C++ Fundamentals

Stephen P Levitt

School of Electrical and Information Engineering
University of the Witwatersrand

2019

Outline

1. Analysing A Simple Program
2. auto
3. Initialisation
4. The C++ Build Process
5. Types
   - Simple Programmer-Defined Types
6. Pointers and References
7. Parameter Passing
   - Understanding Parameter Passing
   - Compiler Optimisations
   - Guidelines

A Simple C++ Program

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Enter your name:" << endl;
    auto name = ""s;

    cin >> name;
    cout << "Hello " << name << endl;

    return 0;
}
```

A Simple C++ Program

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main()
7  {
8      cout << "Enter your name:" << endl;
9      auto name = ""s;
10
11     cin >> name;
12     cout << "Hello " << name << endl;
13
14     return 0;
15 }
```

Preprocessor Directives

```
1  #include <iostream>
2  #include <string>
```

Specified by '#' sign and no semi-colon at the end of the line.
For example the `#include` directive:

```
// NO .h for standard library files!
#include <iostream>
#include <cmath>
#include <some_file.h>
#include "my_file.h"
```

The included file is inserted directly into the including file, prior to compilation.

Namespaces

```
4   using namespace std;
```

- Prevent naming conflicts in the global namespace
- Standard library components are declared within the `std` namespace
- You can define your own namespaces

```
#include<string>

// error: string is not visible
string quote = "I'm falling asleep";
```

Namespaces cont.

Rather

```
#include<string>
using namespace std;

// ok: string is visible
string quote = "I'm falling asleep";
```

Or

```
#include<string>

// ok: use qualified name
std::string quote = "I'm falling asleep";
```

The main function

```
6   int main()

14      return 0;
```

- Only *one* main function per program — the entry point
- There has to be a main function for every executable (where is it for test executables?)
- Can take arguments (where do they come from?) and return an integer (where does it go?)

Iostreams

```
8    cout << "Enter your name:" << endl;

11   cin >> name;
12   cout << "Hello " << name << endl;
```

- `cout` is an object of the class `ostream` tied to the standard output device — the monitor
- `cin` is an object of the class `istream` tied to the standard input device — the keyboard
- The `cout` and `cin` objects are globally accessible objects declared in the `iostream` header file
- Output can also be directed to `cerr`
- `endl` ends the line and starts a new one

Variable Declaration and Initialisation

```
9     auto name = ""s;
```

- `auto` is used to deduce the type of `name` as a `std::string`
- Prefer to use `auto` instead of explicitly declaring types

Primitive or Fundamental Types (on Windows)

Integral Types - can be signed and unsigned

- char (1 byte): -128 to 127 / 0 to 255
- short (2 bytes)
- int (4 bytes)
- long (4 bytes)

Floating Point Types

- float (4 bytes)
- double (8 bytes)
- long double (8 bytes)

Boolean (1 byte)

Pointers (4 bytes on 32-bit systems, 8 bytes on 64-bit systems)

See the types and their ranges here:
https://docs.microsoft.com/en-us/cpp/cpp/fundamental-types-cpp

Using auto

What types are deduced for the following variables?

```
// auto and literals
auto x = 42;       // ?
auto x = 42.0;     // ?
auto x = 42.0f;    // ?
auto x = 42ul;     // ?
auto x = "hello";  // ?
auto x = "42"s;    // std::string
```

```
// auto and objects
auto e = Employee{empid};          // stack allocation
auto w = make_unique<Widget>();  // heap allocation
```

Almost always `auto`

Using `auto` and type deduction has the following benefits:

- Correctness and Maintainability
- Performance
- Convenience

Uniform Initialisation

- Prefer the {} syntax; avoid ()
- Simpler, more general, less ambiguous and safer

```
// object initialisation - constructors
auto s = string{"Hello"};
auto p = Person{"Julia", 20001212};

// container initialisation
// vector of 3 elements with values 2, 4 and 6
auto even_numbers = vector<int>{2, 4, 6};

// class member initialisation
D::D(int a, int b):
  m{a, b}
{
    // ...
};
```

Uniform Initialisation – When Not to Use

Initialising simple arithmetic types

You could do this: `auto x{5};`

but this seems more natural:

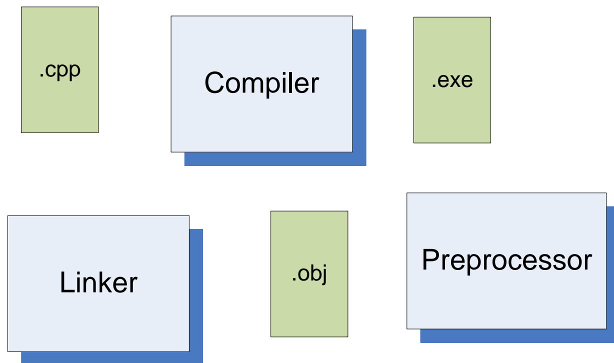
```
auto x = 5;
auto y = f(99);
```

Initialising containers to a specific size

```
auto v1 = vector<int>(10); // 10 element vector with default values of 0
auto v2 = vector<int>{10}; // 1 element vector with the value 10

// vector of 2 elements with the default value 4
auto even_numbers = vector<int>(2, 4);
// vector of 2 elements with values 2 and 4
auto even_numbers_2 = vector<int>{2, 4};
```

Building a C++ Project - What's Happening Behind the Scenes



Translation Units and Header Files

- A *translation* unit is:
  - the basic unit of compilation in C++
  - consists of the contents of a single source file (`.cpp`) and all of its included files
- Compile each translation unit in isolation by selecting the `cpp` file and pressing Ctrl-F7 in CodeLite
- Make each file self-sufficient by having it directly include any headers its contents depends upon.
- Don't include headers you don't need
- Always provide *#include* guards:

```
#ifndef MY_CODE
#define MY_CODE
// ...header file contents...
#endif
```

Error Types

Compiler errors

- Syntax errors
- Type errors
- Re-definition errors (often caused by missing include guards)

Linker errors

- Multiple definitions (caused by definitions in header files, functions and variables)
- Missing definitions (often related to files missing from the project)

Run-time errors

- Divide by zero, file does not exist
- Memory leaks

Logical errors

- A plus sign instead of minus sign

Inline versus Out-Of-Line Definitions I

Header file (maths_functions.h)

```cpp
class Maths
{
public:
  // declaration
  int add(int x, int y)
  {
      return x + y; // inline definition
  };

  // declaration only
  int factorial(int n);
};
```

Inline versus Out-Of-Line Definitions II

CPP file

```
#include "maths_functions.h"

// "out-of-line" definition
int Maths::factorial(int n)
{
  // find factorial using recursion
  return n == 0 ? 1 : n * factorial(n - 1);
}
```

Simple Programmer-Defined Types

What could go wrong when using this function?
How can we improve this function's signature?

```cpp
const int input = 1;
const int output = 2;
const int append = 3;

// a file can be opened in one of three states
bool open_file(string filename, int open_mode);
```

Scoped Enumeration Types

There is no way to constrain the value being passed into the function. Rather, get the compiler to help enforce our intention:

```
enum class OpenModes { input,
                       output,
                       append
                       };

bool open_file(string filename, OpenModes mode);
```

Usage:

```
open_file("myfile.txt", OpenModes::input);
```

- Enumeration types are useful for modelling small sets of distinct values like *Colours*, *Months* etc.
- Can be used in `switch` and `if` statements

Sketch a memory map of the following:

```
string name("Bob");

string* string_ptr = &name;

string& string_ref = name;

string** string_ptr_ptr = &string_ptr;
```

Pointers and References

Pointers hold the address of an object.
References provide an *alias* for an object.
Both are forms of indirection.

```
// construct a string object
string name("Bob");

// pointer to the string object
string* string_ptr = &name;
cout << *string_ptr << endl;

// a reference, or alias, for the string object
string& string_ref = name;
cout << string_ref << endl;
```

Is a Pointer Just a Reference in Disguise?

Is a Pointer Just a Reference in Disguise?

A pointer is a separate object (from the pointee) with its own distinct set of operations, lifetime, size, and memory location

```
// construct a string object
string name("Bob");
string* string_ptr = &name;

// dereferencing, incrementing
// only valid for ptrs - not string
*string_ptr;  // *name;  <- error
string_ptr++; // name++; <- error
```

Operations for a reference are defined by what it is referring to:

```
string& string_ref = name;
// clear() is a member function of string
string_ref.clear();
```

A pointer can point to null but a reference has to alias an *existing* object.

Pass-by-Value vs Pass-by-Reference

By default arguments are passed by value.

Pass-by-value is not always suitable

- When a function must modify the arguments passed

```
// unable to swap arguments
void swap (int num1, int num2);

// pass-by-reference - now able to swap arguments
void rswap (int& num1, int& num2);
```

- When we are concerned about efficiency

Pass-by-Value and Efficiency

```
class Person
{
public:
  // Constructor
  Person(const string& name): name_(name)
  { cout << "Creating a person" << endl; }

  // Copy constructor
  Person(const Person& rhs): name_(rhs.name_)
  { cout << "Copying a person" << endl; }

  // Destructor
  ~Person()
  { cout << "Destructing a person" << endl; }
};
```

Pass-by-Value and Efficiency cont.

```cpp
Person doNothing(Person p) // not something you would normally do
{ return p; }

int main()
{
    Person Thabo("Thabo");
    cout << "---------------" << endl;
    doNothing(Thabo);
    cout << "---------------" << endl;

    return 0;
}
```

What will the output be?

Pass-by-Value and Efficiency cont.

So:

```
// highly inefficient function
Person doNothing(Person p)
{ return p; }

// efficient function
Person& doNothing(Person& p)
{ return p; }
```

Pass-by-Value and Efficiency cont.

So:

```
// highly inefficient function
Person doNothing(Person p)
{ return p; }

// efficient function
Person& doNothing(Person& p)
{ return p; }
```

Output using the *efficient* version:

```
Creating a person
---------------
---------------
Destructing a person
```

```
Person newPerson()
{ return Person{"Ferial"}; }
```

Why can we expect the output from `main` to be the following?

```
Creating a person
Copying a person
Destructing a person
Copying a person
Destructing a person
Destructing a person
```

Return Value Optimisation — Copy Elision

Actual output:

```
Creating a person
Destructing a person
```

Guidelines

- Passing in parameters
  - If the argument will be changed, pass it by reference or non-constant pointer
  - If the argument will not be changed, pass it by a constant reference (avoid copies entirely), or if it is cheap to copy, by value
- Returning values
  - Prefer to return *by value* (taking advantage of compiler optimisations or move constructors)
  - Return multiple values using a tuple

```
// function returning multiple values
tuple<string, int, char> get_student_details()
{ return {"Homer Simpson", 234556, 'F'}; }

// calling the function
auto [student, student_number, grade] = get_student_details();
```