

Notes B

C for Programmers of Daughter Languages



CMPE1700
Algorithms and Data Structures
Winter 2018
2 Jan 2018

Reference:

Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language*, Prentice Hall, 1978



B.1 Introduction

Thankfully, we all have some experience with C# programming. Because C++, C#, Java and (to a lesser extent), Objective C all inherit from C and use similar syntax and keywords, you are already familiar with most of C. The main difference is that C is a much simpler language, and can be learned quite quickly. Important differences include:

- C has a much smaller standard library.
- C is not *object-oriented**, so objects, interfaces and classes are not present.
- C only automatically reserves storage space for variables that are declared inside a function. If you need more storage space, you must allocate it (and clean it up) yourself.
- The facilities for printing to the console are quite different.

The main things we need to be aware of to move to programming in C are therefore (some of these depend a bit on which version of the ISO C standard you are using. We'll tend to default to things that have been supported from the beginning):

- C comments usually begin with `/*` and end with `*/`, although the `//` syntax is supported.
- Instead of `bool` variables, we will use `int` values like 1 and 0.
- I/O will be performed using `stdio.h` functions.
- Casting is performed with *C-Style Casting*.
- `struct` declarations are slightly different.
- Memory allocation as performed with `malloc`, `realloc` and `free`.

*Whatever that means

B.1.1 Arrays and Strings

Arrays in C are just different enough from those in C# that they may occasionally be the source of confusion. Arrays in C are simply blocks of memory containing some set number of a smaller type. They cannot be resized or dynamically allocated.

Syntax B.1: *type name[size] = {set, of, initial, values}*

Allocate an array with label *name* of type *type*, with *size* members. Optionally, the values of the members of the array may be initialized with a comma-separated list of values in braces.

For example, an array of integers could be created with

```
1 int values[5] = {1,2,3,4,5};  
   printf("%i", values[2]); /* Prints the value 3 (arrays count from 0)
```

Strings are not a distinct type, like in C#, but simply a special case of an array of characters. By convention, we always have the value 0 as the last element in a string array to indicate the end. This means we don't have to always send the size of the string along with the variable when passing it to other methods. The quotation mark syntax makes use of this convention:

```
char Name[] = {'B', 'o', 'b', 0}; /* Size optional with value list */  
char SameName[] = "Bob"; /* Exact same values (0 added automatically) */
```

B.1.2 Casting

In C, type casting is syntactically similar to what we saw in C#, although the underlying mechanics are a bit different.

Syntax B.2: *(type) value*

Returns *value* of any type, converted to type *type*.

For example:

- `int iX = (int) cVal`
- `fAve = (float) iTot/iNum` - in this case, `iTot` is converted to a float prior to the division.
- `pFoo = (int *) pBar` - conversion of pointer types

B.1.3 Standard I/O

In C#, we are accustomed to using the *Console* class for I/O. In C, that classes (and all classes) are unavailable, so we must use *standard I/O*, as defined in the `stdio.h` library. All of these functions and techniques require a `#include stdio.h` declaration. When that declaration is made, the standard streams, `stdin` (standard input), `stdout` (standard output) and `stderr` (standard error,) are automatically available.

Output

To send output to the standard output (`stdout`, typically the active console, unless redirected), you can use one of several functions:

Syntax B.3: `int putchar(int c);`

Puts the single character denoted by `c` into `stdout`. Returns the ASCII value of the character written.



Syntax B.4: `int puts(const char * s);`

Puts the zero-terminated string pointed to by `s` into `stdout`. This function will always output a new line at the end of output. Returns a non-negative number on success, EOF on error.



For an example, see below.

Messages can also be sent to standard error (`stderr`) using `fputs` and `fprintf`, which are identical to `puts` and `printf`, save they also take an argument of a target stream:

Syntax B.5: `int fputs(const char * s, FILE * stream)`

Puts the zero-terminated string pointed to by `s` into the stream pointed to by `stream`. Use `\n` to output a new line. Returns a non-negative number on success, EOF on error.



For example:

```
fputs("Error: Invalid Operand\n",stderr);
```

In fact, all error messages should be sent to `stderr`.

Input

Input can be obtained from `stdin` (typically mapped to the keyboard, unless redirected) using some standard functions:

Syntax B.6: `int getchar(void);`

Returns the next character from `stdin`.



Syntax B.7: `char * gets(char * s);`

Reads a line from `stdin` and stores it in a zero-terminated string pointed to by `s`. **This function should never be used, because it does not check for buffer overrun**



Because of the buffer overrun problem with `gets`, you should always use `getchar` or `fgets` instead:

Syntax B.8: `char * fgets(char * s, int size, FILE* stream);`

Retrieves at most `size-1` characters from the stream pointed to by `stream` and stores them in a zero-terminated string pointed to by `s`. Will stop reading if a newline(carriage return) or end-of-file is encountered.



For example:

```
char szBuffer [256];
puts("Give me some words:");
fgets(szBuffer,256,stdin);
```

For additional examples, see below.

Examples

The following example illustrates the use of getchar and putchar:

```
1 /* Demo of getchar and putchar */
2
3 #include<stdio.h>
4
5 int main(void) {
6     int i = 0;
7     char c = 0;
8     char string[] = "This is a message";
9     /* Print my message out, one char at a time */
10    while(string[i] /* Is the character at i equal to 0? */)
11    {
12        putchar(string[i]);
13        ++i;
14    }
15    putchar('\n'); /* New Line */
16
17    /* Echo what the user types, up to CR */
18    puts("Type something and I'll play it back...\n");
19    while (c != '\n' && c != '\r') /* While c isn't a newline or carriage return */
20    {
21        c = (char)getchar(); /* getchar returns int */
22        putchar(c);
23    }
24
25    fflush(stdin); //Make sure we clear any chars in the buffer
26    fflush(stdout); //Make sure our code goes out to screen
27
28    return 0;
29 }
```

Listing 1: notes.B/chario.c

And this one illustrates the use of puts and fgets:

```
1 /* Demo of puts and fgets */
2
3 #include<stdio.h>
4 #define BUFFSIZE 256
5
6 int main(void) {
7
8     char buffer[BUFFSIZE] = ""; /* String Buffer */
9     puts("Say something:");
10    fgets(buffer,256,stdin);
11    fprintf(stdout,"You said: %s",buffer);
12    fputs("This is how you emit an error\n", stderr);
13
14    return 0;
15 }
```

Listing 2: notes.B/stringio.c

Formatted Output

In addition to the simple I/O described above, the `stdlib.h` library includes facility for fairly complex formatted output with the `printf` function:

Syntax B.9: `int printf(const char * format, ...);`
Prints the variables and values included as ... according to the format string specified as `format`. Valid format flags include `%i,%d` - integer numeric output; `%s` - string output, `%p` - addresses ; `%i=c` - character; `%f` - floating point value ; `%%` - a percent sign.

The `printf` function (you can read more with `man 3 printf`) inserts the values in its argument list into the flag placeholders in the format string and prints out the string (without a terminating newline). To add a newline, simply use `\n`. For example:

```
/* Demo of printf */
#include<stdio.h>

int main(void) {
5  /* declare and initialize sample variables */
   char c = 'A';
   int i = -7;
   unsigned int u = 42;
   double d = 3.14159;

10
   /* c is <A>, but <65> as a number. */
   printf ( "c is <%c>, but <%d> as a number.\n", c, c );
   /* i is <-7>, but <4294967289> if you use %u */
   printf ( "i is <%d>, but <%u> if you use %%u\n", i, i);
15  /* u is <42>, but still <42> if you use %d */
   printf ( "u is <%u>, but still <%d> if you use %%d\n", u, u);

   /* d is < 3.14> */
   printf ( "d is <%.6.2f>\n", d );
20  /* d is <3.142> */
   printf ( "d is <%.3f>\n", d );
   /* d is < 3.142> */
   printf ( "d is <%.10.3f>\n", d );

25  return 0;
}
```

Listing 3: notes.B/printf.c

Formatted Input

You can also use `scanf` to input several variables, using similar formatting commands. The main difference to bear in mind is that the arguments to the variables to store input in must be pointers, so `scanf` takes addresses as arguments. Note further that arrays (such as strings) automatically convert to addresses, so no `&` is required for an array name.

Syntax B.10: `int scanf(const char * format, ...);`
Retrieves values for storage in variables included as ... according to the format string specified as `format`. Valid format flags include `%i,%d` - integer numeric output; `%s` - string output, `%p` - addresses ; `%i=c` - character; `%f` - floating point value ; `%%` - a percent sign. Variables should be represented by a pointer to their memory location.

For example:

```

/* A nice demo of scanf from Wikipedia*/
#include<stdio.h>

4 int main(void) {
    int n;
    while (scanf("%d", &n) == 1)
        printf("%d\n", n);
    return 0;
9 }

/* Will take integers typed messily, with tabs, spaces and */
/* returns, and cleanly convert them to just the numbers */

```

Listing 4: notes.B/scanf.c

Note also that you can limit the maximum number of characters input by adding a number between the % and the type specifier (e.g: %79s). There are a number of other options available for format specifiers, see `man scanf`.

B.1.4 Structures

In C, the syntax for declaring structures is similar to that of C#:

```

struct person
3 {
    char LastName[40];
    char FirstName[40];
    int Age;
}; /* Remember the semi-colon */

```

However, unlike C#, when declaring structures in C, you must include the keyword `struct`:

```

struct person me;
strcpy(me.LastName, "Mbogo");

```

You can also declare pointers (more on this later in the course):

```

struct person * pDude = &me; /* Pointer to instance created above */
strcpy(pDude->LastName, "Foobar");

```

Note that the use of the member operators (`.` and `->`) to access members of a struct and pointer-to-struct, respectively.

It is also common to define and declare structures simultaneously:

```

struct card
3 {
    char * face;
    char * suit;
} a, deck[52], *cPtr;

8 /******
   Creates a single object (a),
   Array of objects (deck),
   Pointer to object (*cPtr)
   *****/

```

Typedefing Structures

It is common to use a typedef to simplify the declaration of structs, which allows you to use syntax similar to that for C#:

```
typedef struct person Person;
Person bob;
strcpy(bob.FirstName, "Bob");
```

You can also declare the struct and typedef it simultaneously (in the example below, struct person and Person) are the same type):

```
typedef struct person
2 {
    char LastName[40];
    char FirstName[40];
    int Age;
} Person;
7 Person bob;
strcpy(bob.FirstName, "Bob");
```

You can even skip giving the struct a name and just typedef an anonymous struct:

```
typedef struct
2 {
    char LastName[40];
    char FirstName[40];
    int Age;
} Person;
7 Person bob;
strcpy(bob.FirstName, "Bob");
```

This is the way that we will tend to declare structures—with a typedef giving the struct a friendly name, followed by using that name to declare actual instances.

B.1.5 Dynamic Memory Allocation

In C, Dynamic Memory Allocation (DMA) is accomplished using the malloc and free operators to allocate and deallocate memory from the free store.

Note: In order to use dynamic memory allocation in C, you must include the standard library: `stdlib.h`.



malloc - void * malloc(int)

malloc is used to allocate memory from the free store (heap) in much the same way that new is used in C++, or even new in C# (which automatically manages dynamic memory for you). The primary differences are:

- malloc allocates a particular number of *bytes* (the number of bytes is the only argument to malloc, and will not automatically allocate memory to the size of a type).

- malloc always returns a void pointer (void *), so the return value must be cast to a pointer to the type desired.

Syntax B.11: `(type *) malloc(sizeof(type) * length)`

Returns a pointer to *type* that points to a dynamically allocated array of type *type* and length *length*. In the degenerate case of allocating one item, *length* = 1 and is left out.

For example, if I wished to create a dynamically allocated int variable, I could use:

```
int * foo = (int *) malloc(sizeof(int));
```

Note the use of the sizeof function to return the size (in bytes) of a type or variable.

I could also allocate an array of 10 ints with:

```
int * bar = (int *) malloc(sizeof(int) * 10);
```

Note: malloc returns NULL if it is unsuccessful in allocating memory

It is also quite common to allocate structs:

```
#include <stdlib.h>

/*Simple Linked List */
4 struct node; //Forward declaration of node, needed for {\tt next} pointer.

struct node
{
    int data;
9    struct node * next;
}


/* Create and return a 3 node list
14 struct node * BuildList()
{
    /* Create three nodes */
    n1 = (struct node *) malloc(sizeof(struct node));
    n2 = (struct node *) malloc(sizeof(struct node));
    n3 = (struct node *) malloc(sizeof(struct node));
19
    /* Set dat values, and connect nodes to each other */
    n1->data = 1;
    n1->next = n2;
    n2->data = 2;
24    n2->next = n3;
    n3->data = 3;
    n3->next = NULL; /* Use NULL pointer for end of list */
    return n1; /* Return first item (head) of list */
}
```

If, for some reason, we wanted an array of nodes:


```
2 #define NUMNODES 10
   nodes = (struct node *) malloc(sizeof(struct node) * NUMNODES);
```



free - void free(void *)


C# is nice enough to handle all this memory stuff behind the scenes. In languages that are designed for real programmers, we must ensure that we release any memory that we dynamically allocate, or we will have a potential memory leak. This is done using the `free()` operation, passing a pointer to the memory to be freed as the only argument.

Note: Do not attempt to free a NULL pointer, it will generate a runtime error. 

```
#include <stdlib.h>
3 if (foo != NULL) free(foo);
  foo=NULL;
  if (bar != NULL) free(bar);
  bar=NULL;
8 if (nodes !=NULL) free(nodes);
  nodes=NULL;
```


Note: Even though `free` accepts a `void *` for its argument, we do not need to cast our pointers, because any pointer will implicitly cast to `void *`, even though the reverse is not true. 

Warning: You must be careful to only de-allocate the same location once. A good way to accomplish this is to set the pointer to NULL after freeing it and testing for NULL before attempting a `free` (as in the example above). 

Note: There are other functions in the `malloc` family, including `calloc` which can simplify the allocation of arrays, and `realloc`, which can be used to resize arrays. We will not cover them in detail here, but we may look at them when we take up DMA later in the course. Students may wish to investigate the `man` pages. 

B.1.6 File I/O

In addition to the standard I/O streams (`stdin`, `stdout`, and `stderr`), it is possible to use the I/O commands to read and write from files in the filesystem. The only additional requirement is that the file in question be linked to from a file stream (the `FILE` type).

Note: Recall that nearly everything in Unix and Linux is represented as a file, so File I/O is a very powerful tool. 

In order to instantiate a stream, create a `FILE` pointer (note that `FILE` is in all caps, and is defined in the standard library, `stdlib.h`):

```
#include <stdlib.h>
2 FILE * fptr;
```

The `FILE` pointer may then be used to open a file using the `fopen()` (`FILE * fopen (char * , const char *)`) call from the standard library:

Syntax B.12: `fopen(filename, mode` 

Returns a FILE * pointer to the file at *filename*, opened with mode *mode*, when *mode* is one of: "r", "r+", "w", "w+", "a" or "a+".

The effect of the modes are as follows:

mode	Used For	Create File?	Existing File?
"a"	Appending	Yes	Appended To
"a+"	Reading and Appending	Yes	Appended To
"r"	Read Only	No	If not found, fopen returns NULL
"r+"	Read and Write	No	If not found, fopen returns NULL
"w"	Write Only	Yes	Destroyed
"w+"	Read and Writ	Yes	Destroyed

Thus, a file may be opened for reading as follows:

```
#include <stdlib.h>
3 #define MYFILE "/var/tmp/myfile"
FILE * fptr;
fptr = fopen(MYFILE, "r");
```

We may then use our fprintf() function (from stdio.h, see below) to write to the file:

```
int i = 4;
if(fptr!=NULL)
4 {
    fprintf(fptr, "I am writing the number %d to my file.", i);
} else fputs("Error Opening File", stderr);
```

Of course, if we open the file, we should close it:

```
fclose(fptr); /* Flush Buffers and Close*/
```

Reading and Writing with Files

In Unix and Linux, console I/O is effectively identical to file I/O, because the standard streams are themselves files (stdin, stdout, and stderr are all of type FILE *).

All we need are versions of our I/O functions that accept a generic FILE pointer instead of using a default. We have already seen fputs and fgets.

In addition, there are generic file versions of putchar and getchar:

Syntax B.13: int fputc (int c , FILE *stream);
Writes the character c, cast to unsigned char, to the stream defined as stream.

Syntax B.14: int fgetc(FILE * stream);
Reads the next character from stream and returns it as an unsigned char cast to an int or EOF on an end-of-file or error.

And, of course, formatted I/O is supported:

Syntax B.15: `int fprintf(FILE * stream, const char * format , ...`

Prints the output specified by the `format` string, with values supplied as additional arguments inserted, to the output stream `FILE`. Returns the number of characters printed.



Syntax B.16: `int fscanf(FILE * stream, const char * format, ...);`

Scans the input `FILE` stream, retrieving types described in the `format` string, storing values in the variables pointed to by the additional argument pointers.



Copyright © 2018, AJ Armstrong. This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/). Queries and comments concerning this document are welcome. Contact the author, AJ Armstrong, at aja@nait.ca