5. C++

Ján Dugáček

September 11, 2017

Table of Contents

1. Introduction
2. Basic changes
   - nullptr
   - Shorter struct
   - Reference
   - Printing
   - Including C libraries
   - Functions with same names
   - Allocating
   - Exercises
3. Objects
   - struct expanded
   - Constructors and destructors
   - Const correctness
   - Operator overloading
   - Encapsulation
   - class
   - friend
   - Exercises
   - std::string
4. Homework

Introduction

- You know most of C now
- Many trivial things are quite hard to do in C
- C++ allows to do these things more practically
- Practicality often comes at the cost of execution speed, but not necessarily
- To remember the keyword easier: `https://www.youtube.com/watch?v=c3zLTpDbyDc`

nullptr

```
float* num = nullptr;
```

- nullptr, is used in the same way as NULL, but it cannot be accidentally assigned as number 0
- It is good for avoiding mistakes

Shorter struct

```
struct stuff {
        int index;
        char abbrev[4];
};
stuff a = {3, "axe"};
```

- You don't have the use the struct keyword before the type name

Reference

```
void increment(int& a) {
        a++;
}
//...
        int a = 2;
        increment(a); // a IS affected
        int& b = a; // Initialisation, not assignment
```

- Reference is similar to pointer, but it's automatically dereferenced every time it is used
- It must be initialised when created and can never be changed (it would change the variable it's initialised to)
- Any change to the reference will change the variable it points to, obtaining its address will obtain the address of the variable it points to
- It is sometimes practical to have a pointer with limited abilities, but the side effect of functions that accept reference arguments is not visible

Printing

```
#include <iostream>
\\...
int a = 2;
std::cout << "Value of a is " << a << std::endl;
```

- More comfortable than `printf()`, but can't do all that `printf()` can do
- The printing functions called are chosen in compile time according to the type of the variable
- If you write `std::end` instead of `std::endl`, you will get a 1000 lines long error report (if you get an error report that long, check for this error)

Including C libraries

- C library functions are not useless in C++
- While you may use `#include <math.h>`, it's better to omit the `.h` suffix and add a `c` prefix, writing `#include <cmath>`
- If you include the C libraries that way, you don't have to use `-lmath` compiler arguments and such

Functions with same names

```
void identify(int a) {
        std::cout << "A is an integer" << std::endl;
}
void identify(float a) {
        std::cout << "A is a float" << std::endl;
}
```

- The functions have different names after compilation, containing also the types of arguments
- The compiler chooses which function is called (C++ is still a statically typed language, the types are fixed but can be deduced by compiler)
- Although `float` can be implicitly converted to `int`, the compiler picks the function whose type is closer (or reports ambiguity)

Allocating

```
int* a = new int; // Allocates an integer
int* b = new int(2); // Allocates an integer and sets it
int* c = new int[2]; // Allocates an array of 2 integers
delete a; // Instead of free
delete[] c; // Use this to free arrays
```

- You can still use `malloc()`, but it's less practical
- `delete` what was allocated with `new`, `free()` what was allocated with `malloc()`

Exercises

1. Pick a program from previous exercises and rewrite it to use the C++ habits
2. Pick another program from previous exercises and rewrite it to use the C++ habits

Terminate files with `.cpp`, compile C++ using `g++ main.cpp -std=c++11 -o main`, arguments are the same.

struct expanded

```
struct counter {
        int count_;
        void increment() {
                count_++;
        }
};

counter a = { 0 };
a.increment();
```

- `struct` can have functions (called *methods*) that are attached to them and can access the variables they contain as local variables
- To avoid mistaking parts of the `struct` with local variables, they should have a common prefix or suffix
- *Methods* are internally regular functions that use the pointer to the class as first argument, they are not a part of the `struct`

Constructors and destructors

```
struct pointerGuard {
        int* pointer_;
        pointerGuard(int* pointer) : pointer_(pointer) { }
        ~pointerGuard() {
                delete pointer_;
        }
};
void doStuff {
        int* pointer = new int(4);
        pointerGuard guard(pointer);
} // guard expires, destructor is called
```

- Function with same name as the `struct` and no return value is a *constructor*, it is called when it is created
- Function named after the `struct` but with the ~prefix and no arguments is a destructor, it's called when the object is deleted or expires

Const correctness

```
struct cmplx {
        float real;
        float imag;
        float abs() const {
                return sqrt(real * real + imag * imag);
        }
};

const cmplx a = {2, 3};
float ab = a.abs();
```

- Only methods declared as const may be called on objects declared as const
- Methods declared as const may not modify attributes (unless these attributes have the `mutable` modifier)

Operator overloading

```
struct counter {
        int count_;
        counter() : count_(0) {}
        void operator++(int unused) {
                count_++;
        }
};
counter a;
a++; // operator++() without argument would be for ++a
```

- You can define operations that can be done with the object using *operators*
- If the operations is not unary, the method accepts an argument that is the other operand

Operator overloading #2

```
struct matrix_3x3 {
        float val_[3][3];
        bool operator== (matrix_3x3& other) {
                for (int i = 0; i < 3; i++)
                        for (int j = 0; j < 3; j++)
                            if (val_[i][j]!=other.val_[i][j])
                                return false;
                return true;
        }
        void operator+= (matrix_3x3& other);
        matrix_3x3 operator* (float m); // Defined elsewhere
        matrix_3x3 operator* (vector_3& v);
        matrix_3x3 operator* (matrix_3x3& other);
};
```

Encapsulation

```
struct counter {
public:
        counter() : count_(0) {}
        void operator++(int unused) { count_++; }
        int getCount() { return count_; }
private:
        int count_;
};
```

- To prevent mistakes, it is possible to make some content unavailable unless accessed from the right place
- Anything behind `public:` is available from everywhere, anything behind `private:` is available only from methods of that object type and anything behind `protected:` is available only for methods of that object
- It can be circumvented by changing the type to something with the same memory layout, but this is to prevent mistakes, not to prevent hacking

class

```
class counter {
        int count_;
public:
        counter() : count_(0) {}
        void operator++(int unused) { count_++; }
        int getCount() { return count_; }
};
```

- In `struct` or `union`, everything is public until explicitly set, which is consistent with C and used for smaller objects in C++
- `class` is totally like `struct`, but it has everything private until a change (*everything* private is quite useless)
- The variable types of `struct` and `class` are called *classes*, initialised variables in memory are called *objects*

friend

```
class counter {
        int count_;
        friend void incrementCount(counter& a);
};

void incrementCount(counter& a) {
        a.count_++;
        if (a.count_ % 10) {
                std::cout << "Ten counts called!" << std::en
        }
}
```

- Any function or class declared as `friend` of some class, allowing it to access all its member variables
- Friendship is **not** mutual (even) in programming

Exercises

1. Write a `complex` class that represents complex numbers and define operators for usual operations with them
2. Write a `class` that acts like a `str` class, can be created from a `char*` with a constructor, has methods for appending other `str`, `char*` or `char` and obtaining its length (its content have to be extended if necessary)
3. Add to that class operators +, +=, ==, !=

Advanced:

1. Write classes `matrix` and `vector` that support operations +, - and * between each other and `float` and can be set to any dimension according to constructor arguments

std::string

```
std::string a("bla");
std::string b("42-1");
std::string c = a + b; // writes bla42-1 to c
a += c; // appends c to a, resulting in blabla42-1
a.append(".co"); // appends to a, resulting in blabla42-1.co
a.push_back('m'); // appends a character, blabla42-1.com
b = a.substr(a.find("bla"), 4); // picks blab
```

- std::string does most of the text things that is annoying to do with C
- They are, however, often slower and not always practical
- Use std::getline(std::cin, a); to read a line of input and save it into string a
- It needs to include string

Homework

- Write a class that describes a function given by the values of its elements from 0 up to a value given in constructor (ideally a dynamically allocated array), elements are accessed with [] like in array, must have a method `differentiate` that replaces the array with its derivative (1 element shorter, needs resize)

Advanced homework:

- *Your friend who is used to Python wants to use C++, but he is afraid of the static typing.* Create a class named `var` with size 8 bytes, that can hold an integer, a float, a short string up to 7 characters, a pointer to a string of any size (for longer strings) or a pointer to an array of these variables (resizeable, expanded when writting behind end). Usual arithmetic should apply to all types as it would if the types were known, decided in runtime. See previous slides' section about `union` to see how to do it.

```
enum types : unsigned char { // The internal type can be set
        isPointer = 0,
        //...
```