CHAPTER 3

Message, instance and initialization

Message Passing

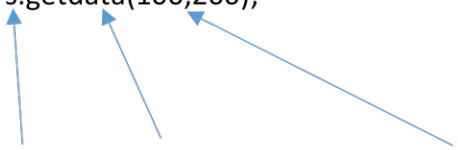
Message passing (sometimes also called method lookup) means the dynamic process of asking an object to perform a specific action.

- A message is always given to some object, called the receiver.
- The action performed in response to the message is not fixed but may be differ, depending on the class of the receiver .That is different objects may accept the same message the and yet perform different actions**.**

There are three identifiable parts to any message-passing expression. These are

1) **Receiver**: the object to which the message is being sent
2) **Message selector:** the text that indicates the particular message is being sent.
3) **Arguments** used in responding the message.

s.getdata(100,200);



receiver Message arguments

Example of message passing

```
#include<iostream>
#include<conio.h>
using namespace std;
class student
{
int roll;
public:
void getdata(int x)
{
roll=x;
}
void display()
{
cout<<"Roll number="<<roll;
}
};
```

```
int main()
{
student s;
s.getdata(325); //objects passing message
s.display(); //objects passing message
getch();
return 0;
}
```

Message passing vs procedure calls

The message passing and procedure calls are differ in following aspects.

Message passing

- In a message passing there is a designated receiver for that message; the receiver is some object to which message is sent.
- Interpretation of the message (that is method is used to respond the message) is determined by the receiver and can vary with different receivers. That is, different object receive the same message and yet perform different actions.
- Usually, the specific receiver for any given message will not be known until run time, so determination of which method cannot be made until then. Thus, we say there is a late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message.

Procedure calls

- In a procedure call, there is no designated receiver.
- Determination of which method to invoke is very early(compile-time or link time) binding of a name to fragment in procedure calls.

Constructor

- A constructor is a special member function which initializes the objects of its class. It is called special because its name is same as that of the class name.
- The constructor is automatically executed whenever an object is created. Thus, a constructor helps to initialize the objects without making a separates call to a member function.
- It is called constructor because it constructs values of data members of a class.

Characteristics of constructor

- A constructor has same name as class name.
- They are invoked automatically when the objects are created.
- They should be declared in the public section.
- They do not have return types.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the new and delete operators when a memory allocation is required.

Types of Constructor:

1. Default constructor

- A constructor that accepts no arguments (parameters) is called the default constructor.
- If a class does not include any constructor, the compiler supplies a default constructor.

The default Constructor is declared and defined as follows:

```
class Complex
{
private:
int real,imag;
public:
Complex
{
real=0;
imag=0;
}
. . . . . .
. . . . . .
};
```

When a class contains a constructor like the defined above, it is guaranteed that an object by class will be initialized automatically. For example, the declaration
Complex c1; not only creates the object **c1** of type **Complex** but also initializes its data members real and imag to zero. There is no need to write any statement to invoke the constructor function (as we do with normal member functions).

Program:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag ;
public:
Complex()
{
        real=0;
        imag=0;
}
void display( )
{
 cout<<"Real part="<<real<<endl;
 cout<<"Imaginary part="<<imag<<endl ;
}
} ;
int main( )
{
Complex c1;
c1.display();
return 0;
}
```

Output:

```
Real part=0
Imaginary Part=0
```

2. Parameterized Constructor

The constructor that can take arguments are called parameterized constructor.
Arguments are passed when the objects are created. This can be done in two ways.

- by calling the constructor explicitly
- by calling the constructor implicitly

For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type int.

```
class Complex
{
private:
int real,imag;
public:
Complex (int r, int i) //parameterized constructor
{
real=r;
imag=i;
}
. . . . . .
. . . . . .
};
int main()
{
Complex c1(5,2); //implicit call
Complex c2 =Compex(7,4); //explicit call
. . . . . .
}
```

Example:

```
#include <iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
Complex(int r,int i)
{
          real=r;
          imag=i;
}
 void display( )
  {
 cout<<"Real part="<<real<<endl;
 cout<<"Imaginary part="<<imag<<endl ;
  }
};

int main( )
{
Complex c1(5,2);
c1.display();
return 0;
}
```

Output:

```
Real part=5
Imaginary part=2
```

3) Copy Constructor

A copy constructor is used to declare and initialize an object with another object of the same type.
For example, the statement
Complex c2(c1);
Creates new object c2 and performs member-by-member copy of c1 into c2. Another form of this statement is
Complex c2 = c1;

The process of initializing through assignment operator is known as copy initialization.
A copy constructor takes reference to an object of the same class as its argument. For example,

```
Complex (Complex &x)
{
real=x.real;
Imag=x.imag;
}
```

Remember: ***We cannot pass the argument by value to a copy constructor***

When no copy constructor is defined, the compiler supplies its own copy constructor.

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real, imag ;
public:
Complex( )
{
}
Complex (int r,int i)
{
real=r ;
imag=i ;
}
Complex (Complex &x)
{
real=x.real ;
imag=x.imag ;
}
void display( )
{
cout<<"Real part="<<real<<endl;
cout<<"Imaginary part="<<imag<<endl;
}
};
```

```
int main( )
{
Complex c1(5,10) ;
Complex c2(c1) ; // copy constructor called
Complex c3;
c3=c1;
cout<<"Details of c1"<<endl;
c1.display( ) ;
cout<< "Details of c2"<<endl;
c2.display( ) ;
cout<< "Details of c3"<<endl;
c3.display( ) ;
return 0;
}
```

Constructor Overloading

We can have more than one constructor in a class with same name and different number of arguments. This concept is known as Constructor Overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

```
#include<iostream>
using namespace std;
class Complex
{
int real,imag;
public:
Complex() //Default Constructor
{
real=0;
imag=0;
}
Complex(int r,int i)  //Parameterized Constructor
{
real=r;
imag=i;
}
```

```
Complex(Complex &x) //Copy Constructor
{
real=x.real;
imag=x.imag;
}

void display()
{
cout<<"Real part:"<<real<<endl;
cout<<"Imaginary part"<<imag<<endl;
}
};
int main()
{
Complex c1;
Complex c2(5,2);
Complex c3(c2);
c1.display();
c2.display();
c3.display();
return 0;
}
```

In the above example of class Complex, we have defined three constructors. The first one is invoked when we don't pass any arguments. The second gets invoked when we supply two argument, while the third one gets invoked when an object is passed as an argument.
Example
Complex c1;
This statement would automatically invoke the first one.
Complex c2(102,300); invokes 2nd constructor
Complex c3(c2); invokes 3rd constructor

Constructor with default argument

It is possible to define constructor with default arguments.

```
#include<iostream>
using namespace std;
class test
{
int a,b,c;
public:
test(int x=1,int y=2,int z=3);
void display();
};
test::test(int x,int y,int z)
{
a=x;b=y;c=z;
}
void test::display()
{
cout<<"a="<<a<<"b="<<b<<"c="<<c<<endl;
}
int main()
{
test t1;
test t2(5,10,15);
test t3(6);
t1.display();
t2.display();
t3.display();
return 0;
}
```

Output:

```
a=1b=2 c=3
a=5b=10 c=15
a=6b=2 c=3
```

Destructors

- Destructor is a member function that destroys the object that have been created by a constructor.
- Destructor name is similar to class name but preceded by a tilde sign (~).
- They are also defined in the public section.
- Destructor never takes any argument, nor does it return any value. So, they cannot be Overloaded.

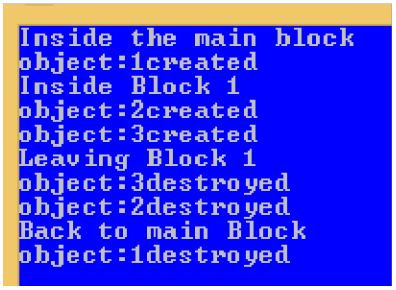
- Destructor gets invoked, automatically, when an object goes out of scope (i.e. exit from the program, or block or function).

  Example:
  A destructor for the class test will look like

  ```
  ~test()
  {
   - - - - - -
  - - - - - -
  }
  ```

```
#include<iostream>
using namespace std;
class test
{
static int count;
public:
test()
{
count++;
cout<<"object:"<<count<<"created"<<endl;
}
~test()
{
cout<<"object:"<<count<<"destroyed"<<endl;
count--;
}
};
int test::count;
int main()
{
cout<<"Inside the main block"<<endl;
test t1;
{
        //Block 1
        cout<<"Inside Block 1"<<endl;
        test t2,t3;
        cout<<"Leaving Block 1"<<endl;
}
cout<<"Back to main Block"<<endl;
return 0;
}
```

Output:



New and Delete Operator

New

- Dynamic memory is allocated using operator **new**.
- **new** is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [ ].
- It returns a pointer to the beginning of the new block of memory allocated.

***Syntax*:** pointer-variable = new data-type;

This expression is used to allocate memory to contain **one single element of type data-type.**

For Example: int *ptr;
ptr=new int;

Similarly,

Syntax: pointer-variable = new data-type [size];

This one is used to allocate **a block (an array) of elements of type data-type,** where size is an integer value representing the number of elements.

For Example:

```
int *ptr;
ptr = new int [5];
```

Delete

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program. Once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- **delete** operator free the memory allocated to the variable i.e. it deletes the variables memory.

Syntax is:

```
delete pointer-variable;
delete [size]  pointer-variable;
```

- The first statement releases the memory of a single element allocated using new
- The second one releases the memory allocated for arrays of elements using new and a size in brackets ([ ]).

Note: The size specifies the number of elements in the array to be freed. The problem with this form is that programmer should remember the size of the array. Recent version of c++ do not require the size to be specified. for example,
delete[] ptr;

Example Program for new & delete operator

```
#include <iostream>
using namespace std;
int main ()
{
int i,size;
int *ptr;
cout << "How many numbers would you like to Enter? ";
cin >>size;
ptr= new int[size];
for (i=0; i<size; i++)
{
cout << "Enter number:"<<i+1<<endl;
cin >> ptr[i];
}
cout << "You have entered:"<<endl;
for (i=0; i<size; i++)
{
cout << ptr[i]<<endl ;
}
delete[] ptr;
return 0;
}
```

Memory Mapping, Allocation and Recovery

Stack & Heap Memory

Stack

- It's a region of computer's memory that stores temporary variables created by each function (including the main() function).
- The stack is a "Last in First Out" data structure and limited in size. Every time a function declares a new variable, it is "pushed" (inserted) onto the stack. Every time a function exits, all of the variables pushed onto the stack by that function, are freed or popped (that is to say, they are deleted).
- Once a stack variable is freed, that region of memory becomes available for other stack variables.
- A key to understanding the stack is the notion that when a function exits, all of its variables are popped off (Removed) of the stack (and hence lost forever).

Advantages

- Memory is managed to store variables automatically. We don't have to allocate memory by hand, or free it once we don't need it any more.
- CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

Limitations

- Limit (varies with OS) on the size of variables that can be store on the stack.

Heap

- The heap is a region of computer's memory that is not managed automatically and is not as tightly managed by the CPU.
- Dynamic memory allocation is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory -- instead, it is allocated from a much larger pool of memory managed by the operating system called the heap. It is a more free-floating region of memory (and is larger).
- To allocate memory on the heap, you must use **new** operator in C++.Once you have allocated memory on the heap, you are responsible for using **delete** to de-allocate that memory once you don't need it any more.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of computer).
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.

- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Stack Vs Heap

Stack	Heap
1) Only Local variables can be accessed.	1) Variables can be accessed globally
2) Limit on stack size dependent on OS.	2) Does not have a specific limit on memory size.
3) It can access faster.	3) Relatively slower access.
4) Don't have to explicitly de-allocate variables	4) We must allocate and de-allocate variable explicitly.(for allocating variable new and for freeing variables delete operator is used.
5) Variables cannot be resized.	5) Variables can be resized using new operator
6) Space is managed efficiently by CPU, memory will not become fragmented.	6) No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed.
7) Main issue is shortage of memory.	7) Main issue is Memory fragmentation.
8) It is used for static memory allocation, meaning memory is allocated at compile time before the program executes.	8) It is used for dynamic memory allocation meaning the memory can be allocated and freed in random order.

When to use the Heap or stack?

We should use heap when we require to allocate a large block of memory. For example, you want to create a large size array or big structure to keep that variable around a long time then you should allocate it on the heap.

However, If we are working with relatively small variables that are only required until the function using them is alive. Then we need to use the stack, which is faster and easier.

Dynamic Constructor

Dynamic constructor is used to allocate the memory to the objects at the run time. Memory is allocated at run time with the help of 'new' operator .This will enable the system to allocate right amount of memory for each object when objects are not of the same size, thus resulting in the saving of memory.
Allocation of memory to objects at the time of their construction is known as dynamic constructions of objects. The memory is allocated with the help of new operator.

Example:
Program to find the sum of two Complex number using dynamic constructor.

```
#include<iostream>
using namespace std;
class complex
{
int *real,*imag;
public:
complex()
{
real=new int;
imag=new int;
*real=0;
*imag=0;
}
complex(int r,int i)
{
real=new int;
*real=r;
imag=new int;
*imag=i;
}
void display()
{
cout<<*real<<"+i"<<*imag<<endl;
}
void addcomplex(complex c1,complex c2)
{
*real=*c1.real+*c2.real;
*imag=*c1.imag+*c2.imag;
}
};
```

```
int main()
{
complex c1(5,10);
complex c2(2,4);
complex c3;
cout<<"First complex number=";
c1.display();
cout<<"Second complex number=";
c2.display();
cout<<"Sum =";
c3.addcomplex(c1,c2);
c3.display();
return 0;
}
```

WAP to concatenate two string using the concept of dynamic constructor.

```
#include<iostream>
#include<string.h>
using namespace std;
class stringc
{
char *name;
int length;
public :
stringc()
{
length=0;
name=new char[length + 1];
}
stringc(char s[])
{
length=strlen(s);
name=new char[length+1];
strcpy(name,s);
}
void join(stringc &a,stringc &b)
{
length=a.length+b.length;
delete name;
name=new char[length+1];
strcpy(name,a.name);
strcat(name,b.name);
}
```

```
void display()
{
cout<<name<<endl;
}
};
int main ()
{
stringc s1("pokhara");
stringc s2("university");
stringc s3;
s3.join(s1,s2);
s3.display();
return 0;
return 0;
}
```

Dynamic initialization of object

Dynamic initialization of object refers to initializing the objects at run time i.e. the initial value of an object is to be provided during run time. Dynamic initialization can be achieved using constructors and passing parameters values to the constructors. This type of initialization is required to initialize the class variables during run time.

Need /Advantages of dynamic initialization of object

- It utilizes memory efficiently.
- Various initialization formats can be provided using overloaded constructors.
- It has the flexibility of using different formats of data at run time considering the situation.

Program example to explain the concept of dynamic initialization

```
#include <iostream>
using namespace std;
class simple_interest
{
float principle , time, rate ,interest;
public:
simple_interest (float a, float b, float c)
{
principle = a;
time =b;
rate = c;
}
```

```
void display ( )
{
interest =(principle* rate* time)/100;
cout<<"interest ="<<interest ;
}
};
int main()
{
float p,t,r;
cout<<"principle amount, time and rate"<<endl;
cin>>p>>t>>r;
simple_interest s1(p,t,r);//dynamic initialization
s1.display();
return 0;
}
```

As we know, one of the advantage of dynamic initialization of objects is that we can provide various initialization formats using overloaded constructor.
Let us consider the following program,

```
#include<iostream>
using namespace std;
class Fixed_deposit
{
private:
long int principal;
int years;
float rate;
float returnvalue;
public:
        Fixed_deposit()
        {    }
        Fixed_deposit(long int p,int y,float r)
        {
                principal=p;
                years=y;
                rate=r;
                returnvalue=principal;
                for(int i=1;i<=y;i++)
                {
                        returnvalue=returnvalue*(1.0+r);
                }
        }
```

```
        Fixed_deposit(long int p,int y,int r)
        {
                principal=p;
                years=y;
                rate=r;
                returnvalue=principal;
                for(int i=1;i<=y;i++)
                {
                        returnvalue=returnvalue*(1.0+float(r)/100);
                }
        }
        void display()
        {
                cout<<"principal amount="<<principal<<endl;
                cout<<"Return value="<<returnvalue<<endl;

        }
};
int main()
{
        Fixed_deposit  fd1,fd2,fd3;
        long int p;
        int y,r;
        float R;
        cout<<"Enter amount ,period and interest rate (in percent)"<<endl;
        cin>>p>>y>>r;
        fd1=Fixed_deposit(p,y,r);
        cout<<"Enter amount, period and interest rate(decimal form)"<<endl;
        cin>>p>>y>>R;
        fd2=Fixed_deposit(p,y,R);
        cout<<"Deposit 1:"<<endl;
        fd1.display();
        cout<<"Deposit 2:"<<endl;
        fd2.display();
        return 0;
}
```

This program provides the flexibility of using different format of data at runtime depending upon situation.

This program uses three overloaded constructors. The parameters values to these constructors are provided at runtime. The user can provide input in the following forms:

1. Amount, period and interest in percent form
2. Amount, period and interest in decimal form

Since the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked.

For example, the third constructor is invoked for the forms (1) and second constructor is invoked for forms(2).

Previous Board Exam Questions

1) What is message passing? Describe with example**.[PU:2014 fall]**

2) What is the difference between message passing and function call? Explain the basic message formalization. **[PU:2006 spring]**

3) Differentiate message passing and procedure call with suitable example. What are the possible memory errors in programming.**[PU:2014 spring]**

4) Explain the following term with suitable examples. Agents ,Responsibility, Messages and methods.**[PU:2009 fall]**

5) Explain message passing formalism with syntax in C++.What is stack Vs heap memory allocation? **[PU:2016 spring]**

6) What does constructor mean? Explain different types of constructor with suitable examples. **[PU:2005 fall]**

7) What is constructor? Explain copy constructor with suitable examples.[PU:2009 fall]

8) What is constructor? Write an example of Copy constructor and explain each line of code. **[PU:2017 spring**]

9) "A constructor is a special member function that automatically initializes the objects of its class", support this statement with a program of all types of constructors. Also enlist the characters of constructors. **[PU:2015 spring]**

10) What is constructor? Is it mandatory to use constructor in class. Explain **[PU:2006 spring]**

11) What are default and parametrized constructors? Write a program to illustrate parametrized constructor.**[PU:2020 fall]**

12) What is copy constructor in C++? Is it possible to pass object as argument in Copy constructor? Explain with suitable program**.[PU:2020 fall]**

13) Differentiate between constructor and destructor. Can there be more than one destructor in a program for destroying a same object. Illustrate you answer.**[PU:2010 fall]**

14) What are constructors and destructors? Explain their types and uses with good illustrative example? What difference would be experienced if the features of constructors and destructors were not available in C++. **[PU:2009 spring]**
15) What is de-constructor? can you have two destructors in a class? Give example to support your reason.**[PU:2014 spring]**
16) Discuss the various situations when a Copy constructor is automatically invoked. How a default constructor can be equivalent to a constructor having default arguments. **[PU:2013 spring]**
17) Can we have more than one destructor in a class? Write a Program to add two complex numbers using the concept of constructor. **[PU:2015 spring]**
18) What do you mean by dynamic constructor? Explain its application by a program to compute complex numbers. **[PU:2016 fall]**
19) Differentiate methods of argument passing in constructor and destructor.**[PU:2017 fall]**
20) Why destructor function is required in class? Can a destructor accept arguments? **[PU:2017 spring]**
21) What is constructor? Can constructor can be overloaded? If yes how that is possible with reference of an example? **[PU:2018-fall][PU:2017 fall]**
22) Discuss about stack vs heap storage allocation. **[2010 spring]**
23) What do you mean by stack vs Heap? Explain the memory recovery. Explain the use of new and delete operator. **[PU:2009 spring]**
24) Explain and contrast memory recovery, stack and heap with suitable example.**[PU:2013 fall]**
25) What are the advantages of dynamic memory allocation? Explain with suitable example. **[PU:2016 spring]**
26) What is dynamic memory allocation? How memory is allocated and de-allocated in C++?Explain with examples**.[PU:2019 spring]**
27) What is memory recovery? How does stack differ from heap memory allocation. **[PU:2005 fall]**
28) Write a short notes on:
    - Copy constructor**[PU:2014 spring]**
    - Stack vs Heap based Allocation
    - Memory Recovery**[PU:2015 spring]**
    - Message passing in C++**[PU:2019 spring]**

1) **WAP to find the area of rectangle using the concept of constructor**

```
#include<iostream>
using namespace std;
class Rectangle
{
  private:
  float length,breadth,area;
  public:
    Rectangle(float l,float b)
    {
       length = l;
       breadth = b;
    }
     void calc( )
     {
          area= length * breadth;
     }
     void display( )
     {
      cout << "Area = " << area << endl;
     }
};
int main()
{
   Rectangle r(2,4);
   r.calc();
   r.display();
   return 0;
 }
```

2) **WAP to initialize name, roll and marks of student using constructor and display then obtained information.**

```
#include<iostream>
#include<string.h>
using namespace std;
class Student
{
private:
char name[20];
int roll;
float marks;
```

```
public:
Student(char n[],int r,float m)
{
strcpy(name,n);
roll=r;
marks=m;
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll:"<<roll<<endl;
cout<<"Marks:"<<marks<<endl;
}
};
int main()
{
char name[20];
int roll;
float marks;
cout<<"Enter name"<<endl;
cin>>name;
cout<<"Enter marks"<<endl;
cin>>marks;
cout<<"Enter Rollno"<<endl;
cin>>roll;
Student st(name,marks,roll);
st.display();
return 0;
}
```

3) Create a class student with data members (name, roll, marks in English, Maths and OOPs(in 100), total, average) a constructor that initializes the data members to the values passed to it as parameters, A function called calculate ( ) that calculates the total of the marks obtained and average. And function print() to display name, roll no, total and average.

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
   private:
   int roll;
   float me,mm,mo,total,avg;
   char name[30];
   public:
   student(char n[],int r,float e,float m,int o)
   {
     strcpy(name,n);
     roll=r;
     me=e;
     mm=m;
     mo=o;
   }
   void  calculate()
   {
       total= me+mm+mo;
       avg=total/3;
   }
  void print()
  {
   cout<<"Name="<<name<<endl;;
   cout<<"Roll No="<<roll<<endl;;
   cout<<"Total="<<total<<endl;
   cout<<"Average="<<avg<<endl;
  }
};
```

```
int main()
{
   char name[30];
   int roll;
   float meng,mmath,moops;
   cout<<"Enter the name"<<endl;
   cin>>name;
   cout<<"Enter the roll"<<endl;
   cin>>roll;
   cout<<"Enter the marks in english"<<endl;
   cin>>meng;
    cout<<"Enter the marks in maths"<<endl;
   cin>>mmath;
   cout<<"Enter marks in oops"<<endl;
   cin>>moops;
   student st(name,roll,meng,mmath,moops);
   st.calculate();
   st.print();
   return 0;
}
```

4) Write a Program to add two complex number using the concept of Constructor/Constructor overloading.[PU:2015 spring]

```
#include<iostream>
using namespace std;
class Complex
{
private:
int real,imag;
public:
Complex()
{
real=2;
imag=4;
}
Complex(int r,int i)
{
real=r;
imag=i;
}
```

```
void addcomplex(Complex c1,Complex c2)
{
        real=c1.real+c2.real;
        imag=c1.imag+c2.imag;
}
void display()
{
 cout<<real<<"+"<<imag<<"i"<<endl;
}
};
int main()
{
Complex c1;
Complex c2(10,20);
Complex c3;
cout<<"First complex number=";
c1.display();
cout<<"Second complex number=";
c2.display();
c3.addcomplex(c1,c2);
cout<<"sum of complex number=";
c3.display();
return 0;
}
```

5) Create a class person with data members Name, age, address and citizenship number. Write a constructor to initialize the value of the person. Assign citizenship number if the age of the person is greater than 16 otherwise assign value zero to citizenship number. Also create a function to display the values.[PU:2013 fall]

```
#include<iostream>
#include<string.h>
using namespace std;
class Person
{
  private:
   char name[20];
   char address[20];
   int age;
   long int citzno;
```

```
public:
   Person(char n[],int a,char addr[],long int  c)
   {
     strcpy(name,n);
     age=a;
     strcpy(address,addr);
     citzno=c;
   }

void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Address:"<<address<<endl;
cout<<"Age:"<<age<<endl;
cout<<"Citizenship no:"<<citzno<<endl;
}
};
int main()
{
   char name[20];
   char address[20];
   int age;
   long int citzno;
   cout<<"Enter the name"<<endl;
   cin>>name;
   cout<<"Enter the address"<<endl;
   cin>>address;
   cout<<"Enter the age"<<endl;
  cin>>age;
 if (age>16)
  {
    cout<<"Enter the citizenship number"<<endl;
    cin>>citzno;
    Person p(name,age,address,citzno);
    p.display();
   }
   else
   {
   Person p(name,age,address,0);
   p.display();
   }
   return 0;
}
```

6) **Create a class Mountain with data members name, height, location, a constructor that initializes the members to the values passed it to its parameters, a function called CmpHeight() to compare two objects and DispInf() to display the information of mountain. In main create two objects of the class mountain and print the information of the mountain which is greatest height.[PU:2016 spring]**

```
#include<iostream>
#include<string.h>
using namespace std;
class Mountain
{
private:
char name[20],location[20];
float height;
public:
Mountain(char n[],float h,char l[])
{
 strcpy(name,n);
 height=h;
 strcpy(location,l);
}

void DispInf()
{
        cout<<"Name:"<<name<<endl;
        cout<<"Height:"<<height<<endl;
        cout<<"Location:"<<location<<endl;
}
friend void CmpHeight(Mountain,Mountain);
};

void CmpHeight(Mountain m1,Mountain m2)
{
        if(m1.height>m2.height)
        {
          m1.DispInf();
        }
        else
        {
           m2.DispInf();
        }
}
```

```
int main()
{
char name1[20],name2[20];
float height1,height2;
char location1[20],location2[20];
cout<<"Enter Name,Height and location of first mountain"<<endl;
cin>>name1>>height1>>location1;
cout<<"Enter Name,Height and location of second mountain"<<endl;
cin>>name2>>height2>>location2;
Mountain m1(name1,height1,location1);
Mountain m2(name2,height2,location2);
cout<<"Information of mountain with greatest height"<<endl;
CmpHeight(m1,m2);
return 0;
}
```

7) **Create a class time constructor having hour, minute and second as an arguments is use to take two time data from user. The add function that takes two class objects an arguments add them respectively then display the aggregate result?**
(Apply 60 second =1 minutes and 60 minutes =1 hour)
[PU: 2017 spring]

```
#include<iostream>
using namespace std;
class time
{
        private:
        int hr,min,sec;
        public:
                time()
                {

                }
                time(int h,int m,int s)
                {
                hr=h;
                min=m;
                sec=s;
                }
                void display()
                {
                        cout<<hr<<":"<<min<<":"<<sec<<endl;
                }
                void sum(time,time);
};
```

```
void time::sum(time t1,time t2)
{
        sec=t1.sec+t2.sec;
        min=sec/60;
        sec=sec%60;
        min=min+t1.min+t2.min;
        hr=min/60;
        min=min%60;
        hr=hr+t1.hr+t2.hr;
}
int main()
{
int hr1,min1,sec1,hr2,min2,sec2;
cout<<"Enter first hour,minutes and seconds"<<endl;
cin>>hr1>>min1>>sec1;
cout<<"Enter second hour,minutes and seconds"<<endl;
cin>>hr2>>min2>>sec2;
time t1(hr1,min1,sec1);
time t2(hr2,min2,sec2);
time t3;
cout<<"The 1st time is"<<endl;
t1.display();
cout<<"The 2nd time is"<<endl;
t2.display();
t3.sum(t1,t2);
cout<<"The resultant time is"<<endl;
t3.display();
return 0;
}
```

8) Create a class called employee with data member Code, Name, address, salary. Create a constructor to initialize the member of the class. Also create the another constructor so that we can create an object from another object. Define member function display() to display the information of the class.[PU:2018 fall]

```
#include<iostream>
#include<string.h>
using namespace std;
class employee
{

        private:
        int code;
        char name[20];
        char address[20];
        float salary;
        public:
  employee(int c, char n[],char addr[],float s)
   {
        code=c;
        strcpy(name,n);
        strcpy(address,addr);
        salary=s;
  }

   employee(employee &e)
        {
                code=e.code;
                strcpy(name,e.name);
                strcpy(address,e.address);
                salary=e.salary;
        }
        void display()
             {
                        cout<<"Code="<<code<<endl;
                        cout<<"Name="<<name<<endl;
                        cout<<"Address="<<address<<endl;
                        cout<<"Salary="<<salary<<endl;
                }
        };
```

```
int main()
  {
     int code;
     char name[20];
     char address[20];
     float salary;
      cout<<"Enter Code:"<<endl;
     cin>>code;
     cout<<"Enter Name:"<<endl;
     cin>>name;
     cout<<"Enter Address:"<<endl;
     cin>>address;
     cout<<"Enter Salary:"<<endl;
     cin>>salary;
     employee e1(code,name,address,salary);
    employee e2(e1);
     cout<<"Values in object e1"<<endl;
     e1.display();
     cout<<"Values in object e2"<<endl;
     e2.display();
    return 0;
}
```

9) **Write a simple program using of dynamic memory allocation which should include calculation of marks of 3 subjects of n students and displaying the result as pass or fail & name, roll. Pass mark is 45 out of 100 in each subject**

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
float m1,m2,m3;
public:
void getdata()
{
cout<<"enter the name"<<endl;
cin>>name;
cout<<"enter the roll"<<endl;
cin>>roll;
cout<<"Enter marks of 3 subjects"<<endl;
cin>>m1>>m2>>m3;
}
```

```
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Roll:"<<roll<<endl;
if(m1>=45&&m2>=45&&m3>=45)
cout<<"Result:Pass"<<endl;
else
cout<<"Result:Fail"<<endl;
}
};

int main()
{
        int n,i;
        student *ptr;
        cout<<"Enter the number of students:";
        cin>>n;
        ptr= new student[n];
        for(i=0;i<n;i++)
        {
        cout<<"Enter the information of student"<<i+1<<endl;
        ptr[i].getdata();
        }
        for(i=0;i<n;i++)
        {
        cout<<"Information of student"<<i+1<<endl;
        ptr[i].display();
        }
        delete[] ptr;
        return 0;
}
```

10) Write a simple program explaining the use of dynamic memory allocation which should include calculation of marks of 5 subjects of students and displaying the result of pass or fail.(Pass marks is 45 out of 100 in each subject) [PU: 2010 fall 3(b)]

```
#include<iostream>
using namespace std;
class Student
{
private:
float sub1,sub2,sub3,sub4,sub5;
public:
void getdata()
{
cout<<"Enter marks of 5 subjects"<<endl;
cin>>sub1>>sub2>>sub3>>sub4>>sub5;
}
void display()
{
if(sub1>=45&&sub2>=45&&sub3>=45&&sub4>=45&&sub5>=45)
{
cout<<"Student is passed"<<endl;
}
else
{
cout<<"Student is failed"<<endl;
}
}
};
int main()
{
        int n,i;
        Student *ptr;
        cout<<"Enter the number of students:";
        cin>>n;
        ptr= new Student[n];
        for(i=0;i<n;i++)
        {
        cout<<"Enter marks of student"<<i+1<<endl;
        ptr[i].getdata();
        ptr[i].display();
        }
        delete[] ptr;
        return 0;
}
```

11) WAP to find the sum of n numbers Entered by user using Dynamic Memory Allocation in C++.

```
#include <iostream>
using namespace std;
int main ()
{
int i,size,sum=0;
int *ptr;
cout << "How many numbers would you like to Enter?"<<endl;
cin >>size;
ptr= new int[size];
for (i=0; i<size; i++)
{
cout << "Enter number"<<i+1<<endl;
cin >> ptr[i];
}
cout << "You have entered:"<<endl;
for (i=0; i<size; i++)
{
cout << ptr[i] << endl;
}
for (i=0; i<size; i++)
{
sum=sum+ptr[i];
}
cout<<"sum of numbers="<<sum<<endl;
delete[] ptr;
return 0;
}
```