8. Other useful things

Ján Dugáček

September 21, 2017

Table of Contents

Smart pointers: What are they?

- It is very easy to forget to `delete` some objects
- There are occasions where the program must decide when to `delete` the object in run time
- Smart pointers solve this with minimal time penalty
- Code with smart pointers is longer if `typedef` is not used wildly

std::shared_ptr

- `std::shared_ptr` is a template class that holds a pointer that will be deleted when the last of its copies expires
- This is for cases where the pointer is copied on various places and there's no way of telling which one will hold it last
- There is a performance impact when creating, copying and destroying (it needs to keep track of its quantity), but dereferencing itself has no performance impact as the function call will be inlined by the compiler

```
std::shared_ptr<int> a = std::make_shared<int>(4);
std::shared_ptr<int> b = a;
*a = 5;
if (*b == 5)
```

- Its usage is very similar to raw pointers, but its creation is slightly altered
- You need to include `memory` before using smart pointers
- Uninitialised smart pointer is `nullptr`

std::unique_ptr

- std::unique_ptr is a template class that holds a pointer that will be deleted when the class expires
- The pointer is not to be copied, it's only meant to prevent you from forgetting to call the destructor
- There is no performance impact, because the destructor will be inlined during compilation

```
std::unique_ptr<int> a = std::make_unique<int>(4);
*a = 5;
if (a) *a = 6;
```

- Its usage is very similar to raw pointers, but its copying is restricted
- Its method reset() can be used to delete the contents (if not null) and replace it with new contents, its method swap() can be used to swap its contents with another std::unique_ptr

A small problem

```
struct a {
        std::shared_ptr<a> p;
};
std::shared_ptr<a> x = std::make_shared<a>();
std::shared_ptr<a> y = std::make_shared<a>();
x->p = y;
y->p = x;
```

- In this case, **both `x` and `y` are not deallocated**, because `x` holds one copy of pointer to `y` and `y` holds one copy of pointer to `x`; you get a memory leak despite using smart pointers
- This can be dealt with something that checks the reachability of memory blocks and deletes what is unreachable, but it would lead to unwanted deletions if pointer arithmetic was used (but it's used in languages that don't allow pointer arithmetic, like Java)
- In C++, you have to explicitly declare the order of dependence

std::weak_ptr

- std::weak_ptr is a copy of a shared pointer, but does not increase the reference count, it has to be converted to a (local variable) shared pointer before any use

```
struct a {
        std::shared_ptr<b> p;
        int value;
};
struct b {
        std::weak_ptr<a> p;
};
std::shared_ptr<b> x = std::make_shared<b>();
std::shared_ptr<a> y = std::make_shared<a>();
x->p = y;
y->p = x;
x->p.lock()->value = 2;
```

- Its method expired() checks if the contents it points to has been deallocated or not

Exercises

1. Change your last homework (about XML) to use smart pointers

Reminder:

```
typedef std::shared_ptr<std::string> strPtr;
typedef std::make_shared<std::string> mkStrPtr;
//...
strPtr str = mkStrPtr("This is shorter!");
```

Note:

- If you have not done your last homework, pick some other exercise that you have done and has a lot of dynamic allocation

Multiple files

```
// square.h
float square(float); // Declaration

// square.cpp
float square(float x) { // Definition
        return x * x;
}
// tesseract.cpp
#include "square.h"
float tesseract(float x) {
        return square(square(x));
}
```

- The definition may be in another file, it can be used as long as its declaration is available
- This must be compiled as `g++ tesseract.cpp square.cpp -std=c++11 -o tesseract` (headers are not listed there)

Multiple include

- We don't want one file to be included more than once, if 20 headers include themselves a header with a common function (like `sqrt()`), it will be compiled 20 times, possibly resulting in multiple definition
- The usual way of preventing this is to check if a macro is present, if not, declare it and include the contents

```
#ifndef SQUARE_H
#define SQUARE_H
float square(float);
// Everything else here
#endif
```

- `#ifdef`, `#ifndef`, `#else` and `#endif` decide which parts of the code are ignored when loading, it can also be used to add some debugging checks that can be easily disabled

Separate compilation

- If the program is large, all the compilation data would not fit into RAM and recompiling the whole thing from scratch after every little change would become very time-consuming
- The product of compilation does not have to be an executable file, it may be an object file (`.o` affix) and these object files are linked into an executable when all are done
- It's quite annoying to do manually, so it's usually done by *makefile scripts*, here is a simple example (it compiles all `.cpp` files in its folder)
- Initialise it with `cmake .` and run it with `make -j3` (use another number for a different number of CPU cores to use)

```
project(myAwesomeProject)
cmake_minimum_required(VERSION 2.8)
aux_source_directory(. SRC_LIST)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 -O3 -g")
add_executable(${PROJECT_NAME} ${SRC_LIST})
include_directories(${GLIB_PKG_INCLUDE_DIRS})
target_link_libraries(${PROJECT_NAME} -pthread)
```

Performance considerations

- If a function is compiled and placed in different file, it cannot be inlined and the function call will really be a function call
- This is not a problem with larger functions where inlining would bloat the code and cause more cache misses in executed code
- If the function is short and does not require including large headers that slow down compilation too much, it makes sense to define it in the header

```
class bam {
        int bom;
        inline int getBom() { return bom; }
        inline void operator++() { bom++; }
        inline bam() : bom(0) {}
};
```

- Functions declared in a header would be compiled multiple times witch each file that includes it, so the `inline` modifier must be used to make the compiler take care of it

Exercises

1. Include your last homework (about XML) into a simple program that uses XML to save some data, but is written in different file(s) than your XML library

Note:

- If you have not done your last homework, pick some other exercise that you have done and reminds of a library

Namespaces

- Namespace is like a folder for functions, classes and such, to allow entities with the same name to be used in the same file without problems

```
namespace expectations {
        const int expectedSize = 20;
}
//...
for (int i = 0; i < expectations::expectedSize; i++) {
```

- Various entities can be set into one namespace at different locations
- We can omit the namespace if we set it

```
using expectations::expectedSize;
//...
for (int i = 0; i < expectedSize; i++) {
```

- We can enable omitting an entire namespace as well

```
using namespace std;
//...
shared_ptr<string> a = make_shared<string>("Hi there");
```

Streams

- Stream is a generalised interface for reading and writing text
- Streams for reading inherit from `std::istream` and streams for writing inherit from `std::ostream`
- Stream for program input is `std::cin`, for program output are `std::cout` and `std::cerr`, for files are `std::ifstream` and std::ofstream (input and output respectively), for writing into string is `std::stringstream`, headers are `iostream`, `fstream` and `sstream` respectively

```
std::ofstream fs("the_reply.txt", std::ofstream::out);
if (!fs.good()) return; // Check if it didn't break;
fs << "The file was written." << std::endl;
fs.close();
std::stringstream ss;
ss << "We have found " << bodies << " bodies." << std::endl;
std::string got = ss.str();
```

auto

- Sometimes, the type name is terribly long, but easy to determine from the type assigned
- In this case, the type name can be replaced with auto that derives it from the return type when compiling
- It's not dynamic typing, the type is known at compile time, it's just deduced when reading

```
std::unordered_map<std::string, std::vector<std::string>> m;
// ...
for (auto it = m.begin(); it != m.end(); ++it) {
```

Function pointers

- A function is, in assembly, identified by its location in program memory
- Although we cannot read nor write into program memory, we can choose which part to execute
- The function types must be respected, because we still need to know what arguments to give it so that it would work

```
float (*sqrtPtr)(float) = sqrt;
float a = sqrtPtr(4);
```

- This works also in C
- Function pointers can be assigned to class members and used as a pseudo-method that can be changed (this also works in C), but occupies space in memory and has no access to member variables
- It's often used to call a function that would call a given function on its arguments (i.e. a sorting function that accepts a container and a function that compares two elements)

Lambda functions

- Lambda function is a function declared within a function that can use variables that surround it
- It's commonly used to avoid repeating parts of code without defining a function that needs to be placed elsewhere and may need a load of arguments; lambdas are very easily inlined
- It may be used instead of a function pointer (note that it's *not* a function pointer itself), with the benefit of keeping its surrounding variables with itself

```
float power = 0.5;
std::function<float(float)> root = [&] (float x) -> float {
        return pow(x, power);
};
float sqrt3 = root(3);
```

- The part between square brackets is called capture area
- [] means that no variables are accessible, [&] means all are taken by reference and [=] means that all are copied (copying is necessary if the lambda's lifetime exceeds the variables' lifetime)

Lambda functions #2

- Variables not mentioned in the lambda code are not copied or referenced or anything
- `std::function` can contain both a function and a lambda
- Because the lambda's type is rather complicated and can be easily determined from the assignment part, it's useful to use `auto` (its main disadvantage is that it prevents the lambda function from calling itself)
- If an argument to lambda has type `auto`, the lambda is a template

std::pair

- Sometimes, it's practical to wrap two variables into a single variable without creating a custom class for it

```
auto divide = [] (int x, int y) -> std::pair<float, int> {
        return std::make_pair((float)x / (float)y, x / y);
};
std::pair<float, int> got = divide(3, 4);
std::cout << "Either " << got.second << " or " << got.first;
```

- `std::map` and `std::unordered_map` have a `std::pair` of the value and its index in their iterators
- It's very much like:

```
template<typename T1, typename T2> struct pair {
        T1 first;
        T2 second;
};
```

Exceptions

- Sometimes a file fails to open, sometimes a file has some errors, sometimes a user interrupts a process, sometimes something is set badly and the program needs to be ready for it, a crash would lead to unsaved data being lost
- It can be done with a lot of `if` checks, but they might be needed in such quantities that it would slow down the program and be very tricky to code
- So, if an exception is thrown, the execution will leave all blocks until a proper `catch` block is found

```
int** readData(std::string file) {
        std::ifstream in(file);
        if (!in.good()) throw(std::string("Shit happened"));
//...
try {
        data = readData("/dev/null");
} catch (std::string happened) {
        std::cout << "Problem: " << happened << std::endl;
}
```

Exceptions #2

- The `try` block takes a lot of memory, exception throwing is slow, but as long as the number of `try` block is low and no exception is thrown, the performance impact is small compared to the performance impact of a lot of checks
- Exceptions should therefore be used only if the behaviour is exceptional
- Some parts of memory may not be deallocated because the `delete` part was not reached, leading to memory leaks, it can be avoided with a unique pointer (destructors are called)
- If `new` fails to allocate memory (full RAM, for example), exception of type `std::bad_alloc` is thrown
- Standard exceptions should be child classes of `std::exception` that has a `what()` method that returns a string that was used in its constructor, typical exception types used are `std::logic_error` (to detect bad programming) and `std::runtime_error` (to report errors that happen due to bad circumstances)
- There may be specialised `catch` blocks for each exception type, with child types before parent classes (parent catch would otherwise catch the child exception); `catch(...)` catches everything, but because it does not know the type, there is nothing it can report except that an exception was thrown

Homework

- Create a math library (with header and code parts) that has real numbers, complex numbers and quaternions (complex number is real number with some extras, quaternion is a complex number with some extras) that use inheritance for common operations like addition and subtraction (you can use real number addition to add a real number to a complex number), supports also multiplication and throws exceptions if something is done badly (assigning a complex number to a real number if the complex number has an imaginary part)
- Write a few functions that use it in a different file
- About quaternions: `https://en.wikipedia.org/wiki/Quaternion`

Advanced homework:

- Same as the regular one, but implement also inverse, division and square root (selecting only one possible result if more are possible)