University of Northumbria - School of CEIS

A quick guide to C

MJB September 2010

These notes are intended to introduce the main features of C to students who have previously studied Java and are taking an Operating Systems or Embedded Systems module.

Introduction

C or C++ are the languages most commonly used for systems programming. These languages allow the programmer to have direct access to the operating system and, potentially, the real hardware. Java, by contrast, achieves its portability by running on a virtual machine that deliberately isolates the Java programmer from the underlying physical machine. The C language is about 30 years old, and was the language used to write Unix. C++ was originally written as an extension of C, to introduce object oriented ideas such as a **clas**s, and **inheritanc**e. Java uses many of the same ideas, and has a similar syntax to C++, but since it was devised as a complete new language some of the less attractive aspects of C, that C++ inherited, are not present in Java.

For this module you will need to understand some simple programs, and write some pieces of code to extend or modify these programs. The programs don't use object orientation, but do make use of relatively low level operating systems interfaces, so Java is not appropriate, so C is used. Only a subset of C is needed, and those students who have previously used Java should find many similarities between the languages.

These notes are intended to help a Java programmer understand how to read or write programs for this module. They should not be regarded as a definitive description of C. For more information on C programming, novice programmers could look at "A First Course in Computer Programming Using C" by King, Pardoe, and Vickers (McGraw Hill, 1995) Experienced programmers are recommended to look at Kernighan & Richie's definitive book "The C Programming Language" (Prentice Hall).

Program structure

C programs are a collection of functions, which may be written in one or more source code files. Unlike Java, the filename does not need to correspond to the function or program name, but the compiler does require that the filenames end with `.c`

All the functions in C programs are "public" throughout the program, there is no need to distinguish between public and private functions in C. (This simplifies easy programs, but is really a weakness in the language) Each program must have one function called `main`. When the program is executed, the execution starts with the first instruction of `main`.

A simple C program is

```
#include <stdio.h>
/* a simple C program */
int main() {
   printf( "hello world\n" );
   return 0;
}
```

Example 1

– This program consists of a single function, called `main` , which does not return any value, so like its Java equivalent, is of type `void` .

– The first line of the program specifies a "header file", `stdio.h`. The compiler will read this file before dealing with the rest of the program. (There are two types of header file – local files, which need their filename in double quote marks and system files, which need their filename in angle bracket <>) In this example `<stdio.h>` is a system header file that contains the definition of the function `printf` used inside `main` . The `#include` line is a directive to the C compiler to include in the source file the contents of another file: in this case `stdio.h`, which declares a number of standard I/O functions.
– As with Java, C++ statements end with a semi-colon.
– Comments begin with /* and end with */. They can extend over several lines. Take care to make sure you use */, or the compiler will ignore most of your program.

To run a C program you need to first compile it. If the program is called myprog.c, then in Unix it can be compiled with

```
     cc myprog.c
or
     gcc myprog.c
```

Variables, and Types.

C has a number of intrinsic or "built in" data types. The only ones we will need are `int` , `long` , `char` `float` and `double`. These more-or-less correspond to the Java types with the same names (though Java defines them more carefully than C). C doesn't have a `boolean` type with values `true` and `false`. C programmers have to make do with using an `int` variable, where 0 represents true and any other value, typically -1, represents false. One way round this omission from C is to define `boolean` (or perhaps `bool` for short), `true` and `false` in a header file. Once the compiler has read the header file, boolean variables can seem to exist (though usually they are really still `int` variables, so `true+2` can be a valid - but silly - expression in a C program).

Variables are declared in the same way in C and Java:

```
     int length, height;
     long largeNumber;
     float area;
```

Unlike Java, C provides the key-word `unsigned` for use with integer types. Thus `unsigned int length` declared the variable as a 16-bit unsigned integer – it can take a value between 0 and 65535 (= $2^{16} – 1$) while declares it as a 16-bit signed integer – possible values are between -32768 (= $-2^{15}1$) and +32767 (= $2^{15} – 1$). Signed integer types are implemented using *two's complement* while unsigned types are not.

Generally, the best place to declare variables is in the statements at the beginning of a function, and these variables are then in scope for the whole of the function. Sometimes variables are declared at the beginning of a file, outside of any functions. These variables are global to all the functions in the file. Variable names in C (like Java) are case sensitive, so `largeNumber` and `largenumber` are two different variables.

Operations and Assignments

The numerical operators +, -, *, /, % have the same meanings as in Java, applied to integer or floating-point expressions. Generally, identical operations in each language will result in identical values, but errors are dealt with differently in the two languages and division by zero will crash one of our C programs, whereas in Java it produces `NaN`. When these operators are used solely with operands of type `int`, the result is an `int`. Care should be taken with the division operator, since the result of 5/2 is 2 but the result of 5.0/2 is 2.5.

The assignment operators =, +=, -=, ++, -- in C are also similar to Java. The comparison operators ==, <=, >=, <, > are as in Java except they are integer-valued because C does not have a separate `boolean` type. Likewise with the logical operators !, &&, ||.

Beware: in C, something like

```
if (x = 1) {
  x++;
}
```

will compile cleanly even if you meant to write `(x == 1)` because C does not distinguish integer and boolean types and interprets a nonzero int as "true" -- so your program could have an obscure bug if you make this mistake!

Bit Twiddling

You may not have used the bitwise operators &, |, ~, ^ in Java: where x and y are of integer type,

`x & y` means AND each bit of `x` with the corresponding bit of `y`;
`x | y` means OR each bit of `x` with the corresponding bit of `y`;
`x ^ y` means XOR each bit of `x` with the corresponding bit of `y`;
`~x` means apply NOT to each bit of `x`.

The results are of integer type (not boolean as in the case of &&, ||, !). These all work the same way in C where you will find them used in embedded system software. For instance,

```
int MASK = 0x0248;
if (val & MASK == MASK) {... }
```

tests whether bits 3, 6 and 9 of `val` are on.

Similarly if we wanted to switch on bit 2 of `val`, `val = val | 0x0004` would do, and `val = val & 0xFFFB` would would switch bit 2 off. (More briefly, `val |= 0x0004, val &= 0xFFFB`).

Such operations are often used in embedded systems to test inputs and do output as a bit may control (or test) the state of an electric signal, 1 being "on" and 0 being "off".

Hexadecimal constants are notated 0x.... just as in Java. Hex is really good for helping map between an integer value and a bit pattern:

0000 = 0x0 = 0;	0001 = 0x1 = 1;	0010 = 0x2 = 2;	0011 = 0x3 = 3;
0100 = 0x4 = 4;	0101 = 0x5 = 5;	0110 = 0x6 = 6;	0111 = 0x7 = 7;
1000 = 0x8 = 8;	1001 = 0x9 = 9;	1010 = 0xA = 10;	1011 = 0xB = 11;
1100 = 0xC = 12;	1101 = 0xD = 13;	1110 = 0xE = 14;	1111 = 0xF = 15.

Arrays

In C arrays are declared and instantiated in a single statement eg

```
char buffer[1024];
```

declares an array to hold 1024 characters, in locations numbered from 0 to 1023. In C, unlike Java, arrays aren't objects with methods, so you can't ask an array what size it is. The size can't change after it has been declared and the programmer is expected to know the size.

Strings

Strings of characters, such as "hello world" can be dealt with in a variety of ways. In our programs we use the original approach to strings, which is to store the string as a sequence of characters ending with a special character called `null` - this is the character which has ASCII code zero. So the string "hello world" consists of 12 characters! A variable to store a student identifier (which has 7 characters) could be declared with

```
char id[8];
```

These null terminated strings are the way Unix (and many other systems) internally deals with strings. The problem with this way of handling strings is that they can't be manipulated using =, ==, or + in the way that Java (or even C++ with a suitable string class) allows. To test if two strings are equal the function `strcmp` is used, and `strcpy` or `strcat` can be used to copy or concatenate strings. Example programs using these functions will be available. (When using these functions, the system header `<string.h>` should be included in your program – see example 2).

String values can be set in a program using double quote marks. Single quotes are reserved just for single character values. so 'm' is a single character value, but "m" is a string containing two characters 'm' and the null character.

```
char ch;
char s;
char str[50];
ch='m'; /*legal */
s="m"; /* error */
strcpy(str,"m"); /* legal */
strcpy(ch, "m"); /* error */
strcpy(str,'m'); /* error */
```

Example 2

To convert a number into a string representation, or vice versa, is beyond the scope of these notes. For now, simply remember that it is not trivial!

Output

Values of types `int`, `char` and `string` (array of char including a null terminator) can be output in a C program with the function printf – "print formatted".

```
char ch = 'a';
int age = 13;
char name[20];
strcpy(name, "Robert");
printf("The first letter of the alphabet is %c\n", ch);
printf("%s is %d years old today!\n", name, age);
printf("The initial letter of %s is %c\n", name, name[0]);
```

Example 3

The first argument to prinf is the string to be output. It may contain *format specifiers*:

`%c` for a character
`%s` for a string
`%d` for an int
`%ld` for a long int
`%u` for an unsigned int
`%f` for a float

There are other format specifiers too. The `printf` function must have additional arguments of the same number and in the same order as the as the format specifiers embedded in the output string.

Getting this wrong can have amusing consequences. For instance `printf("The value is %c", 65);` will output `"The value is A"` because a `%c` was specified and 65 is the ASCII code for 'A'. And `printf("The ASCII code of %c value is %d", 65);` will output `"The ASCII code of A is 65"`.

The `\n` means the same as in Java – a new-line character. All the \ escape sequences in Java originated in C.

Input

is performed using the function `scanf(...)`:

`scanf("%s", stg)` – to input from keyboard to a string variable `stg` (eg `char stg[20]`);
`scanf("%c", ch)` – to input from keyboard to a single `char` variable `ch`;
`scanf("%d", &num)` – to input from keyboard to an `int` variable `num`;
`scanf("%f", &x)` – to input from keyboard to a `float` variable `x`;

Note the & -- this means "the address of". The scanf function has to be provided with the *address* to store its data to. You will see below why the & prefix is necessary to achieve this with variables of simple data type (`char`, `int`, `long`, `float`, `double` etc) but not with a string variable (which is actually an array data type).

Control structures

Conditional statements and **loops** can be programmed the same way in both Java and C. Some C examples are:

```
int age;
printf("Type your age: ");
scanf("%d", age);
if ( age < 18) {
    printf("too young to buy alcohol\n");
} else {
    printf("too old to get a child fare");
}
```

Example 4

```
for (i=0;i<11;i++) {
    printf("%d  times 7 = %d", i,  i*7);
}
```

Example 5

```
while (ans!=49) {
    printf("What is 7 * 7?");
    scanf("%d", &an);
}
```

Example 6

Functions

A function in C looks like a Java function (method), except that it doesn't belong to a class (C doesn't have classes and objects) and you don't have to worry about qualifiers such as `public`,

`private` or `static`. If the function generates a value, that value can be passed back using the `return` statement. Parameters can be used to pass values into a function. Each parameter has to have its type declared, and the type should match the type used when the function is called.

```
int twomore(int n) {
   return(n+2);
}
void main() {
   int p; /*local variables */
   int q;
   p=4;
   printf("%d\n", twomore(p)); /* OK */
   q = twomore(467); /* OK */
   r = twomore(45.67); /* the parameter will be converted
            to an integer before the function executes */
```

Example 7

Pointers

The declaration

```
int x;
```

declares an integer variable, called `x`. The declaration

```
int *px; or
int* px; or
int * px;
```

declare px to be *pointer to an integer*. This means that the variable px can contain an *address* at which an integer can be stored. Initially px doesn't contain a suitable address, but it can be given one with an instruction such as

```
px = &x;
```

The '&' means "the address of". In this example the address of x is stored in `px`, and the expression `*px` means "the value pointed at by `px`". Writing an expression like `*px` is often called *dereferencing* the pointer.

Pointers are often used with strings (arrays of characters), for example

```
char *message;
message ="Hello World";
```

Here a pointer variable is declared and then in the next instruction the compiler will store the null terminated string "Hello world" somewhere in memory, and then put the address where the string starts into message. The string could be printed using `printf("%s", message);`

Warning!

Technically, an array of some data type (`char`, `int` or whatever) *is* a pointer to the memory address used by the array. But an array declaration is slightly more than a pointer declaration. One of the most common C programming errors is to declare a string variable as `char *` and then use it to receive string data:

```
char * name;
scanf("%s", name);
```

The pointer name could be pointing anywhere in memory; poking user-input data there may well crash the computer[1]. The correct way to do this is

[1] Eg (if the value of the pointer = 0) the interrupt vector table in low memory!

```
char name[100];
scanf("%s", name);
```

This will reserve 100 bytes of memory and set the char pointer name to point at it, before poking user-input data into it. *Always* declare string variables to be char arrays big enough to hold the string including the null terminator.

More generally, *always* set a pointer to point somewhere meaningful (or at least, harmless) before copying data to the location it is ponting at.

Pointer arithmetic

The upshot of the remarks above is that a pointer variable is the same thing as an array variable, except that an array declaration additionally causes memory to be allocated and the variable initialised to point at it. In fact, array [] notation can be used with a pointer variable just as with an array variable. If we have the declaration

```
int * px;
```

then the expression `*px` is an int. We can also denote it `px[0]`

Furthermore, we can do arithmetic with pointers. px + 1 means the memory location 1 x sizeof(`int`) after px and `px-3` means the location 3 x sizeof(`int`) places before px. We can write *(px + 1) instead of px[1] and write `*(px-3)` instead of `px[-3]`. These are not type-safe expressions, of course – the data at these locations might not actually by `int`s

Similarly, if px is declared as an array: `int px[20];`
...then we can write dereferencing expressions: `*px` is an int, as is `*(px+1)` (which means the same as `px[1]`), etc.

These remarks apply equally to pointers to (arrays of) other data types – `char`, `float`, etc.

Parameters for functions

One of the main uses for pointers (and/or '&' operators) is when using a function to change the values in one or more variables passed to it as parameters. In Java, "object variables" are passed to functions *by reference*. This means that if the function does something to the object, changes its state, the actual object on which the function was called gets changed. In C, parameters are passed to a function *by value* the function uses a *copy* of the parameter, and any changes to the parameter within the function are lost when the function has finished.
eg

```
void addtwo(int x) {
  x=x+2;
}
main() {
  int n = 56;
  addtwo(n);         /*does this add 2 to n? */
  printf("%d", n);  /*nah, it is still 56   */
}
```

Example 8

If you want a function to change the value of a variable passed to it as a parameter, then the function must be given the address of the variable – in other words, a *pointer* to the variable.

```
void addtwo(int* x) {
  *x=*x+2;
```

```
}

int main() {
  int n = 56;
  addtwo(&n);         /*does this add 2 to n? */
  printf("%d", n);    /*yep - we now have 58  */
}
```

Example 9

Do you see now why we used the '&' in the `scanf` function when inputting to an `int` variable but *not* to a string?

```
char name[100];
scanf("%d",  name );
```

A string is an array or char – a pointer to char. When using an array as a parameter, remember that the array name is really just a pointer, so there is no need to use the & operator.

Safe use of strings; String functions

Strings are stored in arrays of char. The array must be declared, with a fixed size and this gives the maximum size of string that can be stored in the array. Functions such as `scanf`, `strcpy` or `strcat` which input or copy strings can store a string larger than the array and write beyond the bounds of the array. When the program executes this will lead to other data in the program's memory being overwritten, and the program may crash.

Safer versions of these functions, that allow the programmer to specify the maximum size of the string, are `strncpy` and `strncat`.

- `strcpy(char * target, char * src)` copies the source string to the target
- `strcat(char * target, char * src)` *concatenates* the source string to the target
- `strncpy(char * target, char * src, int n)` copies the source string to the target, *up to n characters*
- `strncat(char * target, char * src)` concatenates the source string to the target, *up to n characters*

```
#include <stdio.h>
#include <string.h>
int main() {
  char firstname[20], secondname[20], fullname[30];
  scanf("%d", firstname);
  scanf("%d", secondname);
  strcpy(fullname,firstname);
  strncat(fullname,secondname,30);
  printf("%s\n", fullname);
}
```

Example 10

These functions are provided by the string library -- `#include <string.h>` -- but their implmentations are simple in terms of `char` pointers:

```
char* strcpy(char * tgt, char * src) {
  int k = 0;
  while (src[k] != 0) { /* not the terminal null char */
    tgt[k] = src[k];
    k++
  }
```

```
  tgt[k] = 0; /*stick a null char on the end */
  return tgt;
}

char* strncpy(char * tgt, char * src, int n) {
  int k = 0;
  while (src[k] != 0 && k < n) {
    tgt[k] = src[k];
    k++
  }
  tgt[k] = 0;
  return tgt;
}
```

You can probably think of ways of implementing strcat and strncat similarly.

One other useful <string.h> function is to *compare* strings:
`int strcmp(char * stg1, char * stg2)` returns a negative number if `stg1` < `stg2` in the sense of Java, a positive number if `stg1` > `stg2`, and 0 if `stg1` and `stg2` have exactly the same characters. The implementation is probably something like

```
int strcmp(char * stg1, char * stg2) {
  int k = 0;
  while (stg1[k] == stg2[k] && stg2[k] != 0)
    k++;
  return (stg1[k] – stg2[k]); /*difference of ASCII values */
}
```

Structures

The syntax of a *structure* is indicated by this example:

```
struct PayRollRecord {
      int EmployeeNumber;
      char Name[25];
      float PayRate;
      };
```

This creates a user-defined data type struct `PayRollRecord` with **fields** `EmployeeNumber`, `Name`, etc. You can make user-defined type out of this structure:

```
typedef struct PayRollRecord PAYROLLREC;
```

The following declares a variable `PayRec` of this type:

```
PAYROLLREC PayRec;
```

and now the various fields of this variable can be accessed as `PayRec.EmployeeNumber`, `PayRec.Name`, `PayRec.PayRate` etc. Use the struct somewhat like a *class* in Java: but there are no methods -- only data fields -- and everything is *public*.
The declarations

```
PAYROLLREC * pPayRec;
PAYROLLREC payroll[10];
```

declares `pPayRec` to be a pointer to a `PAYROLLREC` record, and `payroll` *also* to be a pointer to a `PAYROLLREC` record. Remember that an array variable is actually a pointer with some additional functionality.

There is a special syntax you can use for dereferencing a pointer to a stucture and selecting one of the fields in the structure:

`pPayRec->PayRate`
means the same as `(*pPayRec).PayRate` or `pPayRec[0].PayRate`.

After the code
pPayRec = payroll + 5;

`pPayRec->PayRate` is actually referring to the same data as `payroll[5].PayRate`.

Try the following example:

```
#include <stdio.h>
#include <string.h>
typedef struct Person {
  char name[20];
  int age;
} PERSON;

int main() {
  PERSON myGroup[6];
  PERSON * pPerson;
  int k;

  for (k=0; k<6; k++) {
    printf("Person no %d: Name: ", k);
    scanf("%s", myGroup[k].name);
    printf("Age: ");
    scanf("%d", &myGroup[k].age); /* NB the & */
  }

  for (k=0; k<6; k++) {
    pPerson = myGroup + k;
    printf("%s is aged %d\n", pPerson->name, pPerson->age);
  }

  return 0;
}
```

Example 11

Exercises

Exercise 0:

Compile and run the HelloWorld program--

[compile] `>gcc helloWorld.c -o helloWorld`

[run] `>helloWorld`

Exercise 1:

(a) Write a C program to display the integers 0--15 as hexadecimal digits. (Use a for loop and a `printf()` function with a `%x` format specifier)

(b) Adapt it to display the integers 0 -- 255 in hex. Experiment with the formatting and try to get a neat output.

Exercise 2:

Write a program to input a string and report its length.

Exercise 3:

Write a program to input a string and display all the ASCII codes if its chasracters, in hex.

Exercise 4:

(a) Write a program to input a string and iterate through all its characters, switching ON the 32-bit of any character that is between 'A' and 'Z'.

(b) Write a program to input a string and iterate through all its characters, switching OFF the 32-bit of any character that is between 'a' and 'z'.

(c) Write a program to input a string and iterate through all its characters, toggling the state of the 32-bit of any alphabetic character.

(d) Develop functions (subroutine) to

- put a string into lower case,
- put a string into upper case,
- toggle the case of the string.

Write a program to drive these functions to test them.

Exercise 5:

Write a program to do input two integers and a character (&, |, ^, or ~) and do the corresponding bitwise operation between the integers, outputting the integers and the result in hex.

Exercise 6:

The code below uses *command-line arguments* for input. The for-loop simply iterates through all item entered on the command line of the program invocation, and displays them.

```
1      #include <stdio.h>
2
3      int main(int argc, char * * args) {
4        int i;
5        for (i = 0; i < argc; i++)
6          printf("Argument %d: %s\n", i, args[i]);
7        return 0;
8      }
```

The Java equivalent of line 3 is `public static void main(String[] args) {...}`. Any list of strings (separated by white space) entered on the command-line invoking the program (>java MainClass arg arg ...) end up in the array of Strings here denoted `args`.

In C `char * * args` also denotes an array of strings. A `char *` points to (the beginning of) an array of char – ie, denotes a string – and so `char * * args` points to an array of `char *` – an array of strings. The argument strings fed into the array are accessed inside the program using pretty much the same syntax as in Java: `args[0]` is the 0th argument, `args[1]` is the 1th argument, etc. Two things more are different in C: first, `args[0]` refers to the *name of the program* and `args[1]` is the first *actual* argument (it would be `args[0]` in an equivalent Java program); second, an array in C does not know its own length so this has to be passed in as an additional argument, `argc`. You don't need this in Java, there you would get the came information as `args.length`.

For this exercise, build the program and run it with a variety of command-lines: eg

```
[compile] >gcc argsExpt.c -o  argsExpt
[run]     > argsExpt One flew over the cuckoos nest
[run]     > argsExpt "One flew over the cuckoos nest"
[run]     > argsExpt One "flew over" "the cuckoos nest"
```

... and so forth. Can you understand the operation of line 3 now?

Exercise 7:

The code below is version of program 5 using *command-line arguments* for input. It is saved as Ex5Arg.c. Compile it (`gcc Ex5Arg.c -o Ex5Arg`) and exercise it using the command lines -

```
[run 1]    > Ex5Arg "~" 12345
[run 2]    > Ex5Arg 12345 "&" -46789
[run 3]    > Ex5Arg 12345 "|" -46789
[run 4]    > Ex5Arg 12345 "^" -46789
```

Explain the output of run 1 in terms of line 13—17. Explain the outputs of runs 2,3,4 in terms of lines 18—33. For each run, list the lines that are executed and state in "english" what the line is doing.

```
1   #include <stdio.h>
2
3   int main(int argc, char * * args) {
4     int a, b, ans;
5     char op;
6
7     if (argc < 2) {
8       printf("Usage: %s ~ <integer> or %s <integer> [&|^] <integer>\n",
9         args[0], args[0]);
10      return 1;
11    }
12
13    if (argc < 4 && args[1][0] == '~') {
14      sscanf(args[2], "%d", &a);
15      ans = ~a;
16      printf("~ %08x = %08x\n", a, ans);
17    }
18    else {
```

```
19          sscanf(args[1], "%d", &a);
20          sscanf(args[3], "%d", &b);
21          op = args[2][0];
22
23          if (op == '&')
24            ans = a & b;
25          else if (op == '|')
26            ans = a | b;
27          else if (op == '^')
28            ans = a ^ b;
29          else
30            ans = 0;
31
32          printf("%08x %c %08x = %08x\n", a, op, b, ans);
33        }
34        return 0;
35      }
```

Exercise 8:

Develop command-line-input versions of the programs in exercises 2, 3.

Exercise 9:

Exercise the program of example 11 (page 10). Write a similar program to use the PayRollRecord structure (page 9) to store a payroll of 5 people. Have it give everyone a 10% pay rise and print employee details out afterwoods.

Solutions

appear in the present exercise folder but try not to look until you have thought the problem through.

C Resources

Text

Stephen G Kochan, *Programming in C*, 3ed, Developer's Library (SAMS), 2005

Tutorials

(PDF) http://cslibrary.stanford.edu/101/EssentialC.pdf

(PDF; a classic, updated)
http://phy.ntnu.edu.tw/~cchen/ctutor.pdf

On-line tutorials -

http://www.imada.sdu.dk/~svalle/courses/dm14-2005/mirror/c/

http://einstein.drexel.edu/courses/Comp_Phys/General/C_basics/c_tutorial.html

Reference

An on-line C reference -

http://www.acm.uiuc.edu/webmonkeys/book/c_guide/