Mathematics U11601: C programming handbook

Last update September 26, 2021

1 Header files

```
#include <stdio.h>
  printf, scanf, fgets, snprintf, fopen, fclose

#include <stdlib.h>
  atoi, atof, exit, malloc, calloc, free, abs, fabs, srand48, drand48

#include <string.h>
  strlen, strcmp, strncmp

#include <math.h>
  sqrt, sin, etc.  NOTE: gcc requires -lm
```

2 Main program

```
int main (){...} or
int main ( int argc, char * argv[] ){...}
```

3 Files and input/output

```
fopen ( file_name_string, "r/w/a" );
   returns a file pointer value
fclose ( file );   Essential for writeable files.

standard files: stdin, stdout, stderr.

printf ( "format string", item_1, ... );

scanf ( "format string", item_1, ... );
  returns a value; item_1 ,... MUST BE addresses
fgets ( buffer, size, input file );
   returns a value; reads a line of text.

fscanf( file, .... )
fprintf( file, ... );
```

3.1 Format control items

```
printf, snprintf format control items %d %c %f %g %s
   These are in the simplest form.
   The %g format control item will choose either %f format
   or %e format (scientific notation, which we shall not study),
   whichever fits better.
   Also, it prints numbers like 1.0, which have no fractional part,
   as integers without the decimal point.  This can be useful.

printf("%f\n", 1.0) prints 1.0000000
printf("%g\n", 1.0) prints 1

   For printf, one
   can control the field width, number of digits, etcetera.

For scanf, one MUST use %lf when inputting a double-precision variable.
```

In `printf` and `scanf`, *format control items are matched with items to be printed.* It is up to the programmer to make sure these are correct.

```
printf("%d\n",1.0); prints
472707256
```

3.2 snprintf

```
snprintf( target string, length bound, "format string", item_1, ... );
```

This doesn't write to an output file; it is just a way of arranging data in a character string; it means 'string bounded length quasi-printf.'

4 Arithmetic expressions

```
+ - * / %
'bodmas' precedence rule brackets, division, multiplication,
      addition, subtraction.
also abbreviations like x++, --y, x += 4, y /= 10
```

Integer division rounds towards zero. This has a knock-on effect with the remainder operator $\%$.

If a and b are positive integers, then a/b is rounded down to the nearest integer. If $a < 0$ and $b > 0$, both integers, then a/b is rounded up to the nearest integer. This is 'rounding towards zero.'

Ignore the case $b < 0$: it is confusing and not useful.

If a and b are integers, and $b > 0$, then

$$a = (a/b) * b + a\%b$$

It follows that $a\%b$ is nonnegative if a is nonnegative and it non-positive if a is non-positive.
Thus the remainder can be negative, which differs from mathematical convention.
Assignment statement:

```
  x = 2 * y + x;
Variant
  x += 2*y;  +=,*=, etc, are ‘assignment operators.’
```

5 Equality versus assignment

```
  x = 1;
     is an assignment statement.  It assigns the value 1 to x.

  x == 1
     is a condition, true when x is 1 and false otherwise.


  If one forgets this, and writes = instead of ==, it may
  cause strange program errors.
```

6 Control flow

```
  logical connectives
    < <= == >= > != && || !

  No boolean values: integers; zero false,
    nonzero true.

  for ( i=0; i<10; ++i )
  { sum = sum+a[i]; }

  while ( scanf ( "%d", &n ) == 1 )
  { }
       Don’t use ‘!= 0’
       because some systems return -1 at end of data.

  ok = ( n == 10 );
  if ( ok && a[9] < 0 )
    if ( a[8] >= 0 )
    { lastvalue = a[8]; }
    else
    { ok = 0; }
       two ifs: is the indentation correct?
```

7 Data types

```
Basic:
  char        character (1 byte)
  char [ .. ] character string
  short
  int
  float
  double
  address ( indicated by * )
```

8 Arrays

```
  int a[15];    array of 15 ints, starting address in a.
  double b[3][4] 2-dimensional array of 3x4 doubles,
        starting address in b.

  In C, arrays and pointers are similar.
```

9 Names for variables (and functions and types)

These names are called *identifiers*. Any nonempty string of letters, digits, and underscore _ is a valid identifier, *provided it doesn't begin with a digit.*

Small letters and capital letters are *different.*

```
  int aB, Ab, _aB, __Ba_a;
```

is correct in C, *though of course* one should not use weird or eccentric names for variables.

In naming variables it is common to use small letters, reserving capital letters for the names of structure types.

9.1 Mixed types

- Constants have implicit types. So, 1.0 is implicitly double, 1 is int.
- With expressions of mixed type, conversion is applied so that types match. Actually, 'promotion' occurs. Where int and double are to be combined, the int is converted, so we get the combination of two doubles.
- In C, *this is very unusual,* **char** types are considered as 8-bit integer and if necessary converted to **int**. On Intel machines, it is even odder: *sign extension* is applied, so if a char has high-order bit 1, it converts to a negative int.

10 Operator precedence

```
Level 1 LR   [],  .,  ->, postfix ++, --,  has value and effect
      2 RL   !, * (dereference), & (address), casts, sizeof,
             prefix ++ --, has value and effect
      3 LR   *, /, %  arithmetic
      4 LR   +, -    arithmetic
      5 LR   <, <=, >=, >
      6 LR   ==,     !=
      7 LR   &&
      8 LR   ||
      9 RL   =, +=, *=, %=, etcetera: assignment operators
              Assignment operators have value and effect
```

11 Functions and subtroutines, and prototypes

```
  type name ( type arg1, type arg2, ... ){..}
    for a function with given name.  type: int, double, etc.

  void name ( type arg1, type arg2, ... ){..}
    for a subroutine

  Prototypes give the return type (which can be void)
  and the arguments, while the full function or routine
  is written somewhere else.  For example

  void printhex ( int n, char v[] );

  describes the routine, without giving the routine
  body.  A semicolon replaces the part between braces,
  {...}
```

Arguments are always *call-by-value,* though because of C treating arrays as pointers, arrays are effectively passed by reference.

12 Structures

```
typedef struct { double re, im; } COMPLEX;

  COMPLEX a = {1, 2};
  COMPLEX * b = (COMPLEX*) calloc(1, sizeof(COMPLEX));
  COMPLEX * product ( COMPLEX *a, COMPLEX *b );
```

```
typedef struct { int m,n; double ** entry; } MATRIX;
  MATRIX a, *b;
```

13 Dynamic memory allocation

```
for example
  COMPLEX * new = (COMPLEX *) calloc ( 1, sizeof ( COMPLEX ) );
---------------
A full example, where memory is allocated and a copy made

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  char hello[] = "hello";
  int size = strlen ( hello ) + 1; // room for null character
  char * copy = (char*) calloc( 1, size );
  snprintf( copy, size, "%s", hello );
  printf("Original %s\n", hello);
  printf("Copy %s\n", copy);
}

% a.out
Original hello
Copy hello
```

14 Random number generators

Linear congruential

$$X_{n+1} = aX_n + c \mod N$$

```
  lrand48() % k    for random numbers 0..k-1?  Not
    recommended if k is even with a linear congruential
    generator, but it seems not to matter nowadays.

  (int) (drand48() * k) is recommended.

  srand48() to set the seed.  ONCE ONLY.  Repetitions will
    destroy randomness.

  Setting seed from clock:
```

```
#include sys/time.h
struct timeval tv;
gettimeofday ( & tv, NULL );
srand48 ( tv.tv_usec );
```