Effective Code Development Practices (Chapter 5)

March 18, 2025

The following content is a highlight of Chapter 5 of your e-textbook (with my own notes), which deals with effective code development practices. For additional info on any of these topics, you can refer to the textbook, or let me know.

1 typedef & Struct Review

`typedef` allows you to create an alias for a type. Especially useful when used together with structs. Consider the following:

```c
#include <stdio.h>

// Using typedef with struct
typedef struct {
    int x;
    int y;
} Point;

Point createPoint(int x, int y) {
    Point p;
    p.x = x;
    p.y = y;
    return p;
}

// Accessing struct data by reference and by value
// 1) Pass by reference: access the x and y values using p arrow (->)
void movePoint(Point* p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

// 2) Pass by value: access x and y values using p dot (.)
void printPoint(Point p) {
    printf("Point at (%d, %d)\n", p.x, p.y);
}
```

```c
int main() {
    Point p1 = createPoint(1, 5);
    printf("Initial position:\n");
    printPoint(p1);
    movePoint(&p1, 5, -3);
    printf("After moving:\n");
    printPoint(p1);
    return 0;
}
```

Allows us to use 'Point' to refer to a struct that contains x and y information. Without typedef, we would need to include 'struct Point' everywhere instead.

1.1 #define vs. typedef

Why not use macros/`#define` for everything instead? Consider:

```c
typedef char* String;
String s1, s2;  // s1 and s2 are both char* (good)

#define String char*
String s1, s2;  // s1 is char*, but s2 is just char (bad)
```

Remember, macros take effect before the compiler even executes (pre-processing stage), replacing String with char* as a simple textual substitution. Sometimes the behaviour can be unexpected. typedef is actually processed by the compiler, and will correctly replace the types in the example above.

In general:

- use typedef when substituting existing types with a new name for simplicity
- use #define when substituting simple constants or expressions

2 Memory Leaks and Detection

As we previously discussed, memory which you have allocated on the heap (using malloc, calloc, realloc) must be freed. If not, you can have memory leaks, which can reduce performance in the long term.

2.1 valgrind

```
# install through your package manager if you don't have it
sudo apt install valgrind
```

valgrind is a debugging/profiling tool that can detect memory leaks in your code. It's very simple to use:

```
# compile normally
gcc myprog.c -o myprog

# use valgrind to check for memory issues on compiled program
valgrind --leak-check=full ./myprog
```

Try it yourself on some sample code (or maybe even your assignment 3). It tells you the total number of allocations, de-allocations, total bytes allocated on the heap, and if all heap blocks were freed or not upon exiting your program.

3 Compiler Optimization

You can pass the `O#` flag to gcc to optimize your code. These come at the cost of higher compilation times, or with very aggressive options, you can lose some precision too. The choice of optimization depends entirely on the goals of your project.

- -O0: No optimization. Default, and best used for debugging.
- -O1: Basic optimization. We can expect slight execution time improvements.
- -O2: Moderate optimization.
- -O3: Max optimization for speed. Likely increases code size.
- -Ofast: Like -O3 but may go against code standards.

Usage:

```
# Choose moderate opimization as a decent middle ground. Gain some
   performance at cost of compilation times
gcc myprog.c -o myprog -O2
```

Refer to Section 5.3 in the textbook to see a longer example where you can measure CPU times to see the differences between optimizations.

4 Analyzing Performance

We analyze the performance of our programs through what is called **profiling**. Here we'll go over a few profiling techniques with C:

4.1 gprof

```
# Include the -pg flag during compilation
gcc myprog.c -o myprog -pg

# Execute your program
```

```
./myprog

# Now you'll see a file called 'gmon.out'
# To store results into a text file
gprof myprog gmon.out > analysis.txt

# Now open the text file to view results
```

It will tell you the total and relative time spent within each function of your program, which is helpful for analyzing performance.

4.2 Code Coverage (gcov)

Code coverage allows you to gain confidence that you are testing your program thoroughly. Once you have designed a set of test cases, you can compile/run them alongside `gcov` to ensure things like statement coverage. If you see that some lines are never executed by any test cases, then you can update your test cases to achieve better coverage.

```
# Verify it's installed (it should come with gcc)
gcov --version

# If not, install through your package manager
sudo apt install build-essential
```

Usage:

```
# Include the flags for coverage
gcc myprog.c -o myprog -fprofile-arcs -ftest-coverage

# Execute
./myprog

# Generate coverage data with gcov on source file (.c)
gcov myprog.c
```

This should generate a file called `myprog.c.gcov`, which you can view to see the report for statement coverage across all source files.

5 Documentation

5.1 Doxygen

Doxygen is a tool for producing code documentation using special comment blocks within your source code. If you're interested, there are some videos provided in Section 5.6 of your e-textbook.