Professional C++ notes

Anas Syed

September 3, 2015

Contents

Chapter 1

Basics

1.1 Preprocessor directives

`#pragma` is not standard across all compilers, so don't use it.

Listing 1.1: Prevent circular includes

```
#ifndef [key]
#define [key]
//code
#endif
```

1.2 Casting

1. `bool someBool = (bool)someInt;`
2. `bool someBool = bool(someInt);`
3. `bool someBool = static_cast<bool>(someInt);`

Item 3 is considered to be the cleanest.

1.3 Structs

Structs are the same as classes in `C++`, except the default access specifier for a struct is public, whereas for a class it's private. If we define a struct as follows:

```
struct employee {
  int age;
  float salary;
  char initial;
} employee1, employee2;
```

then when we call the struct in C, we must prefix `employee` with `struct`. In C++, this is optional. Alternatively, in C, we can `typedef` the struct:

```
typedef struct {
  int age;
  float salary;
  char initial;
} employee;
employee employee1, employee2;
```

or

```
struct employee {
  int age;
  float salary;
  char initial;
} employee1, employee2;
typedef struct employee employee;
```

1.4 Stack and heap

The stack is a Last in First out data structure. If you call a function `foo()`, then all of the static variables (those not created using `new` or `malloc`) in `foo` exist in a stack frame. If `foo` was called from `main`, then you cannot easily change or access the static variables in the stack frame of `foo` from within `main`, because they are in a different stack frame.

However, if you allocate some dynamic memory to a variable in `foo`, then you could access or modify this variable in main.

1.4.1 Freeing memory

When you allocate memory with the `new` operator, you must eventually free it with `delete`. Note that this even applies to `int` objects. For example, the following code has a 4 byte memory leak:

Listing 1.2: Memory leak

```
int main() {
  int* a = new int(5);
}
```

Running `valgrind --tool=memcheck ./a.out` gives the error message:

Listing 1.3: Valgrind output

```
==11934== Memcheck, a memory error detector
==11934== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==11934== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==11934== Command: ./a.out
==11934==
==11934==
==11934== HEAP SUMMARY:
==11934==     in use at exit: 4 bytes in 1 blocks
==11934==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==11934==
==11934== LEAK SUMMARY:
==11934==    definitely lost: 4 bytes in 1 blocks
==11934==    indirectly lost: 0 bytes in 0 blocks
==11934==      possibly lost: 0 bytes in 0 blocks
==11934==    still reachable: 0 bytes in 0 blocks
==11934==         suppressed: 0 bytes in 0 blocks
==11934== Rerun with --leak-check=full to see details of leaked memory
==11934==
==11934== For counts of detected and suppressed errors, rerun with: -v
==11934== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

See `http://www.cprogramming.com/debugging/valgrind.html` for more information on Valgrind.

On the other hand, the following does not create a leak:

Listing 1.4: No memory leak

```
int main() {
  int a = int(5);
}
```

1.5 Strings

1.5.1 C style

```
char arrayString[20] = "Hello, World"; //allocates 20 bytes
char arrayString[] = "Hello, World"; //allocates 13 bytes
char* pointerString = "Hello, World"; //allocates 13 bytes, deprecated in C++. Should be
    ↪ const char*
const char* pointerString = "Hello, World"; //allocates 13 bytes
```

Use `#include <cstring>` for standard `C` library functions.

1.5.2 C++ style

```
#include <string>
int main() {
  std::string str1 = "Hello";
  std::string str2 = str1 + ", World";
  std::cout << str2.size() << std::endl; //12
}
```

Since C++11, there is a new `std::to_string` function which returns an `std::string`, just include `#include <string>`.

1.6 Exceptions

```
#include <iostream>
#include <stdexcept>

double divideNumbers(double inNumerator, double inDenominator) {
  if (inDenominator == 0) {
    throw std::exception();
  }
  return (inNumerator / inDenominator);
}

int main(int argc, char** argv) {
  try {
    std::cout << divideNumbers(2.3, 0) << std::endl;
  } catch (std::exception exception) {
```

```
    std::cout << "An exception was caught!" << std::endl;
  }
}
```

1.7 `auto` and `decltype` (C++11) and `decltype(auto)` (C++14)

`auto` uses type inference to determine variable type, but it strips away reference and `const` qualifiers.

If we have the following:

```
const string message = "Test";

const string& foo() {
  return message;
}
auto f1 = foo();
const auto& f2 = foo();
decltype(foo()) f3 = foo()
```

Then `f1` has type `string`, and `f2` and `f3` have type `const string&`. However, there is code duplication for `f3`, so the solution is to do

```
decltype(auto) f4 = foo();
```

And `f4` has type `const string&`.

1.8 `std::array` (C++11)

C++ has provided a thin wrapper around standard arrays in the `array` header.

```
#include <array>

int main() {
  std::array<int, 5> a {1,4,6,8,4};
  a[4] = 99;
  for (const int& i : a) {
    std::cout << i << std::endl;
  }
  std::cout << a.size() << std::endl;
```

```
}
```

Note that the size provided in the type declaration can be greater than the initialiser list, the extra elements will all be cleared to 0.

1.9 Range based `for` loops (C++11)

The following can all be used in a range based `for` loop.

```
std::vector<int> a = {1, 3, 5, 23, 52};
int a[] = {1, 4, 6, 2, 5};
std::array<int, 3> a {1,4,6,8,4};
std::string a = "Hello, World";
char a[] = "Hello, World";
```

And the `for` loop is written down as follows, and can even create a `const` or reference as the iterating variable:

```
for (const int& i : a) {
  std::cout << i << std::endl;
}
// or
for (const auto& i : a) {
  std::cout << i << std::endl;
}
```

1.10 `nullptr` (C++11)

The keyword `NULL` was the standard way to create a null pointer. However, it is just syntactic sugar for the integer literal 0. Since C++11, there is a special keyword `nullptr` which is a pointer type. This can resolve the issue in the following code:

```
1 void func(char* str) {cout << "char* version" << endl;}
2 void func(int i) {cout << "int version" << endl;}
3 int main() {
4   func(NULL);
5   return 0;
6 }
```

Clearly, the intention is to call the `char*` version of the function, but because `NULL` is an `int`, the `int` version is called. Running `func(nullptr)` solves the issue.

Chapter 2

Object oriented programming

2.1 Access specifiers

Any method of the class can call a protected method and access a protected member. Methods of a subclass can call a protected method or access a protected member of an object.

2.2 Reminder of syntax

Listing 2.1: SpreadsheetCell.h

```
class SpreadsheetCell {
  public:
    SpreadsheetCell();
    ~SpreadsheetCell();
    double getValue();
  private:
    double mValue;
};
```

Listing 2.2: SpreadsheetCell.cpp

```
#include "SpreadsheetCell.h"
SpreadsheetCell::SpreadsheetCell() : mValue(0) {
  std::cout << "Creating cell" << std::endl;
}

SpreadsheetCell::~SpreadsheetCell() {
  std::cout << "Destroying cell " << mValue << std::endl;
```

```
}

double SpreadsheetCell::getValue() {
  return (mValue);
}
```

2.3 Creating objects on the stack or heap

Listing 2.3: Creating objects on the stack

```
SpreadsheetCell myCell, anotherCell;
```

Listing 2.4: Creating objects on the heap

```
SpreadsheetCell* myCellp = new SpreadsheetCell();
//run some code
delete myCellp; //Don't forget to deallocate memory
```

2.3.1 Using smart pointers

```
#include <memory>
auto myCatp = std::make_unique<cat>(5); // C++14
// Equivalent to:
std::unique_ptr<cat> myCatp(new cat(5)); // C++11
```

2.4 Constructors

Whenever an object is created, one of its constructors *must* be called.

Say we had a constructor as follows:

```
SpreadsheetCell::SpreadsheetCell(double initialValue) {
  setValue(initialValue);
}
```

Then to create an object on the stack, we must do the following:

```
SpreadsheetCell myCell(5), anotherCell(4);
```

The following do not work:

```
SpreadsheetCell myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
//or
SpreadsheetCell myCell;
myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

And for allocation on the heap, use

```
SpreadsheetCell *myCellp = new SpreadsheetCell(5);
SpreadsheetCell *anotherCellp;
anotherCellp = new SpreadsheetCell(4);
delete anotherCellp;
```

For overloaded constructors, calling one constructor from another does not apply to the variable of the outer constructor (as expected). The following code shows this problem:

```
SpreadsheetCell::SpreadsheetCell(string initialValue) {
  SpreadsheetCell(stringToDouble(initialValue));
  //Does not call the constructor for this object
}
```

2.4.1 Initialiser lists

Consider a class as follows:

```
class SpreadsheetCell {
  public:
    SpreadsheetCell();
    SpreadsheetCell(int mValue, string mString);
  private;
    int mValue;
    string mString;
}
```

Now one way of implementing the default constructor is as follows:

```
SpreadsheetCell::SpreadsheetCell () : mValue(0), mString("") {
}
```

And the other constructor:

Listing 2.5: C++ gets it right with the name clash

```
SpreadsheetCell::SpreadsheetCell (int mValue, string mString) : mValue(mValue),
  mString(mString) {
}
```

Data type	Explanation
const data members	You cannot legally assign a value to a const variable after it is created. Any value must be supplied at the time of creation
Reference data members	References cannot exist without referring to something.
Object data members for which there is no default constructor	C++ attempts to initialize member objects using a default constructor. If no default constructor exists, it cannot initialize the object.
Superclasses without default constructors	The superclass must be initialised before the constructor begins execution

Table 2.1: Objects which must be included in the initialiser list

Note that this is different to the following:

```
SpreadsheetCell::SpreadsheetCell (int mValue, string mString){
  this->mValue = mValue;
  this->mString = mString;
}
```

Once the body of the constructor has been entered, all member objects and parent objects have already been created. So `this->mValue = mValue` modifies the value of the existing `mValue` object. In listing 2.5, the default constructor for an `int` is not called, but it is constructed with the correct value right from the start by calling a non-default constructor. This is more efficient and allows one to initialise member objects by something other than their default constructor.

The constructor for the parent class is called first. The order in which the constructor calls to the remaining elements in the initialiser lists are executed is determined by the order in which they are declared in the class definition.

Table 2.1 shows the objects for which inclusion in the initialiser list is mandatory.

See `http://www.cprogramming.com/tutorial/initialization-lists-c++.html` for an alternative explanation.

2.4.2 Copy constructors

An example of a copy constructor:

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src) :
  mValue(src.mValue), mString(src.mString) {
```

```
}
```

If you don't write a copy constructor, one is not written for you. The one that the compiler writes for you is not always suitable, especially when you have memory being dynamically allocated on the heap from within the object methods. More on this in

The copy constructor is implicitly called whenever you pass an object by value to a function. It can also be called explicitly like `SpreadsheetCell myCell(anotherCell);`.

The compiler generated copy constructor recursively calls the copy constructor to all object members.

2.4.2.1 What constructors the compiler generates

If you define any constructor (including a copy constructor), the compiler won't generate a default constructor, otherwise it will. If you define a copy constructor, the compiler won't generate a copy constructor, otherwise it will.

2.4.3 `default` and `delete` for compiler generated constructors (C++11)

If we have a class `object`, then we can explicitly create or delete the generated constructor or operator in the class definition:

```
class object {
  public:
    object() = default;
    object(const object& objToBeCopied) = delete;
    object& operator=(const object& rhs) = delete;
}
```

2.4.4 Default constructors

A default constructor is required if you want to be able to declare an object but not initialise it, like `SpreadsheetCell myCell`. Use a default constructor on the stack like so:

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

A word of warning: do not call the default constructor with parentheses when creating the object on the stack. The following is incorrect:

```
1 SpreadsheetCell myCell(); // WRONG, but will compile.
2 myCell.setValue(6); // However, this line will not compile.
```

Line 1 does compile. This is because the compiler thinks that you are declaring a function called `myCell` which returns a `SpreadsheetCell` and takes no arguments. The next line doesn't compile because the compiler thinks you are trying to call a method on the function!

When creating an object on the stack, omit parenthesis for the default constructor.

But when using the default constructor to create objects on the heap, you should use the functional syntax:

```
SpreadsheetCell* myCellp = new SpreadsheetCell(); // Note the function-call syntax
```

2.4.4.1 When you need a default constructor

2.4.4.1.1 Arrays If there is no default constructor for `SpreadsheetCell`, the following fail:

```
SpreadsheetCell cells[2]; // FAILS compilation without a default constructor
SpreadsheetCell* myCellp = new SpreadsheetCell[10]; // Also FAILS
```

There is no syntax to specify a different constructor for all elements. For stack based arrays, one can construct each element individually as follows:

Listing 2.6: Example of *initialisers*

```
SpreadsheetCell cells[2] = {SpreadsheetCell(0), SpreadsheetCell(23)};
```

There is no such syntax for heap based arrays.

2.4.4.1.2 Creating objects inside another class If you have a class as follows:

```
class twoCats {
  private:
    cat firstCat;
    cat secondCat;
};
```

Then either the class `cat` must have a default constructor, or every constructor for `twoCats` must initialise `firstCat` and `secondCat` with a non-default constructor of `cat` explicitly.

2.4.4.1.3 Subclasses It is convenient for a superclass to have a default constructor, as this makes the construction of subclasses simpler, as subclasses must always call a constructor of their respective superclass, in which case, having a default constructor for the superclass can be useful.

2.5 Destructors

Objects on the stack are destroyed when they go out of scope, and are destroyed in the order Last In First Out.

2.6 Assignment operator

The following code assigns `myCell` to the object `anotherCell` using the assignment operator. If you don't write an assignment operator, C++ writes one for you.

```
SpreadsheetCell myCell(5), anotherCell;
anotherCell = myCell; //Assignment
```

This is different to copying because the object is always being initialised when copying, whereas assignment applies only to already created objects.

This is the declaration of the assignment operator:

```
class SpreadsheetCell {
  public:
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
}
```

This returns a reference because assignment operators can be chained. Every definition of the assignment operator must return `*this`.

Note the following:

```
myCell = anotherCell = aThirdCell;
// is equivalent to:
myCell.operator=(anotherCell.operator=(aThirdCell));
```

2.7 Distinguishing copying from assignment

Note that in the following, the copy constructor is called for `anotherCell`:

```
SpreadsheetCell anotherCell = myCell;
```

2.8 In class member initialisers

Since C++11, you can initialise member variables in the class definition. Before, you could only do this in the constructor, or do it for `static const` integral member variables.

2.9 `const` and `static` functions

2.9.1 `const` functions

A `const` member method is one that does not modify any of the objects member variables (but it may modify other variables passed to it by reference or via a pointer). These are called *inspectors* as opposed to *mutators*. For example:

```
class cat {
  public:
    cat (int inputHeight);
    int getHeight(int& input) const;
    void setHeight(int input);
    int height;
};

cat::cat (int inputHeight) : height(inputHeight) {
}

int cat::getHeight (int& input) const {
  input++; // Allowed
  height--; // Won't compile
  return height;
}

void cat::setHeight(int input) {
  height = input;
}
```

If one declares a `const` object, then you may only call `const` member methods on that object.

Bibliography

[Return value optimisation]