G52CPP
C++ Programming
Lecture 8

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

Last lecture

- Dynamic memory allocation
- Memory re-allocation to grow arrays
- Linked lists

Aside: do not use variable sized arrays

- Variable length arrays are **NOT** valid in C++
  - Sadly, gcc on avon, bann etc will allow them in C++
- E.g.:

```
int myfunc( int iSize )
{
      char array[iSize];
      …
}
```

- Size of array is not a constant, it depends upon the value of variable
- **You must use a numeric literal or a constant for a size**
  - You can use a `#define` to set it to a literal (see later)
- If you need variable size arrays, use `malloc()` or `new`
- Use: `g++ -pedantic myfile.cpp` to get a warning

This lecture

- const
  - Constants, including pointers
- The C pre-processor
  - And macros
- Compiling and linking
  - And multiple header files
- Linkage and visibility

const

Defensive programming

const : constant/unchanging

- `const`ant *variables* cannot be changed
- E.g. `const int maxvalue = 4;`
- Or `int const maxvalue = 4;`

- Not really '*variable*'s anymore? Cannot be 'varied'

- `#define` could have same effect – see later
  - But, using **text replacement** in the **preprocessor**
- `const` is nicer for declaring constants
  - Multiple contradictory definitions will be caught
  - Unlike for `#define`

Pointers to constant data

- The thing pointed at through a **pointer to const** cannot be changed **using the pointer**
- E.g. `const char* p = “Hello”;`
- Or `char const* p = “Hello”;`
- **Note: `const` is to the left of the `*`**
- The following code will NOT compile:

```
const char* pc = "Hello";
*pc = 'B'; // BAD
```

- String literals should be `const char*` not `char*` and good compilers will ensure this (warnings)

Constant pointers

- You can also prevent the pointer itself from being changed, by using `const`. E.g.:

```
char* const p = “Hello”;
```

Note: the `const` is to the **right** of the `*`

- You cannot change this **pointer** to make it point at something else
- The following code will **not compile**:

```
char* const cp = "Hello";
cp = "Bye"; // BAD
```

  – i.e. catch errors at compilation!

For pointers, it matters where the const is

For constant pointers it matters which side of the `*` the `const` is:

- The **pointer** is constant – constant `short*` :

```
short * const pcs = &s;
```

- The short **pointed at** cannot be changed through the pointer – pointer to constant `short` :

```
short const * cps = &s;
const short * cps = &s;
```

- Can change neither pointer nor thing pointed at :

```
short const * const cpcs = &s;
const short * const cpcs = &s;
```

How to remember this...

- Read backwards with `*` meaning ‘pointer to’

```
float * const pcf = &f;
```

- “Constant pointer to a float”

The **pointer** is constant – constant **`float*`**

```
float const * cpf = &f;
```

- “Pointer to constant float”

`const float * cpf = &f;` (same as `float const *`)

- “Pointer to float which is constant”

The float **pointed at** cannot be changed through the pointer

```
const float * const cpcf = &f;
```

- “Constant pointer to float which is constant”

Neither the pointer nor the **thing it points at** can be changed

String literals again

- String literals should not be changed
- i.e. use **`const`** pointers

- Should use:

```
const char* str = “Hello”;
```

- Not:

```
char* str = “Hello”;
```

- Compiler should give warnings otherwise

Volatile and register

(so that you know that they exist)

‘volatile’ and ‘register’ keywords

- The `volatile` keyword is important if other threads or processes may access the data
  - **Know that it exists and when you should use it**
- Tells the compiler that data may change outside thread or program (similar meaning in Java)
- Will turn off some potential optimisations
  - Value must be checked every time it is needed
  - Compiler cannot assume it is unchanged

Example:

```
   volatile int v = 4;
   volatile float f = 16.7f;
```

- Another one to know: the ‘`register`’ keyword
  - Request to store value in a register not a variable
  - **Again, know that it exists and what it does**
  - Not usually needed with modern optimising compilers

The C/C++ pre-processor

The C/C++ Preprocessor

- Runs BEFORE passing code to the compiler
  - Compiler will only see the code after the preprocessor has changed it
- It affects statements beginning with **#**
- Examples:
  - **`#include`**
  - **`#define, #undef`**
  - **`#if, #ifdef, #ifndef, #else, #endif`**
  - **`#pragma`**

#include

- **Replaces this statement by the text of the specified file**
  - For example, to include function declarations
- E.g. **`#include <stdio.h>`**
  - Include the file with standard input/output function declarations in it (e.g. **`printf`**)
  - Looks in the directories on the include path
  - **Normally used for system header files**
  - Note: C++ standard header files may differ – but same effects
- E.g. **`#include “myheader.h”`**
  - The **`“”`** usually means look in the project path as well as the main include path
  - **Normally used for your own, project-specific header files**
- Do not confuse with Java’s ‘**`import`**’:
  - **`import`** defines the packages to look in for resolving class names (more like the C++ keyword **`using`**, but still different)
  - **`#include`** replaces the line, potentially with function declarations

Using multiple files

Reminder

- Declare functions before usage
  - Called function prototyping
  - Definitions are also declarations
    - So, sorting functions into reverse order works - all declared before use
- 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; }
```

Note: no parameter names are needed. The **return type**, **function name** and **parameter types** must be specified

Sharing things between files

- In general, you can put functions (and classes) in **any files** you wish (the filename is totally unimportant)

- **Global** variables and functions are always accessible from anywhere within the ***same*** file
  - You can ***hide*** them from ***other*** files by using the `static` keyword, e.g. :

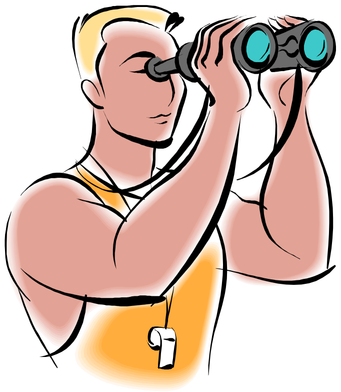
```
static int g_hidden = 1;
```

  - They are then accessible everywhere within the ***same*** file but ***not from other files***

- If ***not static*** (i.e. hidden), then:
  - You can access global functions from other files
    - Just **declare** them and the linker will do the work
  - You can access global variables from other files
    - Use the keyword '`extern`' in a **declaration**
    - **`extern` changes a definition into a declaration**

Visibility (Linkage)

- What can be seen where?



File1.cpp

```
static int hidden_in_file;
int visible_from_outside;

static void myintfunc()
{
}

void myextfunc( char c )
{
   int local_var = 2;
   static int persistant = 4;
   myintfunc();
}
```

File2.cpp

```
extern int
  visible_from_outside;

void myextfunc( char );

void myfunc()
{
   char c = 4;
   myextfunc( c )
}
```

Key Idea: Encapsulation

- The idea of hiding the internals – the data
  - Give access to the data to as few ‘things’ as possible
  - i.e. hide it as much as possible
- Controlling the *interface* which can be seen
- Why?
  - Helps with debugging and structure
  - You can see what can alter each thing
- C encapsulation can be performed using files
  - External interface (global functions)
  - Internal functions (static global functions)
- C++/Java use classes (much more control)

Summary: the `static` keyword

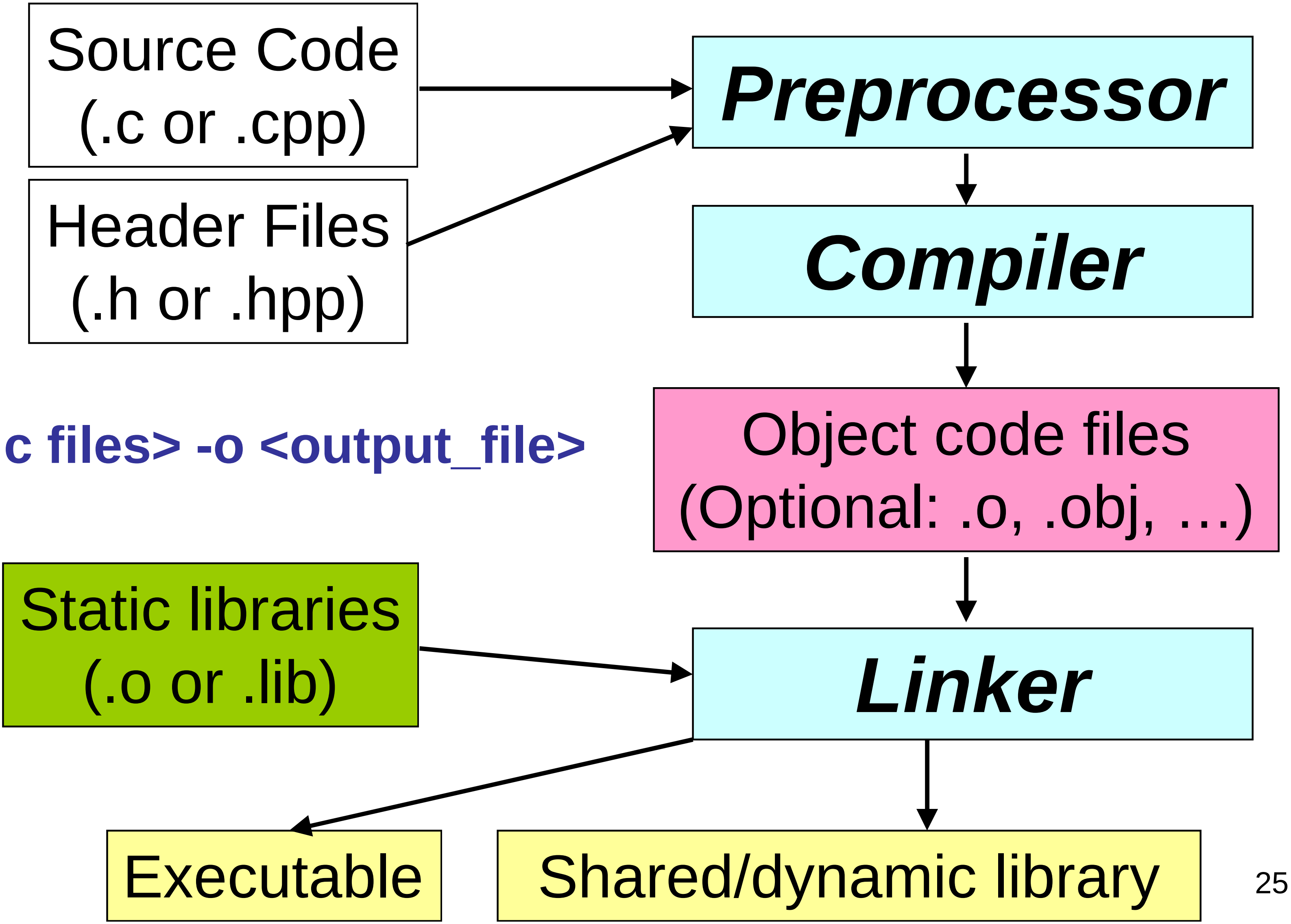
- `static` is used for three different things
  - For local variables
    - `static` means the value is maintained between function calls
  - For global variables and functions
    - `static` limits visibility/access to within the file
  - For C++ (not C!) classes:
    - Method or variable is associated with the class not the object (one copy per class, no this pointer)
    - The same as Java for this one

Compiling and linking

Types of files

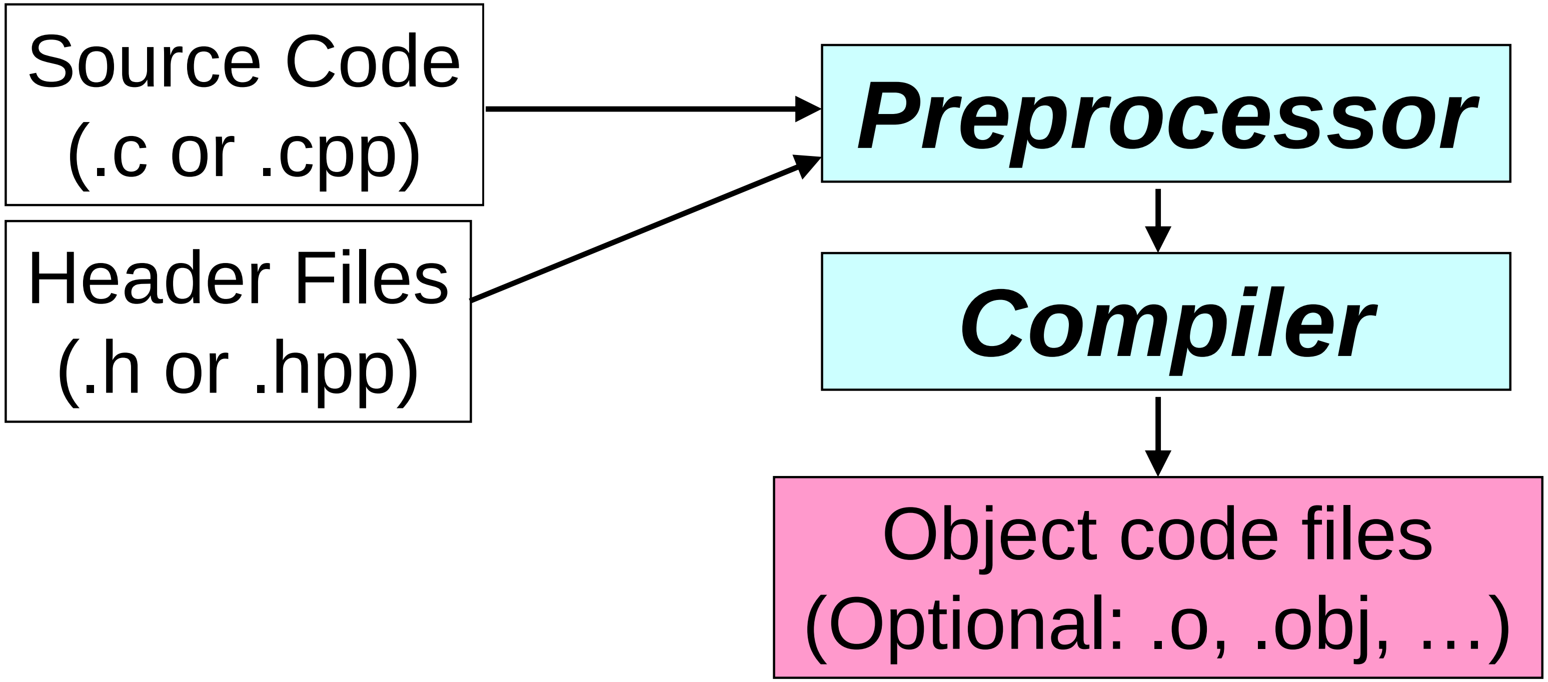
- Source code files, named .cpp or .c
  - Contain your functions and classes
- Header files, named .h or .hpp
  - **Declarations** for all functions which you want to make available to other files
    - i.e. function name, return type, parameter types
  - **Declarations** for classes, in C++
  - Any 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



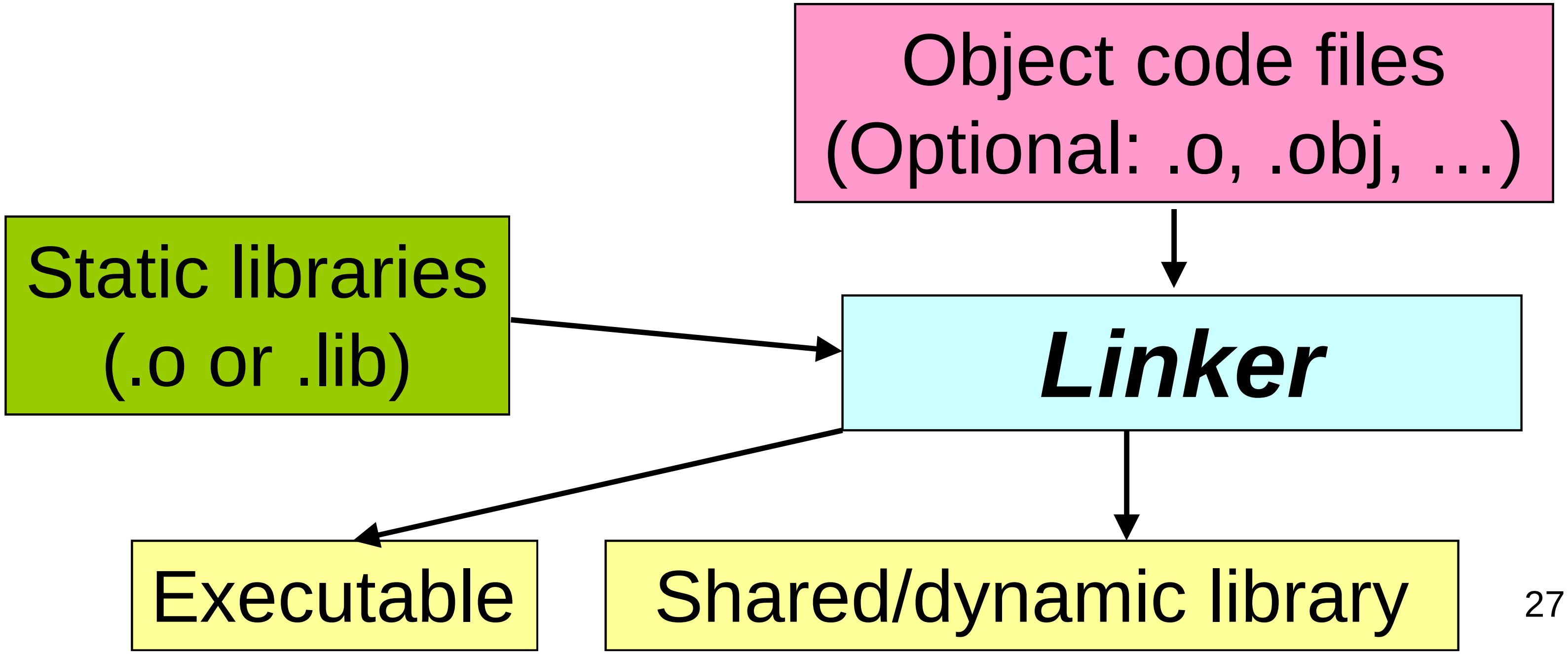
You can just compile to object code files (not link):

```
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 (often) can use **`g++`** instead of **`gcc`**

#define and #ifdef

#define

- An **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 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 it was defined>
#else
    <Include this code if it was not defined>
#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

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 slide
3. Ensure that header files can be included in any order
   - A consequence of the first two rules

#define and macro definitions

- You can use **#define** to define a **macro**:

```
#define max(a,b) (((a)>(b)) ? (a) : (b))

int v1 = max( 40, 234 );
int v1 = (((40)>(234)) ? (40) : (234))

int v2 = max( v1, 99 );
int v2 =(((v1)>(99)) ? (v1) : (99))

int v3 = max ( v1, v2 );
int v3 =(((v1)>(v2)) ? (v1) : (v2))
```

- **Remember: done by the pre-processor!**
  - **NOT a function call**

What is the output here?

MyHeader.h

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

#define max(a,b) (((a)>(b)) ? (a) : (b))

#endif
```

MyTest.cpp

```
#include <cstdio>
#include “MyHeader.h”
int main( int argc, char* argv[] )
{
  int a = 1, b = 1;
  while ( a < 10 )
  {
      printf( “a = %d, b = %d “, a, b );
      printf( “max = %d\n”, max(a++,b++) );
  }
}
```

The (surprise?) output

```
printf( “a = %d, b = %d “, a, b );
printf( “max = %d\n”, max(a++,b++) );
```

- **The output is:**

```
a = 1, b = 1 max = 2
a = 2, b = 3 max = 4
a = 3, b = 5 max = 6
a = 4, b = 7 max = 8
a = 5, b = 9 max = 10
a = 6, b = 11 max = 12
a = 7, b = 13 max = 14
a = 8, b = 15 max = 16
a = 9, b = 17 max = 18
```

- **Why?**

  `max( a++, b++ )` expands to:

  ```
  ((a++)>(b++)) ? (a++) : (b++)
  ```

- So, whichever number is greater will get incremented twice, and the lesser number only once

Next lecture

- **`class`**es (and C++ **`struct`**s)
- Member functions
- **`inline`** functions