G52CPP
C++ Programming
Lecture 2

Dr Jason Atkin

Content

- Hello World
- printf()
- Data types and sizes
- #include, #define, #ifdef, #endif
- Writing a simple program – practical

- Many of the following slides are for reference/revision and will not all be displayed in the lecture

The “Hello World” Program

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello world!\n");
  return 0;
}
```

C version

```cpp
#include <cstdio>

int main(int argc, char* argv[])
{
  printf("Hello world!\n");
  return 0;
}
```

C++ version

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 );
```

Data types

Sizes of types…

- **The size of types (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`**
- 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`
- `sizeof()` **operator** exists to tell us the size (later lecture)

Basic Data Types - Summary

Type	Minimum size (bits)	Minimum range of values (Depends upon the size on your platform)
`char`	8	-128 to 127 (WARNING: Java char is 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

`bool` type (C++ only, not C)

- **`bool`** : **`true`**/**`false`**
- Similar to java's boolean type
- Boolean expressions have results of type '**`bool`**' in C++
  - But type **`int`** in C – a difference
- **IMPORTANT: `bool`** and **`int`** **can** be converted **implicitly / automatically** to each other
  - i.e. C++ is backward compatible
  - **`true`** defined to be **`1`** when converted to **`int`**
  - **`false`** defined to be **`0`** when converted to **`int`**
  - **`0`** is defined to be **`false`**, non-zero as **`true`**

`ints` and `bools`

- **In both C and C++ any integer types can be used in conditions (i.e. char, short, long, int)**
  - In C++ the value is ***silently*** converted to a C++ `bool` type
- When using integer types:
  - true is equivalent to non-zero (or 1), false is equivalent to zero
- Example:

```
int x = 6;
while ( x )
{
   printf( "X is %d\n", x );
   x -= 2;
}
```

- In Java this would be an error : “x not boolean”
- In C/C++ this is valid (it means ‘**`while( x != 0 )`**’)

`wchar_t` type (C++ only, not C)

- **`wchar_t`** : wide character
  - Like a Java '**`char`**'
- ASCII limited to values 0 to 127 (7 bits)
  - Not enough characters for some languages
- **`wchar_t`** is designed to be big enough to hold a character of the : "largest character set supported by the implementation's locale"

  (Bjarne Stroustrup, The C++ Programming Language)

signed/unsigned values

- 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

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

auto

- The auto type is really useful
- New to C++11
  - older compilers will not support it
- Used for a variable which is initialised
- Compiler will work out the type at compile time from the type of the initialisation value

- E.g.

```
auto str = “Hello World”;
```

Platform specific types

- Some platforms (notably Microsoft Windows) have platform specific types. E.g.:
  - WORD = 16 bit value
  - DWORD = 32 bit value
  - See “Windows Data Types” : https://msdn.microsoft.com/en-us/library/aa383751%28VS.85%29.aspx

- Using these has advantages (portability across versions of that platform) and disadvantages (may need to do some #defines to port to other platforms)

#define

#define

- An **semi-intelligent** '*find and replace*' facility
- Often considered ***bad*** in C++ code (useful in C)
  - `const` is used more often, especially for members
  - Template functions are better than macros
- Example: define a 'constant':
  - `#define MAX_ENTRIES 100`
  - Replace occurrences of "`MAX_ENTRIES`" by the text "`100`" (without quotes), e.g. in:

```
if ( entry_num < MAX_ENTRIES ) { … }
```

- **Remember:** Done by the pre-processor!
  - E.g. **NOT** actually a **definition** of a **constant**
- 'Constant' `#define`s usually written in CAPITALS

Conditional compilation

- You can remove parts of the source code if desired
  - Done by the pre-processor (not even compiled)
- E.g. Only include code if some name has been defined earlier (in the code or included header file)

```
#ifdef <NAME_OF_DEFINE>
    <Include this code if there was a matching #define>
#else
    <Include this code if there was NOT a matching #define>
#endif
```

- To include only ‘if **not** defined’ use **`#ifndef`**
- There is also a **`#if <condition>`**

Conditional compilation

- Platform-dependent code can be included
- e.g. Include only if on a specific machine:

```
#ifdef __WINDOWS__
… windows code here …
#elif __SYS5UNIX__
… System 5 code here …
#endif
```

- Often used for cross-platform code
- The correct **`#define`** has to be made *somewhere* to specify the current platform
- Know that this can be done, recognise it

Avoiding multiple inclusion

- Code to include the contents of a file only once:

```
#ifndef UNIQUE_DEFINE_NAME_FOR_FILE
#define UNIQUE_DEFINE_NAME_FOR_FILE
… include the rest of the file here …
#endif
```

- To work, the name in the **`#define`** has to be unique throughout the program
  - E.g. you probably should include the path of the header file, not just the filename
  - Example: mycode/game/graphics/screen.h could be called **`MYCODE_GAME_GRAPHICS_SCREEN_H`**
  - By convention, **`#defines`** are in upper case

Example coursework file

```
#ifndef DISPLAYABLEOBJECT_H
#define DISPLAYABLEOBJECT_H

#include "BaseEngine.h"

class DisplayableObject
{
public:
        // Constructor
        DisplayableObject(BaseEngine*
        // Destructor
        virtual ~DisplayableObject(voi
private:
        // True if item is visible
        bool m_bVisible;
};

#endif
```

If not already marked as included

Mark it as included now, by setting the #define

Includes the header files it needs

Example file from the coursework to show that
#ifndef, #define, #endif
are used in real code, and how they are used.

Do not try to compile this sample – it will not compile without the rest of the coursework files.

End of the #ifdef around the contents

Three rules for header files

1. Ensure that the header file `#include`s everything that it needs itself
   - i.e. `#include` any headers it depends upon
2. Ensure that it doesn't matter if the header file is included multiple times
   - See previous slides
3. Ensure that header files can be included in any order
   - A consequence of the first two rules

Demo

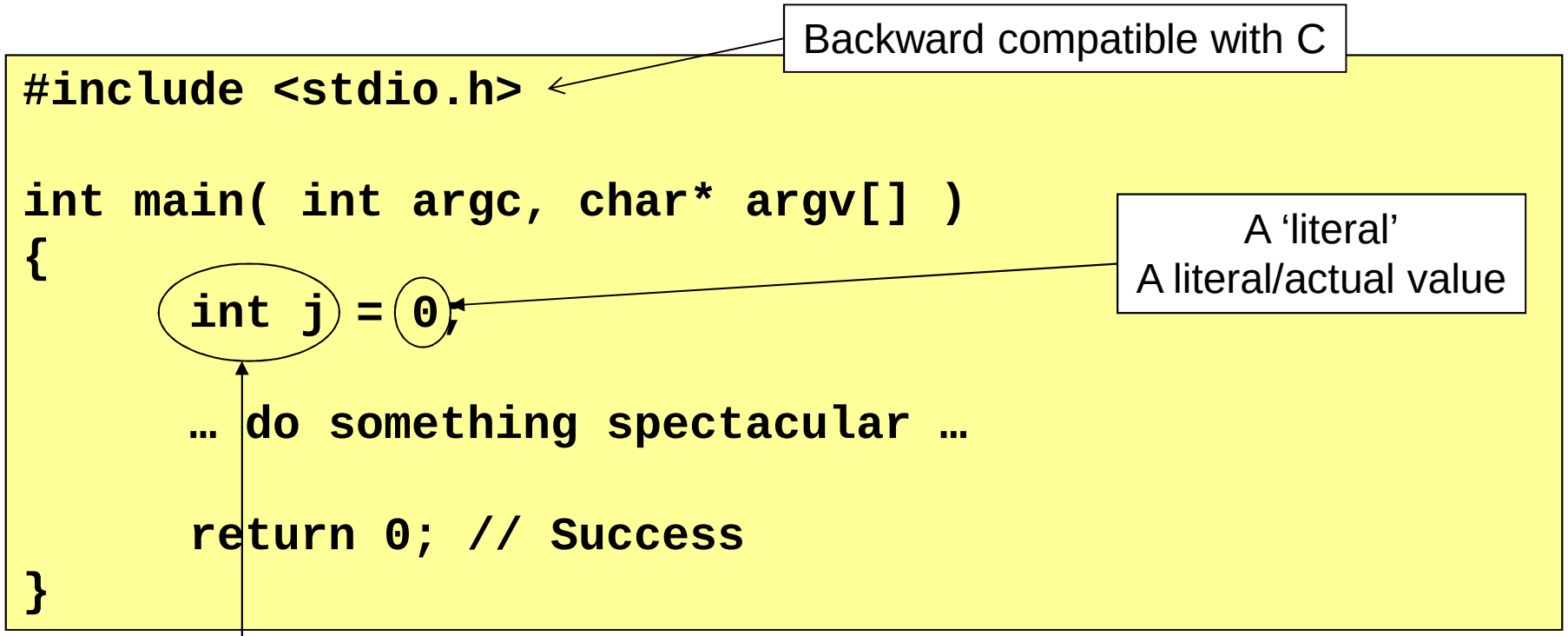
- Linux C++ : g++
- Compiling and linking – incl multiple files
- Using multiple files
- Data types and sizes
- Functions and reverse order declaration
- Loops and conditionals
- printf
- Characters and strings

Next Lecture + G53OSC lecture

- Next lecture: Pointers and arrays
- **IMPORTANT: If you do not know (well) the basics of C pointers, make sure that you attend the Thursday G52OSC lecture about pointers before the Friday G52CPP lecture**
- **NO LABS THIS WEEK**

Reference only slides

Variables and literals



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

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`**')

Simple C-style casts

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:

    Warnings mean "Are you sure?"

Type casts

- **Can explicitly change the type via a cast**
  - C version is exactly the same as Java, and works in C++
  - Put the new type inside brackets **()**, **e.g.:**

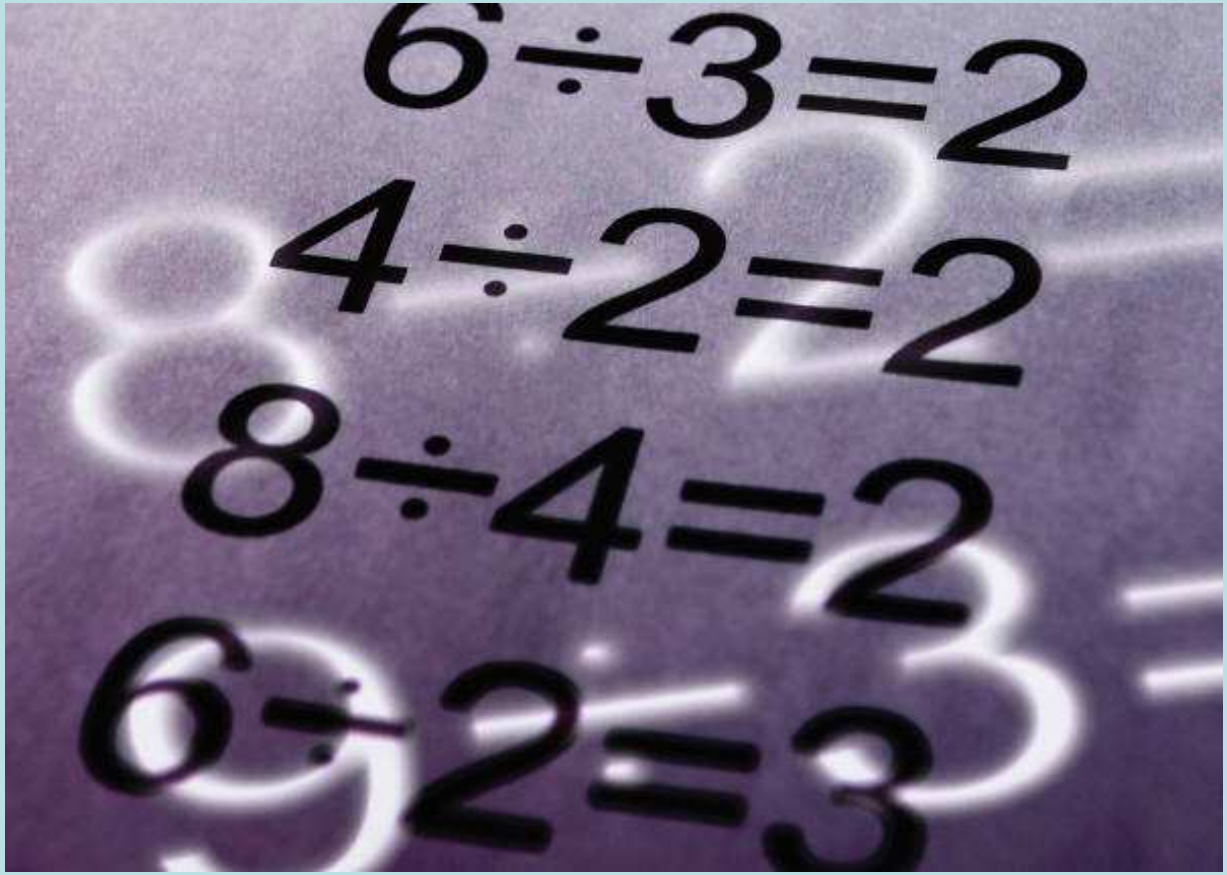
```
long l = 100L;
short s = (short)l;
```

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

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

- **C++ also adds new types of casts**
  - `… = static_cast<NEWTYPE>(VARIABLE);`
  - `… = dynamic_cast<NEWTYPE>(VARIABLE);`
  - `… = const_cast<NEWTYPE>(VARIABLE);`
  - `… = reinterpret_cast<NEWTYPE>(VARIABLE);`
  - **E.g.** `int i = static_cast<int>(longValue);`
  - Safer and better, see later lecture

Operators (same as Java)



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

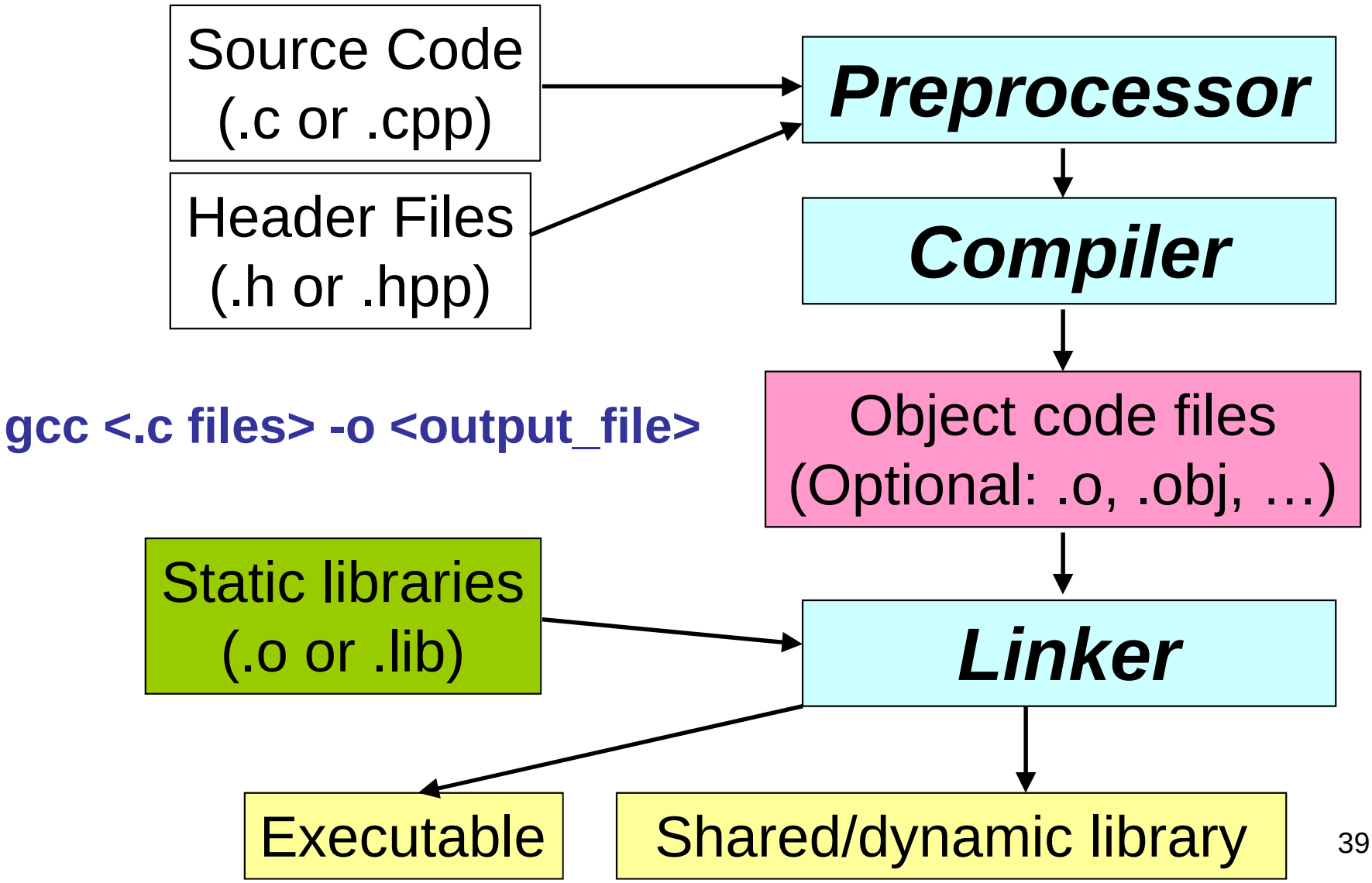
- **Operator precedence matters!**
- **Many style guides state that operator precedence should not be relied upon**
  - Makes code less readable
  - Prone to reliability of programmer's memory
- **I will NOT mark you down for adding unnecessary brackets (within reason)**
  - I do it myself where I think 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

Compiling and linking

Types of files

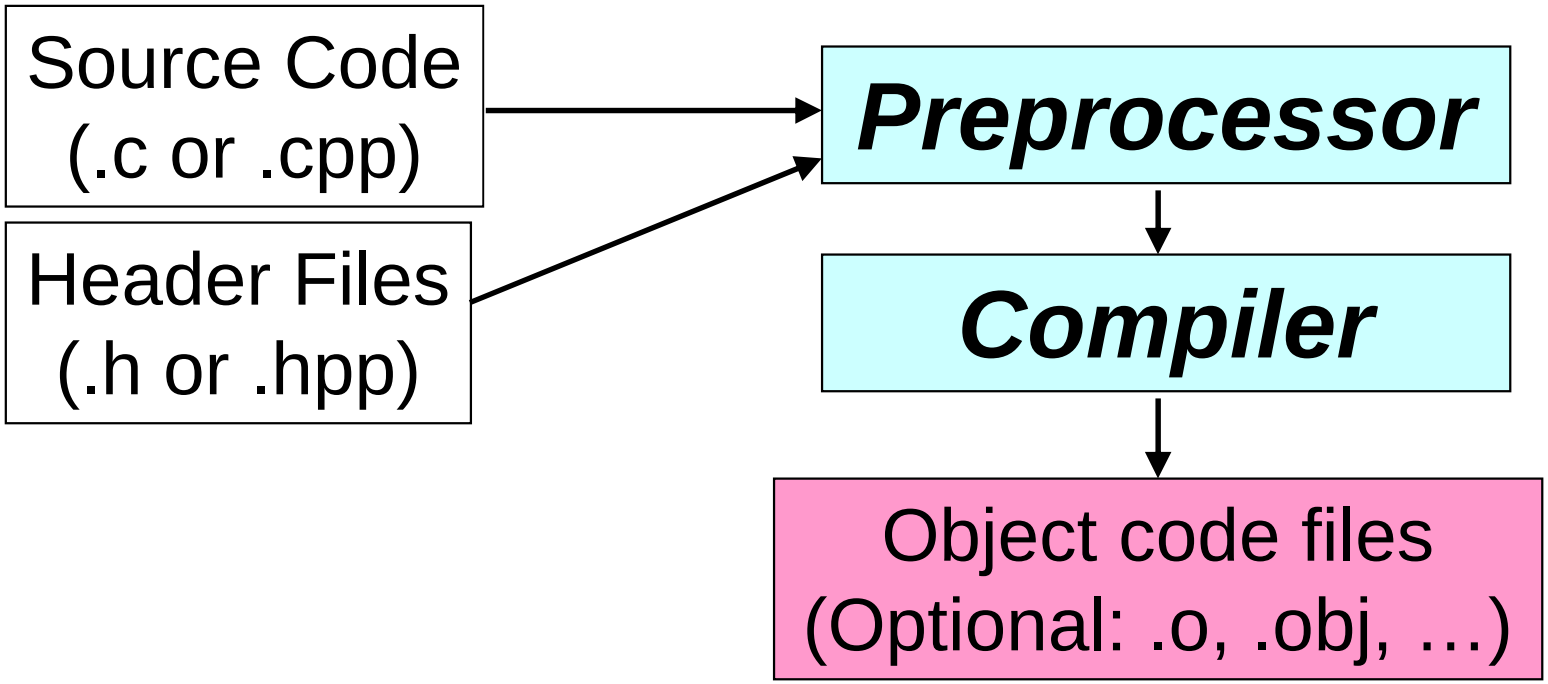
- Source code files, named .cpp or .c
  - Contain your functions and classes
- Header files, named .h or .hpp
  - **Declarations** for all functions which you want to make available to other files
    - i.e. function name, return type, parameter types
  - **Declarations** for classes, in C++
  - Any constants you want to make available
  - Any **`#defines`** to apply to other files
  - Anything else you want to share
- Library files, named .o, .lib, …
  - Already compiled
  - Contain **implementation** of library functions

Compiling and linking



gcc <.c files> -o <output_file>

Compiling and linking



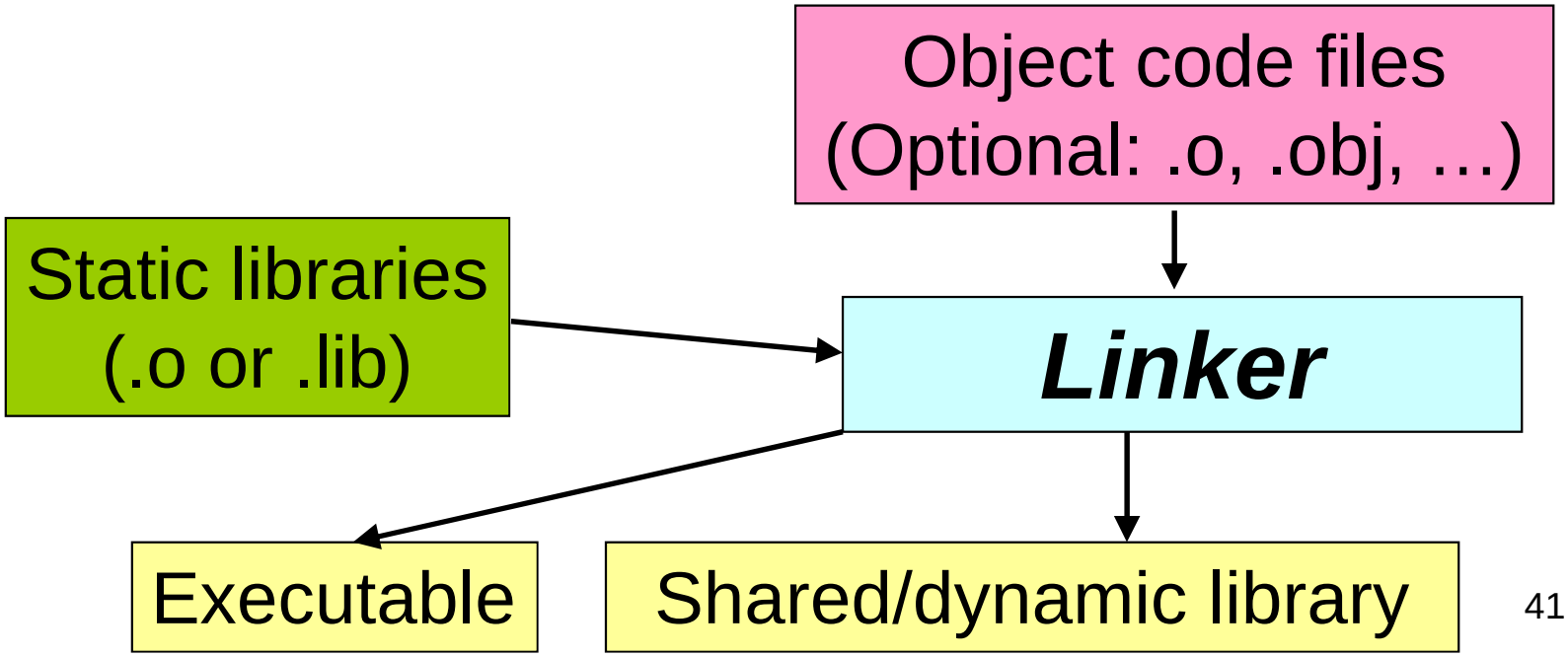
You can just compile to object code files (not link):

```
gcc –c <.c files> -o <output_file>
```

Compiling and linking

You can then link the files by passing the .o files to gcc (instead of the .c files)

`gcc` <.o files> **`-o`** <output_file>



Compiling with gcc

- **gcc** uses the file extension to determine file type when compiling/linking:
  - .c for C files
  - .cpp (and others) for C++ files
  - .o for object code files (just need linking)
- Standard C library is linked by default when compiling C code
- When compiling C++ you **need** to link in the standard C++ library files manually
  - e.g. use **`-lstdc`**++ on **`gcc`** command line
  - or (usually) can use **`g++`** instead of **`gcc`**

Stuff you should already know…

Control statements

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

Other similarities to Java

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**

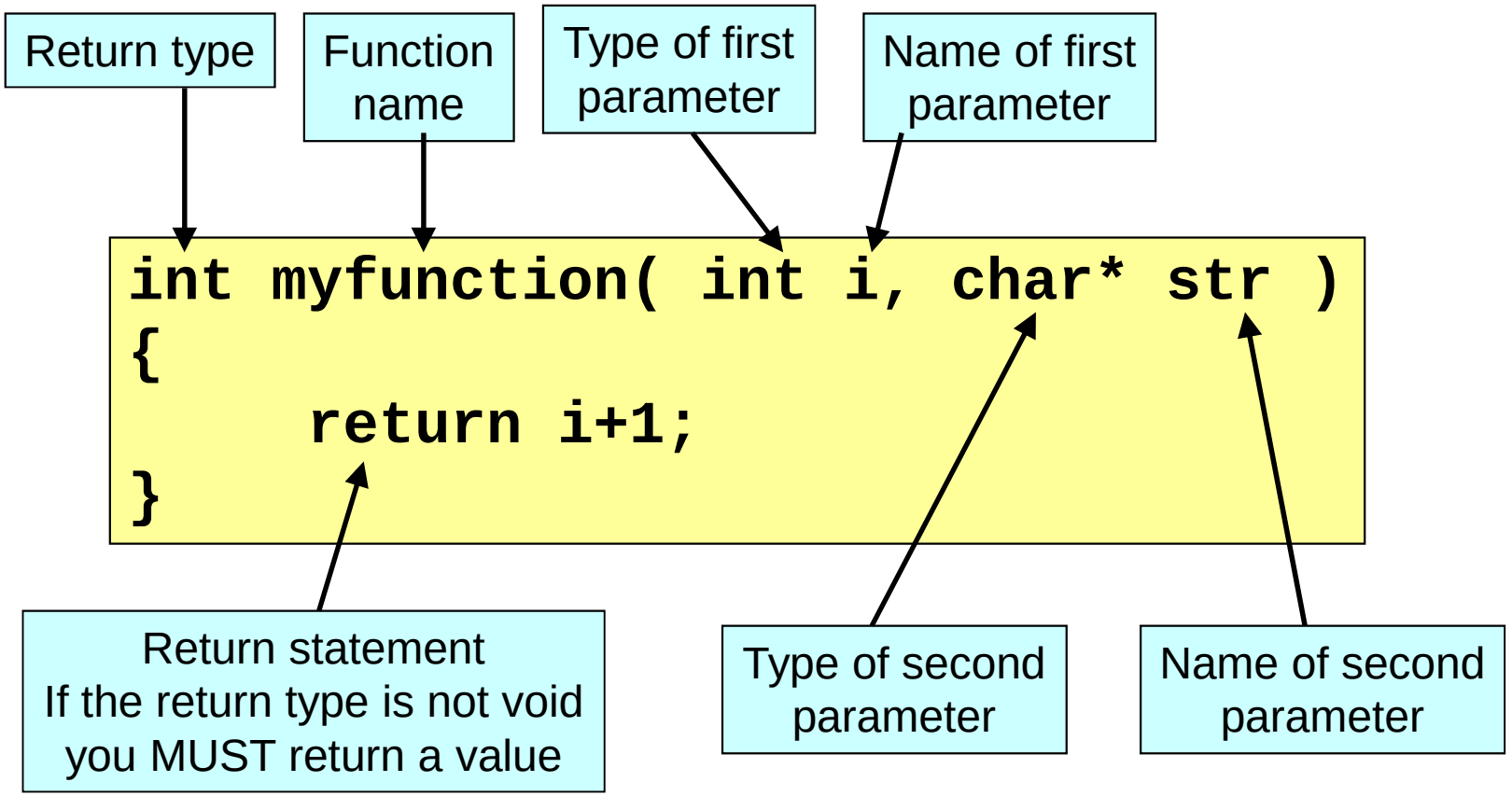
Braces: { } *

- 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
                    }
```

Functions

Functions in C



- Functions in **C** are global, not class members
- You structure your code using files not classes

Identifying functions in C vs C++

- In C a function is **identified** by its **name**
  - The name must be unique
- In C++ (and Java) the types of parameters are also considered (function overloading)
- This example is NOT valid in C89/C90
  but is valid in C++, or Java

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

Declarations and definitions (1)

- Functions should be declared before they are called (so compiler can warn about errors)
- Definitions are also declarations
- One trick is to define functions in reverse order

order_functions.cpp

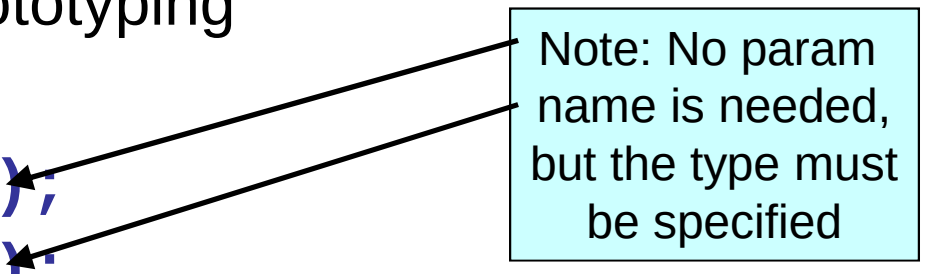
```
int myfunc2()
    { return 1; }
int myfunc1()
    { return myfunc2(); }
int main( int argc, char* argv[] )
    { return myfunc1(); }
```

Declarations and definitions (2)

- Otherwise, declare functions before usage
  - Called function prototyping
- e.g.:

```
int myfunc1(int);
int myfunc2(int);

int main( int argc, char* argv[] )
    { return myfunc1(argc); }
int myfunc1( int i1 )
    { return myfunc2(i1) + 1; }
int myfunc2( int i2 )
    { return 1 + i2; }
```



prototype.cpp

Function declarations

- **You must declare functions before use**
  - But definitions are also declarations
- Declarations usually go in header files
  - **`cstdio`** has many standard i/o function declarations
  - **`cstring`** has many string function declarations
  - You will *usually* have one header file per .c or .cpp file
    - Containing ***declarations*** of everything in the file that should be available from outside the file (i.e. functions & variables in C, also classes in C++)
- Function declarations specify only:
  - Function name
  - Return type
  - Type(s) of parameter(s)