UNIT II

IMPLEMENTING ADTS AND ENCAPSULATION

Aggregate type structure

An aggregate type is an array, class, or structure type which:

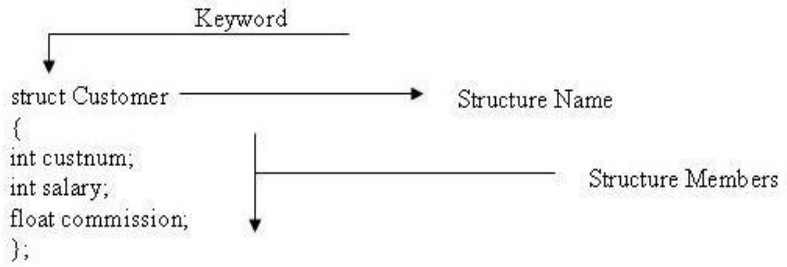
- Has no constructors
- Has no nonpublic members
- Has no base classes
- Has no virtual functions

Structure:

A structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

Declaring a Structure:

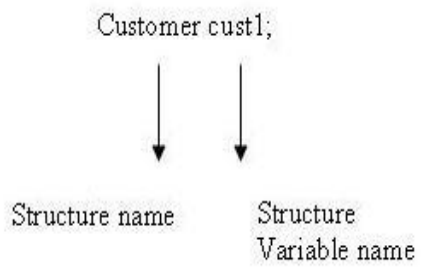
The structure is declared by using the keyword struct followed by structure name, also called a tag. Then the structure members (variables) are defined with their type and variable names inside the open and close braces "{"and "}". Finally, the closed braces end with a semicolon denoted as ";" following the statement. The above structure declaration is also called a Structure Specifier



How to declare Structure Variable?

This is similar to variable declaration. For variable declaration, data type is defined followed by variable name. For structure variable declaration, the data type is the name of the structure followed by the structure variable name.

In the above example, structure variable cust1 is defined as:



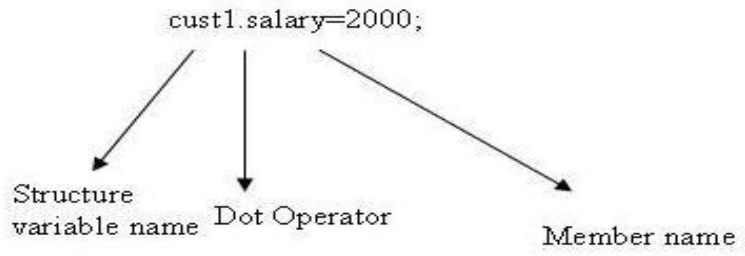
How to access structure members in C++?

To access structure members, the operator used is the dot operator denoted by (.). The dot operator for accessing structure members is used thusly:

structure_variable_name.member_name

For example:

A programmer wants to assign 2000 for the structure member salary in the above example of structure Customer with structure variable cust1 this is written as:




```
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  database employee;  //There is now an employee variable that has modifiable
                      // variables inside it.
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

Structure pointer operators:

This pointer indirection operator(→) is used to access a member of a structure using pointer.

```
#include <iostream.h>
struct MyStructure
{
   int n;
   float f;
   float f2;
};

int main()
{
 MyStructure myMyStructure;
 MyStructure *ptrMyStructure;

 //set the f of the structure
 myMyStructure.f = 1000;

 //intialize the pointer
 ptrMyStructure = &myMyStructure;

 //change the pointers f
 ptrMyStructure->f = 2000;

 //print out the structures f
 cout << myMyStructure.f << "\n";

 return 0;
}
```

Unions:

Union is similar to `struct` (more that `class`), unions differ in the aspect that the fields of a `union` share the same position in memory and are by default `public` rather than `private`. The size of the `union` is the size of its largest field. The **union** keyword is used to define a union type.

Syntax

```
union union-name {
Public-members-list;
Private:
Private-members-list;
} object-list;
```

ACCESSING UNION FIELDS

To access the fields of a union, use the dot operator(.) just as you would for a structure. When a value is

assigned to one member, the other member(s) get whipped out since they share the same memory. Using the example above, the precise time can be accessed like this:

```
union time
{
long simpleDate;
double perciseDate;

}mytime;

printTime( mytime.perciseDate );
```

Bit fields:

Classes and structures can contain members that occupy less storage than an integral type. These members are specified as bit fields. The syntax for bit-field *member-declarator* specification follows:

```
declaratoropt  : constant-expression
```

The *declarator* is the name by which the member is accessed in the program. It must be an integral type (including enumerated types). The *constant-expression* specifies the number of bits the member occupies in the structure. Anonymous bit fields — that is, bit-field members with no identifier — can be used for padding.

Note An unnamed bit field of width 0 forces alignment of the next bit field to the next *type* boundary, where *type* is the type of the member.

The following example declares a structure that contains bit fields:

```
// bit_fields1.cpp
struct Date
{
   unsigned nWeekDay  : 3;
   unsigned nMonthDay : 6;
   unsigned nMonth    : 5;
   unsigned nYear     : 8;
};
```

Data handling:

Access specifiers defines the access rights for the statements or functions that follows it until another access specifier or till the end of a class. The three types of access specifiers are "private", "public", "protected".

private:

The members declared as "private" can be accessed only within the same class and not from outside the class.

public:

The members declared as "public" are accessible within the class as well as from outside the class.

protected:

The members declared as "protected" cannot be accessed from outside the class, but can be accessed from a derived class. This is used when inheritance is applied to the members of a class.

```
class MyClass
{
    public:
        int a;
    protected:
        int b;
    private:
        int c;
};

int main()
{
    MyClass obj;
    obj.a = 10;     //Allowed
    obj.b = 20;     //Not Allowed, gives compiler error
    obj.c = 30;     //Not Allowed, gives compiler error
}
```

Member functions:

Classes can contain data and functions. These functions are referred to as "member functions." Any nonstatic function declared inside a class declaration is considered a member function and is called using the member-selection operators (**.** and **–>**). When calling member functions from other member functions of the same class, the object and member-selection operator can be omitted. For example:

Copy

```
#include <iostream.h>

class Point
{
public:
   int x()
   {
      return _x;
   }

int y()
   {
```

```
      return _y;
   }

   void Show()
   {
      cout << x() << ", " << y() << "\n";
   }
private:
   short _x, _y;
};

int main()
{
   Point pt;
   pt.Show();
}
```

Note that in the member function, `Show`, calls to the other member functions, `x` and `y`, are made without member-selection operators. These calls implicitly mean `this->x()` and `this->y()`. However, in **main**, the member function, `Show`, must be selected using the object `pt` and the member-selection operator (**.**).

Classes:

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
  } object_names;
```

Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

```
// classes example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};
```

```
void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Constructors:

A *constructor* is a member function with the same name as its class. For example:

```
class X {
public:
  X();      // constructor for class X
};
```

Constructors are used to create, and can initialize, objects of their class type.

You cannot declare a constructor as `virtual` or `static`, nor can you declare a constructor as `const`, `volatile`, or `const volatile`.

You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value.

Default Constructor:

A *default constructor* is a constructor that either has no parameters, or if it has parameters, *all* the parameters have default values.

In C++, default constructors are significant because they are automatically invoked in certain circumstances:

- When an object value is declared with no argument list, e.g. `MyClass x;`; or allocated dynamically with no argument list, e.g. `new MyClass`; the default constructor is used to initialize the object
- When an array of objects is declared, e.g. `MyClass x[10];`; or allocated dynamically, e.g. `new MyClass [10]`; the default constructor is used to initialize all the elements
- When a derived class constructor does not explicitly call the base class constructor in its initializer list, the default constructor for the base class is called
- When a class constructor does not explicitly call the constructor of one of its object-valued fields in its initializer list, the default constructor for the field's class is called

```
class MyClass
{
```

```
    int x;
    int y;
public:
    MyClass();                                  // constructor declared
};

MyClass :: MyClass()                            // constructor defined
{
    x = 100;
    y = 200;
}

int main()
{
    MyClass* object_1 = new MyClass();                          // object
created
}
```

Parameterized Constructor:

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to 1. For example:

```
#include <iostream.h>

class myclass {
  int a, b;
public:
  myclass(int i, int j) {
     a=i;
     b=j;
  }

  void show() {
     cout << a << " " << b;
  }
};
int main()
{
  myclass ob(3, 5);
  ob.show();
  return 0;
}
```

Copy constructors

Copy constructors define the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object).

It is used to create a copy of an existing object of the same class. Even though both classes are the same, it counts as a conversion constructor.

Destructors:

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared `const`, `volatile`, `const volatile` or `static`. A destructor can be declared `virtual` or pure `virtual`.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class `A` has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for `A`:

```
  A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class `A` before defining the implicitly declared destructor of `A`

Static member:

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>
```

```
using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

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

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{

   // Print total number of objects before creating object.
   cout << "Inital Stage Count: " << Box::getCount() << endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() << endl;
```

```
   return 0;
}
```

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

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

This pointer:

- Every object has a special pointer "this" which points to the object itself.
- This pointer is accessible to all members of the class but not to any static members of the class.
- Can be used to find the address of the object in which the function is a member.
- Presence of this pointer is not included in the sizeof calculations.

Syntax:

```
this
this->member-identifier
```

```
#include <iostream>
using namespace std;

class MyClass {
        int data;
    public:
        MyClass() {data=100;};
        void Print1();
        void Print2();
};

// Not using this pointer
void MyClass::Print1() {
    cout << data << endl;
}

// Using this pointer
void MyClass::Print2() {
    cout << "My address = " << this << endl;
    cout << this->data << endl;
}


int main()
{
    MyClass a;
```

```
    a.Print1();
    a.Print2();

    // Size of doesn't include this pointer
    cout << sizeof(a) << endl;
}

OUTPUT:
100
My address = 0012FF88
100
4
```

Implementation of simple ADTs:

An ADT is *implemented* by supplying

- a *data structure* for the type name.
- coded *algorithms* for the operations.

We sometimes refer to the ADT itself as the *ADT specification* or the *ADT interface*, to distinguish it from the code of the ADT implementation.

```
class Book {
public:
  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  Author* getAuthor (int authorNumber) const;
  void addAuthor (Author*);
  void removeAuthor (Author*);

  std::string getIdentifier() const;
  void setIdentifier(std::string id);

private:
  std::string title;
  int numAuthors;
  std::string identifier;
  :
};
```

In C++, this is generally done using a C++ `class`. The class enforces the ADT contract by dividing the information about the implementation into `public` and `private` portions. The ADT designer declares all items that the application programmer can use as public, and makes the rest private. An application programmer who then tries to use the private information will be issued error messages by the compiler.