A Quick Introduction to the C Programming Language

Walter Pharr, Department of Computer Science, College of Charleston
Slightly updated by R. McCauley, likely now includes errors

0 Background

C is a general-purpose computer programming language developed between 1969 and 1973 by Dennis Ritchie at Bell Labs. C is one of the most popular languages (see http://www.langpop.com/). Designed for use in developing systems software, low-level enough to be used for applications previously implemented in assembly code.

Characteristics:

- Imperative language, has features that support structured programming
- Lexically scoped
- Static typing
- Partial weak typing
- Supports recursion
- Parameters passed by value, but pass by reference can be simulated
- Provides low-level access to computer memory by converting machine addresses to typed pointers
- Designed to be easy to compile and run, requires minimal run-time support
- Designed to be compiled in a single pass – thus the need to declare variables and functions before they are accessed.

1 Introduction

Every C program must have a function called `main( )`, with a parameter list, which may be empty. The text of the program is surrounded by braces, `{` and `}`, which define a **block**. A simple complete program:

```
#include <stdio.h>
main()
{
  printf("Hello, world\n");
  return (0);  // indicates everything executed OK
}
```

The main method in C is a function and it always returns an int, even though it does not have to be defined as such. That is the default. Not including the "return" statement may generate a complaint/error by a compiler.

C uses the semicolon as a statement **terminator**, so a semicolon is required at the end of every statement in C. The `printf( )` causes output to go to the terminal, and the `\n` at the end goes to the next line. The quotes around the string to be printed are double quotes. Identifiers in C may contain letters and underscores (_). C is case-sensitive. A comment in C is surrounded by the symbols `/*` and `*/`.

Before a C program is compiled, it is passed through the **C preprocessor**, which interprets lines beginning with `#`. Lines beginning `#include` cause code from various C libraries to be included in the program. Four especially useful `#include` statements are:

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

```
#include   <strings.h>
#include   <math.h>
```

These cause functions to be included for input-output, for character conversion, for string manipulation, and for floating-point mathematics, respectively. The `#include` statements should be entered exactly as here, *with* angle brackets and *without* a semicolon at the end (these are not C code). Lines beginning `#define` allow constants to be defined for the C program. There is no semicolon. Traditionally in C constants are in upper case. Some examples:

```
#define   TRUE   1
#define   FALSE  0
#define   PI     3.14159
```

The C preprocessor replaces references to defined constants throughout the program text, with the text of the constant's "value". Constants defined this way can be used wherever a literal constant is allowed, such as in declaring the size of an array.

The **built-in types** in C, with definitions, include:

`int`	like Java's `int`
`float`	like Java's `float`
`double`	like Java's `double`
`char`	*not* like Java's `char`, `char` is a byte-size int type

(There are also `long int` and `short int` as in Java; and `unsigned` and signed (the default) ints of all sizes. In doing real arithmetic, use `double` instead of `float`, because the functions in the C library expect `double`. Declarations are made within the bracketed block, and the type precedes the identifier, like this:

```
{
  int i, j;
  float f;
  double d;
}
```

2 Expressions and Assignments

Arithmetic expressions in C include the **assignment** (that is, it's not a statement). In this:

```
a = 3;
```

the value 3 is assigned to the variable a, and this value is also *returned* (since the assignment is an expression). Because it is an expression it can be used as part of a larger expression as well as by itself. In this:

```
a = 2 + b = 4;
```

the value 4 is assigned to the variable b, the value of this expression is added to 2, and 6 is assigned to a (this value also being returned by the entire line, but "thrown away").

The **arithmetic operators** `+`, `-`, `*`, `/`, and `%` are the same as in Java, giving integer results for integer arguments and real results for real and mixed arguments. `%` is for integer arguments only. The **relational operators** in C are `==`, `!=`, `<`, `>`, `<=`, and `>=`, the same as in Java. *Watch out* for using `=` as a test for equality! It is an assignment, and it is an expression, not a statement, so

```
if (a = b)
...
```

will *not* produce a compile-time error, but will perform an assignment right in the middle of what you thought was a conditional. Moreover, whether the test comes out true or false depends on the value of b only. C has no type "boolean". Anything that evaluates to 0 is treated as **false**, and anything that evaluates to *anything else* is treated as **true**. (So, in the incorrect example above, if b has the value 0, the condition will evaluate to false, and if it has any other value, the condition will evaluate to true.) When the compiler evaluates a relational expression, it uses 1 for true. If you need a boolean-like variable, declare an integer, and set its value to TRUE or FALSE for true or false, respectively. (This presupposes that you have used the `#define`'s mentioned in Section 1.)

C has the increment operator, `++`, the decrement operator, `--`, and operators that combine arithmetic and assignment: `+=`, `-=`, `*=`, `/=`, and `%=`. (These all work the same as in Java, which inherited them from C.)

3 Control Structures

C's control structures are just like Java's, because Java's are derived from C.

3.1 Iteration

The `while` loop has this syntax:

```
while (expression)
  statement
```

or for a compound statement:

```
while (expression)
{
  statement-1;
  ...
  statement-n;
}
```

The `do` loop moves the test at the bottom. Both loops exit on a false condition.

```
do
  statement;
while (expression);
```

The `for` loop semantically is another `while` loop. The syntax for this construct:

```
for ( initial-expr; loop-condition; loop-expr)
  statement
```

where `initial-expr` is made true initially, the loop is executed as long as `loop-condition` is true, and `loop-expr` is executed once every time the bottom of the loop is reached. Thus the above loop means the same as:

```
initial-expr;
while (loop-condition)
{
  statement;
  loop-expr;
}
```

Hence `initial-expr` and `loop-expr` are like statements (often they are assignments).

3.2 Selection

`if` statements in C are for two-way selection:

```
if (expression)                if (expression)
  statement                      statement-1
                               else
```

```
                    statement-2
```

Compound conditions in C are expressed by `!` for NOT, `&&` for AND, and `||` for OR:

```
if (! expr-1)
  statement-1;
if (expr-2 && expr-3)
  statement-2;
if (expr-4 || expr-5)
  statement-3;
```

Do not use a single `&` or `|`, because those mean something else in C (unlike Java).

For multi-way selection, there is the switch statement:

```
switch (expression)
{
  case value-1:
    statement-1;
    statement-2;
    break;
  case value-2:
  case value-3:
    statement-3;
    break;
  default:
    statement-4;
}
```

The `break;` statement must be present at the end of each case, or else control will "fall through" into the next case. This allows either value-2 or value-3 to invoke statement-3. A `break;` statement may also be used within any iterative control structure to cause a permanent exit from the loop.

4 Output and Input

4.1 Writing to the terminal

To write to the terminal, use a `printf()` statement. We have already seen an example of a `printf()` using a string literal, but usually a `printf()` has a **control string** followed by a list of variables. The control string is surrounded by double quotes, and may consist of text and **conversion characters**, which are preceded by a `%`. The most common ones and their meanings:

`%d`	decimal integer
`%f`	floating-point number
`%e`	floating-point in exponential notation
`%c`	single character
`%s`	string

Thus the equivalent of the Java statement:

```
System.out.print("The sum of" + v1 + " and " + v2 + " is " + v3);
```

is the C statement:

```
printf("The sum of %d and %d is %d", v1, v2, v3);
```

(assuming integer data throughout). The conversion characters may contain other formatting information. Illustrating with `%d`, replace `n` and `p` with integers in the following:

`%nd`	right-justify in a field n characters wide
`%-nd`	left-justify in a field n characters wide

`%n.pd`	left-justify in a field n characters wide, with a precision p

Other useful characters which may be used in the control string are the **escape characters**, which make certain characters "visible". Some of these are:

`\n`	newline
`\t`	tab
`\r`	carriage return
`\b`	backspace one character

Thus a `printf` with a `\n` at the end of the control string is equivalent to a `println()` in Java. Another commonly used function is `putchar()`, which has the format:

```
putchar(c);
```

and which writes the single character in the variable `c` to the standard output.

4.2 Reading from the terminal[1]

The `scanf()` statement may be used to read from the terminal. The format is similar to that of `printf()`, so that if `n` is an integer variable

```
scanf("%d", &n);
```

will read an integer from the terminal and put it in `n`. An `l` ("ell", not "one") may be placed before `d` or `f` if the datum is to be placed in a long or double, respectively. The `&` in front of the n is an "address operator", discussed below. Another commonly used function is `getchar()`, which has the format:

```
c = getchar();
```

It reads a single character from the standard input into the variable `c`.

5 Functions

A C **function** is a kind of a hybrid of a procedure and function. It can be called as a statement or in an expression. If it is called as a statement, any value returned is "thrown away". If it is called in an expression, it *must* return a value, else it is undefined.

Every C program has a function called `main()`, which is executed first. By default, `main` is a function of type `int`. Execution begins at the start of the `main` function. The order in which the other functions are declared does not matter, although traditional, "one-pass through the code" compiler, it is expected that a function or variable be declared before it is accessed. If a function is not declared before you call it, C assumes that the function will return an int. (Since square below returns an int, we don't have to declare it before it is called.)

An example:

```
#include <stdio.h>
main ()
{
      int s;
      s = square(2);
      printf("The square of 2 is %d  \n", s);
      return (0);
}
int square(int x)
```

[1] Note “codepad” (codepad.org) does not support external input. This is this simple environment, not about the C-language itself.

```
{
      return(x * x);
}
```

In `square()`, x is the parameter. If there were any local variables, they would be declared after the opening `{` (unlike Pascal, where they would be declared before the first `BEGIN`). As mentioned above, by default, a function returns a result of type `int`, but it is considered bad style not to make this explicit. Any other return type must be declared before the name of the function:

```
double square_root(x)
```

If the function definition *follows* the function call, then the value must be declared *before* the function call is encountered. In the example above, if the definition of square_root followed that of main, then main should include, after the {, the declaration

```
double square_root();
```

This line terminates with semicolon, but in the definition of the function, the header line *does not* terminate with a semicolon. If the function definition is in another file, then the function name must be declared within the calling function. If a function is not intended to return any value, then it may be declared to be of type void, like this:

```
void exchange (num1, num2)
```

In C all parameters are **passed by value** and therefore a C function can return a value only via the `return()` statement. An example incorporating many of the above points follows. The file `sqrt.c` contains the following:

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

main () {
  double x,y,z;
  double square_root ();

  printf("Enter a real number now (negative to stop)\n=> ");
  scanf("%lf",&x);
  while (x >= 0.0) {
    y = sqrt(x);
    z = square_root(x);
    printf("Square root of %.10f by built-in    = %.10f\n",x,y);
    printf("Square root of %.10f by my function = %.10f\n",x,z);
    printf("\nEnter a real number now (negative to stop)\n=> ");
    scanf("%lf",&x);
    return 0;
  }
}
```

The file `sqrtfunc.c` contains the following:

```
/* This function calculates a square root
   by the Newton-Raphson method.*/

double square_root (double number)
{
  double approx;
  double epsilon = .0000001;
  double absolute ();

  approx = number / 2.0;
  do
    approx = (number / approx + approx) / 2.0;
  while (absolute((number / (approx*approx)) - 1) >= epsilon);
  return(approx);
}

double absolute(double x)
{
  if (x < 0)
    x = -x;
  return (x);
}
```

6 Data Structures
6.1 Arrays
Java's array syntax is derived from C, so a C array looks quite familiar. We could declare

```
double vector[10];
```

There are two differences.
(1) A C array is not an object. The array `vector` exists as soon as it is declared.
(2) C doesn't test for subscript out of bounds. The compiler will let you say

```
vector[10] = x;
```

but this array element doesn't exist. The value of `x` is put into a memory location outside the bounds of the array. The results of this are unpredictable, but probably won't be good. Arrays are commonly dealt with by `for` statements. For example, to add the elements of you might say

```
for (i = 0; i < 10; i++)
      sum = sum + vector[i];
```

You can use an increment operator in an array reference:

```
vector[++i];            /* increments the subscript i */
++vector[i];            /* increments what is in vector[i] */
```

Multi-dimensional arrays look like Java's:

```
double matrix [2] [3];  /* 2 rows, 3 columns */
```

To pass an array as a parameter, use the name of the array without subscripts.

```
maximum(vector, num);
```

In the function definition, you can then include or omit the number of values in the array, the latter providing greater generality:

```
double maximum(double vector[], int num)
```

Data declarations may be made external to any particular function, including `main()`, and any function that uses these declarations should redeclare them within the function, using the keyword `extern`. If the externally declared data happens to be an array, the number of values may be omitted. For example:

```
int n;
double vector[50];
main ()
{
    extern int n;
    extern double vector[];
    ...
}
```

Like in Java, in C the reference to an array argument is always passed, so any changes made to the array in the called function carry back to the calling function. (This is another simulation of "pass by reference.")

6.2 Strings

A string in C (unlike Java) is really an array of characters. A **string constant** is enclosed in double quotes. A **character constant** is enclosed in single quotes. For variable length strings, C does not accompany each string with an integer giving the string's length, but uses a different method. Every string in C ends with the **null character**, a special unprintable character denoted by `\0`, to mark its end. Here is a function which uses this feature to determine the length of a string:

```
int str_length (char string[])
{
      int count = 0;
      while (string[count] != '\0')
            ++count;
      return(count);
}
```

`printf()` uses `\0` to know when to stop printing. This character is added automatically to any string enclosed in double quotes, but otherwise you must remember to put it in yourself, or you may get "unpredictable results". For example, here is a function to concatenate two strings. The null character is put onto the end of the concatenated string once.

```
concatenate (char str1[], char str2[], char result[])
{
      int i, j;
      for (i=0; str1[i] != '\0'; ++i)
            result[i] = str1[i];
      for (j=0; str2[j] != '\0'; ++j)
            result[i+j] = str2[j];
      result[i+j] = '\0';
}
```

Since strings are really arrays, they cannot be directly compared or assigned by simple statements like `if (a == b)` or `a = b`. Here is a function which copies one string into another ("assigns" one to the other):

```
/* copy t to s; simulates the assignment "s = t;" */
strcpy(char s[], char t[])
{
      int i = 0;
      while ((s[i] = t[i]) != '\0')
            i++;
}
```

Look carefully at the test in the `while`. It is typical of "real C programming". It takes advantage of the fact that an assignment is an expression, and so embeds it within the inequality test. The assignment not only makes the assignment, but also *returns* the value that is being assigned, just as any other expression returns a value. All of those parentheses are necessary, because `!=` has a higher precedence than `=`.

6.3 Structures

The C equivalent of a Pascal `RECORD` is called a **structure**, which is divided into fields called **members**, and has an identifying name called a **structure tag** (the equivalent of a type name in Pascal). The structure is declared using the format:

```
struct <tag>
        {
                <type 1> <variable 1>;
                ...
                <type n> <variable n>;
        };
```

For example, the following are two ways to declare a variable instance of type `struct sample`.

```
struct sample                               struct sample
        {                                           {
                int integer;                                int integer;
                double real;                                double real;
                char character;                             char character;
        };                                          } instance;
struct sample instance;
```

7 Pointers

C pointers can point to anything that they have been defined to point to. A **pointer** is denoted by an identifier preceded by an asterisk, called the **indirection** operator, like the "up-arrow" of Pascal. Available in C (but not in Pascal) is the **address** operator, denoted by an ampersand, which precedes the identifier whose address is desired. Consider this example:

```
double num1, num2;
double *ptr;
ptr = &num1;
num2 = *ptr;
```

The second line means that `ptr` is a pointer that points to real numbers (not that it is a real number that is somehow being used as a pointer). The third line uses the address operator to find the address of `num1`, and then assigns that address to `ptr` (in other words, it makes `ptr` point to `num1`). Note that `num1` does not have to be brought into existence by `NEW`, as it would in Pascal, and that it has its own name, which it would not in Pascal. The fourth line assigns what `ptr` points to (that is, `num1`) to `num2`. The `*` is not used in line three, because `ptr` itself is being modified, but is used in line four, where the value that `ptr` is pointing to is being used. In this simple example, the net result is the same as the assignment `num2 = num1;`.

Pointers can also point to structures, which makes linked data structures possible in C. For example:

```
struct node
{
        int field1;
        ...
};
struct node *nodep;
```

```
    nodep = (struct node *) malloc (sizeof (struct node));
```

The sixth line means that `nodep` is a pointer which points to a structure of type `struct node.` (C programmers often end pointer names with "p".) The last line creates in memory a structure of type `struct node`, and points `nodep` at it. It is the equivalent of the Pascal code:

```
    NEW (nodep)
```

The `malloc(n)` (Memory ALLOCate) is a built-in function, which returns `n` bytes of memory. `sizeof(x)` is a built-in function which returns the size of whatever item `x` is defined to be; hence in this case, the size of a structure of type `struct node`. Thus the result of `malloc (sizeof (struct node))` is to return a block of memory equal in size to that of one structure of type `struct node`. However, `malloc()` expects to return a pointer to a character, so it is preceeded by `(struct node *)`, a **cast**, and it changes the type of whatever follows it to that in the cast. In this case, it changes the type that `malloc()` returns to that of a pointer (the `*`) to a structure of type `struct node`.

To refer to a member of node called `field1` C, we could use the construct:

```
    (*nodep).field1
```

where the asterisk before `nodep` in C means that `nodep` is to be dereferenced. The parentheses are needed because the period has a higher precedence than `*`. But C programmers hardly ever use this construct, preferring the exactly equivalent one:

```
    nodep -> field1
```

This means "the member called `field1` in the structure being pointed to by `nodep`".

Earlier we said that all parameters to C functions are passed by value. Here are two functions to swap two values. The calls:

```
  /* WRONG !!! */                        /* RIGHT !!! */
  exchange( num1, num2);                 exchange( &num1, &num2);
```

and the functions:

```
  /* WRONG !!! */                        /* RIGHT !!! */
  void exchange (int num1, int num2)     void exchange(int *num1, int *num2)
  {                                      {
     int temp;                              int temp;
     temp = num1;                           temp = *num1;
     num1 = num2;                           *num1 = *num2;
     num2 = temp;                           *num2 = temp;
  }                                      }
```

The function on the left *will not work!* But if we call the function with address operators on the arguments, and if in the function declaration we use indirection operators on the parameter declarations, then the function will work as expected. (This amounts to a way of turning value parameters into reference parameters.) By now you should be able to figure out for yourself why the `scanf()` function which we discussed above had to have `&n`, not just `n`, as the variable into which an integer would be read.

To declare a pointer to an array, use the type of what's in the array, not the type of the array.

```
int values[10];
int *ptr;
ptr = values;
```

Note that the address operator `&` is not used in the third line, because an array name without a subscript *is* a pointer. To pass an array to a function, we just used its name, and we could change its contents (as in pass by reference). That's because the array name is a pointer to the array. Hence inside a function we can use the name of the array and a subscript, or we can use a pointer to the array.