CH-230-A

Programming in C and C++

C/C++

Tutorial 6

Dr. Kinga Lipskoch

Fall 2019

The C Preprocessor: Macro Substitution (5)

- Defined names can be undefined using
  - `#undef NAME`
- Formal parameters are not replaced within quoted strings
- If parameter name is preceded by `#` in replacement text, the actual argument will be put inside quotes
  - `#define DPRINT(expr) printf(#expr " = %g\n", expr)`
  - `DPRINT(x/y)` will be expanded to
  - `printf("x/y" " = %g\n", x/y);`

The C Preprocessor: Conditional Inclusion (1)

- Preprocessing can be controlled by using conditional statements which will be evaluated while preprocessor runs
- Enables programmer to selectively include code, depending on conditions
- `#if`, `#endif`, `#elif` (i.e., else if), `#else`

```
#if defined(DEBUG)  // short: #ifdef DEBUG
  printf("x: %d\n", x);
#endif
```

The C Preprocessor: Conditional Inclusion (2)

- `#ifdef`, `#ifndef` are special constructs that test whether name is (not) defined
- `gcc` allows to define names using the `-D` switch
- Ex: `gcc -DDEBUG -c program.c`
- Previous line is equivalent to
  `#define DEBUG`

The C Preprocessor: Conditional Inclusion (3)

- Write a small program in which you illustrate the use of conditional inclusion for debugging purposes
- Ex: If the name `DEBUG` is defined then print on the screen the message "This is a test version of the program"
- If `DEBUG` is not defined then print on the screen the message "This is the production version of the program"
- Also experiment with `gcc -D`

The Structure of a Header File with Conditional Inclusion

```c
/* Student.h */
#ifndef _LIST_H
#define _LIST_H
struct list {
  int info;
  struct list *next;
};
void printList(struct list *);
struct list * push_front(struct list *, int);
...
#endif // this matches the initial #ifndef
```

Bit Operations

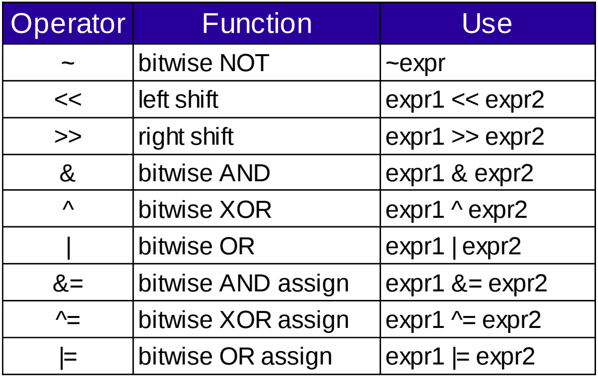
- The bit is the smallest unit of information
  - Represented by `0` or `1`
- Eight bits form one byte
  - Which data type could be used for representation?
- Low-level coding like writing device drivers or graphic programming require bit operations
- Data representation
  - Octal (format `%o`), hexadecimal (format `%x`, representation prefix `0x`)
- In C you can manipulate individual bits within a variable

Bitwise Operators (1)

Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal	128	64	32	16	8	4	2	1
Binary number	0	1	0	1	1	1	0	1

- Allow you to store and manipulate multiple states in one variable
- Allows to set and test individual bits in a variable

Bitwise Operators (2)



Operator	Function	Use
~	bitwise NOT	~expr
<<	left shift	expr1 << expr2
>>	right shift	expr1 >> expr2
&	bitwise AND	expr1 & expr2
^	bitwise XOR	expr1 ^ expr2
|	bitwise OR	expr1 | expr2
&=	bitwise AND assign	expr1 &= expr2
^=	bitwise XOR assign	expr1 ^= expr2
|=	bitwise OR assign	expr1 |= expr2

Bitwise and Logical AND

```c
#include <stdio.h>
int main()
{
  int i1, i2;
  i1 = 6; // set to 4 and suddenly check 3 fails
  i2 = 2;
  if ((i1 != 0) && (i2 != 0))
    printf("1: Both are not zero!\n");
  if (i1 && i2)
    printf("2: Both are not zero!\n");
  // wrong check
  if (i1 & i2)
    printf("3: Both are not zero!\n");
  return 0;
}
```

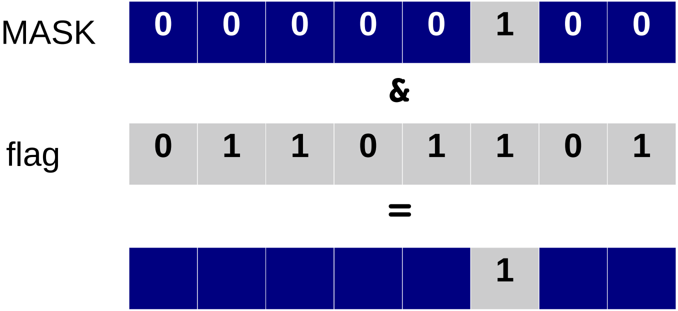
The Left-Shift Operator

- Moves the data to the left a specified number of bits
- Shifted out bits disappear
- New bits coming from the right are `0`'s
- Ex: `10101101 << 3` results in `01101000`

The Right-Shift Operator

- Moves the data to the right a specified number of bits
- Shifted out bits disappear
- New bits coming from the right are:
  - 0's if variable is unsigned
  - Value of the sign bit if variable is signed
- Ex:
  - ` 7 = 00000111 >> 2 results in 00000001`
  - `-7 = 11111001 >> 2 results in 11111110`

Using Masks to Identify Bits



MASK	0	0	0	0	0	1	0	0
				&				
flag	0	1	1	0	1	1	0	1
				=				
						1		

Using Masks

- Bitwise AND often used with a mask
- A mask is a bit pattern with one (or possibly more) bit(s) set
- Think of `0`'s as opaque and the `1`'s being transparent, only the mask `1`'s are visible
- If `result > 0` then at least one bit of mask is set
- If `result == MASK` then the bits of the mask are set

binary.c

```c
#include <stdio.h>
char str[sizeof(int) * 8 + 1];
const int maxbit = sizeof(int) * 8 - 1;
char* itobin(int n, char* binstr) {
  int i;
  for (i = 0; i <= maxbit; i++) {
    if (n & 1 << i) {
      binstr[maxbit - i] = ’1’;
    }
    else {
      binstr[maxbit - i] = ’0’;
    }
  }
  binstr[maxbit + 1] = ’\0’;
  return binstr;
}
int main()
{
  int n;
  while (1) {
    scanf("%i", &n);
    if (n < 0) break;
    printf("%6d: %s\n", n, itobin(n, str));
  }
  return 0;
}
```

How to Turn on a Particular Bit

- To turn on bit `1` (second bit from the right), why does `flags += 2` not work?
  - If `flags = 2 = 000000010`$_{(2)}$
  - Then `flags +=2` will result in
  - `flags = 4 = 00000100`$_{(2)}$ which "unsets" bit `1`
- Correct usage:
  - `flags = flags | 2` is equivalent to
  - `flags |= 2` and turns on bit `1`

How to Toggle a Particular Bit

- To toggle bit `1`
  - `flags = flags ^ 2;`
  - `flags ^= 2;` toggles on bit `1`
- General form
  - `flags ^= MASK;`

How to Test a Particular Bit

- To test bit `1`, why does `flags == 2` not work?
- Testing whether any bit of `MASK` are set:
  - `if (flags & MASK) ...`
- Testing whether all bits of `MASK` are set:
  - `if ((flags & MASK) == MASK) ...`

Using Bits Operations: A Problem

- Think of a low-level communication program
- Characters are stored in some buffer
- Each character has a set of status flags

`ERROR`	true if any error is set
`FRAMING_ERROR`	framing error occurred
`PARITY_ERROR`	wrong parity
`CARRIER_LOST`	carrier signal went down
`CHANNEL_DOWN`	power was lost on device

Size Considerations

- Suppose each status is stored in additional byte
  - 8k buffer (real data)
  - But 40k status flags (admin data)
- Need to pack data

A Communication System

- `0 - ERROR`
- `1 - FRAMING_ERROR`
- `2 - PARITY_ERROR`
- `3 - CARRIER_LOST`
- `4 - CHANNEL_DOWN`

How to Initialize Bits

- `const int ERROR = 0x01;`
- `const int FRAMING_ERROR = 0x02;`
- `const int PARITY_ERROR = 0x04;`
- `const int CARRIER_LOST = 0x08;`

- If more states needed: `0x10`, `0x20`, `0x40`, `0x80`
- It is not intuitive to know which hexadecimal-value has which bit set

How to ”Nicely” Set Bits

- `const int ERROR = (1 << 0);`
- `const int FRAMING_ERROR = (1 << 1);`
- `const int PARITY_ERROR = (1 << 2);`
- `const int CARRIER_LOST = (1 << 3);`
- `const int CHANNEL_DOWN = (1 << 4);`

Everyone will directly understand encoding of the bits, additional documentation can be greatly reduced