OOPS UNIT 2.1

By Deepti Loharikar Talnikar

DYNAMIC ALLOCATION

Suppose you want to put a toy in a box, but you only have an approximate idea of its size. For that, you would require a box whose size is equal to the approximate size of the toy.

The new operator is used to allocate memory at runtime. The memory is allocated in bytes.

To allocate a variable dynamically.

```
int *ptr = new int;
```

By writing new int, we allocated the space in memory required by an integer. Then we assigned the address of that memory to an integer pointer ptr.

Suppose we declared the array with the array size 30 as follows.

```
char name[30];
```

And if the user enters the name having only 12 characters, then the rest of the memory space which was allocated to the array at the time of its declaration would become waste, thus unnecessary consuming the memory.

we will be using the new operator to dynamically allocate the memory at runtime. We use the new operator as follows.

```
char *arr = new char[length];
```

```
#include <iostream>

using namespace std;

int main()
{
        int length, sum = 0;
        cout << "Enter the number of
students in the group" << endl;
        cin >> length;
        int *marks = new int[length];
        cout << "Enter the marks of the
students" << endl;
for( int i = 0; i < length; i++ )            // entering marks of
students
        {
                cin >> *(marks+i);
        }
        for( int i = 0; i < length; i++ )            // calculating sum
        {
                sum += *(marks+i);
        }
        cout << "sum is " << sum << endl;
        return 0;
}
```

DELETE

Different groups have a different number of students, therefore we are asking the number of students (i.e.the size of the array) every time we are running the program. In this example, the user entered the size as 4.
int *marks = new int[length]; - We declared an array of integer and allocated it some space in memory dynamically equal to the size which would be occupied by length number of integers. Thus it is allocated a space equal to 'length * (size of 1 integer)' and assigned the address of the assigned memory to the pointer marks.

We call this array dynamic because it is being assigned memory when the program runs. We made this possible by using the new operator. This dynamic array is being allocated memory from heap unlike other fixed arrays which are provided memory from stack. We can give any size to these dynamic arrays and there is no limitation to it.

Now what if the variable to which we dynamically allocated some memory is not required anymore?

In that case, we use the delete operator.

To delete the memory assigned to a variable, we simply need to write the following code.

```
delete ptr;
```

```
int main()
{
        int length, sum = 0;
        cout << "Enter the number of students in the group" << endl;
        cin >> length;
        int *marks = new int[length];
        cout << "Enter the marks of the students" << endl;
        for( int i = 0; i < length; i++ )            // entering marks of students
        {
                cin >> *(marks+i);
        }
        for( int i = 0; i < length; i++ )            // calculating sum
        {
                sum += *(marks+i);
        }
        cout << "sum is " << sum << endl;
        delete[] marks;
        return 0;
}
```

Overloading New and Delete operator in C++

Syntax for overloading the new operator :

```
void* operator new(size_t size);
```

The overloaded new operator receives size of type size_t, which specifies the number of bytes of memory to be allocated. The return type of the overloaded new must be void*.The overloaded function returns a pointer to the beginning of the block of memory allocated.

Syntax for overloading the delete operator :

```
void operator delete(void*);
```

The function receives a parameter of type void* which has to be deleted. Function should not return anything.

```
#include<iostream>
#include<stdlib.h>

using namespace std;
class student
{
    string name;
    int age;
public:
    student()
    {
        cout<< "Constructor is called\n" ;
    }
    student(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
    void display()
    {
        cout<< "Name:" << name << endl;
        cout<< "Age:" << age << endl;
    }
void * operator new(size_t size)
    {
        cout<< "Overloading new operator with size: " << size << endl;
        void * p = ::operator new(size);
        //void * p = malloc(size); will also work fine

        return p;
    }

    void operator delete(void * p)
    {
        cout<< "Overloading delete operator " << endl;
        free(p);
    }
};
int main()
{
    student * p = new student("Yash", 24);

    p->display();
    delete p;
}
```

O/P

```
Overloading new operator with size: 40
Name:Yash
Age:24
Overloading delete operator
```

In the above new overloaded function, we have allocated dynamic memory through new operator, but it should be global new operator otherwise it will go in recursion

```
void *p = new student(); // this will go in recursion as new will be overloaded again and again
void *p = ::new student(); // this is correct
```

```
// C++ program to illustrate free()
// and delete keyword in C++
#include "bits/stdc++.h"
using namespace std;

// Class A
class A {
	int a;

public:
	int* ptr;
	// Constructor of class A
	A()
	{
		cout << "Constructor was Called!"
			<< endl;
	}
// Destructor of class A
	~A()
	{
		cout << "Destructor was Called!"
			<< endl;
	}
};
```

```
int main() //driver's code
{
	// Create an object of class A
	// using new operator
	A* a = new A;
	cout << "Object of class A was "
		<< "created using new operator!"
		<< endl;
	delete (a);
	cout << "Object of class A was "
		<< "deleted using delete keyword!"
		<< endl;
	cout << endl;

	A* b = (A*)malloc(sizeof(A));
	cout << "Object of class A was "
		<< "created using malloc()!"
		<< endl;
	free(b);
	cout << "Object of class A was "
		<< "deleted using free()!"
		<< endl;
	return 0;
}
```

O/P

```
Constructor was Called!
Object of class A was created using new operator!
Destructor was Called!
Object of class A was deleted using delete keyword!

Object of class A was created using malloc()!
Object of class A was deleted using free()!
```

Operator Overloading using Friend Function

```
#include <iostream>
using namespace std;
class Complex
{
    private:
        int real;
        int img;
    public:
        Complex (int r = 0, int i = 0)
        {
            real = r;
            img = i;
        }
        void Display ()
        {
            cout << real << "+i" << img;
        }
        friend Complex operator + (Complex c1, Complex c2);
};
```

```
Complex operator + (Complex c1, Complex c2)
{
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.img = c1.img + c2.img;
    return temp;
}

int main ()
{
    Complex C1(5, 3), C2(10, 5), C3;
    C1.Display();
    cout << " + ";
    C2.Display();
    cout << " = ";
    C3 = C1 + C2;
    C3.Display();
}
```

```
O/P: 5 + i3 + 10 + i5 = 15 + i8
```

Points to Remember While Overloading Operator using Friend Function:

The Friend function in C++ using operator overloading offers better flexibility to the class.
The Friend functions are not a member of the class and hence they do not have 'this' pointer.
When we overload a unary operator, we need to pass one argument.
When we overload a binary operator, we need to pass two arguments.
The friend function in C++ can access the private data members of a class directly.
An overloaded operator friend could be declared in either the private or public section of a class.
When redefining the meaning of an operator by operator overloading the friend function, we cannot change its basic meaning. For example, we cannot redefine minus operator + to multiply two operands of a user-defined data type.

Syntax to use Friend Function in C++ to Overload Operators:

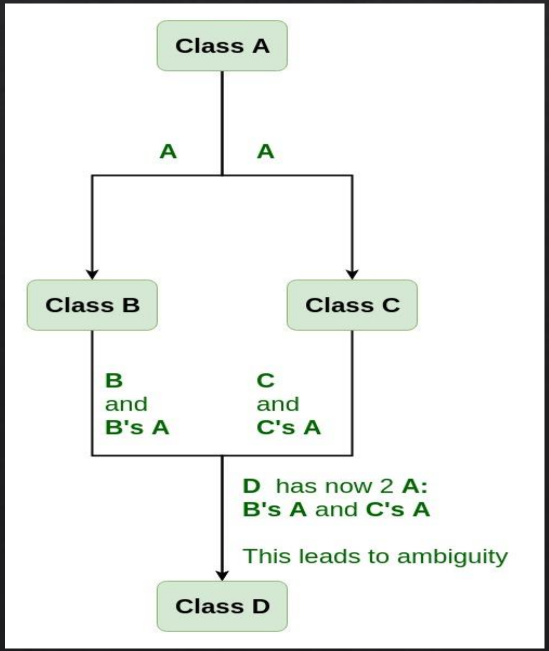
```
Friend return-type operator operator-symbol (Variable1, Variable2)
{
        //statements;
 }
```

VIRTUAL BASE CLASSES

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:
Consider the situation where we have one class A .This class is A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.

As we can see from the figure that data members/function of class A are inherited twice to class D. One through class B and second through class C. When any data / function member of class A is accessed by an object of class D, ambiguity arises as to which data/function member would be called? One inherited through B or the other inherited through C. This confuses compiler and it displays error.



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

class A {
public:
        void show()
        {
                cout << "Hello form A \n";
        }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
        D object;
        object.show();
}
```

How to resolve this issue?
To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as :

Syntax for Virtual Base Classes:

Syntax 1:
```
class B : virtual public A
{
};
```

Syntax 2:
```
class C : public virtual A
{
};
```

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

class A {
public:
        int a;
        A() // constructor
        {
                a = 10;
        }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};
```

```
int main()
{
        D object; // object creation of class d
        cout << "a = " << object.a << endl;

        return 0;
}
```

O/P :

```
a= 10
```

The class A has just one data member a which is public. This class is virtually inherited in class B and class C. Now class B and class C becomes virtual base class and no duplication of data member a is done.

Function Overriding

inheritance

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.

This is known as function overriding in C++. The function in derived class overrides the function in base class.

```
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

In early binding, the compiler matches the function call with the correct function definition at compile time. It is also known as Static Binding or Compile-time Binding.

late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as Dynamic Binding or Runtime Binding.

By default, early binding takes place. So if by any means we tell the compiler to perform late binding

This can be achieved by declaring a virtual function.

VIRTUAL FUNCTION

A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
Virtual Function is a member function of the base class which is overridden in the derived class. The compiler performs late binding on this function.

To make a function virtual, we write the keyword virtual before the function definition.

```
#include <iostream>

using namespace std;

class Animals
{
	public:
		virtual void sound()
		{
			cout << "This is parent class" << endl;
		}
};

class Dogs : public Animals
{
	public:
		void sound()
		{
			cout << "Dogs bark" << endl;
		}
};
```

```
int main()
{
	Animals *a;
	Dogs d;
	a= &d;
	a -> sound();
	return 0;
}
```

O/P :

```
Dogs bark
```

Since the function sound() of the base class is made virtual, the compiler now performs late binding for this function. Now, the function call will be matched to the function definition at runtime. Since the compiler now identifies pointer a as referring to the object 'd' of the derived class Dogs, it will call the sound() function of the class Dogs.

ABSTRACT FUNCTIONS

Pure virtual function is a virtual function which has no definition. Pure virtual functions are also called abstract functions.

To create a pure virtual function, we assign a value 0 to the function as follows.

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

ABSTRACT CLASS

An abstract class is a class whose instances (objects) can't be made. We can only make objects of its subclass (if they are not abstract). Abstract class is also known as abstract base class.

An abstract class has at least one abstract function (pure virtual function).

ABSTRACTION

```
class Shape {
  public:
    // All the functions of both square and rectangle are clubbed together in a single
class.
    void width(int w) {
        shape_width = w;
    }
    void height(int h) {
      shape_height = h;
    }
    int perimeterOfSquare(int s) {
      return 4 * s;
    }
    int perimeterOfRectange(int l, int b) {
      return 2 * (l + b);
    }

  protected:
    int shape_width;
    int shape_height;
};
```

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

// Shape class.
class Shape {
  public:

    // Function to calculate the parameter, declared
as pure virtual, so all the derived classes necessarily
need to implement this.
    virtual int perimeter() = 0;
    void width(int w) {
      shape_width = w;
    }
    void height(int h) {
      shape_height = h;
    }

  protected:
    int shape_width;
    int shape_height;
};
```

```
class Rectangle: public Shape {
  public:
    // Class Rectangle provided implementation of
perimeter() function.
    int perimeter() {
      return (2 * (shape_width + shape_height));
    }
};

class Square: public Shape {
  public:
    // Class Square provided implementation of perimeter()
function.
    int perimeter() {
      return (4 * shape_width);
    }
};
```

```
int main() {
  Rectangle R;
  Square S;

  R.width(10);
  R.height(5);
  S.width(10);

  cout << "Perimeter of Rectangle: " << R.perimeter() << endl;
  cout << "Perimeter of Square: " << S.perimeter() << endl;
  return 0;
}
```

O/P:

```
Perimeter of Rectangle : 30
Perimeter of Square: 40
```

Characteristics of Abstract Class in C++

Abstract Classes must have at least one pure virtual function.

```
virtual int perimeter() = 0;
```

Abstract Classes cannot be instantiated, but pointers and references of Abstract Class types can be created. You cannot create an object of an abstract class. Here is an example of a pointer to an abstract class.

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

class Base {
  public:
    virtual void print() = 0;
};

class Derived: public Base {
  public:
    void print() {
      cout << "This is from the derived class \n";
    }
};
```

```
int main(void) {
  Base* basePointer = new Derived();
  basePointer -> print();
  return 0;
}
```

```
O/P: This is from the derived class
```

Abstract Classes are mainly used for Upcasting, which means its derived classes can use its interface.
Classes that inherit the Abstract Class must implement all pure virtual functions. If they do not, those classes will also be treated as abstract classes.

One interface and multiple methods.

Virtual Destructors

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destruction of the object when the program exits.

```
class Base
{
    public:
    virtual ~Base()
    {
        cout << "Base Destructor\n";
    }
};

class Derived:public Base
{
    public:
    ~Derived()
    {
        cout<< "Derived Destructor";
    }
};
```

```
int main()
{
    Base* b = new Derived;     //
Upcasting
    delete b;
}
```

O/P:

```
Derived Destructor
Base Destructor
```

Generic programming with templates

The advantages of Generic Programming are

Code Reusability
Avoid Function Overloading
Once written it can be used for multiple times and cases.
Generics can be implemented in C++ using Templates

writing code in a way that is independent of any particular type.

You can use templates to define functions as well as classes

The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types.

template function definition is shown here –

```
template <class type> ret-type func-name(parameter list) {
   // body of function
}
```

Generic Function using Templates

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

// One function works for all data types.
template <typename T>

T myMax(T x, T y)
{
	return (x > y) ? x : y;
}

int main()
{
	// Call myMax for int
	cout << myMax<int>(3, 7) << endl;

	// call myMax for double
	cout << myMax<double>(3.0, 7.0) << endl;

	// call myMax for char
	cout << myMax<char>('g', 'e') << endl;
	return 0;
}
```

O/P :

```
7
7
g
```

Generic Class using Template:

class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

```
template <class type> class class-name {
   .
}
```

In generic functions and classes, the type of data upon which the function or class operates is specified as parameter. Thus you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

SYNTAX:

```
Template<class Ttype> ret-type func-name(parameter list)

{
        // body of function
}
```

```
Ex: template <class X> void swapargs(X &a, X &b)
{ X temp;
Temp = a;
 a=b;
b=temp;
}
```

Generic Class

```
template<class X> void swapargs(X &a, X &b)
{
        X temp;

        temp=a;
        a=b;
        b=temp;
}

int main()
{
        int i=10, j=20;
        double x=10.1, y=23.1;
        char a=‘x’, b=‘z’;

        cout<<“original i, j : ”<< i<< ‘ ’<< j << ‘/n’;
        cout<<“original x, y : ”<< x<< ‘ ’<< y << ‘/n’;
        cout<<“original a, b : ”<< a<< ‘ ’<< b<< ‘/n’;
```

```
Swapargs(i, j) ; // swap intergers
Swapargs(x, y) ; // swap float
Swapargs(a, b) ; // swap charactes


        cout<<“swapped  i, j : ”<< i<< ‘ ’<< j << ‘/n’;
        cout<<“swapped x, y : ”<< x<< ‘ ’<< y << ‘/n’;
        cout<<“swapped a, b : ”<< a<< ‘ ’<< b<< ‘/n’;

return 0;
}
```

Template <class X> void swapargs(X &a , X &b)

Template is being created and that a generic definition is beginning. Here X is a generic type that is used as a placeholder. After the template portion, the function swapargs() is declared, using X as the datatype of the value that will be swapped.

In main(), the swapargs() function is called using 3 different types of data: int, double, float.

swapargs() is a generic function , the complier automatically creates 3 versions of swapargs(): one that will exchange integer value, ….

Function Overloading

```
// Function to calculate square
void square(int a)
{
	cout << "Square of " << a
		<< " is " << a * a
		<< endl;
}

// Function to calculate square
void square(double a)
{
	cout << "Square of " << a
		<< " is " << a * a
		<< endl;
}
```

```
// Driver Code
int main()
{
	// Function Call for side as
	// 9 i.e., integer
	square(9);

	// Function Call for side as
	// 2.25 i.e., double
	square(2.25);
	return 0;
}
```

```
o/p: Square of 9 is 81
Square of 2.25 is 5.0625
```

Overloading Function Template

```
// C++ program to illustrate overloading
// of template function using an
// explicit function
#include <bits/stdc++.h>
using namespace std;

// Template declaration
template <class T>

// Template overloading of function
void display(T t1)
{
	cout << "Displaying Template: "
		<< t1 << "\n";
}

// Template overloading of function
void display(int t1)
{
	cout << "Explicitly display: "
		<< t1 << "\n";
}
```

```
// Driver Code
int main()
{
	// Function Call with a
	// different arguments
	display(200);
	display(12.40);
	display('G');

	return 0;
}
```

o/p:

```
Explicitly display: 200
Displaying Template: 12.4
Displaying Template: G
```

Overloading Function Template

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

O/P:

```
11
2.5
```

Generic Classses:

```
template <class Ttype> class class-name {
  .
  .
  .
}
```

```
// Demonstrate a generic queue class.



#include <iostream>
using namespace std;

const int SIZE=100;
// This creates the generic class queue.
template <class QType> class queue {

QType q[SIZE];
 int sloc, rloc;
public:
 queue() {
 sloc = 0;
 rloc = 0;
}
 void qput(QType i);
 QType qget();
};
```

```
// Put an object into the queue.
template <class QType> void queue<QType>::qput(QType i)
{
 if(sloc==SIZE) {
   cout << "Queue is full.\n";
   return;
 }
 sloc++;
 q[sloc] = i;
}

// Get an object from the queue.
template <class QType> QType queue<QType>::qget()
{
 if(rloc == sloc) {
   cout << "Queue Underflow.\n";
   return 0;
 }
 rloc++;

  return q[rloc];
}
```

```
int main()
{
   queue<int> a, b;  // create two integer queues

   a.qput(10);
   b.qput(19);
   a.qput(20);
   b.qput(1);

   cout << a.qget() << " ";
   cout << a.qget() << " ";
   cout << b.qget() << " ";
   cout << b.qget() << "\n";

   queue<double> c, d;  // create two double queues

   c.qput(10.12);
   d.qput(19.45);
   c.qput(-20.0);
   d.qput(0.123);

cout << c.qget() << " ";
   cout << c.qget() << " ";
   cout << d.qget() << " ";
   cout << d.qget() << "\n";

   return 0;
}
```

O/P:

```
10   20   19   1
10.12   -20    19.45   0.123
```

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;    // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element

    bool empty() const {      // return true if
empty.
      return elems.empty();
    }
};
```

```
template <class T>
void Stack<T>::push (T const& elem) {
  // append copy of passed element
  elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
  if (elems.empty()) {
    throw out_of_range("Stack<>::pop(): empty stack");
  }

  // remove last element
  elems.pop_back();
}

template <class T>
T Stack<T>::top () const {
  if (elems.empty()) {
    throw out_of_range("Stack<>::top(): empty stack");
  }

  // return copy of last element
  return elems.back();
}
```

Overloading Function Template: A template function can be overloaded either by a non-template function or using an ordinary function template.

Function Template: The function template has the same syntax as a regular function, but it starts with a keyword template followed by template parameters enclosed inside angular brackets <>.

```
template <class T>

T functionName(T arguments)
{
    // Function definition
    ……….  ……   ….. …….
}
```

where, T is template argument accepting different arguments and class is a keyword.

Template Function Overloading:

The name of the function templates are the same but called with different arguments is known as function template overloading.
If the function template is with the ordinary template, the name of the function remains the same but the number of parameters differs.
When a function template is overloaded with a non-template function, the function name remains the same but the function's arguments are unlike.

In this case, we have used T as the template parameter name, instead of SomeType. It makes no difference, and T is actually a quite common template parameter name for generic types.

In the example above, we used the function template sum twice. The first time with arguments of type int, and the second one with arguments of type double. The compiler has instantiated and then called each time the appropriate version of the function.

Note also how T is also used to declare a local variable of that (generic) type within sum:

```
T result;
```

Therefore, result will be a variable of the same type as the parameters a and b, and as the type returned by the function. In this specific case where the generic type T is used as a parameter for sum, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with:

```
k = sum<int> (i,j);
h = sum<double> (f,g);
```

Inheritance with class templates

Inheriting from a template class is feasible.

If we need the new derived class to be general, we must make it a template class with a template argument sent to the base class. This is because inheritance is only possible with a class, and a template is not a class unless it is instantiated by passing some data type to it.

SYNTAX :

```
template <class T>
class derived : public Base<T>{};
```

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

template<class T>
class Base {
    private:
        T val;

    public:
        void setVal(T a)
        {
            val = a;
        }
};
template<class T>
class Derived : public Base<T> {
    public:
        void setVal (T b)
        {
            Base<T>::setVal(b);
        }
};
```

```
int main()
{
    Derived<int> a;
    a.setVal(4);
    return 0;
}
```

THANK YOU !!