C - Basics, Bitwise Operator

Jinyang Li

Lesson plan

- Overview
- C program organization
- Bitwise operators
- Control flow

C is an old programming language

C (1972)

Java (1995) Python (2.0, 2000)

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 directly on hardware	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	
Tiny standard library	Very Large library	Humongous library

Why learn C for CSO?

- C is a systems language
  - Language for writing OS and low-level code
  - System software written in C:

- Why learning C for CSO?
  - simple, low-level, “close to the hardware”

The simplest C program: “Hello World”

```c
#include <stdio.h>

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

hello.c

Equivalent to “importing” a library package

A function “exported” by stdio.h

Compile:

```
gcc hello.c –o hello
```

Run
:

```
./hello
```

If –o is not given, output executable file is a.out

C program with multiple files: naïve organization

```c
int sum(int x, int y)
{
   return x+y;
}
```

sum.c

```c
#include <stdio.h>
#include <assert.h>

void test_sum()
{
  int r = sum(1,1);
  assert(r == 2);
}

int main()
{
   test_sum();
}
```

test.c

```c
#include <stdio.h>

int main()
{
  printf("sum=%d\n", sum(-1,1));
}
```

main.c

Compile:

```
gcc sum.c test.c –o test
```

```
gcc sum.c main.c
```

Run:

```
./test
```

```
./a.out
```

Sum.c compiled twice.
Wasteful

C program with multiple files: *.h vs *.c files

```
int sum(int x, int y)
{
   return x+y;
}
```

sum.c

```
int sum(int x, int y);
```

sum.h

Equivalent to “importing” a package

```
#include <stdio.h>
#include <assert.h>
#include “sum.h”


void test_sum()
{
  int r = sum(1,1);
  assert(r == 2);
}

int main()
{
   test_sum();
}
```

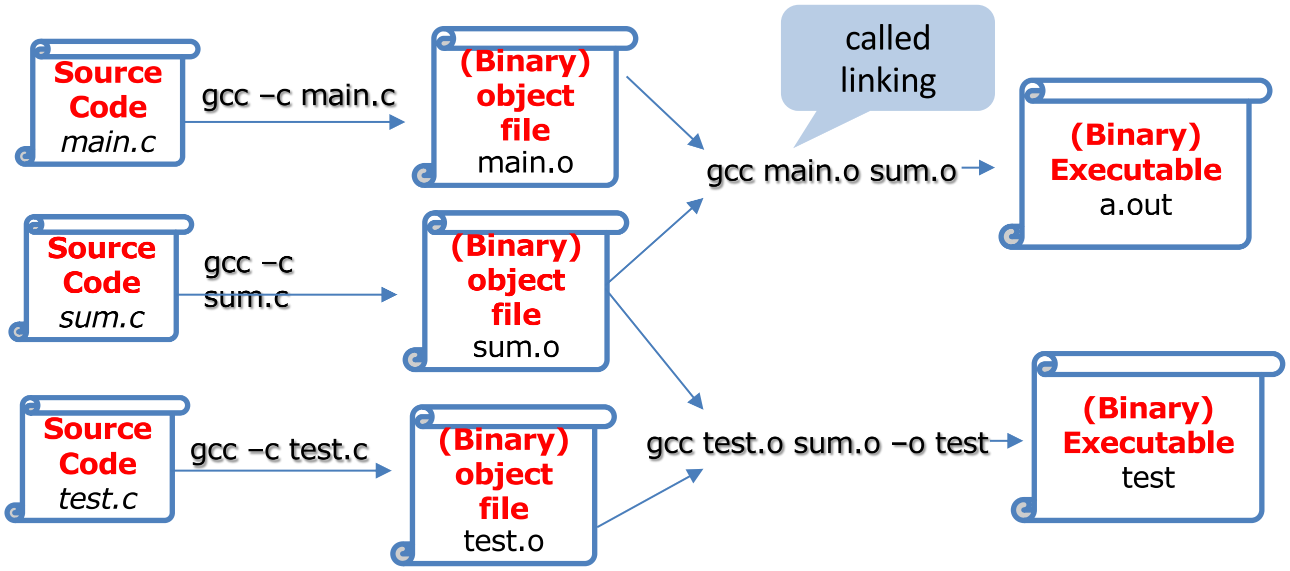
test.c

```
#include <stdio.h>
#include “sum.h”

int main()
{
  printf(“sum=%d\n”, sum(-1,1));
}
```

main.c

Common compilation sequence



C project uses the `make` tool to automate compiling with dependencies.

```
all: a.out test
test: test.o sum.o
    gcc $^ -o $@
a.out: main.c sum.o
    gcc $^ -o $@
%.o: %.c
    gcc -c $^
```

Makefile

Basic C

- C's syntax is very similar to Java
  - Java borrowed its syntax from C

Initial value

⚠ If uninitialized, variable can have any value

Variable declaration: `int a = 1;`

Type *Name*

Primitive Types (64-bit machine)

Either a character or an intger

type	size (bytes)	example
(unsigned) char	1	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

Next lecture

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;
}
```

```
$gcc test.c
$./a.out
-1 is larger than 1
```

No compiler warning!

Compiler converts int types to the one with the larger value (e.g. char → unsigned char → int → unsigned int)

-1 is implicitly cast to unsigned int $(4294967295)_{10}$

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 AND: &

Truth table (of boolean function AND)

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

How many rows if function has n boolean (aka 1-bit) inputs?

2^n

Operator & applies AND bitwise to two integers

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

  b & 0 = ?

  b & 1 = ?

b is any bit

```
int clear_msb(int x) {
  return x & 0x7fffffff;
}
```

Bitwise OR: |

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

Operator | applies OR bitwise to two integers

Result of 0x69 | 0x55

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

Example use of |

- | can be used to turn some bits on

  b | 1 = ?

  b | 0 = ?

```
int set_msb(int x) {

   return x | 0x80000000;

}
```

Bitwise NOT: ~

x	NOT x
0	1
1	0

Operator ~ applies NOT bitwise to two integers

result of ~0x69

$$\frac{\sim (\; 0\;1\;1\;0\;1\;0\;0\;1\;)_2}{(\; 1\;0\;0\;1\;0\;1\;1\;0\;)_2}$$

Bitwise XOR: ^

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Operator ^ applies XOR bitwise to two integers

result of 0x69^0x55

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

Bitwise left-shift: <<

x << y, treat x as a bit-vector, shift x left by y positions

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

result of `0x69<<3` ? $(0\ 1\ 1\ 0\ 1\ 0\ 0\ 1)_2$

$=(0\ 1\ 0\ 0\ 1\ 0\ 0\ 0)_2$

Bitwise right-shift: >>

- 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
  - **Arithmetic shift**: Replicate msb on the left

Result of (logical) right-shift: 0xa9 >> 3

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

$=(0\ 0\ 0\ 1\ 0\ 1\ 0\ 1)_2$

Result of (arithmetic) right-shift: 0xa9 >> 3

$=(1\ 1\ 1\ 1\ 0\ 1\ 0\ 1)_2$

Which shift is used in C ?

Arithmetic shift for signed numbers

Logical shifting on unsigned numbers

```
int a = 1;
a = a>>31;

int b = -1;
b = b >>31;
```

```
a= ??  b= ??
```

```
unsigned int b = -1;
b = b >>30;
```

```
b = ??
```

Example use of shift

```
int multiply_by_powers_of two(int x, int p)
{
        return x << p;
}
```

```
int divide_by_powers_of two(int x, int p)
{
        return x >> p;
}
```

Caveat: right-shift rounds down, different from integer division "/" which rounds towards zero. e.g. -1>>1=-1, but -1/2 = 0

Example use of shift

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

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

Lesson plan

- Overview
- C program organization
- Bitwise operators
- Control flow

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*

```
while (cond) {
    ...
}
B:
...
```

Any control flow primitive can be expressed as a bunch of goto's.

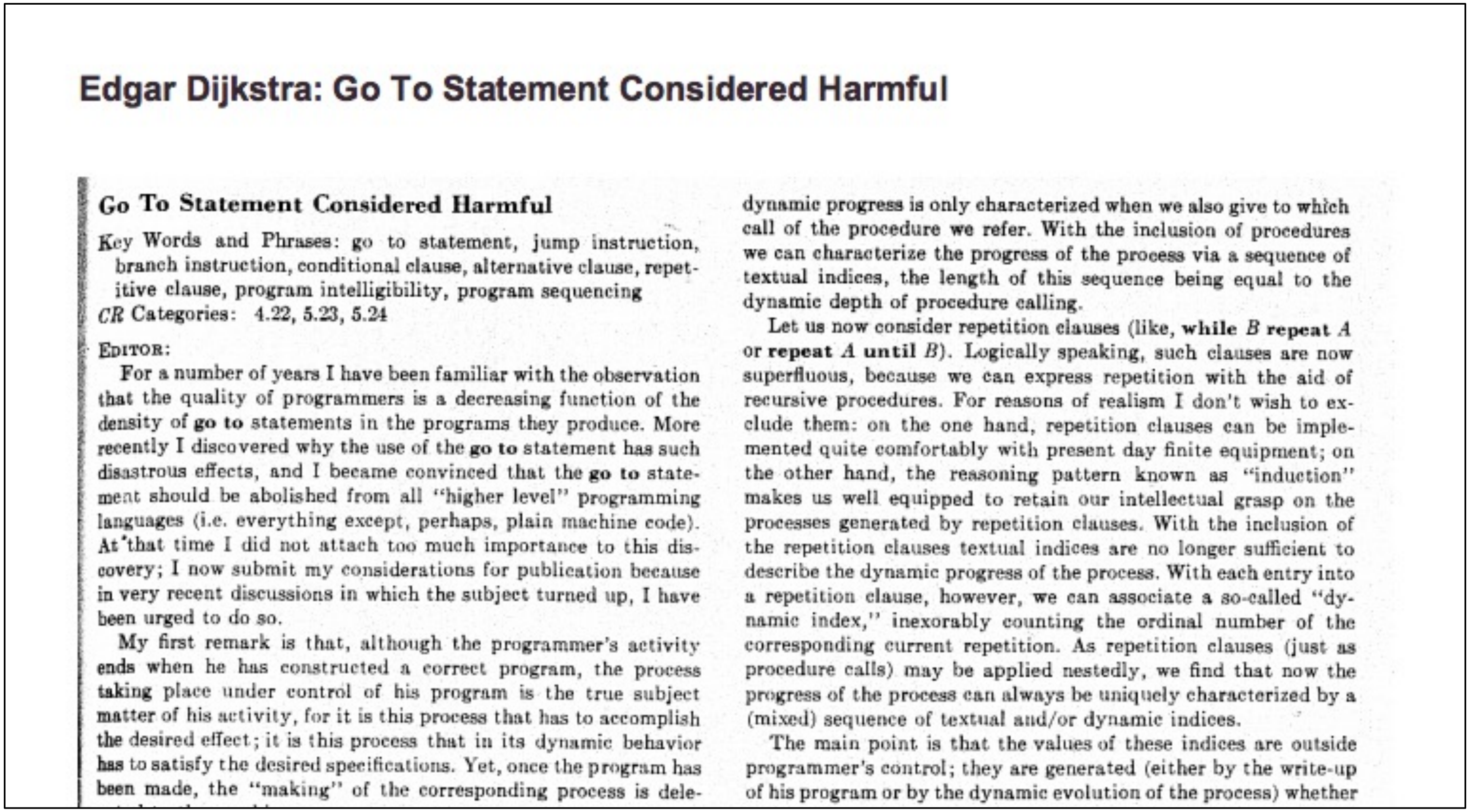
```
A:
 if (cond = false) goto B;
 ...
 goto A
B:
 ...
```

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

error:
    code handling error
```

The only acceptable scenario for using goto

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.

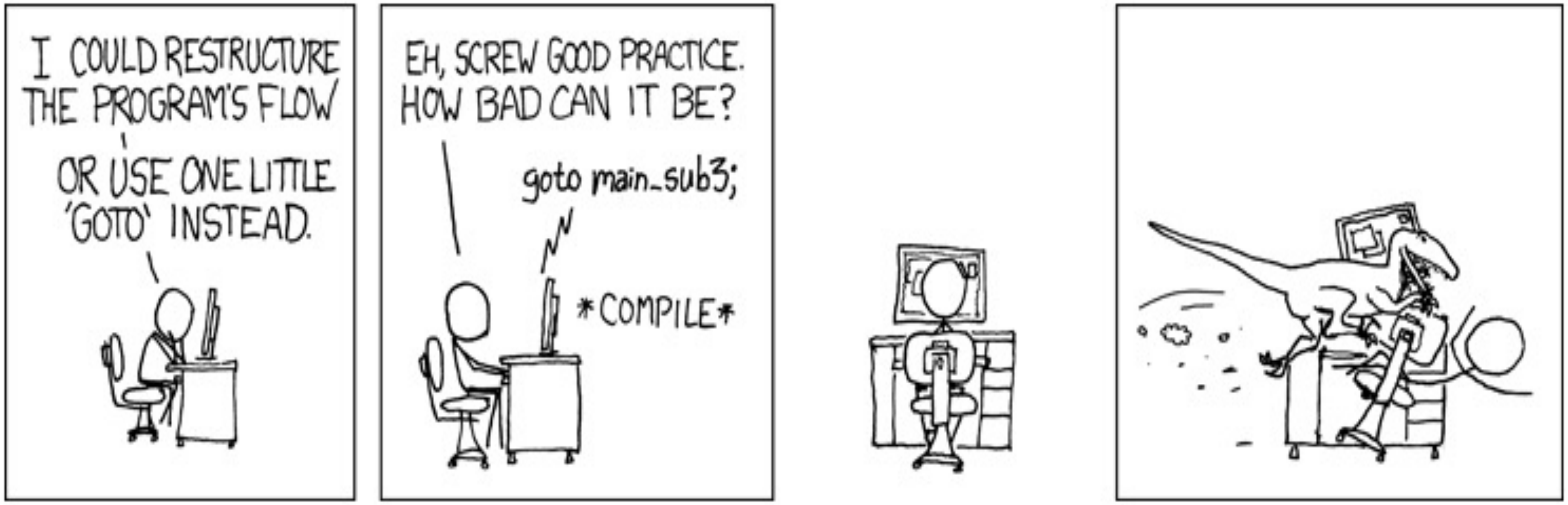
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its 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


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

Summary

- C program’s basic organization
  - *.c vs. *.h files
  - Compilation and make
- Bitwise operators, &, |, ~, ^, >>, <<
  - >> (arithmetic vs. logical)
- Control flow
  - goto is general, but results in spaghetti code