C - Basics, Bitwise Operator

Shuai Mu

Based on Tiger Wang and Jinyang Li' 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 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”

```
#include <stdio.h>

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

“Hello World”

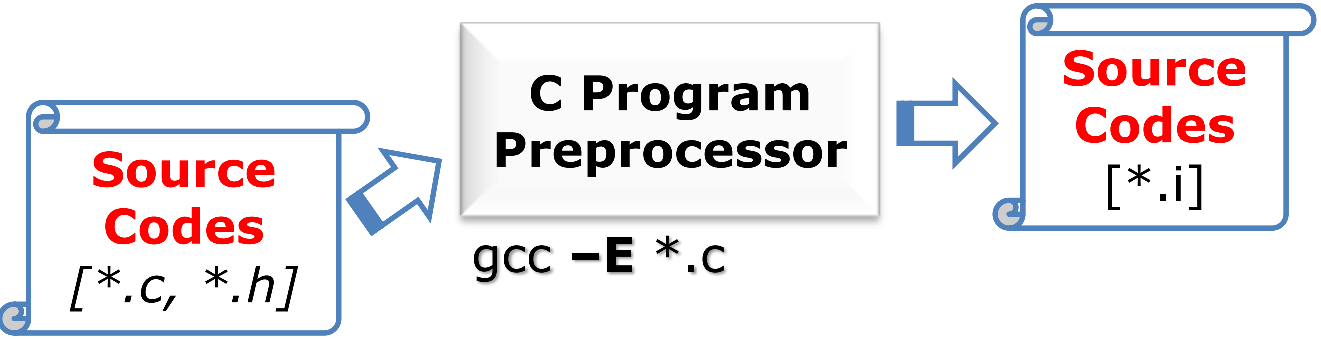
```c
#include <stdio.h>

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

Header file

Standard Library

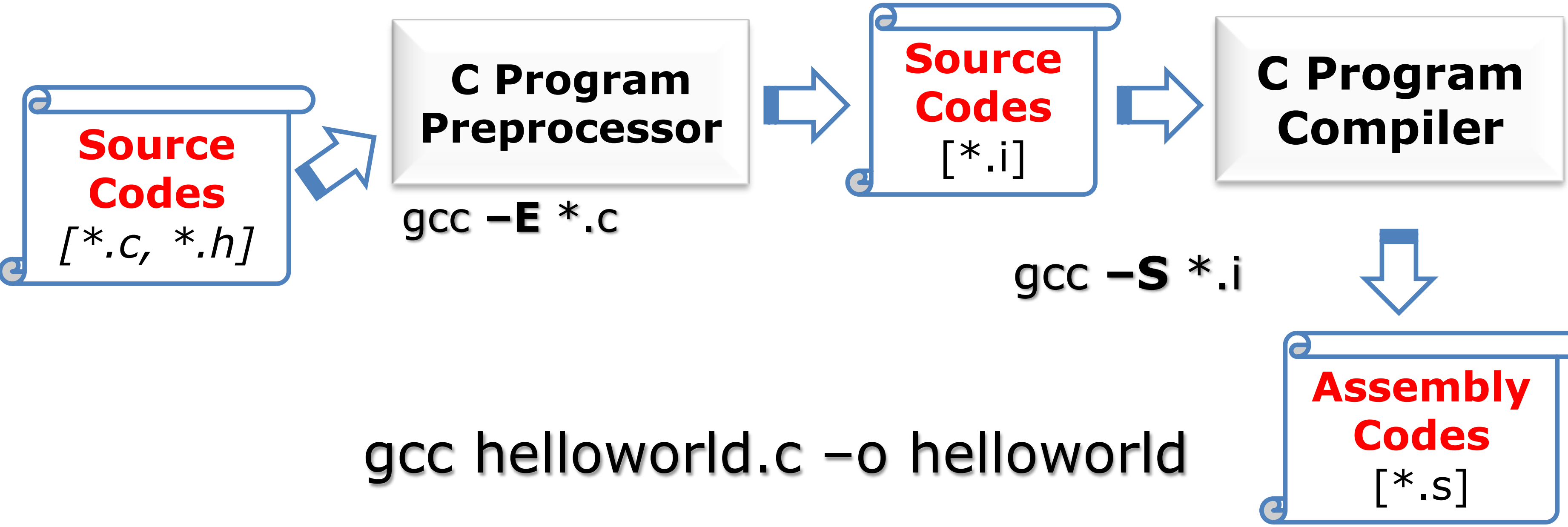
```
gcc helloworld.c –o helloworld
```

Compiling

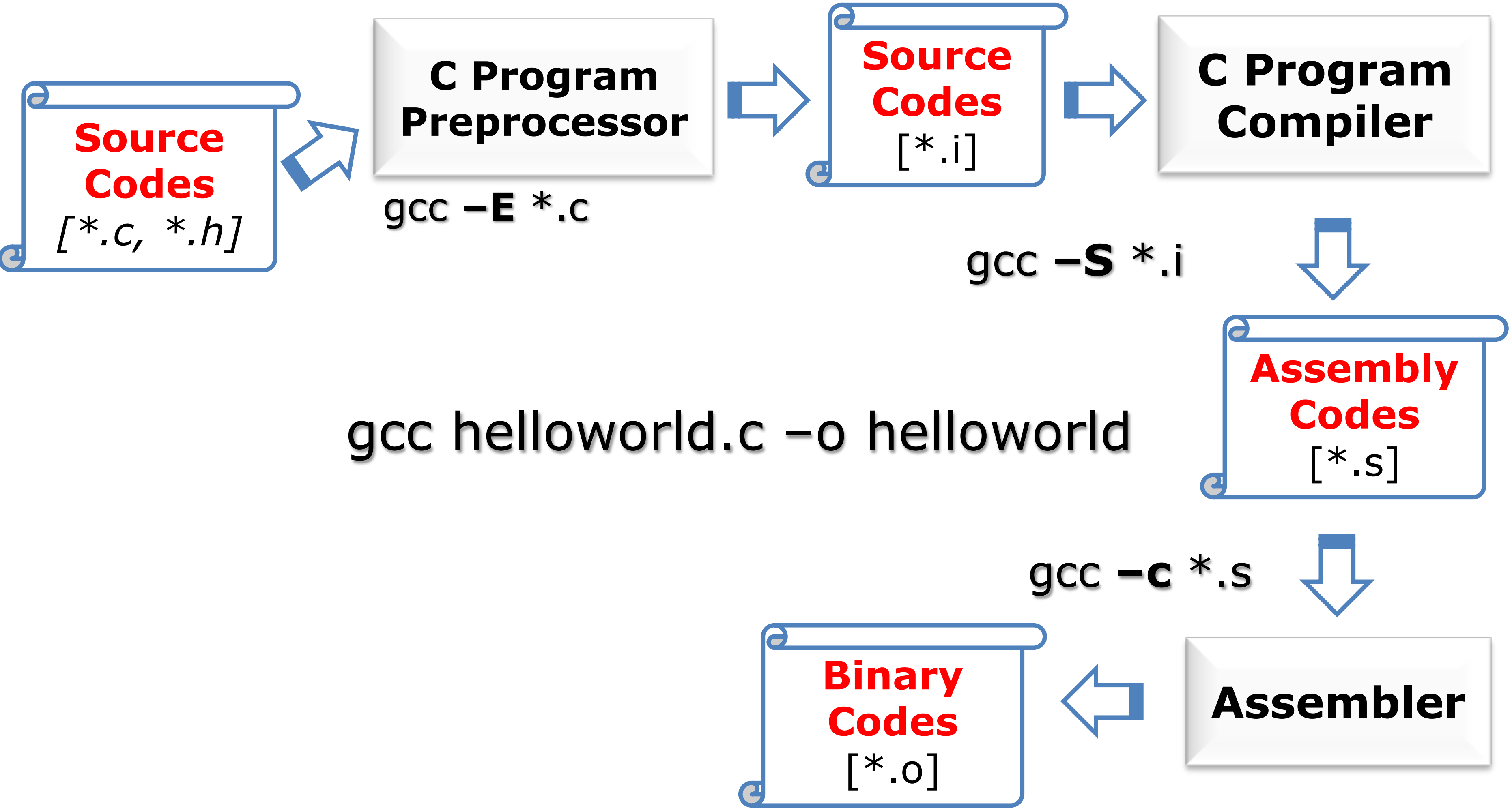


gcc helloworld.c –o helloworld

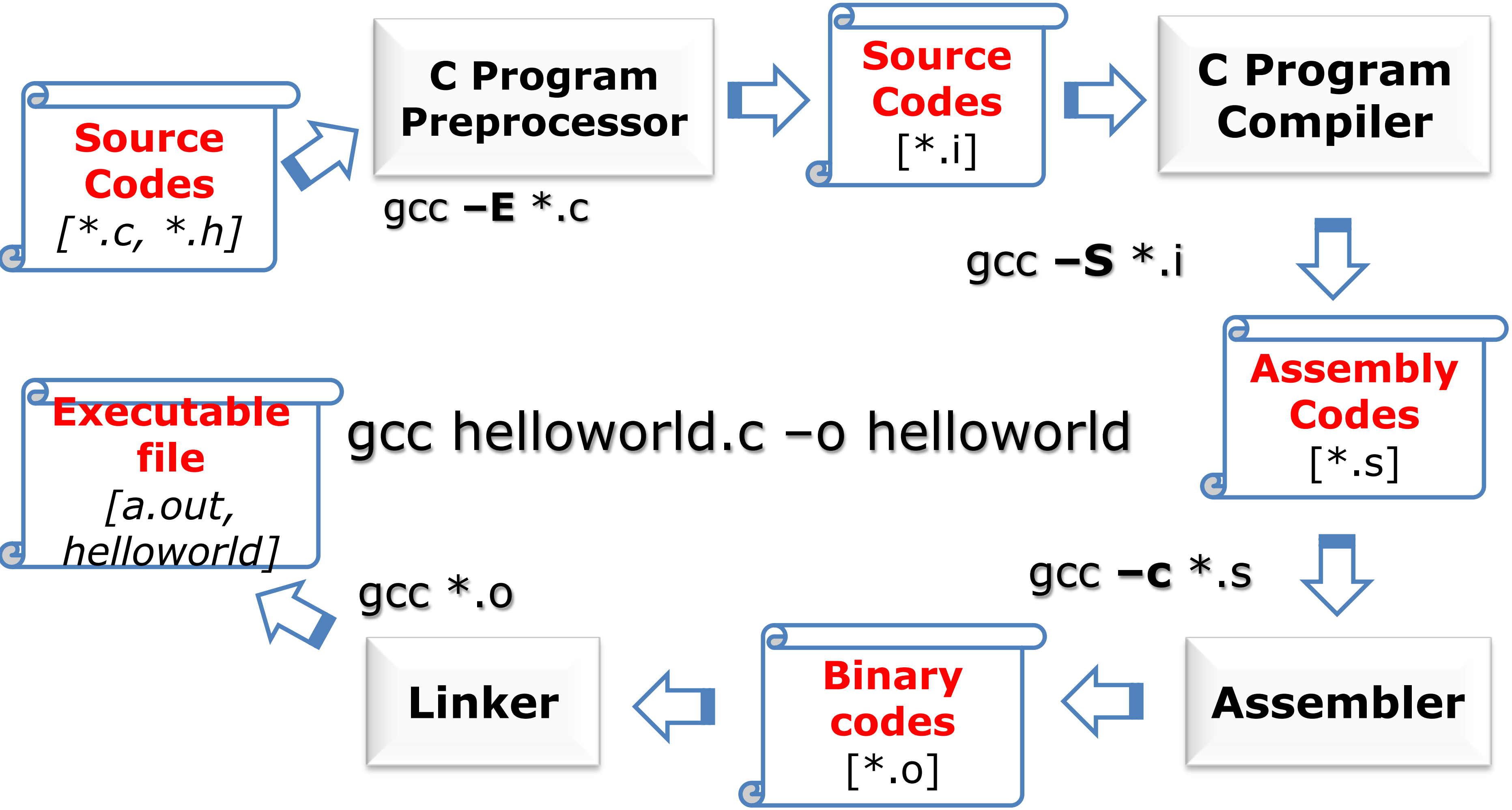
Compiling


Source Codes
[*.c, *.h]
C Program Preprocessor
gcc –E *.c
Source Codes
[*.i]
C Program Compiler
gcc –S *.i
Assembly Codes
[*.s]
gcc helloworld.c –o helloworld

Compiling


Source Codes
[*.c, *.h]
C Program Preprocessor
gcc **–E** *.c
Source Codes
[*.i]
C Program Compiler
gcc **–S** *.i
Assembly Codes
[*.s]
gcc helloworld.c –o helloworld
gcc **–c** *.s
Assembler
Binary Codes
[*.o]

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
Assembler
Binary codes
[*.o]
Linker
gcc *.o
Executable file
[a.out, helloworld]
gcc helloworld.c –o helloworld

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

Initial value

Declaration: `int a = 1;`

Type

Name

Variables

Declaration: `int a;`

Type *Name*

If not initialized, a can have any value

Value assignment: `a = 0;`

Primitive Types

64 bits machine

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

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 = 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

```
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)

```
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

		x	
	&	0	1
y	0	0	0
	1	0	1

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

Bitwise operator &

And (&)

- given two bits x and y, x & y = 1 when both x = 1 and y = 1

		x	
	&	0	1
y	0	0	0
	1	0	1

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

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

x

y	&	0	1
	0	0	0
	1	0	1

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

Bitwise operator |

Or (|)

- given two bits x and y, x | y = 1 when either x = 1 or y = 1

		x	
	|	0	1
y	0	0	1
	1	1	1

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

Bitwise operator |

Or (|)

– given two bits x and y, x | y = 1 when either x = 1 or y = 1

<table>
<tr><td></td><td></td><td>x</td><td></td></tr>
<tr><td></td><td>|</td><td>0</td><td>1</td></tr>
<tr><td>y</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td></td><td>1</td><td>1</td><td>1</td></tr>
</table>

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

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

$$
\begin{array}{cc|cc}
 & & \multicolumn{2}{c}{x} \\
 & | & 0 & 1 \\
\hline
y & 0 & 0 & 1 \\
 & 1 & 1 & 1
\end{array}
$$

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

Bitwise operator ~

Not (~)

- given a bit x, ~x = 1 when x = 0
- One's complement

~	x
0	1
1	0

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

Bitwise operator ~

Not (~)

- given a bit x, ~x = 1 when x = 0
- One's complement

~	x
0	1
1	0

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

Bitwise operator ^

Xor (^)

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

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

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

Bitwise operator ^

Xor (^)

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

		x	
	^	0	1
y	0	0	1
	1	1	0

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

Bitwise operator <<

Left shift (“<<”)

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

x 0 1 1 0 1 0 0 1

x << 3

Bitwise operator <<

Left shift ("<<")

- x << y, shift bit-vector x left y positions
  - Throw away extra bits on left
  - Fill with 0's on right

x	0 1 1 0 1 0 0 1
x << 3	0 1 0 0 1 0 0 0

Bitwise operator >>

- Right shift (“>>”)
  - x >> 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

	x	1 0 1 0 1 0 0 1
Logical	x >> 3	0 0 0 1 0 1 0 1

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
- Arithmetic shift
  - Replicate most significant bit on the left

	x	1 0 1 0 1 0 0 1
Logical	x >> 3	0 0 0 1 0 1 0 1
Arithmetic	x >> 3	1 1 1 1 0 1 0 1

Bitwise operator >>

Right shift (“>>”)

- x >> 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

	x	1 0 1 0 1 0 0 1
Logical	x >> 3	0 0 0 1 0 1 0 1
Arithmetic	x >> 3	1 1 1 1 0 1 0 1

Which operation is used in C?

Arithmetic shifting on signed number, logical shifting on unsigned number

```
#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

```
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

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

Control flow

```
int a = b + 1
```

expression

Expression

Combine the variables and constants to produce new values

```
int a = b + 1

int c = ( d << 1 ) + 2

float f = (float) c
```

Control flow

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

statement

Control flow

```
{
    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)
    statement₁
else
    statement₂
```

```
if (expression₁)
    statement₁
else if (expression₂)
    statement₂
else
    statement₃
```

Control flow

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

Control flow

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

Control flow

```
while (expression) {
   statement
}


for(expr1; expr2; expr3) {
   statement
}
```

Control flow

```
expr₁;
while(expr₂) {
    statement
    expr₃;
}

for(expr1; expr2; expr3) {
    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.

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

error:
    clean up the mess
```

Exercises

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

```
bool isEven(int n) {

}
```

Exercises

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

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

Exercises

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

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

Exercises

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

```
bool isPowerOfTwo(int n) {

}
```

Exercises

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

```
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?

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

Exercises

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

```
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)

```
int count_one(int n) {


}
```

Exercises

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

(n > 0)

```
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 ?

```
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 ?

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