Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 11
21.11.2018

Bitwise operators

Bitwise operators

- C allows to perform operations on the bit representation of a variable
- Not to be confused with logical operators (&&, ||, !)

Operator symbol	Operation
&	and
|	or
^	xor
~	One complement (not)
<<	Left shift
>>	Right shift

& (Bitwise AND)

- Takes two numbers as operand and does AND on every bit of two numbers
- The result of AND is 1 only if both bits are 1
- AND operation:

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a & b = %d\n", a & b);// prints 1
```

| (Bitwise OR)

- Takes two numbers as operand and does OR on every bit of two numbers
- The result of OR is 1 if any of the two bits is 1
- OR operation:

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a | b = %d\n", a | b);// prints 13
```

^ (Bitwise XOR)

- Takes two numbers as operand and does XOR on every bit of two numbers
- The result of XOR is 1 if the two bits are different
- XOR operation:

XOR Truth Table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

```
// a = 4(00000101), b = 9(00001001)
unsigned char a = 5, b = 9;
printf("a ^ b = %d\n", a ^ b); // prints 12
```

~ (Bitwise NOT)

- Takes **one** number and inverts all the bits
- NOT operation:

```
// a = 4(00000101)
unsigned char a = 5;
printf("~a = %d\n", ~a); // prints 250
```

<< (Bitwise left shift)

- Takes two numbers, left shifts the bits of first operand
- The second operand decides the number of places to shift
- left shift operation:

```
[variable] << [number of places]
```

```
// b = 9(00001001)
unsigned char b = 9;
printf("b << 1 = %d\n", b << 1);   // prints 18

printf("b << 2 = %d\n", b << 2);   // prints 36
```

What does >> (*shift right)* computes?

<<, >> (Bitwise left and right shift)

- **The left shift and right shift operators should not be used for negative numbers** !!!
- The result of << and >> is undefined behaviour if any of the operands is a negative number
- For example results of both -1 << 1 and 1 << -1 is undefined
- Also, if the number is shifted more than the size of integer, the behaviour is undefined
- For example, 1 << 33 is undefined if integers are stored using 32 bits

How to check if the number is odd or even?

- The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

```
int main()
{
    int x = 19;
    (x & 1)? printf("Odd"): printf("Even");
    return 0;
}
```

Bitwise operations to manipulate single bits

- Each char can represent 8 independent boolean variables
- For this we need to find a way to
  - Find the value of a single bit
  - Change the value of a single bit

Finding the value of a single bit

We can do that with the **<<** and **&** operators:

```
int isZero (int variable, int position)
{
    return (variable & (1 << position)) == 0;
}
```

Example:

- We have a variable with the bit-pattern: **00011001**
- We want to test the value of the 3rd bit, from the least-significant bit, (starting from place 0, this is place 2):

Construct a mask: `00000001 << 2 == 00000100`

Apply: `00011001 & 00000100 == 00000000 == 0`

Toggling a single bit

Use the << and ^ operators:

```
int toggleBit (int variable, int position)
{
    return (variable ^ (1 << position));
}
```

XOR Truth Table

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Example:

- We have a variable with the bit-pattern: **00011001**
- We want to change the value of the 3rd bit (from the least-significant bit, starting from place 0, this is place 2):

Construct a mask: `00000001 << 2 == 00000100`

Apply: `00011001 ^ 00000100 == 00011101`

Using masks

```
flag1 = 00000001
flag2 = 00000010
flag3 = 00000100

mask = flag1 | flag2 | flag3

mask == 00000111
```

Using masks – low level file open

```
int open(const char *path, int oflag, ... );
```

- `oflags` is an options mask – how to open the file
- You can use several options together using the bitwise OR operator
- Read more at: http://linux.die.net/man/3/open

E.g.: `fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC)`

Open for write only

Create if not already exist

Clear file if already exist

Variadic functions

Variadic functions

A variadic function is a function with **variable number of arguments**

Example:

?

Variadic functions

A variadic function is a function with **variable number of arguments**

Example:

```
// 1 argument
printf(“Hello world\n”);
// 2 arguments
printf(“The value of i is: %d\n”, i);
// 3 arguments
printf(“The value of i is: %d\t its address
       is: %p\n”, i, &i);
```

Variadic function definition

```
<return type> <function name>(<first parameter>,...)
```

Examples:

```
//std lib. function
int printf(const char *format, ...)

//function we define
int countIntegers(int integerNum, ...)
```

The main challenge

- Since we do not know the function parameters in advance, we can't give them names

- We have to develop a technique to access the variables based on their ***type and place*** relative to the first parameter

stdarg.h

To define variadic functions we should include the header **stdarg.h**

- definition of the type: `va_list`
  - A type for iterating the arguments (most likely `void*`)

- definition of the **macros**:
  - `va_start` – start iterating the argument list with `va_list` and the last named parameter of the function
  - `va_arg` – retrieve the current argument
  - `va_end` – free the `va_list`

Example

```
#include <stdarg.h>
int sumInts(int argsNum, ...)
{
    va_list ap;
    int i, sum=0;
    va_start(ap, argsNum); // now we point to the place
                           // right after the first argument
    for (i = 0; i < argsNum; ++i) {
       sum += va_arg(ap, int);// access current argument and
                              // move ap to the next argument
    }
    va_end(ap); // free ap
    return sum;
}
//in main:
sumInts(5,1,1,2,3,4) //returns 11
sumInts(2,7,1) //returns 8
```

Note!

- Variadic functions must have at least one named parameter

```
void wrong(...);
```

- There is no mechanism defined for determining the number or types of the unnamed arguments – we can use one of the following ways:
  - **format string** (like in `printf`)
  - **sentinel value** at the end of variadic arguments
  - **count argument** indicating the number of the variadic arguments

snprintf & vsnprintf

New in C99, **snprintf** and **vsnprintf** are nice examples of variadic functions

```
int snprintf (char *s, size_t n,
              const char * format, ...)

int vsnprintf(char *s, size_t n,
              const char * format, va_list arg)
```

Read example at:
http://www.cplusplus.com/reference/cstdio/vsnprintf/
("passing on" variable argument list)

snprintf example

```
#include <stdio.h>

int main () {
  char buffer [100];
  int cx;
  cx = snprintf ( buffer, 100,
                  "The half of %d is %d", 60, 60/2 );
  if (cx>=0 && cx<100) { // check returned value
       snprintf ( buffer+cx, 100-cx,
                  ", and the half of that is %d.", 60/2/2 );
  }
  puts (buffer);
  return 0;
}
```

Optimization

What smart people say about it...

“Premature optimization is the root of all evil (or at least most of it) in programming”

– Donald Knuth

So, what to do?

- ✓ Check if you **need to optimize**
- ✓ Profile - check **where to optimize**
  (gprof if you use gcc; need to add –g and –pg to the compilation line)
- ✓ Remember to **“turn off” debugging**
  (gcc -DNDEBUG)
- ✓ Check what your compiler can do for you on your **specific hardware**
  (-O3, -mcpu=arm7, etc…)

- ➢ We'll learn more about optimization in **labcpp**

Cache-friendly coding

- Fast memory is expensive, so we have very little of it
- Goal: achieve good performance with what we have
- Method: memory is organized in a hierarchic order:
  - Registers
  - Cache (several levels)
  - **RAM**
- Search from the fastest to the slowest memory
- Fetch important memory before it is used!
  How? When a given location is accessed, most likely its neighborhood will be accessed in the near future

Cache-friendly coding: iterating 2D array

C requires “row-major ordering”

```
int i,j;
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<ROWS_NUM;++i){
  for(j=0;j<COLS_NUM;++j){
    arr[i][j] = i*j;
  }
}
```

```
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<COLS_NUM;++i){
  for(j=0;j<ROWS_NUM;++j){
    arr[j][i] = i*j;
  }
}
```

Cache-friendly coding: iterating 2D array

C requires “row-major ordering”

```
//efficient
int i,j;
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<ROWS_NUM;++i){
  for(j=0;j<COLS_NUM;++j){
    arr[i][j] = i*j;
  }
}
//not efficient
int arr[ROWS_NUM][COLS_NUM];
for(i=0;i<COLS_NUM;++i){
  for(j=0;j<ROWS_NUM;++j){
    arr[j][i] = i*j;
  }
}
```

`inline` functions

New in C99

inline functions

- Tell the compiler to substitute calls for the function body by inline expansion - meaning, by inserting the function code
- This helps save the overhead of function invocation and return (placing data on stack and retrieving the result)
- However, this may result in a larger executable as the code for the function has to be repeated multiple times

inline functions

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}

// somewhere in the code ...
a = max(x, y);

// may actually be compiled as ...
a = (x > y) ? x : y;
```

inline functions

- The `inline` specifier is only a **hint** for the compiler to perform optimizations. Compilers can (and usually do) ignore the presence or absence of the `inline` specifier for the purpose of optimization
- **Inline function must be defined in the same translation unit (so we usually just define the function in the header)**
- If an external definition also exists in the program, it's unspecified whether the inline definition (if present in the translation unit) or the external definition is called
- Read more at home… http://en.cppreference.com/w/c/language/inline

inline functions vs macros

- ✓ Macro invocations do not perform type checking
- ✓ A macro cannot use the return keyword with the same meaning as a function would do (it would make the function that asked the expansion terminate, rather than the macro).
  In other words, a macro cannot return anything which isn't the result of the last expression invoked inside it
- ✓ C macros use mere textual substitution, which may result in unintended side-effects and inefficiency due to re-evaluation of arguments and order of operations

http://en.wikipedia.org/wiki/Inline_function

inline functions vs macros

- ✓ Compiler errors within macros are often difficult to understand, because they refer to the expanded code, rather than the code the programmer typed
- ✓ Many constructs are awkward or impossible to express using macros, or use a significantly different syntax. Inline functions use the same syntax as ordinary functions, and can be inlined and un-inlined at will with ease
- ✓ Many compilers can also inline expand some recursive functions (up to some depth); recursive macros are typically illegal

http://en.wikipedia.org/wiki/Inline_function

inline functions vs macros

Bjarne Stroustrup, the designer of C++, likes to emphasize that macros should be avoided wherever possible, and advocates extensive use of inline functions

http://en.wikipedia.org/wiki/Inline_function