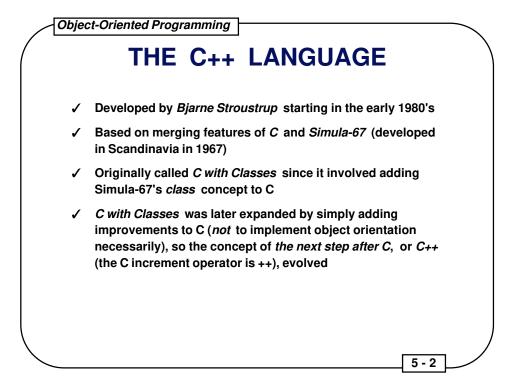
		-	IG
C++	Classes	const	Inheritance
Using C++	Automatic typedef	inline	Derived Classes
Types	Function Overload	Reference types	Composition
Scope Resolution	Type-safe link	new, delete	Binding
Protection	Constructors	Containers	Virtual Functions
friend Functions	Destructors	Header Files	Operator Overloading
	·	·	5-1

Objectives of Module 5

- Present and discuss the syntax of the C++ Programming Language
- Present and discuss the features of C++ which support object-oriented programming

Suggested Reading

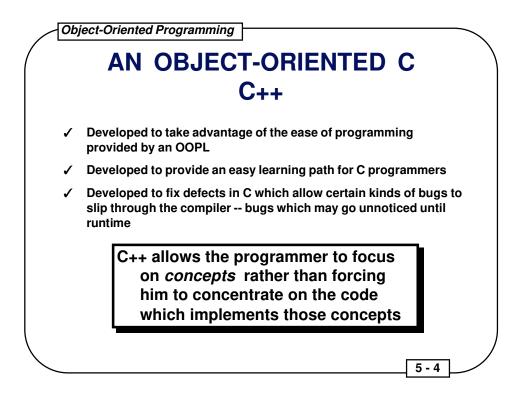
- Margaret A. Ellis and Bjarne Stroustrup, **The Annotated C++ Reference Manual**, Addison-Wesley Publishing Company, 1990, ISBN 0-201-51459-1
- Stanley B. Lippman, **C++ Primer**, AT&T Bell Labs, Addison-Wesley Publishing Company, 1989, ISBN 0-201-16487-6
- Lee and Mark Atkinson, **Using Borland C++**, Que Corporation, 1991, ISBN 0-88022-675-7

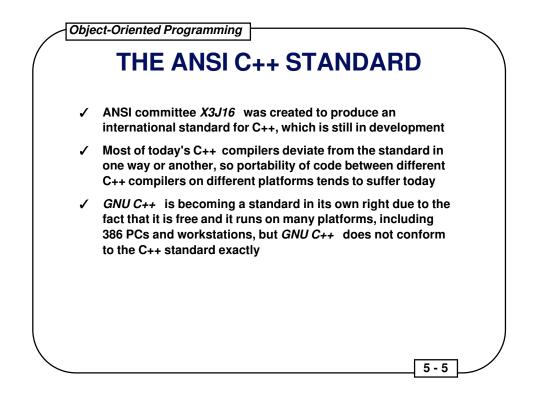


- The C++ is suffering from a lack of standardization today, with many distinct dialects emerging.
- This set of course notes reflects the *de facto* standard associated with the **Borland C++ language, version 3.0**. Borland C++ is quite close to AT&T C++ (cfront).

OBJECT ORIENTATION

- ✓ Simula-67 supports the creation of simulations, and simulations of systems usually involve many discrete, independently operating entities
- ✓ The authors of *Simula-67* called these entities *objects*
- ✓ Rather than perform actions on objects in a simulation, Simula-67 evolved the concept of sending messages to objects, and that's what object-oriented programming (OOP) entails
- ✓ OOP later proved to be an easy way to think about many other types of problems, so a number of other *object-oriented programming languages (OOPLs)* were developed, most notably *Smalltalk*
- ✓ These OOPLs provided many benefits, but the steep learning curve and significant period of limited productivity were drawbacks

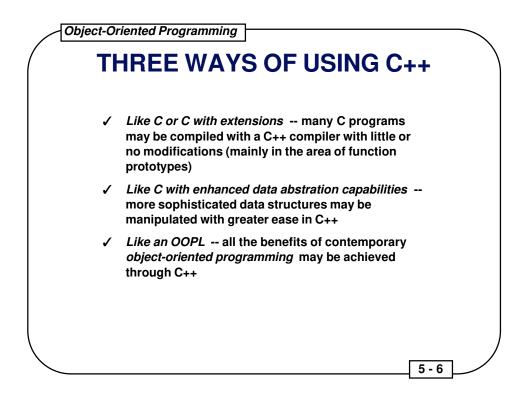




Note that the ANSI C++ standard does not yet exist.

In the interim, Stroustrup's book is the ANSI Base Document:

Margaret A. Ellis and Bjarne Stroustrup, **The Annotated C++ Reference Manual**, Addison-Wesley Publishing Company, 1990, ISBN 0-201-51459-1



TYPES = STRUCTS + FUNCTIONS				
A type is	a C struct with functions			
struct complex {	/* The C struct */			
<pre>float real_part;</pre>				
<pre>float imag_part;</pre>				
};	J			
<pre>struct complex {</pre>	// The C++ struct			
<pre>float real_part;</pre>				
<pre>float imag_part;</pre>				
<pre>complex();</pre>	<pre>// a constructor (discussed later)</pre>			
void add (complex,	complex); // operates on object			
_};				
	5-7			

This brings up the concepts of member data and member functions. Member data is duplicated with each object created under C++. Member functions are often implemented as inline functions, so their code is often duplicated when called.

typedef FOR C++ STRUCTS IS AUTOMATIC

struct complex a, b; /* C form is supported */

5 - 8

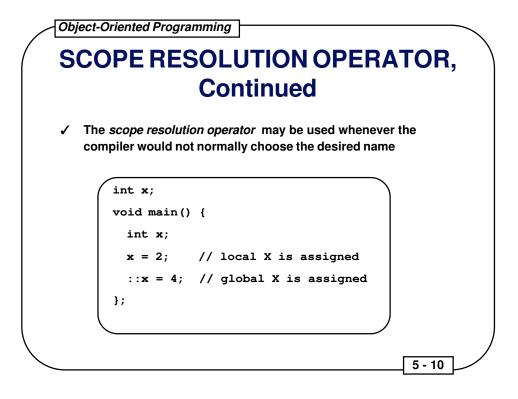
complex x, y; // "struct" is not required

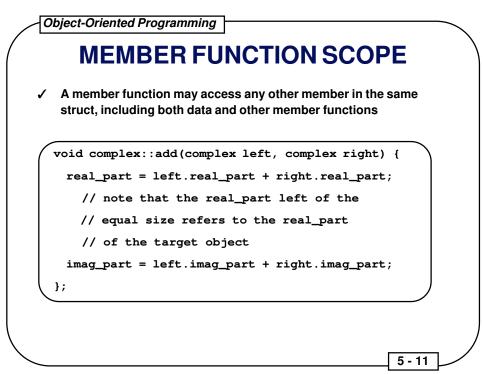
x.real_part = 2.2;

x.imag_part = 3.3; y.real_part = 4.2;

- y.imag_part = 4.3;
- y.add(x, y); // y = x + y

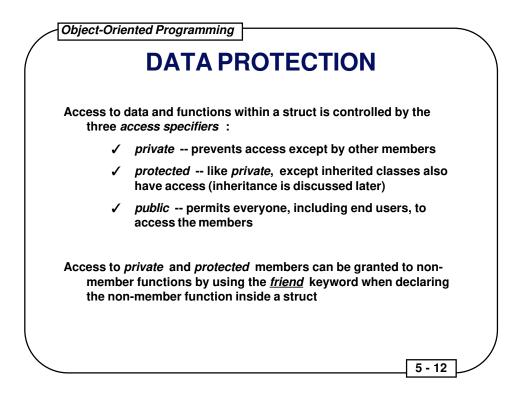
/	Object-Oriented Programming
(SCOPE RESOLUTION OPERATOR
	::
1	<i>Member functions</i> associated with a struct are <i>declared</i> as <i>function prototypes</i> in the struct
1	When <i>member functions</i> are <i>defined</i> , their associated struct is specified using the <i>scope resolution operator</i> (::)
vo	<pre>bid struct_name::member_function_name() { /* body */ };</pre>
as	in
VC	oid complex::add (complex left, complex right)
{ /	/* body */ };
	5-9





An example of a C++ class declaration and definition:

```
#include <stdio.h>
struct elevator {
  int floor_selected;
  int floor_number;
  void initialize(void);
  void select_floor (int floor);
  void go(void);
};
void elevator::initialize(void) {
  floor_selected = 0; floor_number = 0;
}
void elevator::select_floor (int floor) {
  floor_selected = floor;
}
void elevator::go (void) {
  if (floor_number < floor_selected)</pre>
    while (++floor_number < floor_selected)</pre>
      printf("floor: %d going up\n", floor_number);
  else
    while (--floor_number > floor_selected)
      printf("floor: %d going down\n", floor_number);
  printf("floor %d: stopping\n", floor_number);
}
```



```
Object-Oriented Programming

public AND private WITH friend

#define SIZE 10

struct int_array {

    private:

        int a[SIZE];

    public:

        void init(); // a member function

        friend void print (int_array); // a friend function

    };

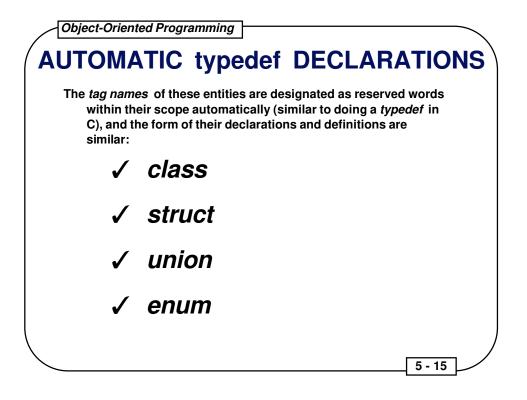
void print (int_array x) { // not a member function

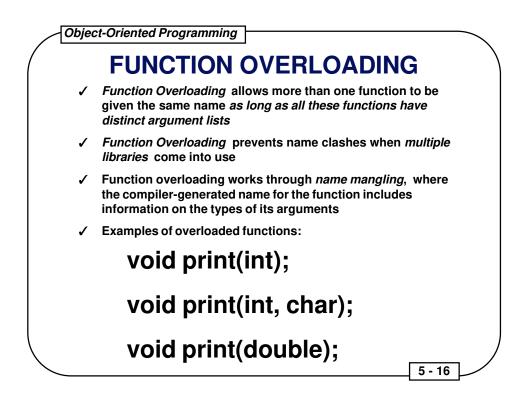
    for (int i=0; i<SIZE; i++) cout << x.a[i] << " ";

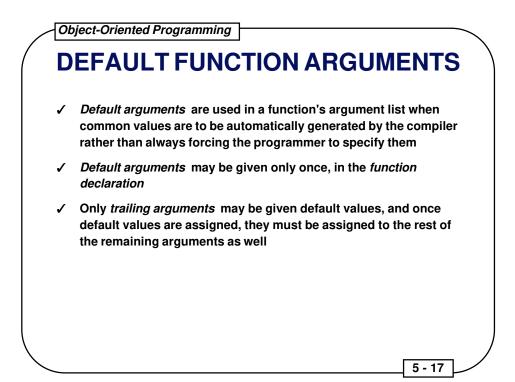
        cout << "\n";

5-13
```

CLA	SSES			
class				
is the preferred keyword for d	lefining new types in C++			
•	for the access of its members for the access of its members			
class typename {	struct typename {			
<pre>// private members</pre>	// public members			
public:	private:			
// public members	<pre>// private members</pre>			

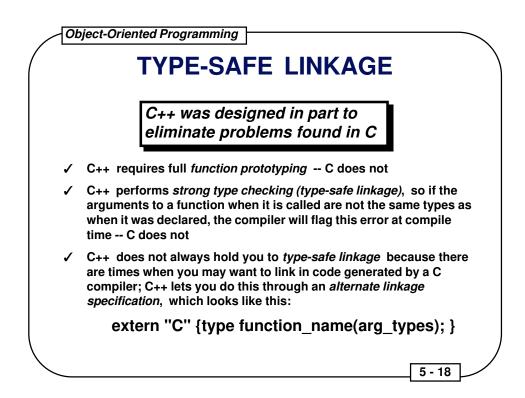






Example of function overloading:

```
#include <iostream.h>
struct ints {
 int a;
 int b;
};
void print(int value, int times = 1);
void print(char *value, int times = 1);
void print(struct ints *value, int times = 1);
void print (int i, int times) {
 for (int j = 0; j < times; j++) cout << "Integer: " << i << "\n";
}
void print (char *s, int times) {
 for (int j = 0; j < times; j++) cout << "String: " << s << "\n";
}
void print (struct ints * is, int times) {
 for (int j = 0; j < times; j++)
  cout << "Ints: " << is->a << ", " << is->b << "\n";
}
```

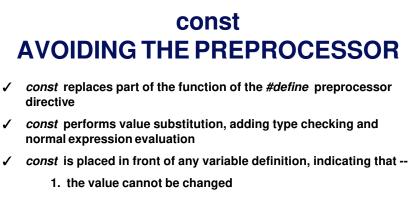


CONSTRUCTORS

- ✓ A constructor is used to initialize a variable based on a class when the variable is created
- ✓ A constructor is a member function of the class that has the same name as the class
- ✓ Constructor calls occur automatically at the point the variable is created, and the programmer cannot access the variable before the constructor is called
- ✓ Constructor functions may be overloaded like other member functions so that various kinds of initialization may be done
- ✓ Default arguments may also be used with constructor functions so long as ambiguities are not created
- ✓ Constructor functions are not required by C++, but they are often very convenient

DESTRUCTORS

- ✓ Destructor functions are used to ensure proper cleanup when a variable is destroyed
- ✓ A destructor function is a member function with the same name as the class preceded by a <u>tilde</u>
- ✓ Calls to *destructor functions* are automatic, occurring when a variable goes out of scope
- ✓ Destructor functions may not have any arguments
- ✓ Destructor functions are optional, like constructor functions
- ✓ Unlike *constructor functions*, only one *destructor function* may be declared



2. the compiler should try not to allocate storage, keeping the information in the symbol table instead

const float pi = 3.14159;

5 - 21

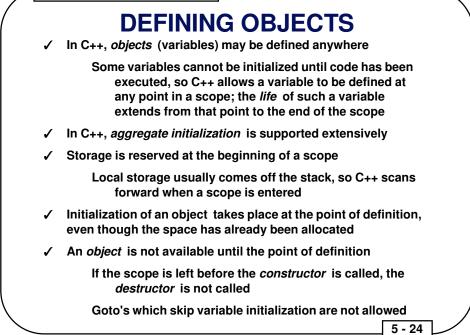
const IN ANSI C AND C++

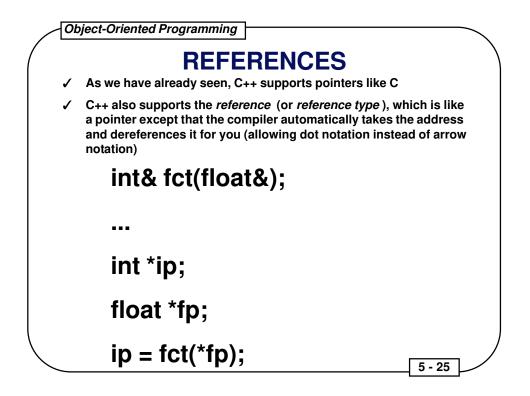
- ✓ const behaves differently in ANSI C and C++
- ✓ Linkage --
 - In C, const defaults to external linkage (global)
 - In C++, const defaults to internal linkage (local)
- ✓ Memory allocation --
 - In C, const always allocates storage for the value
 - In C++, const tries to store values in the symbol table
- ✓ Constant expressions (like array definitions) --
 - In C, const variables cannot be used in constant expressions (e.g., cannot be used in header files)
 - In C++, const variables can be used in constant expressions if symbol table storage is possible (i.e., elaborate structures are not involved)

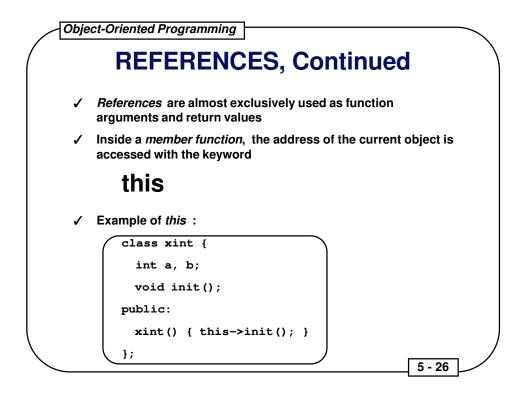


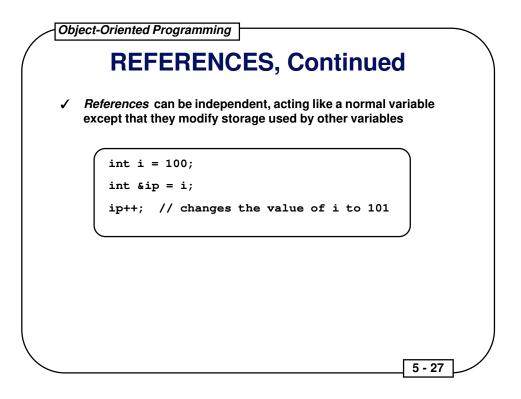
- ✓ In C++, the user can create *inline* functions, where, when they are called, their code itself is placed at the point of the call rather than a subroutine call instruction
- ✓ inline functions were created to replace the macro functions required in C to perform such code optimization
- ✓ Functions defined within a class declaration are automatically inline
- ✓ Global functions must use the *inline* keyword to become *inline*
- ✓ Full C++ type checking is performed on *inline* functions, like any other functions
- ✓ The prototype and function body of an *inline* function are stored in the *symbol table*

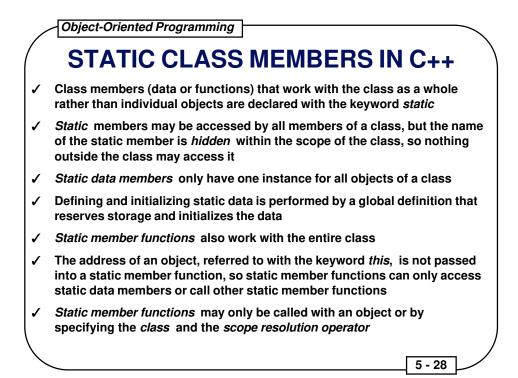






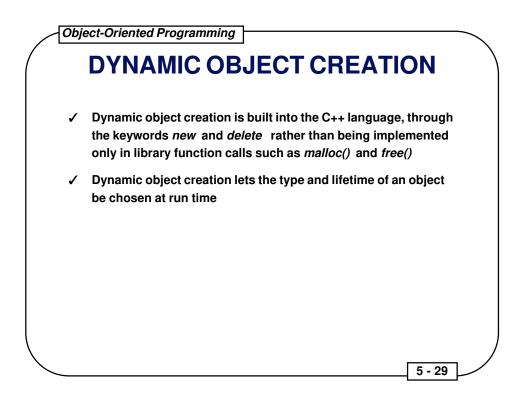


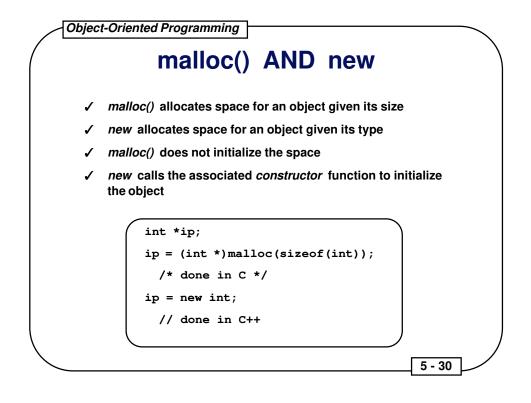




Example of static class members:

```
#include <iostream.h>
class objcounter {
 char *object_name;
 int number:
 static int counter; // one copy for all instances
public:
 objcounter (char *name); // constructor
 void whoami(void);
 static int number_of_objects(void);
};
int objcounter::counter = 0;
objcounter::objcounter(char *name) {
 object_name = name;
 counter++; // increment static data
 number = counter:
}
void objcounter::whoami(void) {
 cout << "Name: " << object name << ", Number: " << number << "\n";
}
int objcounter::number_of_objects(void) {
 return counter;
}
```

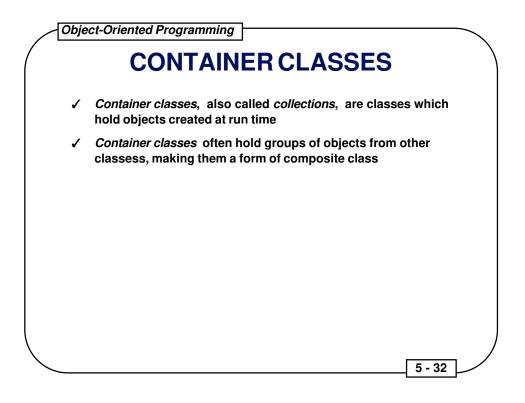




free() AND delete

- ✓ free() deallocates space provided by malloc()
- ✓ *delete* deallocates space provided by new
- ✓ free() does no cleanup other than freeing the space
- ✓ delete calls a destructor for the object

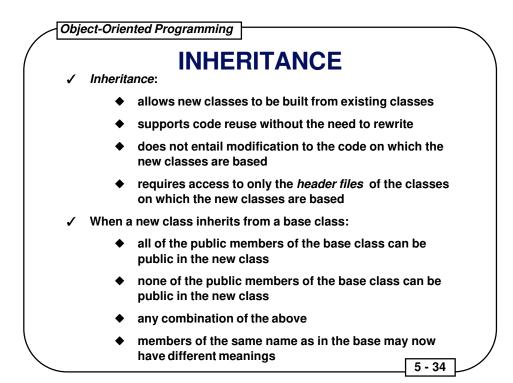
With the advent of *new* and *delete* in C++, there is no reasonable need for *malloc()* and *free()* except for compatibility with C

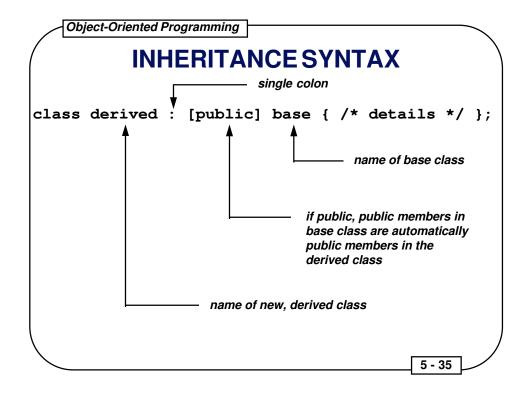


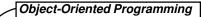
HEADER FILES

- ✓ In C++, a header file contains declarations only, not definitions
- ✓ A *header file* includes:
 - class declarations
 - function declarations
 - const values
 - anything else that is a part of the *public interface* to a class or library

- ✓ A header file must be insulated so the compiler sees its contents only once when compiling a file; preprocessor statements, like those used before for STORABLE.H, should be used to perform this insulation
- ✓ In essence, these preprocessor statements direct the header file to be skipped if it has already been included

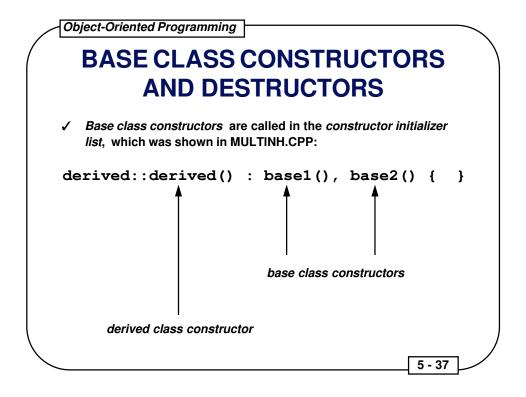






INHERITANCE

- ✓ Inheritance requires a lot of design-oriented thought in order to be applied correctly
- ✓ Use inheritance only when it makes sense -- is the derived class really an offspring of the base class, and does it make sense that the derived class should inherit capabilities from the base class?
- ✓ Breaking a problem into classes has the effect of *partitioning* the problem





- ✓ The way C++ calls base class constructors ensures that all derived class constructors can depend on the base class being properly initialized
- ✓ Up to one *destructor* may be defined for each class
- ✓ Destructors are called automatically, and all destructors are called for an object, which includes the destructors for its base classes, their base classes, and so on
- ✓ There is no destructor equivalent for the constructor initializer list
- ✓ Destructors are called from the top down (the opposite to the order of constructor calls)

5 - 38

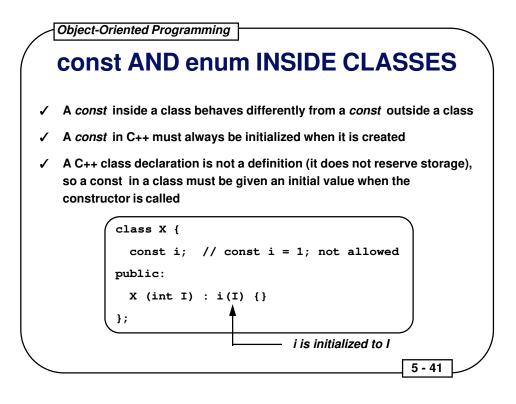
	WITH COMPOSI	TION
1	Inheritance is not the only way to create new classes from existing classes in C++; inheritance is sometimes said to represent an <i>is-a</i> relationship <i>Composition</i> is a method of building classes that <i>contain</i> objects of other classes; composition is sometimes said to represent a <i>has-</i> <i>a</i> relationship	
1		
	A car is a type of vehicle	inheritance
	A car has an engine and four wheels	composition

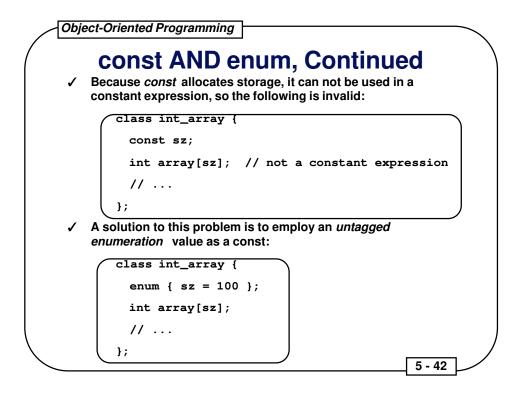
CREATING CLASSES WITH COMPOSITION

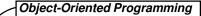
- ✓ Composition involves creating instances of a class inside another class
- ✓ If the objects have constructors which require arguments, those objects must be explicitly initialized in the *constructor initializer* list
- ✓ The order of calls in a *constructor initializer list* is not necessarily the order in which they appear; instead, the base class constructor is called first, and so on, and the member object constructors are called in the order in which the objects are declared in the class

5 - 40

✓ The constructor initializer list only determines the arguments given to the constructors, not the order of constructor calls





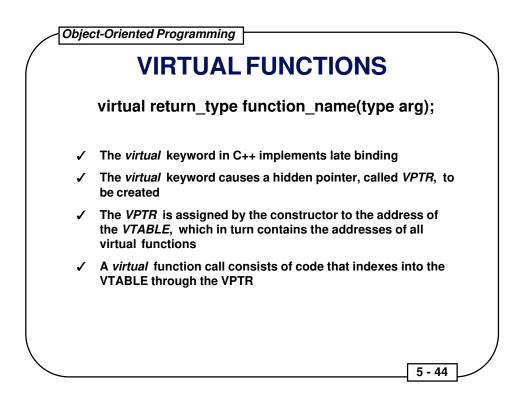


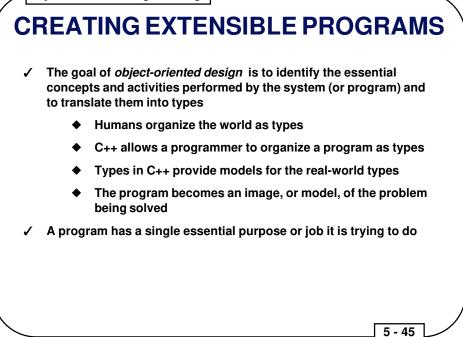


- Binding -- a linkage between a function call and a function definition
- ✓ Compile-time, static, or early, binding -- those linkages resolved during the run of the compiler and linker
- ✓ Run-time, dynamic, or late, binding -- linkages are resolved through a table of addresses of possible routines to call; this table is provided, and a particular table entry is selected during execution of the code

5 - 43

✓ The virtual function is the particular C++ feature which supports late binding





EXTENDING AN OBJECT-ORIENTED DESIGN

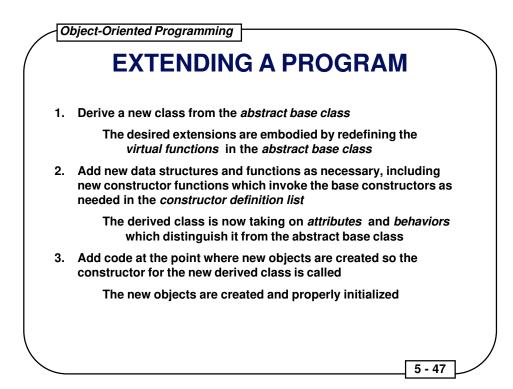
- ✓ Base classes generally represent the primary concepts of an object-oriented program
- Most base classes are *abstract*, representing concepts rather than specific things, so it does not make sense to create objects of an *abstract base class*
- ✓ C++ allows an abstract base class to contain pure *virtual* functions by assigning the function body to zero:

virtual void f() = 0;

✓ No objects can be created of such a class; objects may be created only from classes derived from this *abstract base class*

5 - 46

✓ These derived classes contain definitions for the pure virtual functions in the base class



OPERATOR OVERLOADING

- ✓ In C++, the meaning of almost any operator may be changed when that operator is used with variables of particular types
- ✓ The meaning of an operator changes only when an operator is used with the indicated types
- ✓ This permits the operators to be used as *infix* functions:

a + b;

✓ In the above example, the function "+" is applied to the target object "a" with the argument "b", just like set() below is applied to the target object "A" with the argument "B":

A.set(B);

✓ The syntax used for declaring the operator function for the "+" operator is:

5 - 48

return_type operator+ (type arg);