# Notes for C and C++ programming

Tips, hints, and tricks for developers in programming C/C++.

#notes #c #c++

Last update: 2021-08-10 16:50:49

Size of a datatype

Do not use sizeof for array parameters

Use goto if it can reduce the complexity

Array[index] is actually pointer accessing

Short-circuit evaluation

Cached data

Compiler warnings

Buffered stdout but non-buffered stderr

Negative error code

Bit fields

# Size of a datatype

The size of a type is determined by the compiler, which doesn't have anything to do with the actual hardware. The returned value of sizeof(char) is always 1 by definition, and sizeof(int) always returns 4. Starting at C99, bool is present as an 1-byte datatype.

Note that, size of a pointer should be 4 bytes on any 32-bit C/C++ compiler, and be 8 bytes on any 64-bit C/C++ compiler.

Use int instead of char or uint8\_t? It depends on the target machine: if it is incapable to access unaligned memory (e.g. Cortex-M0 processors), then using int is much faster.

# Do not use sizeof for array parameters

The function fun() below receives an array parameter arr[] and tries to find out the number of elements in the array arr[] using the sizeof operator. In main, there is also a statement calculating the number of elements in the array arr[]. But 2 methods return different results.

```
int fun(int arr[]) {
    return sizeof(arr)/sizeof(arr[0]); // WRONG
void main() {
    int arr[4] = \{0, 0, 0, 0\};
    int arr_size = sizeof(arr)/sizeof(arr[0]); // RIGHT
    if (arr_size == fun(arr)) {} // ???
}
```

In C, array parameters are treated as pointers. So the expression:

```
sizeof(arr)/sizeof(arr[0])
```

becomes

```
sizeof(int *)/sizeof(int)
```

which finally is evaluated as sizeof(arr) = sizeof(int \*) which is 4 or 8 depending on the compiler.

# Use goto if it can reduce the complexity

Using goto is considered as bad and harmful, but if it is used with care, it can be helpful to reduce the complexity.

## For example:

```
void func(...) {
    byte* buf1=malloc(...);
    byte* buf2=malloc(...);
    FILE* f=fopen(...);
    if (f==NULL)
        goto func_cleanup_and_exit;
    if (something_goes_wrong_1)
        goto func_close_file_cleanup_and_exit;
func_close_file_cleanup_and_exit:
    fclose(f);
func_cleanup_and_exit:
    free(buf1);
    free(buf2);
func_exit:
    return;
};
```

# Array[index] is actually pointer accessing

The expression a[i] is translated to \*(a+i). That's why this weird code printf("%c", 3["abcdef"]); still runs.

C/C++ does not check the index range, therefore it is easy to access a memory block which is out of range of the array:

```
int a[] = {1, 2};
int b[] = {3, 4, 5};
// if a and b are allocated next to each other, b[-1] will be a[1]
printf("%d", b[-1]);
```

There are *Buffer overflow* exploiting techniques based on this problem of accessing memory block outsides of the array.

#### A Short-circuit evaluation

At runtime, in the expression with AND operator if(a && b && c), the part (b) will be calculated only if (a) is true, and (c) will be calculated only if (a) and (b) are both true.

The same story about OR operator: in the expression if  $(a \mid \mid b \mid \mid c)$ , if the sub-expression (a) is true, others will not be computed.

This helps in case the next condition depends on the previous condition, such as accessing to pointer:

```
if (pointer != NULL && pointer->member == 3) {}
```

or, checking for a higher priority condition first:

```
if (flag_a || flag_b) {}
```

However, as the consequence, do not expect the  $2^{nd}$  or later conditions are executed in all cases.

# Cached data

Modern CPUs have L1, L2, and L3 caches. When fetching data from memory, it usually read a whole line of L1 cache. For example, L1 cache has 64-bytes lines, it will fetch 64 bytes from memory at once when it accesses to a memory.

So if a data structure is larger than 64 bytes, it is very important to divide it by 2 parts: the most demanded fields and the less ones. It is desirable to place the most demanded fields in the first 64 bytes. C++ classes are also concerned.

# **6** Compiler warnings

Is it worth to turn on **-Wall** to see all possible problems which usually are small error and hard to get noticed. In GCC, it is also possible to turn all warnings to errors with **-Werror**. If enabled, any problem marked as error will halt the compiler, and it must be fixed.

#### Example 1:

```
int f1(int a, int b, int c) {
   int ret = a + b + c;
   printf("%d", ret);
}
int main() {
   printf("%d", f1(1,2,3));
}
```

The main() function still runs but prints out wrong value due to non-returned function. A random value will be seen because compiler still use the return register which is not set the desired value.

#### Example 2:

```
bool f1() {
    return condition ? true : false;
}
```

# bool is 1-byte datatype

Compiler will generate a code to set the low byte of the AL register to 0x01 or 0x00, because the return size of 1 byte. Other higher bytes of the AL register won't change.

However, in a different file, f1() is assumed to return *int*, not *bool*, therefore compiler generates code to compare 4 bytes of an *int* which is only updated its lowest byte after f1() returns. There maybe some random value in higher bytes of that *int* memory block, and it will cause a wrong comparison:

```
void main() {
   if(f1())
}
```

## Buffered stdout but non-buffered stderr

The stdout or cout is a buffered output, so a user, usually not aware of this, sees the output by portions. Sometimes, the program outputs something and then crashes, and if buffer did not have time to *flush* into the console, a user will not see anything. This is sometimes inconvenient.

Thus, for dumping more important information, including debugging, it is better to use the un-buffered stderr or cerr.

#### Megative error code

The simplest way to indicate to caller about its children's success is to return a boolean value: *false* — in case of error, and *true* in case of success. However, to indicate more status than just 2 states, a number can be returned. And it can be extended more to indicate different status of failure or success using signed numbers:

```
/**
* func() - A function that does something
* input:
*    none
* output:
*    -2: error on buffer
*    -1: error on transmission
*    0: success but no response
*    1: success with a response = NACK
*    2: success with a response = ACK
*//
```

# **b** Bit fields

A very popular thing in C, and also in programming generally is working with bits. For flags specifying, in order to not make a typo and mess, they can be defined using bit shifting:

```
#define FLAG1 (1<<0)
#define FLAG2 (1<<1)
#define FLAG3 (1<<2)
#define FLAG4 (1<<3)
#define FLAG5 (1<<4)
```

and some macros can be use to work on bits:

```
#define IS_SET(flag, bit) (((flag) & (bit)) ? true : false)
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))
```

For better performance, use *bool* for each bit, but it will cost memory.