Introduction to Computers and Programming

Lecture 3 – Loop and Data type

Chap 6 & 7

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Loops

Iteration Statements

- C’s iteration statements are used to set up loops.
- A ***loop*** is a statement whose job is to repeatedly execute some other statement (the ***loop body***).
- In C, every loop has a ***controlling expression***.
- Each time the loop body is executed (an ***iteration*** of the loop), the controlling expression is evaluated.
  - If the expression is true (has a value that’s not zero) the loop continues to execute.

Iteration Statements

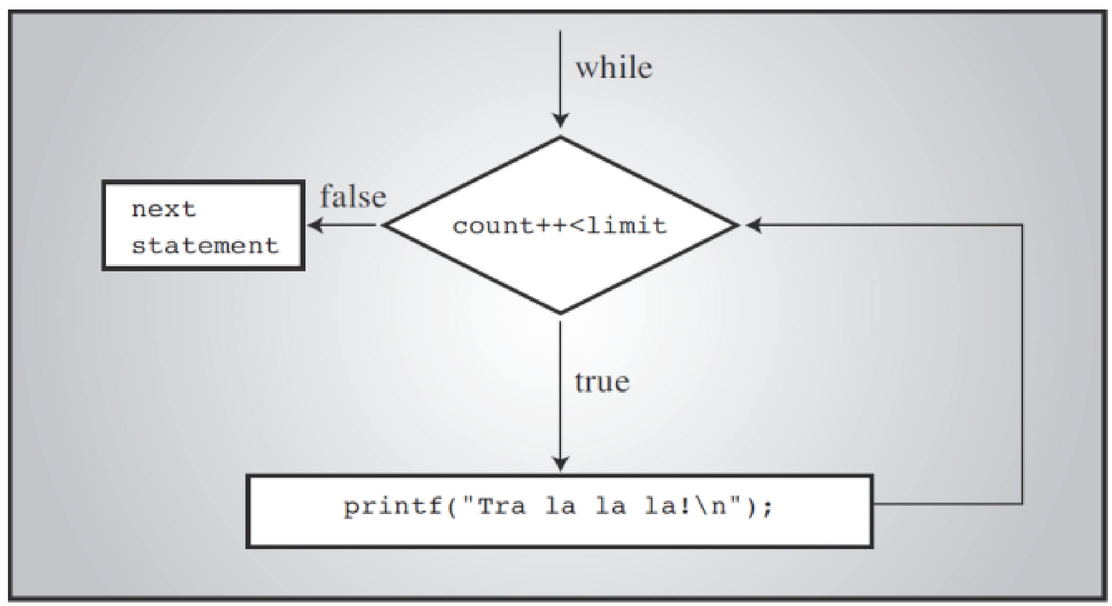
- C provides three iteration statements:
  - The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed.
  - The `do` statement is used if the expression is tested *after* the loop body is executed.
  - The `for` statement is convenient for loops that increment or decrement a counting variable.

The `while` Statement

- Using a `while` statement is the easiest way to set up a loop.
- The `while` statement has the form

  `while (` *expression* `)` *statement*

- *expression* is the controlling expression; *statement* is the loop body.



The `while` Statement

- Example of a `while` statement:

```
while (i < n)   /* controlling expression */
  i = i * 2;    /* loop body */
```

- When a `while` statement is executed, the controlling expression is evaluated first.
- If its value is nonzero (true), the loop body is executed and the expression is tested again.
- The process continues until the controlling expression eventually has the value zero.

The `while` Statement

- A `while` statement that computes the smallest power of 2 that is greater than or equal to a number `n`:

```
i = 1;
while (i < n)
  i = i * 2;
```

- A trace of the loop when `n` has the value 10:

`i = 1;`	`i` is now 1.
Is `i` < `n`?	Yes; continue.
`i = i * 2;`	`i` is now 2.
Is `i` < `n`?	Yes; continue.
`i = i * 2;`	`i` is now 4.
Is `i` < `n`?	Yes; continue.
`i = i * 2;`	`i` is now 8.
Is `i` < `n`?	Yes; continue.
`i = i * 2;`	`i` is now 16.
Is `i` < `n`?	No; exit from loop.

The `while` Statement

- If multiple statements are needed, use braces to create a single compound statement:

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {
  i = i * 2;
}
```

The `while` Statement

- The following statements display a series of "countdown" messages:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- The final message printed is `T minus 1 and counting`.

The `while` Statement

- Observations about the `while` statement:
  - The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by `i > 0` terminates, `i` must be less than or equal to 0.
  - The body of a `while` loop may not be executed at all, because the controlling expression is tested *before* the body is executed.
  - A `while` statement can often be written in a variety of ways. A more concise version of the countdown loop:

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

Infinite Loops

- A `while` statement won't terminate if the controlling expression always has a nonzero value.
- C programmers sometimes deliberately create an ***infinite loop*** by using a nonzero constant as the controlling expression:

```
while (1) …
```

- A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

Program: Summing a Series of Numbers

- The `sum.c` program sums a series of integers entered by the user:

```
This program sums a series of integers.
Enter integers (0 to terminate): 8 23 71 5 0
The sum is: 107
```

- The program will need a loop that uses `scanf` to read a number and then adds the number to a running total.

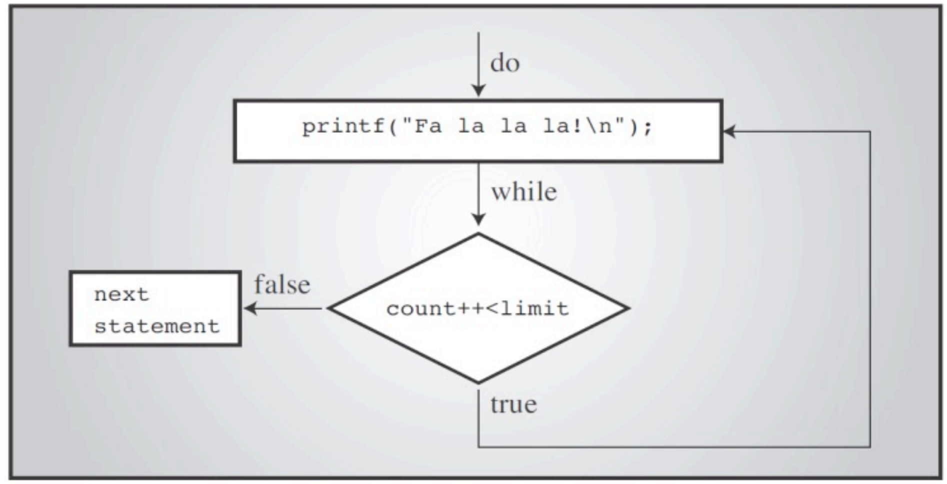
sum.c

```
/* Sums a series of numbers */

#include <stdio.h>

int main(void)
{
  int n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%d", &n);
  while (n != 0) {
    sum += n;
    scanf("%d", &n);
  }
  printf("The sum is: %d\n", sum);

  return 0;
}
```

The do Statement

- General form of the `do` statement:

```
do
    statement; …..
while ( expression ) ;
```



- When a `do` statement is executed, the loop body is executed first, then the controlling expression is evaluated.
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.

The do Statement

- The countdown example rewritten as a `do` statement:

```
i = 10;
do {
  printf("T minus %d and counting\n", i);
  --i;
} while (i > 0);
```

- The `do` statement is often indistinguishable from the `while` statement.
- The only difference is that the body of a `do` statement is always executed at least once.

The `do` Statement

- It's a good idea to use braces in *all* `do` statements, whether or not they're needed, because a `do` statement without braces can easily be mistaken for a `while` statement:

```
do
  printf("T minus %d and counting\n", i--);
while (i > 0);
```

- A careless reader might think that the word `while` was the beginning of a `while` statement.

Program: Calculating the Number of Digits in an Integer

- The `numdigits.c` program calculates the number of digits in an integer entered by the user:

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

- The program will divide the user's input by 10 repeatedly until it becomes 0; the number of divisions performed is the number of digits.
- Writing this loop as a `do` statement is better than using a `while` statement, because every integer—even 0—has at least one digit.

numdigits.c

```
/* Calculates the number of digits in an integer */

#include <stdio.h>

int main(void)
{
  int digits = 0, n;

  printf("Enter a nonnegative integer: ");
  scanf("%d", &n);

  do {
    n /= 10;
    digits++;
  } while (n > 0);

  printf("The number has %d digit(s).\n", digits);

  return 0;
}
```

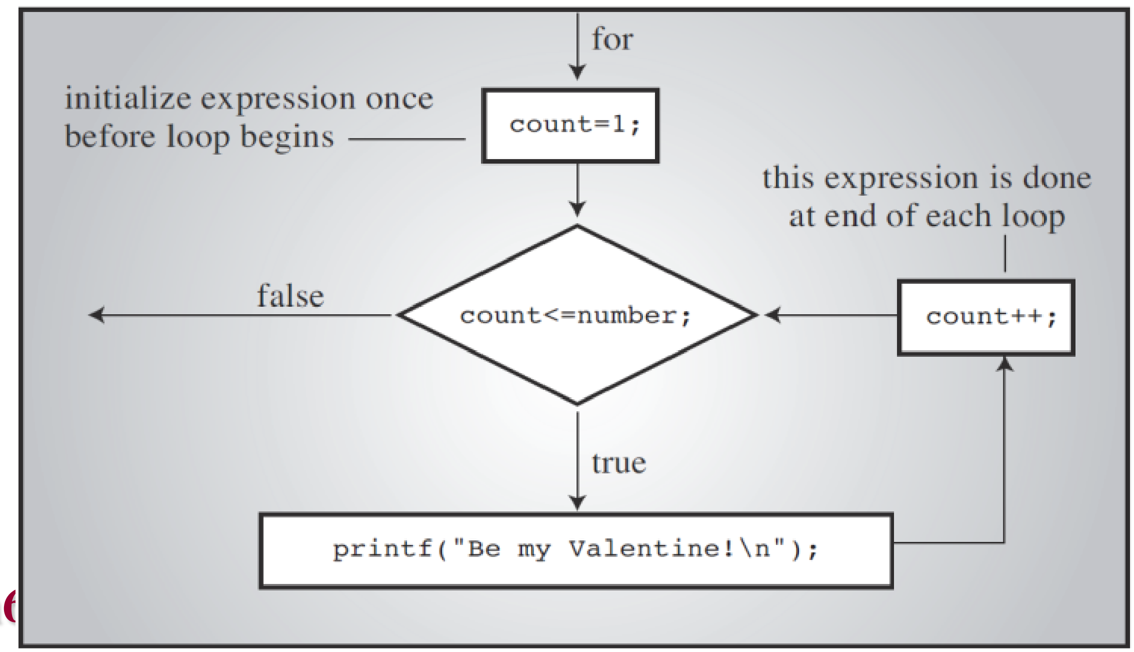
The `for` Statement

- The `for` statement is ideal for loops that have a “counting” variable.
- General form of the `for` statement:

  `for (` *expr1* `;` *expr2* `;` *expr3* `)` *statement*

- Example:

```
for (count = 1; count <= N; count++)
  printf(“Print this value %i\n", count);
```



The `for` Statement

- The `for` statement is closely related to the `while` statement.
- Except in a few rare cases, a `for` loop can always be replaced by an equivalent `while` loop:

expr1;

`while (` *expr2* `) {`

    statement

    expr3`;`

`}`

- *expr1* is an initialization step that's performed only once, before the loop begins to execute.

The `for` Statement

- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
- *expr3* is an operation to be performed at the end of each loop iteration.
- The result when this pattern is applied to the previous `for` loop:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

The `for` Statement

- Studying the equivalent `while` statement can help clarify the fine points of a `for` statement.
- For example, what if `i--` is replaced by `--i`?

```
for (i = 10; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

- The equivalent `while` loop shows that the change has no effect on the behavior of the loop:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  --i;
}
```

The `for` Statement

- Since the first and third expressions in a `for` statement are executed as statements, their values are irrelevant—they're useful only for their side effects.
- Consequently, these two expressions are usually assignments or increment/decrement expressions.

`for` Statement Idioms

- The `for` statement is usually the best choice for loops that "count up" (increment a variable) or "count down" (decrement a variable).
- A `for` statement that counts up or down a total of `n` times will usually have one of the following forms:

Counting up from 0 to* `n`*–1: `for (i = 0; i < n; i++) …`

Counting up from 1 to* `n`*: `for (i = 1; i <= n; i++) …`

Counting down from* `n`*–1 to 0:

```
for (i = n - 1; i >= 0; i--) …
```

Counting down from* `n` *to 1:

```
for (i = n; i > 0; i--) …
```

`for` Statement Idioms

- Common `for` statement errors:
  - Using < instead of > (or vice versa) in the controlling expression. "Counting up" loops should use the < or <= operator. "Counting down" loops should use > or >=.
  - Using == in the controlling expression instead of <, <=, >, or >=.
  - "Off-by-one" errors such as writing the controlling expression as `i <= n` instead of `i < n`.

Omitting Expressions in a `for` Statement

- C allows any or all of the expressions that control a `for` statement to be omitted.
- If the *first* expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false:

```
for (i = 10; i > 0;)
  printf("T minus %d and counting\n", i--);
```

Omitting Expressions in a `for` Statement

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a while statement in disguise:

```
for (; i > 0;)
  printf("T minus %d and counting\n", i--);
```

  is the same as

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

- The `while` version is clearer and therefore preferable.

Omitting Expressions in a `for` Statement

- If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion).
- For example, some programmers use the following `for` statement to establish an infinite loop:

```
for (;;) …
```

`for` Statements in C99

- In C99, the first expression in a `for` statement can be replaced by a declaration.
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
   …
```

- The variable `i` need not have been declared prior to this statement.

`for` Statements in C99

- A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);
  /* legal; i is visible inside loop */
  …
}
printf("%d", i);    /*** WRONG ***/
```

`for` Statements in C99

- Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand.
- However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.
- A `for` statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
  …
```

The Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
- This effect can be accomplished by using a ***comma expression*** as the first or third expression in the `for` statement.
- A comma expression has the form

  expr1 , *expr2*

  where *expr1* and *expr2* are any two expressions.

The Comma Operator

- A comma expression is evaluated in two steps:
  - First, *expr1* is evaluated and its value discarded.
  - Second, *expr2* is evaluated; its value is the value of the entire expression.
- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
- When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated.
  - If `i` and `j` have the values 1 and 5, respectively, the value of the expression will be 7, and `i` will be incremented to 2.

The Comma Operator

- The comma operator is left associative, so the compiler interprets

  ```
  i = 1, j = 2, k = i + j
  ```

  as

  ```
  ((i = 1), (j = 2)), (k = (i + j))
  ```

- Since the left operand in a comma expression is evaluated before the right operand, the assignments `i = 1`, `j = 2`, and `k = i + j` will be performed from left to right.

The Comma Operator

- The comma operator makes it possible to "glue" two expressions together to form a single expression.
- Certain macro definitions can benefit from the comma operator.
- The `for` statement is the only other place where the comma operator is likely to be found.
- Example:

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

- With additional commas, the `for` statement could initialize more than two variables.

Exiting from a Loop

- The normal exit point for a loop is at the beginning (as in a `while` or `for` statement) or at the end (the `do` statement).
- Using the `break` statement, it's possible to write a loop with an exit point in the middle or a loop with more than one exit point.

The `break` Statement

- The `break` statement can transfer control out of a switch statement, but it can also be used to jump out of a `while`, `do`, or `for` loop.
- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

The `break` Statement

- After the loop has terminated, an `if` statement can be use to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The `break` Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end.
- Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape only one level of nesting.
- Example:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop.

The `continue` Statement

- The `continue` statement is similar to `break`:
  - `break` transfers control just past the end of a loop.
  - `continue` transfers control to a point just before the end of the loop body.
- With `break`, control leaves the loop; with `continue`, control remains inside the loop.
- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

The `continue` Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```

The `continue` Statement

- The same loop written without using `continue`:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```

The `goto` Statement

- The `goto` statement is capable of jumping to any statement in a function, provided that the statement has a ***label.***
- A label is just an identifier placed at the beginning of a statement:

  identifier `:` *statement*

- A statement may have more than one label.
- The `goto` statement itself has the form

  `goto` *identifier* `;`

- Executing the statement `goto` *L*`;` transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

The `goto` Statement

- If C didn't have a `break` statement, a `goto` statement could be used to exit from a loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;
done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The `goto` Statement

- The `goto` statement is rarely needed in everyday C programming.
- The `break`, `continue`, and `return` statements—which are essentially restricted `goto` statements—and the `exit` function are sufficient to handle most situations that might require a `goto` in other languages.
- Nonetheless, the `goto` statement can be helpful once in a while.

The `goto` Statement

- Consider the problem of exiting a loop from within a `switch` statement.
- The `break` statement doesn't have the desired effect: it exits from the `switch`, but not from the loop.
- A `goto` statement solves the problem:

```
while (…) {
  switch (…) {
    …
    goto loop_done;    /* break won't work here */
    …
  }
}
loop_done: …
```

- The `goto` statement is also useful for exiting from nested loops.

The Null Statement

- A statement can be ***null***—devoid of symbols except for the semicolon at the end.
- The following line contains three statements:

```
i = 0; ; j = 1;
```

- The null statement is primarily good for one thing: writing loops whose bodies are empty.

The Null Statement

- Consider the following prime-finding loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

- If the `n % d == 0` condition is moved into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)
  /* empty loop body */ ;
```

- To avoid confusion, C programmers customarily put the null statement on a line by itself.

The Null Statement

- Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement.
- Example 1:

```
if (d == 0);                                /*** WRONG ***/
  printf("Error: Division by zero\n");
```

  The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.
- Example 2:

```
i = 10;
while (i > 0);                              /*** WRONG ***/
{
  printf("T minus %d and counting\n", i);
  --i;
}
```

  The extra semicolon creates an infinite loop.

The Null Statement

- Example 3:

```
i = 11;
while (--i > 0);                     /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

The loop body is executed only once; the message printed is:

```
T minus 0 and counting
```

- Example 4:

```
for (i = 10; i > 0; i--);            /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

Again, the loop body is executed only once, and the same message is printed as in Example 3.

Data types

Basic Types

- C's ***basic*** (built-in) ***types:***
  - Integer types, including long integers, short integers, and unsigned integers
  - Floating types (`float`, `double`, and `long double`)
  - `char`
  - `_Bool` (C99)
- Values of an ***integer type*** are whole numbers.
  - two categories: signed and unsigned
- Values of a floating type can have a fractional part as well.

Signed and Unsigned Integers

- Binary: $X_2 \rightarrow X_{10}$

  $X_{10} = b_3{*}2^3 + b_2{*}2^2 + b_1{*}2^1 + b_0{*}2^0$
- Two's complement to represent -A

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- The leftmost bit of a ***signed*** integer (known as the ***sign bit***) is 0 if the number is positive or zero, 1 if it's negative.
- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be ***unsigned.***

Integer Types

- The `int` type is usually 32 bits, but may be 16 bits on older CPUs.
- ***Long*** integers may have more bits than ordinary integers; ***short*** integers may have fewer bits.
- The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form integer types.
- Only six combinations produce different types:

```
short int    unsigned short int
int          unsigned int
long int     unsigned long int
```

- The order of the specifiers doesn't matter. Also, the word `int` can be dropped (`long int` can be abbreviated to just `long`).

Representing information as bits

- Bit: a basic unit to store 0 & 1
- Byte = 8 bits
- Word = 4 bytes, 32 bits, or 64 bits
- Representing information as bits

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Integer Types

- Typical ranges on a 32-bit machine:

Type	***Smallest Value***	***Largest Value***
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Values for *W* = 16

	Decimal	Hex	Binary
UMax	**65535**	**FF FF**	11111111 11111111
TMax	**32767**	7F FF	01111111 11111111
TMin	**-32768**	80 00	10000000 00000000
-1	**-1**	**FF FF**	11111111 11111111
0	**0**	00 00	00000000 00000000

Integer Types

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	-2^{63}	$2^{63}-1$
`unsigned long int`	0	$2^{64}-1$

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Octal and Hexadecimal Numbers

- Octal numbers use only the digits 0 through 7.
- Each position in an octal number represents a power of 8.
  - The octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.
- A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  - The hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Integer Constants

- ***Decimal*** constants contain digits between 0 and 9, but must not begin with a zero:

```
15   255   32767
```

- ***Octal*** constants contain only digits between 0 and 7, and must begin with a zero:

```
017   0377   077777
```

- ***Hexadecimal*** constants contain digits between 0 and 9 and letters between `a` and `f`, and always begin with `0x`:

```
0xf   0xff   0x7fff
```

- The letters in a hexadecimal constant may be either upper or lower case:

```
0xff   0xfF   0xFf   0xFF   0Xff   0XfF   0XFf   0XFF
```

Integer Constants

- To force the compiler to treat a constant as a long integer, just follow it with the letter `L` (or `l`):

  `15L   0377L   0x7fffL`

- To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

  `15U   0377U   0x7fffU`

- `L` and `U` may be used in combination:

  `0xffffffffUL`

  The order of the `L` and `U` doesn't matter, nor does their case.

Reading and Writing Integers

- Reading and writing unsigned, short, and long integers requires new conversion specifiers.
- When reading or writing an *unsigned* integer, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification.

```
unsigned int u;

scanf("%u", &u);  /* reads  u in base 10 */
printf("%u", u);  /* writes u in base 10 */
scanf("%o", &u);  /* reads  u in base  8 */
printf("%o", u);  /* writes u in base  8 */
scanf("%x", &u);  /* reads  u in base 16 */
printf("%x", u);  /* writes u in base 16 */
```

Reading and Writing Integers

- When reading or writing a *short* integer, put the letter `h` in front of `d`, `o`, `u`, or `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter `l` ("ell," not "one") in front of `d`, `o`, `u`, or `x`.
- When reading or writing a *long long* integer (C99 only), put the letters `ll` in front of `d`, `o`, `u`, or `x`.

Floating Types

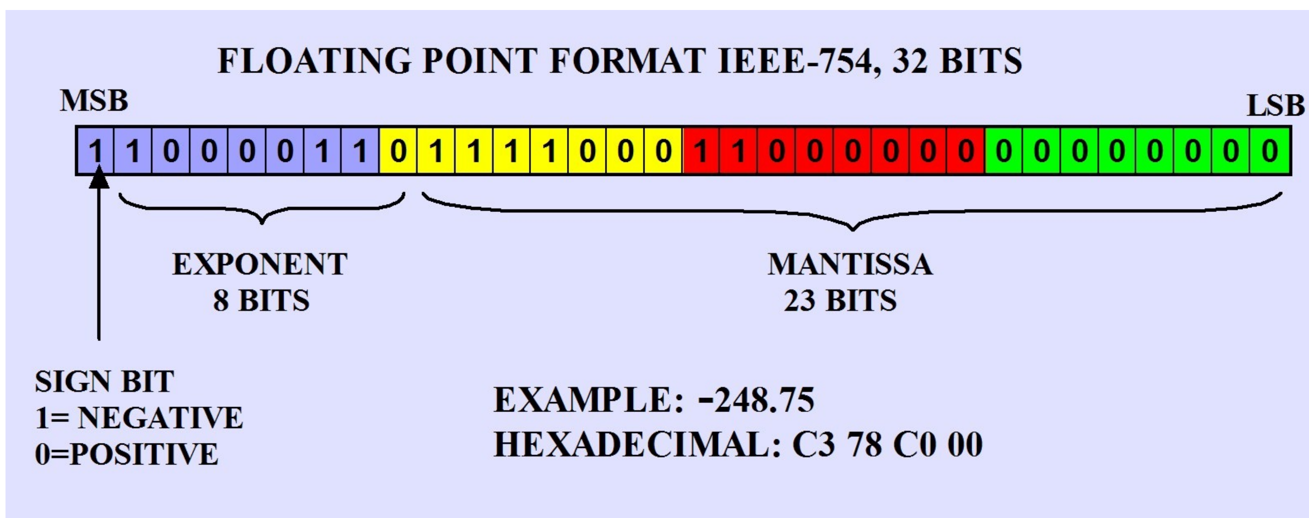
- C provides three ***floating types,*** corresponding to different floating-point formats:
  - `float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).

The IEEE Floating-Point Standard

- IEEE Standard 754 was developed by the Institute of Electrical and Electronics Engineers.
- Two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits).
- Numbers are stored in a form of scientific notation, with each number having a ***sign,*** an ***exponent,*** and a ***fraction.***



Floating Types

- Characteristics of `float` and `double` when implemented according to the IEEE standard:

Type	*Smallest Positive Value*	*Largest Value*	*Precision*
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	2.22507×10^{-308}	1.79769×10^{308}	15 digits

- On computers that don't follow the IEEE standard, this table won't be valid.
- In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`.

Floating Constants

- Floating constants can be written in a variety of ways.
- Valid ways of writing the number 57.0:

```
57.0   57.   57.0e0   57E0   5.7e1   5.7e+1
.57e2  570.e-1
```

- A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled.
- If an exponent is present, it must be preceded by the letter `E` (or `e`). An optional `+` or `-` sign may appear after the `E` (or `e`).

Floating Constants

- By default, floating constants are stored as double-precision numbers.
- To indicate that only single precision is desired, put the letter `F` (or `f`) at the end of the constant (for example, `57.0F`).
- To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

Reading and Writing Floating-Point Numbers

- The conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers.
- When reading a value of type `double`, put the letter `l` in front of `e`, `f`, or `g`:

```
double d;

scanf("%lf", &d);
```

- *Note:* Use `l` only in a `scanf` format string, not a `printf` string.
- In a `printf` format string, the `e`, `f`, and `g` conversions can be used to write either `float` or `double` values.
- When reading or writing a value of type `long double`, put the letter `L` in front of `e`, `f`, or `g`.

Character Sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero         */
ch = ' ';    /* space        */
```

- Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers.*
- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.
- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.
- Character constants actually have `int` type rather than `char` type.

Operations on Characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';        /* i is now 97   */
ch = 65;        /* ch is now 'A' */
ch = ch + 1;    /* ch is now 'B' */
ch++;           /* ch is now 'C' */
```

Operations on Characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.
- These values depend on the character set in use, so programs that use `<`, `<=`, `>`, and `>=` to compare characters may not be portable.

Operations on Characters

- The fact that characters have the same properties as numbers has advantages.
- For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

- Disadvantages of treating characters as numbers:
  - Can lead to errors that won't be caught by the compiler.
  - Allows meaningless expressions such as `'a' * 'b' / 'c'`.
  - Can hamper portability, since programs may rely on assumptions about the underlying character set.

Signed and Unsigned Characters

- The `char` type—like the integer types—exists in both signed and unsigned versions.
- Signed characters normally have values between –128 and 127. Unsigned characters have values between 0 and 255.
- Some compilers treat `char` as a signed type, while others treat it as an unsigned type. Most of the time, it doesn't matter.
- C allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;
```

Arithmetic Types

- The integer types and floating types are collectively known as ***arithmetic types.***
- A summary of the arithmetic types in C89, divided into categories and subcategories:
  - Integral types
    - `char`
    - Signed integer types (`signed char`, `short int`, `int`, `long int`)
    - Unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`)
    - Enumerated types
  - Floating types (`float`, `double`, `long double`)

Escape Sequences

- A character constant is usually one character enclosed in single quotes.
- However, certain special characters—including the new-line character—can't be written in this way, because they're invisible (nonprinting) or because they can't be entered from the keyboard.
- ***Escape sequences*** provide a way to represent these characters.
- There are two kinds of escape sequences: ***character escapes*** and ***numeric escapes.***

Escape Sequences

- A complete list of character escapes:

Name	*Escape Sequence*
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Question mark	`\?`
Single quote	`\'`
Double quote	`\"`

Escape Sequences

- Character escapes are handy, but they don’t exist for all nonprinting ASCII characters.
- Character escapes are also useless for representing characters beyond the basic 128 ASCII characters.
- Numeric escapes, which can represent any character, are the solution to this problem.
- A numeric escape for a particular character uses the character’s octal or hexadecimal value.
- For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex.

Escape Sequences

- An ***octal escape sequence*** consists of the `\` character followed by an octal number with at most three digits, such as `\33` or `\033`.
- A ***hexadecimal escape sequence*** consists of `\x` followed by a hexadecimal number, such as `\x1b` or `\x1B`.
- The `x` must be in lower case, but the hex digits can be upper or lower case.

Escape Sequences

- When used as a character constant, an escape sequence must be enclosed in single quotes.
- For example, a constant representing the escape character would be written `'\33'` (or `'\x1b'`).
- Escape sequences tend to get a bit cryptic, so it's often a good idea to use `#define` to give them names:

```
#define ESC '\33'
```

- Escape sequences can be embedded in strings as well.

Reading and Writing Characters Using `scanf` and `printf`

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);   /* reads one character */
printf("%c", ch);   /* writes one character */
```

- `scanf` doesn't skip white-space characters.
- To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:

```
scanf(" %c", &ch);
```

Reading and Writing Characters Using `scanf` and `printf`

- Since `scanf` doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character.
- A loop that reads and ignores all remaining characters in the current input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters Using `getchar` and `putchar`

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- `getchar` returns an `int` value rather than a `char` value, so `ch` will often have type `int`.
- Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Reading and Writing Characters Using `getchar` and `putchar`

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves execution time.
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats.
  - They are usually implemented as macros for additional speed.
- `getchar` has another advantage. Because it returns the character that it reads, `getchar` lends itself to various C idioms.

Reading and Writing Characters Using `getchar` and `putchar`

- Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
   scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
   ch = getchar();
} while (ch != '\n');
```

Reading and Writing Characters Using `getchar` and `putchar`

- Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
  ;
```

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character:

```
while (getchar() != '\n')
  ;
```

Reading and Writing Characters Using `getchar` and `putchar`

- `getchar` is useful in loops that skip characters as well as loops that search for characters.
- A statement that uses `getchar` to skip an indefinite number of blank characters:

```
while ((ch = getchar()) == ' ')
  ;
```

- When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Reading and Writing Characters Using `getchar` and `putchar`

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

  `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character.
- `getchar` will fetch the first leftover character.

Type Conversion

- For a an arithmetic operation, the operands must usually be of the same size.
- When operands of different types are mixed in expressions,
  - the C compiler may have to generate instructions that change the types of some operands.
  - If we add a 16-bit `short` and a 32-bit `int`, the compiler will arrange for the `short` value to be converted to 32 bits.
  - If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format.

The Usual Arithmetic Conversions

- The compiler handles these conversions automatically, known as ***implicit conversions.***
- C also allows the programmer to perform ***explicit conversions,*** using the cast operator.

- for expression `f + i`
  - convert `i` to type `float` (matching `f`'s type)
  - the worst that can happen is a minor loss of precision.
- Converting a floating-point number to `int` causes the fractional part of the number to be lost.

The Usual Arithmetic Conversions

- ***The type of either operand is a floating type.***
  - If one operand has type `long double`, then convert the other operand to type `long double`.
  - Otherwise, if one operand has type `double`, convert the other operand to type `double`.
  - Otherwise, if one operand has type `float`, convert the other operand to type `float`.
- Example: If one operand has type `long int` and the other has type `double`, the `long int` operand is converted to `double`.

The Usual Arithmetic Conversions

- Example of the usual arithmetic conversions:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;     /* c is converted to int                */
i = i + s;     /* s is converted to int                */
u = u + i;     /* i is converted to unsigned int       */
l = l + u;     /* u is converted to long int           */
ul = ul + l;   /* l is converted to unsigned long int */
f = f + ul;    /* ul is converted to float             */
d = d + f;     /* f is converted to double             */
ld = ld + d;   /* d is converted to long double        */
```

Conversion During Assignment

- The usual arithmetic conversions don't apply to assignment.
- Instead, the expression on the right side of the assignment is converted to the type of the variable on the left side:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int    */
f = i;   /* i is converted to float */
d = f;   /* f is converted to double */
```

Conversion During Assignment

- Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;      /* i is now 842 */
i = -842.97;     /* i is now -842 */
```

- Assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;      /*** WRONG ***/
i = 1.0e20;     /*** WRONG ***/
f = 1.0e100;    /*** WRONG ***/
```

Sign Extension for Type Conversion

- Converting from smaller to larger integer data type
- C automatically performs sign extension

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Casting

- Although C’s implicit conversions are convenient, we sometimes need a greater degree of control over type conversion.
- For this reason, C provides ***casts.***
- A cast expression has the form

  （ *type-name* ） *expression*

  type-name specifies the type to which the expression should be converted.

Casting

- Using a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;

frac_part = f - (int) f;
```

- The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.
- Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f;  /* f is converted to int */
```

Casting

- Cast expressions also let us force the compiler to perform conversions.
- Example:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

- To avoid truncation during division, we need to cast one of the operands:

```
quotient = (float) dividend / divisor;
```

- Casting `dividend` to `float` causes the compiler to convert `divisor` to `float` also.

Casting

- C regards (*type-name*) as a unary operator.
- Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

- Other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
quotient = (float) dividend / (float) divisor;
```

Casting

- Casts are sometimes necessary to avoid overflow:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

- Using a cast avoids the problem:

```
i = (long) j * j;
```

- The statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.

Type Definitions

- The `#define` directive can be used to create a "Boolean type" macro:

```
#define BOOL int
```

- There's a better way using a feature known as a ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names.
- Example:

```
Bool flag;    /* same as int flag; */
```

Advantages of Type Definitions

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

  ```
  typedef float Dollars;
  ```

  and then writing

  ```
  Dollars cash_in, cash_out;
  ```

  is more informative than just writing

  ```
  float cash_in, cash_out;
  ```

Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine `Dollars` as `double`, only the type definition need be changed:

```
typedef double Dollars;
```

- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

  ```
  i = 100000;
  ```

  is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

  ```
  typedef int Quantity;
  ```

  and use this type to declare variables:

  ```
  Quantity q;
  ```

- When we transport the program to a machine with shorter integers, we'll change the type definition:

  ```
  typedef long Quantity;
  ```

- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

- In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits.

The `sizeof` Operator

- The value of the expression

  `sizeof (` *type-name* `)`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name.*
- `sizeof(char)` is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, `sizeof(int)` is normally 4.

The `sizeof` Operator

- The `sizeof` operator can also be applied to constants, variables, and expressions in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.
- When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses.
  - We could write `sizeof i` instead of `sizeof(i)`.
- Parentheses may be needed anyway because of operator precedence.
  - The compiler interprets `sizeof i + j` as `(sizeof i) + j`, because `sizeof` takes precedence over binary `+`.