9. Classes

Ján Dugáček

November 21, 2018

Table of Contents

Motivation

- It sucks to have a heterogenous table stored in a vector of vectors

```
data[x][1] = (data[x - 1][1] + data[x + 1][2]) / 2;
// ...
data[x].fx = (data[x - 1].fx + data[x + 1].fx) / 2;
```

std::pair

```
std::pair<int, float> a;
a.first = 3;
a.second = 3.5;
std::vector<std::pair<int, float>> v;
v.push_back(a);
v.push_back(std::make_pair(3, 3.5));
```

- `std::pair` is a convenient way to create small classes of two elements of preset types, accessed as `first` and `second`
- They shine when set as pairs of values in vectors and other containers
- `std::make_pair` is a function that returns a `std::pair` object with types as its two arguments

struct

```
struct functionPoint {
        bool valid;
        float x;
        float fx;
};
std::vector<functionPoint> func;
functionPoint point{false, 1, 2};
point.valid = true;
func.push_back(point);
```

- struct creates new type of variable that is composed of other types
- The variables it contains can be accessed using the dot after variable name
- The variables inside are called *members*

struct #2

```
struct emergency {
        bool nuclearWar = false;
        bool alienInvasion = false;
        bool leakingToilet = false;
};
emergency situation;
leakingToilet = true;
std::cout << situation.nuclearWar << std::endl;
```

- You can set the default values of the variables

Methods

```
struct vec3D {
        float x;
        float y;
        float z;
        void normalise() {
                float length = sqrt(x * x + y * y + z * z);
                x /= length; z /= length; y /= length;
        }
};
// ...
vec.normalise();
```

- Functions defined in `structs` (called *methods*) can access and modify its variables
- They are called in a similar way than members are accessed

Methods #2

```
struct quaternion {
        float real, i, j, k;
        quaternion operator+(const quaternion& o) {
                quaternion result;
                result.real = real + o.real;
                result.i = real + o.i;
                result.j = real + o.j;
                result.k = real + o.k;
                return result;
        }
};
// ...
quat3 = quat1 + quat2;
```

- Same applies to operators, allowing you to get normally working algebraic types
- Uses one less argument, because the object left from the operator is the method's object itself

Constructor and destructor

```
struct superStruct {
        superStruct() {
                std::cout << "SuperStruct has been created!\r
        }
        ~superStruct() {
                std::cout << "SuperStruct has been destroyed!
        }
};
// ...
superStruct super;
```

- Constructor is a method called when the object is created
- Destructor is a method called when the object is being deallocated

Constructor

```
struct keeper {
        std::vector<float>& vec;
        const int size;
        keeper(std::vector<float>& vec)
        : vec(vec), size(vec.size()) {
        }
};
```

- Constructors can have an initialisation section that can set constant variables and references

class

```
class Privacy {
        int secret;
public:
        void setSecret(int newSecret) {
                secret = newSecret;
        }
private:
        int revealSecret() {
                return secret;
        }
};
```

- `class` is like `struct`, but its members are `private` by default and can be accessed only by methods of that `class`
- Members or methods after the `public` declaration are accessible from everywhere
- Here, the `secret` is quite hard to get from the objects
- `struct` can also have `private` members, but they are `public` by default

Exercise

1. Write a function that transforms an inconvenient convenient vector of vectors into a vector of `std::pairs`
2. Create a class that has a method that consecutively returns strings like 0000, 0001, ... 0042, ..., 0997 etc.
3. Create a `radionuclide` class that has a chance to change its `decomposed` member when a certain method is called
4. Create a `rock` class that represents a rigid body in gravitational field, give it a method that makes its properties develop in time

Advanced exercise

1. Create a `triplet` class that is like `std::pair`, but it contains three elements
2. Create a class that represents numbers in modular arithmetic and implement some of its operators

Inheritance

```
struct toRead {
        unsigned int index;
};
struct warning : public toRead {
        std::string text;
};
struct message : public toRead {
        std::string text;
        user author;
};
```

- structs warning and message inherit members and methods of toRead
- They can be assigned to a variable of type toRead, allowing the same function to access their index member

Advanced exercise

1. Create a class that represents arithmetic functions composed of variables, addition, subtraction, multiplication and division (the easiest way to do it is to make a tree structure of classes using inheritance)

Motivation

- Normal variable assignment is deep copy, the whole object is copied
- This is a problem for larger objects or objects we want to access from more locations
- Reference is a shallow copy, the variable may have a different name but address the same variable
- References are fine when used as function arguments, but objects often outlive the blocks they are created in
- `Pointers` are more powerful references

std::shared_ptr

```
std::shared_ptr<HugeObject>
                makeHugeObject(const std::string& file) {
        std::ifstream in(file);
        std::shared_ptr<HugeObject> made
                = std::make_shared<HugeObject>(in);
        return made;
}
//...
std::shared_ptr<HugeObject> huge = makeHugeObject("megadat");
```

- std::shared_ptr is a class that contains a single object that doesn't copy it if copied
- All copies of the shared pointer contain the same object
- The object stops existing when the last shared pointer is deallocated
- You have to make sure the object will not contain a copy of the shared pointer (or some other circular reference), otherwise it will keep existing until the program exits (it's called *memory leak*)

std::shared_ptr #2

```
std::shared_ptr<std::string> krupa
                = std::make_shared<std::string>("A");
krupa->push_back('B');
std::string betterKrupa = *krupa;
krupa->append("CD");
std::string& literateKrupa = *krupa;
krupa->append("E");
```

- Accessing members of the object in std::shared_ptr is done through the -> operator
- Use the left asterisk * to obtain the object inside (it's not a copy if not assigned to a non-reference variable)
- std::shared_ptr is much like a reference, but it survives the deletion of the original and can be replaced at the cause of slightly harder usage

Naked pointer

```
std::string* superKrupa = krupa.get();
std::cout << *superKrupa << std::endl;
superKrupa->push_back('F');
std::string* krupaPtr = &betterKrupa;
```

- Naked pointer allows accessing the variable as other pointer types, but it's just a number and has no methods
- It can be obtained from any variable using the left & operator
- **If accesseed after the object was deleted, bad mojo will happen!**
- It can be useful to allow the object inside a `std::shared_ptr` to keep access to an object that holds the `std::shared_ptr`
- It can also be used instead of reference if for some reasons a reference cannot do the trick

Empty pointers

```
krupa.reset();
superKrupa = nullptr;
if (superKrupa)
        std::string << "There is a superKrupa" << std::endl;
```

- Unlike references, pointers can be empty
- An empty pointer contains address 0 (for readability, it's written as `nullptr`)
- Accessing an empty pointer causes the program to reliably crash:

```
*((float*)nullptr) = 0;
```

- An empty pointer is considered `false`, a non-empty one is considered `true`

std::unique_ptr

```
std::unique_ptr<HugeObject> makeHugeObject
                (const std::string& file) {
        std::ifstream in(file);
        std::unique_ptr<HugeObject> made
                = std::make_unique<HugeObject>(in);
        return std::move(made);
}
// ...
std::unique_ptr<HugeObject> huge = makeHugeObject("megadat");
```

- std::unique_ptr is very much like std::shared_ptr, but it cannot be copied
- It can be moved using std::move, a function that clears the original variable and saves it into the one it's assigned to
- If you have to access it from elsewhere, you can use references or naked pointers
- Like naked pointer, it's faster than std::shared_ptr

Exercise

1. Use `std::shared_ptr` to create a class that keeps the following lines in a tree structure when it parses, writes and allows accessing the following markup

```
Tools
    Hammers
        Small hammers
        Big hammers
    Screwdrivers
        Cross Screwdrivers
```

Homework

- Write a function that analyses a line of noisy data (can be a vector) where it finds the point where it starts increasing and the point where it stops increasing and returns the interval where it increases and the amount it increased in a struct
- You have two weeks to do it