C - Basics, Bitwise Operator

Shuai Mu

Based on Tiger Wang and Jinyang Li’ slides
Python programmers
C programmers
C is an old programming language

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>1972</td>
<td>1995</td>
<td>2000 (2.0)</td>
</tr>
<tr>
<td>Style</td>
<td>Procedure</td>
<td>Object oriented</td>
<td>Procedure &amp; object oriented</td>
</tr>
<tr>
<td></td>
<td>Compiled to machine code, runs on bare machine</td>
<td>Compiled to bytecode, runs by another piece of software</td>
<td>Scripting language, interpreted by software</td>
</tr>
<tr>
<td>Type</td>
<td>static type</td>
<td>static type</td>
<td>dynamic type</td>
</tr>
<tr>
<td></td>
<td>Manual memory management</td>
<td>Automatic memory management with GC</td>
<td></td>
</tr>
</tbody>
</table>
Why learn C for CSO?

• C is a systems language
  – Language for writing OS and low-level infrastructure code
  – Systems written in C:
    • Linux, Windows kernel, MacOS kernel
    • MySQL, Postgres
    • Apache webserver, NGIX
    • Java virtual machine, Python interpreter

• Why learning C for CSO?
  – simple, low-level, “close to the hardware”
"Hello World"

```c
#include <stdio.h>

int main() {
    printf("hello, world\n");
    return 0;
}
```
```
#include <stdio.h>  // Header file

int main()
{
    printf("hello, world\n");
    return 0;
}
```
Compiling

Source Codes
[*.c, *.h]

C Program Preprocessor

gcc -E *.c

Source Codes
[*.i]

gcc helloworld.c -o helloworld
Compiling

Source Codes

\([*.c, *.h]\)

C Program Preprocessor

\(gcc -E \ast.c\)

Source Codes

\([*.i]\)

C Program Compiler

\(gcc -S \ast.i\)

gcc helloworld.c –o helloworld

Assembly Codes

\([*.s]\)
Compiling

Source Codes
[*.c, *.h]

C Program Preprocessor

gcc -E *.c

Source Codes
[*.i]

C Program Compiler

gcc -S *.i

Assembly Codes
[*.s]

gcc -c *.s

Binary Codes
[*.o]

Assembler

gcc helloworld.c -o helloworld
Compiling

Source Codes
[*c, *.h]

gcc -E *.c

C Program Preprocessor

Source Codes
[*.i]

gcc -S *.i

C Program Compiler

Assembly Codes
[*.s]

gcc -c *.s

Assembly

Executable file
[a.out, helloworld]

gcc *.o

Linker

Binary codes
[*.o]

gcc helloworld.c -o helloworld

Assembler
Three basic elements

Variables
- The basic data objects manipulated in a program

Operator
- What is to be done to them

Expressions
- Combine the variables and constants to produce new values
Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>1</td>
</tr>
</tbody>
</table>

Declaration: `int a = 1;`
Variables

Declaration:    int a;

Value assignment:  a = 0;

If not initialized, a can have any value
## Primitive Types

64 bits machine

<table>
<thead>
<tr>
<th>type</th>
<th>size (bytes)</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) char</td>
<td>1</td>
<td>char c = ‘a’</td>
</tr>
<tr>
<td>(unsigned) short</td>
<td>2</td>
<td>short s = 12</td>
</tr>
<tr>
<td>(unsigned) int</td>
<td>4</td>
<td>int i = 1</td>
</tr>
<tr>
<td>(unsigned) long</td>
<td>8</td>
<td>long l = 1</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>float f = 1.0</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>double d = 1.0</td>
</tr>
<tr>
<td>pointer</td>
<td>8</td>
<td>int *x = &amp;i</td>
</tr>
</tbody>
</table>

Old C has no native boolean type. A non-zero integer represents true, a zero integer represents false.

C99 has “bool” type, but one needs to include `<stdbool.h>`
Implicit conversion

```c
int main()
{
    int a = 0;
    unsigned int b = 1;

    if (a < b) {
        printf("%d is smaller than %d\n", a, b);
    } else if (a > b) {
        printf("%d is larger than %d\n", a, b);
    }

    return 0;
}
```

Compiler converts types to the one with the largest data type (e.g. char → unsigned char → int → unsigned int)
Implicit conversion

```c
int main()
{
    int a = -1;
    unsigned int b = 1;

    if (a < b) {
        printf("%d is smaller than %d\n", a, b);
    } else if (a > b) {
        printf("%d is larger than %d\n", a, b);
    }

    return 0;
}

-1 is promoted to unsigned int and thus appears to be a large positive number. (4294967295)_{10}
```
Explicit conversion (casting)

```c
int main() {
    int a = -1;
    unsigned int b = 1;

    if (a < (int) b) {
        printf("%d is smaller than %d\n", a, b);
    } else if (a > (int) b) {
        printf("%d is larger than %d\n", a, b);
    }

    return 0;
}
```

(type-name) expression
Operators

Arithmetic +, -, *, /, %, ++, --
Relational ==, !=, >, <, >=, <=
Logical &&, ||, !
Bitwise &, |, ^, ~, >>, <<

Arithmetic, Relational and Logical operators are identical to java’s
Bitwise operator &

And (&)
- given two bits \( x \) and \( y \), \( x \& y = 1 \) when both \( x = 1 \) and \( y = 1 \)

\[
\begin{array}{c|ccc}
& x & 0 & 1 \\
\hline
y & \& & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
\end{array}
\]

\( \binom{0 1 1 0 1 0 0 1}{2} \)
\& \( \binom{0 1 0 1 0 1 0 1}{2} \)

\( = \binom{0 1 1 0 0 0 1 1}{2} \)
Bitwise operator &

And (&)
- given two bits $x$ and $y$, $x \& y = 1$ when both $x = 1$ and $y = 1$

<table>
<thead>
<tr>
<th>$y$</th>
<th>$x$</th>
<th>$(01101001)_2$ &amp; $(01010101)_2$</th>
<th>$(01000001)_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$(01000001)_2$
Bitwise operator &

And (&)

- given two bits \( x \) and \( y \), \( x \& y = 1 \) when both \( x = 1 \) and \( y = 1 \)
- \& is often used to mask off some set of bits

\[
\begin{array}{c|cc}
\text{y} & 0 & 1 \\
\hline
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|cccccccc}
\text{x} & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
\hline
\& & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\[
(0000011111)_2 \quad \& \quad (01101001)_2 = (0000010001)_2
\]
Bitwise operator \(|\)

- given two bits \(x\) and \(y\), \(x \mid y = 1\) when either \(x = 1\) or \(y = 1\)

\[
\begin{array}{c|cc}
  x & 0 & 1 \\
  \hline
  0 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|cccccccc}
  (01101001)_2 & (01010101)_2 \\
  \hline
  0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
  0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]
Bitwise operator $|$ 

Or $(|)$
- given two bits $x$ and $y$, $x | y = 1$ when either $x = 1$ or $y = 1$
Bitwise operator $|$

Or ($|$)
- given two bits $x$ and $y$, $x | y = 1$ when either $x = 1$ or $y = 1$
- $|$ is often used to turn some bits on

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|cc}
\text{y} & \text{x} & \text{y} | \text{x} \\
0 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
(01101001)_{2} & (0101010101)_{2} \\
\hline
(011111101)_{2}
\end{array}
\]
Bitwise operator \( \sim \)

**Not \( (\sim) \)**
- given a bit \( x \), \( \sim x = 1 \) when \( x = 0 \)
- One’s complement

<table>
<thead>
<tr>
<th>( \sim )</th>
<th>( x )</th>
<th>( \sim (01101001)_{2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Bitwise operator ~

Not (~)
- given a bit $x$, $\sim x = 1$ when $x = 0$
- One’s complement

<table>
<thead>
<tr>
<th>$\sim$</th>
<th>$x$</th>
<th>$\sim (01101001)_{2}$</th>
<th>$(10010110)_{2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bitwise operator ^

Xor (^)
- given two bits x and y, \( x \oplus y = 1 \) when either \( x = 1 \) or \( y = 1 \), but not both

\[
\begin{array}{c|cc}
\wedge & 0 & 1 \\
\hline
y & 0 & 0 & 1 \\
1 & 1 & 0
\end{array}
\]

\[
(01101001) \oplus (01010101) = 01010101
\]
**Bitwise operator ^**

**Xor (^)**

- given two bits $x$ and $y$, $x \ ^\ ^\ ^\ ^\ y = 1$ when either $x = 1$ or $y = 1$, but not both

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$(01101001)_2 \ ^\ ^\ ^\ ^\ (01010101)_2 = (00111100)_2$
Bitwise operator \texttt{<<}

Left shift (\texttt{“<<”})

- \( x \ll y \), shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

\[
\begin{array}{c}
\begin{align*}
x & : 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
x \ll 3 & : & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1
\end{align*}
\end{array}
\]
Bitwise operator \( \ll \)

Left shift (\( \ll \))
- \( x \ll y \), shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

\[
\begin{array}{c}
0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]
Bitwise operator $\gg$

Right shift ("$\gg$")
- $x \gg y$, shift bit-vector $x$ right $y$ positions
  - Throw away extra bits on right
  - Fill with ??? on left
    - Logical shifting
    - Arithmetic shifting
Bitwise operator >>

Right shift (">>")
- \(x >> y\), shift bit-vector \(x\) right \(y\) positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0’s on left

\[
\begin{array}{c|c}
\text{x} & 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\
\text{Logical} \ x >> 3 & 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\
\end{array}
\]
Bitwise operator $\gg$

Right shift ("$\gg$")
- $x \gg y$, shift bit-vector $x$ right $y$ positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0’s on left
- Arithmetic shift
  - Replicate most significant bit on the left

$$
\begin{align*}
\text{x} & \quad 10101001 \\
\text{Logical} \quad x \gg 3 & \quad 00010101 \\
\text{Arithmetic} \quad x \gg 3 & \quad 11110101
\end{align*}
$$
Bitwise operator $\gg$

Right shift ("$\gg$")
- $x \gg y$, shift bit-vector $x$ right $y$ positions
  - Throw away extra bits on right
- Logical shift (shr)
  - Fill with 0’s on left
- Arithmetic shift (sar)
  - Replicate most significant bit on the left

\[
\begin{array}{c|c}
\text{Logical} & x \gg 3 \\
\hline
x & 10101001 \\
xx & 00010101 \\
Arithmetic & x \gg 3 \\
\hline
& 111110101
\end{array}
\]
Which operation is used in C?

Arithmetic shifting on signed number, logical shifting on unsigned number

```c
#include <stdio.h>
int main()
{
    int a = 1;
    unsigned int b = 1;
    printf("%d %d\n", a>>10, b>>10);
}
```
Logical shift on signed number

```c
int lsr(int x, int n)
{
    ???
}
```
Logical shift on signed number

Observation

– It do the logical shift on unsigned number

Solution

– Convert the signed type into unsigned
Logical shift on signed number

```c
int lsr(int x, int n)
{
    return (int)((unsigned int)x >> n);
}
```
Control flow

\[
\text{int } a = b + 1
\]

expression
Combine the variables and constants to produce new values

```java
int a = b + 1

int c = (d << 1) + 2

float f = (float) c
```
Control flow

```c
int a = b + 1;
```

*statement*
Control flow

```java
{  
    int a = b + 1;
    int c = a * 2;
}
```

block
Control flow

if (expression) {
    int a = b + 1;
    int c = a * 2;
}

control statement
Control flow

if (expression)
    statement₁
else
    statement₂
Control flow

if (expression)  
  statement_1  
else  
  statement_2

if (expression_1)  
  statement_1  
else if (expression_2)  
  statement_2  
else  
  statement_3
Control flow

```plaintext
switch (expression) {
    case const-expr₁: statements₁
    case const-expr₂: statements₂
    default: statements₃
}
```
Control flow

while (expression) {
    statement
}

Control flow

```java
while (expression) {
    statement
}
```

```java
for(expr1; expr2; expr3) {
    statement
}
```
Control flow

\( expr_1; \)
\( \textbf{while}(expr_2) \) { 
  \hspace{1em} \text{statement} \\
  \hspace{1em} expr_3; 
} 

\( \textbf{for}(expr1; expr2; expr3) \) { 
  \hspace{1em} \text{statement} 
}
Control flow

Break
- cause the innermost enclosing loop or switch to be exited immediately

Continue
- cause the next iteration of the enclosing for, while, or do loop to begin.
Control flow

goto label

– Usable C provides the infinitely-abusable goto statement, and labels to branch to.
– Abandon processing in some deeply nested structure.

```c
for(...) {
    for(...) {
        for(...) {
            goto error
        }
    }
}

error:
    clean up the mess
```
Exercises

Given a number, write a function to decide if it is even?

```c
bool isEven(int n) {
}
```
Exercises

Given a number, write a function to decide if it is even?

```c
bool isEven(int n) {
    return (n & 1) == 0;
}
```
Exercises

Given a number, write a function to decide if it is even?

```cpp
bool isEven(int n) {
    return (n % 2) == 0;
}
```
Exercises

Given a number, write a function to decide if it is a power of two?

```c
bool isPowerOfTwo(int n) {
}
```
Exercises

Given a number, write a function to decide if it is a power of two?

```c
bool isPowerOfTwo(unsigned int n) {
    if (n==0) return false;
    while (n > 1) {
        if (n % 2) // (n%2)!=0
            return false;
        n = n / 2;
    }
    return true;
}
```
Exercises

Given a number, write a function to decide if it is a power of two?

```cpp
bool isPowerOfTwo(unsigned int n) {
    return (n & (n - 1)) == 0;
}
```
Given a number, write a function to decide if it is a power of two?

```cpp
bool isPowerOfTwo(unsigned int n) {
    return n != 0 && (n & (n - 1)) == 0;
}
```
Exercises

Count the number of ones in the binary representation of the given number (n > 0)?

```c
int count_one(int n) {
}
```
Exercises

Count the number of ones in the binary representation of the given number \( n > 0 \)

```c
int count_one(int n) {
    int count = 0;
    while (n != 0 ) {
        count += (n % 2);
        n = (unsigned int)n>>1;
    }
    return count;
}
```
Exercises

Count the number of ones in the binary representation of the given number?

```cpp
bool count_one(int n) {

}

A trick – clear the rightmost one: n & (n -1)
Exercises

Count the number of ones in the binary representation of the given number?

```cpp
bool count_one(int n) {
    while(n != 0) {
        n = n&(n-1);
        count++;
    }
    return count;
}
```