Programming in the Large

Tips, Tools and Techniques for Managing Larger Programs

CSCI 3700 — Data Structures and Objects

Department of Computer Science and Information Systems
Youngstown State University

Robert W. Kramer

Outline

Planning

- Small tasks need little planning

  e.g., planting a small garden

- Larger tasks demand planning

  e.g., building a house

Note

The same principle applies in programming!

Large programs need forethought

How Do We Plan Programs?

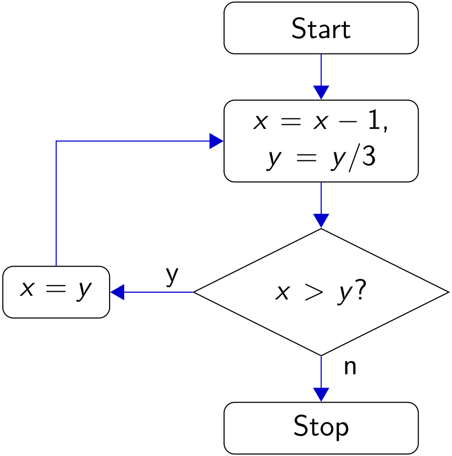
- We need a design methodology
  Rules / guidelines for program development
- Top-down design (from 2605 / 2610)
  - Break large problem into smaller parts
  - Defer details of solutions to smaller parts
  - Recursively break down smaller parts if necessary

How To Write A Program Plan

- Plans are written using *algorithms*
  - Step-by-step procedure
  - Steps are precise
    - No ambiguity!
  - Steps are finite
    - They come to an end eventually
- Two common methods for writing algorithms
  - Flowcharts
  - Pseudocode

Flowcharts

- Visual representation of control flow
  - Shows how program proceeds
- Unwieldy
  - Too large for any but smallest programs
  - Does not show structured programs well



Pseudocode

- English-like
  - Cross between English and code
- Two key properties
  - Precise
    - Accurately express an algorithm
  - Expressive
    - Easy to read

Set $s \leftarrow 0$
Set $d \leftarrow 1$
While $n > 0$, do {
    $s{+}{+}$
    $n \leftarrow n - d$
    $d \leftarrow d + 2$
}

Pseudocode

- No set rules for writing
  - Good pcode just has to have the key properties
- Everyone has their own style
  - Mine is a cross between English and C

Program Development

- Obtain and *understand* problem specifications
- Top-down design / stepwise refinement
  - Break problem into small chunks
- Develop algorithms for each chunk
  - This determines chunk size
- Convert algorithms to code
- Test the resulting program
- Repeat previous steps as necessary

Where Problems Occur
Problem Specification Issues

Problem Specifications

Not fully / correctly understanding the problem

Suggestions

- Start early
- Ask questions
- Try to freeze specifications

Where Problems Occur

Top-Down Design Issues

Top-Down Design

Not breaking down the problem sufficiently

Suggestions

- Problem is sufficiently small when...
  - There is only one task involved
  - You understand how to solve it

Where Problems Occur

Algorithm Development Issues

Algorithm Development

- Ambiguity
- Not correctly solving the problem at hand

Suggestions

- Write out the pseudocode!
- “Desk check” the code
  - You play the role of computer
  - Test code by hand

Where Problems Occur
Coding Issues

Coding

Not correctly coding the algorithm

Suggestions

- Incorporate pseudocode into program
- Test, test, test!

Naming Conventions
What Do We Give Names To?

- Variables and objects
- Constants
  - *e.g.*, better to use `M_PI` than 3.14159265358979323846
- Functions
- Files

Naming Conventions

Guidelines 1/2

Two opposing guidelines:

- Make names descriptive
  - *e.g.*, `thetaTable` instead of `qq`
- Keep them short
  - `antidisestablishmentarianism = supercalifragilisticexpialidocious * zungguzungguguzungguzeng`

Naming Conventions

Guidelines 2/2

Two more guidelines

- Break apart multiple words
  - Underscore *e.g.*, `two_words`
  - Capitalize *e.g.*, `twoWords` or `TwoWords`
- Abbreviate (consistently!)
  - *e.g.*, `nLoci` or `pPedigree`

Documentation
Where to Find Documentation

Two locations

- Internal — within the program
    - This is our focus here
- External — separate document
    - Covered in other courses

Literate programming — a hybrid

- Documentation and program interleaved in one file
- Special tools used to generate PDF and source code

Documentation

Types of Internal Documentation

Two types of internal documentation

- Comments
- Self-documenting code

Comments

Comments should be used sparingly

- Top of file
- Top of function
- Key steps of algorithm
- Unclear code

Comments
Top of a File

- Name of file
- What's in it
- Who wrote it
- Who changed it

```
//
// rns.h
//
// macros for residue number
// system arithmetic
//
// written 29 july 2014 by rwk
//
// modification history
//   2 august 2014 - rwk
//     fixed error in approximate
//     logarithmic value
//
```

Comments
Top of a Function

- Function prototype
- One-line description
- Description of parameters
- Description of return value
- Other information
- Modification history

```
//
// bool isPrime(u64 n)
//   determine if n is prime
//
// Parameter
//   n — unsigned long long int to test
//
// Returns
//   true iff n is prime
//
// Note:
//   Uses global table primeList and
//   global int nPrimes
//
```

Comments
Key Steps in Algorithm

Show each step of algorithm

Hint!

Type your pseudocode into your source file before coding

```
int main(void) {
  // step 0: initialization
  ...
  // step 1: gather required data
  ...
  // step 2: process data
  ...
  // step 3: output results
  ...
  return 0;
}
```

Comments
Unclear Code

Comment code that...

- uses advanced algorithms
- is purposely obfuscated
- uses a complex calculation
- is otherwise hard to read

```
// step 3.2: find all occurrences
...
// note: the next block uses the
//  Boyer-Moore string matching
//  algorithm
...
```

Self-Documenting Code

The Art of Making Your Code Readable

Three guidelines:

- Good use of names
- Indent your code!!!
  - It helps catch errors
  - It helps make code readable
- Avoid clever code. . .
  - unless there's a good reason for using it

Modularity

Types of Modularity

Modularity comes in three flavors:

- Function modularity
- File modularity
- Object modularity

Function Modularity

One Function, One Task

- Function performs only one task
  - More than one? Break it up
  - Task may be high or low level
- Function performs task efficiently
  - As efficiently as we know how
- Function has clean interface
  - Communicate via parameters and return only
  - No global variables*

File Modularity

A Logical Separation

- Related functions should be grouped together
  - *e.g.*, functions contained within a class
- Groups should be kept separate
  - Easier to maintain
- Tools exist to combine separate files
  - *e.g.*, `make` and `ant`

Object Modularity

A Logical Combination

- Sometimes data should only be accessed in a certain way
- Sometimes code designed to only work with certain data
- Sometimes code and data designed to only work with each other

An Object

An object contains:

- Data
- Functions (*methods*) that access the data

Why Modularity?

Why Bother if it's More Work?

- Testing and debugging
  - Small code blocks are easier to work with
  - Self-contained function $\rightarrow$ bugs are localized
- Efficiency
  - Easier to replace module with more efficient version
- Project development
  - Parallel / independent development
- Software reuse
  - Drop modules into other programs

Summary

Three ideas examined:

- Software development
    - Many steps involved, follow them all
    - Each has its own issues
- Documentation
    - Good use of names and comments
- Modularity
    - Logical collections of code and data
    - Many advantages to doing this