Last Open 11/27/2019 10:30 AM

4-Nov-19

Chapter 1 Principal of OOP

Layer of Computer Software

Machine -> Assembly -> Procedure -> Object-oriented

Organizing instruction in group is known as **function**

Procedure oriented programming there is **no relation between function and data** so if there are multiple function so which function change which global data its hard to troubleshoot. and it also does not model the real world. function is action oriented

OOP treat **data as critical** element and does **not allow** data to move **freely** around the system.

POP	**OOP**
Top-Down approach	Bottom-up approach

Concept of OOP

Object:

Run time entity of system. object is created from class and its has data and member function. so object are variable of type class.

Class

Class is user-defined data-type and behave like built-in type of programming language.

Encapsulation

the wrapping up of data and function into a single unit(called class) is known as encapsulation.

Abstraction

Representing essential feature without including the background details of explanations.

Inheritance

Object of one class can acquire property of another class. this is the idea of re-usability.

Polymorphism

ability to take more than one form.

the behaviour depends upon the type of data used in operation.

opration of (add) : if **int** var **10 20** to addition will **30**. if **string hello world** become **helloworld**

operator overloading : operator to exhibit different behaviour in different instance.

function overloading : single function name to perform different type of tasks.

Chapter 2 : Beginning with CPP

C plus plus is object oriented programming language is designed by **Bjarne Strostrup** at AT&T bell laboratories in New Jersey USA **early 1980'**s.

Cpp is superset of C.

Namespace : new concept introduced in ANSI C++ standard committee.define scope of a identifier used in program.

cout and **cin** is object iostream class. cin can read only one word.

Structure of CPP program

include files
class declaration
member function declaration
main function

Data-type Feature added in cpp

Array: char string[3] = "xyz"; // valid in c

char string[4] ="xyz"; // valid in cpp *have to add null character \0

Pointer: Two new feature added in pointer

1. char * const ptr1 = "GOOG"; // constant pointer
2. int const *ptr = &m; // pointer to constant

```
const int size = 10;
char name[size];      // valid in cpp.
```

const size = 10; // replacement of #define
const int size = 10; // its eqal to const size = 10;
in C sizeof('X') = = sizeof(int)
but in cpp sizeof('x') == sizeof(char)

can declare define variable anywhere in program.

Reference Variable

alias name to existing variable.

```
data-type & reference-name =  variable-names
```

Limitation of Reference Variable

1. Must initialize
2. cant refer to constant
3. cant not re-refer . means once reference variable created it cant refer to another variable

```
int main()
{
	int x = 10, y = 20;
//	int &r;			// error "r declare as refrence but not initialized"
//	int &r = 10;		//error non const ref type to int &
//	int &r = (const int)10;	//error non const ref type to int &
	int &r = x;		// ok
//	int &r = y;		// error redeclaration of 'int &r'
	cout << "x = " << x << endl;
	cout << "y = " << y << endl;
	cout << "r = " << r << endl;
	return 0;
}
```

Scope Resolution Operator

```
:: variable name
```

its **always refer to global variable**. Highest priority to global than local blocks variable:

example:

```
int m = 10;
/* Main Program */
int main()
{
	int m = 20;
	{
		int m = 30;
		cout << "inside block m = "<< m << endl;
		cout << "inside block ::m = "<< ::m << endl;
	}
	cout << "main block m = "<< m << endl;
	cout << "main block ::m = "<< ::m << endl;
	return 0;
}
```

Output

```
inside block m = 30
inside block ::m = 10
main block m = 20
main block ::m = 10
```

5-Nov-19

Free Store Operator :

new and **delete** two unary operator that perform task of allocation of dynamic memory and freeing memory in better and easy way

```
pointer-variable = new data-type;
pointer-variable = new data-type(value);
pointer-variable = new data-type[size];
delete pointer-variable;
```

Example:

```
int main()
{
   int *i = new int;                    // allocating dynamic memory
   float *f = new float(3.5);           // allocating dynamic memory with initialize
   char *ch = new char[20];             // allocating dynamic memory to array
   strcpy(ch,"hello");
   cout << "i = " << *i << " f = " << *f << " char = " << ch << endl;
   delete i;                            // free memory
   delete f;                            // free memory
   delete [] ch;                        // free array memory
}
```

output

```
i = 0 f = 3.5 char = hello
```

memory leak check with valgrind tool

```
valgrind --leak-check=yes ./a.out
```

output

```
i = 0 f = 3.5 char = hello
==664==
==664== HEAP SUMMARY:
==664==          in use at exit: 0 bytes in 0 blocks
==664==      total heap usage: 5 allocs, 5 frees, 76,828 bytes allocated
==664==
==664== All heap blocks were freed -- no leaks are possible
```

new operator advantage over **malloc()**

1. no need to sizeof operator
2. no need of typecast
3. possible to initialize variable while creating memory space
4. it can be overloaded

Chapter 4 Function in cpp

Function overloading:

same function name to create function that perform a variety of different tasks known as function polymorphism in oop. **same function** name with **different argument** lists.

```
int add(int,int);
//float add(int,int);     // error function overloading is not differ by return type
float add(float,float);
float add(int,float);
float add(float,int);
float add(int,int,float);
```

```
int main(){
   cout << add(10,20) <<endl
        << add(1.1f,1.2f) <<endl
        << add(10,1.1f) <<endl
        << add(1.1f,20) <<endl
        << add(10,20,1.1f) <<endl;
   return 0;
}
int add(int ix,int iy){
return (ix+iy);
}
float add(float fx,float fy){
return (fx+fy);
}
float add(int ix,float fy){
return (ix+fy);
}
float add(float fx,int iy){
return (fx+iy);
}
float add(int ix,int iy,float fz){
return (ix+iy+fz);
}
```

Function Overloading Notes

1. function overload should be done with caution.
2. we should not overload unrelated functions and should reserve function overloading for function that perform closely related tasks.
3. default argument may be used instead of overloading that reduce function to be defined.

Chapter 5 Class and Object

most important feature of cpp is class. extension of idea of structure used in c.

c structure	**cpp structure**
cannot add two structure element directly c = a+b;	can add structure elements
no data hiding . element of structure can be access by any function and it can be modify by it.	data hiding is possible.
no functions	can add functions
struct keyword must use while in the declaration of structure variable	struct keywotd can be omited while in the declaration of structure variable

only difference between structure in cpp and class is by default **member of structure are public** and **member of class are private**

example :

```
class person
{
   private: int pwd;
   public:int id;
         string name;
         int get_data();
         int set_pwd(int);
};
```

```
int person :: get_data()
{
   cout << "name : " <<name << endl
        << "id :" << id << endl
        << "pwd :" << pwd << endl;
}
int person :: set_pwd(int a_pwd)
{
   pwd = a_pwd;
}
int main()
{
   persion praful;
   praful.id = 2154;
   praful.name = "Praful Vanker";
   // praful.pwd  = 1234; // error private within context
   praful.set_pwd(1234);
   praful.get_data();
   return 0;
}
```

note: we can define member function inside of class , it is treated as inline function so all limitation of inline function apply to member function. so it is re commanded if member function body is small we can define inside of class.

Memory Allocation to Class object and member function

memory space for object is allocated when they are declared and not when class is specified. statement is partly true.

member function memory allocation only once when they are defined in class.

member variable memory allocation each of time when new object is created. it is essential because it holds different data.

Static Member Variable :

1. It is initialize to zero when first object of class is created. No other initialization is permitted.
2. Only one copy of that member is created for entire class and shared with all objects of class. no matter how many objects are created.
3. its scope is within class only but lifetime is entire program.
4. static member variable must defined outside of class. cause it store separately rather than part of object. so static member is class variable.
5. static member variable can be private public

example:

```
class Author{
   private:
        int x;
        static int private_static_z;
   public:
        static int z;
        void get_data();
        void set_data(int,int);
};
int Author :: z;                    // static member defination
int Author :: private_static_z;       // static member defination
void Author ::get_data()             // member function defination
{
   cout << " static member z = " << z <<endl;
   cout << " private static member z = " << private_static_z <<endl;
```

```
   cout << " member x = " << x << endl;
}
void Author ::set_data(int a_z,int a_x)
{
   private_static_z = a_z;
   x = a_x;
}
int main()
{
   Author a,b;
   a.set_data(10,100);        // set data and initialze z
   a.get_data();
   a.z = 500;                 // static member can access with object
   a.get_data();
   Author:: z = 999;          // public static member can access outside of class
   cout << "public_static_z = " << Author:: z << endl;
   // Author:: private_static_z = 1000;   // error private can not access outside of class
   b.set_data(20,200);
   b.get_data();
   return 0;
}
```

output:

```
static member z = 0
  private static member z = 10
  member x = 100
  static member z = 500
  private static member z = 10
  member x = 100
public_static_z = 999
  static member z = 999
  private static member z = 20
  member x = 200
```

static member function :

1. A static member function can have access to only static members declared in same class.
2. A static member function can called using the class name (instead of objects)
   class-name :: function-name;
3. static member function deal with static member variable only. means not static variable can not used in static member function.

example:

```
class student{
   public:
         int id;
         static int count;
         static void show_count()
         {
            cout << "count = " << count << endl;    // static member
          // cout << "id = " << id << endl;         // error not static member
         }
         void show_id()
         {
            cout << "id = " << id << endl;
         }
};
int student:: count;
int main()
```

```
{
   student s;
   s.id = 0;
   s.count = 0;
   s.show_count();
   s.show_id();
   return 0;
}
```

Friend Function

6 Nov-19

Non Member Function can not access private data of class. there is any situation where classes want to share there function there friend function works. to make any function friend of a class simply declare this function as a friend of a class
Friend Function characteristics

1. any function can declare as friend of a class
2. friend function can not call with object it can call as normal function
3. we should pass class object in argument to access the data.
4. it can be declare in public or private no meaning change
5. used in operator overloading

example:

```
class A;                              // class declare
class B;
int sum(class A, class B);            // function declare
class A                               // class define
{
   int x;
   friend int sum(class A, class B);  // friend function declare

public:
   void get_data()
   {
      cout << "x = " << x << endl;
   }
   void set_data(int ax)
   {
      x = ax;
   }
};
class B
{
   int y;
   friend int sum(class A, class B);  // friend function declared

public:
   void get_data()
   {
      cout << "y = " << y << endl;
   }
   void set_data(int ay)
   {
      y = ay;
   }
};
```

```
int sum(A ax, B by)
{
   return (ax.x + by.y);
}
int main()
{
   A a;
   B b;
   a.set_data(10);
   b.set_data(20);
   cout << "sum of a + b = " << sum(a, b) << endl;              // calling friend function
}
```

output

```
sum of a + b = 30
```

const MEMBER FUNCTION :

if member function does not alter any member variable data in the class. then we may declare it is a const member function

```
void get_data() const;
```

the **qualifier** const is appending to the function prototype (in both declaration and definition).

Local class:

classes can be defined inside of function or block such classes are called local classes.

note

1. local class can use global variable (declare above function) and static variable declared inside the function but **can not use automatic local variables.**
2. global variable should used with scope operator(::).
3. local class can not have static data member
4. member function must defined inside of class only.
5. enclosed function can not access private data member of local class it can be achieved if function is friend.

example:

```
int global_z = 10;
int main()
{
   int local_x = 20;
   static int static_y = 30;
   class local_class
   {
      int private_x;
      public:
         int public_x;
         void set_data(int pri_a,int pub_b)
         {
            private_x = pri_a;
            public_x = pub_b;
         }
         void get_data()
         {
            cout << "private x  =" << private_x << endl
                 << "public_x   =" << public_x << endl
                 << "staic_y    =" << static_y << endl     // class can access static member of function
                 << "::global_z =" << ::global_z << endl;   // class can access global member with scope oper
ator
         // cout << "local_x = " << local_x << endl;
/* error  use of local variable with automatic storage from containing function */
```

```
        }
  };
  local_class obj;
  obj.set_data(100,200);
  obj.get_data();
}
```

output:

```
private x    =100
public_x     =200
staic_y      =30
::global_z =10
```

Chapter 6 Constructor and Destructors

cpp provide special member function called constructor which enable an object to initialize itself when created. this known as automatic initialization of object. it also provide another member function called destructor that destroy the object when they no longer require.

constructor : constructor is a special member function whose task to **initialize the object of its class**. it is **special because its name is the same as class name.** its invoke automatic whenever object is created.

notes

1. should declare in **public** section
2. it invoked automatic when the object created
3. do not have **return** type (not even void)
4. can not inherited, though derived class can call the base class constructor.
5. can have default argument.
6. can not be **virtual**
7. we can not refer address
8. may implicitly call **new** and **delete** when memory allocation required.
9. if we declare implicitly constructor it is mandatory to initialize all member variable.

Type of Constructor

1. Default constructor
2. Parameterized constructor
3. copy constructor
4. dynamic constructor

Default constructor : A constructor that accept no argument called default constructor. it is implicitly added by compiler if not added explicitly that's why its called default constructor.

```
class-name :: class-name(void)
```

Parameterized constructor:

the constructor that can take arguments are called parameterized constructor.

```
class-name :: class-name (arguments…);
```

Copy constructor : constructor can accept a reference of its own class as a parameter that type of constructor called as copy constructor.

```
class-name :: class-name(class-name &);
```

notes:

it is important to distinguish between default constructor **A::A()** and default argument constructor **A::A(int =0).** the default argument constructor can be called with either one argument or no argument. when called with no argument, its become a default constructor.when both these forms are used in class it **causes ambiguity** .

Default Constructor example:

```
class A{
   int x;
   public:
      // A(){}                  // default constructor that may compiler provide if we didn't provide
                                    explicitly
      A():x(0){};               // default constructor explicitly provided that initialize x with 0
      void get_data()
      {
         cout << "x = " << x << endl;
      }
};
int main()
{
   A obj;
   obj.get_data();
   return 0;
}
```

output:

```
x = 0
```

Parameterized constructor example:

```
class A
{
   int x;
   public:
      A(int a):x(a){};                    // parameterized constructor
      void get_data()
      {
         cout << "x = " << x << endl;
      }
};
int main()
{
   A a(20);
   a.get_data();
}
```

Copy constructor example

```
class A{
   int x;
   public:
      A():x(10){cout << "default constructor called" << endl;}
      A(A &a){
         cout << "copy constructor called " << endl;
         x = a.x;
      }
      void set_data(int a)
      {
         x = a;
      }
      void get_data()
      {
         cout << "x = " << x << endl;
      }
};
int main()
{
```

```
   A a;              // a object created
   a.get_data();
   a.set_data(100);
   a.get_data();
   A b(a);           // b object created with help of copy constructor
   b.get_data();
}
```

output:

```
default constructor called
x = 10
x = 100
copy constructor called
x = 100
```

Destructor : destroy the object that have been created by a constructor.
distructor name same as class name only with tilde symbol **~**.
~A(){}

- ✓ never take any argument nor does it return any value.
- ✓ it invoke illicitly by compiler upon exit from program

example:

```
class A{
   int x;
   public:
      A():x(10){
         cout << "construtor called" << endl;
      }
      ~A(){
         cout << "distructor called" << endl;
      }
};
int main()
{
   cout << "a object created " << endl;
   A a;
   {
      cout << "b object created" << endl;
      A b;
   }
   cout << "b object scope done" << endl;
}
```

output:

```
a object created
construtor called
b object created
construtor called
distructor called
b object scope done
distructor called
```

Chapter 7 Operator Overloading and Type Conversions

- ✓ mechanism to give special meaning to operator known as operator overloading
- ✓ built in data-type we can apply any operator and do operation but if we want to use operator on user-defined data-type like class we need to overload the operator.

following **operator can not be overloaded**

1. Class member access operator (**..,.***).

2. scope resolution operator (::).
3. Size operator (**sizeof**).
4. Conditional operator (**?:**).

we can not overload this operator may be attributed to the fact that these operaotr takes names (class name) as their operand instead of value.
some of grammatical rules govern its use such as number of operands.

7-Nov-19

unary (-) operator overloading:
overloading minus sign operator
if class A object a member **x = 10** after apply **-a** it will become **x = -10**;
example:

```
class A{
   int x;
   public:
      A():x(10){}          // default constructor
      void get_data()
      {
         cout << " x = " << x << endl;
      }
      void operator -()   // unary - operator overloading (it just change sign of data-member)
      {
         x = -x;
      }
};
int main()
{
   A a;
   a.get_data();
   -a;
   a.get_data();
}
```

output:

```
 x = 10
 x = -10
```

Overloading + operator:
with overload the + operator we can add tho complex data-type like C = A+B;
example:

```
class A{
   int x;
   public:
   A():x(10){}
   A operator+(A a)
   {
      A temp;
      temp.x = x + a.x;
      return temp;
   }
   void get_data()
   {
      cout << "x = " << x << endl;
   }
};
int main()
{
   A a,b,c;
```

```
    a.get_data();
    b.get_data();
    c = a + b;     // c = a.operator+(b) usual function call syntax
    c.get_data();
}
```

output:

```
x = 10
x = 10
x = 20
```

some notes on operator +() overloading function

1. it received only one **A** type argument explicitly
2. it returns a **A** type value
3. it is a member function of **A**.

c = a + b; // this invoke operator + () function

we know that member function can called only by object of the same class. here the **a** object takes the responsibility of invoking the function and **b** plays the role of the argument that is passed in function.

so now syntax will equivalent to **c = a.operator+(b);**

without temp variable

```
A(int a):x(a){}     // parameterized constructor
A operator+(A a)
{
        return A(x + a.x); // due to parameterized constructor we can avoid use of temp variable
}
```

there is a **limitation of member function** while overloading operator + ().

a = b + 2;

where **a** and **b** are object of same class this will run fine **but**

a = 2 + b;

will not work, this is because left hand operand which is responsible for invoking member-function should be an object of the same class. with **friend function** we can allow both approach. **how ?**

no need of object to invoke friend function but can be passed as an argument.

```
    friend A operator+(A a,A b);          // define as friend in class
```

```
A operator+(A a, A b)                     // as friend function operator + overloading
{
    return A(a.x + b.x);
}
```

We **can not overload** following operator using **friend function**

1. **=** Assignment.
2. **()** function call.
3. **[]** Subscript.
4. **->** class member access

ALL Opeator overloaded single example:

```
class A
{
    int x;
    int a[5];
public:
    A(int obj) : x(obj) {
        for(int i=0 ; i< 5; i++)
        {
            a[i] = 0;
        }
    } // parameterized constructor
```

```
   friend A operator+(A a, A b);                // overload + operator
   friend ostream &operator<<(ostream &out, A &obj)
   {
      cout << "x = " << obj.x << endl;
      return out;
   }
   void operator-()                          // overload unary - operator
   {
      x = -x;
   }
   void operator = (A obj)                    // overload = assignment
   {
      x = obj.x;
   }
   int operator[] (int index)                 // overload [] subscipt
   {
      return a[index];
   }
   void operator()(int obj)                   // oberload function()
   {
      x = obj;
   }
   void get_data()
   {
      cout << "x = " << x << endl;
   }
   void set_array(int *p)
   {
      for(int i =0; i<5 ; i++)
      {
         a[i] = p[i];
      }
   }
};
A operator+(A a, A b)
{
   return A(a.x + b.x);
}
int main()
{
   A a(10), b(20), c(0);
   a(50);                                  // using function operator overload
   a.get_data();
   int x[5] = {1,2,3,4,5};
   a.set_array(x);
   for(int i = 0; i < 5 ; i++)
   {
      cout << "a[" << i << "] = " << a[i] << endl;    // using object with subscipt operator
   }
   a.get_data();
   b.get_data();
   c = a + b;                                // c = a.operator+(b) usual function call syntax
   c.get_data();
   cout << a << b << c << endl;                  // overload insersion operaotr using friend function
}
```

Chapter 8 Inheritance : Extending Classes

Nov. 8, 19

the mechanism to deriving a new class from an old one is called **inheritance**. or we can say acquiring the property of one class to another class. new class called **derived class** and the exiting class is known as **base class**

example:

```
class person                                        // Base Class
{
   string dob;
   string name;
public:
   void get_person_data()
   {
      cout << "name = " << name << endl
          << "dob  = " << dob << endl;
   }
   void set_person_data(string s_dob, string s_name)
   {
      dob = s_dob;
      name = s_name;
   }
};
class employee : public person                      // Derived Class
{
   int id;
public:
   void set_employee_data(int a_id)
   {
      id = a_id;
   }
   void get_employee_data()
   {
      cout << "employee id = " << id << endl;
   }
};
int main()
{
   cout << "person Base Class" << endl;
   person praful;
   praful.set_person_data("16/10/1990", "praful");
   praful.get_person_data();
   cout << "employee Derived class" << endl;
   employee e_praful;
   e_praful.set_person_data("16/10/1990", "Praful");
   e_praful.set_employee_data(2154);
   e_praful.get_person_data();
   e_praful.get_employee_data();
   return 0;
}
```

above example is on single inheritance. person is a base class where data member are private and member function are public.when we derived a employee class from person the private member is derived as private member and member function as public as it is.

private section of base class can not be public even if we derived it public
public section of base class can be be derived as private.

Base Class Visibility	Derived Public	Derived Protected	Derived Private
public	public	protected	private
protected	protected	protected	private
private	cant inherited	cant inherited	cant inherited

cant inherited means can not access in derived class.

example:

```
class A
{
   int x;
   public:
   int y;
   A():x(10),y(20){
      cout << "Class A Constructor" << endl;
      cout << "x = " << x << " y = " << y << endl;
   }
};
class B : public A
{
   int z;
   public:
   B():z(40){
      cout << "class B Constructor" << endl;
      y = 30;
      cout << "y = " << y << " z = " << z << endl;
   }
};
int main()
{
   A a;
   B b;
   return 0;
}
```

output:

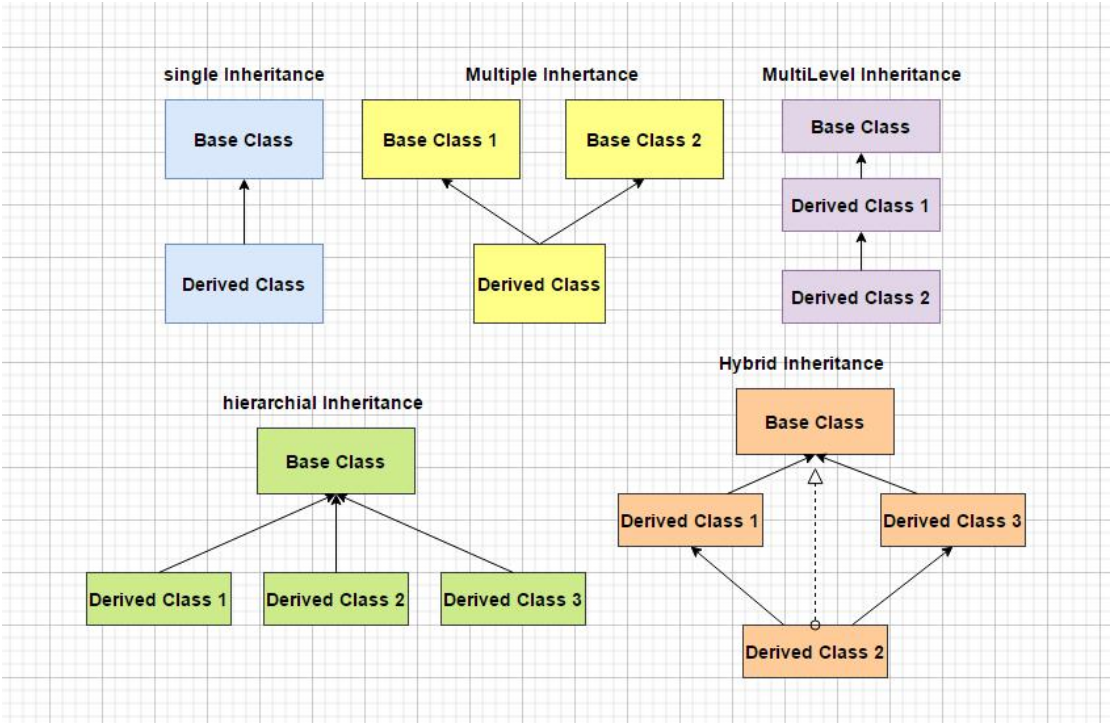
```
Class A Constructor
x = 10 y = 20
Class A Constructor
x = 10 y = 20
class B Constructor
y = 30 z = 40
```

note:

1. constructor of class B is implicitly call constructor of class A.
2. once constructor called finished we can access the public member variable of base class in derived class

Type of inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance
6. Multipath inheritance



,

- ✓ Derivation of class from only one base class is **Single inheritance**
- ✓ Derivation of class from another derived class is **Multilevel inheritance**
- ✓ Derivation of class from several base class is **Multiple inheritance**.
- ✓ Derivation of several class from single class is **Hierarchical inheritance.**
- ✓ Derivation of class from other derived class which is from same base class is **multipath inheritance.**

virtual base class:

Consider situation where all three type of inheritance apply namely **multilevel, multiple and hierarchical**. in this case child would have duplicate sets of member from grand base class this introduces ambiguity and should be avoided.

When class is made a virtual base class it take care that **only one copy of that class is inherited**, regardless of how many inheritance path exists between the virtual base class and derived classes.

11-Nov-19

Abstract Class:

class that is not use to create objects. only design to be used as a base class.

class can only be consider as an abstract class if it has **at least one pure virtual function**.

An abstract class can not be initiated. however with abstract class we can do

Have data member, have non virtual member function, provide implementation for pure virtual function. do everything except instantiate it.

example:

```
class Vehicle                   // Abstract Class
{
   private:
      int id;
   public:
      virtual void show_id() = 0; // pure virtual fun
};
class Lmw: public Vehicle         // Derived Class
{
   public:
      void show_id(){}          // have to override
```

```
};
```

Chapter 9 Virtual Function and Polymorphism

virtual function: to archive run time polymorphism.

1. when it is known what class objects are under consideration, the appropriate version of the function invoked.
2. when we use same function name in both the base class and derived class, the function in base class is declare as virtual using virtual keyword preceding its normal declaration.
3. When function made virtual, Cpp determined which function to use at run time based on **type of object pointed to by base pointer , rather than type of pointer**

example:

```
class base
{
   int x;
   public:
      base():x(10){}
      virtual void v_show()
      {
         cout << "base x = " << x << endl;
      }
      void show()
      {
         cout << "base x = " << x << endl;
      }
};
class derived: public base
{
   int y;
   public:
      derived():y(20){}
      void v_show()
      {
         cout << "Derived y = " << y << endl;
      }
      void show()
      {
         cout << "Derived y = " << y << endl;
      }
};
int main()
{
   base b;
   derived d;
   base *bptr;
   cout << "pointing to Base calling v_show" << endl;
   bptr = &b;
   bptr->v_show();
   cout << "pointing to Derived calling v_show" << endl;
   bptr = &d;
   bptr->v_show();
   cout << "pointing to Base calling show" << endl;
   bptr = &b;
   bptr->show();
   cout << "pointing to Derived calling show" << endl;
```

```
    bptr = &d;
    bptr->show();
    return 0;
}
```

output:

```
pointing to Base calling v_show
base x = 10
pointing to Derived calling v_show
Derived y = 20
pointing to Base calling show
base x = 10
pointing to Derived calling show
base x = 10
```

run time polymorphism archived only when virtual function accessed by base class pointer.

Rules of Virtual Function

1. virtual function must be member of class
2. they can not be static
3. they are access with object pointer
4. virtual function can be a friend of another class
5. virtual function in base class must be defined, even it may not use.
6. derived class must be use same name and argument of base class virtual fun
7. base class pointer can be point to any derived class object but reverse in not possible
8. not be necessarily redefined in the derived class. in such case will invoke the base function.

Pure virtual function

the function inside base class not perform any task but may be used in derived class. this function are called “do-nothing” function may defined as follow:

```
virtual void v_show() = 0;
```

such function called pure virtual function. A pure virtual function is a function declared in base class that has no definition to base class.

derived class must to either **redefined the function or re-declare as virtual function**.

Chapter 10 Managing Console I/O Operations

Stream is a sequence of byte. it act as either as a source from which the input data can be obtained or as a destination to which the output data can be sent.

source stream that provide data to program called **input stream.**

destination stream that receives output from the program called **output stream.**

put(): ostream member function used to opration on single character.

```
cout.put('x');
```

get(): istream member function used to operation on single character.

```
cin.get(c);
```

getline():the getline function reads a whole line of text data that ends with newline character.

```
cin.getline(line, size);
```

write(): displays entire line

```
cout.write(line, size)
```

width(): set width of a field necessary for output of an item.

```
cout.width(5);                  // 5 byte space
```

get() put() example

```
char c;
cin.get(c);
while (c != '\n')
{
        cout.put(c);
        cin.get(c);
```

```
}
```

width() example

```
enum{
   EMPLOYEE_NAME_WIDTH = 10,
   EMPLOYEE_ID_WIDTH = 5
};
class employee
{
   public:
      string employee_name[3];
      int employee_id[3];
      void set_data(string ptr[3],int *id)
      {
         for(int i = 0; i < 3 ; i++)
         {
            employee_id[i] = id[i];
            employee_name[i] = ptr[i];
         }
      }
      void get_data()
      {
         for(int i = 0; i < 3 ; i++)
         {
            cout.width(EMPLOYEE_ID_WIDTH);
            cout << employee_id[i] ;
            cout.width(EMPLOYEE_NAME_WIDTH);
            cout << employee_name[i] << endl;
         }
      }
};
int main()
{
   employee e;
   string s_e[3] = {"tushar","praful","rakesh"};
   int i_e[3] = {1234,5678,9101};
   e.set_data(s_e,i_e);
   cout.width(EMPLOYEE_ID_WIDTH);
   cout<<"Width";
   cout.width(EMPLOYEE_NAME_WIDTH);
   cout<<"Name"<<endl;
   e.get_data();
}
```

output

```
Width         Name
 1234       tushar
 5678       praful
 9101       rakesh
```

Chapter 11 Working With Files

C++ contains set of class that defined file handling methods. this include **ifstream**, **ofstream** and **fstream**.

Opening closing file

for opening file we must first creat a file stream and then link with filename.

1. using constructor of class
2. using member function open() of the class

first method used when we use only one file in thre stream. second method we can use if we want to manage multiple files using one stream.

```
obj.open("file-path",mode)
```

opening file in different modes

mode	Meaning
ios::app	appending
ios::ate	go to end of file on opening
ios::binary	binary file
ios::in	read only
ios::nocreate	open fail if file not exist
ios::noreplace	open fail if file exist
ios::out	write only
ios::trunc	delete content and open

opening file using constructor:

1. Create a file stream object to mange the stream using appropriate class. class **ofstream** used to create **output stream.** and class **ifstream** used to create **input stream.**
2. Initialize the file object with desired file-name.

```
ofstream logfile("logfile");      // logfile is file name
```

this create logfile as an ofstream object that manages output stream.

similarly the following statement declares infile as **ifstream** object and attaches it to the file data for reading.

```
ifstream infile("data");              // data is filename
```

for **closing**

```
infile.close()
logfile.close()
```

disconnect file from stream.

example

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   ofstream w_f;            // output stream object
   ifstream r_f;           // input stream object
   char data[20] = {0};
   char data1[20] = {0};
   w_f.open("file");         // connect file to output stream
   cin.getline(data,20);
   w_f << data;
   w_f.close();             // disconnect the file to input stream
   r_f.open("file");         // connect the file to input stream
   r_f.getline(data1,20);
   cout.write(data1,20);
   r_f.close();             // disconnect the file from input stream
   return 0;
}
```

output:

```
Hello World <press Enter>
Hello World
```

eof(): end of file is a member function of **ios** class. it return non-zero value if EOF condition is encounter.

```
if(fin1.eof() != 0)
{
      exit (1);
}
```

above statement terminates the program on reaching end of file.

File Pointers

seekg() Moves get pointer (input) to a specified location.
seekp() Moves put pointer (output) to specified location
tellg() Give current position of get pointer.
tellp() Gives the current position of the put pointer.
ios::beg start of the file.
ios::cur current position of file.
ios::end end of file

Seek Call	**Action**
fin.seekg(o, ios::beg);	Go to start
fin.seekg(o , ios::cur);	Stay at current position
fin.seekg(o, ios::end);	Go to end of file
fin.seekg(m , ios::beg);	Go to (m + 1)th byte in the file
fin.seekg(m, ios::cur);	Go forward by m byte from current position
fin.seekg(-m, ios::cur);	Go Backward m byte from current position
fin.seekg(-m, ios::end);	Go to Backward by m bytes from end of file

error Handling in file operation

```
w_f.open("file"); // connect file to output stream
  if (!w_f.fail())  // do file operation if it successfully open
  {
     w_f << data;
     w_f.close();
  }
  r_f.open("file"); // connect the file to input stream
  if (!r_f.fail())
  {
     r_f.getline(data1, 20);
     cout.write(data1, 20);
     r_f.close(); // disconnect the file from input stream
  }
```

Chapter 12 Templates

12 Nov-19

Concept that enable us to **define generic class and function** support generic programming.
for example a class template for an **array class** would enable us to create array of various data-type such as int,char, float.
we can define template for function **mul()**, that would help us to create various version of mul() for multiplying int, float and double type value.

General format of class template:

```
template<class T>
class class-name
{
      // .....
      // class member
      // class anonymous type T
      // .....
```

```
};
```

example:

```
template<class T>
class array
{
   T* a;
   int size;
   public:
      array(int m)
      {
         a = new T[size = m];
         for(int i = 0 ; i<size; i++)
         {
            a[i] = 0.5;
         }
      }
};
int main()
{
   array<int> obj_int_array(10);
   array<float> obj_float_array(10);
return (0);
}
```

A class created form a class template is called **template class.**

classname<type> objectname(arglist);

this process of creating class from a class template is called **instantiation**.

class template with multiple parameter

```
template<class T1, class T2, ...>
class class-name
{// body of class
};
```

example:

```
template <class T1, class T2>
class test
{
   T1 a;
   T2 b;
   public:
      test(T1 x,T2 y)
      {
         a = x;
         b = y;
      }
      void get_data()
      {
         cout << "a = " << a << " b = " << b << endl;
      }
};
int main()
{
   test <int,float> i_f(10,3.5);
   test <float,int> f_i(4.8,20);
   i_f.get_data();
   f_i.get_data();
}
```

output:

```
a = 10 b = 3.5
a = 4.8 b = 20
```

Default data-type of class

```
template <class T1 = int, class T2 = int>
```

Function Template:

the function template syntax is similar to that of the class template except that er are defining function instead of classes. we must use template parameter T as and when necessary in the function body and in its arguments.

we can invoke function like ordinary function but it can operate on any type of data. like below example we can use **swap()** function to swap any type of data-type data.

example

```
template<class T>
void swap(T &x, T &y)
{
      T temp = x;
      x = y;
      y = temp;
}
```

bubble sort template function example:

```
template <class T>        // bubble sort template function can sort any type
void bubble(T a[], int s)
{
  for(int i = 0; i < s-1 ; i++)
  {
    for(int j = s-1; i<j; j--)
    {
      if(a[j] < a[j-1])
      {
        T temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
      }
    }
  }
}
template <class X>        // print function can print any type of array
void print(X *ptr,int m)
{
  cout << "data :";
  for(int i =0; i < m;i++)
  {
    cout << ptr[i];
  }
  cout << endl;
}
int main()
{
  int i[5] = {1,3,2,6,0};
  char c[9] = "adcbADCB";
  cout << "before Swap ..."<<endl;
  print(i,5);
```

```
   print(c,9);
   bubble(i,5);
   bubble(c,9);
   cout << "After Swap ..."<<endl;
   print(i,5);
   print(c,10);
}
```

Chapter 13 Exception Handling

We know that that is very rare that program works correctly first time. it might be a bug or error. two type of most common bug are **logical and syntactical errors**.

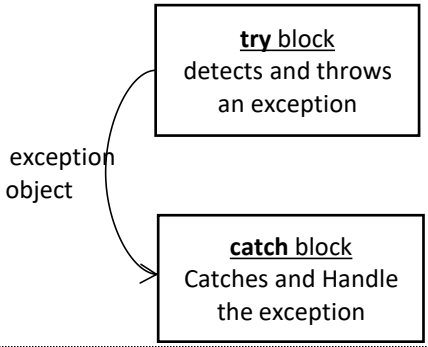
C++ provide built-in language features to detect and handle exception which are basically run time error.

exception handling is not part of C++, it is a new feature that added by **ANSI C++**. today almost all compiler support this feature. there is two type of exceptions.

synchronous exception: error such as "out-of-range index" and "over-flow

asynchronous exception: error that beyond control of program (keyboard interrupt)

1. Find the problem (**hit** exception)
2. Inform that an error has occurred (**Throw** the exception).
3. Received the error information (**catch** the exception).
4. Take corrective actions (**handle** the exception).




```
try
{
      …
      throw exception;          // detect and throw an exception
      …
}
catch(type arg)                 // catch the exception
{
      …
}
```

try us used to preface a block of statement which may generate exceptions.

throw is used to throw an exception if detected.

catch 'catches' the exception 'thrown' by the throw statement in try block.

exception are objects that used to transmit information about problem. if the type of object matches the **arg** type in **catch** statement, then catch block is executed for handling the exception.

if do not match program can be aborted using **abort().**

example: Divide by Zero exception

```
int main()
{
   int a,b,c;
   cout << "enter values :" ;
```

```
   cin >> a >> b;
     try          /* Try block */
   {
      if(b == 0)  /* detecting exception */
      {
         throw(b);   /* throwing exception */
      }
      cout << "a/b = "<< (c = a/b) << endl;
   }
   catch(int m)    /* catching exception */
   {
      cout << "exception Divide by Zero\n";
   }
   return 0;
}
```

Catching all Exceptions

```
  catch(...) /* accept or catch all throw, that is not explicitly catch */
   {
      cout << "default exception...\n";
   }
```

Restrictions on Throw

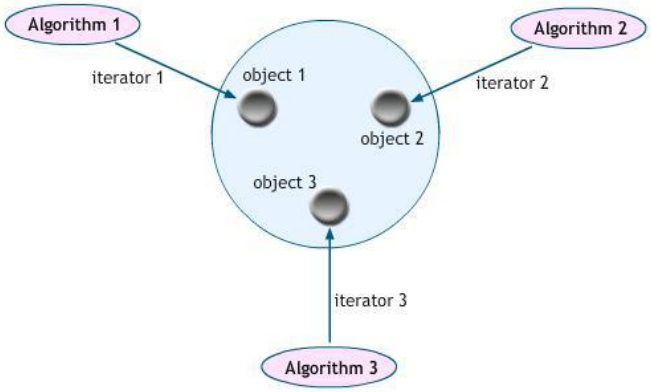
```
void test(int x) throw (int,double)
{
   try
   {
      if(x == 0)
         throw 'x';
      else
      {
         if(x == 1)
            throw x;
         else
            throw 1.1;
      }
   }
   cout << "end of test_fun()\n";
}
```

here test function will not throw any exception of int and double type.
note: A function can only be restricted in what types of exception it throw back to the try block that called it. the restriction applies only when throwing an exception out of the function(not within a function).

Chapter 14 Introduction to the Standard Template Library (S T L)

Alexander Stepanov anf meng lee of Hewlett-Packard developed a set of general purpose templatized classes (data-structure) and function(algorithm) that could be used as standard approach for storing and processing data. the collection of these generic classes and function called the Standard Template Library. no STL is part of ANSI standard c++ class library.

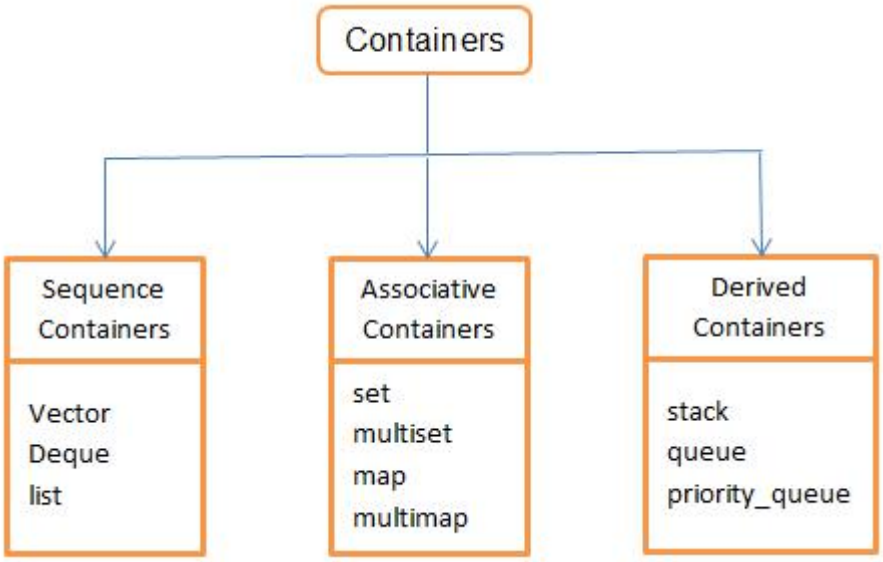
STL component are defined in the namespace **std.** there for we use usinge namespace directive.



Key Component of STL is **container, Algorithm, Iterators**.
A **container** is an object that actually store data. it is a way data in organized in memory. the STL **container** are implemented by template classes and therefore can be easily customized to hold different type of data.

An **algorithm** is a procedure that used to process the data in the containers. The STL includes many different kinds of algorithms to provide support to tasks such as **initializing, searching, copying merging** etc.

An Iterator is an object (like pointer) that points to an element in a container. We can use iterator to move through the contents of containers. iterators are handles just like pointer.



Sequence Containers:
Sequence containers store elements in linear sequence. like a line. each element is related to other elements by its position along the line. they all expand themselves to allow insertion of the elements and all of them support a number of operation on them.

Comparison of sequence containers

Container	**Random access**	**insertion/deletion in the middle**	**insertion or deletion at the end**
vector	Fast	Slow	Fast at Back
List	Slow	Fast	Fast at front
deque	Fast	Slow	Fast at both the ends

Associative Containers

1. Associative containers are designed to support direct access to element using keys. they are not sequential.
2. all container store data in **tree,** which facilitates fast searching, deletion, and insertion.
3. the main difference between a **set** and **multiset** is that multiset allows duplicate items while a set does not.
4. container **map** and **multimap** are used to store pairs of item. one call **key** and other called **value**.
5. Difference between map and multimap is that map allows only one key for given value to be store while multimap permits multiple keys.

Derived Containers

1. its also known as container adaptors.
2. stack, queue and priority queues can be created from different sequence containers.
3. the derived containers **do not support iterator.**
4. support two member function **pop() and push()** for implementing deletion and insertion.

Algorithms

1. Algorithms of each container provides functions for its basic operations
2. STL provide **60+** algorithms to support more extended or complex operation.
3. STL algorithms are not **member function** or **friend function** of containers. they are standalone template function
4. to access STL algorithms we must include **<algorithm>** in program.
5. algorithms categorized as **retrieve or nonmutating, mutating , sorting, set, relation.**

Iterators

1. iterator behave like pointers and used to access container elements.
2. Different types of iterator used with different type of containers.
3. forward iterator supports all operation of input and output iterators and also retains its position in the container.
4. A bidirectional iterator, while supporting all forward iterator operation. provide the ability to move in backward direction in the container.
5. A random access iterator combine the functionality of a bidirectional iterator with ability to jump to any arbitrary location.

Vectors:

1. vector is most widely used container.it store elements in contiguous memory locations and enable direct access to any element using subscript operator [].
2. vector can change its size dynamically and therefor allocation memory as needed run time.
3. vector class contain number of constructor to create vector object.
4. vector class contain several member function.

Using Vectors

```
#include <iostream>
#include <vector>
using namespace std;
void display(vector<int> &v)
{
   cout << "display container data :";
   for(int i=0; i < v.size(); i++)
   {
      cout << v[i] << " ";
   }
   cout << endl;
}
int main()
{
   vector<int> v;
   int x;
   cout << "pushing 5 element before size = " << v.size() << endl;
   for(int i=0; i<5; i++)
```

```
  {
     cin >> x;
     v.push_back(x);
  }
  cout << "After push 5 element size = " << v.size() << endl;
  display(v);
  cout << "add one element" << endl;
  v.push_back(6.6);
  cout << "After 6th element size = " << v.size() << endl;
  display(v);
  cout << "changing 1st element" <<endl;
  vector<int>::iterator itr = v.begin();  // iterator point to begin
  v.insert(itr,1,10);
  display(v);
  cout << "changing 4th element" <<endl;
  itr = itr +3;
  v.insert(itr,1,40);
  display(v);
  cout << "removing element 1st and 4th" <<endl;
  v.erase(v.begin(),v.begin()+4);
  cout << "After removing element size = " << v.size() << endl;
  display(v);
}
```

output:

```
pushing 5 element before size = 0
1 2 3 4 5
After push 5 element size = 5
display container data :1 2 3 4 5
add one element
After 6th element size = 6
display container data :1 2 3 4 5 6
changing 1st element
display container data :10 1 2 3 4 5 6
changing 4th element
display container data :10 1 2 40 3 4 5 6
removing element 1st and 4th
After removing element size = 4
display container data :3 4 5 6
```

14-Nov-19

List : list is container that support bi-directional, linear list and provide an efficient implementation list can access sequentially only.list class provide many manipulated function.

example:

```
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;
void display(list<int> &lst)
{
   list<int>::iterator p;
   for(p=lst.begin(); p!= lst.end(); ++p)
   {
      cout << "*p = " << *p << " ";
   }
   cout << endl;
}
int main()
```

```
{
   list<int> list1;
   list<int> list2(5);
   for(int i= 0; i<3;i++)
   {
      list1.push_back((rand()/100)%100);
   }
   list<int>::iterator p;
   for(p= list2.begin(); p != list2.end(); ++p)
   {
      *p = (rand()/100)%100;
   }
   cout << "list1" <<endl;
   display(list1);
   cout << "list2" <<endl;
   display(list2);
   cout << "Adding font and back list 1\n";
   display(list1);
   list1.push_back(1000);
   list1.push_front(2000);
   display(list1);
   cout << "Removing font and back of list 1\n";
   display(list2);
   list2.pop_back();
   list2.pop_front();
   display(list2);
   list<int> listA = list1;
   list<int> listB = list2;
   cout << "copting list and sorting \n";
   cout << "before sort\n";
   display(listA);
   display(listB);
   listA.sort();
   listB.sort();
   cout << "after sort....\n";
   display(listA);
   display(listB);
   cout << "merging list A AND B\n";
   listA.merge(listB);
   display(listA);
   cout << "reverse list\n";
   listA.reverse();
   display(listA);
   return 0;
}
```

output:

```
list1
*p = 93 *p = 8 *p = 27
list2
*p = 69 *p = 77 *p = 83 *p = 53 *p = 4
Adding font and back list 1
*p = 93 *p = 8 *p = 27
*p = 2000 *p = 93 *p = 8 *p = 27 *p = 1000
Removing font and back of list 1
*p = 69 *p = 77 *p = 83 *p = 53 *p = 4
*p = 77 *p = 83 *p = 53
```

```
copting list and sorting
before sort
*p = 2000 *p = 93 *p = 8 *p = 27 *p = 1000
*p = 77 *p = 83 *p = 53
after sort....
*p = 8 *p = 27 *p = 93 *p = 1000 *p = 2000
*p = 53 *p = 77 *p = 83
merging list A AND B
*p = 8 *p = 27 *p = 53 *p = 77 *p = 83 *p = 93 *p = 1000 *p = 2000
reverse list
*p = 2000 *p = 1000 *p = 93 *p = 83 *p = 77 *p = 53 *p = 27 *p = 8
```

if sorted list are merged the element are inserted in appropriate locations and there for the merged list also sorted one.

maps:

A map is sequential of (**key , value)** pairs where a single value is associated with each **unique key**. key can not change. and **key always in sorted order**.

KEY	**VALUE**
key 1	value 1
key n	value n

A **map** is called associated array. key use subscript operator

```
phone[“praful”] = 9408492091;
```

phone is object of map. “praful” is key “9408492091” is value.

example using map:

```
#include<iostream>
#include<map>
#include<string>
using namespace std;
typedef map<string,int> phoneMap;
int main()
{
   string name;
   int number;
   phoneMap phone;
   cout << "enter three name and number" << endl;
   for(int i=0; i<3; i++)
   {
      cin >> name >> number;
      phone[name]=number;
   }
   phone["vvdn"]=2151;
   cout << "size of phonemap : " << phone.size() << endl;
   cout << "phone data" <<endl;
   phoneMap::iterator p;
   for(p = phone.begin(); p != phone.end(); ++p)
   {
      cout <<"name : "<<(*p).first <<"\tnumber : "<<(*p).second<<endl;
   }
   return 0;
}
```

output:

```
enter three name and number
praful 1
rahul 2
rakes 3
size of phonemap : 4
```

```
phone data
name : praful      number : 1
name : rahul       number : 2
name : rakes       number : 3
name : vvdn         number : 2151
```

array:

- linear collection of similar elements. static array of generic type.
- some important member function for operation on array like **at, [] operator, front(), back(), fill(), swap(), size(), begin(), end().**

example:

```
#include<array>
using namespace std;
int main()
{
   int a[10] = {8 , 3, 5, 6, 2, 1, 9, 4, 7, 0};
   cout << "a[0] = " << a[0] << endl;
   cout << "a[10] = " << a[10] << endl;          // no error while compiling runnig

   array <int,10> arr_int_obj = {8 , 3, 5, 6, 2, 1, 9, 4, 7, 0};
   cout << "arr_int_obj[0] = "<< arr_int_obj[0]<<endl;
   cout << "arr_int_obj[10] = "<< arr_int_obj[10]<<endl;   // no error while compiling and running
   cout << "arr_int_obj.at(1) = "<< arr_int_obj.at(1)<<endl;
   // cout << "arr_int_obj.at(10) = "<< arr_int_obj.at(10)<<endl; // no error while compiling but run ti
me error.
   /*throwing an instance of 'std::out_of_range*/
}
```

pair : pair is template class in STL but not container. it pair two data-type. object is initialize by **make_pair()**.

example:

```
#include<iostream>
using namespace std;
class test
{
   int id;
   public:
      void get_data()
      {
         cout << "id : " << id << endl;
      }
      void set_data(int a)
      {
         id=a;
      }
};
void print_pair(pair<string,int> p)
{
   cout << "string : " << p.first << " int : " << p.second << endl;
}
void print_pair(pair<int,test> p)
{
   cout << "int : " << p.first <<  endl;
   p.second.get_data();
}
int main()
```

```
{
   pair <string,int> e1,e2,e3;
   e1 = make_pair("praful",2154);
   e2 = make_pair("tushar",2156);
   e3 = make_pair("rakesh",1954);
   print_pair(e1);
   print_pair(e2);
   print_pair(e3);
   test t1;                    // user define class pair
   t1.set_data(100);
   pair <int,test> pt;
   pt = make_pair(200,t1);
   print_pair(pt);
}
```

tuple: we can pair multiple data type in one object. object is initialize by **make_tuple()**.
example:

```
#include<tuple>
using namespace std;
int main()
{
   tuple<string,int,float> e1;
   e1 = make_tuple("praful",2151,42.70);
   cout << get<0>(e1) << endl;
   cout << get<1>(e1) << endl;
   cout << get<2>(e1) << endl;
return 0;
}
```

chapter 15 Strings
string is sequence of characters. it is not built-in type. c++ provide class called **string.**although string is not consider as part of **STL.** for using the string class we must include <string>. it include many character, member function and operators.
example:

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

void string_property(string &s)
{
   cout << "-----------------------------------------------\n";
   cout << "string\t:" << s << endl;
   cout << "s.size\t:" <<s.size()<<endl;
   cout << "s.length\t:" <<s.length()<<endl;
   cout << "s.capacity\t:" <<s.capacity()<<endl;
   cout << "s.maximum_size\t:" <<s.max_size()<<endl;
   cout << "s.Empty\t:" <<(s.empty() ? "yes" : "no")<<endl;
   cout << "-----------------------------------------------\n";
}

int main()
{
   string s1;            //empty string
   string s2("First");    // initiaze string
   string s3("Second");
```

```
   string_property(s1);
   string_property(s2);
   string_property(s3);
   string s4;
   cout << s2 << "+" << s3 << "=" << (s4 = s2 + s3) << endl; // concating string
   string_property(s4);
   string s5("hi hello how r you praful");
   cout << s5 << endl;
   cout << "find ""praful"":"<< s5.find("praful")<< endl;
   cout << "find ""rakesh"":" << s5.find("rakesh")<< endl;
   cout << "accessing as charector\n";
   for(int i=0; i < s5.length(); i++)
      cout << s5[i] ;
   cout << endl;
   s1 = "one";
   s2 = "one";
   s3 = "ONE";
   if(s1 == s2){/* comparing */
      cout << s1 << "=" << s2 << endl;
   }
   if(s1 == s3){
      cout << s1 << "=" << s3 << endl;
   }
   else
   {
      cout << s1 << "!=" << s3 << endl;
   }
   s1 = "hi";
   s2 = "hello";
   cout << "s1 = "<<s1<<" s2 = "<<s2<<endl;
   cout << "swaping operation"<<endl;
   s1.swap(s2);    /* swaping */
   cout << "s1 = "<<s1<<" s2 = "<<s2<<endl;
   s1.swap(s3);
   return 0;
}
```

String Important points

The biggest difference between calling reserve and specifying the size at construction is that reserve doesn't initialize the slots in the buffer with anything. Specifically, this means that you shouldn't reference indexes where you haven't already put something:

```
vector<string> vec(100);
string s = vec[50]; // No problem: s is now an empty string
vector<string> vec2;
vec2.reserve(100);
s = vec2[50]; // Undefined
```

Chapter 16 CPP best coding practice

1. **Make sure Header file include only once:**

```
#ifndef MYCLASS_H_ _ // #include guards
#define MYCLASS_H_ _
// Put everything here...
#endif // MYCLASS_H_ _
```

preprocessor #ifndef directive and the symbol that follows. #ifndef tells the preprocessor to continue processing on the next line only if the symbol MYCLASS_H_ _ is not already defined. If it is already defined, then the preprocessor should skip to the closing #endif.

2. Ensuring You Have Only One Instance of a Variable Across Multiple Source Files

declare all global data in single cpp source file, and define all global data in single header file.

example: global.h // header file

```
extern int wifi_flag
extern string software_version
```

global.cpp // gobal source file

```
#include "global.h"
int wifi_flag = 0;
string software_version = "version_1";
```

so whenever you wanna deal with global data you have 2 specific location to look-after. so if your project having 20 source file still you only need to look 2 file for anything related to global data.

3. Preventing Name collisions with Namespaces

use namespace to modularize code. large group of code in saparate file into single namespace.

```
namespace hardware {
class Device {
// ...
};
class DeviceMgr {
// ...
};
}// hardware namespace
```

The mechanism is simple: wrap everything you want to put in your namespace in a namespace. You can also nest namespaces to divide the contents of a namespace into smaller groups.

```
namespace hardware {
namespace net {
// network stuff
}
namespace devices {
// device stuff
}
}
```

include header file of namespace and you can use it by **using** directive.
using namespace hardware to access all.or
using hardware::net for only net namepspace.

important guidelines of namespace

1. Use **using** namespace xxx sparingly (carefully). if you add entire namespace it increase probability of name collision.
2. don't use **using** statement in **header** files. header file is included by many file. so not good practice.

4 . Number Coversion string -> (int, doube, hex)

```
#include <iostream>
#include <string>
#include <boost/lexical_cast.hpp>
using namespace std;
```

```
int main( ) {
string str1 = "750";
string str2 = "2.71";
string str3 = "0x7FFF";
try {
cout << boost::lexical_cast<int>(str1) << endl;
cout << boost::lexical_cast<double>(str2) << endl;
cout << boost::lexical_cast<int>(str3) << endl;
}
catch (boost::bad_lexical_cast& e) {
cerr << "Bad cast: " << e.what( ) << endl;
}
}
```

Testing valid number by using **isValid<type>(string)**

```
void test(const string& s) {
if (isValid<int>(s))
cout << s << " is a valid integer." << endl;
else
cout << s << " is NOT a valid integer." << endl;
if (isValid<double>(s))
cout << s << " is a valid double." << endl;
else
cout << s << " is NOT a valid double." << endl;
if (isValid<float>(s))
cout << s << " is a valid float." << endl;
else
cout << s << " is NOT a valid float." << endl;
}
```

Creating a class object in heap.

```
template<typename T>
T* createObject( ) {
return(new T( ));
}
MyClass* p1 = createObject( );
MyOtherClass* p2 = createObject( );
```

This approach is handy if you want a single factory function to be able to create objects of any number of classes (or a group of related classes) in the same way, with out having to write a redundant factory function multiple times.

Singletone Class

if only one instance is possible of a class it is singletone class.

1. Create Static member that is pointer of current class.
2. Restrict constructor to create by making private.
3. Provide public static function to access single instance.

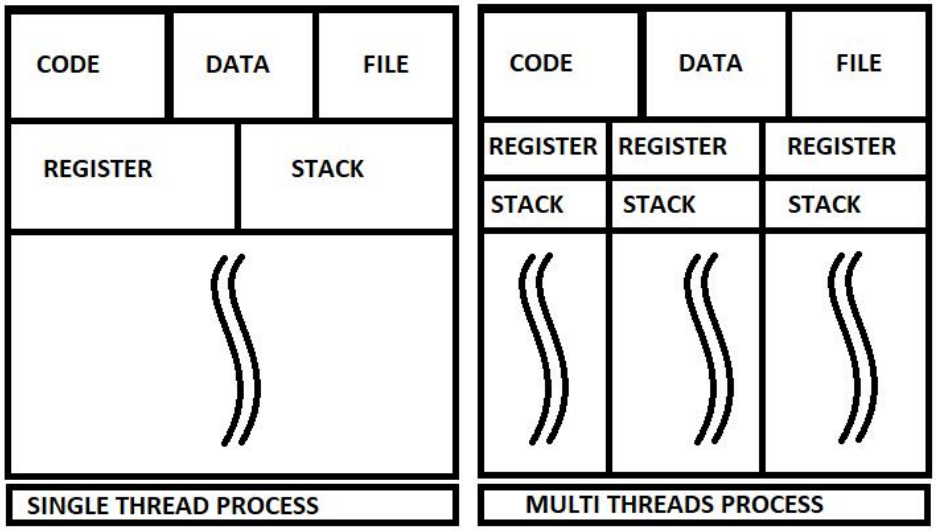
example:

```
class Singletone
{
  private:
     Singletone(): value(0){}       // default constructor
     Singletone(const Singletone&);  // copy constructor
     Singletone& operator=(const Singletone&); // assignment operator ovrload
     static Singletone* instant;
  protected:
```

```
    int value;
  public:
    static Singletone* getInstance();   // client can get instance
    void setValue(int val){value = val;}
    int getValue(){return (value);}
};

Singletone* Singletone:: instant = NULL;  // static instance pointer declare

Singletone* Singletone::getInstance()   // static function to get instance only once
{
  if(instant == NULL)
  {
    instant = new Singletone();
  }
  return instant;
}

int main()
{
  Singletone *p = Singletone::getInstance();
  p->setValue(10);
  cout << "address : "<< p << "p value = " << p->getValue()<< endl;

  Singletone *p1 = Singletone::getInstance();
  p1->setValue(20);
  cout << "address : "<< p1 << "p1 value = " << p1->getValue()<< endl;

  Singletone *p2 = Singletone::getInstance();
  cout << "address : "<< p2 <<"p2 value = " << p2->getValue()<< endl;

  return (0);
}
```

threads

threads are basic unit of execution. threads are sub part of a process. with treads we can do multiple work at same time.



threads share code section and data section and open file descriptor.

Install and use of boost library

```
sudo apt-get install libboost-all-dev
g++ sample.cpp -lboost_system
```

Three basic problem in multi-threaded application
deadlock , racecondition, starvation.

Deadlock is a situation that involves at least two threads and two resources. Consider two threads, A and B, and two resources, X and Y, where A has a lock on X and B has a lock on Y. A deadlock occurs when A tries to lock Y and B tries to lock X. If threads are not designed to break the deadlock somehow, then they will wait forever.

everybody gets a chance to use the resource. But if you let some consumers cut in line, it is possible that those at the back of the line never get their turn. This is **starvation**.

s