Chp 7 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)

- C supports two fundamentally different kinds of numeric types:
  - **integer types**: whole numbers. Two categories: signed (`int`) and unsigned (`unsigned`).
  - **floating types**: may have a fractional part

Signed and Unsigned Integers

- *Signed* integer: left most bit is the *sign bit* -

  Sign bit is 0 if the number is positive or zero,

  Sign bit is 1 if it's negative.
- The largest 16-bit signed integer (`short`) has the binary representation 0111111111111111, which has the value 32,767 ($2^{15} - 1$).
- The largest 16-bit unsigned integer has the binary representation 1111111111111111, which has the value 65,535 ($2^{16} - 1$).

Number of bits

- *int* is usually 32 bits, but may be 16 bits on older CPUs.
- *long* is usually 32 or 64 bits;
  short is usually 16 bits.
  C99 has long long int and unsigned long long int, which are at least 64 bits.
- The <`limits.h`> header defines macros that represent the smallest and largest values of each integer type.
- The specifiers *long*, *short*, *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.

  int can be dropped (e.g. *long int* can be just *long*).

Integer Constants

- *Constants* are numbers that appear in the text of a program.
- C allows integer constants to be written in decimal, octal, or hexadecimal.
- ***Decimal*** constants contain digits between 0 and 9, but must not begin with a zero:
  E.g. 15 255 32767
- ***Octal*** constants contain only digits between 0 and 7, and must begin with a zero:
  E.g. 017 0377 077777
- ***Hexadecimal*** constants contain digits between 0 and 9 and letters between a and f, and always begin with 0x: E.g. 0xf 0xab 0x7ab
  The letters in a hexadecimal constant may be either upper or lower case: 0xF 0XAB 0x7AB
- The type of a decimal integer constant is normally *int*.

  If the value of the constant is too large to store as an *int*, the constant has type *long int* instead.

  If the constant is too large to store as a *long int*, the compiler will try *unsigned long int* as a last resort.
- For an octal or hexadecimal constant, the rules are slightly different: the compiler will go through the types *int*, *unsigned int*, *long int*, and *unsigned long int* until it finds one capable of representing the constant.
- To force the compiler to treat a constant as a long int, just follow it with the letter L (or lower case 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.
- In **C99**, integer constants that end with either LL or ll have type *long long int*.
- Adding the letter U (or u) before or after the LL or ll denotes a constant of *type unsigned long long int*.
- The type of a decimal constant with no suffix (U, u, L, l, LL, or ll) is the "smallest" of the types int, long int, or long long int that can represent the value of that constant.

- For an octal or hexadecimal constant without a suffix, the list of possible types is int, unsigned int, long int, unsigned long int, long long int, and unsigned long long int, in that order.
- **Overflow**: When arithmetic operations are performed on integers, it's possible that the result will be too large to represent.

  For example, when an arithmetic operation is performed on two int values, the result must be able to be represented as an int.

  If the result can't be represented as an int (because it requires too many bits), we say that ***overflow*** has occurred.
- The behavior when integer overflow occurs depends on whether the operands are *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.

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
```

- 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
- The C standard doesn't state how much precision the float, double, and long double types provide.

  Usually (but not always),
  - float is 32 bits,
  - double 64 bits,
  - long double is 128 bits .
- IEEE Standard 754 has two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits).
- Numbers are stored in scientific notation, with each number having a ***sign,*** an ***exponent,*** and a ***fraction.***
- In single-precision, the exponent is 8 bits, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.
- 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.
- Macros that define the characteristics of the floating types can be found in the <float.h> header.
- In C99, the floating types are divided into two categories.
  - ***Real floating types*** (float, double, long double)
  - ***Complex types*** (float _Complex, double _Complex, long double _Complex)

Floating Constants

- Floating constants can be written in a variety of ways.

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).
- 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 a constant 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* numbers.
- When *reading* a double, put the letter l in front of e, f, or g. Note*:* Use l only in a scanf format string, not a printf string.

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

- 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 Types

- 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.
- 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

- 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.
- 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' */
```

- Characters can be compared, just as numbers can.

  Example: 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.
- 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:

- ***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.

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)
- C99 has a more complicated hierarchy:
  - Integer types
    - char
    - Signed integer types, both standard (signed char, short int, int, long int, long long int) and extended
    - Unsigned integer types, both standard (unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int, _Bool) and extended
    - Enumerated types
  - Floating types
    - Real floating types (float, double, long double)
    - Complex types (float _Complex, double _Complex, long double _Complex)

Escape Sequences

- A character constant is usually a 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***.
- 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	\"

- 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. For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex.

  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.

- 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').

- It's a good idea to use #define to name an escape:

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

Read/Write Characters -- scanf and printf

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

```
char ch;

scanf("%c", &ch); //read a character
printf("%c", ch); //write a 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);
```

- Recall that we used scanf for reading numbers --- scanf does skip white-space when reading numbers.

- 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.

  Example: A loop that reads and ignores all remaining characters in the current input line:

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

Read/Write Characters -- getchar, putchar

- getchar and putchar are for single-character input and output.
  putchar writes a character:
  `putchar(ch);`
  getchar reads one character,:
  `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.

- 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.

Some code examples:

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

An equivalent code segment:

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

Another equivalent code segment:

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

Another equivalent code segment:

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

- 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.

Type Conversion

- For an arithmetic operation, operands must be of the same type.
  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 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 add an int and a float, the compiler will arrange for the int to be converted to float format.

- Conversions handled by the compiler automatically, without the programmer's involvement, are known as ***implicit conversions.***
  C also allows the programmer to perform ***explicit conversions,*** using the cast operator.

- Implicit conversions are performed:
  - When the operands in an arithmetic or logical expression don't have the same type.

– 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.

The Usual Arithmetic Conversions

- Operand types can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***).
- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - One of operands is a floating type.

    If one operand has type long double, then convert the other operand to type long double.

    else if one operand has type double, convert the other operand to type double.

    else if one operand has type float, convert the other operand to type float.

  - Neither operand type is a floating type.

    Integral promotions --- convert a character or short integer to type int (or to unsigned int in some cases).

    Promote the operand whose type is narrower:

    int—> unsigned int—> long —>unsigned long
- 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:
- 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. Fr example,

```
char   ch;
ch = 1000 ;
```

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

Examples:

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

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

To avoid overflow:

```
long i;
short  j = 1000;
i = (long) j * j;
i = (long) (j * j); /** overflow **/
```

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 flag;   /* same as int flag; */

typedef float Dollars;
Dollars cash_in, cash_out;
```

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 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 +.

Chp 8. Arrays

Scalar Variables vs. Aggregate Variables

- ***scalar*** variable***:*** holds a single data item.

  aggregate variable: can store collections of values.
- Two kinds of aggregates in C: arrays and structures.

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array a are conceptually arranged one after another:

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

```
int a[10];
```

- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

- To access an array element, write the array name followed by an integer value in square brackets. If a is an array of length 10, its elements are designated by a[0], a[1], …, a[9].
- This is referred to as ***subscripting*** or ***indexing*** the array. The elements of an array of length *n* are indexed from 0 to *n* – 1.
- Many programs contain for loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array a of length N:

```
for(i = 0; i < N; i++)
   a[i] = 0;

for(i = 0; i < N; i++)
  scanf("%d", &a[i]);//reads data into a

for(i = 0; i < N; i++)
  sum += a[i]; //sums the elements of a
```

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;
for (i = 1; i <= 10; i++)
   a[i] = 0;
```

- An array subscript may be any integer expression:

  Example: `a[i+j*10] = 0;`

  Another example:

```
i = 0;
while (i < N)
  a[i++] = 0;
```

- Be careful when an array subscript has a side effect:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- The expression a[i] = b[i++] accesses the value of i and also modifies i, causing undefined behavior.
- The problem can be avoided by removing the increment from the subscript:

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```

Array Initialization

- An array, like any other variable, can be given an initial value at the time it's declared.
- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[6] = {1, 2, 3, 4, 5, 6};
```

- If initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[9] = {1, 2, 3, 4, 5, 6};
```

- initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0}
- Using this feature, we can easily initialize an array to all zeros:

```
int a[5] = {0};
// initial value of a is {0, 0, 0, 0, 0}
```

- It's illegal for an initializer to be completely empty.

```
int a[5] = {}; // Illegal
```

- It's also illegal for an initializer to be longer than the array it initializes.
- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

- The compiler uses the length of the initializer to determine how long the array is.

Designated Initializers (C99)

- It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values.

  An example:

```
int a[15] =
    {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48};
```

  For a large array, writing an initializer in this fashion is tedious and error-prone.

- Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2]=29, [9]=7, [14]=48};
```

  Each number in brackets is said to be a ***designator.***
- Also, the order in which the elements are listed no longer matters. E.g.

```
int a[15] = {[14]=48, [9]=7, [2]=29};
```

- If the length of the array is omitted, a designator can be any nonnegative integer.

  The compiler will deduce the length of the array.

  The following array will have 24 elements:

```
int b[]={[5]=10,[23]=13,[11]=36,[15]=29};
```

- An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int a[6]={[1]=v1, v2, [4]=v4};
```

  is equivalent to

```
int a[6]={0, v1, v2, 0, v4, 0};
```