UOW AUSTRALIA

CSCI251 Spring-2022 Advanced Programming

C++ Foundations II:

Getting started with Procedural Programming

Outline

- Procedural programming.
- Introducing `main()` …
- … explaining **Hello World!**
- Comments.
- Primitive types, variables and memory.
- Functions.
- Multiple files.

Procedural programming

- In procedural programming we typically focus on a specific aim, or end result.
  - The aim is often fairly fixed.
    - If there need to be changes they are often localised.
  - We don't have/need the abstraction that is common within object oriented programming.
- Code is written in a step-by-step way.
  - Typically small, and should be fairly easy to follow.

- With the code being written for a specific purpose it's possible to produce high performance code.
  - But the code has limited re-use value.
- With something like Object Oriented Programming (OOP), you can manage a larger code base that makes extensive use of re-use and allows for the code to be readily developed in independent teams.
- With procedural programming you tend to get a better understanding of the step by step operations, whereas OOP tends to hide the details.

- In procedural programming, procedures, also called routines, subroutines or functions, are declared or defined independent of main (`main()`) program construct.
  - `main()` is similar to a `main()` method in Java, but not the same.
- The program is a list of procedure calls.
  - Effectively a list of operations that change the state.
    - The state being the values of the data attributes.

Step 1: Call procedure B
Step 2: Call procedure A
Step 3: Call procedure C
Step 4: Call procedure B
Step 5: Call procedure C
Step 6: Call procedure A
Step 7: Call procedure A

- It is possible to write programs without any procedures or “function calls”.
- Such code might be referred to as unstructured or sequential.
- While this may be appropriate for simple tasks there are advantages in using functions:
  - Code can be re-used within a program.
    - Test once, changes to the same logic don’t need to be applied in multiple places, …
  - Code defined to carry out a specific function can be easily transferred to another program.
  - Program flow can be more readily tracked.
    - This makes it easier to read and helps with bug fixing.

- Individual functions contain exact rules regarding the input and the output.
  - For example; exactly two integers as input, and one integer as output.
- This exactness is one of the major disadvantages with procedural languages.
  - There is a need to keep track of all the detail.
- Another major disadvantage is that *similar but not identical* pieces of code must be rewritten.
  - For example: The procedure for calculating the area of a rectangle would often be different from the procedure to calculate the area of a triangle; even though both are polygons.

Introducing `main()` …

UOW AUSTRALIA

- Java: every method/function needs to be in a class.
  - Java 8 added functional interfaces, a little different.
- In a lot of languages, including C++, you can have standalone functions.
- In both Java and C++, `main()` serves as an entry point to the program.
- In Java applications, the `main()` method (function) is a static public member of a Runnable class.
  - If you don't know what this means, don't stress…

- In contrast C++ programs **must** have a stand alone `main()` function, that should return an integer, an `int`, to the operating system.
  - A return value of 0 typically means a normal termination.
  - Some other value is usually associated with an error.

… explaining **Hello World!**

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

int main(){
      cout << "Hello World!" << endl;
      return 0;
}
```

- We have our stand alone `main()`, a special case of the general function syntax:

```
return_type function(arguments){ }
```

Types: return_type, arguments?

- Data can be specified to be of a particular type, that's the technical term.
  - Type provides a context to data, it tells us how the string of bits we are dealing with should be treated.
- The **return_type** tells us what the output looks like, the **arguments** tell us what the input looks like.

Type and context…

- So far we have come across `int`, for integer.
- It's reasonable that we can perform arithmetic operations (+,-,…) on integers, and therefore should be able to on `int`'s.
- But if we have something like stores letters, characters ◊ `char`, it's not so clear those operations should make sense.

Primitive types in C++

UOW AUSTRALIA

- The types `int`, `double` and `char`, are all primitive or basic or built-in types in C++.
  - Actual sizes are compiler dependent.

Type	Meaning	Minimum size	
bool	Boolean	1 byte	Java: boolean
char	Character	8 bits	ASCII vs 16 bit Unicode for Java
wchar_t	Wide character	16 bits	
char8_t	Unicode character (UTF-8)	8 bits	Since C++20
char16_t	Unicode character (UTF-16)	16 bits	Since C++11
char32_t	Unicode character (UTF-32)	32 bits	Since C++11
short	Short integer	16 bits	
int	Integer	16 bits	
long	Long integer	32 bits	Since C++11
long long	Long integer	64 bits	Since C++11
float	Single-precision floating-point	4 bytes	
double	Double-precision floating-point	8 bytes	
long double	Extended-precision floating-point	8 bytes	

Signed or unsigned

- Other than `bool`, `wchar_t`, `char8_t`, `char16_t`, and `char32_t`, the rest types can have a qualifier `signed` or `unsigned`.
- Variables that are declared as `unsigned` only represent values greater than or equal to zero.

```
unsigned long long rats_present = 1;
unsigned long long cats_present = -1;
```

- With integer overflow …

```
std::cout << cats_present << std::endl;
```

```
18446744073709551615
```

Back to `main()`

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

int main(){
      cout << "Hello World!" << endl;
      return 0;
}
```

- `#include` is similar to `import` in Java.
- It brings in a header, `iostream` here.
  - Effectively a collection of code written elsewhere.
- The header `iostream` is the part of the standard library for C++ associated with Input/Output.
- Bring in your own files using something like
  `#include "My_file.h"`
- The `.h` suffix conventionally indicates header.
- More on libraries and pre-processing later …

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

int main(){
     cout << "Hello World!" << endl;
      return 0;
}
```

- Having the library iostream allows us to access input and output streams types, `istream` and `ostream`, representing input and output streams respectively.
  - Input: Use standard in: object `cin` of type `istream`.
  - Output: Use standard out: object `cout` of type `ostream`.

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

int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

- So, what about the rest of the line?
- `<<` is the output or insertion operator.
  - It pushes the value right operand, in this case the literal `"Hello World!"`, to the buffer for the stream specified by the left operand, `cout`, …
  - … where it may sit until flushed to the stream by ...
- `endl`, a manipulator.
  - Manipulators modify a stream.

Literal or variable ϑ

- Literals have explicit fixed values.

```
cout << "Whatever!" << endl;
```

- Variables do not.

```
cout << variable << endl;
```

- It's possible variables are fixed but the value isn't explicit here.

Output chaining …

```
cout << "Hello World!" << endl;
```

- Generally, since << is left-associative,

  `return X <<(ostream X, values)`

- … the left referenced `ostream` is returned and the next values in the chain added to it.
- Consider `cout << b << c << endl;`

  `(cout << b) << c << end;`
- Stream (buffer) contains b: `(cout << c) << endl;`
- Stream (buffer) contains b, c: `cout << endl;`
- The manipulator `endl` writes the stream out from the memory buffer to the output stream.
  `flush` is also a manipulator. Practice 1 !

- That just leaves …

```
using namespace std;
```

- This lets us avoid a longer version of …

```
cout << "Hello World!" << endl;
```

- Specifically …

```
std::cout << "Hello World!" << std::endl;
```

- Namespaces are organisational tools.
  - We can use `X::Y` to refer to `Y` in namespace `X`.
  - They can help avoid name clashes.
- The standard namespace (`std`) contains a lot of common C++ functionality.

Not `using namespace std;`

- It's good practice not to use …

```
using namespace std;
```

- … because doing so risks collisions between names in the standard library and names you have used.
  - It's convenient for simple examples though, so a fair few of our examples in the notes *will* use it…
- It's not unusual to see selective use of `using`, something like the following:

```
using std::cout;
using std::cin;
using std::endl;
```

Scope …

- Scope is the context of an entity, like a variable.
- It's where the name of the entity is visible, so we can refer to the entity.
- Using a namespace brings the entities in a namespace into the current scope.
- Something before `main()` has global scope, it's visible anywhere after that in our file, and potentially in other files if we use `extern`.
  - Usually avoid global variables.
- Generally scopes are defined by { … }.
  - Like around `main`, and any other function.

```
int x = 5;          // global scope/variable

int main(){
    int y = 10;   // local scope/variable
}
```

- As stated in the previous set of notes, functions can be standalone in C++, so independent of any class.
- A variable declared within a function is local to that function.

Practice 2!

- A variable defined within braces, and this is typical for loops for example, has local scope within that control structure.

More `main()` …

- The form of `main()` didn't have arguments…

```
int main(){ }
```

- … but it can have them...

```
int main( int argc, char *argv[]){ }
```

- The first parameter (`argc`) is the number of command-line arguments, *including the name of the executable itself*. c is for count.
- The second parameter is an array of C-style strings, so a sequence of characters, terminated with a special null character (`\0`).

- Lets have a look and see how an example of this runs on Ubuntu.

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

int main(int argc, char* argv[]){
      for ( int i=0; i < argc; i++ )
      {
        cout << "arg " << i << " " << argv[i] << endl;
      }
      return 0;
}
```

- This form of `main()` get parameters (input) from the command line when we run a program.
- The first line is the function header:

```
int main(int argc, char* argv[])
```


Practice 3!

`stoi` and friends …

- The function `stoi` is often used to convert arguments to integers.
- You should be exploring this and the related functions in the lab.
- There are alternatives, such as using stringstreams.

Input … `cin` and >>

- So we should look at an example of `cin`, standard in, and >>, the input or extraction operator.

```
# include <iostream>
using namespace std;
int main(){
        string name;
        int age;

        cout << "Enter a first name and age: ";
        cin >> name >> age;
        cout << name << " is " << age << " years old." << endl;
        return 0;

}
```

- Unlike in Java, the name of this program doesn't need to be tied to a class name.

But wait … how did these work

```
cin >> name >> age;
cout << name << " is " << age << " years old." << endl;
```

- … when we have different arguments for the operators >> and <<?
  - Operators are symbols determining particular functionality for an expression.
- These work because the insertion and extraction operators are overloaded to work with the primitive types.
  - Remember type and context earlier…
- An overloaded operator has different functionalities, depending on the arguments given to it.

- So the following all work …

```
int k = 2;
doubled =4/5;
charc = 'x';
cin >> k;                  // read an int
cin >> d;                  // read a double
cin >> c;                  // read a char
cout << k << endl; // write an int
cout << d << endl; // write a double
cout << c << endl; // write a char
```

- Overloading operators is possible in C++, and it's similar to overloading methods in Java.
- We will later expect to use << and >> with our own types.

Practice 4!

Commenting in C++

- It's good practice to include some comments on what you want parts of your code to do …
- Commenting syntax for C++ is the same as for Java.
- Two basic comment types:
  - line comments: `//`
  - block comments: `/* multiple lines */`

```
// This is a one line comment

float price;   //retail price

/* This is a comment that covers
   a block over two lines        */

 /**********************************
  * This is another block comment *
  **********************************/
```

- The third type of comments are the documentation comments.
- A tool such as Doxygen can be used to generate documentation from them.

```
/**
 * Various notes here …
 * Various other notes …
 * …
 */
```

- Doxygen is available from: https://www.doxygen.nl/index.html

Comments and variable names

- Comments should provide clarity to someone reading your code, not get in the way.

```
int intB;       //building number
float price;    //retail price
```

Or ...

```
int building_number;
int buildingNumber;
float retail_price;
float retailPrice;
```

https://www.defprogramming.com/

- "The proper use of comments is to compensate for our failure to express ourselves in code."
  - Robert C. Martin

- "The best reaction to "this is confusing, where are the docs" is to rewrite the feature to make it less confusing, not write more docs."
  - Jeff Atwood

Variable declaration and assignment

- We have seen this a few times already.

```
variable_type variable_name = variable_value;
double d = 4/5;
unsigned int cats_present = 3;
```

- We can chain the operator = too, but = is right associative …

```
int x, y, z;
x=y=z=5;
```

- So `z` is set to `5`, then `y` to `z`, then `x` to `y`.

Practice 5!

Warning: Not initialising …

```
public class Add {

   public static void main(){

      int number1=1, number2 = 2, sum;
//       sum = number1 + number2;

      System.out.println("Sum of these numbers: "+sum);
   }
}
```

- Java doesn't let you use uninitialized variables.

```
12:16:05 $ javac Add.java
Add.java:8: error: variable sum might not have been initialized
    System.out.println("Sum of these numbers: "+sum);
                                                 ^
1 error
```

```
#include<iostream>

int main(){

        int number1 = 1, number2 = 2, sum;
//      sum = number1 + number2;

        std::cout << sum << std::endl;

        return 0;
}
```

- C++ does!

```
12:20:53 $ g++ Sum.cpp
12:20:56 $

12:20:56 $ g++ Sum.cpp -Wall
Sum.cpp: In function 'int main()':
Sum.cpp:5:6: warning: unused variable 'number1' [-Wunused-variable]
 int number1=1, number2 = 2, sum;
     ^~~~~~
Sum.cpp:5:17: warning: unused variable 'number2' [-Wunused-variable]
 int number1=1, number2 = 2, sum;
             ^~~~~~~
Sum.cpp:8:15: warning: 'sum' is used uninitialized in this function [-Wuninitialized]
 std::cout << sum << std::endl;
           ^~~
12:21:09 $
```

Linked to memory

- The primitive types map directly on to memory entities like bytes and words, entities that most processors are designed to work with.
- This allows C++ to efficiently use the hardware, without there being an abstraction in between.
- Memory is effectively seen as a sequence of bytes, each typed object is given a location in memory, and values are placed in such objects.

- We refer to and access the locations using pointers.
  - Pointers are variables that contain locations.
- We will leave pointers for now, and return to them when we look at arrays and dynamic memory.
- Pointers play a critical role in C++.

Functions …

- Procedural suggests procedures, or functions, should be used …

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

void print() {
      cout << "Hello world!" << endl;
}

int main() {
      print();
      return 0;
}
```

- Note: `print()` isn't part of a class.

- The function `print()` needs to be declared prior to reaching `main()`, otherwise the compiler won't recognise it.
- The structure of `main()` often tells a story about what our code does, and sometimes it's clearer if definitions are out of the way.
  - Declarations need to be prior to `main()`, definitions don't.
- So, it's not unusual to separate the declaration and the definition of functions.
  - When we do this we have a forward declaration, using the function header followed by a semi-colon `;`.

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

void print();     // function prototype

int main() {
  print();        // function call
  return 0;
}

// function definition
void print(){
  cout << "Hello world!" << endl;
}
```

This is a forward declaration.

Define once, declare as often as you need!

- Declaring sets up a relationship between a name and it's purpose (variable,function,class).
  - Declaring says something is going to be defined.
- Defining allocates memory and puts the data/code in it.
  - Declaring doesn't involve allocating memory, and we cannot allocate memory multiple times.

Multiple files …

- Code is often, usually and preferably, spread across multiple files.
- It is often helpful to put all of the functions into other files.
- This is particularly useful if there are a lot of functions, and/or classes, and some of them are shared between programs.

Direct including other source files?

Hello.cpp

```
#include "print.cpp"

int main() {
        print();
        return 0;
}
```

print.cpp

```
#include <iostream>

void print() {
    std::cout << "Hello world!" << std::endl;
}
```

Which one will work?
1. $ g++ -c Hello.cpp print.cpp -o Hello.o print.o
2. $ g++ -c Hello.cpp -o Hello.o

Which one will work?
1. $ g++ Hello.o print.o -o Run
2. $ g++ Hello.o -o Run

Which step needs to rerun when print.cpp is changed?
1. Compilation
2. Linking
3. Compilation and Linking

Bad Practice!

Any change to print.cpp will result in re-compilation of Hello.cpp

Including a head file …

print.h

```
void print();
```

This is a forward declaration

Hello.cpp

```
#include "print.h"

int main() {
      print();
      return 0;
}
```

print.cpp

```
#include <iostream>

void print() {
    std::cout << "Hello world!" << std::endl;
}
```

Which one will work?

1. `$ g++ -c Hello.cpp print.cpp -o Hello.o print.o`
2. `$ g++ -c Hello.cpp print.cpp print.h –o Hello.o print.o`

Which one will work?

1. `$ g++ Hello.o print.o –o Run`
2. `$ g++ Hello.o –o Run`

Which one(s) need to be recomile if print.cpp is changed?
Hello.cpp, print.cpp or print.h

Good Practice!

Any change to print.cpp will result in re-compilation of print.cpp only

Other benefits

1. Separation of interface and implementation
2. Code Reusability
3. Encapsulation
4. Code organization

…

- Typically all data structure declarations, and function prototypes that you want to access in other files should be declared in header (`.h`) files, such as `print.h`.
- The definitions of those declarations go in the implementation file, such as `print.cpp`.
- Don't put `using namespace std;` in the included file.
  - You cannot turn it off so it applies to the rest of the file where the `#include` statement is. Λ

- Do include external functionality, like `iostream`, that you need in a file to be included, in the to be included file.
- You wouldn't necessarily need to include that external functionality in main but you shouldn't really assume you can get it through the header file of someone else.
- The file that contains `main()` is sometimes referred to as the driver file.
  - It doesn't typically have a paired header, main.h or similar, whereas usually the other code you write will come in file pairs, `myFunctions.h`, `myFunctions.cpp`.

C++ libraries …

- Sometimes we want to use functions, or classes, that other people have written.
- In particular, we often want to use the standard library.
  - Each C or C++ standard library has a corresponding header file, like `iostream`.
- These header files contain:
  - Function prototypes.
  - Definitions of various data types.
  - Definitions of constants.
  - Declarations of objects.

- For example, the math library, `cmath`, contains a square root function.

```
# include <iostream>
# include <cmath>
using namespace std;

int main() {
        double a;
        cout << "Enter a number: ";
        cin >> a;
        cout << "Square root " << a << " = " << sqrt(a) << endl;
        return 0;
}
```

- More on the library can be found at http://www.cplusplus.com/reference/cmath/

Warning: Is no return okay?

UOW AUSTRALIA

- If you write your `main()` and leave out …

```
return 0;
```

- … your code still runs.
- Whatever is most recently in memory is returned so you can get away without it.
  - You will likely see examples where it's not there, sometimes in this subject to save space, but in practice it's best to explicitly include it.
- The value returned is not necessarily going to be 0 and if there is a system test looking for 0 you will likely fail it.