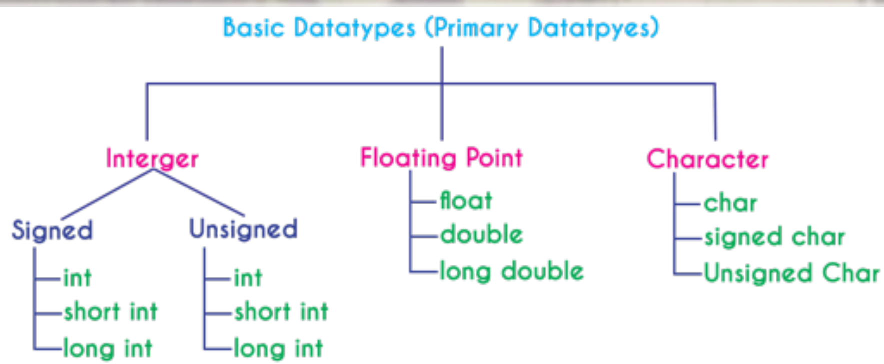
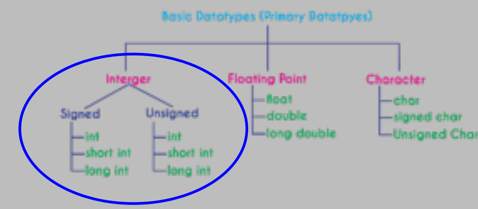
Lecture 4 - Basic Types

Meng-Hsun Tsai
CSIE, NCKU



4.1 Integer Types

Signed and Unsigned Integers



- The integer types are divided into two categories: signed and unsigned.
- 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.
  - By default, integer variables are signed in C.
- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) should be declared as `unsigned`.
  - Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications.

Long and Short Integers

- The `int` type is usually 32 bits, but may be 16 bits on other CPUs (e.g., Arduino or Raspberry pi).
- ***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`).

IMSLab Since 2010

Representing Negative Integers

- Modern computers use two's complement to represent negative integer values.
- Two's complement is obtained by adding one to one's complement, which simply inverts all bits.
- For example, an 8-bit integer 116 (= $0111\ 0100_2$) has one's complement $1000\ 1011_2$ and two's complement $1000\ 1100_2$.
- In the computer system, 116 is stored as $0111\ 0100_2$, -116 is stored as $1000\ 1100_2$.

2's complement

Decimal	116		-116
Binary	$0111\ 0100_2$	$1000\ 1011_2$	$1000\ 1100_2$

1's complement (inverting all bits)

Add 1

Range of Integer Values

- The range of values represented by each of the six integer types varies from one machine to another.
- However, the C standard requires that `short int`, `int`, and `long int` must each cover a certain minimum range of values.
- Also, `int` must not be shorter than `short int`, and `long int` must not be shorter than `int`.
- For an 8-bit integer with 2's complement, the largest positive value is 127 (= $1111\ 1111_2$), the largest negative value is -128 (= $1000\ 0000_2$).
- You can obtain the largest positive values and the largest negative values of 16-bit, 32-bit and 64-bit integers in the same way.

Integer limits defined in `limits.h`

- The `<limits.h>` header defines macros that represent the smallest and largest values of each integer type.

```
#define USHRT_MAX   0xffff          /* max value for unsigned short */
#define SHRT_MAX    0x7fff            /* max value for short */
#define SHRT_MIN    (-0x7fff - 1)   /* min value for short */

#define UINT_MAX    0xffffffff      /* max value for unsigned int */
#define INT_MAX     0x7fffffff        /* max value for an int */
#define INT_MIN     (-0x7fffffff - 1) /* min value for an int */

#define ULONG_MAX   0xffffffffffffffff  /* max for unsigned long */
#define LONG_MAX    0x7fffffffffffffff      /* max for long */
#define LONG_MIN    (-0x7fffffffffffffff - 1) /* min for long */
```

Integer limits defined in `limits.h` (cont.)

```
#define USHRT_MAX   0xffff          /* max value for unsigned short */
#define SHRT_MAX    0x7fff            /* max value for short */
#define SHRT_MIN    (-0x7fff - 1)   /* min value for short */
```

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("%d\n", USHRT_MAX);
    printf("%d\n", SHRT_MAX);
    printf("%d\n", SHRT_MIN);
    return 0;
}
```

Output:

```
65535
32767
-32768
```

Range of Integer Values (cont.)

- Typical ranges of values for the integer types on a 16-bit machine:

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

16-bit

16-bit

32-bit

Range of Integer Values (cont.)

- Typical ranges on a 32-bit machine:

Type	Smallest Value	Largest Value	
`short int`	–32,768	32,767	16-bit
`unsigned short int`	0	65,535	
`int`	–2,147,483,648	2,147,483,647	32-bit
`unsigned int`	0	4,294,967,295	
`long int`	–2,147,483,648	2,147,483,647	32-bit
`unsigned long int`	0	4,294,967,295	

Range of Integer Values (cont.)

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*	
`short int`	–32,768	32,767	16-bit
`unsigned short int`	0	65,535	
`int`	–2,147,483,648	2,147,483,647	32-bit
`unsigned int`	0	4,294,967,295	
`long int`	-2^{63}	$2^{63}-1$	64-bit
`unsigned long int`	0	$2^{64}-1$	

Integer Types in C99

- C99 provides two additional standard integer types, `long long int` and `unsigned long long int`.
- Both `long long` types are required to be at least 64 bits wide.
- The range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807).
- The range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

IMSLab Since 2010

Integer Constants

- ***Constants*** are numbers that appear in the text of a program.
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Integer Constants (cont.)

- ***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 (cont.)

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

```
15L   0377L   0x7fffL
```

- Integer constants that end with either `LL` or `ll` (the case of the two letters must match) have type `long long int`.
- To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

```
15U   0377U   0x7fffU
```

- `L` (or `LL`) and `U` may be used in combination:

```
0xffffffffUL 0xffffffffffffffffULL
```

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

Integer Overflow

- When arithmetic operations are performed on integers, it's possible that the result will be too large to represent. (we say that ***overflow*** has occurred)
- The behavior when integer overflow occurs depends on whether the operands were signed or unsigned.
  - When overflow occurs during an operation on *signed* integers, the program's behavior is **undefined**.
  - When overflow occurs during an operation on *unsigned* integers, the result *is* defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result.

IMSLab Since 2010

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 (cont.)

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

```
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`.

```
scanf("%ld", &l);
printf("%ld", l);
```

- When reading or writing a *long long* integer, put the letters `ll` in front of `d`, `o`, `u`, or `x`.

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

Program: Summing a Series of Numbers (Revisited)

- The `sum.c` program (Lecture 3) sums a series of integers.
- One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable.
- Here's what might happen if the program is run on a machine whose integers are 16 bits long:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

- When overflow occurs with signed numbers, the outcome is undefined.
- The program can be improved by using `long` variables.

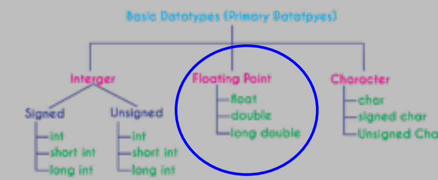
Program: Summing a Series of Numbers (Revisited) (cont.)

sum2.c

```
#include <stdio.h>

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

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

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

  return 0;
}
```

4.2 Floating Types

Floating Types

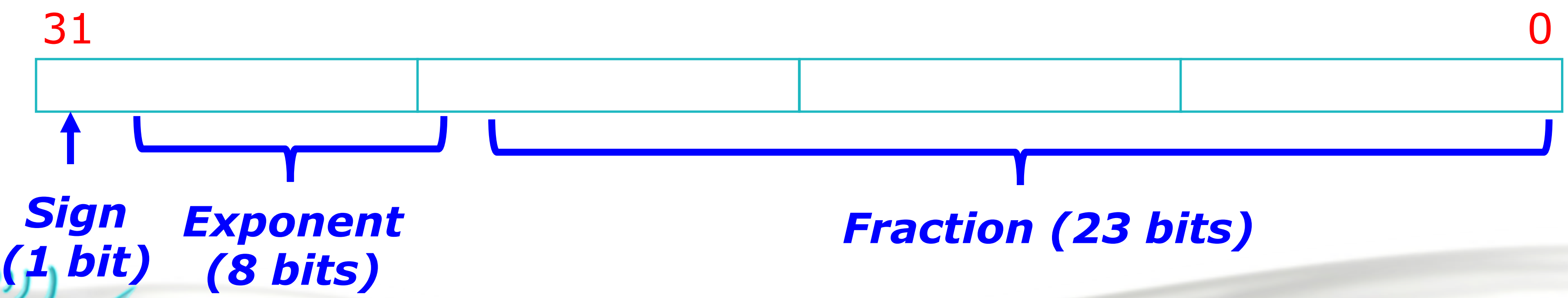


- C provides three ***floating types:*** `float`, `double` and `long double`
- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- The C standard doesn't state how much precision the `float`, `double`, 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).

IMSLab since 2010

The IEEE Floating-Point Standard

- IEEE Standard 754 has 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***.
- In single-precision format, the exponent is 8 bits long, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.



The IEEE 754-1985 Standard – Single Precision

- Sign: 0 for positive, 1 for negative
- Exponent: bias 127 (i.e., 01111111 means 0, 10000000 means 1)
  - All zeros and all ones are reserved for special cases

Exponent	Fraction	Meaning
00000000	Zero	0
00000000	Non-zero	Subnormal number (numbers close to zero) Exp. is -126, w/o leading bit
11111111	Zero	Infinity (+∞ and -∞)
11111111	Non-zero	NaN (Not a Number)
other	Any	Normal number (finite number) w/ leading bit

The IEEE 754-1985 Standard – Single Precision (cont.)

- For a normal number, there is a (invisible) leading bit in front of fraction part

1.

Sign	Exponent	Fraction

$\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$ … $\frac{1}{2^{23}}$

Number 1 $1.0_2 * 2^{(127-127)} = 1.0_{10} * 2^0 = 1$

0	0111 1111	0000000 00000000 00000000

Number 2 $1.0_2 * 2^{(128-127)} = 1.0_{10} * 2^1 = 2$

0	1000 0000	0000000 00000000 00000000

Number 1.5 $1.1_2 * 2^{(127-127)} = 1.5_{10} * 2^0 = 1.5$

0	0111 1111	1000000 00000000 00000000

Characteristics of Floating Types

- According to the IEEE standard 754:

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

- Macros that define the characteristics of the floating types can be found in the `<float.h>` header.

```
#define FLT_MANT_DIG    24              /* p */
#define FLT_DIG         6      /* floor((p-1)*log10(b))+(b == 10) */
#define FLT_MIN_EXP     (-125)            /* emin */
#define FLT_MIN         1.17549435E-38F /* b**(emin-1) */
#define FLT_MIN_10_EXP  (-37)           /* ceil(log10(b**(emin-1))) */
#define FLT_MAX_EXP     128               /* emax */
#define FLT_MAX         3.40282347E+38F /* (1-b**(-p))*b**emax */
#define FLT_MAX_10_EXP  38    /* floor(log10((1-b**(-p))*b**emax)) */
```

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 (cont.)

- 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

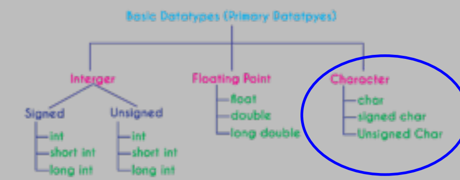
- The conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers.
- When reading a value of type `double`, put the letter `l` in front of `e`, `f`, or `g`:

```
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`.

4.3 Character Types

Character Types



- The only remaining basic type is `char`, the character type.
- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.
- Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.
- ASCII is often extended to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

Character Assignments

- 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.

IMSLab since 2010

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.

Operations on Characters

- Character constants actually have `int` type rather than `char` type.
- 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' */
```

IMSLab since 2010

Operations on Characters (cont.)

- 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 (cont.)

- 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.

IMSLab Since 2010

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;
```

Escape Sequences

- A character constant is usually one character enclosed in single quotes (ex. ‘a’).
- 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*** (enclosed in single quotes) provide a way to represent these characters.
- There are two kinds of escape sequences: ***character escapes*** and ***numeric escapes.***
- Escape sequences can be embedded in strings as well.

Character Escapes

Numeric escapes

- Character escapes are handy, but they don't exist for all nonprinting 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.

Name	Char	Oct	Hex	Dec
Alert (bell)	`\a`	\7	\x07	7
Backspace	`\b`	\10	\x08	8
Form feed	`\f`	\14	\x0c	12
New line	`\n`	\12	\x0a	10
Carriage return	`\r`	\15	\x0d	13
Horizontal tab	`\t`	\11	\x09	9
Vertical tab	`\v`	\13	\x0b	11
Backslash	`\\`	\134	\x27	92
Question mark	`\?`	\77	\x22	63
Single quote	`\'`	\47	\x5c	39
Double quote	`\"`	\42	\x3f	34

Rarely used

Numeric Escapes

- An ***octal escape sequence*** consists of the `\` character followed by an octal number with at most three digits, such as `\33` or `\033` (with normally maximum value \377).
- A ***hexadecimal escape sequence*** consists of `\x` followed by a hexadecimal number, such as `\x1b` or `\x1B` (with normally maximum value \xFF)..
- The `x` must be in lower case, but the hex digits can be upper or lower case.
- 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'
```

Character-Handling Functions

- Calling C's `toupper` library function is a fast and portable way to convert case:

```
ch = toupper(ch);
```

- `toupper` returns the upper-case version of its argument.
- Programs that call `toupper` need to have the following `#include` directive at the top:

```
#include <ctype.h>
```

- The C library provides many other useful character-handling functions.

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` (cont.)

- 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` (cont.)

- 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.

IMSLab Since 2010

Reading and Writing Characters Using **getchar** and **putchar** (cont.)

- Consider the `scanf` loop that we used to skip the rest of an input line. We can rewriting this loop using `getchar` gives us the following:

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

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

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

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

(when the `ch` variable isn't even needed)

Reading and Writing Characters Using `getchar` and `putchar` (cont.)

- `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` (cont.)

- 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:

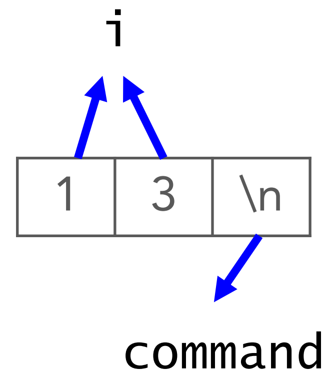
```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

  `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.

Reading and Writing Characters Using **getchar** and **putchar** (cont.)

```c
#include <stdio.h>

int main()
{
    int i, command;
    printf("Enter an integer: ");
    scanf("%d", &i);

    printf("Enter a command: ");
    command = getchar();

    printf("i is %d\n", i);
    printf("command is %x\n", command);

    return 0;
}
```




```
Enter an integer: 13
Enter a command: i is 13
command is  a
```

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`.

Program: Determining the Length of a Message (cont.)

length.c

```
#include <stdio.h>
int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d
  character(s) long.\n", len);
  return 0;
}
```

length2.c

```
#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d
  character(s) long.\n", len);

  return 0;
}
```

```
Enter a message: Hello World
Your message was 11 character(s) long.
```

4.4 Type Conversion

Type Conversion

- For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way.
- When operands of different types are mixed in expressions, the C compiler may have to generate instructions that change the types of some operands so that hardware will be able to evaluate the expression.
  - 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.

Type Conversion (cont.)

- Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions*.**
- C also allows the programmer to perform ***explicit conversions*,** using the cast operator.
- The rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Type Conversion (cont.)

- Implicit conversions are performed:
  - When the operands in an arithmetic or logical expression don't have the same type. (C performs what are known as the ***usual arithmetic conversions.***)
  - When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
  - When the type of an argument in a function call doesn't match the type of the corresponding parameter.
  - When the type of the expression in a `return` statement doesn't match the function's return type.

Lecture 5

IMSLab Since 2010

The Usual Arithmetic Conversions

- The usual arithmetic conversions are applied to the operands of most binary operators.
- If `f` has type `float` and `i` has type `int`, in the expression `f + i`, clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type).
- When an integer is converted to `float`, the worst that can happen is a minor loss of precision.
- Converting a floating-point number to `int`, on the other hand, causes the fractional part of the number to be lost.

3.8 -> 3

- Worse still, the result will be meaningless if the original number is larger than the largest possible integer or smaller than the smallest integer.

ex. 10^{11}

Program: Converting integer to float

```c
#include <stdio.h>
int main()
{
    int x = 0x1ffffff;
    float y;
    y = x;
    printf("x = %d\n", x);
    printf("y = %f\n", y);

    return 0;
}
```

25 bits of value 1

1 leading bit +
23 fraction bits

```
x = 33554431
y = 33554432.000000
```

The Usual Arithmetic Conversions (cont.)

- Strategy behind the usual arithmetic conversions: convert operands to the “narrowest” type that will safely accommodate both values.
- Converting the operand of a narrower type to the type of the other operand is known as ***promotion***.
- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.

IMSLab Since 2010

The Usual Arithmetic Conversions (cont.)

- ***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`.

```
long int + double
    ↓
double   + double
```

The Usual Arithmetic Conversions (cont.)

- ***Neither* operand type is a floating type.** First perform integral promotion on both operands.
- Then use the following diagram to promote the operand whose type is narrower:

```
unsigned long int
        ↑
    long int
        ↑
  unsigned int
        ↑
       int
```

The Usual Arithmetic Conversions (cont.)

- When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value.
- This rule can cause obscure programming errors.
- It’s best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

The Usual Arithmetic Conversions (cont.)

- 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        */
```

```
unsigned long int
        ↑
     long int
        ↑
   unsigned int
        ↑
       int
```

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 (cont.)

- 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 ***/
```

> 255

> 2147483647

> 3.40282×10^{38}

IMSLab Since 2010

Conversion During Assignment (cont.)

- It's a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable:

```
f = 3.14159f;
```

- Without the suffix, the constant `3.14159` would have type `double`, possibly causing a warning message.

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 (cont.)

- 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 (cont.)

- 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 (cont.)

- 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 (cont.)

- 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.

4.5 Type Definitions

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; */
```

- C99 has defined `_Bool` as an unsigned integer with only values 1 (true) and 0 (false).

```
_Bool flag = 5 > 3;
```

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 (cont.)

- 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 (cont.)

- For greater portability, consider using `typedef` to define new names for integer types.
- Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000.
- We could use `long` variables for this purpose, but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values. Also, `int` variables may take up less space.

Type Definitions and Portability (cont.)

- 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 (cont.)

- 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;
```

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

ex. `int32_t;`

IMSLab Since 2010

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 (cont.)

- 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 `+`.

The `sizeof` Operator (cont.)

- Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named `size_t`.
- In C89, it's best to convert the value of the expression to a known type before printing it:

```
printf("Size of int: %lu\n",
       (unsigned long) sizeof(int));
```

- The `printf` function in C99 can display a `size_t` value directly if the letter `z` is included in the conversion specification:

```
printf("Size of int: %zu\n", sizeof(int));
```

4.6 Bitwise Operation

Bitwise Operation

- Some kinds of programs need to perform operations at the bit level:
  - Systems programs (including compilers and operating systems)
  - Encryption programs
  - Graphics programs
  - Programs for which fast execution and/or efficient use of space is critical

Bitwise Operators

- C provides six ***bitwise operators,*** which operate on integer data at the bit level.
- Two of these operators perform shift operations.
- The other four perform bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or* operations.

- & → AND
- | → OR
- ~ → NOT
- << → Left shift
- >> → Right shift
- ^ → XOR

Bitwise Shift Operators

- The bitwise shift operators shift the bits in an integer to the left or right:

  << left shift

  >> right shift

- The operands for << and >> may be of any integer type (including `char`).

Bitwise Shift Operators (cont.)

- The value of `i << j` is the result when the bits in `i` are shifted left by `j` places.
  - For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right.
- The value of `i >> j` is the result when `i` is shifted right by `j` places.
  - If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed.
  - If `i` is negative, the result is implementation-defined.

Bitwise Shift Operators (cont.)

- Examples illustrating the effect of applying the shift operators to the number 13:

```
unsigned short i, j;

i = 13;
  /* i is now 13 (binary 0000000000001101) */
j = i << 2;
  /* j is now 52 (binary 0000000000110100) */
j = i >> 2;
  /* j is now  3 (binary 0000000000000011) */
```

Bitwise Shift Operators (cont.)

- To modify a variable by shifting its bits, use the compound assignment operators <<= and >>=:

```
i = 13;
  /* i is now 13 (binary 0000000000001101) */
i <<= 2;
  /* i is now 52 (binary 0000000000110100) */
i >>= 2;
  /* i is now 13 (binary 0000000000001101) */
```

- The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises:

`i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- There are four additional bitwise operators:
  - ~ bitwise complement
  - & bitwise *and*
  - ^ bitwise exclusive *or*
  - | bitwise inclusive *or*

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

- The ~ operator is unary; the other operators are binary.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or* (cont.)

- Examples of the `~`, `&`, `^`, and `|` operators:

```
unsigned short i, j, k;
i = 21;
  /* i is now    21 (binary 0000000000010101) */
j = 56;
  /* j is now    56 (binary 0000000000111000) */
k = ~i;
  /* k is now 65514 (binary 1111111111101010) */
k = i & j;
  /* k is now    16 (binary 0000000000010000) */
k = i ^ j;
  /* k is now    45 (binary 0000000000101101) */
k = i | j;
  /* k is now    61 (binary 0000000000111101) */
```

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or* (cont.)

- The `~` operator can be used to help make low-level programs more portable.
  - An integer whose bits are all 1: `~0`
  - An integer whose bits are all 1 except for the last five: `~0x1f`

`0`	00000000 00000000
`~0`	11111111 11111111

`0x1f`	00000000 00011111
`~0x1f`	11111111 11100000

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or* (cont.)

- Each of the `~`, `&`, `^`, and `|` operators has a different precedence:

  Highest: `~`

  `&`

  `^`

  Lowest: `|`

- Examples:

  `i & ~j | k` means `(i & (~j)) | k`

  `i ^ j & ~k` means `i ^ (j & (~k))`

- Using parentheses helps avoid confusion.

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or* (cont.)

- The compound assignment operators `&=`, `^=`, and `|=` correspond to the bitwise operators `&`, `^`, and `|`:

```
i = 21;
  /* i is now 21 (binary 0000000000010101) */
j = 56;
  /* j is now 56 (binary 0000000000111000) */
i &= j;
  /* i is now 16 (binary 0000000000010000) */
i ^= j;
  /* i is now 40 (binary 0000000000101000) */
i |= j;
  /* i is now 56 (binary 0000000000111000) */
```

Using the Bitwise Operators to Access Bits

- The bitwise operators can be used to extract or modify data stored in a small number of bits.
- Common single-bit operations:
  - Setting a bit
  - Clearing a bit
  - Testing a bit
- Assumptions:
  - `i` is a 16-bit `unsigned short` variable.
  - The leftmost—or ***most significant***—bit is numbered 15 and the least significant is numbered 0.

00000000 00111000

15 0

Using the Bitwise Operators to Access Bits (cont.)

- ***Setting a bit.*** The easiest way to set bit 4 of `i` is *to or* the value of `i` with the constant `0x0010`:

```
i = 0x0000;
  /* i is now 0000000000000000 */

i |= 0x0010;
  /* i is now 0000000000010000 */
```

- If the position of the bit is stored in the variable `j`, a shift operator can be used to create the mask:

```
i |= 1 << j;          /* sets bit j */
```

- Example: If `j` has the value 3, then `1 << j` is `0x0008`.

```
00000000 00001000
15              j 0
```

Using the Bitwise Operators to Access Bits (cont.)

- ***Clearing a bit.*** Clearing bit 4 of `i` requires a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;
   /* i is now 0000000011111111 */

i &= ~0x0010;
   /* i is now 0000000011101111 */
```

- A statement that clears a bit whose position is stored in a variable:

```
i &= ~(1 << j);      /* clears bit j */
```

```
1 << j   | 00000000 00001000 |       ~(1 << j) | 11111111 11110111 |
          15              j  0                  15              j  0
```

Using the Bitwise Operators to Access Bits (cont.)

- ***Testing a bit.*** An `if` statement that tests whether bit 4 of `i` is set:

```
if (i & 0x0010) …    /* tests bit 4 */
```

- A statement that tests whether bit `j` is set:

```
if (i & 1 << j) …    /* tests bit j */
```

```
00000000 00001000
15           j  0
```

Using the Bitwise Operators to Access Bits (cont.)

- Working with bits is easier if they are given names.
- Suppose that bits 0, 1, and 2 of a number correspond to the colors blue, green, and red, respectively.
- Names that represent the three bit positions:

```
#define BLUE   1        00000000 00000001
#define GREEN  2        00000000 00000010
#define RED    4        00000000 00000100
```

- Examples of setting, clearing, and testing the `BLUE` bit:

```
i |= BLUE;          /* sets BLUE bit    */
i &= ~BLUE;         /* clears BLUE bit  */
if (i & BLUE) …     /* tests BLUE bit   */
```

Using the Bitwise Operators to Access Bits (cont.)

- It's also easy to set, clear, or test several bits at a time:

```
i |= BLUE | GREEN;
  /* sets BLUE and GREEN bits   */
i &= ~(BLUE | GREEN);
  /* clears BLUE and GREEN bits */
if (i & (BLUE | GREEN)) …
  /* tests BLUE and GREEN bits  */
```

BLUE | GREEN

00000000 00000011

- The `if` statement tests whether either the `BLUE` bit or the `GREEN` bit is set.

Using the Bitwise Operators to Access Bit-Fields

- Dealing with a group of several consecutive bits (a ***bit-field***) is slightly more complicated than working with single bits.
- Common bit-field operations:
  - Modifying a bit-field
  - Retrieving a bit-field

Using the Bitwise Operators to Access Bit-Fields (cont.)

- ***Modifying a bit-field.*** Modifying a bit-field requires two operations:
  - A bitwise *and* (to clear the bit-field)
  - A bitwise *or* (to store new bits in the bit-field)
- Example:

```
i = i & ~0x0070 | 0x0050;
  /* stores 101 in bits 4-6 */
```

~0x0070	11111111 10001111
0x0050	00000000 01010000

- The `&` operator clears bits 4–6 of `i`; the `|` operator then sets bits 6 and 4.

Using the Bitwise Operators to Access Bit-Fields (cont.)

- To generalize the example, assume that `j` contains the value to be stored in bits 4–6 of `i`.
- `j` will need to be shifted into position before the bitwise *or* is performed:

```
i = (i & ~0x0070) | (j << 4);
  /* stores j in bits 4-6 */
```

`j`	00000000 00000**101**
`j << 4`	00000000 0**101**0000

Using the Bitwise Operators to Access Bit-Fields (cont.)

- ***Retrieving a bit-field.*** Fetching a bit-field at the right end of a number (in the least significant bits) is easy:

```
j = i & 0x0007;
  /* retrieves bits 0-2 */
```

0x0007	00000000 00000**111**

- If the bit-field isn't at the right end of `i`, we can first shift the bit-field to the end before extracting the field using the `&` operator:

```
j = (i >> 4) & 0x0007;
  /* retrieves bits 4-6 */
```

i	???????? ?**XXX**????
i >> 4	???????? ?????**XXX**

Program: XOR Encryption

- One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key.
- Suppose that the key is the `&` character.
- XORing this key with the character `z` yields the `\` character:

```
    00100110  (ASCII code for &)
XOR 01111010  (ASCII code for z)
    --------
    01011100  (ASCII code for \)
```

Program: XOR Encryption (cont.)

- Decrypting a message is done by applying the same algorithm:

```
        00100110  (ASCII code for &)
    XOR 01011100  (ASCII code for \)
        --------
        01111010  (ASCII code for z)
```

Program: XOR Encryption (cont.)

- The `xor.c` program encrypts a message by XORing each character with the `&` character.
- The original message can be entered by the user or read from a file using input redirection.
- The encrypted message can be viewed on the screen or saved in a file using output redirection.
- A sample file named `msg`:

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
                    --Johann Kaspar Lavater (1741-1801)
```

Program: XOR Encryption (cont.)

- A command that encrypts `msg`, saving the encrypted message in `newmsg`:

```
xor < msg > newmsg
```

- Contents of `newmsg`:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
            --lINGHH mGUVGT jGPGRCT (1741-1801)
```

- A command that recovers the original message and displays it on the screen:

```
xor < newmsg
```

Program: XOR Encryption (cont.)

- The `xor.c` program won't change some characters, including digits.
- XORing these characters with `&` would produce invisible control characters, which could cause problems with some operating systems.
- The program checks whether both the original character and the new (encrypted) character are printing characters.
- If not, the program will write the original character instead of the new character.

Program: XOR Encryption (cont.)

xor.c

```
#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }

  return 0;
}
```

A Quick Review to This Lecture

- Integer types: (all can be `signed` or `unsigned`)
  `short int / int / long int / long long int`
- Floating types: (usually IEEE Standard 754)
  `float / double / long double`
- Character types:
  `char / signed char / unsigned char`
- Boolean type (C99): (`unsigned`, `0` or `1`)
  `_Bool`

A Quick Review to This Lecture (cont.)

- Types of integer constants
  `L` or `l` for `long int` (ex. `15L`)
  `U` or `u` for `unsigned int` (ex. `15U` `15uL`)
  `LL` or `ll` for `long long int` (ex. `15LL` `15Ull`)

- Different bases for integer
  begin with non-zero: decimal (ex. `19`)
  begin with zero: octal (ex. `017`)
  begin with `0x` or `0X`: hexadecimal (ex. `0xff` `0XFf`)

- Integer Overflow
  `signed` integer: undefined behavior
  `unsigned` integer: correct answer modulo 2^n (n: number of bits)

A Quick Review to This Lecture (cont.)

- Reading and writing integers
  `%h` for `short`
  `%l` for `long`
  `%ll` for `long long`

  `%d` for `signed int`
  `%u, %o, %x` for `unsigned int (base 10, 8, 16)`
  `(ex. printf("%ld", var); //signed long (base 10))`

- Floating constants (must contain a decimal point and/or an exponent)
  `57.0` can be expressed as
  `57.0  57.  57.0e0 57E0 5.7e1  5.7e+1 .57e2 570.e-1`
  (`e` for power of `10`)

A Quick Review to This Lecture (cont.)

- Types of floating constants
  `F` or `f` for `float` (ex. `57.0F`)
  (default) for `double` (ex. `57.0`)
  `L` or `l` for `long double` (ex. `57.0L 57.0l`)

- Reading and writing floating-point numbers
  - `scanf`
    `%e, %f, %g` for `float`
    `%le, %lf, %lg` for `double`
    `%Le, %Lf, %Lg` for `long double`
  - `printf`
    `%e, %f, %g` for `float` or `double`
    `%Le, %Lf, %Lg` for `long double`

A Quick Review to This Lecture (cont.)

- Character assignment
  ```
  char ch = 'a'  // single quote, not double quote
  ```
- C treats characters as small integers (mostly ASCII / Latin-1 codes)
- Character constants have `int` type rather than `char` type
- Converting a lower-case letter to upper case
  ```
  1. if ('a' <= ch && ch <= 'z')  // ASCII assumed
         ch = ch - 'a' + 'A';
  2. #include <ctype.h>
     ch = toupper(ch);   // character set independent
  ```
- C allows the use of the words signed and unsigned to modify char:
  ```
  signed char sch;
  unsigned char uch;
  ```

A Quick Review to This Lecture (cont.)

- Character Escapes
- Using macro to define non-printable characters

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

Name	Char	Oct	Hex	Dec
Alert (bell)	`\a`	\7	\x07	7
Backspace	`\b`	\10	\x08	8
Form feed	`\f`	\14	\x0c	12
New line	`\n`	\12	\x0a	10
Carriage return	`\r`	\15	\x0d	13
Horizontal tab	`\t`	\11	\x09	9
Vertical tab	`\v`	\13	\x0b	11
Backslash	`\\`	\134	\x27	92
Question mark	`\?`	\77	\x22	63
Single quote	`\'`	\47	\x5c	39
Double quote	`\"`	\42	\x3f	34

Rarely used

A Quick Review to This Lecture (cont.)

- Reading and writing characters
  1. `printf()` and `scanf()`: using `%c`
  2. `putchar(ch);        //` faster
     `ch = getchar();   //` return `int`, faster
- `scanf` and `getchar` do not skip white-space characters.
- To force `scanf` to skip white space before reading a character:
  `scanf(" %c", &ch);`
  space
- Be careful when mixing `getchar` and `scanf`. `scanf` leaves behind characters that it has "peeked" at but not read,

A Quick Review to This Lecture (cont.)

- Ignoring all remaining characters in the current input line:

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

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

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

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

(when the `ch` variable isn't even needed)

- Using `getchar` to skip an indefinite number of blank characters:

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

A Quick Review to This Lecture (cont.)

- Implicit type conversion
  usual arithmetic conversion or assignment conversion
- Usual arithmetic conversion
  - The type of either operand is a floating type.
    (other type) -> `float` -> `double` -> `long double`
  - Neither operand type is a floating type.
    Integral promotion:
    `int` -> `unsigned int` -> `long int` -> `unsigned long int`
- Assignment conversion
  Right side is converted to left side

A Quick Review to This Lecture (cont.)

- Explicit type conversion (using cast operator)
```
( type-name ) expression
```
- Computing the fractional part of a `float` value:
```
float f, frac_part;
frac_part = f - (int) f;
```
- To avoid truncation during division, we need to cast one operand:
```
float quotient;
int dividend, divisor;
quotient = (float) dividend / divisor;
```
- Casts are sometimes necessary to avoid overflow:
```
long i;
int j = 1000;
i = (long) j * j;
```

A Quick Review to This Lecture (cont.)

- Type definition

  1. `#define BOOL int`
  2. `typedef int Bool;    //` better

- `<stdint.h>` header defines integer types with particular sizes

  `int32_t  i;`

- `sizeof` operator returns required bytes (`size_t`) to store a value

  `sizeof ( type-name )`

  `sizeof ( expression )` or `sizeof expression`

- Print returned value of `sizeof`

```
printf("Size of int: %lu\n",
          (unsigned long) sizeof(int));    // C89
printf("Size of int: %zu\n", sizeof(int));  // C99
```