

# CSE 220: Systems Programming

## Programming Practices

Ethan Blanton

Department of Computer Science and Engineering  
University at Buffalo

# Effective Programming

The difference between a programmer and a good programmer is large.

Some of that difference is talent and knowledge.

A lot of that difference is experience and practice.

There are practices you can adopt to become a better programmer.

# Work Habits

The advice in this section is mostly **things you already know**.

- Start early
- Work diligently
- Comment and document
- Write a second draft
- **Read and write**

# Start Early

Start your programs early.

This is more than just time management.

Think about where and when you've had programming insights.

- Shower?
- Driving?
- Walking?
- Anywhere there's no way you can type it in?

Your subconscious will work for you if you give it time.

# Comment and Document

Comment your code **judiciously**:

- Include insightful comments
- Avoid useless comments

Never do this:

```
i++;                               /* Increment i */
```

Document **while you are writing the code**.

This will help crystallize your ideas and identify logical errors.

# Write a Second Draft

*Plan to throw one away; you will, anyhow.*  
— Fred Brooks, *The Mythical Man Month*

If you find that your approach is getting unwieldy:

- Stop and consider what you've learned
- **Rewrite** as necessary!

# Getting Started

Sometimes the hardest part is **getting started**.

Find **something you know how to do**, and **do it**.

Maybe you can:

- Process program arguments
- Perform a simple sub-calculation
- Define a data structure

Once the problem is **started**, it seems more tractable.

# Read and Write

Read [documentation](#):

- Man pages
- API specifications
- Standards

Read [Programming texts](#)

- There are several in the references

[Write code](#)

- There is [no substitute!](#)

Write [documentation](#)



# Top Down and Bottom Up

For **many projects**, I recommend a two-pass process:

- Divide the task **top down**
- Implement **bottom up**

# Top Down Design

Recursively apply the following steps:

- Identify the problem to be solved
- Determine **what you need** to solve it
- Define a function/data structure/*etc.* to obtain what you need
- Apply this method to each of those things

Try to identify **common functionality among tasks** while doing this.

# Bottom Up Implementation

Recursively apply the following steps:

- Identify sub-tasks you **know how to solve**
- Solve them
- Identify sub-tasks that can now be solved

You may need or want to **refine your top-down design** during this phase!

# Managing Complexity

During development, you may find **complexity growing**.

You can manage this by:

- Identifying routines that can be abstracted into functions
- Defining and using constants
- Creating data structures to simplify computation
- Using standard library functions

# Tools

Using tools effectively is critical to efficient programming.

These tools might include:

- Your editor
- The compiler
- Build system tools such as make
- The debugger
- Text or data processing tools

It's worth taking extra time to learn your tools.

*It will pay itself back!*

# The Compiler

The compiler is **very helpful in producing correct code**.

**Always** compile with `-Wall` and maybe `-Wextra`.

**Silence warnings**.

Use **functions** instead of macros to get type checking.

Use the preprocessor for debugging.

# The Debugger

You **don't have time** to not learn gdb.

Learn when to `printf()` and when to `gdb`.

Explore `xxgdb`, Emacs GUD, scripts, *etc.*

# Your Editor

Find a good editor, and **trust it**.

If it thinks something is hinky, **figure out why**.

For example:

- It wants to indent funny
- It colors a variable name unexpectedly
- It can't find a completion
- ...

This may mean things like:

- You've misplaced braces
- You're shadowing a system variable
- *etc.*



# Helper Functions

Use [helper functions](#) to:

- Factor out repeated operations
- Reduce the state in any given function
- Provide debug assistance

Declare [file local](#) helper functions [static](#).

Declare [project-wide](#) helper functions in header files.

Keep an eye out for [refactoring opportunities](#):

- Easy ways to handle more cases in the same function
- More code that can be lifted into helpers
- Different approaches that factor out larger blocks

# Types

Pay close attention to types!

**Don't** fix type errors **without understanding them!**

Declare variables as the **tightest type possible**:

- Prefer something `*` over `void *`
- Prefer something`[]` over something `*`
- Prefer `int32_t` over `int`
- ...

# Magic Values

**Never** use magic values!

Use named constants instead of integers or strings with semantic meaning.

```
#define MESSAGE 2
#define LIVE 'X'
```

Once you have them, use them.

```
*(int *)packed = 2;    // WHYYYYYYYYYY???????
```

# Format Your Code

Format your code **precisely**.

The style you pick is not as important as **picking a style**.

Badly-indented code **should bother you**.

Code formatting should **help you spot logical errors**.

# Invariants

Invariants are properties of a program that are always true, or predictably true.

We often speak of loop invariants.

You should know and define invariants.

A professor once told me:

*If you write a loop and you don't know its invariant, it's wrong.*

# Violating Invariants

Invariants must often be violated **temporarily**.

If you violate an invariant, you must:

- **not invoke code** that expects it to be maintained
- Know **when and where** it will be restored
- Ensure that it is restored **on every code path**

Example:

A doubly-linked list's prev and next pointers are inconsistent **during node insertion**.

# Pre- and Post-Conditions

Closely related to [invariants](#).

Rules that must be maintained [before and after](#) an operation.

- Loop
- Function
- I/O
- *etc.*

[Identify and document](#) pre- and post-conditions in comments!

[Verify conditions](#) at run time!

# Make Purposeful Changes

Don't just **change code** without forethought.

Make **purposeful changes** designed to **address an issue**.

It is better to take longer and understand the problem.

**Programming by Brownian Motion** is seldom successful.

Sometimes **quick fixes** **cover up a problem** without fixing!



# Summary

- Cultivate **good work habits**
- **Design** your programs purposefully
- **Use your tools!**
- Practice **good style and form**
- Debug **with a plan**

The **only way** to become a good programmer is to **write programs**.

# References I

## Optional Readings

- [1] Andrew Hunt and Dave Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- [2] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. 20th Anniversary Edition. Addison-Wesley, 1995.
- [3] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

# License

Copyright 2019 Ethan Blanton, All Rights Reserved.

Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.