Computer Programming: Skills & Concepts (CP)
Giving Arguments to Programs;
What is there still to C?

Julian Bradfield

Monday 27 November 2017

main()

So far, we've been telling you that the type of the main function is `int main(void)`.
This is a simplification! The actual type is:
`int main(int argc, char *argv[], char *envp[])`
and what people usually use is
`int main(int argc, char *argv[])`

The arguments of `main()` are the way we pass arguments to programs, as when you do `cp a.out myprog`.
In Unix (and many other) systems, a program is called with any number of arguments, each of which is a string.
By convention, the first (zero'th!) argument is the name of the program (e.g. `cp`), and the remaining arguments are what the user sees as arguments (`a.out` and `myprog`).
When your program is called, `argc` is the number of arguments (including the zero'th), and `argv[i]` is the *i*'th argument string.

Argument Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i;
  printf("There are %d arguments. They are:\n",argc);
  for (i=0; i<argc; i++) {
    printf("Arg %d is \"%s\"\n",i,argv[i]);
  }
  return EXIT_SUCCESS;
}
```

```
pula: gcc arg.c
pula: ./a.out first second "third argument" and\ fourth '"quotes"'
There are 6 arguments. They are:
Arg 0 is "./a.out"
Arg 1 is "first"
Arg 2 is "second"
Arg 3 is "third argument"
Arg 4 is "and fourth"
Arg 5 is ""quotes""
```

What about `envp`? This is the array of 'environment variables' that are used as general way to make information available to all programs. The usual way to get at the environment is with `getenv()` and `putenv()` (see their manual pages), so we never usually even notice the existence of `envp`.

Things we haven’t covered

C is a fairly large language, and it also evolves.
We have been teaching ‘C89’, which is supported by all current compilers.
Officially, C89 is obsolete, having been succeeded by ‘C99’. But C99 is not well supported.
Officially, C99 is obsolete, having been succeeded by ‘C11’. But ...
In the open source world, everybody uses gcc, which supports some bits of C99 and a number of extensions of its own. Two of these are so common that I’ve accidentally used them in your code:

- In C89, variable declarations must come before the code (in each block), but in GNU C (and C99) you can mix them.
- GNU C and C99 allow single line comments starting with `//`

The remaining slides list (for personal study) the main features of C89 that we have not covered in this course, concentrating on ones you can expect to see in any program off the Web.

Writing constants

In addition to the decimal integers and floating point number in point and scientific notation:
Hexadecimal integers: `0x123FAB78`
Octal integers: `0123` is 83 decimal. (Beware of this!)
and even hexadecimal floating point.
You can suffix numbers to say that they are `unsigned` or `long` or `long long`:
`50U`   `1234567890L`   `123456789763222LLU`
There are various ways to write weird characters.

Bitwise operators

In low level programming, it’s often useful to use an `int` as a collection of binary *bits* in which we can store boolean flags (or even several very small integers).
C provides the following operators for working bit-by-bit on integers:
`a & b` take the logical AND, bit-by-bit, of `a` and `b`
`a | b` take the logical OR, bit-by-bit, of `a` and `b`
`a ^ b` take the logical XOR, bit-by-bit, of `a` and `b`
`~b` take the logical NOT, bit-by-bit, of `b`
`a << b` shift the bits of `a` left by `b` places
`a >> b` shift the bits of `a` right by `b` places

```
unsigned int flags;
enum { FLAG1 = 1, FLAG2 = 1 << 1, FLAG3 = 1 << 2, FLAG4 = 1 << 3 };

flags |= FLAG3; /* set the FLAG3 bit of flags */
flags &= ~FLAG2; /* clear the FLAG2 bit of flags */
if ( flags & FLAG4 ) ... /* test the FLAG4 bit of flags */
```

Conditional and comma operators

The conditional *expression* `a ? b : c` means ‘if `a`, then value of `b`, else value of `c`’.
Widely used to save typing, e.g.:
`printf(isVowel(x) ? "V" : "C");`
instead of

```
if ( isVowel(x) ) {
  printf("V");
} else {
  printf("C");
}
```

The comma expression `a, b` means evaluate `a`, then return value of `b`.
(In my experience) mostly used by mistake! For example: `(1,2,3,4,5)` is legal C, but just evaluates to 5, and is not an array or list.

static

The keyword `static` has two completely different meanings.
With functions and global variables, it means ‘not externally visible at link time’:

```
int global_option; /* can be used with
                      extern int global_option;
                      by other compiled modules */
static int module_option; /* only visible in this .c file */
```

With local variables, `static` means ‘this variable is created just once and re-used by every call to this function’:

```
int counter(void) {
  static int count = 0;
  return ++count;
}
```

returns the number of times it has been called.

do ... while

The `do ... while` construct is a while-loop that has the test at the end instead of the beginning. So it always executes at least once. Occasionally convenient if the test needs something computed by the loop body to make sense.

```
char c;
do {
  printf("Answer y or n: ");
  scanf(" %c",&c);
} while ( ! ( c == 'y' || c == 'n' ) );
```

break and continue

We have seen `break` in the `switch` statement.
`break` can be used to terminate the current `while` or `for` loop early.
Typical usage:

```
while ( usual condition ) {
  if ( something unusual ) {
    printf("something odd happened, can't continue");
    break;
  }
  normal processing
}
```

`continue` skips to the end of the loop and then goes round again (doing the increment in a `for` loop). Sometimes used to skip the rest of the loop after doing an easy case.

goto

Academic courses try not to admit it, but C does have `goto`. I’ve used it once or twice. Typical usage to break out of multiply nested loops:

```
int blurgle(void) {
  while ( ... ) {
    ...
    while ( ... ) {
      ...
      if ( catastrophe ) {
        printf("Aaarrgh!\n");
        goto clean_and_return;
      }
      ...
    }
  }
clean_and_return:
  clean up memory etc.
  return ret;
}
```

Some consider this the only acceptable use of goto. Many say there is NO acceptable use.

Union types

A union looks like a struct, but puts all the members in the same place. Used for making ‘umbrella types’ which can hold any of several other types.

```
typedef struct triangle { ... } triangle_t;
typedef struct circle { ... } circle_t;
typedef union shape {
  triangle_t tri;
  circle_t cir;
} shape_t;
shape_t s;
```

Now `s` can hold a triangle `s.tri` *or* a circle `s.cir` (but *not* both at the same time).

Preprocessor macros

We’ve seen `#define` for defining *compile-time* constants.
We can also define *macros*, which are like compile-time functions applied to your program: the body code is *textually* substituted for the macro.

```
#define max(a,b) (a > b ? a : b)

x = max(42,y*3);
```

the actual code seen by the compiler is

```
x = (42 > y*3 ? 42 : y*3);
```

People used to do this to avoid expensive function calls, but there are better ways. Nowadays macros are mostly used in conjunction with . . .

Conditional compilation

The preprocessor has ‘if’ statements, which allow you to change the code to be compiled according to conditions on preprocessor constants.
Suppose a system header file does
`#define Win32`
on Windows systems, and not on other systems. Then you can write;

```
int grobble_in_system(messy_data_t arg) {
#ifdef Win32
horrible code for Windows
#else
sane code for Unix
#endif
}
```

Where next?

C is large, but not too large. Learning how to use all the libraries is the hardest part.
Java is very popular. It’s inherited most of C’s basic syntax, but thrown away the really horrible bits. Java is much cleaner – none of this messing with pointers, or arrays of chars. It uses the Object Oriented paradigm, which is a quite a learning hurdle. It’s also slow.
C++ is C with Object Oriented programming bolted on top. It has all the yuck-factor of C, with all the difficulty of the OO approach. Big companies like it, as it gives the design power of OO combined with the speed of C, and it has a huge standard library. I don’t speak it!
Perl is what much of the Web used to be written in. It’s vaguely C-like, but heavily optimized for everyday text manipulation tasks. Its biggest problem is how easy it is to have bugs, and how hard it is to find them. (I use it for the automarkers.)
Python is rather like Perl, but cleaner, and OO. It’s popular at present.