The practice of programming
Tuesday, October 09, 2012
7:59 AM

So far, we've discussed the theory of software engineering.
   How to design things well.
   How to control costs.
But in the end, it all comes down to writing programs.
   How do we best conduct ourselves while writing code?
   How can we best use our time?
"Best practices"

In the following lectures, I present several aspects of programming usually called "best practices". Not necessarily the very best way to program. Simply an agreed-upon set of practices that lead to high-quality programs. No "theoretical" justification: this is a totally practical matter.

Two kinds of justification:
Horror stories: do this "or you'll be sorry".
Success stories: we did this and were happy with it.
Common causes

The most common causes of programming problems

Lack of **readability** of code.
Lack of understanding of **consequences**: cannot determine what will happen if a program is changed.
Lack of ability to **localize problems**: cannot determine where a problem occurs in the code.
Lack of ability to **control versions** of programs: cannot change all versions at the same time; cannot know which version has a problem.
Hierarchical organization of code files.

Consistency in indenting each kind of entity.

Commenting:
Document why, and not what.
Document preconditions, postconditions, and invariants for every subroutine.

Coding:
Conditional debugging: make debugging writes possible to turn on and off as needed.
Antibugging: use code to explicitly check that expectations are fulfilled; stop error propagation.
Portability: use conditional compilation to merge versions of programs intended for different architectures.
Version control: utilize versions to keep track of changes.
Indentation and readability

Choose conventions for indentation.
Be consistent within and across files.

My conventions for C and C++:

```c
int foo(arg1, arg2, ...) {
  ^ aligns
  Contents indented
}

if (foo) {
  ^ aligns
  Contents indented
} else {
  ^ aligns
  Contents indented
}

while (foo) {
  ^ aligns
  Contents indented
}

while (A
  ^ aligns
  && B) {
  Contents indented
}
```
Commenting basics

Comments form a hierarchy:
A comment describing the module.
A comment describing each subsection.
A comment describing each subroutine.
Comments as needed to describe blocks of a subroutine.

Typical practices:

// a top-level comment describing high-level concepts

// ** an intermediate-level comment **

// a low-level comment
Commenting best practices

For every file, document:
The cohesive forces that cause it to be implemented in a single file.

For every section of a file, document:
The theme of the section.

For every function, document:
Purport: what it does.
Arguments
Preconditions
Postconditions

Test cases
Examples
Advanced commenting

**Literate programming**: combining the documentation for the code with the code. (Knuth)

Arguments for literate programming:

**Principle of locality**: keep all documentation "near" the code it documents.

Easiest to update documentation when code changes; discrepancies are easier to notice.

Foremost tool for literate programming: **doxygen**
Doxygen

A documentation generator.
Uses special embedded comments inside code.
Generates latex documents, as well as websites.

www.doxygen.org

Portable among many languages.

(It lies about its capabilities! No luck with <tt> --- </tt> markup.)
Doxygen basics

Too many options: choose one convention and be consistent.

For the purposes of this exercise,

    ///
    /// This is doxygen text embedded in code.
    ///

is the convention we will use.
   A blank /// line before.
   Embedded documentation.
   A blank /// line after.

One other form proves handy:
declaration; ///< documentation of declaration.
Doxygen markups

Start with \Inspired by LaTeX.
Enable cross-referencing of code.

For documenting files
\file filename paragraph of description
\brief Brief description of file.
(no tag) Detailed description of file
(optional/default keyword)
\page pageName paragraphs of
documentation for a "page", which is a unit
of user documentation.

For documenting functions and procedures
\param pname Parameter documentation
  \param[in]: an input parameter.
  \param[out]: an output parameter.
  \param[in,out]: an input/output
  parameter.
\pre Paragraph of preconditions.
\post Paragraph of postconditions.
\return Paragraph describing any return
  values.
\test Paragraph describing a test case.
Doxygen indentation conventions

- Blank line starts a new paragraph.
- "-" starts a new bulleted list entry.
- "-#" starts a new numbered list entry.
- Indentation convention: continuations of paragraphs must be at same indent level.

  - This is a bulleted item without a continuation.
  This is another paragraph.
  - This is a bulleted item with a continuation.
  \param foo A description of foo in two lines.
  \test This is a two-line description of a test case.
Full power of doxygen
Be able to think in OO terms.
With or without language support.

\class className Description of the class.
Markups within a class:
\extends className: is a subclass of.
\implements classname: conforms to an interface.

Markups within a member function
\memberof className: declares a function
as a member function in a language
without native class support.
\public, \private, \protected: same meanings as in C++.
Doxygen page mode

Doxygen page mode

Allows construction of documentation pages by concatenation.

Allows documenting non-code features such as user interfaces and help text.
Limits of doxygen

Only documents interfaces; not useful in documenting code blocks.

User documentation can be constructed from "\page" markups.
A detailed example:

See [http://www.cs.tufts.edu/comp/250PSD/Practice/assert.h](http://www.cs.tufts.edu/comp/250PSD/Practice/assert.h)

First: the file documentation:

```c
/// 
/// \file assert.h
/// \brief Support for C and C++ assertion-based guard clauses and debugging.
/// This file includes defines that report probable errors via \c STDERR
/// including execution context (file and line number). Inline functions that
/// do the actual printing are included here for speed.
/// \version 1.0
///
```
Documentation for a #define

/// /// Test an expression and exit the program if it is false. /// \param[in] bool An expression to test, which should represent an /// integer value. /// \pre \c bool is a boolean expression that evaluates to an integer /// \post If \c bool evaluates to 0 (FALSE), error is printed to \c STDERR /// and the program terminates /// \test \c ASSERT(1) should print nothing. /// \test \c ASSERT(0) should print /// "Assertion 0 failed in line ??? of file ???" /// where \c ??? are the appropriate file name and line number, /// respectively. It should then terminate execution with exit code 1. /// #define ASSERT(bool) assert((bool),#bool,__FILE__,__LINE__)
Documentation for a function

///
/// \brief Test a boolean assertion and exit the program if it fails.
/// This function checks a boolean assertion and prints detailed information
/// on any failures to \c STDERR, before it exits. It is called only from the
/// \c ASSERT macro, which instantiates parameters 2-4 from its context.
/// \param[in] boolean A boolean expression to test.
/// \param[in] boolstr The stringification of the expression, for printing.
/// \param[in] file The file in which the assertion is checked.
/// \param[in] line The line number of the file at which the assertion occurs.
/// \pre Assured by the \c ASSERT macro.
/// \post If \c bool evaluates to 0 (FALSE), error is printed to \c STDERR
/// and the program terminates
/// \test assert(1,"foobar", "/bin/foo.c", 42) should print nothing and return.
/// \test assert(0,"foobar", "/bin/foo.c", 42) should print
/// "Assertion foobar failed in line 42 of file /bin/foo.c"
/// and then the program should terminate with exit code 1.
///
inline void assert(int boolean, char *boolstr, char *file, int line) {
...
}
Static versus dynamic code analysis
Tuesday, October 09, 2012
5:12 PM

Static: determined by the shape of the source code.
   All doxygen can do is static analysis.
Dynamic: determined by the shape of execution.
   Doxygen doesn't help.
   Need other mechanisms.
Self-protecting code

So far, we've discussed how to document preconditions and postconditions for a function.

It is also possible to make a function self-protecting, in the sense that it won't accept inputs not conforming to a specification.

Key to this kind of protection: the assert.h I just documented.

This is a dynamic analysis technique: determined by the shape of execution.
Antibugging

Antibugging
The practice of writing code so that bugs cannot occur without your knowledge. In practice, you write code that checks that preconditions are met before utilizing a function.
The assert function:

inline void assert(int boolean, char *boolstr, char *file, int line) {
  if (!boolean) {
    fprintf(stderr,"Assertion %s failed in line %d of file %s\n",
            boolstr, line, file);
    exit(1); /* exit(0) means correct execution! */
  }
}

Practice Page 22
The assert macro

#define ASSERT(bool)  assert((bool),#bool,__FILE__,__LINE__)

bool: an arbitrary integer expression.

#bool: the *stringification* of bool: makes it a string rather than an integer.

__FILE__: predefined preprocessor constant: the file being compiled.

__LINE__: predefined preprocessor constant: the current line number.

In other words:

    assert(foo>0)

In file "foo.c", line 42, is translated into

    assert((foo>0), "foo>0", "foo.c", 42)
Using assert.h
Determine preconditions for code to work. Determine what must be done to test them in the code. Write these test cases as ASSERT() statements.

Aside: case conventions for C, C++
Lowercase: regular identifiers.
Capitalized: typedefs and classes
ALL CAPS: #defines
Some preconditions can be guarded.
Some preconditions cannot be checked.

struct elem *p=head;
while (p) { p=p->next; }
// if I get here, I'm fine, but if I don't, I don't know why

Next technique: memoize the list, and the guard gets very complex.

Truly sneaky:
ASSERT(((int)head)%(sizeof(struct elem))==0);
In other words, the head pointer points to a struct aligned according to ANSI alignment standards.
Some best practices for assert.h

Tuesday, October 09, 2012
5:26 PM

Code the preconditions you can.
If speed is an issue, then place assertions inside:
#define DEBUG
...
#undef /* DEBUG */
Be aware of the preconditions you can't check.