C Intro Notes

Paul Johnson

Feb 2011

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.

When integers were 16 bits, they could hold signed values in the range (-32,768 to 32,767)

long int (64 bits)

unsigned int

unsigned long

On newer 64bit CPU, an int will default to be 64 bits.

From the GNU C Tutorial:

Type	Bits on 64 bit system	min	max
char	8	-127	127
unsigned char	8	0	255
short		-32,768	32,768
unsigned short	16	0	65,535
int	32	-2,147,483,647	2,147,483,647
long	32(win) or 64(linux mac)	↑↓	↑↓
long long	64	-9,223,372,036,854,775,807	9,223,372,036,854,775,807
unsigned long long	64	0	18,446,744,073,709,551,615

May be difficult to write portable code because systems vary.

Check your system ("/usr/include/limits.h") for integer ranges.

2.2 Real-valued variables

Type	Bits on 64 bit system	min	max
float	32 (~7 digit precision)	-3.4e-38	3.4e+38
double	64 (~15 digit precision)	-4.940656458412e-324	1.79769313486231570e+308
long double	64	same	

Tidbits:

e-38 means 10^{-38} or $\frac{1}{10^{38}}$; e+38 means 10^{38}

A 64 bit floating point number will usually use

- 11 bits for exponent
- 53 bits for the coefficient (aka "significand", about 16 decimal digits)

A 32 bit floating point will allocate

- 8 bits for exponent
- 24 bits for significand, (about 7 decimal digits)

Single or Double precision is a matter of how many bits are used and maintained throughout a calculation

These are system dependent. AIX Unix has 128 bit long double.

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.

3 Scope is defined by squiggly braces

3.1 Style

I tend to prefer the vertically parallel placement of squiggly braces, as in:

```
int myfunction (int y)
  {
     //the body of the function goes here
  }
```

Other people, especially publishers who want to save paper, may prefer:

```
int myfunction (int y){
     //the body of the function goes here
  }
```

People like me prefer if/then like this

```
if (x == 0)
  {
     //some consequence
  }
else
  {
  //another consequence
  }
```

But some people, including the original designers of C, prefer this:

```
if (x == 0) {
     //some consequence
  } else {
  //another consequence
  }
```

As long as you try to be consistent, most people will not get too upset at you about your code.

I tend to write wasteful characters in order to prevent mistakes. For example, I don't mind writing

```
if (x == 0) { y = 2; }
```

even though the squiggly braces are not required.

3.2 Scope

Scope has to do with

- where a piece of code "looks for" values.
- the impact of changes in a variable.

Ordinarily, the effects of changes are confined to the "braces" in which the variable is defined.

A variable exists only within its "scope."

```
{ Scope A
    {   Scope B
        { Scope C }
        { Scope D }
    }
}
```

Variables and functions declared in A are available to B, C, and D.

Variables declared in C are not available in D or B or A.

3.3 Declarations allowed only in the "Top of Blocks"

Even though some compilers may not insist on this, some will. So keep it simple, follow the tradition.

All variables have to be defined at the top of blocks. 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;
}
```

3.4 Take the easy route, unless there is good reason go to the other way.

Declare your integers as "int."

Declare your real-valued variables as "double". If you run out of memory, buy more.

3.5 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.

4 Printing to the screen

4.1 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.

4.2 Access variables from printf or fprintf

C uses a Fortran-like method of declaring "place holders" for material to be printed out.

This is a 'counting game". Match the number of "%" to the number of variables.

Syntax is

```
printf("something with %d or %f or more", oneIntegerVariable,
   oneFloatingVariable);
```

C allows lots of formats, but I mostly remember using

Variable Type	printf abbreviation	
integer	%d or %i	
floating point (6 decimals)	%f	
floating (restrict to 3 digits before decimal, 2 after)	%3.2f	

```
int x = 7;
double y = 7.7;
double z = (double)2/3;
fprintf(stderr,"the value of an int is %d \n", x);
/* note %d means "an integer" follows to fill that in */
fprintf(stderr,"the value of a double is %f \n", y);
fprintf(stderr,"x=%d,y=%f, z=%f, z=%3.4, \n", x, y, z, z);
```

5 Control structures

5.1 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 means "and". It is a mistake. In C,

== means "equal to" in a conditional statement

&& means "AND"

|| means "OR"

5.2 conditional statements

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

Note: Squiggly braces not required on one liners

The following are equivalent:

```
if (x == 1) { z = 5; }
```

And

```
if (x == 1) z = 5;
```

In other words, when there is ONLY ONE consequence, the braces are not required.

BUT, guess how many times I've made this mistake:

Suppose we start with:

```
if (x == 1) z = 5;
```

We want to add an additional option, and forget to add the squiggly braces:

```
if (x == 1) z = 5;
            z2 = 10;
```

But we really need:

```
if (x == 1)
  {
  z = 5;
  z2 = 10;
  }
```

is instead this mistake:

```
if (x == 1) z = 5;
```

5.3 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);
}
```

5.4 while loops

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

6 Struct

The term “struct” in C refers to a collection of variables of different types. In Pascal, this is called a “record.”

6.1 Define a structure

Collect up several variables or arrays and group them together.

```
struct student{
   int id;
   char *first_name;
   char *last_name;
   double age;
   double gpa;
};
```

That declares a structure of variables. I think of that as a variable of type “struct student”. So where I would put “int” or “double” or “double *” in a function, I now put “struct student” or “struct student *”.

A variable of type “struct student” or “struct student *” can be either the input OR the output from a function.

6.2 Create one struct variable.

Now we have to create one student. Since struct student is already defined, we could simply do this:

```
struct student myStudent1;
```

I should mention, when the struct is defined, we can add names of structs to be created on the end of the syntax

```
struct student{
   int id;
   char *first_name;
   char *last_name;
   double age;
   double gpa;
} myStudent1;
```

A struct declared in this way is "automatically" allocated memory from the stack. Like all variables, it lives within a scope. To pass structs among function, we need to make them pointer variables, that have memory allocated from the heap.

6.3 Accessing values from a struct

6.3.1 The "." operator.

A period joins together the name of the struct with the variable inside that is to be accessed.

```
myStudent1.first_name = "Willie";
myStudent1.age = 32;
myStudent1.gpa = 3.1223;
printf("This student's age is %d and the gpa is %f\n", myStudent1.age,
   myStudent1.gpa);
```

6.3.2 The "->" operator.

Suppose we have a pointer to a struct.

```
struct student * myStudent2;
```

To "get" values for that student, we'd first have to de-reference the pointer, meaning "go to the memory location * myStudent2.

```
*myStudent2
```

and then to get one particular variable out of that dereferenced value, we have to parenthesize that and add period and a variable name:

```
(*myStudent2).age;
```

That notation is a bit cumbersome, so a special symbol was introduced to refer to de-referencing a struct pointer and then taking a variable from it.

```
myStudent2->age;
```

It is required that myStudent2 must be a pointer to a struct and age, of course, must be one of the variables in the struct. This makes our student younger.

```
void makeYounger(struct student* st)
   {
    st->age = 0.75 * st->age;
   }
```

7 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’’ */
```

- This array is “statically” declared.
- It exists within the scope where it is created.
- We can read elements like so:

  ```
  blop[5]
  ```

- We can set values like so:

  ```
  blop[5]=32;
  ```

- The elements in an array are indexed from 0 to $N-1$. So the array is

  ```
  blop={blop[0],blop[1],blop[2],blop[3],blop[4],blop[5], blop[6],blop
      [7],blop[8],blop[9]}
  ```

You can initialize at time of declaration like so:

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

Otherwise necessary to individually assign all elements, like this

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

Caution: It is NOT ALLOWED to simply initialize blop=0; Must loop over each spot.

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

Caution: C does not do “bounds checking” automatically. Reference to ’blop[10]’ is bad. Very bad. 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.

8 Include, import, header files.

Terminology

"interface" function names, how functions are accessed from "outside".

> **interface** is stuff in *.h files in C.

"implementation" The actual code that does the work.

> **implementation** is self contained: the only user interaction it requires is in the input variables.
>
> **implementation** should be well done, clear, and forgettable (it 'Just works')!
>
> User should not need to dig into implementation code in order to use the function.

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.

9 C allows "macros"

CPP

CPP: the "pre-processor". Before a program is compiled, gcc runs it through the C pre-processor. One duty of the pre-processor is to inspect for certain "magic words" and then replace them with something else.

The C library has a number of pre-defined variables, and if you put their magic words in your program, the pre-processor comes along and replaces those words with their values. Check "/usr/include/limits.h", for example, where this line is found

```
# define LONG_MAX 9223372036854775807L
```

As a result, if your code has the letters “LONG_MAX” in it, the pre-processor will replace those letters with the number from limits.h. As a result, one can do

```
x = LONG_MAX
```

and CPP will figure out what is legal for your system.

A major benefit of this is improved code protability, because the system’s configuration is likely to know what LONG_MAX is, but the programmer might not.

User Defined Macros

User customized macros. This ability is a convenience that is loved by some and hated vigorously by others. It was hated so vigorously by the designers of Java that they eliminated the user’s ability to create macros at all.

The custom is that macros should be CAPITALIZED. This helps to prevent accidental usage of macro variables. Remember, macros work because C has a “preprocessor” that goes over your code and replaces symbols with the desired values. This is a very powerful and dangerous thing.

Example. Suppose you have a special value of π. Put this at the top of your code MY_PI:

```
# define MY_PI 3.3 /* my pie is rounded */
```

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

```
if ( x > MY_PI)
{
  fprintf(stderr,x is greater than %f \n, MY_PI);
}
```

The pre-processor would replace the letters M_PI with 3.3.

Use Macros While Debugging Code

Some programmers develop new procedures in the middle of if statements.

```
if ( 0 ){
   // a lot of C code that we were using before
}
else {
   C code we are creating in our new version
}
```

The if/else approach works, but it does waste some program time, because the if/else has to be computed whenever the program runs.

What if you could just make the first part irrelevant to the program from the start, as if it were deleted?

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

```
#define DEBUG 1
/* that set the value to 1, which the system sees as true */
```

Then, later in the code, if you want something include for debugging, wrap the code in a "macro sandwich," with "#ifdef DEBUG" at the beginning and "#endif" at the end.

```
#ifdef DEBUG
 C code that you want to run if DEBUG is 1
#endif
```

Or, if you want something to run when DEBUG is NOT true, do this

```
#ifndef DEBUG
x = 92;
y = exp(z);
#endif
```

Why would anybody want to do this?

1. Testing new sub-routines

2. Avoid unnecessary calculations. If you don't need some elements at all, then you make the pre-processor leave them out altogether.

10 Typedef.

10.1 Typedef: Create new variable types in your code

Suppose you find yourself creating the same kind of array variable over and over.

int x[3];

int y[3];

int z[3];

One simplification is to define a new variable type that is a 3-valued integer array.

typedef int myarray [3];

After that, the new variable type "myarray" is created, so a declaration like this

myarray x;

has the effect of creating a 3 valued array, and you can proceed to use it in the ordinary way:

x[0]=7; x[1]=2, x[2]=7;

Typedef can be used to simplify declarations of many variable types, including pointers and structs.

10.2 Typedef: Almost like a macro.

Suppose you want to call all of your integers as type "CHARLIE". You could use a macro

```
#define CHARLIE int
```

Then when you want to declare an int, then instead of

```
int x=99;
```

then type this:

```
CHARLIE x;
```

Some programmers will do that because they want to leave the possibility for customizing a variable type. Suppose we might want a “short int”, “int”, “long int”, but we don’t know which. So a macro will be a convenient way to say “I’ll set that later”. Later we could come back and change one line to convert all of the “ints” to “longs”:

```
#define CHARLIE long int
```

11 functions in C

11.1 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;

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

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

12 What’s a pointer?

This is where all hell breaks loose.

12.1 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[10000000000000000000];
```

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.

Many programming frameworks will supply their own version of malloc. But, if you are in plain old C, you would use this to grab the memory:

```
int *x; //suppose you want 10 items in a dynamically allocated array
int N = 10;
x = (int *)malloc ( N * sizeof(int)); //ask for memory
if ( !x ) exit(0); //check, see you got it.
```

12.2 A pointer is the "first position" of a piece of memory.

```
int *x;
int i;
//allocate this for N items, however you want
x = (int *)malloc (N * sizeof(int));
if ( !x ) exit(0);
x[0] = 3; puts the value of 3 into the first position.
for (i = 0; i< N; i++)
{
   x[i] = i*2;
}
Values in an array can be accessed ‘‘as if’’ they were values in a pointer.
```

Terminology Alert

"reference a pointer" put a value into memory

"dereference a pointer" get a value out of memory from a pointer

12.3 There is other stuff worth knowing about pointers

People can live with pointers, and use pointers, and then later start to understand pointers.

Many programming libraries (like Swarm for agent-based simulation) will supply idioms for working with pointers so it almost seems as if they are nothing special.

But in plain C, there are a few very important things to know.

Functions: access by value protects the input data

Consider

```
int myFunction (double x);
```

"x" is an input value into the function, myFunction cannot change it.

The only way to damage the input data is to be explicit about it:

```
y = myFunction (y);
```

Whatever you had before, you don't have it now.

Functions: access by reference does not protect the input data

Consider

```
(void) myFunction (double * x){
*x = 7.99;
}
```

What is the effect of the next command?

```
double *y;
myFunction(y);
```

The variable y now points at a section of memory that holds the value "7.99".

Functions: safer pass by reference

Use the const to protect the input from change, but put the output into a new pointer variable.

```
(void) myFunction (const double * x, double * y){
*y = 7.99;
}
```

The value pointed to at location x is unchanged, but the output value resides at the location pointed to by y.

All Scientific & Statistical Computing Libraries Use "pass by reference" extensively.

Consider a declaration in the GNU Scientific Library (GSL). This is from :gsl_sf_exp.h

```
/* Provide an exp() function with GSL semantics,
 * i.e. with proper error checking, etc.  *
 * exceptions: GSL_EOVRFLW, GSL_EUNDRFLW  */
int gsl_sf_exp_e(const double x, gsl_sf_result * result);
```

This one takes a floating point input and it sets the result into memory at a pointer location "result".

13 Containers

The idea of a container is from object-oriented programming, it is not native to C programming. But because we can use structs to make C more object-oriented, it should not come as a surprise that there are C methods to group together like kinds of objects into collections. I think the 2 kinds of collections that are most easily understandable are "linked lists" and "maps."

Linked lists and maps are not built into C, but they are available from many libraries. They can be found in programming libraries like "glib" (which is distributed with "gtk", the "Gimp tool kit"). They are also available in GSL.

C++ provides linked lists in a family of containers in its Standard Library. So does Java. This inclusion of a wider class of containers in the base language libraries is one of the major reasons that some people prefer Java or C++ to C.

13.1 Brief Sketch of Idea

Suppose you have a bunch of structs, say, one for each student. You might like to throw them in a container, and when something has to be done to all students, we would have an "iterator" thing that would take the first student, then do something, then take the next, and so forth.

13.1.1 linked list.

Create a struct, that has 3 elements. First, of course, it has the valuable data for a student. Usually, this will be a pointer to a struct. That part is simple, it is just data.

The next two elements are the links, the information that connects this particular student's data to the other students in the collection. Each element in the list only "knows" about 2 other elements, the one before and the one after it. So the second and third elements in this struct are pointers to list members that are before or after this one struct.

We usually build a list by creating items and then adding them to the list. A list is a "grab bag", it is disorganized. The order in which elements are inserted is not supposed to be important in this case, things are just added at the end.

We usually interact with a linked list by

- asking the list to give us the first (or last) element in the list
- doing our work on that one element's data
- asking that one element which element from the list would be next (or is before). And we continue.

An "iterator" is a tool we use to manage this navigation through the list. Iterators are generally able to respond to commands like "getFirst" or "getNext" and so forth.

13.1.2 Map

The disadvantage of a linked list is that we cannot easily find a particular element in the collection. If you tell me, "I want data for student 'willie"', then I have to ask, one-by-one, each element in the linked list if its name is willie. The linked list offers no shortcut. If the list includes only 5 or 10 elements, then that may be sufficient. But if the linked list has 1000s of elements, then this will be very slow.

Suppose instead of simply throwing elements into a list, we insert named elements into a collection. So instead of just "addLast" to put something into a collection, we have a notion of adding a student and giveing it an external identifier, the syntax would be something like "add(student, mapID)." The map container is doing the work of storing the objects and their unique identifiers, so that we can ask for particular objects when we want them.

Iterating through the elements of a Map is not usually as fast as iterating through a linked list. Putting elements into a Map is not as fast as adding them to the end of a linked list. However, retrieval of specific named elements is faster in a Map than a linked list.

13.1.3 Iterator

The power in the use of collections is in the Iterator, a way we have of processing list elements one by one. I've often thought of the iterator as a tiny spotlight that floats above a collection and highlights elements one at a time.