

Informatics 3

Kristóf Kovács

2024-05-20

1. Operator overload 2

1.1. Reference reminder

```
int a = 5;
int& b = a; // must specify what is referenced when creating a
b = 3;
cout << a; // print 3
```

In addition, we only used a reference to trick the copy constructor (because copying here would cause infinite recursion):

```
Complex(Complex& other);
```

But this is not all uses of references. With references, we can use a function as a left value:

```
int main(void) {
    int a[] = {1, 2, 5};
    f(a, 3) = 7; // this will change one of the elements of the array
    for(int i = 0; i < 3; i++) {
        cout << a[i] << endl;
    }
    return 0;
}
```

Sounds strange at first, but if you look at the definition of the f function it makes sense:

```

int& f(int* a, int n) {
    int min = 0;
    for(int i = 1; i < n; i++) {
        if(a[min] > a[i]) {
            min = i;
        }
    }
    return a[min];
}

```

This function finds the smallest number in the given array and returns its reference. This way, if it is used as a left value, we can modify the element found. So the function **main** above prints 725.

1.2. operator[]

Let's go back to the operator overload topic. Write a simple class **Vector** representing a 3-dimensional vector:

```

#include <iostream>

using namespace std;

class Vector {
private:
    float v[3];
public:
    Vector();
    Vector(float x, float y, float z);
    Vector(Vector& other);
    Vector operator+(Vector other);
    friend ostream& operator<<(ostream& os, Vector v);
    float& operator[](int index);
    ~Vector();
};

Vector::Vector() {
    v[0] = 0; v[1] = 0; v[2] = 0;
}

```

```

Vector::Vector(float x, float y, float z) {
    v[0] = x; v[1] = y; v[2] = z;
}

Vector::Vector(Vector& other) {
    v[0] = other.v[0];
    v[1] = other.v[1];
    v[2] = other.v[2];
}

Vector Vector::operator+(Vector other) {
    float x = this->v[0] + other.v[0];
    float y = this->v[1] + other.v[1];
    float z = this->v[2] + other.v[2];
    return Vector(x, y, z);
}

ostream& operator<<(ostream& os, Vector v) {
    os << "(" << v[0] << ", ";
    os << v[1] << ", ";
    os << v[2] << ")";
    return os;
}

float& Vector::operator[](int index) {
    return v[index];
}

Vector::~Vector() {}

int main(void) {
    Vector v1;
    Vector v2(1,1,1);
    Vector v3(5, -1, 2);
    Vector v4 = v2 + v3;
    cout << v4 << endl; // (6, 0, 3)
}

```

```

    cout << v4[0] << endl;    // 6
    v4[0] = 10;
    cout << v4 << endl;    // (10, 0, 3)
    return 0;
}

```

Recall that you can actually think of operator overload as a function call. These 2 lines are equivalent (from the previous chapter):

```

Complex e = c + d;
Complex e = c.operator+(d);

```

The same is true then for `operator[]`, they are equivalent:

```

v4[0] = 10;
v4.operator[](0) = 10;

```

This is exactly the same use of function as a left value as at the beginning of this chapter. So that's why you need to return a reference in `operator[]` if you want to have an indexed element as a left value.

1.3. operator=

The `operator=` is the value assignment operator in case the object we are trying to assign a value to already exists. In other words, it is the value assignment that does not occur at the definition of the variable (because then the copy constructor is called):

```

Vector v1;
Vector v2 = v1; // copy constructor
Vector v3;
v3 = v1; // operator=

```

In the above example, all that really happens is that `v2` is created when we equate it with `v1`, so then we need a constructor call (construct a new object from an existing one). Whereas for `v3`, the object already exists (created with a default constructor) when we equate it with `v1`, so `operator=` is called.

The implementation of `operator=` is very similar to the copy constructor, but there are two new things to note. What if there are dynamically

contained values in the object? Since the object already exists, we potentially need to free them up if we need to include new values in their place. Let's look at the implementation of the **Vector** class a little differently if we need to specify the dimension at creation time:

```
#include <iostream>

using namespace std;

class Vector {
private:
    float* v;
    int dim;
public:
    Vector(int d);
    Vector(Vector& other);
    Vector operator=(Vector other);
    Vector operator+(Vector other);
    friend ostream& operator<<(ostream& os, Vector v);
    float& operator[](int index);
    ~Vector();
};

Vector::Vector(int d) {
    v = new float[d];
    dim = d;
    for(int i = 0; i < dim; i++) {
        v[i] = 0;
    }
}

Vector::Vector(Vector& other) {
    this->dim = other.dim;
    this->v = new float[this->dim];
    for(int i = 0; i < this->dim; i++) {
        this->v[i] = other.v[i];
    }
}
```

```

Vector Vector::operator=(Vector other) {
    if(this == &other) {
        return *this;
    }
    delete[] this->v;
    this->dim = other.dim;
    this->v = new float[this->dim];
    for(int i = 0; i < this->dim; i++) {
        this->v[i] = other.v[i];
    }
    return *this;
}

Vector Vector::operator+(Vector other) {
    Vector retval = Vector(this->dim);
    for(int i = 0; i < this->dim; i++) {
        retval[i] = this->v[i] + other.v[i];
    }
    return retval;
}

ostream& operator<<(ostream& os, Vector v) {
    os << "(";
    for(int i = 0; i < v.dim - 1; i++) {
        os << v[i] << ", ";
    }
    os << v[v.dim - 1] << ")";
    return os;
}

float& Vector::operator[](int index) {
    return v[index];
}

Vector::~~Vector() {
    delete[] v;
}

```

```

int main(void) {
    Vector v1(3);
    Vector v2(3);
    Vector v3(3);
    v3[0] = 5; v3[1] = -1; v3[2] = 3;
    Vector v4 = v2 + v3;
    cout << v4 << endl; // (5, -1, 3)
    cout << v4[0] << endl; // 5
    v4[0] = 10;
    cout << v4 << endl; // (10, -1, 3)
    v2 = v4;
    cout << v2 << endl; // (10, -1, 3)
    return 0;
}

```

Let's take a closer look at `operator=`:

```

Vector Vector::operator=(Vector other) {
    if(this == &other) {
        return *this;
    }
    delete[] this->v;
    this->dim = other.dim;
    this->v = new float[this->dim];
    for(int i = 0; i < this->dim; i++) {
        this->v[i] = other.v[i];
    }
    return *this;
}

```

At the beginning, the condition protects the case where we write `v1 = v1` in which case we don't want to delete anything, since we don't really have anything to do.

The other interesting thing is that we return `*this`. It might even be strange that there is a return value. That's because in C you can use the `=` operator in a chain:

```

v2 = v4 = v3;

```

This command happens in this order:

```
v2 = (v4 = v3);
```

So the first call to **operator=** must have a return value, otherwise the second one would not be able to work from what.

2. Inheritance

Suppose we need to define many general 2-dimensional shapes. Our current solution is:

```
class Circle {
private:
    float x, y;
    float r;
public:
    Circle();
    Circle(float x, float y, float r);
    float area();
};

class Rectangle {
private:
    float x, y;
    float a, b;
public:
    Rectangle();
    Rectangle(float x, float y, float a, float b);
    float area();
};

class Square {
private:
    float x, y;
    float a;
public:
    Square();
```

```

    Square(float x, float y, float a);
    float area();
};

```

You can feel that there is a lot of repetition here. Less between Circle and Rectangle, but Square and Rectangle are almost identical. Their area method, if implemented, would be very similar.

This and similar problems can be solved with inheritance.

2.1. Simple inheritance

Let's first solve for Square to be a special Rectangle (with the two side lengths equal).

```

class Rectangle {
private:
    float x, y;
    float a, b;
public:
    Rectangle();
    Rectangle(float x, float y, float a, float b);
    float area();
};

class Square : public Rectangle {
public:
    Square();
    Square(float x, float y, float a);
};

```

So for the inheritance, we need to specify which class we are inheriting from and what level of accessibility. By far the most common is public inheritance, in which case all permissions on data members and methods will be the same in the descendant class as in the ancestor class.

However, this does not mean that x, y in Square will be private. The x, y will belong to the Rectangle part of the Square class and will be private within it. You cannot access the private data members of the ancestor class through Square, even if you inherited public.

2.2. Initializer list

One way to solve the above problem is to use an initialization list. An example of this without inheritance:

```
class Circle {
private:
    float x, y;
    float r;
public:
    Circle();
    Circle(float x_0, float y_0, float rad) : x(x_0), y(y_0), r(rad) {}
    float area();
};
```

So in constructors, we can pre-populate our data members with values. This can only be done in the constructor definition, not in the declaration. As we can see, our constructor block is empty because we have already given values to the data members, so we have nothing left to do.

It's worth noting here that the initialization list uses the copy constructor of objects by default, so if one of the data members in the example above was a `std::string` or the `String` class we wrote in practice, the initialization list would copy it using the copy constructor. Whereas if we copied it inside the block of the constructor, it would use `operator=` there, because by then the objects must already exist. (In this case, the objects are created with their default constructors.)

Now let's look at how this helps with inheritance:

```
class Rectangle {
private:
    float x, y;
    float a, b;
public:
    Rectangle();
    Rectangle(float x, float y, float a, float b);
    float area();
};

class Square : public Rectangle {
```

```

public:
    Square();
    Square(float x, float y, float a) : Rectangle(x, y, a, a) {}
};

```

So, in the initialization list of the descendant classes, we can call the constructor of the ancestor class, which can set the data members associated with it. This works because the constructor of `Rectangle` is public, so the x, y values are not set by `Square`, but by `Rectangle`, which has access.

Another observation is that we did not write the `area` method for `Square`. This is also inherited from `Rectangle`. All data members and methods are inherited automatically. Since `area` was public and we inherited it as public, it remains public in `Square`. If the implementation of `area` in `Rectangle` is similar to the following, then it will be a good algorithm in `Square`, since the only difference is that we give the same value to both side lengths:

```

float Rectangle::area() {
    return a * b;
}

```

2.3. protected access level

Although the above solution solved the basic problem, it can still be felt that there would be problems when we want to access the ancestor class' data members from the descendant, but we don't want to make them public.

The solution to this is the **protected** keyword. This is a similar access level to `private`, except that it makes them accessible to descendants.

A korábbi kód `protected-el`:

```

class Rectangle {
protected:
    float x, y;
    float a, b;
public:
    Rectangle();
    Rectangle(float x, float y, float a, float b);
    float area();
};

```

```

class Square : public Rectangle {
public:
    Square();
    Square(float x, float y, float a) {
        this->x = x;
        this->y = y;
        this->a = a;
        this->b = a;
    }
};

```

Of course, you can use an initialization list in this case too, I just didn't use it for the sake of example.

What we did here with the Rectangle -> Square inheritance was what is called a restrictive inheritance. Because we didn't add new features to the ancestor class, we restricted its existing properties.

Let's look at an example of inheritance where we add features to our class. For example, we could make our Circle, Rectangle, Square classes all inherit from a Shape ancestor that only stores one point:

```

#include <iostream>

using namespace std;

class Shape {
protected:
    float x, y;
public:
    Shape(float x, float y) : x(x), y(y) {}
};

class Circle : public Shape {
protected:
    float r;
public:
    Circle() : Shape(0, 0), r(0) {}
    Circle(float x, float y, float r) : Shape(x, y), r(r) {}
    float area();
};

```

```

float Circle::area() {
    return r * r * 3.14;
}

class Rectangle : public Shape {
protected:
    float a, b;
public:
    Rectangle() : Shape(0, 0), a(0), b(0) {}
    Rectangle(float x, float y, float a, float b) : Shape(x, y), a(a), b(b) {}
    float area();
};

float Rectangle::area() {
    return a * b;
}

class Square : public Rectangle {
public:
    Square() : Rectangle() {}
    Square(float x, float y, float a) : Rectangle(x, y, a, a) {}
};

int main(void) {
    Square s = Square(1, 1, 2);
    cout << s.area() << endl;
    return 0;
}

```

Thus x, y is part of Shape and all shapes are derived from it. The significance of this will be given in the next topic.

3. Heterogen collection

You can refer to a descendant class with the pointer of the ancestor class:

```

int main(void) {
    Shape* s = new Circle(0, 0, 1);
    Rectangle* r = new Square(1, 1, 2);
    Shape* s2 = new Square(-1, -1, 2);
    return 0;
}

```

When used in an array or some kind of container, this is called a heterogeneous collection:

```

int main(void) {
    Shape* s[5];
    s[0] = new Shape(0, 0);
    s[1] = new Circle(0, 0, 1);
    s[2] = new Rectangle(0, 0, 1, 2);
    s[3] = new Square(0, 0, 2);
    s[4] = new Rectangle(0, 0, 2, 3);
    for(int i = 0; i < 5; i++) {
        delete s[i];
    }
    return 0;
}

```

The strength of this is not just that you can store different types of things in one container. It is also that everything that is available in the parent class whose pointer is used to refer to objects is available through the pointer. For example, if you write `area` in the `Shape` class, you can use it through the `Shape` pointer:

```

class Shape {
protected:
    float x, y;
public:
    Shape(float x, float y) : x(x), y(y) {}
    float area();
};

float Shape::area() {
    return 0;
}

```

```

}
...
int main(void) {
    Shape* s[5];
    s[0] = new Shape(0, 0);
    s[1] = new Circle(0, 0, 1);
    s[2] = new Rectangle(0, 0, 1, 2);
    s[3] = new Square(0, 0, 2);
    s[4] = new Rectangle(0, 0, 2, 3);
    for(int i = 0; i < 5; i++) {
        cout << s[i]->area() << endl;
    }
    for(int i = 0; i < 5; i++) {
        delete s[i];
    }
    return 0;
}

```

However, if you run this, you will see that it only prints 0s. Because it used the Shape area method everywhere. That's not good.

3.1. virtual

The keyword **virtual** helps to solve the previous problem:

```

class Shape {
protected:
    float x, y;
public:
    Shape(float x, float y) : x(x), y(y) {}
    virtual float area();
};

float Shape::area() {
    return 0;
}

```

If we run our previous main function this way, we now execute the corresponding area methods on each object. So the virtual keyword means

that when we access such an object via pointer, we run the corresponding descendant's method.

3.2. Abstract class, purely virtual function

Virtual is really a pretty powerful tool, but you can feel that the Shape class is a bit out of line. After all, its `area` method doesn't compute anything, it doesn't really make sense.

We can make a virtual method purely virtual and thus make its class abstract:

```
class Shape {
protected:
    float x, y;
public:
    Shape(float x, float y) : x(x), y(y) {}
    virtual float area() = 0;
};
```

This leads to two things. Firstly, because part of the class is missing (the `area` method is never implemented in Shape), we cannot create Shape objects. On a similar reasoning, all classes derived from Shape must implement the `area` method, otherwise they would be abstract.

The final code:

```
#include <iostream>

using namespace std;

class Shape {
protected:
    float x, y;
public:
    Shape(float x, float y) : x(x), y(y) {}
    virtual float area() = 0;
};

float Shape::area() {
    return 0;
}
```

```

}

class Circle : public Shape {
protected:
    float r;
public:
    Circle() : Shape(0, 0), r(0) {}
    Circle(float x, float y, float r) : Shape(x, y), r(r) {}
    float area();
};

float Circle::area() {
    return r * r * 3.14;
}

class Rectangle : public Shape {
protected:
    float a, b;
public:
    Rectangle() : Shape(0, 0), a(0), b(0) {}
    Rectangle(float x, float y, float a, float b) : Shape(x, y), a(a), b(b) {}
    float area();
};

float Rectangle::area() {
    return a * b;
}

class Square : public Rectangle {
public:
    Square() : Rectangle() {}
    Square(float x, float y, float a) : Rectangle(x, y, a, a) {}
};

int main(void) {
    Shape* s[5];
    s[0] = new Circle(0, 0, 2);
    s[1] = new Circle(0, 0, 1);

```

```

s[2] = new Rectangle(0, 0, 1, 2);
s[3] = new Square(0, 0, 2);
s[4] = new Rectangle(0, 0, 2, 3);
for(int i = 0; i < 5; i++) {
    cout << s[i]->area() << endl;
}
for(int i = 0; i < 5; i++) {
    delete s[i];
}
return 0;
}

```

3.3. virtual destructor

Not only methods can be virtual, for example destructors can be made virtual and if inheritance is an option, it is always worth making the destructor virtual. But why?

Let's look at a simple game example:

```

#include <iostream>

using namespace std;

class A {
protected:
    float x;
public:
    A(float x) : x(x) {}
    ~A() {
        cout << "A destructor" << endl;
    }
};

class B : public A {
protected:
    float y;
public:
    B(float x, float y) : A(x), y(y) {}
}

```

```

    ~B() {
        cout << "B destructor" << endl;
    }
};

int main(void) {
    A* a[2];
    a[0] = new A(5);
    a[1] = new B(3, 2);
    delete a[0];
    delete a[1];
    return 0;
}

```

If we run this, we can see that it calls the destructor of class A twice, even though it should actually be destructing an object B. But when can this cause a problem?

Let's modify the code a bit, give class B a dynamically created array:

```

class B : public A {
protected:
    float* y;
public:
    B(float x, int y) : A(x) {
        this->y = new float[y];
    }
    ~B() {
        cout << "B destructor" << endl;
        delete[] y;
    }
};

```

Thus, if the destructor of class A is not virtual, we cannot free the dynamically reserved array *y* in an object B if it is stored via pointer A. So you should always make your destructor virtual, because it won't cause problems, but its absence can cause big problems.

The correct code:

```

#include <iostream>

```

```

using namespace std;

class A {
protected:
    float x;
public:
    A(float x) : x(x) {}
    virtual ~A() {
        cout << "A destructor" << endl;
    }
};

class B : public A {
protected:
    float* y;
public:
    B(float x, int y) : A(x) {
        this->y = new float[y];
    }
    ~B() {
        cout << "B destructor" << endl;
        delete[] y;
    }
};

int main(void) {
    A* a[2];
    a[0] = new A(5);
    a[1] = new B(3, 2);
    delete a[0];
    delete a[1];
    return 0;
}

```

If you run this, you can see that not only the destructor of B is executed when object B is destroyed, but also the destructor of A. This is because part of each object B is stored as object A (the ancestor part).

3.4. Multiple inheritance

I will only mention this subject, we didn't have time to speak about it in length. We can inherit from two classes. Then the descendant class gets the data members and methods of both ancestors:

```
#include <iostream>

using namespace std;

class A {
protected:
    float x;
public:
    A(float x) : x(x) {}
};

class B {
protected:
    float y;
public:
    B(float y) : y(y) {}
};

class C : public A, public B {
protected:
    float z;
public:
    C(float x, float y, float z) : A(x), B(y), z(z) {}
};

int main(void) {
    C c = C(1,2,3);
    return 0;
}
```

This brings a lot of potential problems that we haven't had time to go into, but you can read about them.

3.5. Interface

Also only at the mention level, but there is a way to be 100% safe with multiple inheritance. To do this, you need to create a class that on the one hand has all its functions purely virtual, and on the other hand contains no data members, only methods. Such a class is called an interface:

```
#include <iostream>

using namespace std;

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

class B {
protected:
    float y;
public:
    B(float y) : y(y) {}
};

class C : public A, public B {
protected:
    float z;
public:
    C(float y, float z) : B(y), z(z) {}
    void print() {
        cout << y << " " << z << endl;
    }
};

int main(void) {
    C c = C(2,3);
    c.print();
    return 0;
}
```

4. Template

Let's return to the linked list subject for a moment. We have created a container in which we could place any number of items. We could now even implement this in a class and implement operator[], we could make it very convenient to use.

But there would be one big problem. The type of object it can store would be a part of the class. So if we needed a list to store integers and a list to store floats, with our current knowledge, there would be a lot of repeated code in the best solution.

To solve this, we can use **template**. The keyword **template** can be used to generalize algorithms based on types. That is, to make a piece of code type-independent.

4.1. Template function

A simple example of a template function that swaps the values of variables of arbitrary type:

```
#include <iostream>

using namespace std;

template<class T>
void swap(T* x, T* y) {
    T temp = *x;
    *x = *y;
    *y = temp;
}

int main(void) {
    int a = 5;
    int b = 23;
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    swap<int>(&a, &b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```

```

    return 0;
}

```

The syntax is simple, we prefix the function or class with the keyword `template` and specify in relational characters what kind of templates we will use. In this example we have generalized only one type.

When we call a template function we can specify which type to substitute for our template. But when it is clear, we can leave it out:

```

int main(void) {
    int a = 5;
    int b = 23;
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    swap(&a, &b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    return 0;
}

```

And of course it works with multiple types within a program (that's the whole point):

```

int main(void) {
    int a = 5;
    int b = 23;
    float x = 4.3;
    float y = 10.9;
    swap<int>(&a, &b);
    swap<float>(&x, &y);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    cout << "x: " << x << endl;
    cout << "y: " << y << endl;
    return 0;
}

```

4.2. Template class

Template-ing also works with classes. This way we could create a generic linked list to store any kind of data. Let's look at a simpler container:

```

#include <iostream>

using namespace std;

template<class T>
class SmartArray {
private:
    T* arr;
    int s;
public:
    SmartArray() {
        arr = new T[100];
        s = 100;
        for(int i = 0; i < s; i++) {
            arr[i] = 0;
        }
    }

    SmartArray(int s) {
        arr = new T[s];
        this->s = s;
        for(int i = 0; i < s; i++) {
            arr[i] = 0;
        }
    }

    int size() {
        return s;
    }

    T& operator[](int index) {
        return arr[index];
    }

    ~SmartArray() {
        delete[] arr;
    }
};

```

```

int main(void) {
    SmartArray<float> a = SmartArray<float>(5);
    a[0] = 1.5;
    a[1] = 6.4;
    a[2] = 2.1;
    for(int i = 0; i < a.size(); i++) {
        cout << a[i] << endl;
    }
    return 0;
}

```

This class implements a slightly smarter array, which you can request the size of and automatically fills the elements with 0.

4.3. Multiple templates

We can generalise several types in a template. For example:

```

#include<iostream>

using namespace std;

template<class T1, class T2, class T3>
struct triple {
    T1 a;
    T2 b;
    T3 c;
};

int main(void) {
    triple<int, float, int> p;
    p.a = 5;
    p.b = 5.5;
    p.c = 4;
    return 0;
}

```

Also, you can put not only types in the template:

```

#include <iostream>

using namespace std;

template<class T, int default_size = 100>
class SmartArray {
private:
    T* arr;
    int s;
public:
    SmartArray() {
        arr = new T[default_size];
        s = default_size;
        for(int i = 0; i < s; i++) {
            arr[i] = 0;
        }
    }

    SmartArray(int s) {
        arr = new T[s];
        this->s = s;
        for(int i = 0; i < s; i++) {
            arr[i] = 0;
        }
    }

    int size() {
        return s;
    }

    T& operator[](int index) {
        return arr[index];
    }

    ~SmartArray() {
        delete[] arr;
    }
};

```

```

int main(void) {
    SmartArray a = SmartArray<float, 10>();
    a[0] = 1.5;
    a[1] = 6.4;
    a[2] = 2.1;
    for(int i = 0; i < a.size(); i++) {
        cout << a[i] << endl;
    }
    return 0;
}

```

This is a version of the previous smart array where you can specify the default size of the array in the template. So you can put variables in the template and even give them a default value.