

# CS 240 Programming in C

## Constants and Variables

January 4, 2020

# Schedule

- 1 Variables
- 2 Constants
- 3 Format printing – printf

- As we covered before, variables in C are just names to memory addresses.
- And for different data types of a certain variable, the memory allocated for it is different.
- Specifically, there is a void data type of a variable, which only contains an memory address, but not the amount of memory it owns.
- Void data type can be

# Void Data Type

- Specifically, there is a void data type of a variable, which only contains an memory address, but not the amount of memory it owns.
- Void data type can be type cast to other data types, for which the corresponding data structure will be applied to the variable.

# Basic Data types

Today, we are getting at character data type (char), and arithmetic data types (integers and floating-point numbers.)

- Char type in C takes up only 1 byte of memory
- There are signed char and unsigned char, whose values are range from -128 to 127 or 0 to 255. For example,

```
char c1 = 65;  
unsigned c2 = 65;
```

So, a char variable is essentially an one byte integer, why and how it is used to represent a letter?

# ASCII Table

- Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@'.
- To utilize the letter meaning of a char variable, we have to treat it associated with ASCII table.
- In standard library stdio, there are function like putchar and printf which will print out the letter of a char onto screen. Lets see a demo.



# Integer

Based on how large memory an integer variable can hold, there are 3 types of integer in C:

short	2 bytes
int	4 bytes
long	8 bytes

These are signed integer types, for each there are also unsigned:

unsigned	short	2 bytes
unsigned	int	4 bytes
unsigned	long	8 bytes

# Integer

- 1, Things are simple when integers are all positive.
- 2, Things become complex when there are negative integers.
- 3, Because CPU calculates the equivalent addition for a subtraction which results computer stores a negative as its 2's complement form.
- 4, Let's take an example

# 2's Complement

1, Assume we have an data type of `int_4` which contains 4 bits and the first bit is a signed bit, for which 0 stands for positive and 1 for negative. For example:

```
int_4 a = 3; // 0011
int_4 b = -2; // 1010 (without 2's complement form)
```

# 2's Complement

2, The process of 2's complement for negative integers,

-2	1010	(without 2's complement form)
	1101	1's complement form
	1110	2's complement form

3, 2's complement of a positive stays the same.

# 2's Complement Addition

4, let's do  $a + b$  with addition of their 2's complement form.

variable	value	2's complement form
a	-2	1110
b	3	+ 0011
-----		
		10001

# 2's Complement

5, Since `a` and `b` are of `int_4` type which only contains 4 bits, the result 10001 will be cut to 0001 which is 1.

6, Thus  $a + b == 3 - 2 = 1$

7, Let's do another example by letting  $a = -2$ ,  $b = -1$ :

# 2's Complement Addition

Example:  $a = -2$ ,  $b = -1$

variable	value	2's complement form
a	-2	1110
b	-1	+ 1111
-----		
		11101

Let's compute -3's 2's complement form:

-3	1011	(without 2's complement form)
	1100	1's complement form
	1101	2's complement form

# Overflow of Integers'

- Arithmetic data types in C has fixed memory storage, which means if the value to be assigned is larger than a variable can hold, it will just discard those highest bits.
- Like in the above demo which 11101 being cut into 1101, we will see some demo in real C.
- The implementation overflow of unsigned integer is defined behavior, however the overflow of signed integer is not, or implementation-defined behaviour.



# Undefined Behaviour

- In C, some expressions yield undefined behavior. The standard explicitly chooses to not define how a compiler should behave if it encounters such an expression.
- As a result, a compiler is free to do whatever it sees fit and may produce useful results, unexpected results, or even crash.
- Code that invokes UB may work as intended on a specific system with a specific compiler, but will likely not work on another system, or with a different compiler, compiler version or compiler settings.
- For portability, try avoid them

# In memory -0

- In memory -0 means in memory bit level of 1000,0000 for char, and 1000,0000,0000,0000 for short.
- Not the same thing with -0 in your code.
- In memory -0 for a signed integer type, will be treated differently based on machine's architecture. Page 36 in text book.
- It is treated as the least negative number of its data type on a 2's complement machine, for example, -0 for a char type is -128.

# Demo of integer overflow

- Integer overflow and initialization are implementation-dependent behaviour.
- Turn to p257 on our text book.
- header `<limits.h>` defines constants for the sizes of integral types.

# Demo of integer overflow

What will be printed out ?

```
int main(void)
{
    printf("%lu\n",sizeof(int) );
    printf("This if for signed:\n");
    printf("    INT_MAX : %11d\n", INT_MAX);
    printf("1 + INT_MAX : %11d\n", INT_MAX + 1);
    printf("    INT_MIN : %11d\n",INT_MIN);
    printf("INT_MIN - 1 : %11d\n",INT_MIN - 1);
    printf("This if for unsigned:\n");
    printf("    INT_MAX : %11u\n", UINT_MAX);
    printf("1 + INT_MAX : %11u\n", UINT_MAX + 1);
    printf("    INT_MIN : %11u\n",0);
    printf("INT_MIN - 1 : %11u\n", - 1);
    return 0;
}
```

# A pitfall

- We have not talk about typecast yet, but it is a good reminder to make here.
- It is usually not a good idea to mix signed and unsigned integers in arithmetic operations.
- It is better practice to first cast unsigned to signed and then do operations.
- It is often wrong to cast negative to an unsigned integer.

What will be printed out ?

```
int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b)
        printf("a is more than b");
    else
        printf("a is less or equal than b");

    return 0;
}
```

# Fixed Width Integer Types

As different implementations may have different width for int or long etc., since c99 the header `<stdint.h>` provides several fixed-width signed integer type definitions.

The width is explicit with their data type names.

```
/* commonly used types include */
```

```
uint32_t u32 = 32; /* exactly 32-bits wide */
```

```
uint8_t  u8 = 255; /* exactly 8-bits wide */
```

```
int64_t  i64 = -65 /* exactly 64 bit in two's complement  
                    representation */
```

# Floating-point Numbers

There are just signed floating point data type. For example,

Type	Storage	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal
double	8 byte	2.3E-308 to 1.7E+308	15 decimal
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal



# Single-precision floating-point format

Now Let's take a look how C stores floating-point number with float type.



- radix is 2, 1 sign bit, 8 exponent bits and 24 for mantissa bits (one implicit bit and 23 fraction bits)

The real value assumed by a given 32-bit binary data is computed as:

$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

which gives :

$$(-1)^{\text{sign}} \times 2^{(e-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)_2$$

# Constants of Floats

Turn to Page 257 in which you will see:

FLT_RADIX	2	radix of exponent representation
FLT_ROUNDS	1	floating-point rounding mode for addition
FLT_DIG	6	decimal digits of precision
FLT_EPSILON	1E-05	smallest number $x$ such that $1.0 + x \neq 1.0$
FLT_MANT_DIG	24	number of base FLT_RADIX digits in mantissa
FLT_MAX	1E+37	maximum floating-point number
FLT_MAX_EXP	128	maximum $n$ such that $\text{FLT\_RADIX}^{n-1}$ is representable
FLT_MIN	1E-37	minimum normalized floating-point number
FLT_MIN_EXP	-125	minimum $n$ such that $10^n$ is a normalized number

# Constants of Floats

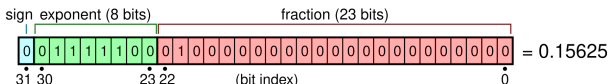
The `FLT_ROUNDS` is an integer which defines the rounding mode for floating point addition.

And these are the possible modes:

- 1 indeterminable
- 0 towards zero
- 1 to nearest
- 2 towards positive infinity
- 3 towards negative infinity

The other constants will be described along with introducing of the Single-precision Floating-Point Format.

# Single-precision floating-point format

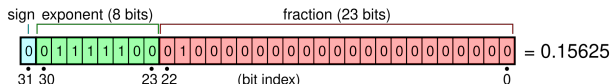


- The true significand includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1, **normal number** unless the exponent is stored with all zeros **subnormal number**.
- Sometimes the largest exponent float number is saved for representing infinity, based on implementation.

Ok, now give the binary format of these float numbers within single-precision floating-point format:

- 1, 1.0
- 2, 2.0
- 3, 2.5
- 4, 0.1
- 5, smallest positive subnormal number
- 6, largest subnormal number
- 7, smallest positive normal number
- 8, largest normal number

# Single-precision floating-point format



Suppose the largest exponent float number is saved for representing infinity.

1, 1.0      0 01111111 000000000000000000000000

2, 2.0      0 10000000 000000000000000000000000

3, 2.5      0 10000000 010000000000000000000000

4, 0.1

5, smallest positive subnormal number

0 00000000 000000000000000000000001

6: largest subnormal number

0 00000000 111111111111111111111111

7, smallest positive normal number

0 00000001 000000000000000000000000

8, largest normal number

0 11111110 111111111111111111111111

Floating point types (float, double and long double) cannot precisely represent some numbers because they have finite precision and represent the values in a binary format.

For example:

```
printf("0.1 is not 0.1 in computer:  %1.30f\n", 0.1f);
```

this will print out:

```
0.1 is not 0.1 in computer: 0.100000001490116119384765625000
```

This will cause problems that hard to notice. Let's take an example.

# Pitfall

What will this print out?

```
float n=0.1;
double a = 0.1;

if (n == 0.1)
    printf("it's all right\n");
else
    printf("Wierd\n");

if (a + a + a + a + a + a + a + a + a + a == 1.0)
    printf("10 * 0.1 is indeed 1.0. \n");
else
    printf("Wierd, This is not guaranteed in the general c
```

- Therefore when compare floating number, we should use a epsilon value.
- Recall that `FLT_DIG` is the decimal digits of precision which we can use to determine the epsilon value that we use.
- Let's see a demo.
- Also turn to page 251, there are two math function we need to use. They are `pow()` and `fabs()`.



# Pitfall Fix

What will this print out?

```
float epsilon_f = 1.0/pow(10.0,(double) FLT_DIG);
double epsilon_d = 1.0/pow(10.0,(double) DBL_DIG);

float n=0.1;    double a = 0.1;

if ( fabs(n -0.1) < epsilon_f)
    printf("it's all right\n");
else    printf("Wierd\n");

if (fabs(a + a + a + a + a + a + a + a + a + a -1.0)
    < epsilon_d)
    printf("10 * 0.1 is indeed 1.0. \n");
else
    printf("Wierd, This is not guaranteed in the general
           case.\n");
```

# Constants

- Constants or literals are fixed value in a C 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 positive.
- 3, for example,

signed char	memory
1	0000,0001
-1	1000,0001

# Illegal Constant Number Expressions

For example:           078

What's wrong with this expression?

# Illegal Constant Number Expressions

For example:            078

Decimal:                {0,1,2,...,9}

Octal    :                {0,1,2,...,7}

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

# Floating-point Constants

decimal form

3.1415926,

3.1415926f,

exponential form

31425927E-5 or 31425927e-5 (denotes double)

31425927E-5f or 31425927e-5f (denotes float)

0, The e/E denoted radix of 10.

- 1, Floating number literals without suffix "f" will be treated as double.
- 2, For the decimal floating number, we have to have the dot part with other parts.
- 3, For the exponential form, we must include the integer part, the e/E and the exponent part.

# Floating-point Constants

324, 314E are illegal floating numbers.

Try them, see what will happen.

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

.1 and 1. are legal



# Character Constants

- Character constants are enclosed in single quotes, e.g., 'x'.
- There are plain character like 'x', and also escape character '\n'
- 0 and '0' are different. '0' represents the ASCII integer value of '0'.
- Page 37. Character literal can also be specified by octal or hexadecimal digits within this form:

```
#define VTAB '\013'  
#define BELL '\007'  
#define VTAB '\xb'  
#define BELL '\x7'
```

- '0' and "0" are also different things in C.

# Character Constants

Here is a list of escape character:

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

# String Constants

- String constants are enclosed in double quotes, like "hello world".
- And they are just a sequence of character constants
- String constants can be concatenated at compile time:

`"hello," " world"`

is equivalent to

`"hello, world".`

- The internal representation of a string has a null character `'\0'` at the end, so the physical storage required is one more than the number of characters written between the quotes
- We will cover string more later.

- On page 154.