CptS 360 (System Programming)
Unit 3: System Programming in C

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- C++ is an object-oriented application language. C is a system programming language. Code looks different.
- Using UNIX-style development tools lets you port your code to more platforms.
- Most of these tools are command line based: GUIs aren't very useful when editing over *ssh(1)*[1].
- Don't be afraid of the command line: It is not couth to write system programs with an IDE.

[1] Unless you have X11 forwarding, but let's not go there.

Reference

- Stevens & Rago, Ch. 3

C vs C++

- If you come from a C++ background, C does things a little differently.
- All of the stuff we cover about C can be done in C++, but that's not always good C++ practice.
- C is a "lower level language", but that's the way system programmers want it.

Memory Allocation on the Heap

- in C++, you used
  - `new` to allocate dynamic ("heap") memory
  - `delete` to deallocate heap memory
- in C, you use
  - *malloc(3)* to allocate dynamic memory
    recommended syntax for N elements of type *type*:

    ```
    type *name;
    ...
    name = (type *) malloc(N * sizeof(type));
    ```

    type can be a base type (e.g. `int`, `char *`, etc.) or a `struct`.

    (See the `demos/dn_allocarray` demo.) – a macro that makes this more intuitive
  - *free(3)* to deallocate dynamic memory
  - *or* we build our own memory allocation schemes

Data Aggregation

- In C++, we use `class` to combine data together to describe the contents of an "object".
- in C, we use `struct` to do much the same thing, but there are no distinct "objects" as such.
  - In C++ terms, `structs` are classes with all-public members and no methods.
  - We can use `structs` to approximate simple object-oriented programming (as is done in the Linux kernel and as we'll see in the `Rational` lab).
  - Sometimes, system programmers use `structs` to do their own sort of low-level polymorphism (as we'll see in networking).

Compiler Errors

- In C++, compiler messages can be cryptic at times.
- In C, compiler messages are usually pretty straightforward.

In both languages, always fix the first compiler bug first. It may be the cause of bugs that come after it.

Base Type typedefs

typedefs help make your code and your intents clearer. Here's one that's already defined and used for many system calls:

```
typedef unsigned int size_t;
```

that simply says that whenever you use size_t, you really mean "unsigned int" and you intend it to be the size of something.

Used consistently, typedefs like this can make changes in type choices very easy (e.g. pid_t).

struct typedefs

You can also use them to make `struct` declarations look like type declarations. Instead of

```
struct Student {
    char *firstName;
    char *lastName;
};
```

you can say

```
typedef struct {
    char *firstName;
    char *lastName;
} Student;
```

so that you can declare "`Student thisPerson;`" instead of "`struct Student thisPerson;`". (Big deal, I know, but some system programmers like it.)

enum typedefs

You can also use typedefs for enums:

```
typedef enum {
    RED, GREEN, BLUE
} Color;
...
Color colorOfBook;
```

(*gdb(1)* knows about *enum*s, typedef'd or not, and will print symbols.)

Boolean Expressions

- In C++,
  - Boolean variables are of type `bool`.
  - `true` and `false` are built-in constants.
- In C,
  - There is no `bool` type, but `int`s are typically used.
  - 0 is "false" and anything else (but often 1) is "true". (This also works in C++.)

- Especially when used for system programs, it is not couth to use

```
if (expression)
    return 1;
else
    return 0;
```

when

```
return expression;
```

does the same thing.

I/O

- In C++, the `iostream` library performs input and output using the >> and << operators.
- In C, the `stdio` library performs input and output with functions like
  - *scanf(3)*
  - *printf(3)*
  - *getc(3)*
  - *putc(3)*
  - *fread(3)*
  - *fwrite(3)*

Declaring Constants

In C (or C++), you can declare a constant values as ..

a *cpp(1)* macro...

```
#define N_A 10

int a[N_A];

int main(void)
{
    int i;

    a[0] = 1;
    a[1] = 1;
    for (i=2; i<N_A; i++) {
        a[i]=a[i-1]+a[i-2];
    }
    return 0;
}
```

an `enum` symbol...

```
enum { N_A = 10 };

int a[N_A];

int main(void)
{
    int i;

    a[0] = 1;
    a[1] = 1;
    for (i=2; i<N_A; i++) {
        a[i]=a[i-1]+a[i-2];
    }
    return 0;
}
```

a `const` variable...

```
const int N_A = 10;

int a[N_A];

int main(void)
{
    int i;

    a[0] = 1;
    a[1] = 1;
    for (i=2; i<N_A; i++) {
        a[i]=a[i-1]+a[i-2];
    }
    return 0;
}
```

(Here, it doesn't work.)

Note the convention that constant values are in all-caps.

Other C++ Stuff Missing from C

These are C++ features not built in to C. You can sometimes provide equivalents ...

- methods
  Use function naming and calling conventions?
- inheritance (esp. multiple)
  Use `struct` embedding, and naming conventions?
- call-by-reference
  Pass a (`const`) pointer.
- operator overloading
  Use (inline) functions.
- templates
  Use *cpp(1)* macros?
- namespaces
  There's no obvious way.
- smart pointers[a]
  C pointers are C++ `raw_ptrs`.

[a] `unique_ptr`, `shared_ptr`, `weak_ptr`

... but they'll probably be idiosyncratic (non-standard).

Data Sizes Matter

System programmers need to keep track of the sizes of variables,

- Efficient space allocation can improve performance (esp. taking advantage of caching).
- In C, the `sizeof()` operator returns the size of its argument in bytes.
- Do not confuse the `sizeof()` operator with the *strlen(3)* function.

(Run the `demos/d`*n*`_sizes` demo.) – And understand it!

chars vs. C++ strings

- C++ has the `string` class.
- C uses the `char` type for strings.
  - `char`s are 1-byte `int`s.
  - You can do arithmetic and comparisons on them:

```
int isPassing(char grade)
{
    return 'A' <= grade && grade <= 'C';
}
```

  - C "strings" are NUL-terminated arrays of `char`s, so

```
char s[] = "spam";
```

  means the same thing as

```
char s[] = { 's', 'p', 'a', 'm', '\0' };
```

CFLAGS for the Paranoid

In this class, your code is expected to compile without warnings using these values passed to *gcc(1)*, usually as the CFLAGS macro in your makefile:

```
-g -Wall -Wstrict-prototypes
```

But that's just the start. There are *lots* of *gcc(1)* compiler flags to control how the compiler looks for possible errors.

If you're interested, some combinations are listed in demos/d3_cflags/cflags.mk.

UNIX System Calls

- two kinds of (system) library:
  - static
  - dynamic (most system libraries are this)
- POSIX Standard
  - IEEE Std. 1001.1-1989 (a.k.a. ISO/IEC 9945)
  - includes commands and the API (threads, real-time, IPC, etc.)
  - supported just about everywhere, except Windows,
    - Interix environment subsystem (up to and including Windows 7)
    - deprecated in Windows 8
    - alternatives: Cygwin (separate library), MinGW (built-in)
  - this class mostly concerns the API (POSIX.1[abc])

Some common POSIX programming features...

POSIX Error Handling

If a system call returns a negative value (usually, but not always, -1), something went wrong. Look in the global `errno` for details.

- To get `errno`:

```
#include <errno.h>
```

- To get the string associated with *errno(3)*:

```
#include <string.h>
char *strerror(int ernnum);
```

- To print an error string:

```
#include <errno.h>
void perror(const char *msg);
```

prints "*msg*: *error string*" on *stderr(3)*.

Remember: `errno` is set when an error occurs, but never cleared.

Limits

- Q: How large should I declare an array that will hold a path?
  A: Take a look at `file:///usr/include/limits.h`.
- Other useful quantities (to name a few):

`CHAR_MIN`	`CHAR_MAX`	`UCHAR_MAX`
`SHRT_MIN`	`SHRT_MAX`	`USHRT_MAX`
`INT_MIN`	`INT_MAX`	`UINT_MAX`
`LONG_MIN`	`LONG_MAX`	`ULONG_MAX`

cpp(1) Feature Test Macros

Pay attention: These are sometimes required in the *man(1)* page synopsis.

- `_GNU_SOURCE`
  to enable non-POSIX GNU features:

  ```
  #define _GNU_SOURCE
  #include <whatever>
  ```

- `_POSIX_SOURCE`
  to force POSIX compliance:

  ```
  #define _POSIX_SOURCE
  #include <whatever>
  ```

- `__STDC__`
  (mainly) to create macros to avoid function prototyping, which you shouldn't do. You probably won't use it, but you might come across it.

glib: Don't Reinvent the Wheel

glib (https://docs.gtk.org/glib) is a classic C utility library with many useful functions, macros, and data structures.

- some of it is trivial, some is amazing (e.g. C objects)
- maintained by the GNOME Project, a leading free software organization
- spun off from (but still used by) gtk, the Gnome Toolkit
- very stable (started in 1998)
- works everywhere
- comparable to C++'s Boost or STL
- not to be confused with glibc (aka libc), which contains the POSIX functions (the main focus of our course).

You are free to use glib facilities in your work for this class.

Aside: Use the above web page or browse file:///usr/share/gtk-doc/html (file:///usr/share/gtk-doc/html/glib/index.html, for example), not the glib man page.

Development Tools

- There is quite an assortment of development tools available.
- We'll cover debugging tools in the next unit.

Getting Help

You are going to need help[2]. Several sources:

- manual pages
  Use *man(1)* (or browse `https://www.kernel.org/doc/man-pages`) (bookmark?)
- *info(1)*
- *grep(1)* (esp. under `/usr/include`)
- `/usr/share/doc`
- `https://google.com` (of course)

(Of course, there's always the instructor.)

[2] Nothing personal.

man(1) pages

Traditional UNIX manual page sections are:

section	contains
1	commands
2	system calls (most POSIX API pages)
3	general-purpose libraries (e.g. `math`, `stdio`, `dbm`)
4	devices
5	file formats
6	games
7	conventions and miscellany
8	system administration

To get help on these classifications, use "`$ man` *n* `intro`".

`man` Command Invocations

- `$ man` *`n entry`*
  gets information on *`entry`*, optionally restricting the search to section *`n`*
- `$ whatis` *`entry`*
  gets a one-line description of *`entry`*
- `$ apropos` *`keyword`* (or `$ man -k` *`keyword`*)
  gets one-line description of one or more pages whose description includes *`keyword`*
- `man:`*`entry`* or `man:`*`entry`*`(`*`section`*`)`
  in browser (Firefox, at least)

info

- emacs-like hypertext help browser.
- part of GNU
- Run "$ info" by itself to start at the top level with a list of (all) commands.
- Run "$ info *command*" for information on *command*.
- especially useful for compilers and make

Installing Documentation

On Linux systems, if you want documentation on *packageName*, look for the package *packageName*-doc and install it.

grep(1)

`grep(1)` is a very useful tool, if you know what to grep for...

Use one of these commands

```
$ grep pat /usr/include/*.h
$ grep pat /usr/include/*/*.h
$ grep pat /usr/include/**/*.h
```

to find a `#define` or function prototype that matches *pat*. (That last one needs you to enable the *bash(1)* `globstar` option.)

Use

```
$ zgrep -l pat /usr/share/man/mann/*.gz
```

to find some remembered phrase or keyword in a man page you may have seen before.

Other help source: `/usr/share/doc/*`

make(1)

- System programmers use *make(1)* to minimize compile time and remember necessary options.
- We'll touch on the basics.
- It is not couth to make the makefile executable.
- Good for more than just compiling.
- It helps to draw a dependency diagram to design the makefile.

(See the `demos/d`*n*`_make` demos.)

Makefile Syntax

- comments
  - begin with "#" to end-of-line
- targets
  - left of ":"
  - may be real files or nonexistent ("phony") goals (e.g. `clean`)
- dependencies
  - right of ":"
  - may be real files or other targets
- rules
  - need to use `Tab`: spaces won't do
  - each line is distinct shell command, unless you use ";\" at the end of each line
- suffix rules (advanced)
  - may be built-in or user-defined

make(1) Macros

- definition syntax:

  name=*value*

- usage syntax:

  $(*name*)

- environment variables imported by default
- various macro tricks:
  - new-style substitution:

    ```
    OBJS=$(subst .c,.o,$(SRCS))
    ```

  - old-style substitution:

    ```
    OBJS=$(SRCS:.c=.o)
    ```

  - nested macros:

    ```
    CFLAGS=$(CFLAGS_$(ARCH))
    ```

- remember: D-R-Y

Commonly-used Macros

Most systems have these automatically defined:

`CC`	the C compiler
`CFLAGS`	C compiler options
`CPP`	the C++ compiler (*c++(1)*, *not cpp(1)*)
`CPPFLAGS`	C++ compiler options
`LD`	the loader (usually = `$(CC)`)
`LDFLAGS`	loader options
`MAKE`	*make(1)* itself, with arguments (for recursive makes)

These can be overridden in the makefile or on the command line.

make(1) Command Line Options

`-k`

Don't stop at the first error; keep going as long as possible.

`-n`

Echo the required operations, but do nothing.

`-f` *`filename`*

Use an alternate to `makefile` or `Makefile`.

`-j` *`[#_of_jobs]`*

Run in parallel, if possible. (Really impressive on a multicore system!)

Commonly-Used Targets

(Assuming we're using *make(1)* to compile stuff.)

- `default` (must be first target)
  Compile all executables. (Okay to have several.)
- `install`
  If you're sure they're working, install executables and such in their proper place.
- `clean`
  Get rid of easily-reconstructed temporary files.
- `immaculate`
  In addition to `clean`, get rid of the executable(s).

None of these are actual file names. It's a good practice to label them all as "`.PHONY`" (see demo) so *make(1)* knows that.

makedepend(1)

- analyzes source files to create dependencies of its own.
- handles nested includes
- appends dependencies to the makefile (e.g. `Makefile`)
- "transitive closure on `#include`"

tar(1)

- syntax

  `$ tar [`*`options`*`] file ...`

- where *options* are
  - `-c`
    create new archive (a.k.a. "tarball")
  - `-v`
    be verbose
  - `-f` *filename*
    use *filename* for new or old tarball name
  - `-x`
    extract output
  - `-z`
    compress/uncompress tarball with *gzip(1)*/*gunzip(1)*
  - `-t`
    list the contents of a tarball

Modern Revisioning Systems

- *Subversion* (aka *SVN*)
  - manages whole directory trees of files
  - uses database for central repository
  - allows easy renaming of files and directories
- *Git*
  - devised by Linus Torvalds for the Linux kernel[3]
  - project $\leftrightarrow$ repository
  - repositories often hosted on GitHub (`https://www.github.com`) or GitLab (`https://www.gitlab.com`)
  - each user gets their own copy of the whole repository
  - see `https://xkcd.com/1296` and `https://xkcd.com/1597`
  - Have a GIT cheat sheet!
- *Mercurial*
  - don't know much about this one

[3] UNIX Philosophy: The best tools are the ones you build yourself.

Older Revisioning Systems

You may come across these:

- *SCCS* (Source Code Control System) (obs.)
  - developed in 1970
  - ported to UNIX V7 in 1973
- *RCS* (Revision Control System)
  - developed in 1982
  - still maintained by GNU
  - was mostly superseded by...
- *CVS* (Concurrent Versions System)
  - developed in 1986
  - for multiple developers
  - now superseded by *Subversion* and *Git*

And Then There's...

- the C preprocessor:

```
#if 1
// experimental code
#else
// code that you're replacing,
// but want to keep around just in case
#endif
```

- the demos/d5_new/new script

Text Editors

System programmers should know at least one of these UNIX editors:

- *emacs(1)*
  - "East Coast" (MIT) origin
  - also a development environment
  - programmable in Elisp
  - windowed or console ("`emacs -nw`") mode
- *vim(1)*
  - "West Coast" (UCB) origin (as *vi*)
  - most widely-used by system programmers
  - programmable in Python
  - console mode only (although there's *gvim(1)*)

Or maybe

- *jed(1)* or *jove(1)*
  - low-overhead versions of *emacs(1)*
  - console mode only
- *gedit(1)*
  - Linux (Gnome) only
  - GUI mode only
- Q: Anyone want to plug their favorite editor?

Document Processing

System programmers produce documentation. Popular choices:

- *nroff(1)*, *groff(1)*, and *troff(1)*
  - easy to figure out
  - outdated for regular documents, but...
  - still used for `man` pages
- *Microsoft Word*
  - if you absolutely *must*
- *LibreOffice* or *OpenOffice*
  - WYSIWYG, mostly
  - OpenDoc format (`.od[bgmpst]`) easy to analyze (zipped XML archive) and generate.
- LaTeX
  - a pain to learn, but...
  - worth the effort, esp. for professional publications
  - best book (IMHO): Kopka & Daley's *Guide to LaTeX*
- *ReST* (ReStructured Text)
  - intuitive
  - uses readable ASCII files
  - generates HTML, LaTeX, XML, and PDF slides from the *same* input
  - can insert LaTeX if needed

IDEs: Integrated Development Environments

If you *really* need one:

- *Eclipse*
  - runs ∼everywhere
- *Visual Studio*
  - runs ∼everywhere
- *Qt Creator*
  - C++ only
  - runs ∼everywhere
  - uses Qt4 GUI framework (with Qt Designer)
- *kdevelop(1)*
  - Linux (KDE) only
- *kate(1)*
  - Linux only
- *XCode*
  - MacOS only
- *Code::Blocks*
  - runs ∼everywhere
  - uses wxWidgets GUI framework

Most of these support multiple languages/file formats.
Any others you'd like to plug?