Beyond this course

Readings: CP:AMA 2.1, 15.4

Machine code

In Section 05 we briefly discussed **compiling**: converting *source code* into *machine code* so it can be “run” or *executed*.

Each processor has its own unique machine code language, although some processors are designed to be compatible (*e.g.,* Intel and AMD).

The C language was *designed* to be easily converted into machine code. This is one reason for C’s popularity.

As an example, the following source code:

```
int sum_first(int n) {
  int sum = 0;
  for (int i=1; i <= n; ++i) {
    sum += i;
  }
  return sum;
}
```

generates the following machine code (shown as bytes) when it is *compiled* on an Intel machine.

```
55 89 E5 83 EC 10 C7 45 F8 00 00 00 00 C7 45 FC 01 00 00
00 EB 0A 8B 45 FC 01 45 F8 83 45 FC 01 8B 45 FC 3B 45 08
7E EE 8B 45 F8 C9 C3.
```

How to compile code is covered in CS 241.

When source code is compiled, the identifiers (names) disappear. In the machine code, only *addresses* are used.

The machine code generated for this function

```
int sum_first(int n) {
  int sum = 0;
  for (int i=1; i <= n; ++i) sum += i;
  return sum;
}
```

is identical to the machine code generated for this function

```
int fghjkl(int qwerty) {
  int zxcv = 0;
  for (int asdf=1; asdf <= qwerty; ++asdf) zxcv += asdf;
  return zxcv;
}
```

One of the most significant differences between C and Racket is that C is *compiled*, while Racket is typically ***interpreted***.

An *interpreter* reads source code and "translates" it into machine code **while the program is running**. JavaScript and Python are popular languages that are typically interpreted.

Another approach that Racket supports is to compile source code into an intermediate language (*"bytecode"*) that is not machine specific. A *virtual machine* "translates" the bytecode into machine code while the program is running. Java and C# use this approach, which is faster than interpreting source code.

Compilation

There are three separate steps required to *compile* a C program.

- **preprocessing**
- **compilation**
- **linking**

In modern environments the steps are often *merged* together and simply referred to as “compiling”.

Preprocessing

In the preprocessing step the preprocessing *directives* are carried out (Section 03).

For example, the `#include` directive "cut and pastes" the contents of one file into another file.

The C preprocessor is not *strictly* part of the C language. Other languages can also use C preprocessor and support the # directives.

Compiling

In the compiling stage, each source code (`.c`) file is analyzed, checked for errors and then converted into an ***object code*** (`.o`) file.

Object code is **almost** complete machine code, except that many of the global identifiers (variable and function names) remain in the code as "placeholders", as their final addresses are still unknown.

An object file (`module.o`) includes:

- object code for all functions in `module.c`
- a list of all identifiers "provided" by `module.c`
- a list of all identifiers "required" by `module.c`

Linking

In the linking stage, all of the object files are combined and each global identifier is assigned an address. The final result is a single ***executable file***.

The *executable file* contains the **code** section as well as the contents of the **global data** and **read-only data** sections.

The ***linker*** also ensures that:

- all of the "required" identifiers are "provided" by a module
- there are no duplicate identifiers
- there is an entry point (*i.e.*, a `main` function)

The simplified view of **scope** (local/module/program) presented in this course is really a combination of:

- **scope:** *block scope* (local) or *file scope* (global)
- **storage:** *static storage* (*e.g.,* global or read-only memory) or *automatic storage* (stack section)
- **linkage:** *internal linkage* (when `static` is used for module scope) or *external linkage* (the default for a global is program scope) or *no linkage* (local variables)

See AP:AMA 18.2 for more details.

Command-line (shell) interface

To see compilation at work, we will first explore how to interact with an Operating System (OS) via the ***command-line***.

To start, launch a “Terminal” or similarly named application on your computer. A text-only window will appear with a “prompt” (*e.g.,* **`$`**).

You can launch programs directly from the command line.

For example, type **`date`** and press return (enter).

We will be providing examples in Linux, but Windows and Mac also have similar command line interfaces. There are numerous online guides available to help you.

Directory navigation

You are most likely familiar with file systems that contain directories (folders) and files organized in a “tree” structure.

At the command line, you are always “working” in one directory. This is also known as your “current” directory or the directory you are “in”.

pwd (print working directory) displays your current directory.

```
$ pwd
/u1/username
```

The full directory name is the ***path*** through the tree starting from the *root* (`/`) followed by each “sub-directory”, separated by `/`’s.

When you start the command-line, your current directory is likely your “home directory”.

`cd` (change directory) will return you to your home directory.

```
$ pwd
/somewhere/else
$ cd
/u1/username
```

Just like functions, programs can have *parameters* (although they are often *optional*). **`cd dirname`** will change your current directory.

```
$ pwd
/u1/username
$ cd /somewhere/else
$ pwd
/somewhere/else
```

The argument passed to **`cd`** can be a full *(absolute)* path (starting with the root **`/`**) or it can be a path *relative* to the current directory.

There are also three “special” directory names:

`.`	the current directory
`..`	the current directory’s parent in the tree (“one level up”)
`~`	your home directory

```
$ cd ~
$ pwd
/u1/username
$ cd ..
$ pwd
/u1
$ cd username         <-- relative path
$ pwd
/u1/username
```

The following commands are useful for working with files and navigating at the command-line.

`ls`	list the contents of the current directory
`mkdir d`	make a new directory **`d`**
`rmdir d`	remove an empty directory **`d`**
`cp a b`	make a copy the file **`a`** and call it **`b`**
`mv a b`	move (rename) file **`a`** and call it **`b`**
`rm a`	delete (remove) the file **`a`**
`cat a`	display the contents of the file **`a`**

A file name may also include the *path* to the file, which can be absolute (from the root) or relative to the current directory.

SSH

SSH (Secure SHell) allows you to use a command-line interface on a **remote** computer.

For example, to connect to your user account at Waterloo:

```
$ ssh username@linux.student.cs.uwaterloo.ca
```

In Windows, a popular (and free) SSH tool is known as PuTTY.

Text Editor

It is often useful to edit a text file in your terminal (or SSH) window, especially when you are connecting to a remote computer.

`Emacs` and **`vi`** (`vim`) are popular text editors and there is a long-standing friendly rivalry between users over which is better.

One of the easiest text editors for beginners is **`nano`**. To start using **`nano`**, you only need to remember two commands. To save (output) your file, press `(Ctrl-O)`, and to exit the editor, press `(Ctrl-X)`.

Create hello.c

1) Create a new folder and a new file:

```
$ mkdir cs136
$ cd cs136
$ nano hello.c
```

2) Type in the following program:

```
#include <stdio.h>

int main(void) {
  printf("Hello, World!\n");
}
```

3) `(Ctrl-O)` to save (press enter to confirm the file name) and `(Ctrl-X)` to exit.

```
$ ls
hello.c
```

gcc

We are now ready to *compile* and execute our program. The most popular C compiler is known as **`gcc`**.

```
$ gcc hello.c
$ ls
a.out hello.c
```

`gcc`'s default executable file name is `a.out`.

To execute it, we need to specify its path (the current folder `.`):

```
$ ./a.out
Hello, World!
```

In the `Seashell` environment we use `clang`, which is similar to gcc.

To specify the executable file name (instead of `a.out`), a *pair* of parameters is required. The first is **`-o`** (output) followed by the name.

```
$ gcc hello.c -o hello
$ ./hello
Hello, World!
```

Optional program parameters often start with a hyphen (**`-`**) and are known as options or “switches”. Options can modify the behaviour of the program (*e.g.,* the option **`-v`** makes **`gcc`** verbose and display additional information). Options like **`gcc`**’s **`-o`** (output) often require a second parameter.

The **`--help`** option often displays all of the options available.

`gcc` can generate object (`.o`) files by compiling (**`-c`**) and not linking.

```
$ gcc -c module1.c
$ ls
module1.c module1.o
```

This is really useful when distributing your modules to clients. The client can be provided with just the interface (`.h`) and the object (`.o`) file. The implementation details and source file (`.c`) can remain hidden from the client.

The default behaviour of **`gcc`** is to *link* (or combine) multiple module files (`.c` and `.o`) together.

```
$ gcc module1.o module2.c main.c -o program
```

Command-line arguments

We have seen how programs can have parameters, but we have not seen how to create a program that accepts parameters.

In Section 03 we described how the `main` function does not have any parameters, but that is not exactly true. They are optional.

```
int main(int argc, char *argv[]) {
  //...
}
```

`argv` is an array of strings, and `argc` is the length of the array.

The length of the array is always at least one, because `argv[0]` contains the name of the executable program itself. The number of parameters is `(argv - 1)`.

```c
int main(int argc, char *argv[]) {
  int num_param = argc - 1;
  if (num_param == 0) {
    printf("Hello, Stranger!\n");
  } else if (num_param == 1) {
    printf("Hello, %s!\n", argv[1]);
  } else {
    printf("Sorry, too many names.\n");
  }
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, Stranger!
$ ./hello Alice
Hello, Alice!
$ ./hello Bob
Hello, Bob!
$ ./hello Bob Smith
Sorry, too many names.
```

Streams

In Section 07 we discussed how programs can interact with the "real world" through input (*e.g.*, `scanf`) and output (*e.g.*, `printf`).

A popular programming abstraction is to represent I/O data as a ***stream*** of data that moves (or "flows") from a **source** to a **destination**.

A program can be both a destination (receives input) and a source (produces output).

The source/destination of a stream could be a device, a file, another program or another computer. The stream programming *interface* is the same, regardless of what the source/destination is.

Some programs connect to specific streams, but many programs use the *"standard"* input & output streams known as `stdin` & `stdout`. `scanf` reads from `stdin` and `printf` outputs to the `stdout` stream.

The default source for `stdin` is the keyboard, and the default destination for `stdout` is the "output window".

However, we can ***redirect*** (change) the standard streams to come from any source or go to any destination.

To test I/O, we will create a program that reads characters from `stdin` and then prints the reverse-case letters to `stdout`.

```c
// swapcase.c
#include <stdio.h>

int main(void) {
  char c;
  while(1) {
    if (scanf("%c", &c) != 1) break;
    if ((c >= 'a')&&(c <= 'z')) {
      c = c - 'a' + 'A';
    } else if ((c >= 'A')&&(c <= 'Z')) {
      c = c - 'A' + 'a';
    }
    printf("%c", c);
  }
}
```

Redirection

To *redirect* output **to** a file, the **>** symbol is used (*i.e.*, **> `filename`**).

```
$ ./hello > message.txt
$ cat message.txt
Hello, Stranger!
```

Above, the output is stored in a file named `message.txt` instead of displaying the output in the window.

To redirect input **from** a file, use the **<** symbol (*i.e.*, **< `filename`**).

```
$ ./swapcase < message.txt
hELLO, sTRANGER!
```

You can redirect input and output at the same time.

```
$ ./swapcase < message.txt > swapped.txt
$ cat swapped.txt
hELLO, sTRANGER!
```

To redirect directly to or from another **program**, it is known as ***piping***, and the pipe (**|**) symbol is used.

```
$ ./hello Bob | ./swapcase
hELLO, bOB!

$ ./hello DoubleSwap | ./swapcase | ./swapcase
Hello, DoubleSwap!
```

The Seashell environment

We can now understand all of the tasks that `Seashell` performs.

- scan the "run" file for `#include`s to determine the required modules, then compile and link all of the modules together
- if no `.in` files exist, execute the program, otherwise, for each `.in` file execute the program redirecting from `.in` files and to `.out` files

  ```
  $ ./program < mytest.in > mytest.out
  ```

- if `.expect` files exist, compare the `.out` files to the `.expect` files and store the differences in `.check` files

Full C language

We have skipped many C language features, including:

- `union`s and `enum`erations
- `int`eger and machine-specific types
- `switch`
- multi-dimensional arrays
- #`define` macros and other directives
- bit-wise operators and bit-fields
- advanced file I/O
- several C libraries (*e.g.*, `math.h`)

CS 246

The successor to this course is:

CS 246: Object-Oriented Software Development

- the C++ language
- object-oriented design and patterns
- tools (bash, svn, gdb, make)
- introduction to software engineering

Feedback welcome

Please send any corrections, feedback or suggestions to improve these course notes to:

Dave Tompkins

`dtompkins@uwaterloo.ca`

Good Luck on your final exams!