C - Basics, Bitwise Operator

Jinyang Li

Some are based on Tiger Wang’s slides

Python programmers

C programmers

C is an old programming language

C	Java	Python
1972	1995	2000 (2.0)
Procedure	Object oriented	Procedure & object oriented
Compiled to machine code, runs on bare machine	Compiled to bytecode, runs by another piece of software	Scripting language, interpreted by software
static type	static type	dynamic type
Manual memory management	Automatic memory management with GC	

Why learn C for CSO?

- C is a systems language
  - Language for writing OS and low-level 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”

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("hello, world\n");
6    return 0;
7  }
```

“Hello World”

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("hello, world\n");
6    return 0;
7  }
```

Header file

Standard Library

gcc helloworld.c –o helloworld

Three basic elements

Variables

- Basic data objects manipulated in a program

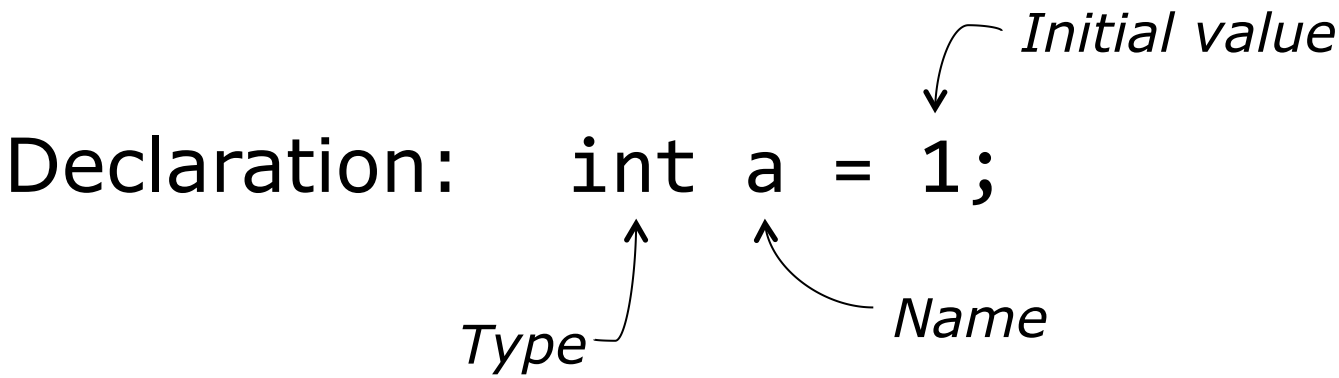
Operator

- What is to be done to them

Expressions

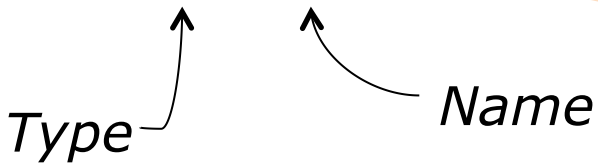
- Combine the variables and constants to produce new values

Variables



Variables

Declaration: `int a;`



If uninitialized, variable can have any value

Value assignment: `a = 0;`

Primitive Types

64 bits machine

type	size (bytes)	example
(unsigned) char	1	char c = 12 char c = 'a'
(unsigned) short	2	short s = 12
(unsigned) int	4	int i = 1
(unsigned) long	8	long l = 1
float	4	float f = 1.0
double	8	double d = 1.0
pointer	8	int *x = &i

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

```
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;
}
```

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

Implicit conversion

```
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 implicitly cast to unsigned int `0xffffffff`

Explicit conversion (casting)

```
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;
}
```

Operators

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

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

Bitwise operator &

bit-value of x & y

&	x = 0	x = 1
y = 0	0	0
y = 1	0	1

Result of 0x69 & 0x55

$$
\begin{array}{r}
(\,0\,1\,1\,0\,1\,0\,0\,1\,)_2 \\
\&\ (\,0\,1\,0\,1\,0\,1\,0\,1\,)_2 \\
\hline
(\,0\,1\,0\,0\,0\,0\,0\,1\,)_2
\end{array}
$$

Example use of &

- & is often used to mask off bits
  - Let b be an arbitrary bit, then b & 0 = 0
  - Let b be an arbitrary bit, then b & 1 = b

```
int clear_msb(int x) {

    return x & 0x7fffffff;
}
```

Bitwise operator |

bit-value of x | y

|	x = 0	x = 1
y = 0	0	1
y = 1	1	1

Result of 0x69 | 0x55

$$
\begin{array}{rr}
 & (\,0\,1\,1\,0\,1\,0\,0\,1\,)_2 \\
| & (\,0\,1\,0\,1\,0\,1\,0\,1\,)_2 \\
\hline
 & (\,0\,1\,1\,1\,1\,1\,0\,1\,)_2
\end{array}
$$

Example use of |

- | can be used to turn some bits on
  - Let b be an arbitrary bit, then b | 1 = 1
  - Let b be an arbitrary bit, then b | 0 = b

```
int set_msb(int x) {

    return x | 0x80000000;
}
```

Bitwise operator ~

bit-value of ~x

~	x
0	1
1	0

result of ~0x69

$$\sim (\,0\,1\,1\,0\,1\,0\,0\,1\,)_2$$

$$(\,1\,0\,0\,1\,0\,1\,1\,0\,)_2$$

Bitwise operator ^

bit-value of x^y

^	x = 0	x = 1
y = 0	0	1
y = 1	1	0

result of 0x69^0x55

$$
\begin{array}{rr}
 & (\,0\,1\,1\,0\,1\,0\,0\,1\,)_2 \\
\wedge & (\,0\,1\,0\,1\,0\,1\,0\,1\,)_2 \\
\hline
 & (\,0\,0\,1\,1\,1\,1\,0\,0\,)_2
\end{array}
$$

Example use of ^ (XOR)

- Often used in encryption/decryption, checksum calculation
- Let x be any integer, what is x ^ x?

Bitwise operator <<

x << y, shift bit-vector x left by y positions

- Throw away bits shifted out on the left
- Fill in 0's on the right

	0 1 1 0 1 0 0 1
result of 0x69 << 3	0 1 0 0 1 0 0 0

Bitwise operator >>

- x >> y, shift bit-vector x right by y positions
  - Throw away bits shifted out on the right
  - (Logical shift) Fill with 0's on left

		1 0 1 0 1 0 0 1
Logical	result of 0xa9 >> 3	0 0 0 1 0 1 0 1

Bitwise operator >>

- x >> y, shift bit-vector x right by y positions
  - Throw away bits shifted out on the right
  - (Logical shift) Fill with 0's on the left
  - (Arithmetic shift) Replicate msb on the left

Arithmetic result of 0xa9 >> 3

1 0 1 0 1 0 0 1

1 1 1 1 0 1 0 1

Which shift is used in C ?

Arithmetic shift for signed numbers

Logical shifting on unsigned numbers

```
#include <stdio.h>
int main()
{
   int a = -1;
   unsigned int b = 1;
   printf("%d  %d\n", a>>1, b>>1);
}
```

```
Result: -1 0
```

Which shift is used?

Arithmetic shift for signed numbers

Logical shifting on unsigned numbers

```
#include <stdio.h>
int main()
{
  int a = -1;
  unsigned int b = 1;
  printf("%d %d\n”,(unsigned int)a>>1,
                   b>>1);
}
```

```
Result: 2147483647 0
```

Example use of shift

- Shift does multiplication (or division*) by 2's power cheaper

```
char x = 10;
x = x * 8; //equivalent to?
x = x / 4; //equivalent to?

char y = -2;
y = y * 2; //equivalent to?
y = y / 2; //equivalent to?
```

Example use of shift

```
// clear bit at position pos
// rightmost bit is at 0th pos

int clear_bit_at_pos(int x, int pos)
{
    int mask = 1 << pos;
    return x & (~mask);
}
```

Example use of shift

```
// set bit at position pos
// rightmost bit is at 0th pos

int set_bit_at_pos(int x, int pos)
{
    int mask = 1 << pos;
    return x | mask;
}
```

C's Control flow

- Same as Java
- conditional:
  - if ... else if... else
  - switch
- loops: while, for
  - continue
  - break

goto statements allow jump anywhere

goto *label*

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

error:
    code handling error
```

Avoid goto's whenever possible

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is dele-

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether

Avoid goto's whenever possible