OBJECT-ORIENTED PROGRAMMING LANGUAGES -- C++ --

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

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++ LANGUAGE

- ✓ Developed by *Bjarne Stroustrup* starting in the early 1980's
- ✓ Based on merging features of *C* and *Simula-67* (developed in Scandinavia in 1967)
- ✓ Originally called *C with Classes* since it involved adding Simula-67's *class* concept to C
- ✓ *C with Classes* was later expanded by simply adding improvements to C (*not* to implement object orientation necessarily), so the concept of *the next step after C*, or *C++* (the C increment operator is ++), evolved

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**

AN OBJECT-ORIENTED C
C++

- ✓ **Developed to take advantage of the ease of programming provided by an OOPL**
- ✓ **Developed to provide an easy learning path for C programmers**
- ✓ **Developed to fix defects in C which allow certain kinds of bugs to slip through the compiler -- bugs which may go unnoticed until runtime**

C++ allows the programmer to focus on *concepts* rather than forcing him to concentrate on the code which implements those concepts

THE ANSI C++ STANDARD

- ✓ **ANSI committee *X3J16* was created to produce an international standard for C++, which is still in development**
- ✓ **Most of today's C++ compilers deviate from the standard in one way or another, so portability of code between different C++ compilers on different platforms tends to suffer today**
- ✓ ***GNU C++* is becoming a standard in its own right due to the fact that it is free and it runs on many platforms, including 386 PCs and workstations, but *GNU C++* does not conform to the C++ standard exactly**

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

THREE WAYS OF USING C++

- ✓ ***Like C or C with extensions*** **-- many C programs may be compiled with a C++ compiler with little or no modifications (mainly in the area of function prototypes)**
- ✓ ***Like C with enhanced data abstration capabilities*** **-- more sophisticated data structures may be manipulated with greater ease in C++**
- ✓ ***Like an OOPL*** **-- all the benefits of contemporary *object-oriented programming* may be achieved through C++**

TYPES = STRUCTS + FUNCTIONS

A *type* is a C *struct* with *functions*

```
struct complex {      /* The C struct */
  float real_part;
  float imag_part;
};
```

```
struct complex {      // The C++ struct
  float real_part;
  float imag_part;
  complex();          // a constructor (discussed later)
  void add (complex, complex);  // operates on object
};
```


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 */
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
```

SCOPE RESOLUTION OPERATOR ::

- ✓ ***Member functions*** **associated with a struct are** ***declared*** **as** ***function prototypes*** **in the struct**
- ✓ **When** ***member function*****s are** ***defined*****, their associated struct is specified using the** ***scope resolution operator*** **(::)**

void struct_name::member_function_name() { /* body */ };

as in

void complex::add (complex left, complex right)

{ /* body */ };

SCOPE RESOLUTION OPERATOR, Continued

✓ **The *scope resolution operator* may be used whenever the compiler would not normally choose the desired name**

```
int x;
void main() {
  int x;
  x = 2;    // local X is assigned
  ::x = 4;  // global X is assigned
};
```

MEMBER FUNCTION SCOPE

✓ **A member function may access any other member in the same struct, including both data and other member functions**

```
void complex::add(complex left, complex right) {
  real_part = left.real_part + right.real_part;
    // note that the real_part left of the
    // equal size refers to the real_part
    // of the target object
  imag_part = left.imag_part + right.imag_part;
};
```


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)
    while (++floor_number < floor_selected)
      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);
}
```

DATA PROTECTION

Access to data and functions within a struct is controlled by the three *access specifiers* :

- ✓ ***private* -- prevents access except by other members**
- ✓ ***protected* -- like *private*, except inherited classes also have access (inheritance is discussed later)**
- ✓ ***public* -- permits everyone, including end users, to access the members**

Access to *private* and *protected* members can be granted to non-member functions by using the *<u>friend</u>* keyword when declaring the non-member function inside a struct

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";
}
```

CLASSES

class

is the preferred keyword for defining new types in C++

- ✓ ***struct* defaults to *public* for the access of its members**
- ✓ ***class* defaults to *private* for the access of its members**

`class typename {`
`  // private members`
`public:`
`  // public members`
`};`
	`struct typename {`
	`  // public members`
	`private:`
	`  // private members`
	`};`

AUTOMATIC typedef DECLARATIONS

The *tag names* of these entities are designated as reserved words within their scope automatically (similar to doing a *typedef* in C), and the form of their declarations and definitions are similar:

- ✓ ***class***
- ✓ ***struct***
- ✓ ***union***
- ✓ ***enum***

FUNCTION OVERLOADING

- ✓ ***Function Overloading*** **allows more than one function to be given the same name** ***as long as all these functions have distinct argument lists***
- ✓ ***Function Overloading*** **prevents name clashes when** ***multiple libraries*** **come into use**
- ✓ **Function overloading works through** ***name mangling*****, where the compiler-generated name for the function includes information on the types of its arguments**
- ✓ **Examples of overloaded functions:**

```
void print(int);

void print(int, char);

void print(double);
```

DEFAULT FUNCTION ARGUMENTS

- ✓ ***Default arguments*** **are used in a function's argument list when common values are to be automatically generated by the compiler rather than always forcing the programmer to specify them**
- ✓ ***Default arguments*** **may be given only once, in the** ***function declaration***
- ✓ **Only** ***trailing arguments*** **may be given default values, and once default values are assigned, they must be assigned to the rest of the remaining arguments as well**

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";
}
```

TYPE-SAFE LINKAGE

C++ was designed in part to eliminate problems found in C

- ✓ **C++ requires full *function prototyping* -- C does not**
- ✓ **C++ performs *strong type checking (type-safe linkage)*, so if the arguments to a function when it is called are not the same types as when it was declared, the compiler will flag this error at compile time -- C does not**
- ✓ **C++ does not always hold you to *type-safe linkage* because there are times when you may want to link in code generated by a C compiler; C++ lets you do this through an *alternate linkage specification*, which looks like this:**

```
extern "C" {type function_name(arg_types); }
```

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 tilde**
- ✓ **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**

const
AVOIDING THE PREPROCESSOR

- ✓ ***const*** **replaces part of the function of the *#define* preprocessor directive**
- ✓ ***const*** **performs value substitution, adding type checking and normal expression evaluation**
- ✓ ***const*** **is placed in front of any variable definition, indicating that --**
  1. **the value cannot be changed**
  2. **the compiler should try not to allocate storage, keeping the information in the symbol table instead**

```
const float pi = 3.14159;
```

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)

inline FUNCTIONS

- ✓ **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***

DEFINING OBJECTS

✓ In C++, *objects* (variables) may be defined anywhere

  Some variables cannot be initialized until code has been executed, so C++ allows a variable to be defined at any point in a scope; the *life* of such a variable extends from that point to the end of the scope

✓ In C++, *aggregate initialization* is supported extensively

✓ Storage is reserved at the beginning of a scope

  Local storage usually comes off the stack, so C++ scans forward when a scope is entered

✓ Initialization of an object takes place at the point of definition, even though the space has already been allocated

✓ An *object* is not available until the point of definition

  If the scope is left before the *constructor* is called, the *destructor* is not called

  Goto's which skip variable initialization are not allowed

REFERENCES

- ✓ As we have already seen, C++ supports pointers like C
- ✓ C++ also supports the *reference* (or *reference type*), which is like a pointer except that the compiler automatically takes the address and dereferences it for you (allowing dot notation instead of arrow notation)

```
int& fct(float&);

...

int *ip;

float *fp;

ip = fct(*fp);
```

REFERENCES, Continued

- ✓ ***References*** **are almost exclusively used as function arguments and return values**
- ✓ **Inside a *member function*, the address of the current object is accessed with the keyword**

this

- ✓ **Example of *this* :**

```
class xint {
  int a, b;
  void init();
public:
  xint() { this->init(); }
};
```

REFERENCES, Continued

✓ *References* can be independent, acting like a normal variable except that they modify storage used by other variables

```
int i = 100;
int &ip = i;
ip++;  // changes the value of i to 101
```

STATIC CLASS MEMBERS IN C++

- ✓ **Class members (data or functions) that work with the class as a whole rather than individual objects are declared with the keyword *static***
- ✓ ***Static* members may be accessed by all members of a class, but the name of the static member is *hidden* within the scope of the class, so nothing outside the class may access it**
- ✓ ***Static data members* only have one instance for all objects of a class**
- ✓ **Defining and initializing static data is performed by a global definition that reserves storage and initializes the data**
- ✓ ***Static member functions* also work with the entire class**
- ✓ **The address of an object, referred to with the keyword *this*, is not passed into a static member function, so static member functions can only access static data members or call other static member functions**
- ✓ ***Static member functions* may only be called with an object or by specifying the *class* and the *scope resolution operator***

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;
}
```

DYNAMIC OBJECT CREATION

- ✓ **Dynamic object creation is built into the C++ language, through the keywords *new* and *delete* rather than being implemented only in library function calls such as *malloc()* and *free()***
- ✓ **Dynamic object creation lets the type and lifetime of an object be chosen at run time**

malloc() AND new

- ✓ *malloc()* allocates space for an object given its size
- ✓ *new* allocates space for an object given its type
- ✓ *malloc()* does not initialize the space
- ✓ *new* calls the associated *constructor* function to initialize the object

```
int *ip;
ip = (int *)malloc(sizeof(int));
  /* done in C */
ip = new int;
  // done in C++
```

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

CONTAINER CLASSES

- ✓ ***Container classes***, **also called** ***collections***, **are classes which hold objects created at run time**
- ✓ ***Container classes*** **often hold groups of objects from other classess, making them a form of composite class**

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***:
  - allows new classes to be built from existing classes
  - supports code reuse without the need to rewrite
  - does not entail modification to the code on which the new classes are based
  - requires access to only the *header files* of the classes on which the new classes are based
- When a new class inherits from a base class:
  - all of the public members of the base class can be public in the new class
  - none of the public members of the base class can be public in the new class
  - any combination of the above
  - members of the same name as in the base may now have different meanings

INHERITANCE SYNTAX

single colon

```
class derived : [public] base { /* details */ };
```

name of base class

if public, public members in base class are automatically public members in the derived class

name of new, derived class

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**

BASE CLASS CONSTRUCTORS AND DESTRUCTORS

✓ ***Base class constructors*** **are called in the** ***constructor initializer list*****, which was shown in MULTINH.CPP:**

```
derived::derived() : base1(), base2() {  }
```

base class constructors

derived class constructor

DERIVED CLASSES

- ✓ **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)**

CREATING CLASSES WITH COMPOSITION

- ✓ ***Inheritance*** **is not the only way to create new classes from existing classes in C++; *inheritance* is sometimes said to represent an *is-a* relationship**
- ✓ ***Composition*** **is a method of building classes that *contain* objects of other classes; composition is sometimes said to represent a *has-a* relationship**

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

const AND enum INSIDE CLASSES

- ✓ A *const* inside a class behaves differently from a *const* outside a class
- ✓ A *const* in C++ must always be initialized when it is created
- ✓ A C++ class declaration is not a definition (it does not reserve storage), so a const in a class must be given an initial value when the constructor is called

```
class X {
  const i;  // const i = 1; not allowed
public:
  X (int I) : i(I) {}
};
```

i is initialized to I

const AND enum, Continued

✓ Because *const* allocates storage, it can not be used in a constant expression, so the following is invalid:

```
class int_array {
  const sz;
  int array[sz];  // not a constant expression
  // ...
};
```

✓ A solution to this problem is to employ an *untagged enumeration* value as a const:

```
class int_array {
  enum { sz = 100 };
  int array[sz];
  // ...
};
```

EARLY AND LATE BINDING

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

VIRTUAL FUNCTIONS

virtual return_type function_name(type arg);

- ✓ **The *virtual* keyword in C++ implements late binding**
- ✓ **The *virtual* keyword causes a hidden pointer, called *VPTR*, to be created**
- ✓ **The *VPTR* is assigned by the constructor to the address of the *VTABLE*, which in turn contains the addresses of all virtual functions**
- ✓ **A *virtual* function call consists of code that indexes into the VTABLE through the VPTR**

CREATING EXTENSIBLE PROGRAMS

- ✓ The goal of *object-oriented design* is to identify the essential concepts and activities performed by the system (or program) and to translate them into types
  - ◆ Humans organize the world as types
  - ◆ C++ allows a programmer to organize a program as types
  - ◆ Types in C++ provide models for the real-world types
  - ◆ The program becomes an image, or model, of the problem being solved
- ✓ A program has a single essential purpose or job it is trying to do

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

EXTENDING A PROGRAM

1. **Derive a new class from the *abstract base class***

   The desired extensions are embodied by redefining the *virtual functions* in the *abstract base class*

2. **Add new data structures and functions as necessary, including new constructor functions which invoke the base constructors as needed in the *constructor definition list***

   The derived class is now taking on *attributes* and *behaviors* which distinguish it from the abstract base class

3. **Add code at the point where new objects are created so the constructor for the new derived class is called**

   The new objects are created and properly initialized

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:

```
return_type  operator+  (type arg);
```