KASNEB

CICT

PART 2

SECTION 4

Object Oriented Programming

Contents

TOPIC 1

AN OVERVIEW OF PROGRAMMING TECHNIQUES AND PARAGIGMS

Object-oriented programming (**OOP**) - A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

Object-oriented programming (**OOP**) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.* A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this"). In object-oriented programming, computer programs are designed by making them out of objects that interact with one another. There is significant diversity in object-oriented programming, but most popular languages are class-based, meaning that objects are instances of classes, which typically also determines their type.

Programming paradigms

A **programming paradigm** is a fundamental style of computer **programming**, a way of building the structure and elements of computer programs.

The following are considered the main programming paradigms. There is inevitably some overlap in these paradigms but the main features or identifiable differences are summarized below:

- **Imperative programming** – defines computation as statements that change a program state

- **Procedural programming**, structured programming – specifies the steps the program must take to reach the desired state.

- **Declarative programming** – defines computation logic without defining its control flow.

TOPIC 3

CLASS DEFINITION AND IMPLEMENTATION

Creating Classes

An Example of a structs vs. classes

Drawbacks to using a struct:

1. Initialization not specifically required, and may be done incorrectly
2. Invalid values can be assigned to struct members (as in example)
3. If struct implementation changes, all the programs using the struct must also change
4. Structs aren't *first-class*: they cannot be printed as a unit or compared

Class Definition

Basic form is:

```
class class-name {
      member-list
};
```

Class name has file scope and external linkage. E.g. *Point* class for points on a 2D cartesian graph:

```
class Point {
public:
      int GetX();        // Returns the value of x.
      void SetX();       // Sets the value of x.
private:
      int x;        // x-coordinate
      int y;        // y-coordinate
};
```

Keywords *public* and *private* are *access specifiers*. Note that storage is not allocated until an instance of the class is created (there is also an access specifier *protected*, which we will avoid for now)

Class Objects

General format is: *class-name identifier.* We can access class *methods* using the dot operator ("."),
E.g.

```
Point P;            // Note this is different than Point P();
P.SetX( 300 );
cout << "x-coordinate is: " << P.GetX();
```

Copying from one class object to another by default does a bit-wise copy, such as

```
Point P1, P2;
...
P2 = P1;
```

Controlling Member Access

A *member* may be either a function or a data item

1. Access specifiers:
    a. public: non-member function can access these members
    b. private: member of same class only can access these members
    c. protected: same as private, only derived classes can access these members
2. If no access specifier is given, the default is *private*.
3. Access specifiers may be listed multiple times, though good style dictates that all *public* members are given first, then *private*.
    E. Example: the Point class

```
class Point {
public:
     int GetX()    // Returns value of X
     {
         return x;
     }                    .

     void SetX( int xval) // Sets value of X
     {
         x = xval;
     }
private:
     int x;        // x-coordinate
     int y;        // y-coordinate. Note y fcns. not shown
```

```
    }; //End of Point class
```

Short functions as shown above may be given all on one line.

Types of Member functions

1. *Accessor*, or *selector* functions (e.g. GetX)
2. *Mutator* functions (e.g. SetX)
3. *Operators* that let you define C++ operators for each class object (e.g. move a point)
4. *Iterators* that process a collection of objects (e.g. recompute each Account balance)

Inline and Out-of-line definitions

Rather than list a member function definition along with the declaration under the *public* access specifier, we can leave the member function *declaration* as public but hide the implementation details. E.g.

```
class Point {
public:
     void SetX(int xval); // Sets value of X.
     ...
private:
     int x;         // x-coordinate
     ...
}; //End of Point class

void Point::Setx( int xval)
{
     return x;
}
```

1. The binary scope resolution operator "::" specifies that *Setx* corresponds to the *Point* class. It can access the private data members only because it has this correspondance.
2. The member function *SetX* must specify the *type* of parameters, though parameter names need not be given. (In fact, given parameter names need not be the same as in the definition) Since this declaration specifies the interface, it is a good idea to provide the parameter names as well as brief documentation.

TOPIC 4

FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

Different programming languages name functions differently like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers

int max(int num1, int num2)  {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

TOPIC 6

MEMORY MANAGEMENT

Dynamic Memory Management

There are two ways to allocate memory:

(i) Static memory allocation
(ii) Dynamic memory allocation

(i) Static memory allocation

To allocate memory at the time of program compilation is known as static memory allocation.

```
        i.e. int a[10];
```

It allocates 20 bytes at the time of compilation of the program. Its main disadvantage is wastage or shortage of memory space can takes place.

(ii) Dynamic memory allocation

To allocate memory at the time of program execution is known as dynamic memory allocation. C++ provides two dynamic allocation operators: new and delete. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs.

These are included for the sake of compatibility with C. However, for C++ code, you should use the new and delete operators because they have several advantages. The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

The general forms of new and delete are shown here:

```
     p_var = new type;
     delete p_var;
```

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type.

<u>Memory Allocation to an integer.</u>

Solution:

```
#include <iostream.h>
void main()
{
   int *p;
   try
{
   p = new int;
}
   catch (bad_alloc xa)
{
   cout << "Allocation Failure\n";
   return 1;
}
   *p = 100;
   cout << "At " << p << " ";
   cout << "is the value " << *p << "\n";
   delete p;
}
```

This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately.The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Memory in your C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

Memory allocation Issues and Problems

As a general rule, dynamic behavior is troublesome in real time embedded systems. The two key areas of concern are determination of the action to be taken on resource exhaustion and nondeterministic execution performance.

There are a number of problems with dynamic memory allocation in a real time system. The standard library functions (malloc() and free()) are not normally reentrant, which would be problematic in a multithreaded application. If the source code is available, this should be straightforward to rectify by locking resources using RTOS facilities (like a semaphore). A more intractable problem is associated with the performance of malloc(). Its behavior is unpredictable, as the time it takes to allocate memory is extremely variable. Such nondeterministic behavior is intolerable in real time systems.

TOPIC 7

REFERENCES AND ARGUMENT PASSING

A **reference** is a value that enables a **program** to indirectly access a particular data, such as a variable or a record, in the computer's memory or in some other storage device. The **reference** is said to refer to the datum, and accessing the datum is called dereferencing the **reference**.

Basic Syntax

Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name, such as this "reference to an int"

```
int& foo = ....;
```

Did you notice the "...."? (Probably, right? After all, it's 25% of the example.) When a reference is created, you must tell it which variable it will become an alias for. After you create the reference, whenever you use the variable, you can just treat it as though it were a regular integer variable. But when you create it, you must initialize it with another variable, whose address it will keep around behind the scenes to allow you to use it to modify that variable.

In a way, this is similar to having a pointer that always points to the same thing. One key difference is that references do not require dereferencing in the same way that pointers do; you just treat them as normal variables. A second difference is that when you create a reference to a variable, you need not do anything special to get the memory address. The compiler figures this out for you:

```
int x;
int& foo = x;

// foo is now a reference to x so this sets x to 56
foo = 56;
std::cout << x <<std::endl;
```

Note:

Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (`&`), known as ***address-of operator.*** For example:

```
foo = &myvar;
```

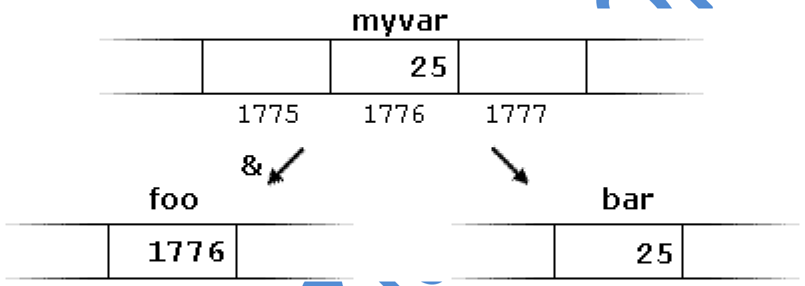
This would assign the address of variable `myvar` to `foo`; by preceding the name of the variable `myvar` with the *address-of operator* (`&`), we are no longer assigning the content of the variable itself to `foo`, but its address.

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that `myvar` is placed during runtime in the memory address `1776`.

In this case, consider the following code fragment:

```
myvar = 25;
foo = &myvar;
bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:



First, we have assigned the value `25` to `myvar` (a variable whose address in memory we assumed to be `1776`).

The second statement assigns `foo` the address of `myvar`, which we have assumed to be `1776`.

Finally, the third statement, assigns the value contained in `myvar` to `bar`. This is a standard assignment operation, as already done many times in earlier chapters.

The main difference between the second and third statements is the appearance of the *address-of operator* (`&`).

The variable that stores the address of another variable (like `foo` in the previous example) is what in C++ is called a *pointer*. Pointers are a very powerful feature of the language that has many uses in lower level programming. A bit later, we will see how to declare and use pointers.

Dereference operator (*)

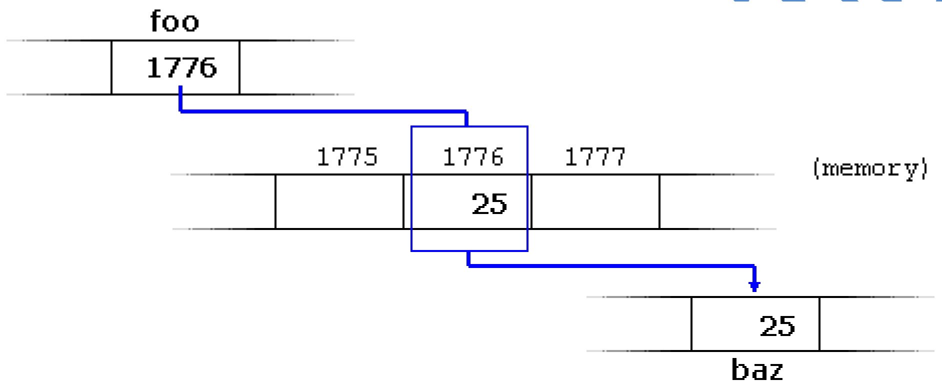
As just seen, a variable which stores the address of another variable is called a *pointer*. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (`*`). The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, the following statement:

```
 baz = *foo;
```

This could be read as: "`baz` equal to value pointed to by `foo`", and the statement would actually assign the value `25` to `baz`, since `foo` is `1776`, and the value pointed to by `1776` (following the example above) would be `25`.



It is important to clearly differentiate that `foo` refers to the value `1776`, while `*foo` (with an asterisk `*` preceding the identifier) refers to the value stored at address `1776`, which in this case is `25`. Notice the difference of including or not including the *dereference operator* (I have added an explanatory comment of how each of these two expressions could be read):

```
baz = foo;   // baz equal to foo (1776)
baz = *foo;  // baz equal to value pointed to by foo (25)
```

The reference and dereference operators are thus complementary:

- `&` is the *address-of operator*, and can be read simply as "address of"
- `*` is the *dereference operator*, and can be read as "value pointed to by"

Thus, they have sort of opposite meanings: An address obtained with `&` can be dereferenced with `*`.

Earlier, we performed the following two assignment operations:

```
myvar = 25;
foo = &myvar;
```

Right after these two statements, all of the following expressions would give true as result:

```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

The first expression is quite clear, considering that the assignment operation performed on `myvar` was `myvar=25`. The second one uses the address-of operator (`&`), which returns the address of `myvar`, which we assumed it to have a value of `1776`.

TOPIC 9

INTRODUCTION TO INHERITANCE

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE : All members of a class except Private, are inherited

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

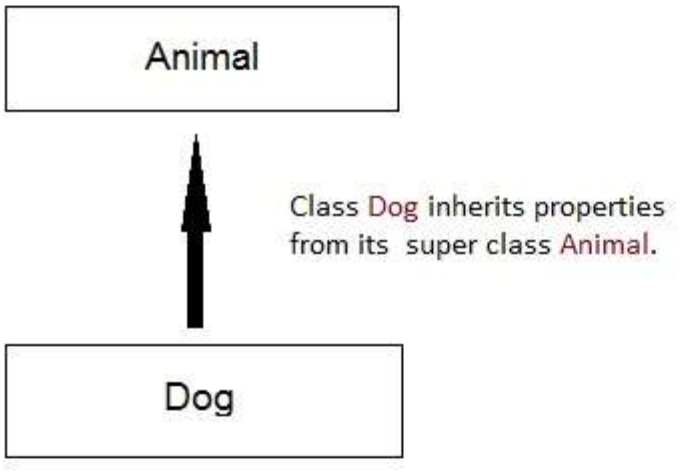
Basic Syntax of Inheritance

```
class Subclass_name : access_mode Superclass_name
```

While defining a subclass like this, the super class must be already defined or atleast declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Example of Inheritance



```
class Animal
{ public:
  int legs = 4;
};

class Dog : public Animal
{ public:
  int tail = 1;
};

int main()
{
 Dog d;
 cout << d.legs;
 cout << d.tail;
}
```

Output :

```
4 1
```

Base & Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape  {
   public:
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total area: 35
```