

CSCI251/CSCI851

Autumn-2020

Advanced Programming

(S3c)

Getting organised III:

*Pre-processing, macros, and
Makefiles*

Outline

- Pre-processing: Beyond `#include`.
- Header guards.
- Assertions.
- Macros.
- More files and Makefiles.

Pre-processing: Beyond #include

- We have seen `#include` used for the inclusion of libraries.
- This is dealt with using the pre-processor.
- The pre-processor takes pre-processor directives and applies them prior to the object code being generated, and then linked.
- Effectively the text in the program we have written is modified prior to the rest of compilation.

- General syntax:
 - # at the start, no semi-colon at the end.
- Directives:
 - Source file inclusion: `#include`
 - Macro definition/replacement: `#define`
 - Conditional compilation: `#ifndef`, `#ifdef`, `#else`, `#endif`...
- Use of the preprocessor:
 - Can make the code easier to develop, read, and modify.
 - Can make the C/C++ code portable, via conditional compilation, among different platforms.
 - The `#define`, `#ifdef`, and `#ifndef` directives are sometimes referred to as header guards.

Conditional ...

- Platform dependent code ...

```
#ifdef WIN32
    ... code special to WIN32
#elif defined CYGWIN
    ... code special to CYGWIN
#else
    ... code for default system
#endif
```

Header guards

- Definitions are often only allowed to be made once.
- This is certainly true of classes and since classes are typically defined in header (.h) files we need to make sure we don't include header files through multiple paths.
- To do this we use `#define`, which generally specifies a pre-processor variable used in the text of our program prior to the rest of the compilation.

- Combined with `#ifndef` and `#endif` we can avoid multiple inclusion ...
- Here's an example from the textbook:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

#define macros

- These are used to provide replacements throughout text.

```
#define MACRO replacement-text
```

```
#define PI 3.1415926
```

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

- The pre-processor replaces instances of MACRO with the specified replacement text.
 - Change once, update everywhere...
 - Often used for constants across our code, better practise than a global variable → we don't store anything.

- We would often replace macros associated with function like operations by inline functions ...

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

inline int Max(const int a, const int b){
    if(a>b)return a;
    return b;
}
```

- Functions that are inline are not called, but rather the function is inserted in the code.
 - Or at least we *request* the compiler do this.
 - Some compilers do it automatically anyway...
- So calls to Max would possibly be replaced by
`((a)>(b))?(a):(b)`

But: ... don't use macros for constants-or-functions anyway ...

- There are better options ... see

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#e31-dont-use-macros-for-constants-or-functions>

- **Reason from there:**

- “Macros are a major source of bugs. Macros don't obey the usual scope and type rules. Macros don't obey the usual rules for argument passing. Macros ensure that the human reader sees something different from what the compiler sees. Macros complicate tool building.”

- Replace

```
define PI 3.14;
```

- with

```
constexpr double pi = 3.14;
```

Seeing pre-processing

- You can see the effect of pre-processing by using a flag on g++ compilation:

```
$ g++ -E file.cpp > ready.cpp
```

- The file `ready.cpp` tends to be much larger.

Keyword: `extern`

- **Define once, declare as often as you need.**
- Files can be compiled separately, but may still reference each other.
- The keyword `extern` is used to indicate a variable has been defined elsewhere, it's not used in the original.
- This is useful if we use a variable in file A, when it was defined in file B.

```
extern int value;
```

- This is declaring the variable exists, not defining the variable, but if we initialise the variable then the `extern` is overridden, it's a definition now.

```
extern int value = 4;
```

Predefined macros ...

- The predefined macros are mostly used for debugging...

`__TIME__`, `__LINE__`, `__DATE__`, `__FILE__`,
`__func__`

- They can be undefined using `#undef MACRO`

- Meaning:

`__TIME__` : The time the source file was compiled, a string literal of the form hh:mm:ss.

`__DATE__` : Similar but it substitutes the date, again as a string literal.

`__LINE__` : Expands to the current source line number, an integer.

`__FILE__` : The name of the file being compiled, as a string.

`__func__` : The name of the function being debugged.

Compilation and debugging...

- We can set the value of `#define` pre-processor variables at compile time.
- This is particularly useful for including debugging statements.

```
$ g++ -DDEBUG code.cpp
```

```
#ifdef DEBUG
    cout << __TIME__ << endl;
    cout << __DATE__ << endl;
    cout << __LINE__ << endl;
    cout << __FILE__ << endl;
    cout << __func__ << endl;
#endif
```

- You could set the `DEBUG` variable on in the code too but it's tidier using the command line compilation time version.

- You can include a line like ...

```
cout << TEST << endl;
```

- ... in your code and define `TEST` at compile time.

```
$ g++ -DTEST=5 prep.cpp
```

Be assertive

- The function-like pre-processor macro `assert` is used in defensive programming.
 - It's accessed using the `cassert` header.
`assert (expr) ;`
- It is typically used to check for conditions that cannot happen...
- ... more on defensive programming soon.


```
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
    #ifndef NDEBUG
        cout << "We are in debug mode" << endl;
    #endif

    assert( 5 > 3 );
    // assert( 3 > 5 );

    cout << "All is well" << endl;

    return 0;
}
```

- If the `expr` given to `assert` is false, and we are in debug mode, `assert` writes a message and terminates the program.

- The effect of `assert` depends on whether we are in debugging mode.
- How do we know?
 - After all we defined a macro for it before.
- We do something similar but use the pre-processor variable `NDEBUG`, which `assert` references.
- If `NDEBUG` is undefined we are in debug mode, so `assert` does its checks.
- But if `NDEBUG` is defined, `assert` does nothing.

```
$ g++ -DNDEBUG debug-test.cpp
```

Tracers ...

- You can add in output that appears when we are in debug mode, that is when NDEBUG is not defined.
- They can help you determine where particular problems using appropriate output.

Personalising with your own message ...

- You can something like one of these things

...

```
assert( 3 > 5 && "This is a test");
```

```
assert(("This is a test", 3 > 5));
```

- ... but commenting probably makes more sense because the compiler tells people where to look anyway.

```
assert ( 3 > 5 ); // This is a test
```

More typical use ...

- We can use `assert` with some sort of calculated function that you couldn't check the state of until run time.
- Possibly something dependent on compile time that you couldn't test until run time.

```
$ g++ -DTEST=6 file.cpp
```

```
assert( ("TEST too large", TEST < 5) );
```

- Or, something like testing the Mersenne Twister.

```
std::mt19937 mtg;
```

```
assert(mtg.min() == 0 && mtg.max() == 4294967295);
```

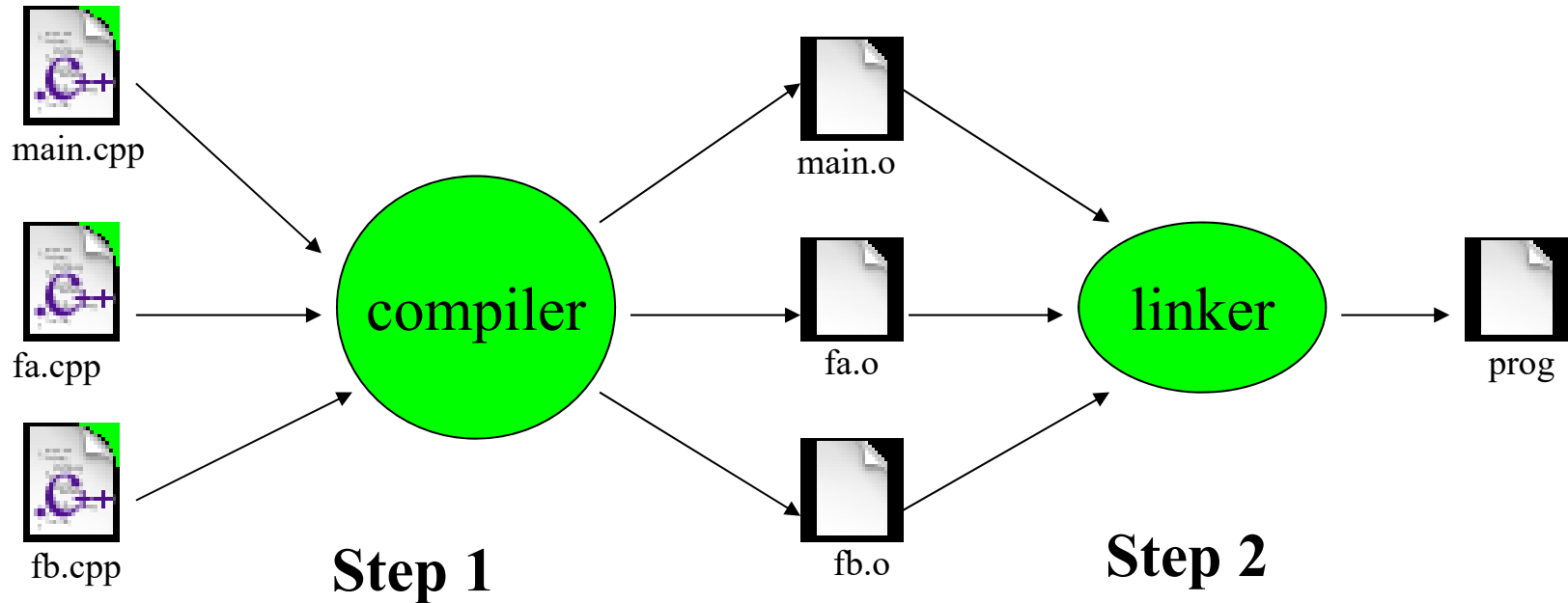
```
assert(mtg() == 3499211612);
```

```
assert(mtg() == 581869302);
```

```
assert(mtg() == 3890346734);
```

Code management ...

- If we were going to be defining the value of quite a few variables at compile time, it might be useful to set up a systematic management mechanism.
- Another part of managing code development involves handling multiple files.
 - But this can introduce some problems...



The compiler is given a number of source files.
The compiler will check the syntax of each source file and, for each, produce an object file.

The linker is called by the compiler and given all the object files.
It links the objects together with any system libraries (resolution of symbol table) and produces an executable.

```
$ g++ -o prog main.cpp fa.cpp fb.cpp
```


- If `fb.cpp` was unchanged, we could use

```
$ g++ -o prog main.cpp fa.cpp fb.o
```

- If for some reason we only wanted to produce the object file, use the `-c` flag.

```
$ g++ -c main.cpp
```

- Fine, but if we have 50 files it's going to be a pain having to correctly differentiate between the ones that have changed and those that haven't, and compiling them all may be very time consuming if we just use ...

```
$ g++ *.cpp
```

Makefiles to the rescue ...

- With *make* programming we describe how our program can be constructed from source files.
- The construction sequence is described in a *makefile* which contains *dependency* and *construction rules*.
- The makefile is itself just a text file.

- A dependency rule has two parts, a left side and a right side, separated by a colon :

left side : right side

- The left side specifies the names of *targets*; these are programs or system files to be built or processed.
- The right side lists the names of the files of which the target depends upon, e.g. source files or header files.
- If the target is older than any of the constituent parts, construction rules are obeyed.
- The construction rules describe how to create the target.

■ Let's return to our example, and note the dependencies of the source files...

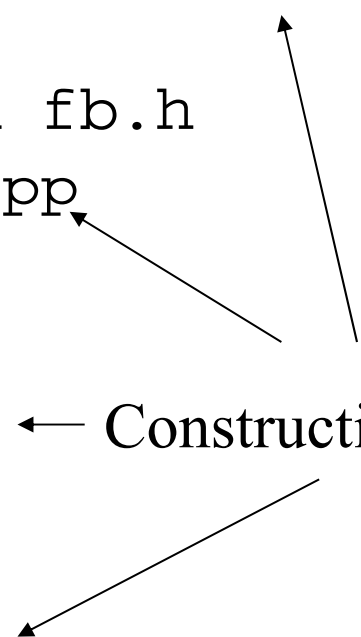
- `fa.cpp` depends on `fa.h`.
- `fb.cpp` depends on `fb.h`.
- `main.cpp` depends on `fa.h` and `fb.h`.

```
prog:      main.o fa.o fb.o
           g++ -o prog main.o fa.o fb.o

main.o:    main.cpp fa.h fb.h
           g++ -c main.cpp

fa.o:      fa.cpp fa.h
           g++ -c fa.cpp

fb.o:      fb.cpp fb.h
           g++ -c fb.cpp
```



Construction rules

Indentation
is
critical
☹

The indentation ...

- The dependency and command string should have tabs as the first character. ☹️
- This is kind of weird but if you don't do it you get something like ...

make: fatal error in reader: Makefile, line 9: unexpected end of line seen

Targeting different targets...

- When we run

```
$ make
```

- it looks for the makefile in the current directory and grabs the top target by default, but we can specify a specific target.
- This is done using the target as the argument for make ...

```
$ make fa.o
```

- Not terribly useful here but often other functionality is written into the makefile too...

- Comments are added using #, illustrated below for the fairly common tidying up target...

```
# This is a tidy up target.
```

```
clean:
```

```
    rm *.o
```

- Note there are no dependencies for clean, nothing to check.
- The `rm` above isn't the same as `rm` in Unix, it's built into make, but it serves the same purpose.
 - There are other built in commands, and you can call non built in commands too!

Arguments for makefiles

- You can list the command which make would run, without running them using the `-n` option

```
$ make -n
```

- If we use the `-d` option, as in

```
$ make -d
```

we get information about why particular actions are taken.

- You can also tell make to use a different file instead of 'Makefile' to express rules and commands, using the `-f` argument.

```
$ make -f filename
```


Macros in makefiles

- These are much like `#defines`.
 - But there must be spaces in the definition.
- They are accessed using the `$` operator, with brackets if the name is more than one character.

```
OBJECTS = x.o y.o z.o
prog:    $(OBJECTS)
         g++ $(OBJECTS) -o prog
```

- Conventionally, we include two macros, one for the compiler and one for compiler flags.

```
CCC= g++
CCFLAGS=
TARGETS= x.o y.o z.o
prog:$(TARGETS)
    $(CCC) $(CCFLAGS) $(TARGETS) -o prog

x.o: x.c x.h
    $(CCC) $(CCFLAGS) -c x.c

y.o: y.c y.h
    $(CCC) $(CCFLAGS) -c y.c

z.o: z.c z.h
    $(CCC) $(CCFLAGS) -c z.c
```

Makefiles for other purposes

```
LATEX_ARGS=  
LATEX=latex  
TARGET=Tutorial
```

```
$(TARGET).ps: $(TARGET).dvi  
    dvips $(TARGET) -o $(TARGET).ps
```

```
$(TARGET).pdf: $(TARGET).tex  
    pdflatex $(LATEX_ARGS) $(TARGET)
```

```
$(TARGET).dvi: $(TARGET).tex  
    $(LATEX) $(LATEX_ARGS) $(TARGET)
```

```
clean:  
    rm *.toc *.aux *.ps *.eps *.log *.dvi
```



```
latex:  
    latex $(TARGET)  
  
gvshow: $(TARGET).ps  
    gv $(TARGET).ps  
  
pdfshow: $(TARGET).pdf  
    xpdf $(TARGET).pdf  
  
acroshow: $(TARGET).pdf  
    acroread $(TARGET).pdf  
  
dvishow: $(TARGET).dvi  
    xdvi $(TARGET).dvi
```

