Introduction to Compilers and Language Design

Hardcover ISBN: 978-0-359-13804-3
Paperback ISBN: 978-0-359-14283-5
First edition.

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at `http://compilerbook.org`

Revision Date: May 2, 2019

Appendix C – Coding Conventions

C has been the language of choice for implementing low level systems like compilers, operating systems, and drivers since the 1980s. However, it is fair to say that C does not enforce a wide variety of good programming practices, in comparison to other languages. To write solid C, you need to exercise a high degree of self-discipline. [1]

For many students, a compilers course in college is the first place where you are asked to create a good sized piece of software, refining it through several development cycles until the final project is reached. This is a good opportunity for you to pick up some good habits that will make you more productive.

To that end, here are the coding conventions that I ask my students to observe when writing C code. Each of these recommendations requires a little more work up front, but will save you headaches in the long run.

Use a version control system. There are a variety of nice open source systems for keeping track of your source code. Today, Git, Mercurial, and Subversion are quite popular, and I'm sure next year will bring a new one. Pick one, learn the basic features, and shape your code gradually by making small commits.[2]

Go from working to working. Never leave your code in a broken state. Begin by checking in the simplest possible sketch of your program that compiles and works, even if it only prints out "hello world". Then, add to the program in a minor way, make sure that it compiles and runs, and check it in. [3]

Eliminate dead code. Students often pick up the habit of commenting out one bit of code while they attempt to change it and test it. While this is

[1] Why not use C++ to address some of these disciplines? Although C++ is a common part of many computer science curricula, I generally discourage the use of C++ by students. Although it has many features that are attractive at first glance, they are not powerful enough to allow you to dispense with the basic C mechanisms. (For example, even if you use the C++ string class, you still need to understand basic character arrays and pointers.) Further, the language is so complex that very few people really understand the complete set of features and how they interact. If you stick with C, what you see is what you get.

[2] Some people like to spend endless hours arguing about the proper way to use arcane features of these tools. Don't be one of those people: learn the basic operations and spend your mental energy on your code instead.

[3] This advice is often attributed as one of Jim Gray's "Laws of Data Engineering" in slide presentations, but I haven't been able to find an authoratative reference.

a reasonable tactic to use for a quick test, don't allow this dead code to pile up in your program, otherwise your source code will quickly become incomprehensible. Remove unused code, data, comments, files, or anything else that is unnecessary to the program, so that you can clearly see what it does now. Trust your version control system to allow you to go back to a previously working version, if needed.

Use tools to handle indenting. Don't waste your time arguing about indenting style; find a tool that does it for you automatically, and then forget about it. Your editor probably has a mode to indent automatically. If not, use the standard Unix tool `indent`.

Name things consistently. In this book, you will see that every function consists of a noun and a verb: `expr_typecheck`, `decl_codegen`, etc. Each one is used consistently: `expr` is always used for expressions, `codegen` is always used for code generation. Every function dealing with expressions is in the `expr` module. It may be tempting to take shortcuts or make abbreviations in the heat of battle, but this will come back to bite you. Do it right the first time.

Put only the interface in a header file. In C, a header file (like `expr.h`) is used to describe the elements needed to call a function: function prototypes and the types and constants necessary to invoke those functions. If a function is only used within one module, it should *not* be mentioned in the header file, because nobody outside the module needs that information.

Put only the implementation in a source file. In C, a source file (like `expr.c`) is used to provide the definitions of functions. In the source file, you should include the correpsonding header (`expr.h`) so that the compiler can check that your function definitions match the prototypes. Any function or variable that is private to the module should be declared `static`.

Be lazy and recursive. Many language data structures are hierarchically nested. When designing an algorithm, take note of the nested data structures, and pass responsibility to other functions, even if you haven't written them yet. This technique generally results in code that is simple, compact, and readable. For example, to print out a variable declaration, break it down into printing the name, then the type, then the value, with some punctuation in between:

```
printf("%s:\n",d->name);
type_print(d->type);
printf(" = ");
expr_print(d->value);
printf(" ;\n");
```

Then proceed to writing `type_print` and `expr_print`, if you haven't done them already.

Use a Makefile to build everything automatically. Learn how to write a Makefile, if you haven't already. The basic syntax of Make is very simple.

The following rule says that `expr.o` depends upon `expr.c` and `expr.h`, and can be built by running the command `gcc`:

```
expr.o: expr.c expr.h
    gcc expr.c -c -o expr.o -Wall
```

There are many variations of Make that include wildcards, and pattern substitution, and all manner of other things that can be confusing to the uninitiated. Just start by writing plain old rules whose meaning is clear.

Null pointers are your friends. When designing a data structure, use null pointers to indicate when nothing is present. You cannot derefence a null pointer, of course, and so you must check before using it. This can lead to code cluttered with null checks everywhere, like this:

```
void expr_codegen( struct expr *e, FILE *output )
{
    if(e->left) expr_codegen(e->left,output);
    if(e->right) expr_codegen(e->right,output);

    . . .
}
```

You can eliminate many of them by simply placing the check at the beginning of the function, and programming in a recursive style:

```
void expr_codegen( struct expr *e, FILE *output )
{
    if(!e) return;

    expr_codegen(e->left,output);
    expr_codegen(e->right,output);

    . . .
}
```

Automate regression testing. A compiler has to handle a large number of details, and it is all to easy for you to accidentally introduce a new bug when attempting to fix an old one. To handle this, create a simple test suite that consists of a set of sample programs, some correct and some incorrect. Write a little script that invokes your compiler on each sample program, and makes sure that it succeeds for the good tests, and fails on the bad tests. Make it a part of your Makefile, so that every time you touch the code, the tests are run, and you will know if things are still working.

Index