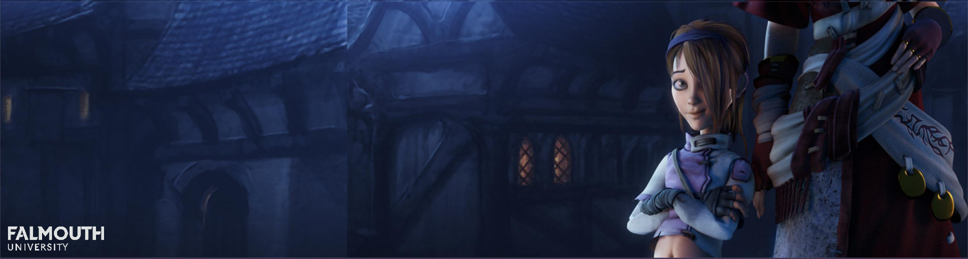

COMP110: Principles of Computing

Transition to C++ I

Learning outcomes

In this session you will learn how to...

- Use Visual C++ 2015 to create, compile and run a C++ application
- Declare variables in C++, and some of the basic types they can have
- Use various control structures in C++, including `if`, `switch`, `while`, `for` and `for each`
- Define your own C++ functions

Your first C++ program

Project setup

- Open **Visual Studio 2015** from the Start menu
- Click **New Project**
- Choose **Templates** $\rightarrow$ **Visual C++** $\rightarrow$ **Win32** $\rightarrow$ **Win32 Console Application**
- Choose an appropriate name and location, and click **OK**
- Click **Finish**
- If asked about source control, click **Cancel**

The code

```
// ConsoleApplication1.cpp : Defines the entry point
    for the console application.

#include "stdafx.h"

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

- Add the following line to the end of `stdafx.h`:

```
#include <iostream>
```

Running it

- Click ▶ Local Windows Debugger, or press **F5**

Running it

- Click ▶ Local Windows Debugger, or press **F5**
- It worked, but the window disappeared before we could see it!

Running it

- Click ▶ Local Windows Debugger, or press **F5**
- It worked, but the window disappeared before we could see it!
- Solution 1: click **Debug** → **Start Without Debugging**, or press **Ctrl + F5**
- Solution 2: click in the left margin next to the `return 0;` line to set a **breakpoint** — a red circle should appear. Then click ▶ Local Windows Debugger

Comments

```
// ConsoleApplication1.cpp : Defines the entry point
    for the console application.
```

Comments

```
// ConsoleApplication1.cpp : Defines the entry point
    for the console application.
```

- // denotes a single-line comment

Comments

```
// ConsoleApplication1.cpp : Defines the entry point
    for the console application.
```

- // denotes a single-line comment
- Equivalent of # in Python

Comments

```
// ConsoleApplication1.cpp : Defines the entry point ↩
    for the console application.
```

- `//` denotes a single-line comment
- Equivalent of `#` in Python
- ↩ denotes a line too long to fit on the slide — in your program this should be a single line

Comments

```
// ConsoleApplication1.cpp : Defines the entry point ↩
    for the console application.
```

- `//` denotes a single-line comment
- Equivalent of `#` in Python
- ↩ denotes a line too long to fit on the slide — in your program this should be a single line
- Multi-line comments, delimited by `/* */`, are also available

```
/* This is an example of a multi-line comment
   More comment text
   Even more comment text */
```

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

- `#include` imports definitions from a **header file**

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

- `#include` imports definitions from a **header file**
- Similar to `import` in Python

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

- `#include` imports definitions from a **header file**
- Similar to `import` in Python
- `#include "..."` (quotes) is used for headers in the current project

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

- `#include` imports definitions from a **header file**
- Similar to `import` in Python
- `#include "..."` (quotes) is used for headers in the current project
- `#include <...>` (angle brackets) is used for external libraries

The #include directive

```
#include "stdafx.h"
```

```
#include <iostream>
```

- `#include` imports definitions from a **header file**
- Similar to `import` in Python
- `#include "..."` (quotes) is used for headers in the current project
- `#include <...>` (angle brackets) is used for external libraries
- `stdafx.h` is the **precompiled header** file — for faster compilation, external library headers should be included here rather than in the main `.cpp` file

Entry point

```
int main()
```

Entry point

```
int main()
```

- All code must be inside a function

Entry point

```
int main()
```

- All code must be inside a function
- The **entry point** of an application is (almost) always named `main`

Entry point

```
int main()
```

- All code must be inside a function
- The **entry point** of an application is (almost) always named `main`
  - Some types of Windows GUI application use a different name for the entry point

Entry point

```
int main()
```

- All code must be inside a function
- The **entry point** of an application is (almost) always named `main`
  - Some types of Windows GUI application use a different name for the entry point
- `int` means the function returns a value of integer type

Entry point

```
int main()
```

- All code must be inside a function
- The **entry point** of an application is (almost) always named `main`
  - Some types of Windows GUI application use a different name for the entry point
- `int` means the function returns a value of integer type
- `()` means the function takes no parameters

Blocks and semicolons

```
{
    ...;
    ...;
}
```

Blocks and semicolons

```
{
    ...;
    ...;
}
```

- Curly braces are used to denote blocks

Blocks and semicolons

```
{
    ...;
    ...;
}
```

- Curly braces are used to denote blocks
- All statements in C++ end with a semicolon `;`

Blocks and semicolons

```
{
    ...;
    ...;
}
```

- Curly braces are used to denote blocks
- All statements in C++ end with a semicolon ;
- Unlike Python, C++ ignores whitespace (indentation and line breaks)

Blocks and semicolons

```
{
    ...;
    ...;
}
```

- Curly braces are used to denote blocks
- All statements in C++ end with a semicolon `;`
- Unlike Python, C++ ignores whitespace (indentation and line breaks)
- ... but whitespace is important for readability, so use it anyway

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement
- `std` is the **namespace** containing most of the C++ standard library

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement
- `std` is the **namespace** containing most of the C++ standard library
- `std::cout` is the console output stream

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement
- `std` is the **namespace** containing most of the C++ standard library
- `std::cout` is the console output stream
- `std::endl` is the end-of-line character

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement
- `std` is the **namespace** containing most of the C++ standard library
- `std::cout` is the console output stream
- `std::endl` is the end-of-line character
- To use `std::cout` and `std::endl`, it is necessary to `#include <iostream>`

Writing to the console

```
std::cout << "Hello, world!" << std::endl;
```

- Equivalent of Python's `print` statement
- `std` is the **namespace** containing most of the C++ standard library
- `std::cout` is the console output stream
- `std::endl` is the end-of-line character
- To use `std::cout` and `std::endl`, it is necessary to
  ```
  #include <iostream>
  ```
- `<<` is the **insertion operator** — used to write values to a stream

Exit code

```
return 0;
```

Exit code

```
return 0;
```

- Returning 0 from `main` tells the OS that the program completed successfully

Exit code

```
return 0;
```

- Returning 0 from `main` tells the OS that the program completed successfully
- Mainly useful for writing tools to be used in DOS/Windows batch scripts or Linux shell scripts — for our purposes, `main` will almost always return 0

FALMOUTH
UNIVERSITY

Variables and types

Variables

In Python, variables exist the moment they are assigned to:

```
a = 10
b = 20
```

Variables

In Python, variables exist the moment they are assigned to:

```
a = 10
b = 20
```

Variables can hold values of any type:

```
a = 10
a = 3.14159
a = "Hello"
```

Variables

In Python, variables exist the moment they are assigned to:

```
a = 10
b = 20
```

Variables can hold values of any type:

```
a = 10
a = 3.14159
a = "Hello"
```

In C++, variables must be **declared** before use, and must be given a **type**:

```
int a = 10;
int b = 20;
```

Variables

In Python, variables exist the moment they are assigned to:

```
a = 10
b = 20
```

Variables can hold values of any type:

```
a = 10
a = 3.14159
a = "Hello"
```

In C++, variables must be **declared** before use, and must be given a **type**:

```
int a = 10;
int b = 20;
```

Variables can only hold values of the correct type:

```
int a = 10;
a = 17;       // OK
a = "Hello"; // Error
```

Integers

- `int` is the basic data type for integers (whole numbers)

```
int a = 42;
int b = -74965;
int c = 0;
int d = 0x19FD; // Hexadecimal
```

Integers

- `int` is the basic data type for integers (whole numbers)

```
int a = 42;
int b = -74965;
int c = 0;
int d = 0x19FD; // Hexadecimal
```

- On Windows (32 and 64 bit), `int` can store numbers from -2^{31} to $2^{31} - 1 \approx \pm 2$ billion

Integers

- `int` is the basic data type for integers (whole numbers)

```
int a = 42;
int b = -74965;
int c = 0;
int d = 0x19FD; // Hexadecimal
```

- On Windows (32 and 64 bit), `int` can store numbers from -2^{31} to $2^{31} - 1 \approx \pm 2$ billion
- `unsigned int` stores **nonnegative** integers, from 0 to $2^{32} \approx 4$ billion

Integers

- `int` is the basic data type for integers (whole numbers)

```
int a = 42;
int b = -74965;
int c = 0;
int d = 0x19FD; // Hexadecimal
```

- On Windows (32 and 64 bit), `int` can store numbers from -2^{31} to $2^{31}-1 \approx \pm 2$ billion
- `unsigned int` stores **nonnegative** integers, from 0 to $2^{32} \approx 4$ billion
- Other integer types exist, for example `long long` is a 64 bit integer

Floating point numbers

- `float` and `double` can store floating point numbers (numbers with a fractional part)

```
double a = 3.14159;
double b = -42;
double c = 3.0e8; // Scientific notation
float d = 123.456f; // Note the 'f' suffix for float
```

Floating point numbers

- `float` and `double` can store floating point numbers (numbers with a fractional part)

```
double a = 3.14159;
double b = -42;
double c = 3.0e8; // Scientific notation
float d = 123.456f; // Note the 'f' suffix for float
```

- `float` uses less space, and can be slightly faster, but is less precise

Floating point numbers

- `float` and `double` can store floating point numbers (numbers with a fractional part)

```
double a = 3.14159;
double b = -42;
double c = 3.0e8; // Scientific notation
float d = 123.456f; // Note the 'f' suffix for float
```

- `float` uses less space, and can be slightly faster, but is less precise
- Generally `double` is the better choice

Characters

- `char` stores a single ASCII character

```
char foo = 'Q';
char bar = '7';
char baz = '@';
char space = ' ';
char newLine = '\n'; // Escape sequence
```

Characters

- `char` stores a single ASCII character

```
char foo = 'Q';
char bar = '7';
char baz = '@';
char space = ' ';
char newLine = '\n'; // Escape sequence
```

- `char` can also be thought of as an 8-bit integer, i.e. an integer between -128 and 127 — C++ makes no distinction between ASCII characters and their numerical codes

Booleans

- `bool` stores a boolean (true or false) value

```
bool isAlive = true;
bool isDead = false;
```

Vectors

- **Vectors** are the C++ equivalent of lists in Python

Vectors

- **Vectors** are the C++ equivalent of lists in Python
- Add `#include <vector>` to `stdafx.h`

Vectors

- **Vectors** are the C++ equivalent of lists in Python
- Add `#include <vector>` to `stdafx.h`
- `std::vector<T>` is a vector with elements of type `T`

Vectors

- **Vectors** are the C++ equivalent of lists in Python
- Add `#include <vector>` to `stdafx.h`
- `std::vector<T>` is a vector with elements of type `T`

```
std::vector<int> numbers = { 1, 4, 9, 16 };
numbers.push_back(25);
```

Strings

- C++ has two main data types for strings:
  - `char*` or `char[]`: low-level array of ASCII characters (more on arrays next week)
  - `std::string`: high-level string class

Strings

- C++ has two main data types for strings:
  - `char*` or `char[]`: low-level array of ASCII characters (more on arrays next week)
  - `std::string`: high-level string class
- Use `std::string` unless you have a compelling reason not to

Strings

- C++ has two main data types for strings:
  - `char*` or `char[]`: low-level array of ASCII characters (more on arrays next week)
  - `std::string`: high-level string class
- Use `std::string` unless you have a compelling reason not to
- Add `#include <string>` to `stdafx.h`

Strings

- C++ has two main data types for strings:
  - `char*` or `char[]`: low-level array of ASCII characters (more on arrays next week)
  - `std::string`: high-level string class
- Use `std::string` unless you have a compelling reason not to
- Add `#include <string>` to `stdafx.h`

```
std::string name = "Ed";
std::string message = "Hello " + name + "!";
std::cout << message << std::endl;
```

Enumerations

- An **enumeration** is a set of named values

Enumerations

- An **enumeration** is a set of named values

```
enum Direction { dirUp, dirRight, dirDown, dirLeft };

Direction playerDirection = dirUp;
```

Enumerations

- An **enumeration** is a set of named values

```
enum Direction { dirUp, dirRight, dirDown, dirLeft };

Direction playerDirection = dirUp;
```

- This is equivalent to using an `int` with 0=up, 1=right etc, but is more readable

Constants

- The `const` keyword can be used to define a “variable” whose value cannot change, i.e. read only

Constants

- The `const` keyword can be used to define a “variable” whose value cannot change, i.e. read only

```
const int x = 7;
std::cout << x << std::endl; // OK
x = 12; // Error
```

Declaring variables

- A variable declaration must specify a **type**, and one or more **variable names**:

```
int i, j, k;
bool isDead;
std::string playerName;
```

Declaring variables

- A variable declaration must specify a **type**, and one or more **variable names**:

```
int i, j, k;
bool isDead;
std::string playerName;
```

- A variable declaration can optionally specify an **initial value**:

```
int i = 0, j = 1, k = 2;
bool isDead = false;
std::string playerName = "Ed";
```

Initial values

- If the initial value is omitted, what happens depends on the type:

Initial values

- If the initial value is omitted, what happens depends on the type:
- Basic data types (`int`, `double`, `bool`, `char` etc): the value is undefined — whatever data happened to be in that memory location already

Initial values

- If the initial value is omitted, what happens depends on the type:
- Basic data types (`int`, `double`, `bool`, `char` etc): the value is undefined — whatever data happened to be in that memory location already
  - Your code should **never** read an uninitialised variable — doing so is **always** a bug

Initial values

- If the initial value is omitted, what happens depends on the type:
- Basic data types (`int`, `double`, `bool`, `char` etc): the value is undefined — whatever data happened to be in that memory location already
  - Your code should **never** read an uninitialised variable — doing so is **always** a bug
- Object types (`std::vector`, `std::string` etc): depends on the type (consult the documentation)

Initial values

- If the initial value is omitted, what happens depends on the type:
- Basic data types (`int`, `double`, `bool`, `char` etc): the value is undefined — whatever data happened to be in that memory location already
  - Your code should **never** read an uninitialised variable — doing so is **always** a bug
- Object types (`std::vector`, `std::string` etc): depends on the type (consult the documentation)
  - `std::vector` and `std::string` are both initialised to empty

Scope

- The **scope** of a variable is the region of the program where it exists

Scope

- The **scope** of a variable is the region of the program where it exists
- Generally the scope of a variable begins when it is declared, and ends when the block in which it is declared ends

Scope

- The **scope** of a variable is the region of the program where it exists
- Generally the scope of a variable begins when it is declared, and ends when the block in which it is declared ends

```
int x = 7;
if (x > 5)
{
    int y = x * 2;
    std::cout << x << std::endl; // OK
    std::cout << y << std::endl; // OK
}
std::cout << x << std::endl; // OK
std::cout << y << std::endl; // Error
```

Control structures

If statement

```
if (x > 0)
{
    std::cout << "x is positive" << std::endl;
}
else if (x < 0)
{
    std::cout << "x is negative" << std::endl;
}
else
{
    std::cout << "x is neither positive nor negative" ↩
        << std::endl;
}
```

If statement

- Works just like the `if` statement in Python

If statement

- Works just like the `if` statement in Python
- There can be zero, one or many `else if` clauses

If statement

- Works just like the `if` statement in Python
- There can be zero, one or many `else if` clauses
- The `else` clause is optional, but if present then there can only be one

Conditions

- Numerical comparison operators work just like Python:
  `==` `!=` `<` `>` `<=` `>=`

Conditions

- Numerical comparison operators work just like Python:
  `==  !=  <  >  <=  >=`
- Boolean logic operators look a little different

Conditions

- Numerical comparison operators work just like Python: `== != < > <= >=`
- Boolean logic operators look a little different

Python uses `and`, `or`, `not`

```
if not (x < 0 or x > 100) and not (y < 0 or y > 100):
    print "Point is in rectangle"
```

Conditions

- Numerical comparison operators work just like Python:
  `==  !=  <  >  <=  >=`
- Boolean logic operators look a little different

Python uses `and`, `or`, `not`

```
if not (x < 0 or x > 100) and not (y < 0 or y > 100):
    print "Point is in rectangle"
```

C++ uses `&&`, `||`, `!`

```
if (!(x < 0 || x > 100) && !(y < 0 || y > 100))
{
    std::cout << "Point is in rectangle" << std::endl;
}
```

Single-statement blocks

- In many cases, if a block contains only a single statement then the curly braces can be omitted

```
if (x > 0)
    std::cout << "x is positive" << std::endl;
else if (x < 0)
    std::cout << "x is negative" << std::endl;
else
    std::cout << "x is neither positive nor negative" ↩
        << std::endl;
```

Single-statement blocks

- Careful though! This can lead to obscure bugs

```
if (z == 0)
    x = 0; y = 0;
```

Single-statement blocks

- Careful though! This can lead to obscure bugs

```
if (z == 0)
    x = 0; y = 0;
```

- This is equivalent to

```
if (z == 0)
{
    x = 0;
}
y = 0;
```

- ... which is probably not what the programmer intended

Switch statement

```
switch (x)
{
case 0:
    std::cout << "zero" << std::endl;
    break;
case 1:
    std::cout << "one" << std::endl;
    break;
case 2:
    std::cout << "two" << std::endl;
    break;
default:
    std::cout << "something else" << std::endl;
    break;
}
```

While loop

```
while (x > 0)
{
    std::cout << x << std::endl;
    x--;
}
```

- Same as Python

Do-while loop

```
do
{
    std::cout << x << std::endl;
    x--;
} while (x > 0);
```

Do-while loop

```
do
{
    std::cout << x << std::endl;
    x--;
} while (x > 0);
```

- `while` loop checks the condition **before** executing the loop body

Do-while loop

```
do
{
    std::cout << x << std::endl;
    x--;
} while (x > 0);
```

- `while` loop checks the condition **before** executing the loop body
- `do-while` loop checks the condition **after** executing the loop body

Do-while loop

```
do
{
    std::cout << x << std::endl;
    x--;
} while (x > 0);
```

- `while` loop checks the condition **before** executing the loop body
- `do-while` loop checks the condition **after** executing the loop body
- e.g. if `x == 0` to begin with, the `while` body does not execute, the `do-while` body executes once

For-each loop

```
std::vector<int> numbers { 1, 3, 5, 7, 9 };

for each (int x in numbers)
{
    std::cout << x << std::endl;
}
```

For-each loop

```
std::vector<int> numbers { 1, 3, 5, 7, 9 };

for each (int x in numbers)
{
    std::cout << x << std::endl;
}
```

- This works like the `for` loop in Python
- Used for iterating over data structures

For loop

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

For loop

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- The `for` loop has three parts:

For loop

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- The `for` loop has three parts:
- The **initialiser** `int i = 0`
  - This is executed at the start of the loop

For loop

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- The `for` loop has three parts:
- The **initialiser** `int i = 0`
  - This is executed at the start of the loop
- The **condition** `i < 10`
  - The loop executes while this evaluates to `true`

For loop

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- The `for` loop has three parts:
- The **initialiser** `int i = 0`
  - This is executed at the start of the loop
- The **condition** `i < 10`
  - The loop executes while this evaluates to `true`
- The **loop statement** `i++`
  - This is executed at the end of each iteration of the loop
  - `i++` means "increment `i`" — this is shorthand for `i = i + 1`

For loops and while loops

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- Any `for` loop can easily be rewritten as a `while` loop

For loops and while loops

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- Any `for` loop can easily be rewritten as a `while` loop

```
int i = 0;
while (i < 10)
{
    std::cout << i << std::endl;
    i++;
}
```

For loops in C++ and Python

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

For loops in C++ and Python

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- In Python, this would be written as a for-each loop, first using the `range` function to construct the list of numbers $0, 1, 2, \ldots, 9$:

```
for i in range(10):
    print i
```

For loops in C++ and Python

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << std::endl;
}
```

- In Python, this would be written as a for-each loop, first using the `range` function to construct the list of numbers $0, 1, 2, \ldots, 9$:

```
for i in range(10):
    print i
```

- The C++ way doesn't require construction of a temporary list, so is more efficient

Socrative 6E8NSW3IN

What would the first code fragment print?

```
for (int i = 0; i < 10; i++)
    std::cout << i << " ";
```

Socrative 6E8NSW3IN

What would the second code fragment print?

```
for (int i = 0; i <= 10; i++)
    std::cout << i << " ";
```

Socrative 6E8NSW3IN

What would the third code fragment print?

```
for (int i = 0; i < 10; i += 2)
    std::cout << i << " ";
```

Socrative 6E8NSW3IN

What would the fourth code fragment print?

```
for (int i = 10; i < 0; i++)
    std::cout << i << " ";
```

Socrative 6E8NSW3IN

What would the fifth code fragment print?

```
for (int i = 10; i > 0; i++)
    std::cout << i << " ";
```

Socrative 6E8NSW3IN

What would the sixth code fragment print?

```
for (int i = 10; i > 0; i--)
    std::cout << i << " ";
```

FALMOUTH UNIVERSITY

Live coding: Hangman