CSCI251: Advanced Programming

Defensive Programming; Reckoning Time; Pre-processing; Macros; Makefile, Debugging; Profiling

Programming defensively …

- You should aim to become familiar with the vulnerabilities of whatever language(s) you are using …

- "The whole point of defensive programming is guarding against errors you don't expect". **Steve McConnell, Code Complete**

Handling errors?

- Difference between handling errors or exceptions and defensive programming.
  - Error handling, exceptions and so on, are about handling errors that are known about and could happen.
    - Bad input and so on. It's being defensive.
- Some sources say defensive programming is *also* about handling things that shouldn't happen.
  - Effectively things that cannot happen unless something goes wrong ← A bug.

So what is defensive programming?

- the spirit of secure and reliable programming.
- defending a procedure from crashing
  - even if this means that the procedure does not perform correctly.
  - this sometimes means bug hiding.

What might it include?

- Consistent indentation makes the source code easier to read and debug.
- Do not use the default target in a switch-statement to handle a real case.
- Have cases for every valid value and throw an exception in the default case.
- “If It Can't Happen, Use Assertions to Ensure It Won't“
- Program modules should be as independent as possible.

What might it include?

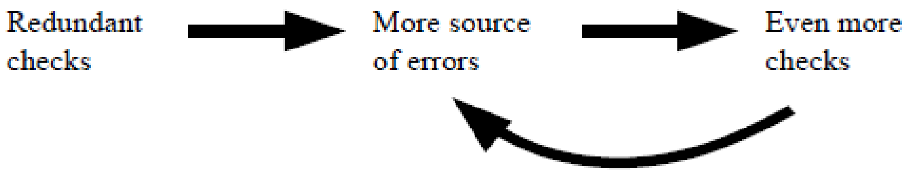
  - Validate all parameters in methods.
  - Validate all return values from methods and system calls.
  - Use meaningful error messages.

- And a definition is given as:
  - “A software development principle aiming to increase software quality, by making every method responsible for its own quality.”

A switch default example

- A function to roll a standard 6-sided die and do something different for each outcome…
- Switching on the value…
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - Default (in the switch keyword sense). Captures the what shouldn't happen case …

making every module check for many possible consistency conditions



Reckoning Time

Outline

- OS time on capa.
- C-style: std::time
- C++11: std::chrono

Looking at the time …

```
$ time ./calling

real    0m0.046s
user    0m0.032s
sys     0m0.009s
```

- If we want to know how long our program takes overall, on capa we can use `time`.
- Real: Start to end.
- User: CPU time in user mode.
- Sys: CPU time in sys mode.
- Total CPU time: user+sys

Time in C++

- Three main types are defined:
  - Clocks:
    - With an epoch and a tick rate.
      - The tick rate is effectively a number of steps within a second.
  - Time points:
    - Duration of time since epoch.
  - Durations:
    - Span of time associated with some number of ticks.

C-style dates and times
std::time in header <ctime>

- https://www.epochconverter.com/

Practice 1

```
#include <ctime>
#include <iostream>

int main()
{
    std::time_t result = std::time(nullptr);
    std::cout << std::asctime(std::localtime(&result))
              << result << " seconds since the Epoch\n";

    std::cout << CLOCKS_PER_SEC << std::endl;
}
```

```cpp
#include <iostream>
#include <chrono>

long fibonacci(unsigned n)
{
    if (n < 2) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main()
{
    auto start = std::chrono::system_clock::now();
    std::cout << "f(35) = " << fibonacci(35) << '\n';
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
}
```

But wait … Sleeping …

- With the headers `thread` and `chrono`, you can access functionality to have your program wait for some specified length of time.

```
std::this_thread::sleep_for(std::chrono::seconds(2));
std::this_thread::sleep_for(std::chrono::milliseconds(2));
```

- Time durations …

`Hours, minutes, seconds, milliseconds, microseconds, nanoseconds.`

- Additional resources:
  - https://en.cppreference.com/w/cpp/thread/sleep_for
  - http://www.cplusplus.com/reference/chrono

C++20

- There is a lot of support introduced for managing calendars and time.
- There are some specific Clocks, associated with various times.
- And functionality for converting between clocks and for partitioning time.

Pre-processing; Macros; Makefile

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
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

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.

Practice 2

- Combined with `#ifndef` and `#endif` we can avoid multiple inclusion …
- Here's an example :

```
#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)`

Practice 3

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.
  - add some pre-processing files, like #include,

Keyword: `extern`

- **Define once, declare as often as you need.**
- The keyword `extern` is used to indicate a variable has been defined elsewhere
- it's not used in the original.
- Example:
  - 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
  - if we initialise the variable then the extern is overridden, it's a definition now.

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

Practice 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.

UNIVERSITY OF WOLLONGONG AUSTRALIA

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
```

```
Codeblocks: project => build options =>
compiler setting => #defines => TEST=5
```

Be `assert`ive

- 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;
}
```

Practice 5

- 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.**
- 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 it's 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.

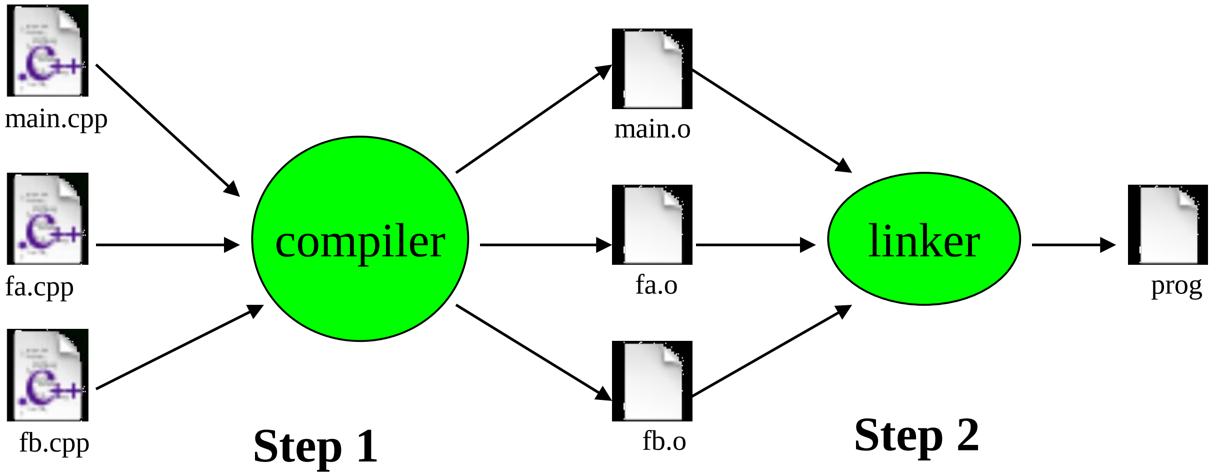
Personalising with your own message …

- You can something like one of these things …

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

Code management …

- If we were 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.



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.

Theory & Practice

```
$ 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.
    - Left depends on right
- 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

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

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

makefile:2: *** missing separator (did you mean TAB
instead of 8 spaces?).  Stop.
```

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
```

- Comments are added using #
- Example:
  - # 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, to get information about why particular actions are taken

```
$ make -d
```

- 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
```

Debugging; Profiling

Debugging …

- In some of the labs you have used the compiler to pick up problems through compilation time errors and warnings.
- That's useful but having a program compile is usually only part of the battle, we should expect there to be <u>run time problems</u> too.

- A debugger is used to step through our programs as they run, and help us pick up errors in our programs.

Using gdb

- The debugger `gdb` is available on capa, generally on Ubuntu.
  - GNU Project Debugger.
- You need to compile a program with –g flag.
  - This pretty much just stores ties to the original.
- So for Test.cpp,

```
$ g++ –ggdb Test.cpp –o Test
```

- Having compiled our test programs with the debugger information turned on we can run `gdb` for debugging.

```
$ gdb Test
```

- This loads the program into the debugger, ready for working on.
- To run …

```
(gdb) run
```

```
struct Test {
   string name;
   int number;

   void setTest(string, int);
   void showTest();
};

void Test::setTest(string TestName, int TestNumber) {
   name = TestName;
   number = TestNumber;
}


void Test::showTest() {
  cout<<"Test string " << name << endl;
  cout<<"Number for this  "<< number << endl;
}
```

```
int main()
{
	Test myTest;
	myTest.setTest("Bob", 19);
	myTest.showTest();
}
```

```
int main()
{
	Test myTest;
	myTest.name="Bobby";
	myTest.number=15;
	myTest.showTest();
}
```

Test.cpp

- We can get the debugger to step through our program, for example stopping when we get to a particular function...

Or give a code line.

```
(gdb) break showTest
Breakpoint 1 at 0xdfa: file Test.cpp, line 19.
(gdb) run
Starting program: /home/lukemc/251/Test

Breakpoint 1, Test::showTest (this=0x7fffffffe900) at Test.cpp:19
cout<<"Test string " << name << endl;

(gdb) where
#0  Test::showTest (this=0x7fffffffe900) at Test.cpp:19
#1  0x0000555555554ef6 in main () at Test.cpp:28
(gdb)
```

- At those breakpoints we can ask for variable values using `print`, as in the following example for our Test.cpp executable…

```
(gdb) print name
$1 = "Bob"

(gdb) print &name
$2 = (std::__cxx11::string *) 0x7fffffffe900
```

CHECK THIS …

- Continuing and stepping.

- You can use the command continue to keep going from a break point…
- And step and next to give/process the next lines …and then stop.

Practice

Checking memory …

- The debugger on Banshee included built in functionality for checking memory leaks.
- Unfortunately `gdb` doesn't.
- It is possible to use other tools, such as `valgrind`, to find memory leaks.
- The tool `valgrind` is not on capa ☹.
- We are going to initially use C++ code that doesn't clear dynamic memory correctly to see that `valgrind` can pick up leaks.
- If you have your own Ubuntu you can install `valgrind` using…

```
$ sudo apt install valgrind
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

MemTest.cpp

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

int main()
{

        int *ptr = new int(3);
        cout << ptr << endl;

     // delete ptr;
}
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

`valgrind` for Memory leaks …

```
lukemc@laptop:~/temp$ valgrind ./a.out
==5217== Memcheck, a memory error detector
==5217== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5217== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5217== Command: ./a.out
==5217==
0x5b7dc80
==5217==
==5217== HEAP SUMMARY:
==5217==     in use at exit: 4 bytes in 1 blocks
==5217==   total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==5217==
==5217== LEAK SUMMARY:
==5217==    definitely lost: 4 bytes in 1 blocks
==5217==    indirectly lost: 0 bytes in 0 blocks
==5217==      possibly lost: 0 bytes in 0 blocks
==5217==    still reachable: 0 bytes in 0 blocks
==5217==         suppressed: 0 bytes in 0 blocks
==5217== Rerun with --leak-check=full to see details of leaked memory
==5217==
==5217== For counts of detected and suppressed errors, rerun with: -v
==5217== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Profiling …

- Another tool that can be used to help our coding is a profiler.
- It allows us to determine how much time we are spending in different parts of our program.
- On capa we have `gprof`.
- We will initially look at this in the context of our `Test.cpp` struct test program.
- For compilation we use the flag `–pg` to prepare for using gprof …

```
$ g++ –pg Test.cpp –o Test
```

```cpp
void funA(){for (int x=0;x<100;x++){}};
void funB(){for (int x=0;x<1000;x++){}};
void funC(){for (int x=0;x<10000;x++){}};

int main()
{
        srand(time(0));
        for (int x=0; x< 1000; x++)
        {
                switch ( rand() % 3 ){
                        case 0: funA();
                                break;
                        case 1: funB();
                                break;
                        case 2: funC();
                                break;
                        default:
                                break;
                }
        }
        return 0;
}
```

calling.cpp

Why?
Functions with different costs…

- Let's look at the output from `gprof` …

```
$ g++ -pg calling.cpp -o call
$ ./call; gprof call | head
```

  - The semi-colon is used to chain multiple commands to one input line.
    - Here it makes it easier to repeat both commands together.
  - The | is a pipe, here so only the top part of the gprof output is displayed.

```
 %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 100.46      0.01     0.01      325    30.91    30.91  funC()
  0.00      0.01     0.00      341     0.00     0.00  funA()
  0.00      0.01     0.00      334     0.00     0.00  funB()
  0.00      0.00     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z4funAv
  0.00      0.00     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
```