Demo lecture slides

- Although I will not usually give slides for demo lectures, the first two demo lectures involve practice with things which you should really know from G51PRG
- Since I covered much of this in the G52CFJ (C and C++) module in previous years, I have included the slides from that module here, to remind people who may not attend the demo lectures, or may forget what was said

Variables and literals

```
#include <stdio.h>

int main( int argc, char* argv[] )
{
      int j = 0;

      … do something spectacular …

      return 0; // Success
}
```

A ‘literal’
A literal/actual value

A variable declaration/definition
Defines a variable of type **`int`**, with name j

Definition:
Defines what it does / creates one
Declaration:
Says it exists, but doesn’t create it
An important difference later for functions, variables and classes

Comments

- Comments:
  - `/* For multi-line comments */`
    - Available in both C and C++
  - `// For single line comments`
    - In C++ but not ***officially*** in C89

- There is no official “Javadoc” for C/C++
  - i.e. /** … */ for code documentation
  - There are unofficial programs, e.g. **doxygen**

Braces: { }

- Braces {} are used in the same way as Java
  - Create a compound statement from multiple statements
  - e.g. for an '**`if`**' or '**`for`**' statement
  - Extra {} can be added to make execution blocks
    - Local variables exist for the lifetime of the **block** they are in (not the function)

```
int main( int argc, char* argv[] )
{
      int j = 0;
      { int k = 3; /* k exists now */ }
      /* k no longer exists now */
      return 0; // Success
}
```

Basic data types

Basic Data Types

- The types available are what you expect:
  - Integer types: **`char`**, **`short`**, **`int`**, **`long`**
  - Floating point: **`float`**, **`double`**, **`long double`**
  - **No *string* type in C!** (treat **`char*`** as string)
- Signed/unsigned variants of integer types
  - **Unlike in Java where they are all signed**
  - Examples:

```
signed char sc;        unsigned short us;
signed long sl;        unsigned int ui;
```

  - Default is **`signed`**
    - If neither '**`signed`**' nor '**`unsigned`**' stated

Sizes of types…

- The size (in bits/bytes) can vary in C/C++
  - For different compilers/operating systems
  - In Java, sizes are standardised, across O/Ss
- Some guarantees are given:
  - A minimum size (bits): char 8, short 16, long 32
  - Relative sizes: `char` ≤ `short` ≤ `int` ≤ `long`
- C & C++ have an operator called sizeof() to ask how man `char`s big a data type is (sometimes this matters)
- An `int` changes size more than other types!
  - Used for speed (not portability)
  - But VERY popular! (fast)
  - Uses the most efficient size for the platform
  - 16 bit operating systems usually use 16 bit `int`
  - 32 bit operating systems usually use 32 bit `int`
  - 64 bit operating systems usually use 64 bit `int`

Basic Data Types - Summary

Type	Minimum size (bits)	Minimum range of values (Depends upon the size on your platform)
`char`	8	-128 to 127 (Java chars are 16 bit!)
`short`	16	-32768 to 32767
`long`	32	-2147483648 to 2147483647
`float`	Often 32	Single precision (implementation defined) e.g. 23 bit mantissa, 8 bit exponent
`double`	Often 64	Double precision (implementation defined) e.g. 52 bit mantissa, 11 bit exponent
`long double`	≥ double	Extended precision, implementation defined
`int`	≥ short	varies

Declaring/defining a variable

- Important: Variables will **NOT** be initialised
  - No default values are given in C++!
  - Nothing MUST be initialised
    - Values are unknown! (whatever was in memory)
    - In Java member variables get default values and local variables MUST be initialised before use
  - **ALWAYS initialise your variables**
    - At worst, errors are then repeatable
- Examples:

```
float f1 = 1.0f; // A single float
int i1=4, i2=52; // Two ints
```

Integer literals

- Integer literals may be:
  - Decimal (base 10) : default, no prefix needed
  - Hexadecimal (base 16) : prefix '**`0x`**'
  - Octal (base 8) : prefix '**`0`**' (Not available in Java)
- Examples:

```
int x = 19, y = 024, z = 0x15;
char c1 = 45, c2 = 67, c3 = 0;
unsigned short s = 0xff32;
```

- The compiler chooses a size based upon the size of an **`int`** and the value of the literal (e.g. **`char`**, **`short`**, **`int`**, **`long`**)
- You can explicitly make a literal value long (add suffix L)

```
long l1 = 1000000000L;
long l2 = 1234567890L;
```

Character literals

- Character literals mean ‘the value of this character using the standard character set for this computer’ (ASCII here)

```
char c1 = ‘h’,  c2 = ‘e’,  c3 = ‘l’,  c4 = ‘l’,  c5 = ‘o’;
```

- A ‘**`char`**’ is a **number** (from -128 to +127)
  - In output functions you specify whether to show ‘the number’(%d) or ‘the ASCII character of that value’ (%c)
- Some character literals have special meanings
  - ‘**`\t`**’ is the character which, when printed, will display a tab character
  - ‘**`\n`**’ is a character which will display as a newline
    - Can be CR, CR+LF, depending on platform
    - In Java \n and \r have fixed values

String literals

- You can have string literals:

```
char* s1 =
"This is a string literal\n";
```

  - Actually: arrays of characters, with a 0 at the end
- Enclose the literal in **double** quotes
  - Format is same as Java: `String s1 = “Hello”;`
- **`printf`** takes a **`char*`** as first parameter:

```
printf( "Hello World!\n" );
printf( s1 );
```

- Remember: character literals have single quotes
  - `‘4’, ‘h’, ‘H’, ‘%’, ‘£’, ‘@’, ‘,’`
- string literals have double quotes

Floating point literals

- Same as Java
- Double precision floating point (**`double`**):
  - **`1.0, 2.4, 1.23e-15, 9.5e4`**
  - **`double d = 1.34283;`**
- Single precision floating point (**`float`**):
  - **`1.0f, 2.4f, 2.9e-3f`**
  - **`float f = 5.634f`**;
  - (note the '**`f`**' to say '**`float`**' rather than the default type of '**`double`**')

Converting between types

- **Data can be converted between types**
- **Sometimes done implicitly**
  - If compiler knows how to safely change the type
  - e.g. `char` to a `short`, `short` to a `long`, `float` to a `double`, `int` to a `double` (same rules as Java)
- **Sometimes it has to be done explicitly**
  - If conversion may lose data
  - e.g. `long` to a `short`, `short` to a `char`, `double` to a `float`, `float` to an `int` (same rules as Java)
  - Or compiler needs to confirm that it isn't an error:

    "Are you sure?"

Type casts

- **Can explicitly change the type via a cast**
  - C version is exactly the same as in Java
  - Put the new type inside brackets **()**

```
long l = 100L;
short s = (short)l;
```

  - Includes signed **<->** unsigned conversion

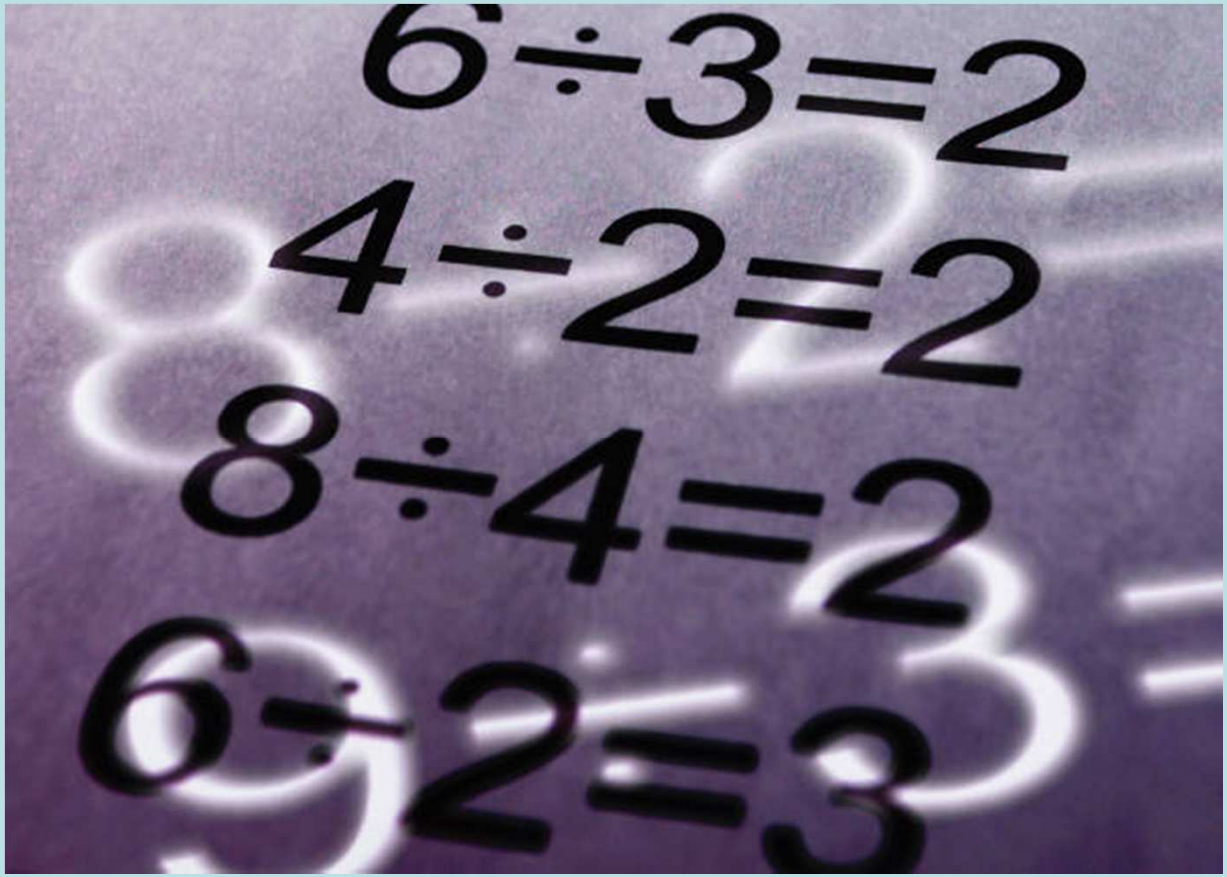
```
unsigned int ui = (unsigned int)i;
```

- **C++ also adds new types of casts**
  - Safer and better
  - We will return to these later

The `void` type

- The **`void`** type is used to mean:
  - No return value,
    - e.g. **`void foo( int a );`**
  - No parameters, optional, (Not Java)
    - e.g. **`int bar(void);`**
  - Will also see later a **`void*`**
- You cannot create a variable of type '**`void`**'
- Some (older) compilers will not accept **`void`**
  - But they should do if they are C89/C90/C99

Operators (same as Java)



Operators

Operators will be familiar to you from Java:

- Arithmetic: **`a*b, a-b, a+b, a/b`**
- Integer division: **`a/b, a%b`**
- Logical AND/OR: **`a && b, a || b`**
- Comparison: **`a<b, a<=b, a>b, a>=b`**
- **`a==b, a!=b`**
- Increment **`a++, ++a`**
- Decrement **`a--, --a`**
- Shorthand: **`a+=b, a-=b, a*=b, a/=b`**
  - e.g. **`a+=b`** is equivalent to **`a = a + b`**

Sample Operator Precedence List

- Operators are evaluated in a specific order
  - Highest operator precedence applies first
- Examples (highest to lowest, not complete)

Increasing precedence ↑

(), [], ++, --	Grouping, array access, post increment/decrement
++, --, *, &	Pre-increment, dereference, address of (right to left)
*, /, %	Multiplication, division, modulus
+ -	Addition, subtraction
<, <=, >, >=	Comparison
==, !=	Comparison: equal to, not equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Ternary conditional
=, +=, -= etc	Assignment and '… and assign' (right to left)

Operator precedence matters

&& has higher precedence than ||

```
if (  a && b  ||  c && d  )
```

means

```
if ( (a && b) || (c && d) )
```

```
if ( a ||  b && c  || d )
```

means

```
if ( a || (b && c) || d )
```

Operators and precedence

- **Operator precedence matters!**
- **Many style guides state that operator precedence should not be relied upon**
  - Makes code less readable
  - Prone to reliability of programmer's memory
- **I will NOT mark you down for adding unnecessary brackets (within reason)**
  - I do it where I think it aids clarity
  - 'Company' coding standards often require them
- **<u>But</u> you need to know the precedence rules**
  - To understand code written by others
  - An exam question may rely on them

The `printf()` function

The printf function

- Reminder: **`printf`** is declared in '**`stdio.h`**'
  - **`#include <stdio.h>`** so compiler knows what it is
- **`printf`** will output **formatted** text
- It uses tags (starting with '%') which are replaced by the supplied parameter values, **in order**
- Examples:

```
int i = 50;
char* mystring = "Displayable string";
printf( "Number: %d\n", i );
printf( "String: %s\n", mystring );
printf( "%d %s\n", i, mystring );
```

More things (almost)
exactly the same

Conditionals

- Example ‘if’ statement: (same as Java!)

```
if ( x == 4 )
  printf( “X is 4\n” );
else
  printf( “X is not 4\n” );
```

- Ternary conditional operator: (same as Java!)

```
char* str =
  (x == 4) ? “X is 4\n” : “X is not 4\n”;
printf( str );
```

- The switch statement (same as Java!)

```
switch( x )
{
case 4: printf(“X is 4\n”); break;
default: printf(“X is not 4\n”); break;
}
```

Loops: same as Java

- Example for loop:

```
int x = 1;
for ( x = 2; x < 10 ; x++ )
{
  printf("X is %d\n", x);
}
```

- Example while statement:

```
int x = 12;
while ( x > 4 )
{
  printf( "X is %d\n", x );
  x--;
}
```

- Example do {…} while statement:

```
int x = 1;
do
{
  printf( "X is %d\n", x );
  x++;
} while ( x < 8 );
```

break and continue

- break
  - Already seen use in a switch
  - Also used in loops : exit the loop
- continue
  - Used in loops
  - End this iteration of the loop
  - i.e. Jump to the for/while control statement

```
int i = 0;
for ( ; i < 30 ; i++ )
{
     if ( i==5 ) continue;
     if ( i==10) break;
     printf( "%d ", i );
}
```

What is the output:

?

break and continue

- break
  - Already seen use in a switch
  - Also used in loops : exit the loop
- continue
  - Used in loops
  - End this iteration of the loop
  - i.e. Jump to the for/while control statement

```
int i = 0;
for ( ; i < 30 ; i++ )
{
     if ( i==5 ) continue;
     if ( i==10) break;
     printf( "%d ", i );
}
```

Output:

0 1 2 3 4 6 7 8 9

Aside: Function Overloading

A C vs C++ difference

Here C++ is like Java

Function overloading

- In C++ a function is identified by the types of its parameters as well as its name
  - As it is in Java, but not in C
- This example is valid in C++ (not in C)

```
int multiply( int a, int b ) {return 0;}
long multiply( long a, long b) {return 0;}
```

  - i.e. can have multiple functions with the same name
- The compiler will check the ***parameter types*** and try to find a matching function (like Java)
- Sometimes the types of the parameters may need to be changed to find a function
  - Will usually be performed automatically (implicitly) as long as a conversion exists (see later)