Welcome to

Data Structures and Algorithms

31251

Topics for Today

- Meet your tutor
- Programming at UTS
- What to expect from this subject
- Why C++
- Pointers and references
- Pass by methods
- This week’s lab

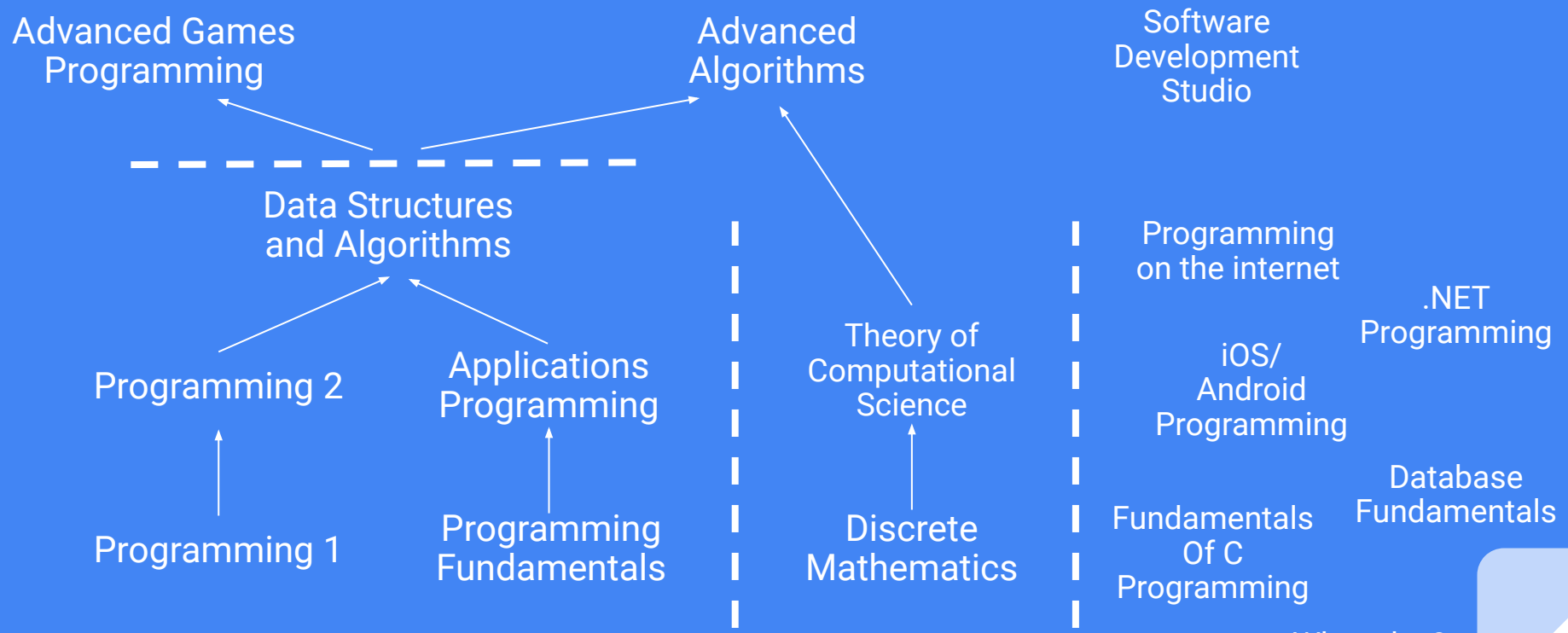
John Collins

- Computer Science (Honours) Graduate
- Gameplay Engineer
  - New, soon to be announced studio
  - Working in Unreal 5 with C++/BP
- Honours Thesis programmed on shadow rendering, programmed in C++/OpenGL

Contact:
john.collins@uts.edu.au
(Do not use my student email)
On the Ed discussion board
(this is a part of the tutors' jobs)

Tutorial Style:
Start tutorials with a brief slideshow, covering the ideas necessary for the lab, and useful C++ concepts.
Describe and discuss each lab problem.
Assign a small part of the problem and time to solve it, before giving its solution, and moving to next sub-problem.

Programming at UTS



What to expect from DSA

- We are going to go through A LOT
- The C++ language
- Pointers and references
- Big O notation, time and space complexity theory
- Basic list/array like data structures
- Stacks, Queues
- Sorting algorithms (many)
- Algorithm design paradigms
- Binary tree data structure
- Tree traversal algorithms
- Hashing, sets and maps
- Graph data structures
- Graph traversal algorithms (many)

This class gets you ready for typical programming job interview questions.

What to expect from DSA

Assessments:

- Programming Assignment 1 - 20%
- Programming Assignment 2 - 30%
- Weekly Exercises (10 in total) - 20%
- Exam - 30%

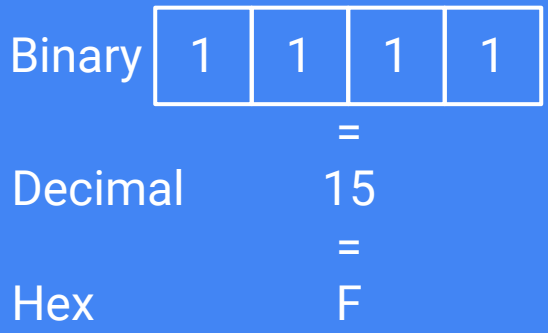
C++

- It remains in use because it is so fast.
- Powerful mix of low and high level programming
  - Low level memory management
  - High level object oriented programming
  - Very specific and precise language
- A historically and conceptually important language.
- Has a much higher skill floor than other languages and some quirks to its design. This is why Java/C# were invented.
- May be hard to get used to if you've primarily used Python, JavaScript.

Hexadecimal: 0x686bf5402b9e

The **0x** prefix indicates that the value is a hexadecimal number. 0x is not a part of the number, it just means its in hex.
An alternative way of writing a number.
In decimal notation each digit represents values [0,9], in hexadecimal, they represent [0,15], A=10, B=11, C=12, D=13, E=14, F=15.
A useful representation, because it maps naturally to binary, 15 in binary is 1111, so a hex digit encapsulates 4 bits.
Two hex digits represents a byte.

Other forms of this number:
Decimal: 114812885412766
Binary: 110100001101011111101010100000001010111001111 0
In the style of an IP: 104.107.64.43.158
Also commonly used to refer to 16 bit colours.



Computer architecture

- RAM - “Memory”
- HDD/SSD - “Storage”
- CPU/GPU - “Processor”

Memory is partitioned into 1-byte (8-bit) segments,
which by convention are identified by a hexadecimal number.

This is what a pointer holds, the name (address) of a particular place in memory.

The smallest amount we can use is a `char` or a `bool` which are both 1 byte.
You can use `sizeof(char)` and it will return 1, try other type specifiers and see how many bytes they are.

Pointers and References

As initialisers

int* x “Pointer”
Initialises a variable containing a memory address.
Contains:
0x00000000

int& x “Reference”
Initialises an *alias* for an existing variable. They both refer to the same memory address to read/write their data.
Contains: data

As operators

&x “Address of”
Get the address of this variable.
Returns:
0x00000000

*x “Dereference”
Gets the data stored in that memory address.
Used on pointers.
Returns: data

As function parameters

Function parameters use the initialiser rules from the previous table.

```
int myFunc(int& x){}
```

```
int myFunc(int x){}
```

```
int myFunc(const int& x){}
```

```
int myFunc(int* x){}
```

As function parameters

Function parameters use the initialiser rules from the previous table.

int myFunc(int& x){}
Gives you the original, editable variable in the function body.

int myFunc(const int& x){}
Same as above, but will not compile if you make any changes to x.

int myFunc(int x){}
Gives you a copy of x to use within the function body. Changes to x within the function do not affect the original variable.

int myFunc(int* x){}
Gives you a pointer, this is useful for objects that we store on the heap (talk about this later).

This Week's Lab

C in and C out:
C++ has a somewhat unusual way of dealing with printing and live user inputs.
We use the << (stream insertion) and >> (stream extraction) operators.
C++ considers the output log and user input as *streams*, which are like sources of data of unknown size. So rather than loading them all into memory at once, because we don't know what size that would be, we do it bit by bit.

To print we use

```
#include <iostream>
std::cout << "hello world";
```

To push this to the stream.
We also like to add a new line "\n"

To print we use

```
#include <iostream>
std::string userInput {};
std::cin >> userInput;
```

To read this from the stream

Writing Functions

We often want to reuse functionality in our programs, for this we encapsulate them into functions.

For this exercise we want you to write a simple function to add two integers together and return the result.

In C++ we define a function by writing its return type, then name, then parameters, and a function scope { } and unless its void a return statement.

I'll give you a couple of minutes to try this.

Writing Functions

We often want to reuse functionality in our programs, for this we encapsulate them into functions.

For this exercise we want you to write a simple function to add two integers together and return the result.

In C++ we define a function by writing its return type, then name, then parameters, and a function scope { } and unless its void a return statement.

```
int add(int a, int b) {
    return a + b;
}
```

```
return type function name(parameter 1 type parameter1 name, ...) {
    return statement return value ;
}
```

Factorial

Factorial (!) is a mathematical operator that returns the product of all integers between the supplied number and one.
Remember: When we add things this is called a *sum*,
When we multiply things this is called a *product*.

So for example:
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
This operator creates large values very very quickly.
Also note 0! = 1

Write your solution in factorial.cpp (NOT factorial.h or main.cpp), make a function that returns the factorial of the given number.

Different Parameter Modes

In this exercise we give you examples of different functions, where you would want to use the different kinds of parameters passing modes, like we spoke about earlier.

When completing this task, the point is to change the definitions of the functions to use the most suitable parameters. They are all pass-by-value in the template, change them to the correct mode.

As function parameters

Function parameters use the initialiser rules from the previous table.

int myFunc(int& x){}
Gives you the original, editable variable in the function body.

int myFunc(const int& x){}
Same as above, but will not compile if you make any changes.

int myFunc(int x){}
Gives you a copy of x to use within the function body. Changes to x within the function do not affect the original variable.

int myFunc(int* x){}
Gives you a pointer, this is useful for objects that we store on the heap (talk about this later).