girmiti software™

Introduction to C++

- Function Overloading
- Constructor Overloading
- Operator Overloading
- Default Arguments

Function Overloading

Function Overloading:

Function overloading is the process of using the same name for two or more functions. Function overloading allows us to use the same name for different functions, to perform, either same or different functions in the same class.

Ways to overload the function:

- By changing number of Arguments.
- By having different types of argument
- By changing order of arguments

Advantage:

Function overloading is usually used to enhance the readability of the program.

Function Overloading

Example of Method Overloading by changing the no. of arguments:

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
#include<iostream>
using namespace std;
class Calculation
 {          public:
            void sum(int a,int b);
            void sum(int a,int b,int c);
 };
void Calculation::sum(int a,int b)
{           cout<<a+b<<endl;
}
void Calculation::sum(int a,int b,int c)
{           cout<<a+b+c<<endl;
}
int main()
{           Calculation c;
            c.sum(10,50,30);
            c.sum(20,40);
 }
```

O/P:30
40

Function Overloading

Example of Method Overloading by changing data type of arguments

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two float arguments.

```
#include<iostream>
using namespace std;
class Calculation
 {          public:
            void sum(int a,int b);
            void sum(float a,float b);
 };
void Calculation::sum(int a,int b)
{           cout<<a+b<<endl;
}
void Calculation::sum(float a,float b)
{           cout<<a+b<<endl;
}
int main()
{           Calculation c;
            c.sum(10.9f,90.8f);
            c.sum(20,40);
}O/P:101.7
```

Function Overloading

Example of Method Overloading by changing the order of arguments:

In this example, we have created two overloaded methods that differs in order. The first sum method receives integer & float argument and second sum method receives float & integer argument.

```
#include<iostream>
using namespace std;
class Calculation
 {          public:
            void sum(int a,float b);
            void sum(float a,int b);
 };
void Calculation::sum(int a,float b)
{           cout<<a+b<<endl;
}
void Calculation::sum(float a,int b)
{           cout<<a+b<<endl;
}
int main()
{           Calculation c;
            c.sum(10.5f,90);
            c.sum(20,40.5f);
}
```

O/P:100.5

60.5

Constructor Overloading

Constructor Overloading:

Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example:

```
#include<iostream>
using namespace std;
class Example
{           public:
            Example()
            {           cout<<"Hello"<<endl;
            }
            Example(int j)
            {           for(int k=0;k<j;k++)
                        cout<<k<<endl;
             }
};
            int main()
            {
                        Example l1;
                        Example l2(5);
            }
```

Operator Overloading

Operator Overloading:

Operator overloading is an important concept in C++. We can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

For example '+' operator can be overloaded to perform addition on various data types, like for int, String(concatenation) etc.

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.

Operator that are not overloaded are follows:
scope operator - ::
sizeof
member selector - .
member pointer selector - *
ternary operator - ?:

Syntax:
return_type operator operator_to_be_overloaded(parameters);

Operator Overloading

Overloading Arithmetic Operators:

Arithmetic operators are most commonly used operators in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the + operator,

```
#include<iostream>
using namespace std;
class MyNum
{
private:
    int number;
public:
    MyNum();
   MyNum(int);
    /* Operator */
    MyNum operator +(MyNum);
    void display();
};
MyNum::MyNum()
{
    number=0;
}
MyNum::MyNum(int num)
{
    number=num;
}
```

Operator Overloading

```
MyNum MyNum ::operator +(MyNum N)
{
    MyNum temp;
    temp.number = number+N.number;
    return temp;
}
void MyNum::display()
{
    cout<<number<<endl;
}
int main()
{
    MyNum n1(100);
    MyNum n2(50);
    MyNum n3;
    n3=n1+n2;
    n1.display();
    n2.display();
    n3.display();
}
```

Output:

100

50

150

Operator Overloading

Overloading unary operator:

Unary operators are those that act on a single operand. Unary operator can be classified as follows:

- Simple prefix unary operators,e.g. –(minus operator)
- Pre & post increment & decrement operators

Example of overloading unary operator:

```
#include <iostream>
using namespace std;
class MyInt
{   private :
   int a;
   int b;
   public :
   void operator -(); // member function
   void accept(int, int);
   void print();
};
void MyInt::accept(int x, int y)
{  a=x;
    b=y;
}
```

Operator Overloading

```
void MyInt::operator-()
{ a=-a;
  b=-b;
}
void MyInt::print()
{ cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
}
int main()
{
  MyInt i1;
  i1.accept(15, -25);
  i1.print();
  -i1;
  i1.print();
  return 0;
}
```

Output:15
-25
-15
25

Operator Overloading

Pre & Post increment operators:

When used with fundamental data types,the prefix operator causes the variable to be incremented first before it is used in an expression whereas the postfix causes the value to be incremented after the value is used in an expression.

- An operator function with no argument is invoked by the compiler for the prefix application of the operator.
- The compiler invokes the operator function with an int argument for the postfix application of the operator.

Example:

```
#include<iostream>
using namespace std;
class MyNum
{private:
  int number;
public:
  MyNum();
  MyNum(int);
  MyNum operator ++();//Pre Increment
  MyNum operator ++(int);//Post Increment
  void display();
};
```

Operator Overloading

```
MyNum::MyNum()
{
  number=0;
}
MyNum::MyNum(int num)
{
  number=num;
}
MyNum MyNum ::operator ++() //PreIncrement
{
  MyNum temp;
  number = number + 1;
  temp.number = number;
  return temp;
}
MyNum MyNum ::operator ++(int)        //PostIncrement
{
  MyNum temp;
  temp.number = number;
  number = number + 1;
  return temp;
}
```

Operator Overloading

```
void MyNum::display()
{
  cout<<number<<endl;
}
int main()
{
  MyNum n1(100);
  MyNum n2;
  n2=n1++;  //post increment
  n1.display();
  n2.display();
  n2=++n1;  //pre increment
  n1.display();
  n2.display();
}
```

Output:101
100
102
102

Default arguments

Default Arguments:
One of the most useful facilities available in C++ is the facility to define default argument values for functions. In the function prototype declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default function prototype declaration.
Default arguments facilitate easy development and maintenance of program.

Consider the following function declaration:

```
int my_func(int a, int b, int c=12);
```

This function takes three arguments, of which the last one has a default of twelve. The programmer may call this function in two ways:

```
result = my_func(1, 2, 3);
result = my_func(1, 2);
```

Default arguments

Example:

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
   return (x + y + z + w);
}

int main()
{
   cout << sum(10, 15) << endl;
   cout << sum(10, 15, 25) << endl;
   cout << sum(10, 15, 25, 30) << endl;
}
```

Output:

```
25
50
80
```

Discussions