counter/index for a loop that transfers digits and commas into szFormatted.

No currency displays begin with a separator. As a statement prior to the loop, you may want to assign the first character from szTemp into the next location in szFormatted. Be sure to advance iTemp and iFormat as characters are assigned from one string to another.

For most currencies, the separators occur every 3 digits. You may want to use the modulus operator, %, to test for a third occurrence. Your loop should start at the beginning of the szTemp string and advance through all characters, incrementing iTemp and iFormat and decrementing iLen. Either the value iLen or the null-character in szTemp will be a stopping point.

Step 3

Begin within BuildNumeric. Assign the currency symbol that is appropriate for your currency to the szFormatted string. If the currency symbol occurs after the amount, place your assignment at the bottom of this function.

Step 4

The lDollars amount is a **long**. Convert the value of lDollars into a string using the szTemp character array provided, and base 10. Depending on the function you use, you may have to add an **#include** to this file.

One recommended solution is the ANSI ltoa function in the <stdlib.h> file.

Step 5

The location of the currency separator characters depends upon the length (iLen) of the character string in szFormatted.

Determine the length of szTemp and save the value in iLen.

Step 6

Loop through szTemp, adding characters and commas to the szFormatted string as needed.

For most currencies, the separators occur every 3 digits. If you want to test for a third occurence, you could use the modulus operator, %. Typically, every iteration of the loop should take a character from szTemp to szFormatted. Whenever the remaining characters in szTemp amount to an even multiple of three, also add the currency separator character.

Step 7

Assign the decimal separator into szFormatted.

Step 8

The cents display has been disappointing. When the cents amount is less than 10, the cent amount has appeared where the "tens" amount should appear.

1. Convert the value of `nCents` to the string `szTemp`. Refer to Step 4, if needed.
2. Insert a conditional statement to ensure that a leading zero appears when needed. Your application must clearly differentiate between .50 and .05.
3. Assign the appropriate characters from `szTemp` to `szFormatted`.

Step 9

The data member `szFormatted` holds all the visible characters. Add the final character that makes it a safe string variable.

Step 10

Build, execute, and test your application before continuing to Exercise 2. Close any open files, and close the MONEY.MAK project before continuing.

Exercise 2 (Optional)
Writing a Simple String-Handling Function

Scenario

The Purchasing group reordered forms and envelopes for their purchase orders. These new envelopes have an address window that is 15% smaller than standard. The address area in the reprinted forms is 20% smaller than in previous versions. They've requested new functionality that truncates a given string to accommodate strings to a given length.

You realize that this is not likely to be a one-time fix. You decide to build a small class and sample program that prompts the user for a string and a number. One function, `LeftString`, will return the leftmost "number" or characters from the string.

A skeleton application, LEFT.CPP, exists in the \STUDENT\LAB12 subdirectory. It contains a class, `MyString`, and a `main` to test the member functions.

Step 1

Locate the skeleton class, `MyString`.

Within the member function, `MyReadString`, write a statement that gets up to `iLen (- 1)` characters from the user.

Step 2

Within the `LeftString` member function, write the loop that copies characters from argument 1, `szSource`, to argument 2, `szDest`. Your loop should be careful not to copy beyond the end of the source string, and should not exceed the size of the destination string.

Step 3

Append a null character after the last character to return a clean string.

Step 4

Within `main`, previous lines have prompted the user for a string and then read those characters. Complete the conditional statement provided to determine whether any characters were entered.

Step 5

Build, execute, and test your application.

Summary

This objective	Was met by...
Manage character manipulations using arrays and subscript notation	Exercise 1, Steps 3, 5, 6, and 7
Convert numeric data types to character strings	Exercise 1, Step 4
Write portions of a string-handling function	Exercise 2, Steps 1, 2, and 3

Lab 13: Pointers and Arrays of Pointers

Objective

At the end of this lab, you will be able to use pointers to perform string-parsing.

Scenario

You're very pleased with changes to the money display routines. You realize that one more variation will satisfy most of the future needs. What's missing? (Hint: Try to print a check.) Class `Money` still lacks a formatted alpha or string output that is typically used to print checks.

Estimated time to complete this lab: 30 minutes

Exercise 1
Using Pointers

A project .MAK file exists in the \STUDENT\LAB13 subdirectory. After closing any open files or projects, open the TESTMONY.MAK project.

TESTMONY.MAK builds TESTMONY.EXE by compiling TESTMONY.CPP and MONEY.CPP using MONEY.H. This application is similar to the final lab from the previous module, with the addition of a display function to print monetary amounts using a string format.

This version won't run correctly because the `main` in TESTMONY.CPP is coded to call a `Money` member function, `DisplayAlpha`, in MONEY.CPP. That function has statements missing. One last detail—in the interests of fiscal responsibility— this version of `DisplayAlpha` will only display amounts less than $1 billion.

Step 1

Open the file MONEY.H. Examine the class `Money`. It has changed two ways:

- The conditional in `DisplayNumeric` has changed.
- A new member function, `DisplayAlpha`, is in class `Money` and contains a similar conditional.

Examine these conditional statements. The objective is to only build the numeric formatted string or alpha formatted string when needed. If either display type is presented, it tries to avoid building the same string again.

Modify those conditionals if that is required for your currency.

Step 2

The alpha formatted string requires more characters. Increase the dimension of `szFormatted` to 180 bytes.

Step 3

Class `Money` has three new member functions. `BuildAlpha` is equivalent to `BuildNumeric`, no arguments, no return value. `HundredsTensOnes` generates words for numeric values and takes one **long** data type as an argument. The third function is `StringCat`. It takes two character pointers as arguments.

Add prototypes for those three functions.

Step 4

Open the file MONEY.CPP. This file contains the growing collection of non-inlined member functions that support the `Money` class. There are numerous helper routines and data definitions added to MONEY.CPP.

Three arrays of strings have been declared and initialized:

```
char* szOnes[10] = { "Zero", "One", ...
char* szTeens[10] = { "Ten", "Eleven", ...
char* szTens[10] = { "?", "Ten", "Twenty", ...
```

They are global, so only one copy of those strings will be in our application, regardless of the number of objects.

Locate the definition for the `DisplayAlpha` function. It has full access to `Money` data members. Read through the function to become familiar with the processing that's given. Trace the logic into the `HundredsTensOnes` function.

You've likely encountered four blank lines within the comments: TODO #4. Good guess! In each of these areas, a digit position from the `lDollars` amount has been identified. That digit will index into an array of strings to output the correct string on the screen.

There are numerous examples in the previous lines and several good clues in the program comments that detail what needs to happen. Complete those four statements.

Step 5

At the bottom of the MONEY.CPP file is the skeleton of a function, `StringCat`. You prototyped it earlier. You'll write the function now.

Your solution should advance the pointer `pStr1` until a NULL character is located. With `pStr1` positioned on the `NULL`, loop through both pointers, concatenating the contents of `pStr2` onto `pStr1` until the `NULL` from `pStr2` is transferred.

Step 6

When you've completed the changes, use Build TESTMONY.EXE. Then use Run to test your application.

Summary

This objective	Was met by...
Use pointers to perform string-parsing	Step 5

Lab 14: Using Commercially Available Classes

Objective

At the end of this lab, you will be able to:

- Create objects using a commercially available class.
- Use operators to manipulate objects.
- Use member functions from a commercially available class.

Scenario

The money display routines work very well. The CString class is intriguing. The code appears clearer and would be easier to maintain. You decide to revisit the class `Money` to modify the alpha or string output used to print checks.

Estimated time to complete this lab: 30 minutes

Exercise 1
Parsing Strings with the CString Class

A project .MAK file exists in the \STUDENT\LAB14 subdirectory. After closing any open files or projects, open the project MONEY.MAK. MONEY.MAK builds MONEY.EXE by compiling TESTMONY.CPP and MONEY.CPP using MONEY.H. This version would run right now—it's identical to the solution from the previous lab.

This two-part exercise modifies the application to use a CString object rather than `szFormatted[180]`. Initially, the operators offered with CString are used. The buffer-access member functions with CString may be used in the later half of the exercise.

Step 1

Using project MONEY.MAK, open the file MONEY.H.

It will include a `CString` object named `strFmt`. Add the statements to include the MFC collection classes in a QuickWin application. These statements were introduced in the "static" module and supplied in Lab 8.

Step 2

Examine the class `Money`. It must be changed four ways:

- The conditional statement in `DisplayAlpha` must determine whether the `CString` object, `strFmt`, is empty. Use Help for a list of `CString` member functions.
- The `cout` statement in `DisplayAlpha` should be changed to output an object named `strFmt`.
- A new data member, `strFmt`, should be declared as a `CString` object.
- The `StringCat` member function will not be needed. Delete the prototype statement.

Step 3

Open the file MONEY.CPP. This file contains the growing collection of non-inlined member functions that support the `Money` class. There are numerous helper routines and data definitions added to MONEY.CPP.

Note Do not change `BuildNumeric` until Step 8.

Three arrays of strings are still there.

Locate the definition for the `BuildAlpha` function. It has full access to `Money` data members. Locate the line that assigns the NULL character to `szFormatted`. That line should assign an empty string to `strFmt`.

Step 4

Read through the rest of the function. It shows a dozen or more locations where the local `StringCat` function is invoked. All of those calls should change to operator += concatenation of the words onto the existing `strFmt` string.

Hint Use the Editor option to find the StringCat function. Notice that the Find window now lists the function name as the last search string. You can easily repeat the previous find by double-clicking the Find window; selecting a word in the Find window, and pressing ENTER; or pressing F3.

Step 5

At the bottom of the MONEY.CPP file, you'll find the function StringCat. Comment or delete those lines.

Step 6

When you've completed the changes, use Build TESTMONY.EXE to test your application.

Note The following steps are optional. They are presented to show you the power of working with a well-designed class. The BuildNumeric function works satisfactorily as it is currently coded.

As an exercise to investigate the buffer-access member functions in CString, the following steps will lead you through a rewrite of BuildNumeric. These steps may be completed if time permits.

Step 7

Within the file MONEY.H, examine the class Money. It must be changed four ways:

- The conditional statement in DisplayNumeric must determine whether a CString object, strNbr, is empty.
- The cout statement in DisplayNumeric should be changed to output an object named strNbr.
- The character array szFormatted will no longer be needed. A second CString object, strNbr, should be created and initialized to 20 spaces (' ').
- The Money class constructors need to change. Currently, each sets a NULL character into szFormatted element 0. In the declaration and construction of the CString objects, the appropriate action is performed. Remove the statements from the constructors that deal with szFormatted.

Step 8

Within the MONEY.CPP file, locate the definition for the BuildNumeric function. It has full access to Money data members. Locate the line that assigns the currency symbol to szFormatted.

The line should set a currency character at position 0 of strNbr object. Help describes the SetAt member function.

Step 9

Read through the rest of the BuildNumeric function. There are numerous places where characters were assigned to szFormatted. Those locations should be changed to set characters into the strNbr object.

Step 10

When you've completed the changes, use Build TESTMONY.EXE to test your application.

Summary

This objective	Was met by...
Create objects using a commercially available class	Steps 2 and 7
Use operators to manipulate objects	Steps 3 and 4
Use member functions in a commercially available class	Steps 2, 3, 4, 7, 8 and 9

Lab 15: Formatting and File I/O

Objective

At the end of this lab, you will bê able to:

- Add file I/O member functions to a class.
- Open, read, write, and close data files.

Scenario

Your development team has returned with newer versions of the building blocks for the inventory system. The new versions of the `Date` class and `Money` class have new member functions that load from and store to disk. These functions take a stream as an argument: `Load` takes an `ifstream` and `Store` takes an `ofstream`.

You'll revisit the Inventory application from earlier modules and investigate file input/output on an object with embedded objects. This version loads text Inventory data from disk, lists an inventory report, and stores binary Inventory data to another disk file.

Estimated time to complete this lab: 30 minutes

Exercise 1
C.asses That Load and Store Data

A project .MAK file exists in the \STUDENT\LAB15 subdirectory. After closing any open files or projects, open the project INVENTRY.MAK.

This project builds INVENTRY.EXE by compiling INVENTRY.CPP. It has four classes: `Date` and `Money` have the updated `Load` and `Store` functions, but `PartID` and `Inventory` still need that functionality.

Step 1

1. Locate the class `Money`. Notice that it has new `Load` and `Store` member functions. The `Money` class has all the code to save and restore its member data. (Each class should be self-contained.)
2. Locate the class `Date`. Examine its existing `Load` and `Store` functions.
3. Locate the class `PartID`.
4. Add `Load` and `Store` functions to the `PartID` class.

Step 2

Locate the class `Inventory`. The Inventory class "knows" about the embedded classes. Your solutions to `Load` and `Store` should handle the Inventory-specific data member, `m_nQuantity`, then invoke the `Load` and `Store` functions for each embedded object. Be sure to have your functions deal with each object in identical order!

The previous `Load` and `Store` functions simply tested the stream to determine whether it was "not bad." During input-stream processing, the stream may be valid, but it may be at the end-of-file marker. Therefore, the Inventory `Load` function should also check whether the input stream is "good" *after* attempting to read the `m_nQuantity` value. If the stream is not good, the `Load` function should return a zero value to indicate there was not another item to load.

Hint Refer to the module topic "Testing for Success" to see an example.

Add `Load` and `Store` functions to this class.

Step 3

Locate the `main` function. Declare an Inventory object named `iItem`.

Step 4

A text disk file named INVENTRY.DAT exists for input. Using the `ifstream` constructor, open `iFile` as the file stream for input.

Step 5

The `Store` functions will update a binary file, INVENTRY.BIN.

1. For a variation, create an `ofstream` object named `oFile`, using the default constructor.

2. As another statement, use the `ofstream` `open` member function to open the stream INVENTRY.BIN for binary mode.

Step 6

A skeleton `while` loop exists. You need to complete the `while` condition such that the Inventory `Load` function is invoked. The loop should continue unless `Load` returns a non-zero value.

Step 7

Build, execute, and test your application.

Summary

This objective	Was met by...
Add file I/O member functions to a class	Steps 1, 2 and 3
Open, read, write, and close data files	Steps 4, 5 and 6

Lab 16: Dynamic Memory

Objective

At the end of this lab, you will be able to use the **new** and **delete** operators.

Scenario

Remember that `Date` class? It's simple, it's current, but it's not able to display all the ways your users want to use dates. Yes, it does handle M/D/Y, D-M-Y, and may have another customized display you added. But the users report that occasionally a transposition error occurs. For example, an order needed by March 4, 1995 was scheduled for 4/3/1995.

The ability to display a date as a string (Weekday, Month, D#, Y###) would be a visual input-confirmation for the users. It would add one more variation to satisfy most future needs. Class `Date` could supply output typically printed on business correspondence (such as follow-up letters to find missing part orders).

Estimated time to complete this lab: 45 minutes

Exercise 1
Building Strings in the Heap

A project .MAK file exists in the \STUDENT\LAB16 subdirectory. After closing any open files or projects, open the project DATE.MAK.

DATE.MAK builds DATE.EXE by compiling TESTDATE.CPP and DATE.CPP using DATE.H. This application is similar to the final lab from the previous module, with the addition of a `Display` function to print dates using one of the formats depicted above.

This version won't run right now because `main` in TESTDATE.CPP is coded to call a `Date` member function, `DisplayAlpha`. That function is incomplete.

Step 1

Open the file DATE.H. Examine the class `Date`. It now has portions of a new member function, `DisplayAlpha`. The function should display the return from the function `BuildAlphaDate`. `BuildAlphaDate` creates a new area in memory, builds a string containing the day of week and the month name, and returns a pointer to that area. This `DisplayAlpha` function should receive the pointer, display the value, and free the memory created by `BuildAlphaDate`.

Within class `Date`, add a prototype for the function `BuildAlphaDate`. It should take no arguments and return a `char *`.

Step 2

1. Locate the function `DisplayAlpha`.
2. Declare a local character pointer, `cpDayMonth`.

Step 3

Invoke a call to `BuildAlphaDate` and receive the return value in `cpDayMonth`.

Step 4

Display the contents the dynamic area pointed at by `cpDayMonth`.

Step 5

The dynamic memory is no longer needed. Release it.

Step 6

Open the file DATE.CPP. It has the code for several member functions you created in earlier labs.

Examine the two character arrays: `Day` and `Month`. They hold the names of the days of the week and the month names. You may modify those strings to fit the reporting standards for your corporation.

Step 7

Locate the BuildAlphaDate member function. It returns a character pointer for the date, day of week, and month. The general format for the text output is "day-of-week, month DD, YYYY" where DD is the day-of-the-month digits and YYYY is the year.

Within BuildAlphaDate, declare and initialize a pointer variable, cpAlphaDate, to have 40 bytes of dynamic memory on the heap.

Step 8

Create a temporary pointer, cpTemp, initialized to the same memory area as cpAlphaDate.

Step 9

The dynamic area exists. You have an initialized, temporary pointer to work with. After Step 7, the existing lines have determined which day of the week should be loaded. It is element tmToday.tm_wday + 1.

(Optionally, you may declare a temporary variable, int iWDay, and use iWDay in this step.)

Write the statement(s) to copy the characters from the above element of the Day character array at the location in the heap area held by the temporary pointer, cpTemp.

Step 10

Build, execute, and test your application before continuing to Exercise 2.

Be sure to close all projects and files before you proceed.

If Time Permits...

Exercise 2
Fun Managing Memory

Scenario

To investigate dynamic memory allocations, you decide to create a guessing game to exercise **new** and **delete** operators.

For fun, no fees, this game allows the player 10 attempts to guess a random number. If successful, the player "wins" 10 points. If unsuccessful, the player is allowed to continue the game, and has up to 10 more guesses with a chance to win an ever-decrementing prize of 10, 9, 8, ... points for guesses 11 through 20. The game terminates after 20 attempts.

During play, the game saves each guess so that it can play back all guesses at the end of the game. Initially, the array has 10 locations. After ten guesses and a confirmation to continue, the array is resized to accommodate 20 guesses. (The first ten guesses must be copied into the "new" larger array.)

As each guess is accepted, the game will report whether the user's guess was too high or too low.

After 20 attempts have been exhausted, or the user correctly guesses the random number, a complete list of all guesses is displayed.

An incomplete source file, GUESSER.CPP, exists in the \STUDENT\LAB16 subdirectory.

Step 1

Class `Guesser` includes a **private** integer pointer, `ipGuess`.

Within the constructor, create a **new** array with room for 10 integer guesses. Your solution must also check for errors to ensure dynamic memory exists for the array.

Step 2

Within the `Guesser` destructor, called after the game is over, release the dynamic memory from Step 1.

Step 3

The original allocation in Step 1 allowed room for 10 guesses. The user has decided to play for up to 20 guesses.

Make the new allocation. Again, your solution must check for errors.

Step 4

The new allocation exists. Copy the first 10 guesses from the old array into the new array.

Step 5

The first 10 guesses (the old array) are no longer needed. Release that dynamic area to the free store.

Step 6

The user has attempted a guess, `m_nUserGuess`. Save that value to the end of the other guesses at `ipGuess`. Consider using `[m_nNumberOfTries]` and incrementing the number of tries.

Step 7

Build, execute, and test your application.

Summary

This objective	Was met by...
Use the new and delete operators	Exercise 1, Steps 2, 5, and 10; Exercise 2, Steps 1 and 2.

Lab 17: Creating Conversions

Objective

At the end of this lab, you will be able to:

- Create and use type casting.
- Create copy constructors and control conversions.

Scenario

The ability to create, set, get, and display `Date` objects in various formats has given the `Date` class a robust interface. That class does nearly everything you'd want to do! What's missing?

How about the ability to add or compare two dates? Fundamentally, the Inventory system needs to use the lead-time for an Inventory part when automatically reordering Inventory. Adding conversions will complete our `Date` class.

A Julian date is a measure of elapsed time from a base date. Many operating systems for personal computers use techniques such as the number of seconds elapsed since January 1, 1980 to represent date and time values. The Inventory will handle Julian dates as a number of days since 1/1/1972.

Estimated time to complete this lab: 45 minutes

Exercise 1
Building Strings in the Heap

A project .MAK file exists in the \STUDENT\LAB17 subdirectory. After closing any open files or projects, open the project DATE.MAK.

DATE.MAK builds DATE.EXE by compiling TESTDATE.CPP and DATE.CPP using DATE.H. This application is similar to the final lab from the previous module, with the addition of a conversion constructor and a casting operator. These two features allow the `Date` object to be created from a single number, and they allow dates to be converted to the long data type.

This version won't run right now because the `main` in TESTDATE.CPP is coded to create, subtract, and convert various dates.

Step 1

1. Open the file TESTDATE.CPP. Examine the new lines within `main`.
2. Open the file DATE.H. Locate and examine the class `Date`. It needs a prototype for a conversion constructor that takes a reference to a `long` data type as an argument.
3. Add the prototype for the new constructor.

Step 2

1. Within the class definition, locate the incomplete prototype for an `operator`.
2. Complete the prototype for an operator to convert a `const` date object to a `long` data type.

Step 3

1. Open the file DATE.CPP. Locate and examine two character arrays: `Day` and `Month`.
2. Modify those character strings as needed to meet corporate standards for date displays.

Step 4

Locate and examine the body of the new conversion constructor. It is coded to process a series of loops, decrementing the argument `lDays`, (a `long` data type) and assigning values to the date members of the `Date` class (actually to the new date object). Complete the formal definition of this conversion constructor.

Step 5

Locate and examine the body of the new cast operator. It calculates and returns a long data type representing the number of days since 1/1/1972. As coded, the function is accurate for more than 100 centuries. You may modify it as needed for your corporate standards. Complete the formal definition of this conversion operator.

Step 6

Build, execute, and test your application.

Summary

This objective	Was met by...
Create and Use type casting	Exercise 1, Steps 1, 2, and 5.
Create copy constructors and control conversions.	Exercise 1, Steps 2, 3, 4, and 5.

Appendix A: Hungarian Notation Table

Prefix	Meaning
Basic types	
f	Flag
ch	Character (no implicit size)
sz	Zero-terminated char *
fn	Function
v	Void
n	Number (no implicit size)
b	Byte
w	Word
l	Long
u	Unsigned
fp	Floating point (no implicit size)
Prefixes	
p	Pointer (don't use lp, hp, np)
r	Reference
rg	Array or &array
i	Index
c	Count
d	Difference
h	Handle
mp	Map array
u	Union
m_	Class member
ff	Bit flags
g	Global
Standard Qualifiers	
Min	First element in a set
Mic	Current first element in a set
First	First element in a set
Last	Last element in a set
Most	Last element in a set
Lim	Upper limit of elements in a set
Mac	Current upper limit of elements in a set
Max	Upper limit of elements in a set
Nil	Special illegal value
Sav	Temporary saved value
T	Temporary value
Src	Source
Dst	Destination

Procedures	
Delete, not Destroy or Free	Each word capitalized, including the first to distinguish from variables.
Macros and defines	
	Macros that accept parameters are named the same way as procedures. (use inline functions) Macros for constants are named the same way as variables. NULL, TRUE, and FALSE are the only exceptions.
Structure names	
struct ImageInfo	
Class names	
class UImage : public CObject	Same as structure names but prefixed with 'U' (to avoid name collisions with other class libraries)
Window types	
at	ACCELTABLE
bm	BITMAP
bfh	BITMAPFILEHEADER
bih	BITMAPINFOHEADER
br	BRUSH
co	COLORREF
cs	CREATESTRUCT
cur	CURSOR
dc	DC (Device Context)
dis	DRAWITEMSTRUCT
dwp	DWP (DeferWindowPos)
elf	ENUMLOGFONT
fix	FIXED
fnt	FONT
gm	GLYPHMETRICS
hk	HOOK
icn	ICON
inst	INSTANCE
lbr	LOGBRUSH
lf	LOGFONT
lpal	LOGPALETTE
lpen	LOGPEN
mis	MEASUREITEMSTRUCT
menu	MENU
mf	METAFILE
mfp	METAFILEPICT

mmi	MINMAXINFO
mod	MODULE
msg	MSG
ntm	NEWTEXTMETRIC
of	OFSTRUCT
otm	OUTLINETEXTMETRIC
ps	PAINTSTRUCT
pal	PALETTE
pe	PALETTEENTRY
pan	PANOSE
pen	PEN
ptw	POINT
fixpt	POINTFX
rcw	RECT
rgn	RGN (region)
rsrc	RSRC (resource)
sizw	SIZE
tm	TEXTMETRIC
wp	WINDOWPOS
wnd	WND (window)
wc	WNDCLASS
fh	HFILE
MFC types	
Window Classes	
wnd	CWnd
wndf	CFrameWnd
wndmf	CMDIFrameWnd
wndmc	CMDIChildWnd
dlg	CDialog
dlgm	CModalDialog
btn	CButton
cbc	CComboBox
edc	CEdit
lbc	CListBox
sbc	CScrollBar
stc	CStatic
GDI Classes	
dc	CDC
dcc	CClientDC
dcm	CMetaFileDC
dcp	CPaintDC
dcw	CWindowDC

bm	CBitmap
br	CBrush
fnt	CFont
pal	CPalette
pen	CPen
rgn	CRgn
Other Classes	
menu	CMenu
pt	CPoint
rc	CRect
siz	CSize
File classes	
fil	CFile
film	CMemFile
fils	CStdioFile
Object IO	
arch	CArchive
dmpc	CDumpContext
Exceptions	
ex	CException
exa	CArchiveException
exf	CFileException
exm	CMemoryException
exns	CNotSupportedException
exr	CResourceException
Collections	
arb	CByteArray
ardw	CDWordArray
aro	CObArray
arp	CPtrArray
ars	CStringArray
arw	CWordArray
lso	CObList
lsp	CPtrList
lss	CStringList
mppw	CMapPtrToWord
mppp	CMapPtrToPtr
mpso	CMapStringToOb
mpsp	CMapStringToPtr
mpss	CMapStringToString
mpwo	CMapWordToOb
mpwp	CMapWordToPtr

Miscellaneous support classes	
s	CString
time	CTime
dtime	CTimeSpan
Utopia types	
x	
y	

Appendix B: Operator Precedence Chart

Operator	Name or Meaning	Associativity
::	Scope Resolution	None
::	Global	None
[]	Array Subscript	Left to right
()	Function Call	Left to right
()	Conversion	None
.	Member selection - object	Left to right
->	Member selection - pointer	Left to right
++	Postfix increment	None
--	Postfix decrement	None
new	Allocate object	None
delete	Deallocate object	None
delete[]	Deallocate object	None
++	Prefix increment	None
--	Prefix decrement	None
*	Dereference	None
&	Address-of	None
+	Unary plus	None
-	Arithmetic negation	None
!	Logical NOT	None
~	Bitwise Complement	None
:>	Base Operator	None
sizeof	Size of object	None
sizeof()	Size of type	None
(type)	Type cast (conversion)	Right to left
.*	Apply pointer to class member	Left to right
->*	Dereference pointer to class member	Left to right
*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
-	Subtraction	Left to right
<<	Left shift	Left to right
>>	Right shift	Left to right
<	Less than	Left to right
>	Greater than	Left to right
<=	Less than or equal to	Left to right
>=	Greater than or equal to	Left to right
==	Equality	Left to right
!=	Inequality	Left to right
&	Bitwise AND	Left to right

^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
e1?e2:e3	Conditional	Left to right
=	Assignment	Right to left
*=	Multiplication assignment	Right to left
/=	Division assignment	Right to left
%=	Modulus assignment	Right to left
+=	Addition assignment	Right to left
-=	Subtraction assignment	Right to left
<<==	Left-shift assignment	Right to left
>>==	Right-shift assignment	Right to left
&=	Bitwise AND assignment	Right to left
|=	Bitwise inclusive OR assignment	Right to left
^=	Bitwise exclusive OR assignment	Right to left
,	Comma	Left to right

Appendix C: Memory Management

The topics covered in this appendix are either advanced topics, or further elucidation of topics introduced in the module on memory management:

I. How the Stack Works

II. Recursion

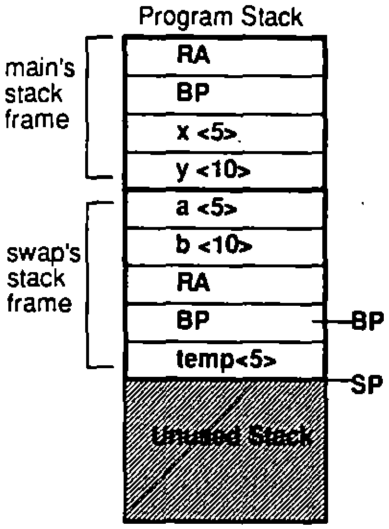
III. Memory Models and Segmentation

IV. Insufficient Memory Conditions

As you read through these sections, remember that many of the specifics are compiler- or operating-system dependent.

How the Stack Works

The stack represents the data work areas for functions. As the name implies, it grows and shrinks in units just as a stack of plates does. Each unit of growth or shrinkage is called a *stack frame*. The stack frame represents the work area for a single invocation of a function. Inside an executing program, when a function is invoked, a new stack frame for that function is allocated on the stack. When a function returns, its corresponding stack frame is discarded. Consider the following source program and a picture of the stack as it would appear at the indicated point of execution:



```
void swap(int, int);

int main()
{   int x=5, y=10;
    swap(x,y);
    cout << x << "  " << y;
 }

 void swap(int a, int b)
 {   int temp;
     temp = a;
     a= b;
     b = temp;
 }
```

Two functions are active at this point: `main` and `swap`. The `main` function invoked `swap`, and `swap` is currently executing. Each stack frame has four portions: a passed argument portion (`main` has no arguments, but `swap` does), an RA slot, a BP slot, and an automatic variable portion. RA stands for return address. It holds the address of the instruction to execute after the current function returns. BP stands for base pointer. It acts as an anchor point in the current stack frame and points back to previous stack frames. (If a function accidentally overwrites the BP or RA area—by writing past the end of a local array, for example—the results will normally be disastrous.) SP, the 80x86 register "variable," always points to the top of the stack (lowest used memory); the register variable, BP, points to the current stack's BP slot.

Because `swap` was coded as call-by-value, only the values of `x` and `y` are copied to the formal arguments `a` and `b`, respectively. The value-swapping of `a` and `b` do not, therefore, affect `x` and `y`. Had `swap` been coded directly using call-by-reference or simulated by passing pointers and using dereference, the `a` and `b` would contain the addresses of `x` and `y`, respectively. When the `swap` function returns, SP will be moved to point to the bottom of the `main` stack frame, effectively discarding the old stack frame for `swap`.

Remember that the stack physically sits above the static area of the data segment. By default, the 16-bit Microsoft compiler adds a small bit of code to a program that checks at run-time to determine whether a new stack frame will overrun the end of the allocated stack region. This stack-checking functionality can be disabled by the `/Gs` command line switch, or through Visual C++ menus. (From the Options menu, choose the Project command, then the Compiler button. Clear the Disable Stack Checking box.) Stack-checking is enabled in Visual C++ Development System for

Windows and Windows NT by the /Ge option. Under Windows NT, it is difficult to overflow the stack since its default size is 1 MB RAM, and the stack can even use virtual memory to grow as required.

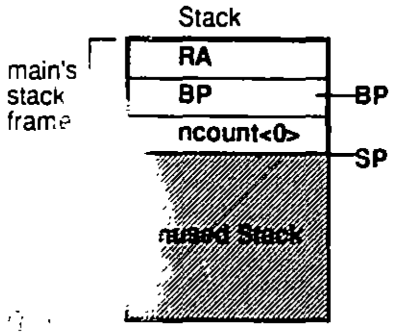
If there is a return value from the function, a Visual C++-based program will send the value back using one of the following mechanisms:

- If the return value is one or two bytes, it is returned in the AX register.
- If the return value is three or four bytes, it is returned in the AX/DX register pair.
- If the return value is greater than four bytes, it is returned in a special area, and a pointer to it is placed in AX (near) or AX/DX register pair.

Recursion

Because C++ is a stack-based language, it is able to support a special type of function invocation called recursion. A function invocation is recursive if it directly or indirectly calls itself. In a recursive situation, there will be multiple instances of a function's stack frame appearing on the stack at the same time. As an example, consider the sequence below.

The initial execution of `main`:



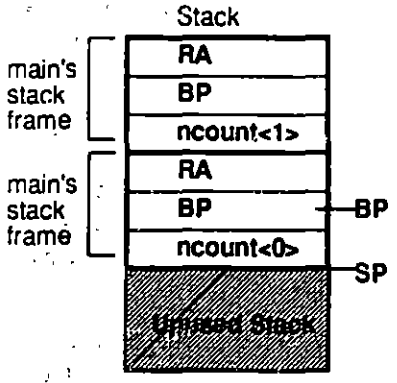
```
int main()
{   int ncount =0;
    cout << ncount << endl;
    ncount++;
    main();
}
```

OUTPUT

```
0
```

The next statement will increment `ncount` to 1. The fourth statement in `main` is unusual: it invokes the current function `main`, and is therefore directly recursive. Because of this call, a new stack frame for `main` is created, control jumps to the first executable statement in `main`, and we output the value of the local variable `ncount`:

This represents the second invocation of `main`:



```
int main()
{   int ncount =0;
    cout << ncount << endl;
    ncount++;
    main();
}
```

OUTPUT

```
0
0
```

This local variable `ncount` is a completely different variable that exists in a different stack frame. Again, the local variable will be incremented, and again `main` will be invoked, and so on. Here the direct recursion is infinite and will inevitably use up the program's stack. Recursion is normally controlled through a conditional call, perhaps using local **static** variables.

Recursion is a powerful programming tool that is essential in many advanced programming situations such as insertions and deletions on complicated, tree-like data structures. It is also useful in many other situations where the simple iterative solution is not obvious. The example above should be considered trivial.

Memory Models and Segmentation

IBM-compatible PCs use the Intel® 80x86-compatible series of CPUs. The original 8088/86 version of this chip had an architecture based on 16-bit words. Standard pointers were also 16 bits wide; in addition, a wider 20-bit version was supported. The shorter, so-called *near* pointers, support memory ranges up to 64K in size, whereas *far* pointers cover 1 MB.

These pointers' sizes had a direct effect on MS-DOS programs, forcing writers to select a specific *memory model* using a specific segmentation scheme:

memory model	max code	max data
tiny	64K combined size--------	
small	64K	64K
medium	64K	1MB
compact	1MB	64K
large	1MB	1MB
huge	1MB	1MB

The tiny model is a primitive one that was modeled after CPM operating system programs. As a result, it generates programs with a .COM file extension. Although the larger memory models allow maximum code and/or data-size to be up to 1 MB, most limit each unit (function or variable) to a size of 64K or less. Only the huge model supports variables (usually arrays) up to 1 MB. However, the huge memory model is rarely used because of its inherent slowness.

The sizes in the table are theoretical limits for 16-bit operating systems. MS-DOS further limits memory use to a combined total of 640K.

To complicate matters, most operating systems recognize two heaps: a near or local heap owned by your program, and a far or global heap owned by the operating system (and that can be shared among programs). Fortunately, the **new** and **delete** operators are implemented in such a way that the average programmer does not have to be concerned about which heap the resources come from.

Sixteen-bit Windows also supports the small-through-huge memory models. Though each variable and function must be smaller than 64K, the total program size is increased to a total of 16 MB in the medium-through-large memory models. Again, huge supports arrays larger than 64K.

Newer versions of the Intel chips, such as the Intel386™, Intel486™, and the Pentium™, do have a 32-bit mode. Normally, only 32-bit operating systems such as Windows NT can be run in this mode, however. Programs written for these newer operating systems have pointers that cover a 4 GB range, so most programs treat memory as a flat field with no segmentation—and thus no memory models. In theory, a Windows NT program can grow to be 64 terabytes by using virtual memory.

Insufficient Memory Conditions

When a 16-bit program is loaded into RAM memory, the subareas of the data segment are allocated using the following scheme:

- The SDA is of fixed size. That size, which is determined by the linker, is calculated by adding up the size required by all the **static** and **extern** variables, and all string literals.
- The stack has a default size of 2K. It can be adjusted at link time by using the command-line option /ST:*nnn* or by using the menus. (From the Options menu, choose Project. Then choose the Linker button and select the Memory Image option under category.) In addition, the Exehdr utility can be used to adjust the stack size of an existing program.
- The remaining memory is the size of the local heap.

The global heap is the memory remaining after the operating system allocates memory for all the running processes and reserve memory areas.

The Visual C++ Development System for Windows and Windows NT uses a related (but more powerful) scheme for suballocation:

- Again, the SDA is of fixed size.
- The stack has a default size of 1 MB. It, too, can be statically adjusted or can be set to grow dynamically by using virtual memory. (From the Options menu, choose Project. Choose the Linker button. Under Image Attributes, select the Stack Allocations option.)
- The heap is unconstrained to grow until maximum program-size is attained.

In a larger project on a 16-bit operating system, it is common to run into low-memory conditions. Although there are many complicating factors, the following troubleshooting chart may be helpful:

Area	Low Mem Indication	Possible Solutions
Heap	new returns NULL	1) Dynamically free unneeded memory. 2) Use larger memory model. 3) Use both local and global heaps.
Stack	1) run-time error: stack overflow. 2) GP fault or crash	1) Set larger stack size. 2) Change local arrays to **static** or **extern**. 3) Limit recursive function calls.
SDA	compile time out of memory condition	1) Use a larger memory model. 2) Dynamically allocate memory instead. 3) Store information in files instead.
Code	compile time out of memory condition	Use a larger memory model (compact or large).

In memory-constraint conditions, memory optimization often involves tradeoffs between the different subareas. When maximum limits need to be exceeded, programs often must resort to unusual and nonstandard measures, such as:

- Expanded (EMS), Extended (XMS), and Virtual Memory Libraries: These replacement libraries allow you to dynamically allocate data from memory above the 1-MB MS-DOS limit. The MS _fmalloc package represents this category.
- Overlaid Programs: These build (usually code only) automatic swapping-to-disk into the program.
- P-code: This reduces file-size by replacing native machine instructions with smaller "virtual machine" instructions that are quickly interpreted at run-time.
- DOS Extenders: These allow 24- and 32-bit programs to run under MS-DOS by acting as an intermediary between MS-DOS and the program. DPMI is an MS-DOS extender that is built into 16-bit Windows.
- Win32s™ API: This allows 32-bit programming under the 16-bit Windows programming environment. The *s* indicates that this interface API is a subset of the full Win32® API found in Windows NT.

Finally, one of the easiest solutions to memory woes is to port the program to a bigger operating system such as 32-bit Windows NT. Win32 programs have an inherent 4GB-RAM maximum, and through the use of virtual memory, this maximum increases to 64 terabytes. Most conveniently, memory constraints and complications usually don't need to be considered.

Appendix D: Reading List

∑ Reading List

C++ Language Resources

The C Programming Language, Second (ANSI) Edition, by Brian W. Kernigham and Dennis M. Ritchie. Prentice-Hall, 1988.

> Reference on the language by the original authors. Very succinct, pithy, style not meant as a tutorial. Superseded as the language definition by the ANSI X3J11 C Language Committee specification.

The Annotated Reference Manual, by Bjarne Stroustrup and Margarett Ellis. Addison-Wesley, 1990. Hardcover.

> Nicknamed the "ARM", this is the de facto specification on the language until the ANSI X3J16 committee issues its spec. Very technical and detailed manual on the C++ language, but does not cover iostreams, the only actual C++ library.

The C++ Programmming Language, second edition, by Stroustrup. Addison-Wesley, 1991.

> The main portion is an advanced manual/tutorial that is much more readable than the ARM. The last portion is a condensed reference on the language. More practical advice and coverage of related topics, such as iostreams. Mixes in explanations of why things are done as they are in C++.

C++ Primer, second edition, by Stanley Lippman. Addison-Wesley, 1991.

> One of the first and still one of the best tutorial/reference manuals on the C++ language. Easier paced than Stroustrup and Ellis.

Learning C++, by Tom Swan. SAMS Publishing, 1991.

> A beginner's tutorial on C/C++, it comes with an older MS-DOS small memory model C++ compiler and a shareware editor. Good, inexpensive introduction to C++ for the student or hobbyist.

A C++ Toolkit, by Jonathan Shapiro. Prentice Hall, 1991.

> A nice, small, practical, hands-on book of object-oriented analysis and design using C++, with a bunch of code examples.

C++ Strategies and Tactics, by Robert Murray. Addison-Wesley, 1993.

> Intermediate to advanced, but highly readable and concise, guide to the C++ language and practical OOAD. Answers many *why* and *how* questions on features of the language. Many small examples and practical threads to improve your C++ implementations.

Effective C++: 50 Specific Ways to Improve Your Programs and Designs, by Scott Meyers. Addison-Wesley, 1992.

> Linked discussion of advanced design and implementation topics in C++. The book answers many of the natural questions that arise when a new C++ programmer starts writing non-trivial code.

C++ Programming Guidelines, by Plum and Saks. Plum Hall, 1991.

> Coding conventions, style, and portability advice for the programmer and team manager alike. Considered by many to be more complete and less rigid than *C Programming Guidelines* by Plum.

Advanced C++ Programming Styles and Idioms, by James Coplien. Addison-Wesley, 1992.

How to design and code "higher-order" abstractions in C++. For the experienced C++ programmer who appreciates OO aesthetics.

An Introduction to Object-Oriented Programming in C++, by Budd. Addison-Wesley, 1991.

An introduction to the OOP paradigm, covering a number of languages, including C++.

Object-Oriented Design with Applications, by Grady Booch. Benjamin & Cummings, 1991.

One of the most highly regarded book on OOAD with examples in ADA, Object Pascal, Small Talk®, and C++.

Designing Object-Oriented Software, by Wirfs-Brock, Wilkerson & Wiener. Prentice Hall, 1990.

Another highly regarded book on OOAD. Creator of CRC cards.

The Design of Everyday Things, by Donald Norman. Doubleday Currency.

Well-written book on how to and how not to design real-world objects and systems.

Periodicals

C++ Report, published by SIGS, bimonthly, $4.95.

Most authoritative, up-to-date magazine on technical issues surrounding C++.

Journal of Object-Oriented Programming (JOOP, published by SIGS, bimonthly, $9.

High-level, academic review of current issues and research into OOPLs and technology.

Object Magazine, published by SIGS bimonthly, [illegible]0.

Readable news magazine, mixing industry news with technological articles.

Other

CompuServe® forums comp.lang.c++ and comp.std.c++

Usenix C++ Workshops and Conferences

OOPSLA Conference Proceedings