CS 240 Programming in C

Constants, Symbolic Constants, Variables, printf, getchar

September 11, 2019

Schedule

1. Constants or Literals
2. Format printing – printf
3. Symbolic Constants
4. getchar()

Constants

- Constants or literals are fixed value in a program.
- There are four basic constant data types in C:
  1. Integer constant/literal
  2. Floating number constant/literal
  3. Character constant/literal
  4. String constant/literal

Integer Constant

- An integer literal can be a decimal, octal, or hexadecimal constant.
- A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.
- An integer literal can also have a suffix that is a combination of U/u and L/l, for unsigned and long, respectively.
- The suffix is not case sensitive.

Integer Constant Example!

```
10,         -10             /* decimal/int */
010,        -010            /* octal       */
0x1A,       -0x1A,          /* hexadecimal */
10u                         /* unsigned int */
10l,        -010l           /* long */
10ul,       10lu            /* unsigned long */
```

Note:

1, the relative position of l and u does not matter.

2, for an signed number the first left bit is the sign bit, if it is 1 means negative if it is 0 means positve.

3, for example, if we know there are 1 Byte for an integer, then,

```
 1    0000,0001
-1    1000,0001
```

Integer Range

- As we know how to represent integer constant in C, what it the value range of it, in another words, what defines or constrains the value range of an integer.
- The answer is the number of bytes used to store it.

- The range is different between different system, but it is fixed when it gets set up.
- we can access it by the function of sizeof().
- Let's see.

Integer Constant Example!

```
#include <stdio.h>
int main(void)
{
    printf("the size of a decimal is :%d Bytes\n", sizeof(10));
    printf("the size of a octal is :%d Bytes\n", sizeof(010));
    printf("the size of a hexadecimal is:%d Bytes\n", sizeof(0x10));
    printf("the size of a negative decimal is:%d Bytes\n", sizeof(-10));
    printf("the size of a long decimal is :%d Bytes\n", sizeof(-10L));
    printf("the size of a long variable is :%d Bytes\n", sizeof(long));
    printf("the size of long long is :%d Bytes\n", sizeof(10ll) );

}
```

And they are all 4 bytes except long long integer which is 8 bytes

Check what the differences between int and long in C. https://stackoverflow.com/questions/271076/what-is-the-difference-between-an-int-and-a-long-in-c

Integer Constant Example!

- As we already covered that 4 Bytes = 32 bits.

- So,

```
for integer,          [-2^31, 2^31-1]
for unsigned integer, [0, 2^32-1]

2^32 - 1 = 4294967295
2^32     = 4294967296
2^32 + 1 = 4294967297
```

Unsigned Integer Range

What will this program print out???

```
#include <stdio.h>
int main(void)
{
    printf("%u\n", 4294967295);          2^32 - 1
    printf("%u\n", 4294967296);          2^32
    printf("%u\n", 4294967297);          2^32 + 1
    return 0;
}
```

Note: %u in printf means get an unsigned integer value.

Unsigned Integer Range

```
#include <stdio.h>
int main(void){
    printf("%u\n", 4294967295);          4294967295
    printf("%u\n", 4294967296);          0
    printf("%u\n", 4294967297);          1
    return 0;
}
```

Why?

Unsigned Integer Range

Because unsigned integer constants only take 4 Bytes.
The extra left bits are ignored simply.

```
4294967295 =   1111_1111_1111_1111_1111_1111_1111_1111
4294967296 = 1_0000_0000_0000_0000_0000_0000_0000_0000
4294967297 = 1_0000_0000_0000_0000_0000_0000_0000_0001
```

By the way, what kind of error should this be categorized in for all the three kinds of errors that we covered?

Signed Integer Constant Range

What about the lower range of unsigned integer?
What will this program print out???

```
#include <stdio.h>
int main(void)
{
    printf("%u\n",  0);
    printf("%u\n", -1);
    printf("%u\n", -2);
    return 0;
}
```

Signed Integer Constant Range

It goes to the biggest unsigned integer 2^32 - 1 = 4294967295,

What will happen if it continues going smaller negative?
I don not have an answer.
So be careful when deal with this situation, you may get unexpected error.

```
#include <stdio.h>
int main(void)
{
    printf("%u\n",  0);        0
    printf("%u\n", -1);        4294967295
    printf("%u\n", -2);        4294967294
    return 0;
}
```

Signed Integer Constant Range

What about the upper range for signed integer?
What will this program print out???

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", 2147483647);          2^31 - 1 =  2147483647
    printf("%d\n", 2147483648);          2^31     =  2147483648
    printf("%d\n", 2147483649);          2^31     =  2147483649
    return 0;
}
```

Note: %d in printf means get a signed decimal integer value.

Signed Integer Constant Range

```
#include <stdio.h>
int main(void){
    printf("%d\n", 2147483647);
    printf("%d\n", 2147483648);
    printf("%d\n", 2147483649);
    printf("%d\n", 2147483650);
    return 0;
}
```

```
 2147483647
-2147483648
-2147483647
-2147483646
```

Why?

Signed Integer Constant Range

```
 2147483647 =   0111_1111_1111_1111_1111_1111_1111_1111U
 2147483648 =   1000_0000_0000_0000_0000_0000_0000_0000U
 2147483649 =   1000_0000_0000_0000_0000_0000_0000_0001U
 2147483650 =   1000_0000_0000_0000_0000_0000_0000_0010U

 2147483647 =   0111_1111_1111_1111_1111_1111_1111_1111
-2147483648 =   1000_0000_0000_0000_0000_0000_0000_0000
-2147483647 =   1111_1111_1111_1111_1111_1111_1111_1111
-2147483646 =   1111_1111_1111_1111_1111_1111_1111_1110
```

For taking 2147483648U as signed integer, it is -2147483648.
For taking 2147483649U as signed integer, it goes back to positive from -2147483648 by the difference to 2147483648U.

Why it is this way? I have no clue.

Signed Integer Constant Range

What about the lower range for signed integer?
What will this program print out???

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", -2147483647);      -2^31 + 1 = -2147483647
    printf("%d\n", -2147483648);      -2^31     = -2147483648
    printf("%d\n", -2147483649);      -2^31 - 1 = -2147483649
    printf("%d\n", -2147483650);      -2^31 - 2 = -2147483650
    return 0;
}
```

Note: %d in printf means get a signed decimal integer value.

Signed Integer Constant Range

```
#include <stdio.h>
int main(void){
    printf("%d\n", -2147483647);          -2147483647
    printf("%d\n", -2147483648);          -2147483648
    printf("%d\n", -2147483649);           2147483647
    printf("%d\n", -2147483650);           2147483646
    return 0;
}
```

Why?

Signed Integer Constant Range

```
-2147483647 =   1111_1111_1111_1111_1111_1111_1111_1111U
-2147483648 =   1000_0000_0000_0000_0000_0000_0000_0000U
-2147483649 =   ?
-2147483650 =   ?
```

Signed Integer Constant Range

What about a integer number bigger than 4 bytes?
What will this program print out???

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", 4294967295);
    printf("%d\n", 4294967296);
    printf("%d\n", 4294967297);
    return 0;
}
```

Signed Integer Constant Range

```
#include <stdio.h>
int main(void){
    printf("%d\n", 4294967295);          -1
    printf("%d\n", 4294967296);          0
    printf("%d\n", 4294967297);          1
    return 0;
}
```

Why?

Signed Integer Constant Range

```
printf("%d\n", 4294967295);          -1
printf("%d\n", 4294967296);          0
printf("%d\n", 4294967297);          1
```

```
4294967295 =   1111_1111_1111_1111_1111_1111_1111_1111
4294967296 = 1_0000_0000_0000_0000_0000_0000_0000_0000
4294967297 = 1_0000_0000_0000_0000_0000_0000_0000_0001
```

For 4294967296, 4294967297, it makes sense. The extra left bit is ignored.

But for 4294967295 prints out -1, I do not have an answer.

Integer Constant

Now let's print out some octal and hex numbers.

```
#include <stdio.h>

int main(void)
{
    printf("%d\n", 10);       /* d means decimal*/
    printf("%d\n", -10);
    printf("%o\n", 010);      /* o means octal*/
    printf("%o\n", -010);
    printf("%x\n", 0x1A);     /* x means hexadecimal*/
    printf("%x\n", -0x1A);
    return 0;
}
```

Integer Constants

- What does this program print out?

Integer Constant

```
#include <stdio.h>

int main(void)
{
    printf("%d\n", 10);          out: 10
    printf("%d\n", -10);         out: -10
    printf("%o\n", 010);         out: 10
    printf("%o\n", -010);        out: 37777777770
    printf("%x\n", 0x1A);        out: 1a
    printf("%x\n", -0x1A);       out: ffffffe6
    return 0;
}
```

Integer Constant

Are there some peculiar things for this?
If so what are these?

```
printf("%d\n", 10);          out: 10
printf("%d\n", -10);         out: -10
printf("%o\n", 010);         out: 10
printf("%o\n", -010);        out: 37777777770
printf("%x\n", 0x1A);        out: 1a
printf("%x\n", -0x1A);       out: ffffffe6
```

Integer Constant

- The first thing is "%o" and "%x" do not print out the prefix of the number system.
- Second, "%o" and "%x" do not print out negative octal and hexadecimal numbers.
- They take them as positive binary number and then use $2^{[bits]}$ subtract this positive number to get the converted positive octal and hexadecimal numbers.
- [*bits*] means the total number of bits of the integer constants.

- In this way when you add a negative octal or hexa and the result is more than 0, it is equivalent to add the stored positive octal or hexa number.
- The octal and hexa reading of numbers do not directly keep the sign of the numbers.

For example

```
printf("%x\n", 0x1A);                    out: 1a
printf("%x\n", -0x1A);                   out: fffffe6

0x1A + 0xffffffe6 = 0x1,0000,0000
0x1,0000,0000 =
b1,0000,0000,0000,0000,0000,0000,0000,0000

0xffffffe6 = 0x1,0000,0000 - 0x1A
the extra bit 1 will be ignored since it exceeds the
maximum length of the integer storage.

Or do a bit wise not operation and add one:
0xffffffe6 = ~0x1A + 1
```

For example

```
#include<stdio.h>
int main()
{
    printf("%x\n",    -0x1A);
    printf("%x\n", 0x0-0x1A);
    printf("%x\n",   0-0x1A);
    printf("-0x1A gets printed out as: %x\n", ~0x1A+1);
}
```

~ is a bit wise operation.

And the same rule goes with hexadecimal numbers.

Illegal Constant Number Expressions

1, Besides the range of constant number, there are also illegal number expression we need to pay attention.

For example: 078

What's wrong with this expression?

Illegal Constant Number Expressions

```
For example:        078

    Yes, for a octal number, 8 is not one of its digit system.
    This kind of error is insidious, seems right but wrong.

    Decimal:        {0,1,2,...,9}
    Octal   :       {0,1,2,...,7}
    Hexadecimal   : {0,1,2,...,9, A,..,F}

    Also the suffix of U  should not repeat itself.
    123UU is illegal.
    But L can, which mean long long number.
    For example, 123ll or 123LL.
```

Floating-point Constants

3.1415926, 31425927E-5 or 31425927e-5

A floating-point constant can be either in decimal form or exponential form.

The size of my system for storing a floating point number is 8 bytes.
For the decimal floating number, we have to have integer part,
For the exponential form, we must include the integer part, the e/E and the exponent part.

324, 314E are illegal floating numbers.
Try them, see what will happen.

```
    printf("%f\n", 314);
    printf("%f\n", 314e);
```

Floating-point Constants

```
#include <stdio.h>

int main(void)
{
    printf("%f\n", 3.14);
    printf("%f\n", 314e5);
    printf("the size of a floating number is
                :%d Bytes\n", sizeof(3.14));
    printf("the size of a floating number is
                :%d Bytes\n", sizeof(314e5));
    return 0;
}
```

Character Constants

- Character constants are enclosed in single quotes, e.g., 'x'.
- There are plain character like 'x', and also escape character '`\n`'.

Character Constants

Try this program yourselves and figure out what it means.

```
#include <stdio.h>

int main(void)
{
    printf("%c\n", 'x');
    printf("%c\n", '\t');
    printf("%c\n", '\n');
    printf("the size of a character number is :%d Bytes\n",
                                        sizeof('x'));
    printf("the size of a character number is :%d Bytes\n",
                                        sizeof('\n'));
    return 0;
}
```

Character Constants

Here is a list of escape character:

\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh...	Hexadecimal number of one or more digits

String Constants

- String constants are enclosed in double quotes.
- And they are just a sequence of character constants

printf

- After we introduced the constants in C, let's go through the printf function.
- cheat sheet `https://alvinalexander.com/programming/printf-format-cheat-sheet`

Symbolic Constants

- Recall that the preprocessor deals with # commands before actual compilation begins
- `#define IDENTIFIER value`
  The C preprocessor goes through the source code and substitutes every IDENTIFIER with value
- A `#define` line defines a symbolic constant to be a particular string of characters
- Convention requires the name of the IDENTIFIER to be all capital letters to mean constants
- Why use a symbolic constant
  - conveys information about numerical constants
  - allows you to easily update them in one place
- Symbolic constant
  - no memory location assigned to hold value (this is a declaration)
  - not an executable statement (no semicolon at end of line)
  - value is known at compilation

Symbolic Constants

- In a function body, we can also define symbolic constants, by the key word const with data type key word.

  For example:

  ```
  const float PI = 3.14;
  ```

  Don't forget the ";", since this expression in a function body is executable.

Fahrenheit to Celsius, Version 1

```
#include <stdio.h>

/* print Fahrenheit-Celsius table for
  fahr = 0, 20, ..., 300 */
int main(void) {
  int fahr, celsius;
  int lower, upper, step;

  lower = 0;   /* lower limit of temperature table */
  upper = 300; /* upper limit of temperature table */
  step  = 20;  /* step size */
  fahr  = lower;
  while (fahr <= upper) {
    celsius = 5 * (fahr - 32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

Variable Declaration

- Variable
  - Name given to a storage area that our programs can manipulate
  - Lowercase by convention
- Declaration
  - Announces the properties of a variable
  - Consists of:
    1. type name
    2. list of identifiers (variable names)
- Type
  - Classification identifying a type of data (`int` for an integer variable)
- All variables must be declared before they can be used
- No memory is allocated by declaration alone
- Example

```
extern int i; (we will explain what it does, why we need
```

An example of declaration that is not definition

Definition

- Definition for an object
  - causes storage to be reserved for that object
- Definition of an enumeration constant or a `typedef` name
- is the (only) declaration of the identifier
- Example

```
int upper;
int lower = 0;
```

Both lines are definitions (because memory is allocated)
The second also initializes the variable with value 0

Assignment

- Change the value of a variable
- Evaluation precedence rules: expression will be evaluated from left side.
- An assignment is an expression which also has a value and it is the value on the left side of "="
-

```
int lower;
int i, j, k;

lower = 0;
i = j = k = 2
```

Try

```
#include <stdio.h>

int main(void)
{
    printf("%d\n", EOF );
    int i,j,k;
    printf("%d\n", i=j=k=2 );
    printf("%d\n", (i=j)==(k=2) );
    printf("%d\n", i = (i=j)==(k=2) );
    printf("%d\n", i = (i=j)!=(k=2) );
    printf("%d\n", i = 1!=2 );
    printf("%d\n", i = 1==2 );

}
```

True and False

- C does not have a Boolean type
- True: any non-zero numerical value
- False: 0
- Logic expression.

```
<           less than
<=          less than or equal
>           greater than
>=          greater than or equal
==          equal
!=          not equal
```

- All the logic expression will have a value either 0 or 1, depending on whether the statement is false or true.

While Loop

```
while (logical expression) {
  statements
  //execute while logical expression is true
}
```

- The logical expression is tested
- If it evaluates to true, the body is executed and then the logical expression is tested again
- When the test becomes false, the loop ends
- Note: no statements are executed if the logical expression is false upon entry
- Below, the loop executes until `fahr > upper`

```
while (fahr <= upper) {
  statements
}
```

Fahrenheit to Celsius, Version 2

```
#include <stdio.h>

#define LOWER 0
#define UPPER 300
#define STEP  20

/* print Fahrenheit-Celsius table for
  fahr = 0, 20, ..., 300 */
int main(void) {
  int fahr;

  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
    printf("%3d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
  return 0;
}
```

For Loop

```
for (initialization; condition; update)
  statement

for (initialization; condition; update) {
  statements
}
```

- initialization: executed once when the loop is started
- condition: the loop test (when to stop looping)
- update: a statement to execute at the end of each loop (usually an increment or decrement)

Functions

- A function is a separate block of code that you can call as part of your program
- A function executes and returns to the next line after you call it in your program
- You can provide a function with arguments inside parentheses following the function name:
  `functionName(arguments);`
- Arguments are passed by value (we will talk about this more later)
- A return value may be passed back:
  `returnValue = functionName(arguments);`

printf()

- A general-purpose output formatting function
- It takes a variable number of arguments, instead of a predefined number of arguments
- The first argument in a call to `printf()` determines the total number of arguments the call requires
- In the first argument, you provide a string of characters that will contain:
  - Literal characters: these print as they appear in the string, "example 1"
  - Escape characters: used for hard to type string elements, "\n"
  - Replacement characters: printf will replace these, "%6.1f"
- For each set of replacement characters in the first argument, there must be a corresponding argument following the first argument
- When printf prints to stdout, it will substitute the replacement characters with the corresponding arguments, using the formatting specified

I/O Redirection

- Output redirection
  - Store the output of a process to a file
  - `$ command > fileName`
    sends standard output of command to the file with the name fileName
  - `$ command >> fileName`
    appends standard output of command to the file
- Input redirection
  - Use the contents of a file as input to a process
  - `$ command < fileName`
    executes command using the contents of the file fileName as its standard input
- Input/Ouput redirection will not count as the command's arguments.

Types

- Sizes reflect the MIC server system(a server that professor Ouyang once used), but may change based on implementation:
- char: 1 byte, capable of holding one character
- int: 4 bytes, holds an integer cannot be longer than a long
- short: often 2 bytes (must be at least 2 bytes), holds an integer cannot be longer than int
- long: 8 bytes (must be at least 4 bytes), holds a long integer
- long long: 8 bytes (must be at least 8 bytes)
- float: 4 bytes, holds a single-precision floating point number
- double: 8 bytes, holds a double-precision floating point number
- signed int: just like int
- unsigned int: or just unsigned
- Read `/usr/include/limits.h`

Char I/O

- Defined in stdio.h
- `getchar()`: Each time it is called, getchar reads the next input character from a text stream and returns that as its value
- No arguments are passed to getchar, it gets input from stdio
- getchar() returns an int value:
  1. the integer value of the character
  2. -1, if the input stream or input file comes to an end.

  `Note: for different systems, there are different judgements of end of input stream, Ctrl+D for Unix, Crtl+Z for Windows. And there are also ways of verdicting the end of file, but as long as getchar() recognize an end of input character stream of file, -1 will be returned.`
- `putchar(c)`: prints the contents of the integer variable c as a character

Copying Input to Output, Version 1

- Pseudocode

```
read a character
while (character is not the end-of-file indicator)
  output the character just read
  read a character
```

- C code

```
#include <stdio.h>
int main(void) {
  int c;
  c = getchar();
  while (c != EOF) {
    putchar(c);
    c = getchar();
  }
}
```

Copying Input to Output, Version 2

```
#include <stdio.h>
int main(void)
{
  int c;
  while ((c = getchar()) != EOF)
    putchar(c);
}
```

- Takes advantage of the fact that an assignment is an expression and has a which is the value on the left side of "="
- EOF is actually integer -1.
- Note: There is no character's value is negative
- -1 can be input by Ctrl+d on Unix and Ctrl+z on Windows
- the "-1"'s integer value is not -1, it's the value of '-' and '1'
- Note the parentheses around c = getchar()
- They are necessary because of precedence rules: expression will be evaluated from left side.

Try it

- Try this program.
- Try this program with a text file as input file.

```
$ ./getchars < hello.txt

where hello.txt has the content:
Hello World!
```

Try this also

```
Question: What will be printed out at the last line?
          What about the second last line?

#include <stdio.h>
int main(void)
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
    putchar(c);
    printf("\n%d\n", c);
    return 0;
}
```