Lecture - 17

Operator Overloading and Type Conversion

17.1 Overloading operators

Operator overloading is one of th many exciting features of C++ language. C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

```
int a, b, c;
a = b + c;
```

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, *it is not so obvious that we could perform an operation similar to the following one:*

```
struct {
  string product;
  float price;
} a, b, c;
a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the *behavior* our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

Overload able operators
+ - * / = < > += -= *= /= << >> <<= >>= == != <= >= ++ -- % & ^ ! | ~ &= ^= |= && || %= [] () , ->* -> new delete new[] delete[]

To overload an operator in order to use it with classes we declare ***operator*** *functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

type operator sign (parameters) { /*...*/ }

Here you have an example that overloads the addition operator (+). We are going to create a class to store bi-dimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bi-dimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).

Program **17.1: Operator overloading**

```
// vectors: overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
/*OUTPUT
4,3
*/
```

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int);// function name CVector (constructor)
CVector operator+ (CVector);// function returns a CVector
```

The function operator+ of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name**:**

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector(){ };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

```
CVector c;
```

included in main() would not have been valid.

Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code have not been included in the example.

Following example explains operator overloading

Program 16.9: Overloaded constructors

//overloaded constructors**.**

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

class complex
{
     float x,y;
public:
     complex(){}      //Constructor with no argument
     complex(float a){x=y=a;}  //Constructor with one
argument
     complex(float real, float imag){x=real, y=imag;}
//Constructor //with two argument

     friend complex sum(complex, complex);
     friend void show(complex);
};
complex sum(complex c1, complex c2)    //friend
{
     complex c3;
    c3.x=c1.x+c2.x;
     c3.y=c1.y+c2.y;
     return (c3);
}
void show(complex c)   //friend
{
      cout<<c.x<<"+j"<<c.y<<"\n";
}
int main()
{
     complex A(2.7, 3.3);   //define & initialise
     complex B(1.6);        //define & initialise
    complex C;        //define

     C=sum(A,B);       // sum() is a friend
     cout<<"A="; show(A);// show() is also a friend
    cout<<"B="; show(B);
     cout<<"C="; show(C);

     //Another way to give initial values (second method)
     complex P,Q,R;     //define P,Q,R
     P=complex(2.5,3.9);
```

```
        Q=complex(1.6,2.5);
        R=sum(P,Q);

        cout<<"\n";
        cout<<"P="; show(P);
        cout<<"Q="; show(Q);
        cout<<"R="; show(R);
        return 0;
}
OUTPUT/*
A=2.7+j3.3
B=1.6+j1.6
C=4.3+j4.9

P=2.5+j3.9
Q=1.6+j2.5
R=4.1+j6.4

Press any key to continue . . .*/
```

There are three constructors in the class complex. The first constructor, which takes no arguments, is used to create arguments which are not initialised; the second, which takes one argument, is used to to create objects and initialise them; and the third, which takes two arguments, is also used to create objects and initialize them to specific values.

The first constructor *complex(){ }* contains the empty body and does not do anything. This is used to create objects without any initial values. It is known that C++ compiler has an *implicit constructor* which creates objects , even though it was not defined in the class.

This works fine as long as we do not use any other constructor in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor.

17.2 Union and classes.

Union is similar to structure data type with a difference in the way the data is stored and retrieved. The union stores values of different types in a single location. If the new assignment is made, the previous value is erased. Member functions of union by default are public. Program 17.2 illustrates the properties of union.

Program 17.2: Example of union. **#include <iostream>**

using namespace std;
void main (void)

```
{
union value {
              int i;
             float j;
);
union value x;
x.i =10;
x.f =-1456.45;
cout<<"first member="<<x.i<<endl;
cout<<"second member =" <<x.f<<endl;
}
Output of the program is:
First member = 3686
Second member = -1456.45
```

In the above program , the union consists of two data members i.e. int and float. Only the float values are stored and displayed correctly, integer value is displayed wrongly. The union only holds a value for one data type which requires a large storage among their members.

17.3 Anonymous union

A union with union_name is called anonymous union . Following example illustrates anonymous union.

Program 17.3 : Example of Anonymous union

```
union{
          int x;
          float y;
         char z;
};
```

17.4 Nested classes.

C++ permits declaration of a class within another class. A class declared as a member of another class is called a nested class or a class within another class.

The general syntax of the nested class declaration is shown below.

```
class outer-class_name{
     Private:
        //data
    protected:
        //data
   public:
      //method
class inner_class_name{
      private:
         //data of inner class
      public:
           //method of inner class
} ;//end of inner class declaration
};//end of outer class declaration
      outer_class_name object1;
      outer_class_name::inner_class_name object2;
```

17.5 Static Data Members: An important storage class specifier in C++ is static. Normally, when you call a function, *all its local variables are reinitialized each time the function is called*. This means that their values change between function calls. **Static variables, however, maintain their value between function calls**.

Every global variable is defined as static automatically. (Roughly speaking, functions anywhere in a program can refer to a global variable; in contrast, a function can only refer to a local variable that is "nearby", where "nearby" is defined in a specific manner.

In C++ a data member of a class can be qualified as static. A static member variable has certain special characteristics as:

- It is initiated to zero when first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example a static data member can be use as **counter** that records the occurrence of all the objects. Following program illustrates the use of static data membe**r.**

Program 17.4: Static data member.

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

class item
{
        static int count;
        int number;
public:
        void getdata(int a)
        {
                number=a;
                count++;
        }
        void getcount(void)
        {
                cout<<"count:";
                cout <<count<<"\n";
   }

};

int item :: count;

int main()
{
        item a, b, c; //count is initialized to zero
        a.getcount();  //display count
        b.getcount();
        c.getcount();

        a.getdata(100);  //getting data into object a
        b.getdata(200);
        c.getdata(300);

        cout<<"After reading data " <<"\n";
   a.getcount();  //display count
   b.getcount();
   c.getcount();

        return  0;
}
/*OUTPUT
count:0
count:0
count:0
After reading data
count:3
count:3
count:3
*/
```

Note that scope and type of each **static** member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as *class variables*.

The **static** variable **count** is initialized to zero when objects are created. The account is incremented whenever the data is read into an object. Since the data is read into object three times, the variable count is incremented three times. Since there is only one copy of the count shared by all the three objects, all the three output statements cause the value 3 to be displayed.

17.6 Dynamic Allocation Memory :

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is *dynamic memory*, for which C++ integrates the operators new and delete.

Operators new and new[ ]

In order to request dynamic memory we use the 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. Its form is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

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

In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type int.

The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression *bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with bobby[1] or *(bobby+1) and so on.

Then what is the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the default method used by new, and is the one used in a declaration like:

```
bobby = new int [5];  // if it fails an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution.

This method can be specified by using a special object called nothrow as parameter for new:

```
bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if bobby took a null pointer value:

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
  // error assigning memory. Take measures.
};
```

This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in coming examples due to its simplicity.

Operator delete and delete[ ]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

Program 17.4 : Demonstration of run time memory allocation

```
// rememb-o-matic
#include <iostream>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you
like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
    cout << "Error: memory could not be
allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
```

```
How many numbers would you like
to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436,
1067, 8, 32,
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

Notice how the value within brackets in the `new` statement is a variable value entered by the user (i), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment. Following program illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization

Program 17.1: Operator overloading

```
//Long term fixed deposit scheme using dynamic initialisation
#include <iostream>
using namespace std;

class Fixed_deposit
{
        long int P_amount; //Principal amount
        int Years; //period of investment
        float rate;  //Interest rate
        float R_value;  //Return value of amount
public:
        Fixed_deposit(){}
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount=p;
        Years=y;
        rate=r;
        R_value=P_amount;
        for(int i=1; i<=y; i++)
                R_value=R_value*(1.0+r);
}

Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
        P_amount=p;
        Years=y;
        rate=r;
        R_value=P_amount;
        for(int i=1; i<=y; i++)
                R_value=R_value*(1.0+float(r)/100);
}
void Fixed_deposit::display(void)
{
        cout<<"\n"
                <<"Principal Amount="<<P_amount<<"\n"
<<"return value  =" <<R_value<<"\n";
}

int main( )
{
        Fixed_deposit FD1, FD2, FD3;  //deposits created
        long int p;    //principal amount
        int y;      //interest period, years
        float  r;   // interest rate, decimal form
        int R;  // interest rate, percent form

        cout<<"enter amount, period, interest rate (in percent)"<<"\n";
        cin>> p>>y>>R;
        FD1= Fixed_deposit(p,y,R);

  cout<<"enter amount, period, interest rate (decimal form)"<<"\n";
        cin>> p>>y>>r;
        FD2= Fixed_deposit(p,y,r);
```

```
cout<<"enter amount, period, "<<"\n";
      cin>> p>>y;
      FD3= Fixed_deposit(p,y);

      cout<<"\ndeposit  1";
      FD1.display();

      cout<<"\ndeposit  2";
      FD2.display();

      cout<<"\ndeposit  3";
      FD3.display();

      return 0;

    }
```

OUTPUT

```
/*  Enter amount, period, interest rate (in percent)
10000 3 18
enter amount, period, interest rate (decimal form)
10000 3 0.18
enter amount, period,
10000 3 18

deposit  1

Principa amount = 10000
Return Value = 16430.00

deposit  2
Principa amount = 10000
Return Value = 16430.00

deposit  3
Principa amount = 10000
Return Value = 14449.3*/
```