ECE 222 System Programming Concepts
Lecture notes – Organizing Code

Organizing code is not easy. Programmers probably spend as much time organizing code as they do actually writing new code. By organizing, we refer to steps taken to modularize it, to track changes to it, and to make it readable and understandable.

There are several methods to organize code. We are going to examine several of them in some detail, especially with how the C language supports them.

1) Break up code into multiple functions.

Writing a program using multiple functions is the most classic approach to making a program modular. Modular code is desirable for two reasons. First, it breaks a coding problem into pieces, where each piece can be tackled independently (divide and conquer). Second, it allows code pieces to be reused in future programming tasks.

One of the drawbacks to C is that it only explicitly supports modularity in functions. Sometimes, it is desirable to consider data or other abstractions in modules. C provides a little support for such modularity; for example, a structure can be used to create a new modular data type. However, other languages (notably C++ and Java) have pushed the modularity principle further, providing additional language constructs. These are outside the scope of this class.

2) Break up functions into multiple files.

As the number of functions in a program increases, it is convenient to break them up into multiple files. It makes it easier to edit a file, because there are fewer lines of source code to scroll through to find the function of interest. It also makes program building more efficient. After changing code within a single function, only the file containing that code needs to be recompiled. The other files (assuming the object code files are kept around between program builds) only need to be linked to create a new executable.

When breaking a program into multiple functions and files, one must be aware of how it affects variables. A variable has four parts:

- data type: int, float, char, double. These tell us how many bytes of storage we have for the variable, and what bit model is used to represent values.

- modifiers: signed, unsigned, short, long. These change how many bytes we have, and how the bits may be used.

- qualifiers: const, volatile. Information to the compiler as to how the variable will be used. We will not address these any further here.

- storage class: auto, static, extern. These affect the scope of the variable, or the visibility of the variable throughout the program.

Using multiple functions and files brings the last part, variable scope, into play. Let's look at each of the three cases of variable scope:

a) auto: visible within function. This is the default storage class for variables inside a function. The variable is only visible inside the function, and when the function ends, the variable (value) disappears.

For example:

```
int Function1(float e)
{
int             x;
auto int        y;
…
}
```

The variables x, y and e all have the same variable scope: auto.

b) static: has two meanings. Unfortunately, the static keyword is used to signify two different scopes:

   i. visible to all functions within file. This is used to make a variable global to all functions within a file, but only within that file. Any function within the file sees the same thing; other functions outside that file cannot see the variable.

For example:

```
static int      a;
int Function1(float e)
{
…
}
int Function2(float g)
{
…
}
```

The variable a has "file static" scope, or is a "file global".

   ii. retains value between function calls. If static scope is used inside a function, then the variable becomes "global" between multiple calls to the same function, retaining its value.

For example:

```
int Function2(float g)
{
static int        c;
…
}
```

The variable c has “function static” scope, or is a “recurring” variable. Between calls, it will retain the same value. This does not mean that the variable can be accessed outside the function, only that it “stays alive” between calls to the same function.

c) extern: visible across all object files. This is used to make a global variable visible across multiple files linked into the same program. Remember that each source code file is compiled independently. It is during linking that the scope of this type of variable allows it to be “matched up” across multiple object files. This is the default for variables declared outside functions.

For example:

[show var-scope ext1.c and ext2.c example]

3) Comments.

Comments are annotations of the programmer, like post-it notes used to document ideas throughout a project. They are used throughout a program for a variety of reasons:

a) function – describe sub-task solved by function, function prototype (inputs and outputs to function)
b) code block – describe sub-task, usually 3-20 lines of code
c) program – program usage, history (revisions), to do list, authorship; usually found in the main source code file (main.c)
d) variable – purpose, or the values expected (e.g. range)
e) line by line – as needed on complex ideas

4) Variable names.

Variable names should usually describe the purpose of the variable. For example:

```
int ProgramRunning=1;        /* 0 => end program, 1 => continue running */
```

Sometimes the variable name describes the process it manages, as in the example above, and sometimes it describes the quantity stored, such as “int age”. Simple, brief variables

names are ok to use then the meaning is obvious. For example, loop indices often use single letters, as most programmers recognize the convention.

5) Pre-processing.

Pre-processing is not strictly a C language construct, it can be supported by any language since it happens before compiling. Text substitutions provide several mechanisms for supporting code organization:

a) readability. It is much easier to view short strings that are to be substituted by longer strings, when the meaning is still clear in the shorter string. For example, it can be convenient to read that some operation is being performed on a variable, instead of verifying all the details each time:

```
#define KAZAM(x)  (((x)*3/4)-2)/5)
```

b) scalability. It is much easier to change a constant using a text substitution:

```
#define MAX_LINES 20
```

c) portability. Text substitutions can help in compiling the same source code for different processors or operating systems. For example, an int is usually 4 bytes, but on some systems it may be two or eight bytes. One can use text substitutions to name a 4 byte entity that is replaced by the appropriate data type depending on which machine the code is compiled upon:

```
#define four_bytes int          /* use this on 32-bit architectures */
#define four_bytes long int     /* use this on 16-bit architectures (e.g. DOS) */
#define four_bytes short int    /* use this on 64-bit architectures (e.g. P5) */

main()
{
four_bytes      x,y,a;
…
}
```

In this example, the variables x, y and a will be four bytes (either an int, a long int, or a short int), regardless of the system architecture, so long as the appropriate pre-processor command is executed before compiling.

6) Typedefs.

A “typedef” is a method for providing an alias for an existing data type. It provides a new name for a data type that already has a name:

```
typedef         SomeExistingType     MyNewNameForIt;
```

For example, suppose we wanted an “int” to have a second name “frog”:

```
typedef     int     frog;
```

Now we can write code using “frog” as an alias for “int”:

```
main()
{
int     a;
frog    b;
…
}
```

Both the variables a and b are ints; frog is only an alias for int.

Why would we want to make up aliases for existing data types? Two reasons.

a) readability. Programmers get tired of using “struct x” data types. Instead of having to type “struct x” every time, you can use a typedef to give a new alias to “struct x”.

For example:

```
struct TemplateName {
        int     field1;
        float   field2;
        } ;
typedef struct TemplateName NewStructType;
```

NewStructType is not a variable name, it is now an alias for “struct TemplateName”. So we can define variables of type “struct TemplateName” in two ways now:

```
struct TemplateName s1;
NewStructType s2;
```

Both variables s1 and s2 are of the same data type, “struct TemplateName”.

Note that a typedef and a structure definition are often clumped together:

```
typedef struct TemplateName {
        int     field1;
        float   field2;
        } NewStructType;
```

This has the same net result as the example above.

b) portability. Typedefs are often used to achieve the same portability goals as described for pre-processing, above.

For example:

```
typedef int         Int32;  /* 4 byte variable */
typedef short int   Int16;  /* 2 byte variable */
```

Now we write code using the data type names “Int32” and “Int16”, and change only the typedef lines if we move the code to another system architecture.

In summary, there are many tools in C that help a programmer organize code. When used properly, they can be a great help. When used improperly, or when not well understood, they can be a great pain. I hate nothing more than seeing code riddled with typedefs when the code is absolutely never going to be ported to any other system. I hate seeing code broken up into multiple functions when the entire program is less than 20 lines. I hate seeing files containing 1 function. I hate seeing multiple levels of pre-processing text substitutions that do not provide any real flexibility. These tools do not exist to amuse us, they exist to help us organize code. They should be used, but they should be used with care and respect to the original goals of organizing code.

A related discussion concerns abstraction. Abstraction refers to the level of detail required by a programmer to implement an idea in code. The more abstract a language construct, the less a programmer needs to understand how the machine actually implements the idea. The more concrete a language construct, the more a programmer needs to understand how the machine implements the idea. The C language is not very abstract. To be good at programming in C, one must understand the details just above the chip level. Other languages try to provide constructs to support more abstract programming, for example removing the need to understand addresses, system calls, and memory management. Mankind’s goal is to one day be able to program a computer without knowing how to program at all. For example, in the popular TV show Star Trek, we see Mr. Spock ask the computer for a data analysis, in spoken English, and the computer writes the program and does all the work. We’re a long way from that of course, but useful abstraction constructs continue to be invented and incorporated into new languages. However, there will almost certainly always be a need for system programmers, the experts at the low level who understand how programming relates to the hardware.