C Intro Notes #1

Paul Johnson

Feb 28, 2005

1 Read some book, do some exercises

No matter how much you watch me “blather on and on,” you don’t learn anything unless you do exercises, make mistakes, try again.

2 Variables types

C is a “strongly typed” language. That means variables are declared by specific types.

2.1 Integers:

The number of bits assigned for each type depends on the kind of CPU and the C library you are using, these are usually about right.

short 16 bits,

int 32 bits, range of values: -32,768 to 32,767

long

unsigned int, unsigned long

2.2 Real-valued variables

float 32 bits

double 64 bits

2.3 Characters

char

unsigned char

Comment: handling of “strings” in C is a pain and you don’t really understand it until you practice.

2.4 Declarations allowed only in the “Top of Blocks”

A variable exists only within its “scope.” A scope is a somewhat abstract term we should talk over.

All variables have to be defined at the top of blocks, and the top of a block is the beginning of a scope. You an do

```
{
  int x;
  int y;
  double z;
  x = 2 * x;
}
```

But you must NOT do some calculations, and then declare another variable

```
{
  int x;
  int y;
  x = 2 * x;
  double z;
  z = x * y;
}
```

2.5 Take the easy route.

Don’t bother over variable types. Declare your integers as “int” and your real-valued variables as “double”. If you run out of memory, buy more.

2.6 Cast one variable as another.

In C, you might need to do some math. And it gets ugly, especially if you divide 2 integers. It does not give you back a floating point number, as you expect. It rounds off. So you can force division to act as if 2 integers are floats:

```
int x=3;
int y=3;
double z;
z = (double)x/(double)y;
```

When you use the “cast” to “promote” or “demote” a variable, you tell the compiler to do its best to transfer the thing from one type to another. A cast like this

```
x = (int)z;
```

can, on most systems, do a “rounding down” of z to the integer value. Its not always predictable, and there are other functions for rounding.

3 Printing to the screen

3.1 fprintf is it!

I’ve tried to shift from using

```
printf(“hello, this is a message to the screen”);
```

to

```
fprintf(stderr,”hello, this is a message to the screen”);
```

I do that because “stderr” is the “standard error” stream of messages that go to the screen and this style encourages the program to send the message right away.

3.2 Access variables from fprintf

```
int x = 7;
double y = 7.7;
fprintf(stderr,”the value of an int is %d \n”, x);
/* note %d means “an integer” follows */
fprintf(stderr,”the value of a double is %f \n”,y);
fprintf(stderr,”x=%d,y=%f \n”,x,y);
```

4 Control structures

4.1 Special use of = and & and |

Please beware that the & is a dangerous thing because it is used in "bitwise" computing. You might use it thinking it measn "and". It is a mistake. In C,

== means "equal to" in a conditional statement

&& means "AND"

|| means "OR"

4.2 conditional statements

```
int x, y, z;
if ( x == 1 )
{
   y = 2;
}
else
{
   y = 5;
}
if ((x == 1) && (y == 2))
{
   z = 10;
}
```

4.3 for loops

This is the predominant method of iterating through some problem.

```
int i;
for (i = 0; i<N; i++)
{
   fprintf(stderr, "this is step %d",i);
}
```

4.4 while loops

Sometimes while loops come in handy, but for me it is rare.

5 Arrays

An array is a homogeneous collection of variable values. Any of the fundamental types can be elements in an array.

```
int blop[10]; /*this is an array called “blop” */
```

You can initialize like so:

```
int blop[10]={1,2,3,4,5,6,7,8,9,10}
```

Or like this

```
int i;
for (i=0, i<10; i++)
{
   blop[i] = i+1;
}
```

Caution: In C, counting begins with 0 and goes up to N-1.

Caution: C does not do “bounds checking”

If your array has 10 items, and you ask for the 11th one, you get back nonsense. Your program won’t necessarily crash, but it might, depending if the system can “make sense” of what it finds.

6 C allows “macros”

A macro is a convenience that is loved by some and hated vigorously by others.

The custom is that macros should be CAPITALIZED. This is done to avoid accidental usage of names.

Macros work because C has a “preprocessor” that goes over your code and replaces symbols with others.

Example. Suppose you define a macro M_PI to hold the value of π.

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

In your code, you could use that value wherever you wanted.

```
if ( x > M_PI)
{
  fprintf(stderr,"x is greater than pi \n");
}
```

The pre-processor would replace the letters M_PI with the desired value.

Macros are often used to mark of big sections of code that are ignored or included.

```
#define WHATEVER 1
/* that set the value to 1, which the system sees as true*/
#ifdef WHATEVER
/*if WHATEVER is true, do this */
#endif
```

You can also make a conditional that includes unless a macro is defined.

```
#ifndef WHATEVER
#endif
```

Why would anybody want to do this? Well, if you want a fast program, and don't need some elements at all, then you make the pre-processor leave them out altogether.

7 functions in C

7.1 single-valued return

In C, a function can take input arguments and give back a single value.

```
double myFunction ( int x, double y)
{
   /* x y are "input variables" */
   double z = (double)x * y;
   return z;
}
```

Please note, inside a function you can declare more "local variables" but you must not use the same names as the inputs

If you don't want a return value from a function, you can give it the type "void" and leave out the return;

7.2 Yes, I mean you can’t return an array

7.3 But you can return a “pointer” to an array.

8 What’s a pointer.

This is where all hell breaks loose.

8.1 Static versus Dynamic memory

Programs can automatically allocate memory, but only up to a point. If you need an array that holds “too much stuff,” then you can’t count on the program to handle it. SO a declaration like this

```
    int x[1000000000000000000];
```

will be a non-starter.

If an array that big is even possible, you have to use dynamically allocated memory. To do that, first declare a “pointer”:

```
    int *x;
```

and then you use a command like “malloc” or “alloc” to claim a block of memory from the computer.

I’m not writing down the malloc or alloc command here because you won’t usually need to do that. In Swarm, there’s an alloc option I use instead.

8.2 A pointer is the “first position” of a piece of memory.

```
    int *x;
    int i;
    x = allocate this for N items, however you want
    x[0] = 3; puts the value of 3 into the first position.
    for (i = 0; i< N; i++)
    {
       x[i] = i*2;
    }
    Note an array works like a pointer.
```

8.3 There is other stuff worth knowing about pointers

but not a hell of a lot, once you move into using Swarm.

9 Include, import, header files.

You can write a big, monstrous program all in one file if you want to. But it gets tough to manage and confusing. So you declare a file in 2 parts, the

1. header file, such as “MyFile.h”
2. implementation file, such as “MyFile.c” in c, “MyFile.cc” in C++, or “MyFile.m” in Objective C.

The “include” statement at the top of a file gives that file access to commands that are defined in the included file.

Most C programs have

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

That’s where printf and other basic features are defined.

Look in /usr/include in your file system to see many *.h files.