ECE264 C Pocket Guide
Version 1.0

Aaron Michaux
`aaron@pageofswords.net`

June/8/2014

C is quirky, flawed, and an enormous success.
— Dennis Ritchie

1 Introduction

Although simple and powerful, C demands the utmost care to use correctly. Bugs can be very subtle – sometimes almost invisible – and it is easy to waste large amounts of time if you do not know what you are doing. To save time and frustration, certain information simply has to be *inside your head* – there is no getting around it. For that reason, everything in this document will be tested in the first midterm, which is closed-book. Cited material will not be tested.

2 How to Learn Code

Learning how to program is mostly a question of practice. It is important to learn and understand some programming theory; however, practice is the most important thing. In this, learning to program is very similar to learning a music instrument, or a sport such as basketball. Listening to music, or watching people make free throws does not help someone master these skills. For this reason, this course emphasizes assignments.

2.1 Practice vs. Copy-Paste Evil

Many weaker (and some stronger) students spend a lot of time looking up code on the internet, and copying and pasting snippets into their assignments. Another common problem is when students shuffle around lines of code in the hope that it will somehow work. This is a horribly inefficient method to go about writing code, and does not count as practice. By analogy with music, this is like someone playing through a piece of music for fun, and not working on the problems in their technique. By analogy with basketball, this is someone who "practices" by playing games with their friends, but never self-consciously improves their dribbling, or practices shooting drills.

To practice, to learn, *one must solve problems*. That means more time thinking and less time fiddling with code. This is important even for seasoned programmers. Much time is saved by thinking about problems, and by learning how code executes. A programmer only develops insofar as they practice.

2.2 Solving Problems

With 30 years plus experience programming, I believe that the best way to write code is with pen and paper. Code usually modifies the contents of memory, which can be difficult to keep track of. Making diagrams is a great way to visualize how one believes that the code should behave. Drawing diagrams helps encode the algorithm mentally, and also identify possible *corner cases* that must be addressed.[1]

For example, in an early assignment, students must implement the C library "strstr" function. This can be done with a pair of nested loops. (That is a big hint.) However, to get the code correct, even with significant experience, I found myself drawing a diagram of the two arrays ("needle" and "haystack"), and specifically how the nested iteration should work.

Once the solution is understood, the programmer *then* converts it to code. A written diagram is invaluable in ensuring that nothing is forgotten, and that no new problems are introduced when handling the process of synthesizing actual code.

2.3 Reading the Manual

A thought can only be formulated out of what already exists in the mind. For this reason, it is important that students have ready cognitive access to how their programming language functions. Without this knowledge, it is literally impossible to formulate thoughts about what is happening inside a computer program.

Once, during a job interview, we asked a programmer called David if he had had experience with Javascript. David, in his 50s, had worked as a unix system administrator for many

[1]A corner case, aka *boundary case*, is a problem that can occur infrequently in code, but nonetheless must be addressed.

years, and had also contributed some code to the linux kernel. (This implies a very advanced understanding of software.) However, David had had no experience with Javascript, which was a key requirement for the job. David replied, “I know how software is meant to work, but I need to read the Javascript manual, and practice, so that it is all cached up here,” at which point he tapped his temple. We gave David the job, and the first thing he did was read the manual, and learn how Javascript in particular in meant to behave. David proved to be an excellent employee and a first rate Javascript programmer.

This is how experienced programmers approach a job. Instead of fiddling, they sit down and learn – quite systematically – how their tools work.

David would have benefited from life-long experience when reading the manual, and mastered the material easily. However, manuals can be confusing to novice programmers. The key difference is two-fold: practice, and a belief that you can do it. Start from where you are, ask questions, and try to understand everything the manual is putting forward. Eventually, with experience, programmers will know what to look for and what to skim. This comes from a deep knowledge of how computer programs execute.

The same applies to error messages, and other outputs of programs such as gcc, gdb, and valgrind. At first these messages will look confusing – they certainly did to me. With experience, and patience, it becomes apparent that the error messages are carefully crafted, and in general, rather easy to follow.

2.4 The Forgetting Curve and Spaced Repetition

When David talked about “caching” Javascript in his mind, he meant memorizing the language fundamentals and *programming idioms*.[2] It is easy to overlook just how immense our memories are, since people in general fuss about how much they forget. There has been a lot of empirical research into memory, and below are some time tested, and scientifically validated findings:

1. *Lack of sleep* is implicated in poor long-term memory performance. Want to remember your life? Then sleep. If students stay up late studying, then they may retain information for a few days, but this is a horribly inefficient and time-consuming method to retain information long-term, since the brain forms fewer long-term memories.

2. Take advantage of the *spacing effect*, whereby people easily remember and learn items studied a few times spaced over a long period of time, rather than cramming material in a short period of time. One reason for multiple exams covering the same material,

[2]A programming idiom is a “method of doing something” that is commonly used but not built into the language itself. One must learn a language’s idioms in order to master it.

is that students will be motivated to revisit material, which improves the chances that they will benefit from courses. Cramming may have short-term benefits, but overall, it is a waste of time.

3. Research in memory encoding shows that material is remembered better when it is *processed deeply*. That means solving problems, as opposed to shuffling bits of code around, or copying and pasting from other sources. A key difference between weak and strong students in any course, is that strong students go out of their way to process course material deeply, for example, by mastering homework questions. For this reason, solving homework assignments yourself is the best way to study for exams. Teamwork is strongly encouraged in this course, but only insofar as students develop their understanding by talking through problems with their peers. It is also helpful to talk through problems with the TAs. Copying others work is self-defeating, and may be penalized for academic dishonesty.

2.5 Writing Code Quickly

Industry research [2] on software professionals demonstrates that "excellent" programmers write code ten times faster than "poor" programmers. That means that if a poor programmer earns $40k per year, a fair wage for an excellent programmer would be $400k per year. (The market is not that efficient, see citation [3].)

The situation is a good deal worse than this, because the bottom 50% of programmers inject most of the bugs into software products. On average, it costs about ten times as much to fix a bug after it has been introduced into the code, than it does to write the code correctly the first time around. [2]

That means that, if a student writes an assignment in 6 hours, and then spends 20 hours in the lab debugging it, *they have wasted 18 hours of their life.* Instead, if the student tests and fixes bugs as they write the code, then they would have been done (on average) in 8 hours.

This brings us to the golden rule of programming effectively: *do not write more code until you are absolutely sure that the previous code is working 100%.* Every good programmer does this. There is no way around it. This is the one key difference between good and bad programmers. Make it a habit, and anybody can be a fast and accurate programmer.

2.6 Writing Excellent Code

While good programmers test as they code, excellent programmers always have a deep understanding of computer programming theory and program design. Beyond testing code, the best way for anybody to improve their coding skill is to read high-quality code, and also to work closely with someone who genuinely knows what they are doing. (Be aware that many people over-estimate their programming skill.)

Excellent code is: simple, clear, and correct.

1. **Simple**. Make simplicity an art form. The human mind can only manage so much complexity, so it is straight-forward that the best programmers create their works of art by keeping code as simple as possible. It doesn't matter how much complexity an individual can manage, making things simple means they can manage more. This means striving for the most straight-forward and concise logic that does the job. Make simplicity an art-form.

2. **Clear**. Remove unnecessary comments, indent correctly, organize each source file logically, and give variables meaningful names. Comments should not be too obvious, and should guide the eye of the reader. Clear code is easy to read.

3. **Correct**. Clearly state the *preconditions* (see below) of your functions, and make sure your code always works within those conditions. Make sure your code really addresses what was specified, and that corner cases have been found and addressed. *This saves time and money.*

Fast is not on the list. Good code is fast; however, good programmers do not bother making code faster until it is already simple, clear, and correct. Even then, code cannot be optimized without using a *profiler* (i.e., a program that measures the execution speed of each part of an executable). This is because human beings are notoriously poor at recognizing the performance bottle-necks in their code. Optimization also requires extensive knowledge of data structures and algorithms, and sometimes specialized knowledge of the executing platform. Good programmers generally do not bother with optimization unless the job specifically calls for it. Fast is not on the list.

2.7 Undefined Behavior

Assumptions are always made whenever a function is written, in any computer language. Programming gets extremely complex surprisingly quickly, and for this reason, good programmers always specify what their assumptions are. These assumptions are often called "preconditions". If the preconditions are met, then the function will execute perfectly. (Because the programmer tested it, right?) If the preconditions are not met, then something will happen, but no guarantees are made. This is called "undefined behavior", and is always absent from good computer programs. Thus, it is always a bug to call a function without meeting its preconditions, even if the function seems to otherwise work.

A precondition of the "strlen(const char * str)" function is that "str" is a pointer to a valid

C string. The behavior is undefined if "str" is a pointer to some other type, an invalid pointer, or a NULL pointer. It is up to the function caller to ensure that "str" is a C string, and the "strlen(...)" function itself is under no obligation to check this. Thus in most implementations, calling "strlen(NULL)" will cause a segfault.

When stating the preconditions of a function, it is important to be both clear and concise. Nobody wants to read long paragraphs when a simple sentence will do.

3 Compiling Programs

C programs are compiled in 4 stages: *preprocessing*, *compilation*, *assembly*, and *linking*. See: [1].

1. Each *source file* (".c" file) is referred to as a *compilation unit.* (Sometimes also *translation unit.*) Each compilation unit is treated separately in a three-step process as follows:

   (a) **Preprocessing** re-writes a ".c" file by removing all comments, and by executing preprocessing commands which always begin with a "#" symbol.

   (b) **Compilation** parses each preprocessed ".c" file from top to bottom, and converts it to assembly instructions.

   (c) **Assembly** takes the assembly instructions and produces machine code. The output is an *object file* (".o" file). Each object file has a "table of contents" that lists the location of things such as constants, global variables, and the machine-code for each function in the source file. Often a function needs to call some function in another compilation unit and are thus only partly assembled, since compilation units do not have knowledge of other compilation units. The object file explains exactly which additional functions are required to "complete" each function. Object files can also contain debug information.

2. **Linking** takes a set of object files, and "links" functions that call each other from different compilation units. When compiling an *executable* (i.e., computer program), the linker looks for the *main function*, which becomes the executable's *entry point* (i.e., where the program starts). If the linker cannot find a function or global variable, it will give an *undefined reference* error.

```
> # Print preprocessor output. Output is preprocessed C code.
> gcc -E main.c
>
> # Preprocess, compile, but do not assemble. Output is assembly.
> gcc -S main.c
```

```
>
> # Preprocess, compile and assemble. Output is an object file.
> gcc -c main.c
>
> # Run main.o through the linker to produce an executable
> gcc main.o -o a.out
```

4 The Preprocesser

The preprocessor is a very simple language that is entirely separate from C. It modifies a source file by executing *preprocessor directives*. These directives always begin with a "#" character. Only the most important and commonly used portions of the preprocessor language are covered in this course.

4.1 The Include Directive

This directive literally copies the contents of a file, and pastes it into the source file. This is usually done so that one compilation unit can "know about" the functions in another compilation unit. (Explained further below.)

```
1 // Include the system header-file
2 #include <stdio.h>
3
4 // Include a programmers header-file that is the same directory.
5 #include "answer01.h"
```

There are two forms of this directive, one with angle brackets (line 2 above), and one with double-quotation marks (line 5 above).

The method with angle brackets will first searches a pre-configured system search path to find the desired header. On Ubuntu Linux, the "stdio.h" header file is located in the "/usr/include" directory. This is the precise file that is copied and pasted. If the file is not found in the pre-configured search path, then the preprocessor will look for the file relative to the source file.

The method with double-quotes will search for the file relative to the directory containing the source file. If this file is not found, then the preprocessor will search for the file in the pre-configured system search path.

To understand which file is selected, a programmer must understand relative and absolute paths.

4.1.1 Relative and Absolute Paths

This is basic computer knowledge required for programming and non-programming tasks on almost all computer systems, including Unix and Windows.

Absolute paths are quite simple to understand, since they specify the full path to the file. They look a little different on Unix and Windows systems. On Unix, absolute paths always begin with a "/" character. On Windows, absolute paths always begin with a drive letter like so: "C:\".

Relative paths look for a file relative to the *working directory*. Every computer program (including Bash) is associated with precisely one current working directory.

```
const char * absolute_path = "/usr/include/stdio.h";
const char * relative_path = "some-file.text"; // file is in working ...
    directory
const char * another_rel_path = "../images/beachball.png"; // Go up one ...
    directory, open the images directory, and find the beachball.png file

// This works on ubuntu linux, but will fail on windows and other platforms.
#include "/usr/include/stdio.h"

// This should work most of the time, but will fail spectacularly if a
// file with the same name happens to be in the same directory as the
// source file.
#include "stdio.h"
```

4.1.2 Include Guards

Copying the contents of one file into another can go haywire when file A includes file B and vice-versa, creating an endless loop. This situation is not that uncommon in mid-sized projects, which is why the preprocessor provides an *include guard*.

```
// Place this at the top of any header file.
#pragma once
```

There are many different pragma directives, and they are meant to provide additional information to the compiler beyond what is contained in the source code. In this case, "#pragma once" instructs the preprocessor to only ever include this file once in a given compilation unit. Thus if header A includes B which includes A, then the preprocessor will not bother with the second attempt to include A.

The “#pragma once” directive is non-standard, and sometimes frowned upon. It is, however, supported by all major C compilers. An alternative include-guard method is described below.

4.1.3 #ifdef, #ifndef, #define, and #endif

These directives are usually used as an include guard blessed by C purists, but is more complex than “#pragma once”.

The “#ifndef” (if-not-defined) and “#endif” directives are used to include text unless the specified *symbol* is defined. The “#define” directive is used to define symbols. (It also defines macros and constants, see further below.)

```
1  #ifndef GEOMETRY_H_INCLUDED
2  #define GEOMETRY_H_INCLUDED
3
4  // Geometry declarations require knowledge of vectors and planes
5  #include "vectors.h"
6  #include "planes.h"
7
8  // Some declarations
9  double distance_to_plane(double X[3], struct Plane plane);
10 // ... etc.
11
12 #endif
```

All of the code above is skipped if the symbol “GEOMETRY_H_INCLUDED” is defined. This symbol is not defined the first time the file is included, so the define directive on line 2 is processed. This immediately defines the “GEOMETRY_H_INCLUDED” symbol preventing future (and circular) includes. The “#endif” at the end is required to match with the “#ifndef” directive on line 1.

The “#ifdef” (if-defined) directive is not used in include guards; however, it is introduced here because of its similarity to “#ifndef”. “#ifdef” will exclude a region of text unless the symbol is defined, and is thus the logical opposite of “#ifndef”.

```
#ifdef WIN32
// Include windows header when building on windows
#include <windows.h>
#endif
```

This can be used to create testing code that is compiled only when a particular symbol is defined. For example:

```
#ifdef MY_MAIN
int main(int argc, char * * argv) {
  const char * needle = "Spain";
  const char * haystack = "The Vandals were first heard of in Poland, ...
      and moved through the Roman empire to make settlements in Spain ...
      and Northern Africa.";
  printf("Test 1: %s", my_strstr(needle, haystack));
  return EXIT_SUCCESS;
}
#endif
```

This main function is only compiled into the code when the symbol "MY_MAIN" is defined. It is possible to tell gcc to define a symbol directly, like so:

```
gcc -DMY_MAIN answer01.c
```

This will be demonstrated in class.

4.1.4 Why Use Include?

In general, C projects are structured as follows:

1. For every source file, a programmer creates a single header file: a file with the same name except for a .h suffix.
2. Every function in the source file should either be "static", or its declaration should be copied into the header file.[3]
3. The header file should always use some include-guard method.

This is a tried and true C idiom.

4.2 More on #define

The "#define" directive can also associate a symbol with a value like so:

```
#define TRUE 1
#define FALSE 0

```

[3]Static functions are private to a single compilation unit, and can never be used by any other compilation unit. Static variables are global within a function. Everyone agrees that this is confusing – C is an imperfect language.

```c
int is_greater(int a, int b)
{
  if(a > b) return TRUE;
  return FALSE;
}
```

The preprocessor replaces every instance of "TRUE" with "1", and every instance of "FALSE" with "0". TRUE and FALSE are defined to aid readability, since C has no boolean type. It is good practice to always use "TRUE" and "FALSE" when dealing with boolean values.

In the example below, the "#define" directive is used to specify a constant.

```c
#define PI 3.14159265358979323846264338327950288

double degrees_to_radians(double theta)
{
  return theta * PI / 180.0;
}
```

This is another common C idiom.

4.3 Macros

The "#define" directive is also used to specify Macros which look similar to functions but are evaluated by a copy-paste approach. Macros are used heavily in C, but are not without criticism. Macros will not be covered in this course; however, students should know of their existence.

```c
#define DEG_TO_RADIANS(theta) ((theta) * PI / 180.0)

void some_fun()
{
  ...
  double phi = 135.0;
  double psi = DEG_TO_RADIANS(180.0 - phi);
  // Expands to:
  // psi = ((180.0 - phi) * PI / 180.0);
  ...
}
```

5 Memory in a C Program

When a C program executes, it is loaded into memory. That memory is divided, roughly speaking, into four regions:

1. **Code segment**, also commonly known as the *text segment* or simply *text*. This memory stores the machine instructions of the executing program's functions. This memory is usually read-only for security reasons. This prevents the program from accidentally modifying itself at run-time, which is a common method of hacking a computer program. Writing to read-only memory causes a segmentation fault, and the operating system terminates the program.

2. **Data segment** is a region of memory that contains all global and static variables. It is subdivided into a read-only section and a read-write section. In the lines of code below, the variable "greeting" is stored in the read-write section of the data segment, and can be reassigned in line 2. However, line 3 causes a segfault because "Hello" and "Farewell" are stored in read-only sections of the data segment.

```
1 static char * greeting = (char *) "Hello World!";
2 greeting = (char *) "Farewell";
3 greeting[0] = 'f'; // segfault: attempt to write to read-only memory
```

3. **Stack** memory is used to control the flow of a program's execution. When a function executes, a *stack frame* is placed onto the stack. A frame is a chunk of memory, and the layout depends on the *calling convention* used by the C compiler. In general, the stack frame contains: the function arguments, all of the function's variables, a return value, and an *instruction pointer* which is a pointer to the next piece of code to be executed once the function terminates. After a function terminates, a CPU "jump" instruction directs the CPU to the instruction pointer, and the stack frame is removed. The memory used by the now defunct stack frame remains valid, in the sense that reading and writing to it will not cause a segfault. However, accessing a defunct stack frame is an error, since this memory is almost always written over very quickly.

4. **Heap** memory is dynamically allocated memory that persists between function calls. Unlike stack memory, heap memory must be asked for, and freed manually. In this sense, it is like a library book. The program "borrows" it from the operating system, and then "returns" it once done. (Note: stack memory is automatically allocated when a stack frame is created, and automatically freed when a stack frame is removed.) In general, programmers use "malloc" to ask for heap memory.[4] The

[4]Other common mechanisms for obtaining heap memory are "calloc" and "mmap" – always use malloc in this course.

operating system is under no obligation to provide it. Memory is freed with the "free" function.

When a program executes, the operating system uses a special program called a *runtime linker* that copies the code and data segments into memory. It then performs various additional initialization steps, and then hands over control to the *C runtime*. The C runtime creates the stack and then calls the "main" function. Not every computer program has a stack; however, C programs always do.

5.1 Variables, Data, Stack and Heap

Every variable (everything that has a variable name) is stored in either the code segment, the data segment, or the stack. There are no exceptions. Heap memory is only ever accessed via pointers, but the pointer variable itself will be on the stack or in the data segment.[5] Note that a pointer can contain the address of any piece of memory. This includes anything in the stack, heap, data segment, and code segment, and also includes random and invalid addresses.

```c
#include <stdio.h>

main()
{
    int i; // on the stack
    const char * s = "Hello!"; // s is on the stack
    printf("The stack top is near %p\n", &i);
    printf("The data segment contains %p\n", s); // "Hello!" in data segment
    printf("The code segment contains %p\n", main);
    return 0;
}

// Prints something like:
The stack top is near 0x7fffdf954ad4
The data segment contains 0x400a10
The code segment contains 0x4007cd
```

5.2 Uninitialized Data

For performance reasons, C does not initialize variables. An uninitialized variable contains random data, but is often zero. This leads to pernicious errors. Often using uninitialized variables will work most of the time, but an unpredictable failure could happen at any

[5]The named variables in the code segment are always function addresses.

time. Valgrind is the best tool for detecting these errors, which show up as *invalid reads*, or sometimes *branch or jump depends on unitialized value.*

5.3 Memory Leaks

Heap memory needs to be freed, which makes it trickier than using stack memory. For this reason, as a rule of thumb, always use stack memory whenever possible. The golden rule of memory management is: *every malloc must be paired with one, and only one, free.* Failure to free memory results in a *memory leak* and is one of the most pernicious problems in computer programming. Freeing a pointer twice will crash the program immediately.

To extend the library analogy of the heap: you cannot use a book you never borrowed, you must return a book that you have borrowed, and you cannot return a book twice. (You cannot use memory until you malloc it, you must free it when done, and you cannot free it twice.)

The creators of C noticed that it is often convenient to be able to free a NULL pointer without causing problems. For this reason, the “free(...)” function will do nothing if NULL is passed. To extend the library analogy, attempting to free a NULL pointer is like going to the library to return a book, only to find out that you never had the book in the first place. So you just go home.

5.4 Segfaults

A segmentation fault occurs when a program attempts to read, write, or execute memory that it is not authorized to use. The operating system will immediately terminate the program by calling a special *signal handler* called SIGSEGV. In certain very rare circumstances, a programmer may need to stop the program crashing by intercepting this signal. Signals are not covered in this course; however, students should be aware that segfaults can be “intercepted”. In any case, the program never performs the invalid read, write or execute.

6 What Happens When a Function Is Called

The details given here will differ depending on the *calling convention* used[6]; however, they are representative. Consider the following code:

[6]A calling convention is a precise set of assembly instructions used to handle function calls.

```c
1  double poly(double x, double y)
2  {
3    double tmp1 = 2.0 * x * x;
4    double tmp2 = -4.0 * y;
5    return tmp1 + tmp2;
6  }
7
8  int main()
9  {
10    double x = 4.0;
11    double y = 1.0;
12    double p = poly(x, y);
13    return 0;
14 }
```

A *stack frame* is pushed onto the stack every time a new function is called. As previously mentioned, this stack frame is a chunk of memory that contains the memory required by the function. It can be thought of like a *struct* (described further in a later section) that is unique to every function. The "struct" for the "poly" function might look like this:

```c
struct Poly_function_stack_frame {
  machine_code * return_address; // points to the instruction after that ...
      which called the function
  double y; // a _copy_ of function argument 2
  double x; // a _copy_ of function argument 1
  double tmp1; // parameter used in the function
  double tmp2; // ditto
};
```

"poly(...)" is first called at line 12. The code for the main function is compiled into machine language, and the runtime linker loads it into memory when the program executes. Therefore, the code at line 12 has an address. The address at the "end" of this line of code is pushed on to the stack. (It becomes the "return_address".) This "return_address" is sometimes called the function's *continuation*.[7] When the function finishes at line 6, a *jump* machine instruction is executed that tells the CPU to start executing the code at the "return_address". This is how C ensures that the continuation for a function is always correct, even when it is called from multiple different locations in the code.

Another thing to note is that the stack frame contains a copy of the function arguments. Function arguments are always copied in C. Thus, the x and y mentioned on lines 10 and 11 are different (i.e., have different memory addresses) to the x and y in the stack frame. They are copied before the function starts to execute (on line 3).

[7] A continuation is the next piece of code to execute after a function is finished.

Finally, note that every variable in a function must be allocated in memory somewhere, and that place is the stack frame that is created when the function is called. That stack frame is invalid as soon as the function terminates; however, it will still be part of the executing program's address space. This means that referencing memory in a defunct stack frame will not cause a segfault. However, the next function call will quickly write over a defunct stack frame, producing highly unpredictable results.

```c
int * evil()
{
  int x = 1; // x is in "evil's" stack-frame when it executes
  return &x; // Using this address will not cause a segfault; however,
} // the memory at &x will quickly get overwritten by the next function call
```

7 Types

7.1 Integer Types

There are five basic integer types: *char*, *short*, *int*, *long*, and *long long*. These types differ primarily in the number of bytes they use to store a value. The *sizeof(...)* operator is used to determine the number of bytes on a given platform. The following is always true for every C platform:

```c
1 ≤ sizeof(char) ≤ sizeof(short); // char is usually 8-bits
2 ≤ sizeof(short) ≤ sizeof(int);  // short is usually 16-bits
2 ≤ sizeof(int) ≤ sizeof(long);   // int is usually 32-bits
4 ≤ sizeof(long) ≤ sizeof(long long); // long is usually 64-bits
8 ≤ sizeof(long long);            // long long is usually 64 bits
```

The *signed* and *unsigned* keywords can be placed before a basic integer type to force it to be signed or unsigned. A signed integer value can contain negative values. Basic integer types are signed by default, except *char* which is sometimes unsigned by default.

Sometimes it is important to select an integer type that is guaranteed to have a preset size. The header <stdint.h> contains such types. For example, "uint32_t" (read u-int-32-type) is guaranteed to be an unsigned 32-bit integer.

7.2 Floating Point Types

Each floating point type has a sign (i.e., positive or negative), an exponent, and a fraction. Thus, they conceptually represent the number $\pm fraction \times 2^{exponent}$. This means that floating point types can contain very large and very small numbers. For example, a double can contain numbers between 10^{-308} and 10^{308}. This flexibility comes at the cost of precision. Floating point mathematics often involves some rounding under the hood, and thus can have an error term which cannot be ignored. For this reason, testing equality (i.e., "==") with floating points should be avoided because it can be difficult to predict the result.

There are three floating point types: *float*, *double*, and *long double*. Their sizes can vary, but C guarantees that:

```
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double);
```

Floats and doubles are usually 32-bit and 64-bit IEEE754 format numbers. These numbers can be special values: ± infinity, and NaN (i.e., "Not a number"). NaNs result from invalid operations, such as 0.0/0.0, or $sqrt(-1)$. The <math.h> header defines two functions: *isinf(...)* and *isnan(...)* that can be used to test if a floating point number is infinity or NaN.

There is much more to learn about floating point numbers beyond this simple introduction.

7.3 Size-type, and Pointer-difference

The result of the *sizeof(...)* operator is *size_t* (read size-type). This is an unsigned integer type that is large enough to represent the size of any object in memory – usually 64 bits.

When using size_t within printf statements, it is important to either use "%zd" (unix only), or cast the size_t to an integer. For example:

```
int x;
printf("sizeof(x) = %zd\n", sizeof(x)); // works on linux
printf("sizeof(x) = %d\n", (int) sizeof(x)); // works on linux and windows
```

C also has a special type for the result of subtracting one pointer from another. This is the *ptrdiff_t* (read pointer-difference-type).

7.4 Structs

Structs (aka: structures) are compound data types that combine together a collection of arbitrary different types. For example, a "DateTime" struct could be defined as follows:

```c
struct CalendarDate {
  int  year;
  char month, day;
};
```

The struct keyword is part of the type's name. The "members" of the struct are accessed using the "." operator. See the example below:

```c
struct CalendarDate update;
update.year = 2014;
update.month = 5;
update.day = 1; // May 1st, 2014
```

The "." operator works out the offset of the variable (year, month or day) within the chunk of memory. This is a tiny bit like an array, except the number of elements are fixed, and the elements can have different sizes. Under the hood, a struct is merely a chunk of memory, and the "." operator is used to find values within it.

The compiler may put some padding bytes into a struct to optimize the memory layout and thus speed of the program. This optimization is tailored to special properties of how memory is accessed in modern CPUs. Sometimes padding a struct can break a computer program, for example, when using a struct to represent the memory layout of data in a file header. This bevahior can be controlled using the *#pragma pack(...)* preprocessor directive. See the example below.

```c
#include <stdio.h>

struct Date {
  int  year;
  char month, day;
};

#pragma pack(1)
struct Date2 {
  int  year;
  char month, day;
};

```

```c
int main()
{
   // sizeof Date is 8 bytes.
   printf("sizeof Date is %zd bytes.\n", sizeof(struct Date));

   // sizeof Date2 is 6 bytes.
   printf("sizeof Date2 is %zd bytes.\n", sizeof(struct Date2));

   return 0;
}
```

The bit-pattern of a struct is copied whenever it is passed to a function, or returned from a function. The bit-pattern is also copied whenever the assignment operator (i.e., "=") is used.

Structs are essential for creating complex data structures.

7.5 Pointers

Imagine a long row of houses on a very long street. Each house has a number. The number of the house is the *address* of the house. Note that the address of the house is distinct from the house itself. A *pointer* is a variable that contains the address of a piece of memory. A pointer is analogous to the house-number on the very long street. The memory the pointer points to is analogous to the house itself.

```c
int x; // some stack memory
int * ptr = &x; // ptr contains the address of x. (x is the house itself.)
```

That may be fairly straight-forward; however, it takes practice to understand this fully. This is because – to extend the house example – a pointer is also a house itself! Why so? Because a pointer contains an *address* which is merely a number that signifies a piece of memory. That number must be stored somewhere in memory, which means that every pointer itself has an address. That means that C supports pointers to pointers (to pointers, etc.). This is a very important feature that students must master in this course.

7.5.1 A Pointer, and What It Points To

In the above example we declared a pointer "int * ptr". This means that "ptr" points to an int, which is usually 4 bytes of memory. When "ptr" is *dereferenced*, we are asking the compiler to access 4 bytes of memory, and interpret the result as an "int". Thus, the fundamental type of any pointer is “pointer”, and the rest (e.g., the “int” part of the declaration in “int * ptr”) is only required for dereferencing the pointer.

```c
int y = 1;
int * ptr = &y;
printf("y = %d\n", *ptr); // y = 1
```

Note that it is an error to dereference a pointer to void.

```c
void * ptr = malloc(100);
*ptr = 0; // What is a "void"? What could it even mean?
```

All pointers have the same size on a given platform. A 64-bit operating system always uses 8-byte pointers. A 32-bit operating system always uses 4-byte pointers. Embedded systems and old operating systems such as DOS often use 16-bit or even 8-bit addresses. A 16-bit address can be stored in a 2-byte pointer, and such a platform can access 64k of memory. (i.e., 2^{16}.) The size of a pointer has nothing to do with the size of what is pointed to. They are different things. All pointers have the same size.

```c
char * p1; // Points to "void", cannot be dereferenced.
int * p2;
short * * p3;

printf("sizeof(p1) = %zd, sizeof(*p1) = %zd\n", sizeof(p1), sizeof(*p1));
printf("sizeof(p2) = %zd, sizeof(*p2) = %zd\n", sizeof(p2), sizeof(*p2));
printf("sizeof(p3) = %zd, sizeof(*p3) = %zd\n", sizeof(p3), sizeof(*p3));
printf("                   sizeof(**p3) = %zd\n", sizeof(**p3));

// Prints:
// sizeof(p1) = 8, sizeof(*p1) = 1
// sizeof(p2) = 8, sizeof(*p2) = 4
// sizeof(p3) = 8, sizeof(*p3) = 8
//                 sizeof(**p3) = 2
```

7.5.2 Pointers and Structs

Structs are often allocated on the heap, and referenced through pointers. C has special “− >” operator for this common case.

```c
struct CalendarDate {
  int year;
  char month, day;
};

...

void init_date(struct CalendarDate * date)
{
  // date is a pointer to struct CalendarDate, thus date must be
  // dereferenced in order to use the "." operator.
  (*date).month = 1;

  // C provides a short-cut for this common operator.
  date->day = 1; // This is precisely the same as (*date).day = 1
  date->year = 1970;
}
```

The "– >" operator can only be used on pointers to structs, and it must dereference the pointer. Thus the pointer must be valid. If the pointer is NULL, then the program will segfault.

7.6 Arrays

Any type can be made into an array, which is stored in a continuous piece of memory. Constant and global arrays are stored in the data segment. When a fixed-sized array is declared inside a function, it is stored on the stack. Dynamically sized arrays are stored on the heap.

The array variable name is always a pointer to the first element of the array. Thus arrays and pointers are often interchangeable. In particular, note that given an array "arr",

```c
arr == &arr[0]; // this is always true
arr + i == &arr[i]; // this is always true
*(arr + i) == arr[i]; // this is always true
```

Note: arrays are not exactly the same as pointers.

```c
int apples[10]; // apples is an array of 10 ints -- memory is allocated ...
    but not initialized
int * fruit; // fruit is a pointer to int -- currently pointing to some ...
    random bit of memory
fruit = apples; // fruit is now pointing at the first element of apples
apples = fruit; // Error: incompatible types when assigning to type int[10]
```

7.6.1 Strings in C

Unlike most languages, C does not have a special data type for strings. Instead, a string is an array of characters that ends in a null terminator. (i.e., '\0'.) Thus, the string "Hello" has five characters, and is stored in an array of length six *chars*. Students often forget about the null terminator, which results in a memory error. Depending on the circumstance, valgrind will often report the error as "invalid read of size 1", or "invalid write of size 1".

7.6.2 String Example

In the example below, the variables "s1", "s2", and "s3" are all stored on the stack. The constant string "Hello" is stored in the data segment. Thus, the contents of the variable "s1" points to somewhere in the data segment. "s2" is entirely on the stack, which means that "&s2 == s2". The six characters: 'W', 'o', 'r', 'l', 'd', and '\0', are all stored on the stack, starting at that address. The contents of "s3" is the result of a call to *malloc*, and is therefore on the heap.

```
int some_fun()
{
    const char * s1 = "Hello"; // Constant in data segment
    char s2[] = { 'W', 'o', 'r', 'l', 'd', '\0' }; // on the stack
    char * s3 = malloc(2 * sizeof(char)); // on the heap
    s3[0] = '!';
    s3[1] = '\0';
    printf("&s1 = %p, s1 = %p, sizeof(s1) = %zd\n", &s1, s1, sizeof(s1));
    printf("&s2 = %p, s2 = %p, sizeof(s2) = %zd\n", &s2, s2, sizeof(s2));
    printf("&s3 = %p, s3 = %p, sizeof(s3) = %zd\n", &s3, s3, sizeof(s3));

    // Prints something like:
    // &s1 = 0x7fffec3886f0, s1 = 0x4008d8, sizeof(s1) = 8
    // &s2 = 0x7fffec388700, s2 = 0x7fffec388700, sizeof(s2) = 6
    // &s3 = 0x7fffec3886f8, s3 = 0x1631010, sizeof(s3) = 8
}
```

7.7 Functions and Function Pointers

It may not seem obvious at first, but functions themselves have addresses, which are always in the *code segment* of the program. A function name can be thought of as the address of the code of that function. This address can be stored in a variable, which is called a *function pointer*.

Function pointers are awkward to use in C, but nonetheless extremely important. The C-library functions *qsort* and *bsearch* both use function pointers to separate the sorting and searching code from the code that compares two elements. This clever design means that the sort and search logic only needs to be written once no matter what is being sorted or searched. PA02 requires the use of function pointers. PA05 can be made much shorter by using function pointers, but this is not necessary to pass the assignment.

```c
// funptr is a pointer to a function that takes two "const char *" ...
    parameters, and returns an int.
int (*funptr)(const char *, const char *);
funptr = strcmp; // funptr points to the code for the strcmp function.
funptr("Hello", "Zap"); // compares two strings as per strcmp.
```

7.8 Type Qualifiers

C has two “type qualifiers”: *const* and *volatile*. A qualifier can be added to any type and it qualifies (i.e., modifies) the type.

A *const* type is read-only, and often used with string constants.

```c
const char * str = "Lady-beetle";
str[0] = 'l'; // Error, assignment to read-only location
strcpy(str, "lady-beetle"); // Warning, strcpy(...) discards const ...
    qualifier from str
```

The const qualifier can be stripped by casting the pointer...

```c
const char * str = "Lady-beetle";
strcpy((char *) str, "lady-beetle"); // No warning, but segfaults... why?
```

The *volatile* keyword is not used in this course; however, it is important to hardware developers, and in other specialized code. The volatile keyword tells the compiler that the type may be modified by anything at anytime. This prohibits some important compiler optimizations.

7.9 Typedef

The “typedef” keyword is used to alias types. It is usually used to conveniently refer to complex types with shorter and more descriptive names. A typedef always has the following form:

```
typedef <full-type-name> <short-name>;
```

For example:

```
typedef char * c_string; // c_string is now an alias for the ``char *'' type
const c_string fav_fruit = "Apples.";
```

Typedef is often used to simplify the names of struct-types:

```
typedef struct Date_tag {
  int year;
  char month, day;
} Date; // "Date" is now an alias for "struct Date_tag"
// "birthday" and "anniversary" have the same type.
Date birthday;
struct Date_tag anniversary;
```

8 Declarations and Definitions

A *declaration* tells the compiler about some variable or function. It can appear multiple times. Functions are often declared in a separate header file. A *definition* defines storage (i.e., memory in the running program) for a variable or function, and may only ever appear once. Every definition is also a declaration.

The C compiler must encounter every variable or function at least once before it is used. Sometimes function A calls function B, and function B calls function A. Declarations are used to get around this circular dependence.

```
int spaceship[100]; // A definition (and also a declaration) -- storage ...
    is allocated
int mooo[]; // A declaration to an array of some (unspecified) size
double square(double x); // Declaration of the function "square"

// Definition of the function "square". Note that the machine-code for ...
    "square" is allocated in the "code segment" of the program. The ...
    address of that code is referenced through the name of the function. ...
    (i.e., "square", with no parameter list refers to the address of the ...
    machine-code -- where-as "square(value)" executes the function.)
double square(double x)
{
  return x * x;
```

```c
}

struct Client; // A declaration that there is a type: "struct Client"
void Client_print(struct Client * client); // declaration of a function ...
    that uses a pointer to "struct Client"

// Declaration of "struct Client" -- note that no storage has been ...
    allocated.
struct Client {
  int client_id;
  const char * name;
};

struct Client client_1; // definition of "client_1" of type "struct Client"
```

A The Invention of the Computer

This appendix outlines how we came to have computers and the internet. It is not important to memorize the details in these sections; however, students should know the gist of the history.

A.1 Part I - From Counting, to Calculation, to Computation

Computers evolved out of special purpose calculation devices. Although it may be hard to imagine, the abacus is one such device. They are as old as civilization, but still in use today. With practice, even complex arithmetic is remarkably quick. The abacus is a testament to practical problem solving, and demonstrates the utility of using devices to perform calculations. The abacus grew out of a particular need: to perform arithmetic. As mathematics became more sophisticated, the practical needs changed. Mathematicians used to publish books full of number tables that were pre-calculated for specific purposes. For example, in 1614, John Napier published a 147-page book that included ninety pages of number tables useful for calculating the positions of planets. These types of number tables became increasingly important, and mathematicians and inventors set about building various machines and devices to speed the process of creating them.

By the time of the industrial revolution, number tables were being used in the design of machine tools and to power factories. To produce these tables, inventors built calculation machines using a wide array of newly available high quality machine parts, such as ratchets and gears. These machines had immediate commercial value, but were always limited to performing particular types of calculations. Although scientists developed ingenious schemes to make calculation machines more flexible, no general theory of computation was known.

One English scientist, ahead of his time, did tackle the problem of general computation. In 1837, Charles Babbage described a type of general purpose computing device, which he called the "Analytical Engine". It is recognized as the first ever specification of a true computer capable of any type of calculation. Remarkably, the Analytical Engine was programmable even though it was designed out 19th century machine components.

The first computer program was written by the mathematician and mystic Augusta Ada King, commonly known as Ada Lovelace. Babbage and Lovelace were correspondents, and wrote numerous letters about the Analytical Engine, and other calculating machines. In these letters, Lovelace described both computers and computer software, and gave an algorithm for the Analytical Engine that computed a sequence of mathematically important numbers called Bernoulli numbers. Lovelace, however, recognized that general purpose computing would have applications well beyond calculating mathematical tables. She put forward the idea that computers could use a mathematical model to compose elaborate musical compositions. Lovelace was correct, and today computers are capable of creating completely novel musical compositions in a variety of styles.

Unfortunately, the Analytical Engine fell into obscurity soon after Babbage's death in 1871. Both industry and academia overlooked Lovelace and Babbage's shared conceptual leap, and the invention of the computer would have to wait almost a century. By the late 1930s, computers were a natural extension of the sophisticated calculating machines of the day, and were independently invented in three different countries. Both Britain and the United States made foundational contributions to computing; however, the first true computer was built by a German engineer working in near complete isolation. In 1941, in Berlin, Konrad Zuse completed the Z3, the third revision of Zuse's mechanical calculating devices. Zuse was primarily concerned with practical calculation problems, such as measuring the surface area of airplane wings, but approached these problems in a principled way that separated the calculation logic from the machinery that performed the calculation. His machines were built out of recycled telephone equipment, and sported numerous innovations: the very apotheosis of practical problem solving. It took centuries of innovation, and a few misstarts, but the computer had finally arrived, it proved to be spectacularly useful.

A.2 Part II - Winning the War with Brains

A lone German engineer built the first true computer during World War II; however, his work was stymied by isolation and under-appreciated by the German leadership. This was not the case across the English Channel. In the early stages of the war, Britain was engaged in a desperate fight for survival. The British attempted to gain an operational advantage over their opponents by intercepting and decrypting enemy communications. Code breaking requires a combination of ingenuity and millions of statistical calculations. Although computers did not yet exist in Britain, engineers and mathematicians successfully defeated German encryption by designing and building enormous calculation machines. These machines were too specialized to be true computers, but their designers would become some of the world's first computer scientists.

One such scientist was Alan Turing, often lauded as the father of computing and artificial intelligence. Turing was a mathematician who designed techniques to break German ciphers. An accomplished marathon runner, Turning sometimes ran the 64km from his workplace in Bletchley Park, to head office in London, in order to attend high level war meetings. In 1945, Turing become an Officer of the Order of the British Empire for his scientific contributions to code breaking, which included designing code-breaking machines that deciphered tens of thousands of enemy communications. Turing's most notable accomplishment, however, was proving that any mathematical algorithm can be expressed and computed by a machine that reads and writes to a tape of 1s and 0s. This type of hypothetical machine is called a ”Turing machine”, and it is the mathematical foundation for modern computing. The term ”Turing-complete” is used to refer to a computer's ability to execute any calculation as opposed to machines limited to particular tasks. Turing's genius is celebrated around the world each year with the Turing Award, which is the computing world's equivalent of the Nobel Prize.

Even though Turing developed the theory for what a computer is, and designed some of the most sophisticated calculation machines ever built, the British never built a ”Turing-complete” computer during the war – they were focused on code-breaking. The Allies did build a true computer, however, and it was the harbinger of a new era of calculation. Between 1943 and 1946, US scientists at the University of Pennsylvania designed and built the first ever fully electric programmable computer. It was called ENIAC, and was about one thousand times faster than anything ever built. Being a true computer, it was superbly flexible, and performed numerous important calculations even in construction, including calculations related to the design of the first atom bomb.

The early pioneers who worked on these computers understood their significance beyond military applications. Popular culture already referred to these machines as ”Brains”, but ENIAC was heralded as a ”Giant Brain”. Scientists and artists had long stoked the public imagination with dreams of synthetic intelligence, and now anything seemed possible. The remarkable usefulness of computers was already well established, and many scientists, including Turing, now turned their attention to solving the mystery of intelligence, perhaps the grandest mystery of all. No one foresaw the strange limitations that computers would have in this regard, but the birth of computers did lead to a deeper understanding of who we are, and ushered in a new age of scientific inquiry.

A.3 Part III - On the Application of Mathematics

Academics were already talking about a global network of inter-connected computers in the early 1960s. A key conceptual goal was a network where any computer could talk to any other computer at any time. By contrast, existing technologies of the day relied on dedicated communication channels. For example, a radio broadcast was transmitted on a certain frequency which listeners tuned in to. Similarly, the telephone system relied on circuit switching, where a dedicated electrical circuit connected the mouthpiece on one phone to the ear piece on another phone, through an elaborate and centralised network that employed people to manually plug different circuits together.

To accomplish their goal of a global network, early computer scientists realised they would need to create channels that could be shared between multiple messengers at the same time. To achieve this, scientists broke messages down into small independent self-addressed packets, and routed these through systems that examined and forwarded these packets somewhere closer to their destination. In essence this is exactly how the postal system works, where a letter is sent to different postal sorting stations, and at each step gets progressively closer to its final destination. The main difference is that computers split a single message (a letter) into many tiny packets (many small letters), and these small packets are generally reassembled in order by the receiving computer.

This technology is called packet switching, and is at the very heart of the Internet Protocol (IP) that drives world-wide communications. It was first conceived and written about in a 1961-62 thesis by MIT (Massachusetts Institute of Technology) graduate student Dr Leonard Kleinrock, who established the mathematical theory for packet networks in the context of computer systems. Over the 60s and 70s, these researchers succeeded in convincing ARPA (Advanced Research Projects Agency) – a research branch of the US military – to fund and build ARPANET, the first packet switching computer network. One of their goals was time-sharing, whereby research institutions could share the processing power of expensive and rare super-computers.

It is a common misconception that ARPANET was originally designed as a communications network that could survive a nuclear attack. Instead, ARPANET was built because, with or without nuclear attacks, circuit switching networks were too unreliable for computer applications over long distances.

Many mathematical innovations were required to develop reliable communications in unreliable and inconsistent environments. By the 1980s, this technology was mature enough to form the basis of a network that integrated universities across the world. It was called the "Internet" (inter-network) because it was a collection of independent networks that anyone could plug themselves into. The convenience and freedom to transmit information to anyone, and at low cost, attracted increasing commercial attention in the late 1990s, spurring software innovation and widespread interest from people who were otherwise not computer enthusiasts. Today the Internet is a cultural institution, and the bedrock of multinational and grassroots economic and political activity. It was made possible by the imagination and tenacity of early computer scientists who applied mathematical principles to address the problem of reliable communication across unreliable equipment.

References

[1] Himanshu Arora. Journey of a c program to linux executable in 4 stages. http://www.thegeekstuff.com/2011/10/c-program-to-an-executable, 2011.

[2] Tom DeMarco, Timothy Lister, and Dorset House. *Peopleware: productive projects and teams*. Pearson Education, 2013.

[3] GeorgeE. Hoffer and MichaelD. Pratt. Used vehicles, lemons markets, and used car rules: Some empirical evidence. *Journal of Consumer Policy*, 10(4):409–414, 1987.