

# Professional C++ notes

Anas Syed

August 30, 2015

## Contents

<b>1</b>	<b>Basics</b>	<b>2</b>
1.1	Preprocessor directives . . . . .	2
1.2	Casting . . . . .	2
1.3	Structs . . . . .	2
1.4	Stack and heap . . . . .	3
1.4.1	Freeing memory . . . . .	3
1.5	Strings . . . . .	4
1.5.1	C style . . . . .	4
1.5.2	C++ style . . . . .	4
1.6	Exceptions . . . . .	5
<b>2</b>	<b>Object oriented programming</b>	<b>6</b>
2.1	Access specifiers . . . . .	6
2.2	Reminder of syntax . . . . .	6
2.3	Creating objects on the stack or heap . . . . .	6
2.4	Constructors . . . . .	7
2.5	Default constructors . . . . .	8
2.5.1	When you need a default constructor . . . . .	8

# 1 Basics

## 1.1 Preprocessor directives

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

---

Listing 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 2: Memory leak

---

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

---

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

---

Listing 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 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* 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";
}
```

---

## 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;
    }
}
```

---

## 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 5: SpreadsheetCell.h

---

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

---

---

Listing 6: 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 7: Creating objects on the stack

---

```
SpreadsheetCell myCell, anotherCell;
```

---

---

Listing 8: Creating objects on the heap

---

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

---

## 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  
}
```

---

## 2.5 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!

<p><b>When creating an object on the stack, omit parenthesis for the default constructor.</b></p>
---

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
```

---

### 2.5.1 When you need a default constructor

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

```
preadsheetCell cells[3]; //FAIL compilation without a default constructor Spread-  
sheetCell* 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 9: Example of *initialisers*

---

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

---

There is no such syntax for heap based arrays.



**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.

**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.