Advanced Operator Overloading

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

Quick Check

How to overload the + operator so that the following code works?

```
int main()
{
    Point p1(10, 10);

    Point p2 = 5 + p1;

    p2.display();          //p2 = (15, 15)
}
```

Solution

We must declare the operator+ function as non-member in this case.

```
Point operator+(int a, Point p)
{
    int x_ = a + p.x;
    int y_ = a + p.y;
    Point ret(x_, y_);
    return ret;
}
```

Overloading the () operator

- Enabling the object to act like a function

```
obj1(param1, param2,...)
```

- Must be a member function of the class
- It can have any number of parameters and any return type
- The object works like a programmable function

Overloading the () operator

Example:

Suppose you are to required to calculate the value of y for the following line equations, where the values of x are from 1 to 5.

$$y = 4x + 3$$

$$y = 7x - 2$$

$$y = 2x + 5$$

Will you write three separate functions?

Overloading the () operator

```
class LineEquation
{
    int m;
    int c;
public:
    LineEquation(int a, int b)
    {
        m = a;
        c = b;
    }

    int operator()(int x)
    {
        return m * x + c;
    }
};
```

y = 4x + 3

y = 7x – 2

y = 2x + 5

Overloading the () operator

$$y = 4x + 3$$

$$y = 7x - 2$$

$$y = 2x + 5$$

```
int main()
{
    LineEquation line1(4, 3);
    LineEquation line2(7, -2);
    LineEquation line3(2, 5);

    cout << "Points of line1:" <<endl;
    for (int i = 1; i<5; i++)
    {
        cout << "(" << i << ", " << line1(i) <<  ")" << endl;
    }

    //similar for line2 and line3
}
```

Functor (Function Object)

Yes, you've read it right. Functor.

- Functor is a C++ class that acts like function.
- It's a class where operator () is defined.
- line1, line2, line3 in the previous example are Functors.

Task

Design an Account class that will hold the profit rate and bank balance. If the number of year is give, it will return how much money the account will have after that year including profit.

profit = rate x balance x year

Sample code:

```
Account a1(1000, .2);
double profit = a1(3);
printf(“%lf”, profit); //1000 + 600 = 1600
```

Functor vs Function Pointer

- Functors are more efficient than Function Pointer. Function pointer may require runtime pointer dereferencing.
- Functor can contain state

Conversion Function

When we want to convert an object of one type to another

Conversion Function

When we want to convert an object of one type to another

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};
```

Conversion Function

When we want to convert an object of one type to another

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }
};
```

```
int main()
{
    Subject cse205(80, 80);


}
```

Conversion Function

When we want to convert an object of one type to another

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;


}
```

Conversion Function

When we want to convert an object of one type to another

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks;     //160
}
```

Conversion Function

Syntax: `operator type() { return value; }`

```
class Subject
{
    int partI;
    int partII;
public:
    Subject(int p1, int p2)
    {
        partI = p1;
        partII = p2;
    }

    operator int()
    {
        return partI + partII;
    }
};
```

```
int main()
{
    Subject cse205(80, 80);

    int total_marks = cse205;

    cout << total_marks;     //160
}
```

Conversion Function

Syntax: `operator type() { return value; }`

- operator and return are keywords
- *type* is the target type we'll be converting our object to
- *value* is the value of the object after the conversion has been performed.

- Returns a value of type *type*
- No parameter can be specified
- Conversion function must be a member function

Task

Design a Fraction class (a, b). If it’s cast into float or double, the divided value is stored in the variable.

```
Fraction f1(3, 4);
double val = f1;
printf(“%lf”, val);
```

Overloading new and delete

```
void * operator new (size_t count);
void operator delete (void * ptr);


Sample Code:
Point * ptr = new Point; //What will be passed?
//...
delete ptr;
```

Overloading new and delete

```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }
    void * operator new (size_t sz)
    {
        cout << "mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }
    void operator delete (void * p)
    {
        cout << "mem deallocated" <<endl;
        free(p);
    }

};
```

```
int main()
{
    Point * pt = new Point(1,2);
    pt->display();
}
```

Difference between delete and delete []

- delete calls destructor of a single object
- delete [] calls destructor for an array of objects

- If delete [] is called for an object created with new, the behavior is undefined.

Overloading `new []` and `delete []`

```
void * operator new [] (size_t count);

void operator delete [] (void * ptr);
```

Overloading new [] and delete []

```
class Point
{
    int x;
    int y;
public:
    Point(int _x, int _y)
    {
        x = _x;
        y = _y;
    }

    void * operator new (size_t sz)
    {
        cout << "mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }

    void * operator new [] (size_t sz)
    {
        cout << "array mem allocated" <<endl;
        void * p = malloc(sz);
        return p;
    }
```

Overloading new [] and delete []

Cont...

```
    void operator delete (void * p)
    {
        cout << "mem deallocated" <<endl;
        free(p);
    }

    void operator delete [](void * p)
    {
        cout << "array mem deallocated" <<endl;
        free(p);
    }

    void display()
    {
        cout << "(" << x << ", " << y << ")" <<endl;
    }
};

int main()
{
    Point * pt = new Point[2] {Point(1,2), Point(3,4)};
    pt[0].display();
    pt[1].display();
    delete [] pt;
}
```

How does compiler know the amount of memory to deallocate?

From C++14 the following prototype is supported.

```
void operator delete(void *p, size_t size);
```

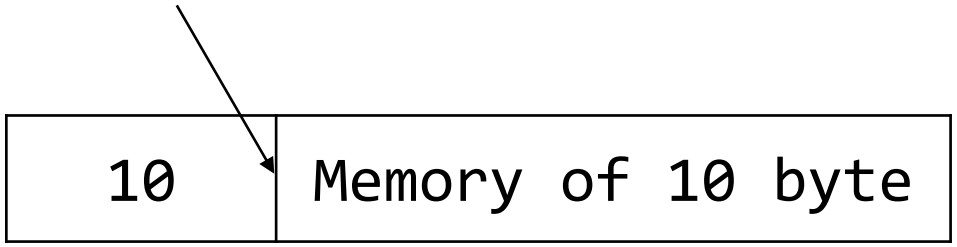
However, old compilers had to use a few tricks to save the amount of memory

Source: https://stackoverflow.com/questions/2571265/overloading-delete-and-retrieving-size

How does compiler know the amount of memory to deallocate?

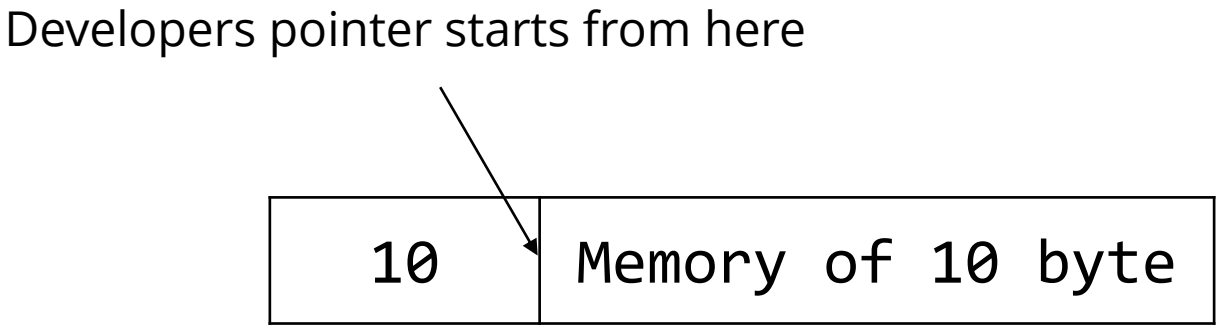
```
void *allocate(size_t size) {
    size_t *p = malloc(sizeof(size_t) + size);
    p[0] = size;            // store the size in the first few bytes
    return (void*)(&p[1]); // return the memory just after the size
                           // we stored
}
```

Developers pointer starts from here



Source: https://stackoverflow.com/questions/2571265/overloading-delete-and-retrieving-size

How does compiler know the amount of memory to deallocate?




```
void deallocate(void *ptr) {
    size_t *p = (size_t*)ptr; // make the pointer the right type
    size_t size = p[-1];      // get the data we stored at the
                              // beginning of this block

    // do what you need with size here...

    void *p2 = (void*)(&p[-1]); // get a pointer to the memory we
                                // originally really allocated
    free(p2);                   // free it
}
```

Source: https://stackoverflow.com/questions/2571265/overloading-delete-and-retrieving-size