Idiomatic Modern C++ for Linux

Week 2: Getting Into The Basics

Today’s agenda

- fundamental types
- functions
- preprocessor macros
- namespaces / scope
- writing our first library

Intro to basic types

- If you've written C, a lot of this will be familiar
- C++ also introduces type introspection (more in a future lesson)
- We'll look at basic types, as well as a form of type aliasing

Basic type rules

```
int some_number;
double some_number = 8.76;
// won't compile. some_number was first defined as an int
```

- This isn’t python. No redefinition of variables as different types

Basic type rules

```
// Now let's declare a variable, but not initialize it.
int b;
// compiles, but technically undefined behavior since b was never initialized
cout << b << endl;
```

Integers

```
// Integer types
int a = -42; // The default integer is signed and guaranteed to be at least 16 bits.
    |    |    // On 32 or 64 bit systems, guaranteed to be at least 32 bits.
|
unsigned int b = 8675309; // Unsigned: never less than zero.

short int this_will_overflow = 32768;

// for short int and long int, you don't actually need to specify "int"
short this_wont_overflow = -32768;
```

Fixed width integers

```cpp
// Starting in C++11, you can use fixed width types (they're just type aliases)
int16_t some_fixed_width_int = 69;
uint64_t some_other_fixed_width_int = 12345678910;

// Hilariously, int8_t and uint8_t are actually just an alias for char and unsigned char.
// The compiler will let you just get away with the following.
uint8_t some_fixed_width_char = 69;
```

Ranges of types

```
default int ranges from: -2147483648 to 2147483647
unsigned int ranges from: 0 to 4294967295
short int ranges from: -32768 to 32767
unsigned short int ranges from: 0 to 65535
long int ranges from: -9223372036854775808 to 9223372036854775807
unsigned long int ranges from: 0 to 18446744073709551615
long long int ranges from: -9223372036854775808 to 9223372036854775807
unsigned long long int ranges from: 0 to 18446744073709551615
float ranges from: 1.17549e-38 to 3.40282e+38
double ranges from: 2.22507e-308 to 1.79769e+308
long double ranges from: 3.3621e-4932 to 1.18973e+4932
char ranges from: □ to (-128 to 127)
unsigned char ranges from:  to □(0 to 255)
```

Functions

- Reusable block of execution
- A ***function call*** tells the CPU to interrupt the current flow of execution and move to the beginning of the ***function body***
- The ***function signature*** includes the function name, return type, and arguments (and their types)

Functions

```cpp
// Takes no arguments,
// returns no values
void printTheBeeMovieIntro() {
    cout <<
    "According to all known laws "
    "of aviation,"
    "there is no way a bee "
    "should be able to fly."
    "Its wings are too small to get "
    "its fat little body off the ground. "
    "The bee, of course, flies anyway "
    "because bees don't care "
    "what humans think is impossible."
    << endl;
}
```

```cpp
printTheBeeMovieIntro();

// the function prints a string literal instead of returning a string
// so we can't print its output.
cout << printTheBeeMovieIntro() << endl; // (won't compile)
```

Functions – Type Coercion

```
// Takes two doubles (by value),
// returns an double
double add(double a, double b) {
    return a + b;
}
```

```
// the literals 5 and 9 will be cast to doubles
double sum = add(5, 9);

// we can call non-void functions without doing anything with the value
add(100000.01, 9.43);

// we can't cast string literals to ints
double everything_nice = add ("sugar", "spice"); // (won't compile)
```

Functions – Declarations & Definitions

You can “forward declare” functions and variables without defining them (often done in **header** or **.h / .hpp** files)

The compiler won’t care if the argument names aren’t consistent between *declaration* and *definition* (but still it’s not a good idea do this)

We can prove the compiler treats both references to **int mul** *as the same function* by commenting out the implementation part

```cpp
// from a compiler perspective, the following two statements
// will refer to *the same function* in memory
// and are not considered overloaded functions.

int mul(int a, int b); // forward declaration

int mul(int x, int y) { // implementation
    return x * y;
}
```

```
[ 25%] Linking CXX executable functions
/usr/bin/ld: CMakeFiles/functions.dir/src/functions.cpp.o: in function `main':
functions.cpp:(.text.startup+0x2f): undefined reference to `mul(int, int)'
collect2: error: ld returned 1 exit status
gmake[2]: *** [CMakeFiles/functions.dir/build.make:97: functions] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:89: CMakeFiles/functions.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
```

Bonus Cursed C++ Fact

```cpp
std::string this_was_undefined_behavior_pre_cpp17() {

    std::string message = "but I have heard it works even if you don't believe in it";
    // Fun fact! Until C++17, the calls to "replace" in the following expression
    // did not have a consistent order of evaluation.
    // From C++17 on, the evaluation of function calls is sequenced relative to each other,
    // generally resulting in FirstSecond due to operator precedence rules concerning <<
    message.replace(0, 4, "").replace(message.find("even"), 4, "only")
           .replace(message.find(" don't"), 6, "");

    // Expected: "I have heard it works only if you believe in it"
    return message;
}
```

The Preprocessor

- When you compile a program, the compiler doesn’t compile it exactly as you’ve written it
- Before compilation, there is a **preprocessing** stage
- The preprocessor does *text replacement* on your source files without modifying them — in memory or with temp files
- Carryover from C
- Nowadays, lots of push to get rid of the preprocessor in modern C++

Things the preprocessor can do

- Define constant literals
- Define inline functions via macros with arguments
- Include libraries
- Conditional compilation
- Configure compiler behavior with #pragma

Why you should minimize your use of the preprocessor

- Modern C++ has better compiler features that achieve the same things (inline functions, modules starting in C++20)
- The preprocessor doesn’t understand C++ semantically or syntactically like the compiler does. It just finds and replaces text!
- Can introduce issues that become impossible to debug
  - e.g. you can smite your enemies with **#define if(x) if(!(x))**
- With C++17 we still can’t entirely get away from the preprocessor, but we can keep the usage very limited

Include guards

- Now that we’ve talked about including modules, and forward-declaring values without defining them you’ve probably put together why header files are useful
- Because #include relies on the preprocessor, we need ***include guards*** to prevent the same text being copypasted more than once

Include guards

- Old way – wrap compilation in a conditional
- “if **MY_COOL_LIBRARY_H** isn’t defined, define **MY_COOL_LIBRARY_H** and compile this code.”

```
1   #ifndef MY_COOL_LIBRARY_H
2   #define MY_COOL_LIBRARY_H
3
4   #include <string>
5
6   namespace MyCoolLibrary
7   {
8   constexpr int MY_COOL_CONSTANT{69};
9   int my_cool_library_function();
10  } // namespace MyCoolLibrary
11
12  #endif // MY_COOL_LIBRARY_H
```

Include guards

- Nowadays, most modern compilers let us use **#pragma once**
- ***#pragma directive*** – a special purpose directive that is used to turn on or off some compiler features
- Support for features is compiler specific

```
1  // preprocessor directive which tells the compiler
2  // to only include the following code once
3  #pragma once
4
5  #include <string>
6
7  namespace MyCoolLibrary
8  {
9  constexpr int MY_COOL_CONSTANT{69};
10 int my_cool_library_function();
11 } // namespace MyCoolLibrary
```

Scope

- Curly braces determine the scope of an expression
- Objects are destroyed when they go out of scope
- You can’t refer to objects not in the current scope
- This will prove to be an essential property of modern C++, especially when we get to smart pointers, locking mutexes, etc

```cpp
{
    // this a will go out of scope and be destroyed
    // once the flow of execution exits these curly brackets.
    double a = 0.420;
}
cout << a << endl; // won't compile. a went out of scope
```

```
[ 41%] Building CXX object CMakeFiles/scope.dir/src/apps/namespaces_and_scope.cpp.o
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp: In function 'int main(int, char**)':
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp:35:13: error: 'a' was not declared in this scope
   35 |     cout << a << endl; // won't compile. a went out of scope
      |             ^
```

Namespaces

```
#include "cards.hpp"
#include "ipod.hpp"

Shuffle();
// the music player or shuffling a deck of cards?
```

- Including multiple libraries could lead to ***naming collisions*** if those libraries define the same functions or terms

Namespaces

- Solve this by using ***namespaces***
- “::” – ***Scope Resolution*** operator
- refer to some object or item within the scope of a namespace or class

```
#include "cards.hpp"
#include "ipod.hpp"


Cards::Deck deck = Cards::Shuffle();
Apple::iPod device = Apple::iPod::Shuffle();
```

Namespaces

```
namespace MyCoolNamespace {
    int a{42};

    namespace MyNestedNamespace {
        std::string a {"forty two"};
    } // namespace MyNestedNamespace

} // namespace MyCoolNamespace

namespace MyOtherNamespace {
    short a {420};

} // namespace MyOtherNamespace
```

The “using” keyword

- Serves multiple purposes
- Define type aliases
  - using SomeType<T> = fully::qualified::type<T>
  - Using IntVector = std::vector<int>
- Bring namespaces into the current scope
  - using namespace std

Be careful with the using keyword

```cpp
// The following is a demonstration of why it's a bad idea to do "using namespace"
// instead of "using Namespace::ThingIWant"
using namespace MyCoolNamespace;
using namespace MyCoolNamespace::MyNestedNamespace;
using namespace MyOtherNamespace;

cout << a << endl; // won't compile. a is ambiguous.
```

```
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp: In function 'int main(int, char**)':
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp:43:13: error: reference to 'a' is ambiguous
   43 |     cout << a << endl; // won't compile. a is ambiguous.
      |             ^
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp:14:21: note: candidates are: 'std::string MyCoolNamespace::MyNestedNamespace::a'
   14 |         std::string a {"forty two"};
      |                     ^
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp:11:9: note:                 'int MyCoolNamespace::a'
   11 |     int a{42};
      |         ^
/code-examples/W2-basics/src/apps/namespaces_and_scope.cpp:20:11: note:                 'short int MyOtherNamespace::a'
   20 |     short a {420};
      |           ^
gmake[2]: *** [CMakeFiles/scope.dir/build.make:76: CMakeFiles/scope.dir/src/apps/namespaces_and_scope.cpp.o] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:145: CMakeFiles/scope.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
```

- Hilariously, C++ has given us the tools to resolve naming collisions and then just go on to reintroduce them again

Putting it all together: Writing our first library

- my_cool_library.hpp: declarations
- my_cool_library.cpp: definitions
- my_cool_executable.cpp: program that uses the library

```
add_library(my_cool_library src/lib/my_cool_library.cpp)
target_include_directories(my_cool_library PUBLIC include)

add_executable(my_cool_executable src/apps/my_cool_executable.cpp)
target_link_libraries(my_cool_executable my_cool_library)
```

Why we need include guards

```cpp
#include <iostream>
#include "my_cool_library.hpp"
// without the include guard,
// everything from my_cool_library.hpp will be blindly pasted in again
#include "my_cool_library.hpp"

int main(int argc, char** argv) {
    std::cout<< MyCoolLibrary::MY_COOL_CONSTANT << std::endl;
    std::cout<< MyCoolLibrary::my_cool_library_function() << std::endl;
    return 0;
}
```

```
[ 91%] Building CXX object CMakeFiles/my_cool_executable.dir/src/apps/my_cool_executable.cp
p.o
In file included from /code-examples/W2-basics/src/apps/my_cool_executable.cpp:5:
/code-examples/W2-basics/include/my_cool_library.hpp:9:15: error: redefinition of 'constexp
r const int MyCoolLibrary::MY_COOL_CONSTANT'
    9 | constexpr int MY_COOL_CONSTANT{69};
      |               ^~~~~~~~~~~~~~~~
In file included from /code-examples/W2-basics/src/apps/my_cool_executable.cpp:2:
/code-examples/W2-basics/include/my_cool_library.hpp:9:15: note: 'constexpr const int MyCoo
lLibrary::MY_COOL_CONSTANT' previously defined here
    9 | constexpr int MY_COOL_CONSTANT{69};
      |               ^~~~~~~~~~~~~~~~
gmake[2]: *** [CMakeFiles/my_cool_executable.dir/build.make:76: CMakeFiles/my_cool_executab
le.dir/src/apps/my_cool_executable.cpp.o] Error 1
gmake[1]: *** [CMakeFiles/Makefile2:223: CMakeFiles/my_cool_executable.dir/all] Error 2
gmake: *** [Makefile:91: all] Error 2
root@4589cdd53c91:/code-examples/W2-basics#
```

Additional Resources

- https://gcc.gnu.org/onlinedocs/gcc-6.3.0/cpp/Macro-Pitfalls.html#Macro-Pitfalls