C++ Programming Bootcamp 2

COSC2802

DAY 1

Day 1 Workshop Overview

1. C++ Introduction

    Why C++?

    C++ ⟺ Java

    C++ Program Structure: first Program and key parts of C++ program

2. Variables and Assignment

3. Branches

4. Loops (iteration statements)

C++ Introduction

Why C++

Learning Programming Skills & Techniques:

Dynamic Memory management; More explicit program control

- far greater understanding of our programming/computers work

If you can learn C++: you can learn basically any programming language

Highly hardware focused:

Allows greater optimization / efficiency

Used extensively by engineers (electrical, mechanical, etc)

Career Opportunities: C++ is still widely used in the industry, and there is a high demand for skilled C++ developers.

C/C++ Versions?

- C++ was originally an extension to C
  - C is a legal subset of C++
  - Biggest introduction are Classes, Generics and the STL (standard template library)
  - BC2 and PS2 are C++, but many concepts fine in C

- C++ has many ***standards*** that require standard compliant compliers to consistently handle:
  - C++11 (2011): major overhaul to the languages
  - C++14 (2014): additional language features, consistency updates, bug fixes
  - **C++17 (2017):** features to improve the usability, performance, code safety and aid usability. This is the version for BC2 and PS2

C++ and Java: what is the same?

- Comments
- Some Types
  - bool
  - int
  - float/double
  - char
- Operators
  - Arithmetic
  - Comparison
- Selection
  - if / elseif / else
- Iteration
  - While
  - For

C++ and Java: what is different?

- Standard I/O
  - cout / cin
- Types
  - Strings
  - Extended types
  - Implicit casting
- Arrays
- Declarations
- Functions
  - Parameter Passing
- #defines
- Global Variables
- Namespaces
- Declare & Initialise?

C++ Program Structure

(a first look)

Program Structure

Java

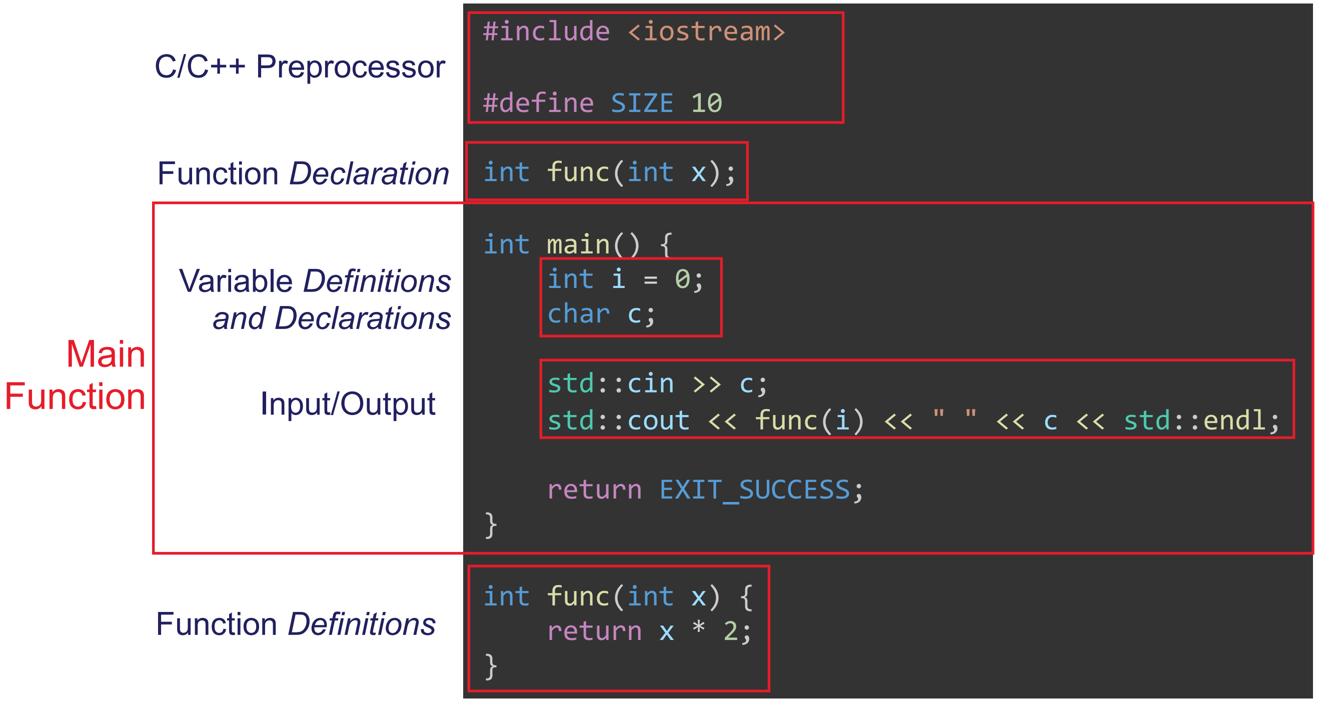
```java
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

C++

```cpp
#include <iostream>

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

Program Structure



```
#include <iostream>

#define SIZE 10

int func(int x);

int main() {
    int i = 0;
    char c;

    std::cin >> c;
    std::cout << func(i) << " " << c << std::endl;

    return EXIT_SUCCESS;
}

int func(int x) {
    return x * 2;
}
```

Variables/Assignment

Types

- **Similar**
  - Bool, int, float, double, char
  - …
- **Different**
  - Auto specifiers
  - Modifiers and Qualifiers
  - Type Casting
  - Strings: C-Style character string ⟺ String class (STL)
  - Extended types e.g. Standard Type Library (STL)
  - ...

Types: primitive built-in

C++ Types include:

- Bool (true / false)
- Char (‘a’, ‘b’, ‘\n’, …)
- Int (1, 2, 3, …)
- Float (1.1, 2.2, 3.1415, … single precision 32 bits of memory)
- Double (higher precision - 64 bits of memory)
- void (valueless)
- …

Typically, we have to declare the type of all variables when we create them, but this can be circumvented.

Operators:

Once you have variables what can we actually do with them?

The most basic statements of code are mathematics based:

1. Assignment (int x = 10;)
2. Addition (*a* + *b*)
3. Subtraction (a – b)
4. Multiplication (a * b)
5. Division (a / b)
6. Modulus (a % b)

Then we have some more interesting ones:

1. Increment (a++;), adds 1 to a
2. Decrement (a--;), subtracts 1 from a

Auto Specifier

- Supported since C++11
- Use *auto* in variable declaration as type specifier causes compiler to automatically deduce the type from the initializer. Example:

```
auto i = 5;          // causes type of i to be int
auto j = 5.0;        // causes type of j to be double
```

Advantages:

- can lead to MORE concise and readable code

Disadvantages:

- can lead to LESS readable or less maintainable code.

Types: type modifiers

- Type modifiers include signed, unsigned, long and short. Use as follows:

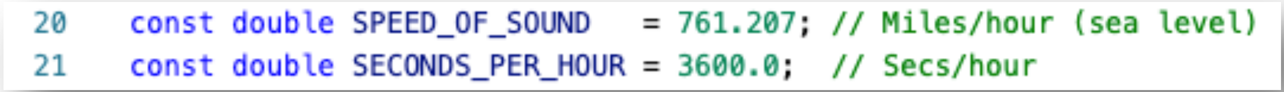
	signed	*unsigned*	*long*	*short*
int	✓	✓	✓	✓
char	✓	✓	✗	✗
double	✗	✗	✓	✗

- Modifiers alter base type so they more precisely fit application e.g

	# bytes	**Typical Range**
int	4	-2147483648 to 2147483647
long int	8	-9223372036854775808 to 9223372036854775807
short int	2	-32768 to 32767

Types: type qualifiers

- Precede constant variables (value can’t change) with keyword ***const*** e.g



```
const double SPEED_OF_SOUND    = 761.207; // Miles/hour (sea level)
const double SECONDS_PER_HOUR = 3600.0;   // Secs/hour
```

- Compiler reports error if attempt made in code to change variables value
- Good practice:
  - name constant variables using upper case letters with words separated by underscores, to make constant variables clearly visible in code.
  - Minimize number of literals in code

Preferred in modern c++ to #define

Type Casting

- **Implicit type casting** to convert between compatible types:

```
int num_int = 4;
float num_float = 12.4;

auto num = num_int + num_float;
std::cout << num << ", type: " << typeid(num).name() << std::endl;
```

- **Explicit type casting** using the *static_cast* operator converts the expression's value to the indicated type. For example:

```
auto num2 = num_int + static_cast<int>(num_float);
std::cout << num2 << ", type: " << typeid(num2).name() << std::endl;
```

NOTE: can also use C-style cast (similar to Java) but this can perform unsafe conversion without compile-time checks so program may crash or produce incorrect results.

Strings

String Literal surrounds a character sequence with double quotes

- Immutable (can't be changed), typically used for string constants known at compile-time
  - e.g. *std::cout* << *"Hello World!"*

Standard Library (STL) string

- The STL provides a string type:
  - Contained in <*string*> header
  - Within the *std* namespace
- *e.g. std::string str1 = "Hello";*
  - You can actually do things with them! i.e. Change them, concatenate them etc.

Example file: string_example1.cpp

std::string Class

- For this course, we will mostly use the std::string class
- As std::string is a class it has many useful methods

length()	Get the length of the string
at(int)	Get the character at the i-th position in the string
append(string)	Append another string to the end of 'this' string
substr(int,int)	Return the sub-string between two locations
c_str()	Get a c-style version of the string (if it is needed)

- Also works with typical operators
  - + to concatenate
  - == to compare
- See https://en.cppreference.com/w/cpp/string/basic_string

Standard I/O: C++ STL (cout)

***std::cout* is an object that points to the console of whatever machine you're using.**

Contained in <*iostream*> within the *std::* namespace

You can write to the console using <<, the output stream operator.

output_location << content_to_output

- *std::endl* is a system independent new line character.

Standard I/O: C++ STL (cin)

***std::cin* is an object that gets data from the console of whatever machine you're using.**

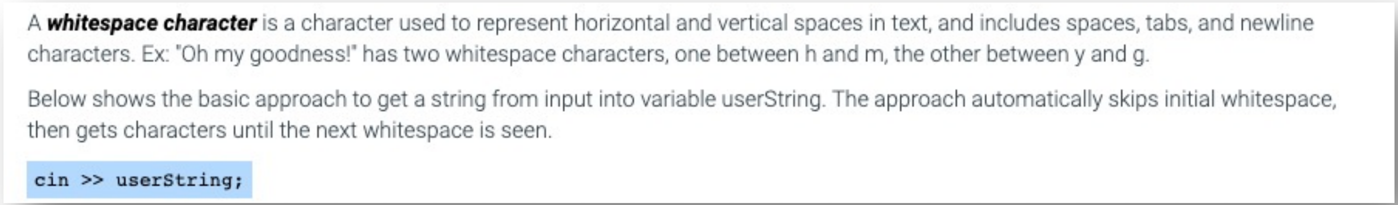
Contained in <*iostream*> within the *std::* namespace

You can read from the console using >>, the input stream operator.

std::cin >> variable

This does automatically interpret variable types (BUT IT CAN BE WRONG).

std::cin reads until the next white space character (so be careful if your input is a string! E.g., if the user types in: "Hello world!", cin may not work as expected)



A ***whitespace character*** is a character used to represent horizontal and vertical spaces in text, and includes spaces, tabs, and newline characters. Ex: "Oh my goodness!" has two whitespace characters, one between h and m, the other between y and g.

Below shows the basic approach to get a string from input into variable userString. The approach automatically skips initial whitespace, then gets characters until the next whitespace is seen.

```
cin >> userString;
```

Standard I/O: C++ STL

Other functions that can be used:

- std::getline() // gets entire line up to (but does not include) the next newline character ‘\n’

NOTE: Do not mix cin >> and getline()! Cin >> leaves new line characters in the input buffer, which can cause issues.

More on this:

- zybooks e.g. section: Input String Streams

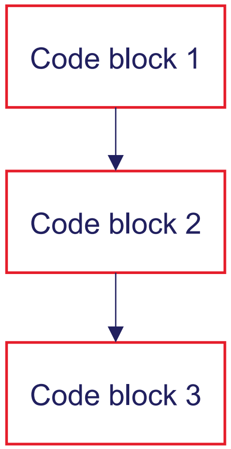
Program Flow Control

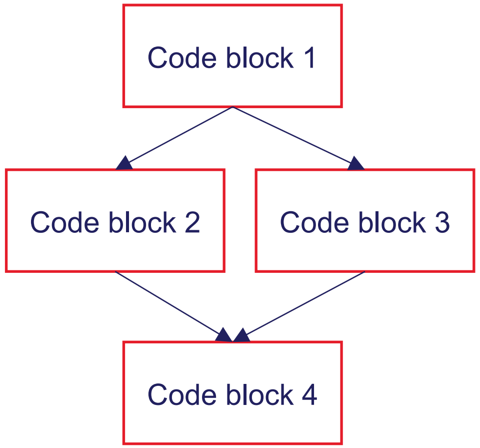
Control Structures

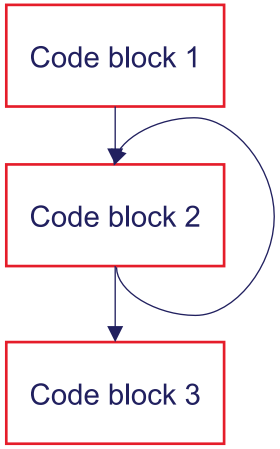
Sometimes we want to be able to run code in different ways; not all algorithms can be expressed solely as mathematical equations (addition, subtraction, multiplication, etc)

All programs can be written in terms of the following 3:

1. Sequence block – execute statements in the order they are written
2. Branches – execute one block of code or another depending on some condition
3. Loops – execute the same block of code multiple times







Branches

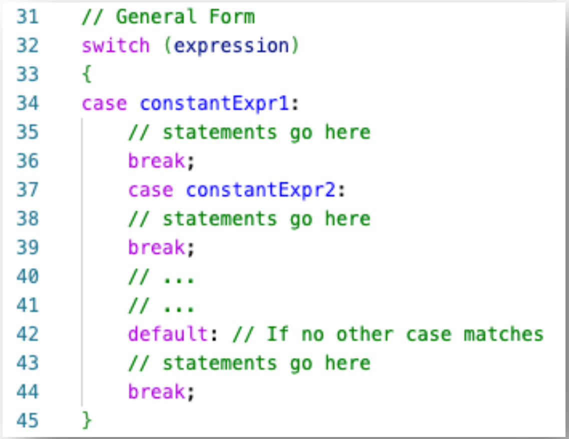
1. if-else branches: C++ and Java

- Similar Syntax with same keywords:
  - *if*, *else*, *else if*
- Same set of comparison operators, including:
  - ==, !=, <, >, <=, >=
- We can combine comparisons using Boolean logic:
  - &&, ||, !
- Type coercion: what happens if compare values of different types?
  - C++: compiler will try to convert one of the operands to the type of the
  - Java, the operands must have the same type – else error

2. Switch Statements

C++ switch statement

- Case values must be integral (represent whole numbers e.g. char, int, bool, …) or **enumeration ty**pe (see zybooks)
- Allows fall-through: if no break statement at the end of a case block, execution continues into next case block. Remember break in each case block



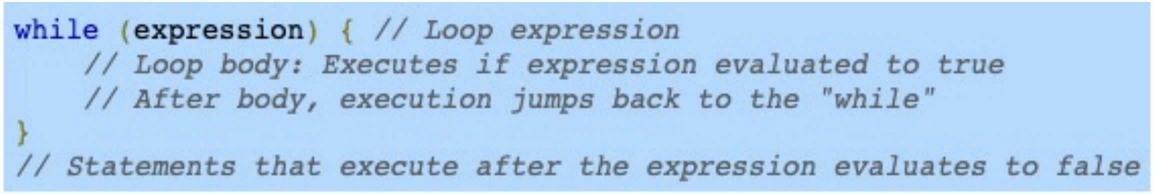
```
// General Form
switch (expression)
{
case constantExpr1:
    // statements go here
    break;
    case constantExpr2:
    // statements go here
    break;
    // ...
    // ...
    default: // If no other case matches
    // statements go here
    break;
}
```

NOTE: switch statement can be written using multi if-else BUT switch may be clearer

Loops

1. While loops

General Form:



```
while (expression) { // Loop expression
    // Loop body: Executes if expression evaluated to true
    // After body, execution jumps back to the "while"
}
// Statements that execute after the expression evaluates to false
```

Similar syntax and functionality C++ ⬄ Java:

- Repeatedly executes a block of code
- Each loop an expression is evaluated, if the expression is true, the block is executed, otherwise, end the loop and continue with the program.

NOTE: Always have an exit condition, infinite loops are bad practice!

You can also use *break;* to end a loop early.

2. For loops

General Form:

```
for (initialExpression; conditionExpression; updateExpression) {
   // Loop body
}
// Statements after the loop
```

The definition has three parts at the top:

- loop variable initialization; loop expression; loop variable update

The initialization is run once at the start of the loop.

The expression is run every at the start of each iteration, when the condition becomes false, the loop is exited.

The update expression is run once at the end of each iteration.

2. For loops: Example

```
int i;
...
for (i = 0; i < N; ++i) {
    ...
}
```

Starting with 0: from *i = 0* and check *i < N, increment i by 1*

This loop will run how many times???

TANGENT!

- The ++ operators:
  - ++i (prefix) increments i, then evaluates result; i++ (postfix) evaluates result then increments i.
  - Prefix generally recommended as safer

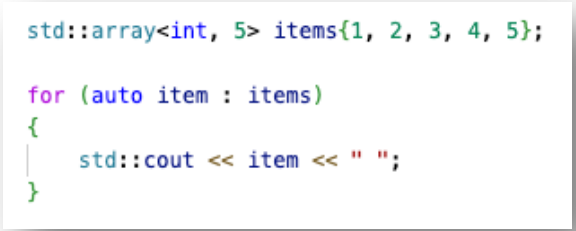
```
int i = 1;
std::cout << ++i << std::endl;
std::cout << i++ << std::endl;
std::cout << i << std::endl;
```

3. Range based for loops

Allows processing of all elements of a container without using a counter:

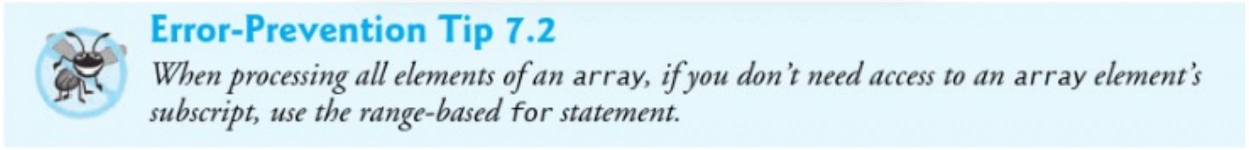
- Can’t go outside the bounds of a container
- Don’t need to do any bounds checking.

General form of range-based for loop:



```
std::array<int, 5> items{1, 2, 3, 4, 5};

for (auto item : items)
{
    std::cout << item << " ";
}
```

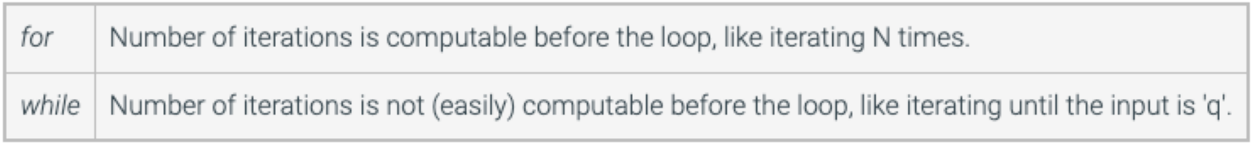


Error-Prevention Tip 7.2

When processing all elements of an array, if you don’t need access to an array element’s subscript, use the range-based `for` *statement.*

NOTE: changing values depends on scope of range variable (revisit this later)

While or For ?



for	Number of iterations is computable before the loop, like iterating N times.
while	Number of iterations is not (easily) computable before the loop, like iterating until the input is 'q'.