Computer Science 433
Programming Languages
The College of Saint Rose
Fall 2012

Topic Notes: C

The C programming language is the lowest-level language (closest to the hardware) we will study in detail this semester. C is a widely-used, general purpose language, well-suited to low-level systems programming and scientific computation.

We will study it from the point of view of a Java programmer, focusing on the features that make C significantly different from Java. Fortunately, Java borrowed much of its syntax from C, so it is not difficult for a Java programmer to read most C programs.

C++ is a superset of C (that is, any valid C program is also a valid C++ program, just one that doesn't take advantage of the additional features of C++). C++ adds object-oriented feautures. For now, we will look only at C, not C++.

A Very Simple C Program

We will begin by seeing how to compile and run a very simple C program (`hello.c`) in a Unix environment.

See Example:
`/home/cs433/examples/hello_c`

We will assume that we are working at the Unix command line. If you use `ssh` to connect to `mogul.strose.edu`, you are already there. If you are going log into the console of a Linux system or a Mac, you will need to open a Terminal window.

For you to run this, you will want to copy the example to your own directory. But first, we'll create a directory for it:

```
mkdir ~/hello
```

This creates a new directory (folder) in your home directory (indicated by the `~`) called `hello`.

To copy the example from the shared area to your new directory:

```
cp /home/cs433/hello_c/hello.c ~/hello
```

We now change directory to the copy in your own directory:

```
cd ~/hello
```

And compile and run it:

```
gcc hello.c
./a.out
```

Things to note from this simple example:

- We run a program named `gcc`, which is a free C compiler.
- `gcc`, in its simplest form, can be used to compile a C program in a single file:

  ```
  gcc hello.c
  ```

  In this case, we're asking `gcc` to compile a C program found in the file `hello.c`.

  Since we didn't specify what to call the executable program produced, gcc produces a file `a.out`. The name is `a.out` for historical reasons.
- When we want to run a program located in our current directory in a Unix shell, we type its name.
  - For example, when we wanted to run `gcc`, we typed its name, and the Unix shell found a program on the system in a file named `gcc`.
  - How does it know where to find it? The shell searches for programs in a sequence of directories known as the *search path*. Try: `env`.
  - So if we want to run `a.out`, we should be able to type its name. But our current directory, always referred to in a Unix shell by "`.`", is not in the search path. We need to specify the "`.`" as part of the command to run:

    ```
    ./a.out
    ```
- Of course, we probably don't want to compile up a bunch of programs all named `a.out`, so we usually ask `gcc` to put its output in a file named as one of the parameters to `gcc`:

  ```
  gcc -o hello hello.c
  ```

  Here, the executable file produced is called `hello`.
- And in the program itself, let's make sure we understand everything:
  - At the top of the file, we have a big comment describing what the program does, who wrote it, and when. Your programs should have something similar in each C file.
  - We are going to use a C library function called `printf` to print a message to the screen. Before we can use this function, we need to tell the C compiler about it. For C library functions, the needed information is provided in *header files*, which usually end in `.h`. In this case, we need to include `stdio.h`. Why? See `man 3 printf`. (More on the Unix manual later.)

  - A C program starts its execution by calling the function `main`. Any command-line parameters are provided to `main` through the first two arguments to main, traditionally declared as `argc`, the number of command-line parameters (including the name of the program itself), and `argv`, an array of pointers to character strings, each of which represents one of the command-line parameters. In this case, we don't use them, but there they are.
  - Our call to `printf` results in the string passed as a parameter to be printed to the screen. The `\n` results in a new line.
  - Our `main` function returns an `int` value. A value of 0 returned from `main` generally indicates a successful execution, while a non-zero return indicates an error condition. So we return a 0.

- Notes for Java programmers:

  - Good news: much of the syntax of Java was borrowed from C, so a lot of things will look familiar.
  - There are no classes and methods, just *functions*, which can be called at any time. Any information a function needs to do its job must be provided by its parameters or exist in *global variables* – variable declared outside of every function and which are accessible from all functions.

A Bit More Complex Example

We next consider an unnecessarily complicated C program that computes the greatest common denominator of two integer values.

See Example:
`/home/cs433/examples/gcd`

Lots of things to notice here:

- We have four files:

  - `gcd.c`: the implementation of the `gcd` function
  - `gcd.h`: a header file with a prototype for the `gcd` function
  - `gcdmain.c`: a main program that determines the input numbers, computes the GCD, and prints the answer, and
  - `Makefile`: a "make file" that gives a set of rules for compiling these files into the executable program `gcdmain`.

  When executing, functions from both `gcdmain.c` (`main`) and `gcd.c` (`gcd`) will be used. Both of these are included in our executable file `gcdmain`.

- Start with `gcd.c`:
  - This is a very simple recursive function to compute the greatest common denominator using the Euclidean Algorithm.
  - There is no `main` function here, so if we try to compile this by itself as we did with `hello.c`, we will get an error.
  - Instead, we have `gcc` use "compile only" mode to generate an *object file* `gcd.o` from `gcd.c`:

    ```
    gcc -c gcd.c
    ```

    `gcd.o` is a compiled version of `gcd.c`, but it cannot be executed.

    C (and many other languages) require a two steps for source code to be converted into an executable. The first step compiles source code into object code, the second takes a collection of object code files and *links* together the references in those files into an executable file. (There's much more to discuss here, but this should suffice for now.)

- Next up, `gcd.h`:
  - Much like `stdio.h` tells the compiler what it needs to know about `printf` (among other things), we have `gcd.h` to tell other C functions what they need to know about the function `gcd`. Namely, that it's a function that takes two `int`s as parameters and returns an `int`.
  - Any C file that contains a function that calls `gcd` should `#include "gcd.h"`.

- The driver program, `gcdmain.c`:
  - We include several header files to tell the compiler what it needs to know about C library functions (and our `gcd` function) that are called by functions defined here.
  - This is where our `main` function is defined.
  - We can define local variables to functions, just like local variables in a Java method.
  - In this case, we look at the arguments to `main` that provide the command-line parameters of our program: `argc` and `argv`.
  - If we have fewer than three command-line parameters, including the program name itself (which is always there), we prompt the user for two numbers (with `printf`), then read in two numbers from the terminal with `scanf`.
  - This is a good time to mention C strings. There is no "string" data type in C. Strings are just `null`-terminated arrays of `char`. Unlike Java, arrays do not come equipped with any way to tell how large they are (like Java's `.length`) so the only way we can tell the length of a C string is to follow it along until we get to the `null` terminator, which is character `'\0'`.

- `scanf` is a very strange thing. It will make a bit more sense when you are more familiar with `printf`, but for now we can summarize what we see there as "read in two integer values (represented by the `%d`'s in the *format string*), and put them into the place pointed at by the address of `a` and the address of `b`, then return the number of values that matched the input with the correct format." Right. And you thought I/O was a pain in Java.
- The `scanf` call forces us to think a bit about *pointers*, which are the key to understanding so much of how C works. `scanf`'s parameters after the format string are always a list of pointers to a place in memory where there is room to put the values being read in. In this case, we want the two `int` values to end up in the local variables `a` and `b`, so we have to take the address of those variables with the `&` operator. Don't worry, it will make better sense when you see more examples.
- Next, we check to make sure that the input to `scanf` did, in fact, represent two `int` values. If not, we print an error message and exit. Otherwise, we continue.
- Some things to notice in the error condition:
  - We use `fprintf` instead of `printf`. This is because we want to give this output special significance. Rather than sending it to the *standard output*, which is what `printf` would do, we send it to *standard error*, by using `fprintf` and specifying `stderr` as the first parameter. Java supports the same idea: use `System.err` instead of `System.out`.
  - Other than that, it works just like `printf`. We give it a format string. In this case, it includes one specifier, a `%s`, which means to expect an additional parameter which is a character string (well, really a pointer to a `null`-terminated array of `char`). Here, the string is `argv[0]`, the first command-line parameter, which is always the name of the program. This labels the error message with the program name.
  - Once we have detected the error, we don't want to continue, so we call the `exit` function with an error code of 1 to terminate execution. We could also use the call `return 1;`.
- In the case where at least two command-line parameters were provided, we try to convert them (`argv[1]` and `argv[2]`) to integer values. This is done with the overly complicated `strtol` function, which we use, then check error conditions.
  - The man page for `strtol` tells us we need to include two additional header files, `stdlib.h` and `limits.h`.
  - It also tells us about the parameters to `strtol`, which are the string which we would like to convert to a number, a pointer into the string at the point beyond which we matched a number (which we don't care about, so we pass in `NULL`), and the base to use for the conversion. We also see that the number is the return value.
  - Error checking for `strtol` is messy – we need to check the variable `errno`, defined in `errno.h`, to see if an error condition was encountered. If so, `errno` will be a non-zero value and we print an error message and exit.

    - Note that the error check here has two `%s`'s, so we have two additional parameters to `fprintf`, both pointers to strings.
  - Finally, we're ready to check that the numbers entered are non-negative, and if so, we print out the answer (obtained by the `gcd` function call inside of a `printf` parameter).
  - This file includes a `main` function, so we might think we could compile it to an executable as we did with `hello.c`, but if we try, we'll find that it doesn't know how to find the `gcd` function. Again, we'll have to compile but not link:

    ```
    gcc -c gcdmain.c
    ```

    This produces the object file `gcdmain.o`. We need to *link* together our two object files, which, together, have the function definitions we need:

    ```
    gcc -o gcdmain gcdmain.o gcd.o
    ```

    This gives us `gcdmain`, which we can run.

- The `Makefile` contains rules to generate a sequence of calls to `gcc` that will correctly compile and link the `gcdmain` executable.

The bad news: that was a lot of trouble just to write a simple program.

The good news: you will have a lot of examples to go on and you can ask a lot of questions.

Example: Matrix Multiplication

Let's consider a slightly larger example: matrix-matrix multiplication.

See Example:
`/home/cs433/examples/matmult`

- This is another example of separate compilation – The function in `timer.c` will be useful if we want to measure execution times. We tell `matmult.c` about it with the line

  ```
  #include "timer.h"
  ```

  This provides a *prototype* of the function in `timer.c`. In many cases, this file would also define any data structures or constants/macros used by the functions it defines.
  This is a good model to use as you move forward and develop more complicated C programs. Group functions as you would group methods in a Java class or member functions in a C++ class.
- Along those same lines, the include files in angle brackets

  ```
  #include <stdio.h>
  #include <stdlib.h>
  #include <sys/time.h>
  ```

specify system-wide header files. By convention (though most compilers don't really make a distinction) system-wide header files are in angle brackets, while your own header files are in double quotes.

- Each file can then be compiled separately to create an *object file* (`.o` file) from the C source. These object files are all listed at the linking step.
  What happens for function `diffgettime()` at compile time? Link time?
- The program uses two system calls: `printf()` and `gettimeofday()`. To see how these work, we can look at their *man pages*:

  ```
  man printf
  ```

  to see everything we wanted to know about a particular system call. But if you do this, you might get a man page for a command-line utility called `printf` instead of the system call `printf()`. Not what we were looking for. The Unix manual is divided up into sections. The most important of these sections, for our purposes, are Section 1: User Commands, and Section 3: Library Functions. If we don't ask for a section, we get section 1. Since section 1 contains an entry for `printf`, that's what it produced. To force it to give you the system call manual page, you can use (under Solaris)

  ```
  man 3 printf
  ```

  This actually tells it to look in section 3, which contains C library functions. How did I know to look in section 3? Mainly because the `printf` man page in section 1 told me so, at the bottom under the "See Also" section.
  Fortunately, you only need to concern yourself with what section of the manual to use when you look something up that it in more than one section. For example,

  ```
  man gettimeofday
  ```

  brings up the man page we want, for the `gettimeofday()` system call in section 2 (the system calls section) under Mac OS X and FreeBSD, and in section P (the POSIX library functions) on some versions of Linux, including that on `mogul.strose.edu`.
  If you see a reference to something like `ctime(3)` in the "See Also" section of a man page, such as that in `gettimeofday()`'s man page, that means the `ctime()` man page is in section 3. I will use this notation as appropriate throughout the semester.
  You will find the Unix manual very helpful as we move forward.
- So what does `gettimeofday(2)` do? See the man page and look at the usage in the example program.
  - what's going on with memory management?
  - what would happen if we declared `struct timeval *` variables instead of `struct timeval`?

  `gettimeofday(2)` returns *wall clock* times. This is the amount of elapsed real time. So if our process is taking turns on the CPU with other processes (see the Operating Systems course) and it is not always running, it continues to accumulate wall clock time, but not *CPU usage time*. There are also system calls to examine CPU usage time which we may consider later.

- The `Makefile` is using the GCC compiler (`gcc`) with the option `-O` for optimization. If you want to run this with a different compiler or optimization flags, you can change the `CC=` line in the `Makefile`.

If we compile and run this program, it reports initialization and matrix multiplication times. Initialization is just filling in matrices `a` and `b`. Then we compute the value of each element of `c` using the dot product of the corresponding row of `a` and column of `b`.

Aside: remember your data structures and algorithms: what is the complexity of matrix-matrix multiply?

More on Arrays

Our next example demonstrates some ways we can deal with arrays in C.

See Example:
`/home/cs433/examples/arrays`

The comments in this program describe its usage of the most important C features. Pay special attention to the usage of `malloc` to allocate chunks of memory and `free` to return them to the system when finished.

This demonstrates one of the key differences between C and Java: we have to tell C when we are finished with our allocated memory. Java uses *garbage collection* to reclaim memory no longer in use automatically. We will consider the merits of both approaches later in the semester; for now we simply need to remember that any memory we allocate in C must be released when we are done with it. Advice: when you add a `malloc()`, immediately add the corresponding `free()` in an appropriate place.

File I/O

This example demonstrated some File I/O in C, along with some character manipulations.

See Example:
`/home/cs433/examples/lettercount`

Comments within the code point out interesting features.

Structures and More Memory Management

Next, we look at a somewhat silly example that demonstrates the use of structures and more complex memory management.

See Example:
`/home/cs433/examples/ratios`

Again, the comments in the program describe in detail what is going on and why.

String Processing

We saw earlier that strings in C are simply represented as `NULL`-terminated arrays of `char`. The following example demonstrates more about this and gives examples of some of C's string processing functions.

See Example:
`/home/cs433/examples/strings`