COMP110: Principles of Computing

Transition to C++ II

Learning outcomes

In this session you will learn how to...

- Split your program into multiple files, and understand the difference between **source files** and **header files**
- Understand the C++ build pipeline, and the roles of the **preprocessor**, **compiler** and **linker**
- Use arrays, and the difference between creating them on the **stack** versus on the **heap**
- Define C++ functions, and how passing **by reference** differs from passing **by value**

FALMOUTH UNIVERSITY

Modular program design

Modular program design

- We saw in session 9 that **splitting your code into several files** is generally a good idea

Modular program design

- We saw in session 9 that **splitting your code into several files** is generally a good idea
- Python makes it easy: any .py file can be `import`ed on demand

Modular program design

- We saw in session 9 that **splitting your code into several files** is generally a good idea
- Python makes it easy: any .py file can be `import`ed on demand
- C++ is a little trickier...

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

A function **declaration** specifies everything **except** the code:

```
double average(double n1, double n2);
```

Definitions and declarations

A function **definition** specifies its name, return type, parameters, and the code it contains:

```
double average(double n1, double n2)
{
    return (n1 + n2) / 2.0;
}
```

A function **declaration** specifies everything **except** the code:

```
double average(double n1, double n2);
```

A declaration tells the compiler that this function exists, but is defined **elsewhere**

Sources and headers

- A C++ project contains two main types of file

Sources and headers

- A C++ project contains two main types of file
- **Source files** (.cpp) usually contain **definitions**

Sources and headers

- A C++ project contains two main types of file
- **Source files** (.cpp) usually contain **definitions**
- **Header files** (.h) usually contain **declarations**

Sources and headers

- A C++ project contains two main types of file
- **Source files** (.cpp) usually contain **definitions**
- **Header files** (.h) usually contain **declarations**
- For example, `myfile.cpp` may contain some function definitions, and `myfile.h` may contain the declarations for those functions

Sources and headers

- A C++ project contains two main types of file
- **Source files** (.cpp) usually contain **definitions**
- **Header files** (.h) usually contain **declarations**
- For example, `myfile.cpp` may contain some function definitions, and `myfile.h` may contain the declarations for those functions
- (Yep, that means you have to type the same thing twice in two different files...)

Example from last week

words.cpp

```
void readWords()
{
    std::cout << "Reading word list" << std::endl;
    // code omitted
}

std::string chooseRandomWord()
{
    // code omitted
}
```

words.h

```
#pragma once

void readWords();
std::string chooseRandomWord();
```

Example from last week

- `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`

Example from last week

- `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`
- `readWords()` and `chooseRandomWord()` are **declared** in `words.h`

Example from last week

- `readWords()` and `chooseRandomWord()` are **defined** in `words.cpp`
- `readWords()` and `chooseRandomWord()` are **declared** in `words.h`
- Any file which does `#include "words.h"` can call these functions as if they were declared in that file

How #include works

- `#include` works **exactly** as if the `#include`d file were copied and pasted at the point where the `#include` directive appears

How #include works

- `#include` works **exactly** as if the `#include`d file were copied and pasted at the point where the `#include` directive appears
- All header files should start with `#pragma once` — otherwise, `#include`ing the same file more than once will result in duplicate declaration errors

How #include works

- `#include` works **exactly** as if the `#include`d file were copied and pasted at the point where the `#include` directive appears
- All header files should start with `#pragma once` — otherwise, `#include`ing the same file more than once will result in duplicate declaration errors
- Putting an `#include` directive in the wrong place (e.g. inside a function) will result in weird compile errors

FALMOUTH UNIVERSITY

The build process

Executing programs

- CPUs execute **machine code**

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution
- There are three main ways of doing this:

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution
- There are three main ways of doing this:
  - An **interpreter** is an application which reads the program source code and executes it directly

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution
- There are three main ways of doing this:
  - An **interpreter** is an application which reads the program source code and executes it directly
  - A **compiler** is an application which converts the program source code into executable machine code

Executing programs

- CPUs execute **machine code**
- Programs must be **translated** into machine code for execution
- There are three main ways of doing this:
  - An **interpreter** is an application which reads the program source code and executes it directly
  - A **compiler** is an application which converts the program source code into executable machine code
  - A **just-in-time (JIT) compiler** is halfway between the two — it compiles the program on-the-fly at runtime

Examples

Interpreted:

- Python
- Lua
- Bespoke scripting languages

Examples

Interpreted:

- Python
- Lua
- Bespoke scripting languages

Compiled:

- C
- C++
- Swift

Examples

Interpreted:

- Python
- Lua
- Bespoke scripting languages

Compiled:

- C
- C++
- Swift

JIT compiled:

- Java
- C#
- JavaScript (in modern web browsers)
- Jython

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter
  - The compiler translates the program **in advance**, on the developer's machine

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine
- Portability: compiler $<$ interpreter

Interpreter vs compiler

- Run-time efficiency: compiler > interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine
- Portability: compiler < interpreter
  - A compiled program can only run on the operating system and CPU architecture it was compiled for

Interpreter vs compiler

- Run-time efficiency: compiler $>$ interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine
- Portability: compiler $<$ interpreter
  - A compiled program can only run on the operating system and CPU architecture it was compiled for
  - An interpreted program can run on any machine, as long as a suitable interpreter is available

Interpreter vs compiler

- Run-time efficiency: compiler > interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine
- Portability: compiler < interpreter
  - A compiled program can only run on the operating system and CPU architecture it was compiled for
  - An interpreted program can run on any machine, as long as a suitable interpreter is available
- JIT compilers have similar pros/cons to interpreters

Interpreter vs compiler

- Run-time efficiency: compiler > interpreter
  - The compiler translates the program **in advance**, on the developer's machine
  - The interpreter translates the program **at runtime**, on the user's machine
- Portability: compiler < interpreter
  - A compiled program can only run on the operating system and CPU architecture it was compiled for
  - An interpreted program can run on any machine, as long as a suitable interpreter is available
- JIT compilers have similar pros/cons to interpreters
- For games, run-time efficiency is usually much more important than portability

The C++ build process

Preprocessor

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files
- Handles other preprocessor directives (`#define`, `#if` etc — more on these another time)

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files
- Handles other preprocessor directives (`#define`, `#if` etc — more on these another time)

Compiler

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files
- Handles other preprocessor directives (`#define`, `#if` etc — more on these another time)

Compiler

- Translates each source file into an **object file** containing machine code

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files
- Handles other preprocessor directives (`#define`, `#if` etc — more on these another time)

Compiler

- Translates each source file into an **object file** containing machine code

Linker

The C++ build process

Preprocessor

- Replaces `#include` directives with the contents of the appropriate header files
- Handles other preprocessor directives (`#define`, `#if` etc — more on these another time)

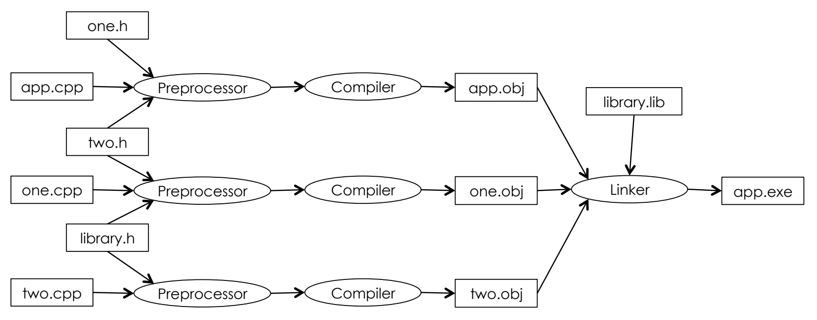
Compiler

- Translates each source file into an **object file** containing machine code

Linker

- Combines the object files together with any external libraries to produce an **executable** (on Windows, a .exe file)

The C++ build process



Modular design revisited

- Your code can call any function for which there is a **declaration** in the current file (in the file itself or `#include`d)

Modular design revisited

- Your code can call any function for which there is a **declaration** in the current file (in the file itself or `#include`d)
- The **definition** of the function may be in another file

Modular design revisited

- Your code can call any function for which there is a **declaration** in the current file (in the file itself or `#include`d)
- The **definition** of the function may be in another file
- The **linker** resolves the function call in this case

Incremental compilation

- Compilation can take a long time

Incremental compilation

- Compilation can take a long time
- Visual C++ does **incremental compilation**: only recompiles source files that have changed

Incremental compilation

- Compilation can take a long time
- Visual C++ does **incremental compilation**: only recompiles source files that have changed
- Visual C++ uses **precompiled headers**: anything `#include`d in `stdafx.h` is only compiled when it changes

Incremental compilation

- Compilation can take a long time
- Visual C++ does **incremental compilation**: only recompiles source files that have changed
- Visual C++ uses **precompiled headers**: anything `#include`d in `stdafx.h` is only compiled when it changes
- Use **Build** → **Clean** or **Build** → **Rebuild** to force everything to be recompiled

Build configuration in VC++

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger
  - **Release** produces optimised code — usually 2–10 $\times$ faster than Debug

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger
  - **Release** produces optimised code — usually 2–10 $\times$ faster than Debug
  - Generally use Debug for development, Release for optimisation and distributing the finished application

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger
  - **Release** produces optimised code — usually 2–10 $\times$ faster than Debug
  - Generally use Debug for development, Release for optimisation and distributing the finished application
- Platform:
  - **x86** runs on 32-bit and 64-bit versions of Windows

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger
  - **Release** produces optimised code — usually 2–10 × faster than Debug
  - Generally use Debug for development, Release for optimisation and distributing the finished application
- Platform:
  - **x86** runs on 32-bit and 64-bit versions of Windows
  - **x64** runs on 64-bit Windows only

Build configuration in VC++

- Configuration:
  - **Debug** allows use of the Visual C++ debugger
  - **Release** produces optimised code — usually 2–10 $\times$ faster than Debug
  - Generally use Debug for development, Release for optimisation and distributing the finished application
- Platform:
  - **x86** runs on 32-bit and 64-bit versions of Windows
  - **x64** runs on 64-bit Windows only
  - Generally use x86 for maximum compatibility, x64 for apps which need to use $>$ 2GB memory or where a significant speed benefit is measured

Anatomy of a Visual C++ solution

- A **project** (.vcxproj) is a collection of source files which make up a single application or library

Anatomy of a Visual C++ solution

- A **project** (.vcxproj) is a collection of source files which make up a single application or library
- A **solution** (.sln) is a collection of projects

Anatomy of a Visual C++ solution

- A **project** (.vcxproj) is a collection of source files which make up a single application or library
- A **solution** (.sln) is a collection of projects
- The following are intermediate files
  - .sdf files (Visual C++ navigation info)
  - ipch folder (precompiled headers)
  - Debug and Release folders (compiled object code and executables)

Anatomy of a Visual C++ solution

- A **project** (.vcxproj) is a collection of source files which make up a single application or library
- A **solution** (.sln) is a collection of projects
- The following are intermediate files
  - .sdf files (Visual C++ navigation info)
  - ipch folder (precompiled headers)
  - Debug and Release folders (compiled object code and executables)
- These files can be deleted to save space, and should be in your `.gitignore` file to prevent them being uploaded to GitHub

Arrays and pointers

Arrays in C++

- An **array** is a fixed-length sequence of elements of a particular type

Arrays in C++

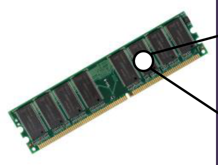
- An **array** is a fixed-length sequence of elements of a particular type
- Not to be confused with a **vector**, which is a variable length sequence

Arrays in C++

- An **array** is a fixed-length sequence of elements of a particular type
- Not to be confused with a **vector**, which is a variable length sequence

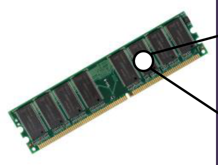
```
// Declare a 5-element array with initial values
int myArray[] = { 1, 3, 5, 7, 9 };

// Declare a 10-element array without specifying
    initial values
int myOtherArray[10];
```

Arrays in memory



Address	Data
00000000	01001000
00000001	01100101
00000002	01101100
00000003	01101100
00000004	01101111
00000005	00100001
...	...

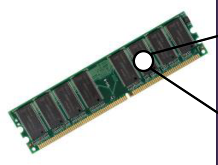
Arrays in memory



Address	Data
00000000	01001000
00000001	01100101
00000002	01101100
00000003	01101100
00000004	01101111
00000005	00100001
...	...

- An array is a contiguous block of memory

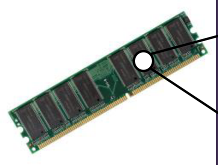
Arrays in memory



Address	Data
00000000	01001000
00000001	01100101
00000002	01101100
00000003	01101100
00000004	01101111
00000005	00100001
...	...

- An array is a contiguous block of memory
- E.g. an `int` is 4 bytes (32 bits), so an array of 10 `int`s is $10 \times 4 = 40$ bytes

Arrays in memory



Address	Data
00000000	01001000
00000001	01100101
00000002	01101100
00000003	01101100
00000004	01101111
00000005	00100001
...	...

- An array is a contiguous block of memory
- E.g. an `int` is 4 bytes (32 bits), so an array of 10 `int`s is $10 \times 4 = 40$ bytes
- The size of the array is **fixed**: a 10 element array holds exactly 10 elements, forever

Index out of range

This Python code will give a "list index out of range" exception

```
myList = [1, 3, 5, 7, 9]
print myList[5]
```

Index out of range

This Python code will give a “list index out of range” exception

```
myList = [1, 3, 5, 7, 9]
print myList[5]
```

This C++ code will give a “vector subscript out of range” exception

```
std::vector<int> myVector = {1, 3, 5, 7, 9};
std::cout << myVector[5] << std::endl;
```

Index out of range

This Python code will give a "list index out of range" exception

```
myList = [1, 3, 5, 7, 9]
print myList[5]
```

This C++ code will give a "vector subscript out of range" exception

```
std::vector<int> myVector = {1, 3, 5, 7, 9};
std::cout << myVector[5] << std::endl;
```

This C++ code will print some arbitrary number

```
int myArray[] = { 1, 3, 5, 7, 9 };
std::cout << myArray[5] << std::endl;
```

Index out of range

- C++ does not check array indices

Index out of range

- C++ does not check array indices
- It is easy to accidentally read or write past the end of the array, and doing so will cause hard-to-fix bugs

Index out of range

- C++ does not check array indices
- It is easy to accidentally read or write past the end of the array, and doing so will cause hard-to-fix bugs
- `myArray` is a 5-element array of `int`s, i.e. a block of $5 \times 4 = 20$ bytes

Index out of range

- C++ does not check array indices
- It is easy to accidentally read or write past the end of the array, and doing so will cause hard-to-fix bugs
- `myArray` is a 5-element array of `int`s, i.e. a block of $5 \times 4 = 20$ bytes
- If `myArray` starts at memory address 1000, then `myArray[i]` is at address $1000 + 4 \times i$

Index out of range

- C++ does not check array indices
- It is easy to accidentally read or write past the end of the array, and doing so will cause hard-to-fix bugs
- `myArray` is a 5-element array of `int`s, i.e. a block of $5 \times 4 = 20$ bytes
- If `myArray` starts at memory address 1000, then `myArray[i]` is at address $1000 + 4 \times i$
- `myArray[5]` is whatever happens to be at memory address $1000 + 4 \times 5 = 1020$ — could be unallocated memory, could be another variable, could be part of another array, could even be part of the machine code being executed

Array size must be a compile-time constant

```
double a[10];              // OK

const int size = 10;
double b[size];            // OK
double c[2 * size + 7];    // OK

int varSize = 10;
double d[varSize];         // Error
```

Dynamic allocation

- If the size of the array is not known in advance, it can be allocated at runtime using the `new` keyword

```
int n = readNumberFromConsole();
int* myArray = new int[n];
```

Dynamic allocation

- If the size of the array is not known in advance, it can be allocated at runtime using the `new` keyword

```
int n = readNumberFromConsole();
int* myArray = new int[n];
```

- Technically `myArray` is no longer an array, it's a **pointer**

Dynamic allocation

- If the size of the array is not known in advance, it can be allocated at runtime using the `new` keyword

```
int n = readNumberFromConsole();
int* myArray = new int[n];
```

- Technically `myArray` is no longer an array, it's a **pointer**
- `int*` is the type "pointer to an `int`"

Dynamic allocation

- If the size of the array is not known in advance, it can be allocated at runtime using the `new` keyword

```
int n = readNumberFromConsole();
int* myArray = new int[n];
```

- Technically `myArray` is no longer an array, it's a **pointer**
- `int*` is the type "pointer to an `int`"
- Pointers can (mostly) be used as if they were arrays

Stack and heap

- Variables and static arrays are stored on the **stack**

Stack and heap

- Variables and static arrays are stored on the **stack**
  - Stack items are automatically freed when they go out of scope

Stack and heap

- Variables and static arrays are stored on the **stack**
  - Stack items are automatically freed when they go out of scope
- Anything created with `new` is stored on the **heap**

Stack and heap

- Variables and static arrays are stored on the **stack**
  - Stack items are automatically freed when they go out of scope
- Anything created with `new` is stored on the **heap**
  - Heap items must be freed with `delete` when they are finished with

Stack and heap

- Variables and static arrays are stored on the **stack**
  - Stack items are automatically freed when they go out of scope
- Anything created with `new` is stored on the **heap**
  - Heap items must be freed with `delete` when they are finished with
  - Forgetting to free them is a **memory leak**

Strings revisited

- `std::string` is the high-level string class

Strings revisited

- `std::string` is the high-level string class
- The low-level way of storing strings is as an array of `char`s

Strings revisited

- `std::string` is the high-level string class
- The low-level way of storing strings is as an array of `char`s

```
char greeting[] = "Hello, world!";
```

- Strings are **null terminated** — they end with ASCII character 0

2-dimensional arrays

Array of arrays approach:

```
const int width = 8, height = 8;
int grid[width][height];
grid[x][y] = 7;
```

2-dimensional arrays

Array of arrays approach:

```
const int width = 8, height = 8;
int grid[width][height];
grid[x][y] = 7;
```

Flat array approach:

```
const int width = 8, height = 8;
int grid[width * height];
grid[x + y * width] = 7;
```

Functions

Function definitions

- We have already seen an example of a function definition

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

- The function `main` takes no parameters, and returns a value of type `int`

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

- This function takes three parameters:

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

- This function takes three parameters:
  - `x` of type `std::string`

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

- This function takes three parameters:
  - `x` of type `std::string`
  - `y` of type `int`

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

- This function takes three parameters:
  - `x` of type `std::string`
  - `y` of type `int`
  - `z` of type `bool`

Function signatures

- The **signature** of a function defines its return type, name, and parameters

```
double foo(std::string x, int y, bool z)
```

- This function takes three parameters:
  - `x` of type `std::string`
  - `y` of type `int`
  - `z` of type `bool`
- It returns a value of type `double`

Functions without return values

- It is possible to define a function which does not return a value, using the `void` keyword in place of its return type

Functions without return values

- It is possible to define a function which does not return a value, using the `void` keyword in place of its return type

```
void printNumber(int n)
{
    std::cout << n << std::endl;
}
```

Pass by value

- Function parameters are passed **by value**: the function receives **copies** of the original variables

Pass by value

- Function parameters are passed **by value**: the function receives **copies** of the original variables

```
void changeName(std::string name)
{
    name = "Ed";
}

int main()
{
    std::string name = "Mike";
    std::cout << name << std::endl; // Mike
    changeName();
    std::cout << name << std::endl; // Mike
}
```

Pass by reference

- Parameters can be passed **by reference** using `&`, allowing the function to modify them

Pass by reference

- Parameters can be passed **by reference** using `&`, allowing the function to modify them

```
void changeName(std::string& name)
{
    name = "Ed";
}

int main()
{
    std::string name = "Mike";
    std::cout << name << std::endl; // Mike
    changeName();
    std::cout << name << std::endl; // Ed
}
```

One area where C++ is "simpler" than Python!

- Recall from COMP110 week 6: in Python, basic data types (numbers, booleans, strings etc) are passed by value, and object types (lists, dictionaries, class instances) are passed by reference

One area where C++ is "simpler" than Python!

- Recall from COMP110 week 6: in Python, basic data types (numbers, booleans, strings etc) are passed by value, and object types (lists, dictionaries, class instances) are passed by reference
- In C++, everything is passed by value unless it is explicitly marked as a reference with `&`

Constant references

```
void greet(std::string name)
{
    std::cout << "Hi " << name << std::endl;
}
```

Constant references

```
void greet(std::string name)
{
    std::cout << "Hi " << name << std::endl;
}
```

- The string will be copied in order to be passed in

Constant references

```
void greet(std::string name)
{
    std::cout << "Hi " << name << std::endl;
}
```

- The string will be copied in order to be passed in
- More efficient to pass a reference, and mark it `const` to prevent accidental modification

```
void greet(const std::string& name)
{
    std::cout << "Hi " << name << std::endl;
}
```

Constant references

```
void greet(std::string name)
{
    std::cout << "Hi " << name << std::endl;
}
```

- The string will be copied in order to be passed in
- More efficient to pass a reference, and mark it `const` to prevent accidental modification

```
void greet(const std::string& name)
{
    std::cout << "Hi " << name << std::endl;
}
```

- (this is only worthwhile for large data structures like strings and vectors, not for basic data types)

FALMOUTH UNIVERSITY

Live coding: Noughts and Crosses