Program style

Program style

1. Take the time for it. **Very important**. If not followed your code will be “**write only**”.
2. Common sense
3. Read “real” coding guidelines document – will give you insight how important it is. e.g. good one from Microsoft or one from Google.

Principles:

1. Readability
2. Common Sense
3. Clarity
4. Right focus

What's in a name

Example

```
#define ONE 1
#define TEN 10
#define TWENTY 20
```

More reasonable

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

What's in a name

Use descriptive names for global variables

```
int npending = 0; // current length of input queue
```

Naming conventions vary (style)

- `numPending`
- `num_pending`
- `NumberOfPendingEvents`

- Be consistent, with yourself and peers.

What's in a name

Consider (wording)

```
int noOfItemsInQ;
int frontOfTheQueue;
int queueCapacity;
...
```

The word "queue" appears in 3 different ways

- Be consistent, with yourself and peers.

What's in a name

Compare

```
for( theElementIndex = 0;
theElementIndex < numberOfElements;
theElementIndex++ )
elementArray[theElementIndex] = theElementIndex;
```

and

```
for( i = 0; i < nelems; i++ )
    elem[i] = i;
```

Use short names for locals

What's in a name

Use active name for functions

```
now = getDate()
```

Compare

```
if( checkdigit(c) ) …
```

to

```
if( isdigit(c) ) …
```

Accurate active names makes bugs apparent

Indentation

Use indentation to show structure

Compare

```
for(size_t n=0; n <100; field[n++] = 0);
c = 0; return ‘\n’;
```

To

```
for(size_t n=0; n <100; n++)
{
   field[n] = 0;
}
c = 0;
return ‘\n’;
```

Expressions

Use parentheses to resolve ambiguity

Compare

```
leap_year = y % 4 == 0 && y %100 != 0
            || y % 400 == 0;
```

to

```
leap_year = ((y % 4 == 0) && (y %100 != 0))
             || (y % 400 == 0);
```

Statements

Use braces to resolve ambiguity

Compare

```
if( i < 100 )
x = i;
i++;
```

To

```
if( i < 100 )
{
  x = i;
}
i++;
```

Idioms

Do not try to make code "interesting"!

```
i = 0;
while( i <= n-1 )
{
    array[i++] = 1;
}
for( i = 0; i < n; )
{
    array[i++] = 1;
}
```

```
for( i = n; --i >= 0; )
{
    array[i] = 1;
}
for( i = 0; i < n; i++ )
{
    array[i] = 1;
}
```

This is the common "idiom" that any programmer will recognize

Idioms

Use "else if" for multiway decisions

```
if ( cond1 )
{
   statement1
}
else if ( cond2 )
{
   statement2
}
…
else if ( condn )
{
   statementn
}
else
{
   default-statement
}
```

Idioms

Compare:

```
if( x > 0 )
   if( y > 0 )
     if( x+y < 100 )
     {
        ...
     }
     else
        printf(Too large!\n" );
   else
      printf("y too small!\n");
else
   printf("x too small!\n");
```

```
if( x <= 0 )
{
   printf("x too small!\n");
}
else if( y <= 0 )
{
   printf("y too small!\n");
}
else if( x+y >= 100 )
{
   printf("Sum too large!\n" );
}
else
{
  ...
}
```

Comments

Don’t write the obvious

```
// return SUCCESS
return SUCCESS;
// Initialize total to number_received
total = number_received;
```

Test:

Does comment add something that is not evident from the code?

Note: “//” Comments are Not Strict ANSI C, but supported by all modern compilers, and clearer in many cases

Comments

Don’t comment bad code – rewrite it!

```
...
// If result = 0 a match was found so return
// true; otherwise return false;
return !result;
```

Instead

```
...
return matchfound;
```

Code needs to tell a story

Every line or at least several lines
should be self explantory
about **what** it does,
even if we don’t understand **how**

Style recap

- Descriptive names
- Clarity in expressions
- Straightforward flow
- Readability of code & comments
- Consistent conventions & idioms

Why Bother?

Good style:

- Easy to understand code
- Smaller & polished
- Makes errors apparent

Sloppy style ➔bad code

- Hard to read
- Broken flow
- Harder to find errors & correct them

Note - style in slides:

In the slides we try to follow the style we dictate.

But due to space restriction we sometimes bend the rules.

Don’t follow everything you see in the **slides**.

Test Driven Development using Unit Testing

The Testing Problems

Should write

Do

programmers

few

Why?

I am so busy

It is difficult

OK, I'll write tests, but why a framework ?

Disadvantages:

- I need to learn a new thing
  - *True*—but done once
- You don't have time to do all that extra work
  - *False*—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite

OK, I'll write tests, but why a framework ?

Advantages of having a test suite

- many **fewer bugs**
- **Easier to catch bugs** if you'll have ones
- **a *lot* easier to maintain and modify your program**
  - This is a **huge win** for programs that, unlike class assignments, get actual use!
- You can get 100 in this course only if you'll write Google tests

Recommended approach

1. Design your code
2. Write a stub for all functions
3. Write tests that just fail for all functions, if it is hard to use the functions, return to step 1
4. For all units (functions, classes, etc.):
    1. For all usage scenarios:
        1. Write a test for the scenario
        2. Replace the stub with code, just enough to pass the test
        3. Run the test
        4. If fails debug the unit and test until it passes
    2. Remove the test that failed that you wrote in step 3 for the unit
5. Whenever you change your code or find a bug add a test for the change or the bug

Refactoring (improving your code)

- Refactoring is for making your code easier to understand, more efficient (only if needed!), etc.
- Refactoring is required because code tends to become messy.
- Refactoring should not change the functionality of the system.
- Automated testing simplifies refactoring as you can see if the changed code still runs the tests successfully.

Unit tests

- Test each unit of program **separately.**
- Test that it fulfills it's **contract**.
- Writing tests – part of the **coding (before implementation)**.
- Running tests – part of the **build** process.
- Tests **output** – only when **failed**.

Unit tests – what is it good for

- Save **time** of debugging.
- Finding problems in **design** early.
- Helps to build **modular** code (if it's hard to find units in your program – it's not modular)
- **Refactoring** easy.
- Easier to work in **teams**.
- Live **documentation**.
- Everybody writes some kind of tests – doing it with some kind of **framework saves time**.
- ...

Example: abs_test.zip

Black box testing

- Checks only that the output is as expected for the input, without checking the internals
- For example: test an efficient and hard to code algorithm with a non-efficient easy to code algorithm

Debugging

Debugging 101

1. “Define” the bug --- reproduce it
2. Use debugger or/and printouts (and other tools e.g. valgrind)
3. Don’t panic --- think!
4. Divide & Conquer
5. Test before instead of debugging after

General purpose debug function

```
// file: my_debug.h
// defines a general purpose debug printing function
// usage: same as printf,
// to disable the printing define the macro NDEBUG
#ifdef NDEBUG
   #define printDEBUG(format, ...) (void (0))
#else
   #define printDEBUG(format, ...)  \
       printf(format, ##__ VA_ARGS__)
#endif
```

Debugger

- See how the program runs, value of variables...
- Breakpoints, break on expressions change...
- Stack Trace: very helpful for seg faults!
- DANGER: debuggers tend to make people not think about the problem!
- Many excellent programmers don’t use a debugger, or as it is called: use “debugging in your mind”

Optimization

What smart people say about Premature optimization…

Premature optimization is the root of all evil (or at least most of it) in programming – Donald Knuth

So, what to do?

- Check if you **need** to optimize and what
- Remember to **“turn off” debugging** (#define NDEBUG)
- Check what **your compiler** can do for you on **your specific hardware** (-O3 -march=pentium4 -mfpmath=sse, inlining of functions, ...)
- **Profile**: check **where** to optimize
- Use **common techniques** such as cache,...