C for math folks

John Kerl

Department of Mathematics, University of Arizona

April 22, 2008

Outline

Why use C?

Why use C?

- To a first-order approximation: don't bother! Modern software such as Mathematica, Maple, Matlab, Python, etc. free you from the kinds of low-level details that C requires you to think about.
- But . . . every once in a while, primarily for performance reasons, you'll need to code all or part of an application in C.
- In my industry career, I found that the paradigm was: *prototype in Matlab* (the code is faster to write), *deploy in C* (the code runs faster). You may need C (or even Fortran!) for certain jobs, and/or when working with certain collaborators.
- C and C++ are the languages that higher-level languages are written in. True software wizardry (if that is your desire) requires mastery of C.

History

Unix and C were born the same year I was. Back in those days, sideburns and horn-rimmed glasses were really cool and people were walking on the moon. Also, the accepted wisdom was that whereas applications could be written in high-level languages like COBOL, Fortran, and PL/I, operating systems needed to be coded entirely in assembler.

Problem: each CPU architecture has its own completely different assembly language. This made porting operating systems a pain!

Solution: Brian Kernighan and Dennis Ritchie (immortalized ever after as "K&R") flew their freak flag and wrote a mid-level language which maps rather directly to machine language. It turns out 95% or so of the OS code *could* be written in C.

Much has changed in the almost 40 years since, but still, C is a mid-level language: portable, efficient, and plain.

What makes C different?

Years ago, one would say "Here's how Matlab is different from C." Now, in the 21st century, higher-level languages are often the first ones most of us encounter — and I find myself telling you how C is different from Matlab.

- C is a compiled language. Instead of just running your program, you must first have another program (the compiler) translate it into machine language. Then, you execute that program.
  Example: `python hello.py` (1 step) vs. `gcc -Wall -Werror hello.c -o hello; ./hello` (2 steps).
  (I always use the flags `-Wall` to enable all warnings and `-Werror` to treat warnings as errors — and so should you.)
- C is a strongly typed language. In Matlab, `x=[1.0:0.1:4.0]` or `x='abc'` or whatever. In C, you have to say whether `x` is an integer, floating-point number, etc.
- There are other differences which we'll see as we go along.

First example

demo1.c

Here's the obligatory first example:

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

To run it, do

```
gcc -Wall -Werror demo1.c -o demo1
./demo1
```

Functions

- There are those darned header files (e.g. `stdio.h`)! We'll soon see why they matter.
- Execution starts at `main`. Command-line arguments may be passed in; see below.
- The syntax for a function is *output-type function-name* (*arguments ...*) { *body ...* }. The keyword `void` means no arguments. Subroutines, i.e. functions with no return value, should have return type `void`.
- Return 0 from `main` to signal successful completion back to the shell. (You can return 1 for an abnormal exit.) When you start scripting multiple invocations of your program, you may find that this is a good habit.
- There are bloody holy wars about indentation style, which other authors will engage in. I won't comment. As long as you don't reformat my perfect, beautiful code, we're at peace.

Second example

demo2.c

Even this simple example contains much of what you need to know:

```
#include <stdio.h>  // For the printf prototype
#include <string.h> // For the sscanf prototype
#include <stdlib.h> // For the exit prototype
#include <math.h>   // For the sin prototype

int main(int argc, char ** argv)
{
    int n = 40;
    int i;
    float x, y;

    if (argc == 2) {
        if (sscanf(argv[1], "n=%d", &n) != 1) {
            fprintf(stderr, "Usage: %s [n=...]\n", argv[0]);
            exit(1);
        }
    }

    for (i = 0; i < n; i++) { // Here is a comment.
        x = (float)i/n * 2.0 * M_PI;
        y = sin(x);
        printf("%3d %11.7f %11.7f\n", i, x, y);
    }

    return 0;
}
```

Command-line arguments; control blocks

- I'll tell you about header files (`#include ...`) soon, I promise!
- Command-line arguments are passed from the shell into your C program through the `argv` array. The program name is always `argv[0]`. If you type `./demo2 xyz pdq` at the shell prompt, then the C program gets `argc=3`, with `argv[0]="./demo2"`, `argv[1]="xyz"`, and `argv[2]="pdq"`. How you handle command-line parsing is completely up to you — including determining whether argument strings are valid string representations of integers, etc.
- Beginnings and endings of control blocks are done with curly braces. This is much cleaner than `begin` and `end` keywords. (Whitespace control structure, e.g. in Python, is even cleaner!)

Comments

All variables are typed, and must be declared at the top of the function. The main types are as follows (remember 8 bits are one byte):

- `int`: signed integer, usually 32-bit i.e. $-2^{31} \leq n < 2^{31}$.
- `unsigned`: unsigned integer, also usually 32-bit, so $0 \leq n < 2^{32}$.
- `float`: single-precision IEEE float, 32 bits: about 7 sigfigs.
- `double`: double-precision IEEE float, 64 bits: about 13 sigfigs.
- `char`: single characters, 8 bits.
- Zero-based arrays of any type, e.g. `int x[10]` has elements `x[0]` through `x[9]`.
- Pointers to variables or starts of arrays, e.g. `int * px = &n` or `int * px = &x[0]`. The ampersand (`&`) is the address-of operator which gives the location of an item in RAM.
- One of the most powerful features of C is the `struct`: compound data types. There's no time to talk about them today, alas.
- I won't talk about `short` or `long` either, let alone `long long`.

Bits and bytes

- Why do we care about the number of bits anyway? Higher-level languages don't ask me to care — why should C?

  Python:

```
print 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15
1307674368000
```

  C:

```
int x = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15;
printf("%d\n", x);
2004310016
```

  What happened? *32-bit arithmetic is done mod* 2^{32}. Surprise! There are arbitrary-precision subroutine libraries, but they're add-ons. (Batteries not included.)

Mixed types; `for` loops

- Why did I type `(float)i/n` instead of `i/n`? In fact, what's $7/4$? Not 1.75 but 1. (Another surprise!) Integer division is quotient and remainder. If you want a floating-point quotient of integers, cast one or both operands to float first. (`int` *operand* `int` is `int`; `int` *operand* `float` is `float`. This is called *promotion*.)
- C has a wonderfully flexible for-loop structure:

```
for (initial statement(s); continuation test; update statement(s)) {
    body ...
    body ...
    body ...
}
```

which means

```
initial statement(s);
while (contiuation test is true) {
    body ...
    body ...
    body ...
    update statement(s);
}
```

Input from strings

- The `sscanf` routine reads strings and stores values into specified variables. It returns the number of variables successfully scanned. Here, I typed

```
if (sscanf(argv[1], "n=%d", &n) != 1) {
    fprintf(stderr, "Usage: %s [n=...]\n", argv[0]);
    exit(1);
}
```

  The `sscanf` routine takes `argv[1]` which the user typed in via `./demo2 n=10`. It sees if `argv[1]` starts with `n=`, followed by numerical characters. If so, it stuffs the resulting number (here, 10) into `n` and returns 1. Else, the value of `n` is undefined and `sscanf` returns 0.

Formatted output

- Formatted I/O consists of formatted input (e.g. `sscanf` to scan from string and `fscanf` to scan from a file) and formatted output (e.g. `printf`).
- Unlike higher-level languages, where a single `print` or `disp` routine will print out anything you throw its way, in C you need to use format specifiers (e.g. `%3d`) which depend on the data type being printed out.
- The main format types are `d`, `f`, and `lf` for integer, float, and double, respectively. Also `e` and `le` to print floats/doubles in scientific notation.
- In between the `%` and the format type are field widths: e.g. `%4d` for making integers take up four spaces, and `%11.7f` for 11 characters wide with 7 of those characters after the decimal point.

Libraries and linking

- Compile this with `gcc -Wall -Werror demo2.c -o demo2` and you get:

```
/tmp/ccoFJT8z.o: In function ‘main’:
demo2.c:(.text+0xa7): undefined reference to ‘sin’
collect2: ld returned 1 exit status
```

- Compile instead with `gcc -Wall -Werror demo2.c -o demo2 -lm` and it works. The problem is that `sin` is in the math library. In standard batteries-not-included fashion ("Oh, you wanted to do *math?* All righty then."), the C linker only links in libraries you specify via `-l`*name*. The exception is the so-called C library, with things like `printf`, `exit`, etc.

The result

```
./demo2 10
Usage: ./demo2 [n=...]
./demo2 n=10
  0   0.0000000   0.0000000
  1   0.6283185   0.5877852
  2   1.2566371   0.9510565
  3   1.8849556   0.9510565
  4   2.5132742   0.5877852
  5   3.1415927  -0.0000001
  6   3.7699113  -0.5877854
  7   4.3982296  -0.9510565
  8   5.0265484  -0.9510565
  9   5.6548667  -0.5877853
```

Third example

Multiple files and subroutines

Here is the file `demo3.c`.

```
#include <stdio.h>  // For the printf prototype
#include <string.h> // For the sscanf prototype
#include <stdlib.h> // For the exit prototype
#include <math.h>   // For the sin prototype
#include "my_stuff.h" // For the my_function prototype

int main(int argc, char ** argv)
{
    int n = 12;
    int i;
    float c = 2.7;
    float x, y;

    for (i = 0; i < n; i++) {
        x = (float)i/n * 4.0;
        y = my_function(x, c); // Mystery function ...
        printf("%3d %11.7f %11.7f\n", i, x, y);
    }

    return 0;
}
```

Multiple files and subroutines

Here is the file `my_stuff.h` ...

```
#ifndef MY_STUFF_H
#define MY_STUFF_H
#define MY_CONSTANT 2.0
// Computes x to the c power.
float my_function(float x, float c);
#endif //MY_STUFF_H
```

... and here's `my_stuff.c`.

```
#include <math.h>
#include "my_stuff.h"
// Computes x to the c power.
float my_function(float x, float c)
{
    return pow(x, c);
}
```

Note (huge bummer for math folks!!) that the C language does not have an exponentiation operator: `**` has to do with pointers to pointers, and `^` is a bitwise XOR. For squaring you can do `x*x`, and so on, but in general you need to call the `pow` function.

Output

```
./demo3
  0    0.0000000    0.0000000
  1    0.3333333    0.0514959
  2    0.6666667    0.3346213
  3    1.0000000    1.0000000
  4    1.3333334    2.1743760
  5    1.6666666    3.9718387
  6    2.0000000    6.4980192
  7    2.3333333    9.8522921
  8    2.6666667   14.1291370
  9    3.0000000   19.4190254
 10    3.3333333   25.8090858
 11    3.6666667   33.3835793
```

Headers

- Always put the `#ifdef` and `#endif` jazz at the top and bottom of your headers, so you don't get "multiply defined ..." errors when headers are included more than once (perhaps indirectly — headers can include other headers!).
- The compiler doesn't like you calling a function it doesn't already know something about. It wants to have seen the prototype

  ```
  float my_function(float x, float c);
  ```

  before it sees you calling the function:

  ```
  y = my_function(x, c); // Mystery function ...
  ```

  This way it knows the data types of the arguments and the return value.

Headers

Header files (i.e. `.h`) files contain principally two things:

(1) Function prototypes, e.g. `math.h` has something like

```
double sin(double x);
```

(2) Constant definitions, e.g.

```
# define M_PI 3.14159265358979323846 /* pi */
```

System header files are typically in `/usr/include` and are included with `#include <...>`. Your header files are typically in `.` and are included with `#include "..."`. If you have headers in another directory, e.g. `../project2`, then compile with `-I../project2` to let the C preprocessor know where to look.

Names of system headers

Question: OK, so this is really annoying! If I'm calling `sin`, `sscanf`, `exit`, etc., how do I know I need to include `math.h`, `string.h`, `stdlib.h`? Isn't this some sort of sadistic guessing game?!?

Answer: Perhaps so! But you can type `man sscanf` etc. at the command prompt. Manpages generally do a good job of telling you which header file contains the prototype for the function you're interested in.

Third example

- Compile this with `gcc -Wall -Werror demo3.c -o demo3` and you get:

```
/tmp/ccCMgV6x.o: In function ‘main’:
demo3.c:(.text+0x4a): undefined reference to
‘my_function’
collect2: ld returned 1 exit status
```

- Compile instead with

```
gcc -Wall -Werror demo3.c my_stuff.c -o demo3 -lm
```

- Question: If I only made changes to `my_stuff.c` and not `demo3.c`, do I really need to recompile both of them? Answer: Certainly not! This is what makefiles are for. Which is another topic . . . .

More

More

- There's a lot more:
  - Pointers
  - Passing arguments by reference
  - Structs
  - Dynamic allocation
  - Reading input from files
  - Interfacing with Matlab
  - Makefiles
  - ...
- Google for "C tutorial" and go nuts!