01. Variables and Types

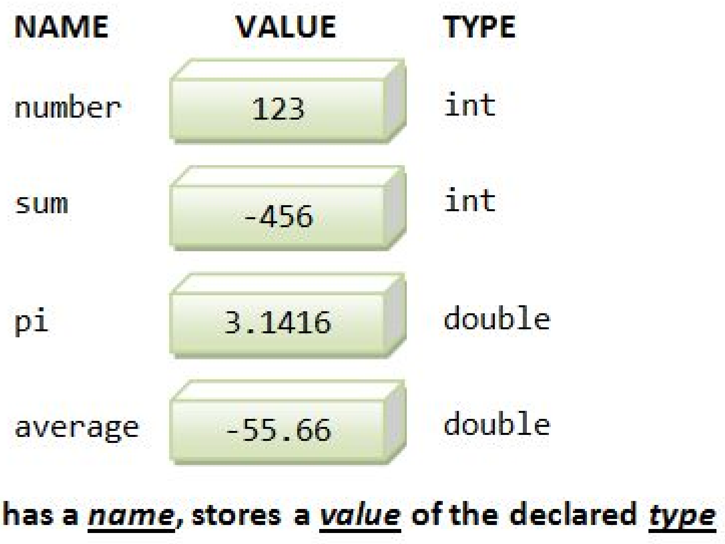
01.1 Variables

Computer programs manipulate (or process) data. A *variable* is used to *store a piece of data* for processing. It is called *variable* because you can change the value stored.

More precisely, a *variable* is a *named* storage location, that stores a *value* of a particular data *type*. In other words, a *variable* has a *name*, a *type* and stores a *value*.

- A variable has a *name* (or *identifier*), e.g., `radius`, `area`, `age`, `height`. The name is needed to uniquely identify each variable, so as to assign a value to the variable (e.g., `radius=1.2`), and retrieve the value stored (e.g., `area = radius*radius*3.1416`).
- A variable has a *type*. Examples of *type* are,
  - `int`: for integers (whole numbers) such as `123` and `-456`;
  - `double`: for floating-point or real numbers such as `3.1416`, `-55.66`, having a decimal point and fractional part.
- A variable can store a *value* of that particular *type*. It is important to take note that a variable in most programming languages is associated with a type, and can only store value of the particular type. For example, a `int` variable can store an integer value such as `123`, but NOT real number such as `12.34`, nor texts such as `"Hello"`.
- The concept of *type* was introduced into the early programming languages to simplify interpretation of data made up of 0s and 1s. The type determines the size and layout of the data, the range of its values, and the set of operations that can be applied.

The following diagram illustrates two types of variables: `int` and `double`. An `int` variable stores an integer (whole number). A `double` variable stores a real number.



01.2 Identifiers

An *identifier* is needed to *name* a variable (or any other entity such as a function or a class). C imposes the following *rules on identifiers*:

- An identifier is a sequence of characters, of up to a certain length (compiler-dependent, typically 255 characters), comprising uppercase and lowercase letters `(a-z, A-Z)`, digits `(0-9)`, and underscore `"_"`.
- Whitespace (blank, tab, new-line) and other special characters (such as `+`, `-`, `*`, `/`, `@`, `&`, commas, etc.) are not allowed.

- An identifier must begin with a letter or underscore. It cannot begin with a digit. Identifiers beginning with an underscore are typically reserved for system use.
- An identifier cannot be a reserved keyword or a reserved literal (e.g.,`int`, `double`, `if`, `else`, `for`).
- Identifiers are case-sensitive. A `rose` is NOT a `Rose`, and is NOT a `ROSE`.

Caution: Programmers don't use *blank* character in names. It is either not supported, or will pose you more challenges.

Variable Naming Convention

A variable name is a noun, or a noun phrase made up of several words. The first word is in lowercase, while the remaining words are initial-capitalized, with no spaces between words. For example, `thefontSize`, `roomNumber`, `xMax`, `yMin`, `xTopLeft` and `thisIsAVeryLongVariableName`. This convention is also known as *camel-case*.

Recommendations

1. It is important to choose a name that is *self-descriptive* and closely reflects the meaning of the variable, e.g., `numberOfStudents` or `numStudents`.
2. Do not use meaningless names like `a`, `b`, `c`, `d`, `i`, `j`, `k`, `i1`, `j99`.
3. Avoid single-alphabet names, which is easier to type but often meaningless, unless they are common names like `x`, `y`, `z` for coordinates, `i` for index.
4. It is perfectly okay to use long names of says 30 characters to make sure that the name accurately reflects its meaning!
5. Use singular and plural nouns prudently to differentiate between singular and plural variables. For example, you may use the variable `row` to refer to a single row number and the variable `rows` to refer to many rows (such as an array of rows - to be discussed later).

01.3 Variable Declaration

To use a variable in your program, you need to first "introduce" it by *declaring* its *name* and *type*, in one of the following syntaxes:

Syntax	Example

`// Declare a variable of a specified type`
`type identifier;`
	`int option;`

`// Declare multiple variables of the same type, separated by commas`
`type identifier-1, identifier-2, ..., identifier-n;`
	`double sum, difference, product, quotient;`

`// Declare a variable and assign an initial value`
`type identifier = value;`
	`int magicNumber = 88;`

`// Declare multiple variables with initial values`
`type identifier-1 = value-1, ..., identifier-n = value-n;`
	`double sum = 0.0, product = 1.0;`

Example,

```
int mark1;            // Declare an int variable called mark1
mark1 = 76;           // Use mark1
int mark2;            // Declare int variable mark2
mark2 = mark1 + 10;   // Use mark2 and mark1
```

```
double average;        // Declare double variable average
average = (mark1 + mark2) / 2.0;   // Use average, mark1 and mark2
int mark1;             // Error: Declare twice
mark2 = "Hello";       // Error: Assign value of a different type
```

Take note that:

- In C, you need to declare the name of a variable before it can be used.
- C is a "strongly-type" language. A variable takes on a type. Once the *type* of a variable is declared, it can only store a value belonging to this particular type. For example, an `int` variable can hold only integer such as `123`, and NOT floating-point number such as `-2.17` or text string such as `"Hello"`. The concept of *type* was introduced into the early programming languages to simplify interpretation of data made up of 0s and 1s. Knowing the *type* of a piece of data greatly simplifies its interpretation and processing.
- Each variable can only be declared once.
- In C, you can declare a variable anywhere inside the program, as long as it is declared before used. (In C prior to C99, all the variables must be declared at the beginning of functions.) It is recommended that your declare a variable just before it is first used.
- The type of a variable cannot be changed inside the program.

CAUTION: Uninitialized Variables

When a variable is declared, it contains garbage until you assign an initial value. It is important to take note that C does not issue any warning/error if you use a variable before initialize it - which certainly leads to some unexpected results. For example,

```
#include <stdio.h>

int main() {
  int number;                // Declared but not initialized
  printf("%d\n", number); // Used before initialized
                             // No warning/error, BUT unexpected result
  return 0;
}
```

01.4 Constants (const)

Constants are *non-modifiable* variables, declared with keyword `const`. Their values cannot be changed during program execution. Also, `const` must be initialized during declaration. For examples:

```
const double PI = 3.1415926;  // Need to initialize
```

Constant Naming Convention: Use uppercase words, joined with underscore. For example, `MIN_VALUE`, `MAX_SIZE`.

01.5 Expressions

An *expression* is a combination of *operators* (such as addition `'+'`, subtraction `'-'`, multiplication `'*'`, division `'/'`) and *operands* (variables or literal values), that can be *evaluated to yield a single value of a certain type*. For example,

```
1 + 2 * 3             // give int 7

int sum, number;
```

```
sum + number         // evaluated to an int value

double principal, interestRate;
principal * (1 + interestRate)  // evaluated to a double value
```

01.6 Assignment (=)

An *assignment statement*:

1. assigns a literal value (of the RHS) to a variable (of the LHS); or
2. evaluates an expression (of the RHS) and assign the resultant value to a variable (of the LHS).

The RHS shall be a value; and the LHS shall be a variable (or memory address).

The syntax for assignment statement is:

Syntax	Example

`// Assign the literal value (of the RHS) to the variable (of the LHS)`
`variable = literal-value;`
	`number = 88;`

`// Evaluate the expression (RHS) and assign the result to the variable (LHS)`
`variable = expression;`
	`sum = sum + number;`

The assignment statement should be interpreted this way: The *expression* on the right-hand-side (RHS) is first evaluated to produce a resultant value (called *rvalue* or right-value). The *rvalue* is then assigned to the variable on the left-hand-side (LHS) (or *lvalue*, which is a location that can hold a *rvalue*). Take note that you have to first evaluate the RHS, before assigning the resultant value to the LHS. For examples,

```
number = 8;           // Assign literal value of 8 to the variable number
number = number + 1;  // Evaluate the expression of number + 1,
                      //  and assign the resultant value back to the variable number
```

The symbol "=" is known as the *assignment operator*. The meaning of "=" in programming is different from Mathematics. It denotes *assignment* instead of *equality*. The RHS is a literal value; or an expression that evaluates to a value; while the LHS must be a variable. Note that `x = x + 1` is valid (and often used) in programming. It evaluates `x + 1` and assign the resultant value to the variable `x`. `x = x + 1` illegal in Mathematics. While `x + y = 1` is allowed in Mathematics, it is invalid in programming (because the LHS of an assignment statement must be a variable). Some programming languages use symbol "`:=`", "←", "->", or "→" as the assignment operator to avoid confusion with equality.

01.7 Fundamental Types

Integers: C supports these integer types: `char`, `short`, `int`, `long`, `long long` (in C11) in a non-decreasing order of size. The actual size depends on the implementation. The integers (except `char`) are *signed* number (which can hold zero, positive and negative numbers). You could use the keyword `unsigned [char|short|int|long|long long]` to declare an *unsigned* integers (which can hold zero and positive numbers). There are a total 10 types of integers - `signed|unsigned` combined with `char|short|int|long|long long`.

Characters: Characters (e.g., `'a'`, `'Z'`, `'0'`, `'9'`) are encoded in ASCII into integers, and kept in type `char`. For example, character `'0'` is `48` (decimal) or `30H` (hexadecimal); character `'A'` is `65` (decimal) or `41H` (hexadecimal); character `'a'` is `97` (decimal) or `61H` (hexadecimal). Take note that the type `char` can be interpreted as character in ASCII code, or an 8-bit integer. Unlike `int` or `long`, which is `signed`, `char` could be `signed` or `unsigned`, depending on the implementation. You can use `signed char` or `unsigned char` to explicitly declare `signed` or `unsigned char`.

Floating-point Numbers: There are 3 floating point types: `float`, `double` and `long double`, for single, double and long double precision floating point numbers. `float` and `double` are represented as specified by IEEE 754 standard. A `float` can represent a number between `±1.40239846×10^-45` and `±3.40282347×10^38`, approximated. A `double` can represented a number between `±4.94065645841246544×10^-324` and `±1.79769313486231570×10^308`, approximated. Take note that not all real numbers can be represented by `float` and `double`, because there are infinite real numbers. Most of the values are approximated.

The table below shows the *typical* size, minimum, maximum for the primitive types. Again, take note that the sizes are implementation dependent.

Category	Type	Description	Bytes (Typical)	Minimum (Typical)	Maximum (Typical)
Integers	int (or signed int)	Signed integer (of at least 16 bits)	4 (2)	-2147483648	2147483647
	unsigned int	Unsigned integer (of at least 16 bits)	4 (2)	0	4294967295
	char	Character (can be either signed or unsigned depends on implementation)	1		
	signed char	Character or signed tiny integer (guarantee to be signed)	1	-128	127
	unsigned char	Character or unsigned tiny integer (guarantee to be unsigned)	1	0	255
	short (or short int) (or signed short) (or signed short int)	Short signed integer (of at least 16 bits)	2	-32768	32767
	unsigned short (or unsigned shot int)	Unsigned short integer (of at least 16 bits)	2	0	65535
	long (or long int) (or signed long) (or signed long int)	Long signed integer (of at least 32 bits)	4 (8)	-2147483648	2147483647
	unsigned long (or unsigned long int)	Unsigned long integer (of at least 32 bits)	4 (8)	0	same as above
	long long (or long long int) (or signed long long) (or signed long long int)	Very long signed integer (of at least 64 bits)	8	-2^{63}	$2^{63}-1$
	unsigned long long (or unsigned long long int)	Unsigned very long integer (of at least 64 bits)	8	0	$2^{64}-1$
Real Numbers	float	Floating-point number, ≈7 digits (IEEE 754 single-precision floating point format)	4	3.4e38	3.4e-38
	double	Double precision floating-point number, ≈15 digits (IEEE 754 double-precision floating point format)	8	1.7e308	1.7e-308
	long double	Long double precision floating-point number, ≈19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		

In addition, many C library functions use a type called `size_t`, which is equivalent (`typedef`) to a `unsigned int`, meant for counting, size or length, with 0 and positive integers.

*The `sizeof` Operator

C provides an unary `sizeof` operator to get the size of the operand (in bytes). The following program uses `sizeof` operator to print the size of the fundamental types.

```
/*
 * Print Size of Fundamental Types (SizeofTypes.cpp).
 */
#include <stdio.h>

int main() {
  printf("sizeof(char) is %d bytes.\n", sizeof(char));
  printf("sizeof(short) is %d bytes.\n", sizeof(short));
  printf("sizeof(int) is %d bytes.\n", sizeof(int));
  printf("sizeof(long) is %d bytes.\n", sizeof(long));
  printf("sizeof(long long) is %d bytes.\n", sizeof(long long));
  printf("sizeof(float) is %d bytes.\n", sizeof(float));
  printf("sizeof(double) is %d bytes.\n", sizeof(double));
  printf("sizeof(long double) is %d bytes.\n", sizeof(long double));
  return 0;
}
```

```
sizeof(char) is 1 bytes.
sizeof(short) is 2 bytes.
sizeof(int) is 4 bytes.
sizeof(long) is 4 bytes.
sizeof(long long) is 8 bytes.
sizeof(float) is 4 bytes.
sizeof(double) is 8 bytes.
sizeof(long double) is 12 bytes.
```

The results may vary among different systems.

*Header `<limits.h>`

The `limits.h` header contains information about limits of integer type. For example,

```c
/* Test integer limits in <limits.h> header */
#include <stdio.h>
#include <limits.h>  // integer limits

int main() {
  printf("int max = %d\n", INT_MAX);
  printf("int min = %d\n", INT_MIN);
  printf("unsigned int max = %u\n", UINT_MAX);

  printf("long max = %ld\n", LONG_MAX);
  printf("long min = %ld\n", LONG_MIN);
  printf("unsigned long max = %lu\n", ULONG_MAX);

  printf("long long max = %lld\n", LLONG_MAX);
  printf("long long min = %lld\n", LLONG_MIN);
  printf("unsigned long long max = %llu\n", ULLONG_MAX);

  printf("Bits in char = %d\n", CHAR_BIT);
  printf("char max = %d\n", CHAR_MAX);
  printf("char min = %d\n", CHAR_MIN);
  printf("signed char max = %d\n", SCHAR_MAX);
  printf("signed char min = %d\n", SCHAR_MIN);
  printf("unsigned char max = %u\n", UCHAR_MAX);
  return 0;
}
```

```
int max = 2147483647
int min = -2147483648
unsigned int max = 4294967295
long max = 2147483647
long min = -2147483648
unsigned long max = 4294967295
long long max = 9223372036854775807
long long min = -9223372036854775808
unsigned long long max = 18446744073709551615
Bits in char = 8
char max = 127
char min = -128
signed char max = 127
signed char min = -128
unsigned char max = 255
```

Again, the outputs depend on the system.

The minimum of unsigned integer is always 0. The other constants are SHRT_MAX, SHRT_MIN, USHRT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX. Try inspecting this header (search for `limits.h` under your compiler).

*Header <float.h>

Similarly, the float.h header contain information on limits for floating point numbers, such as minimum number of significant digits (FLT_DIG, DBL_DIG, LDBL_DIG for float, double and long double), number of bits for mantissa (FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG), maximum and minimum exponent values, etc. Try inspecting this header (search for cfloat under your compiler).

Choosing Types

As a programmer, you need to choose variables and decide on the type of the variables to be used in your programs. Most of the times, the decision is intuitive. For example, use an integer type for counting and whole number; a floating-point type for number with fractional part, char for a single character, and boolean for binary outcome.

Rule of Thumb

- Use int for integer and double for floating point numbers. Use byte, short, long and float only if you have a good reason to choose that specific precision.
- Use int (or unsigned int) for *counting* and *indexing*, NOT floating-point type (float or double). This is because integer type are precise and more efficient in operations.
- Use an integer type if possible. Use a floating-point type only if the number contains a fractional part.

Read my article on "Data Representation" if you wish to understand how the numbers and characters are represented inside the computer memory. In brief, It is important to take note that char '1' is different from int 1, short 1, float 1.0, double 1.0, and String "1". They are represented differently in the computer memory, with different precision and interpretation. For example, short 1 is "00000000 00000001", int 1 is "00000000 00000000 00000000 00000001", long long 1 is "00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001", float 1.0 is "0 01111111 0000000 00000000 00000000", double 1.0 is "0 01111111111 0000 00000000 00000000 00000000 00000000 00000000 00000000", char '1' is "00110001".

There is a subtle difference between int 0 and double 0.0.

Furthermore, you MUST know the type of a value before you can interpret a value. For example, this value "00000000 00000000 00000000 00000001" cannot be interpreted unless you know the type.

*The typedef Statement

Typing "unsigned int" many time can get annoying. The typedef statement can be used to create a new name for an existing type. For example, you can create a new type called "uint" for "unsigned int" as follow. You should place the typedef immediately after #include. Use typedef with care because it makes the program hard to read and understand.

```
typedef unsigned int uint;
```

Many C compilers define a type called size_t, which is a typedef of unsigned int.

```
typedef unsigned int size_t;
```

01.8 Output via `printf()` Function

C programs use function `printf()` of library `stdio` to print output to the console. You need to issue a so-called *preprocessor directive* "`#include <stdio.h>`" to use `printf()`.

To print a string literal such as "Hello, world", simply place it inside the parentheses, as follow:

```
printf(aStringLiteral);
```

For example,

```
printf("Hello, world\n");
```

```
Hello, world
_
```

The `\n` represents the *newline* character. Printing a newline advances the cursor (denoted by _ in the above example) to the beginning of next line. `printf()`, by default, places the cursor after the printed string, and does not advance the cursor to the next line. For example,

```
printf("Hello");
printf(", ");
printf("world!");
printf("\n");
printf("Hello\nworld\nagain\n");
```

```
Hello, world!
Hello
world
again
_
```

Formatted Output via `printf()`

The "f" in `printf()` stands for "formatted" printing. To do formatted printing, you need to use the following syntax:

```
printf(formattingString, variable1, variable2, ...)
```

The *formattingString* is a string composing of normal texts and *conversion specifiers*. Normal texts will be printed as they are. A conversion specifier begins with a percent sign (%), followed by a code to specify the type of variable and format of the output (such as the field width and number of decimal places). For example, %d denotes an int; %3d for an `int` with field-width of 3. The conversion specifiers are used as *placeholders*, which will be substituted by the variables given after the formatting string in a sequential manner. For example,

```
/*
 * Test formatted printing for int (TestPrintfInt.c)
 */
#include <stdio.h>

int main() {
  int number1 = 12345, number2 = 678;
  printf("Hello, number1 is %d.\n", number1);             // 1 format specifier
  printf("number1=%d, number2=%d.\n", number1, number2); // 2 format specifiers
```

```
   printf("number1=%8d, number2=%5d.\n", number1, number2);  // Set field-widths
   printf("number1=%08d, number2=%05d.\n", number1, number2); // Pad with zero
   printf("number1=%-8d, number2=%-5d.\n", number1, number2); // Left-align
   return 0;
}
```

```
Hello, number1 is 12345.
number1=12345, number2=678.
number1=   12345, number2=  678.
number1=00012345, number2=00678.
number1=12345   , number2=678  .
```

Type Conversion Code

The commonly-used type conversion codes are:

Type	Type Conversion Code	Type & Format
Integers	`%d` (or `%i`)	`(signed) int`
	`%u`	`unsigned int`
	`%o`	`int` in octal
	`%x, %X`	`int` in hexadecimal (%X uses uppercase A-F)
	`%hd, %hu`	`short`, `unsigned short`
	`%ld`, `%lu`	`long`, `unsigned long`
	`%lld`, `%llu`	`long long`, `unsigned long long`
Floating-point	`%f`	`float` in fixed notation
	`%e, %E`	`float` in scientific notation
	`%g, %G`	`float` in fixed/scientific notation depending on its value
	`%f`, `%lf` (`printf`), `%lf` (`scanf`)	`double`: Use `%f` or `%lf` in `printf()`, but `%lf` in `scanf()`.
	`%Lf`, `%Le`, `%LE`, `%Lg`, `%LG`	`long double`
Character	`%c`	`char`
String	`%s`	string

Notes:

- For `double`, you must use `%lf` (for long float) in `scanf()` (or `%le`, `%lE`, `%lg`, `%lG`), but you can use either `%f` or `%lf` in `printf()` (or `%e`, `%E`, `%g`, `%G`, `%le`, `%lE`, `%lg`, `%lG`).
- Use `%%` to print a `%` in the formatting string.

For example,

```
int anInt = 12345;
float aFloat = 55.6677;
double aDouble = 11.2233;
char aChar = 'a';
char aStr[] = "Hello";
```

```
printf("The int is %d.\n", anInt);
//The int is 12345.
printf("The float is %f.\n", aFloat);
//The float is 55.667702.
printf("The double is %lf.\n", aDouble);
//The double is 11.223300.
printf("The char is %c.\n", aChar);
//The char is a.
printf("The string is %s.\n", aStr);
//The string is Hello.

printf("The int (in hex) is %x.\n", anInt);
//The int (in hex) is 3039.
printf("The double (in scientific) is %le.\n", aDouble);
//The double (in scientific) is 1.122330e+01.
printf("The float (in scientific) is %E.\n", aFloat);
//The float (in scientific) is 5.566770E+01.
```

Using the wrong type conversion code usually produces garbage.

Field Width

You can optionally specify a field-width before the type conversion code, e.g., `%3d`, `%6f`, `%20s`. If the value to be formatted is shorter than the field width, it will be padded with spaces (by default). Otherwise, the field-width will be ignored. For example,

```
int number = 123456;
printf("number=%d.\n", number);
// number=123456.
printf("number=%8d.\n", number);
// number=  123456.
printf("number=%3d.\n", number);  // Field-width too short. Ignored.
// number=123456.
```

Precision (Decimal Places) for Floating-point Numbers

For floating-point numbers, you can optionally specify the number of decimal places to be printed, e.g., `%6.2f`, `%8.3f`. For example,

```
double value = 123.14159265;
printf("value=%lf;\n", value);
//value=123.141593;
printf("value=%6.2lf;\n", value);
//value=123.14;
printf("value=%9.4lf;\n", value);
//value= 123.1416;
printf("value=%3.2lf;\n", value);  // Field-width too short. Ignored.
//value=123.14;
```

Alignment

The output are right-aligned by default. You could include a "-" flag (before the field width) to ask for left-aligned. For example,

```
int i1 = 12345, i2 = 678;
printf("Hello, first int is %d, second int is %5d.\n", i1, i2);
//Hello, first int is 12345, second int is   678.
printf("Hello, first int is %d, second int is %-5d.\n", i1, i2);
//Hello, first int is 12345, second int is 678  .

char msg[] = "Hello";
printf("xx%20sxx\n", msg);
//xx     Helloxx
printf("xx%-20sxx\n", msg);
//xxHello     xx
```

Others

- + (plus sign): display plus or minus sign preceding the number.
- # or `0`: Pad with leading # or `0`.

C11's `printf_s()/scanf_s()`

C11 introduces more secure version of `printf()/scanf()` called `printf_s()/scanf_s()` to deal with mismatched conversion specifiers. Microsoft Visual C implemented its own versions of `printf_s()/scanf_s()` before C11, and issues a *deprecated* warning for using `printf()/scanf()`.

01.9 Input via `scanf()` Function

In C, you can use `scanf()` function of `<stdio.h>` to read inputs from keyboard. `scanf()` uses the type-conversion code like `printf()`. For example,

```
/*
 * TestScanf.c
 */
#include <stdio.h>

int main() {
   int anInt;
   float aFloat;
   double aDouble;

   printf("Enter an int: ");  // Prompting message
   scanf("%d", &anInt);       // Read an int from keyboard and assign to
variable anInt.
   printf("The value entered is %d.\n", anInt);

   printf("Enter a floating-point number: ");  // Prompting message
   scanf("%f", &aFloat);     // Read a double from keyboard and assign to
variable aFloat.
   printf("The value entered is %f.\n", aFloat);

   printf("Enter a floating-point number: ");  // Prompting message
   scanf("%lf", &aDouble);     // Read a double from keyboard and assign to
variable aDouble.
   printf("The value entered is %lf.\n", aDouble);

   return 0;
}
```

Notes:

- To place the input into a variable in `scanf()`, you need to prefix the variable name by an ampersand sign (&). The ampersand (&) is called address-of operator, which will be explained later. However, it is important to stress that missing ampersand (&) is a common error.
- For double, you must use type conversion code `%lf` for `scanf()`. You could use `%f` or `%lf` for `printf()`.

Return-Value for `scanf()`

The `scanf()` returns an `int` indicating the number of values read.

For example,

```
int number1 = 55, number2 = 66;
int rcode = scanf("%d", &number);
printf("return code is %d\n", rcode);
printf("number1 is %d\n", number1);
printf("number2 is %d\n", number2);
```

The `scanf()` returns 1 if user enters an integer which is read into the variable `number`. It returns 0 if user enters a non-integer (such as "hello"), and variable number is not assigned.

```
int number1 = 55, number2 = 66;
int rcode = scanf("%d%d", &number1, &number2);
printf("return code is %d\n", rcode);
printf("number1 is %d\n", number1);
printf("number2 is %d\n", number2);
```

The `scanf()` returns 2 if user enters two integers that are read into `number1` and `number2`. It returns 1 if user enters an integer followed by a non-integer, and `number2` will not be affected. It returns 0 if user enters a non-integer, and both `number1` and `number2` will not be affected.

Checking the return code of `scanf()` is recommended for *secure coding*.

01.10 Literals for Fundamental Types and String

A *literal* is a *specific constant value*, such as 123, -456, 3.14, 'a', "Hello", that can be assigned directly to a variable; or used as part of an expression. They are called *literals* because they literally and explicitly identify their values.

Integer Literals

A whole number, such as 123 and -456, is treated as an int, by default. For example,

```
int number = -123;
int sum = 4567;
int bigSum = 8234567890;  // ERROR: this value is outside the range of int
```

An int literal may precede with a plus (+) or minus (-) sign, followed by digits. No commas or special symbols (e.g., $ or space) is allowed (e.g., 1,234 and $123 are invalid). No preceding 0 is allowed too (e.g., 007 is invalid).

Besides the default base 10 integers, you can use a prefix '0' (zero) to denote a value in octal, prefix '0x' for a value in hexadecimal, and prefix '0b' for binary value (in some compilers), e.g.,

```
int number1 = 1234;       // Decimal
int number2 = 01234;      // Octal 1234, Decimal 2322
int number3 = 0x1abc;     // hexadecimal 1ABC, decimal 15274
int number4 = 0b10001001; // binary (may not work in some compilers)
```

A long literal is identified by a suffix 'L' or 'l' (avoid lowercase, which can be confused with the number one). A long long int is identified by a suffix 'LL'. You can also use suffix 'U' for unsigned int, 'UL' for unsigned long, and 'ULL' for unsigned long long int. For example,

```
long number = 12345678L;     // Suffix 'L' for long
long sum = 123;              // int 123 auto-casts to long 123L
long long bigNumber = 987654321LL;  // Need suffix 'LL' for long long int
```

No suffix is needed for short literals. But you can only use integer values in the permitted range. For example,

```
short smallNumber = 1234567890;   // ERROR: this value is outside the range of short.
short midSizeNumber = -12345;
```

Floating-point Literals

A number with a decimal point, such as 55.66 and -33.44, is treated as a double, by default. You can also express them in scientific notation, e.g., 1.2e3, -5.5E-6, where e or E denotes the exponent in power of 10. You could precede the fractional part or exponent with a plus (+) or minus (-) sign. Exponent shall be an integer. There should be no space or other characters (e.g., space) in the number.

You MUST use a suffix of 'f' or 'F' for float literals, e.g., -1.2345F. For example,

```
float average = 55.66;      // Error! RHS is a double. Need suffix 'f' for float.
float average = 55.66f;
```

Use suffix 'L' (or 'l') for long double.

Character Literals and Escape Sequences

A printable `char` literal is written by enclosing the character with a pair of *single quotes*, e.g., `'z'`, `'$'`, and `'9'`. In C, characters are represented using 8-bit ASCII code, and can be treated as a 8-bit *signed integers* in arithmetic operations. In other words, `char` and 8-bit signed integer are interchangeable. You can also assign an integer in the range of `[-128, 127]` to a `char` variable; and `[0, 255]` to an `unsigned char`.

You can find the ASCII code table HERE.

For example,

```
char letter = 'a';                // Same as 97
char anotherLetter = 98;          // Same as the letter 'b'
printf("%c\n", letter);           // 'a' printed
printf("%c\n", anotherLetter); // 'b' printed instead of the number
anotherLetter += 2;               // 100 or 'd'
printf("%c\n", anotherLetter);  // 'd' printed
printf("%d\n", anotherLetter); // 100 printed
```

Non-printable and control characters can be represented by so-called *escape sequences*, which begins with a back-slash (\) followed by a code. The commonly-used escape sequences are:

Escape Sequence	Description	Hex (Decimal)
\n	New-line (or Line-feed)	0AH (10D)
\r	Carriage-return	0DH (13D)
\t	Tab	09H (9D)
\"	Double-quote (needed to include " in double-quoted string)	22H (34D)
\'	Single-quote	27H (39D)
\\	Back-slash (to resolve ambiguity)	5CH (92D)

Notes:

- New-line (`0AH`) and carriage return (`0dH`), represented by `\n`, and `\r` respectively, are used as *line delimiter* (or *end-of-line*, or *EOL*). However, take note that UNIX/Linux/Mac use `\n` as EOL, Windows use `\r\n`.
- Horizontal Tab (`09H`) is represented as `\t`.
- To resolve ambiguity, characters back-slash (`\`), single-quote (`'`) and double-quote (`"`) are represented using escape sequences `\\`, `\'` and `\"`, respectively. This is because a single back-slash begins an escape sequence, while single-quotes and double-quotes are used to enclose character and string.
- Other less commonly-used escape sequences are: `\?` or `?`, `\a` for alert or bell, `\b` for backspace, `\f` for form-feed, `\v` for vertical tab. These may not be supported in some consoles.

The `<ctype.h>` Header

The `ctype.h` header provides functions such as `isalpha()`, `isdigit()`, `isspace()`, `ispunct()`, `isalnum()`, `isupper()`, `islower()` to determine the type of character; and `toupper()`, `tolower()` for case conversion.

String Literals

A `String` literal is composed of zero of more characters surrounded by a pair of *double quotes*, e.g., `"Hello, world!"`, `"The sum is "`, `""`.

String literals may contains escape sequences. Inside a `String`, you need to use `\"` for double-quote to distinguish it from the ending double-quote, e.g. `"\"quoted\""`. Single quote inside a `String` does not require escape sequence. For example,

```
printf("Use \\\" to place\n a \" within\ta\tstring\n");
```

```
Use \" to place
 a " within   a      string
```

TRY: Write a program to print the following picture. Take note that you need to use escape sequences to print special characters.

```
         '__'
          (oo)
  +========\/
 / || %%% ||
*  ||-----||
   ""     ""
```

Example (Literals)

```
/* Testing Primitive Types (TestLiteral.c) */
#include <stdio.h>

int main() {
   char gender = 'm';             // char is single-quoted
   unsigned short numChildren = 8; // [0, 255]
   short yearOfBirth = 1945;      // [-32767, 32768]
   unsigned int salary = 88000;   // [0, 4294967295]
   double weight = 88.88;         // With fractional part
   float gpa = 3.88f;             // Need suffix 'f' for float

   printf("Gender is %c.\n", gender);
   printf("Number of children is %u.\n", numChildren);
   printf("Year of birth is %d.\n", yearOfBirth);
   printf("Salary is %u.\n", salary);
   printf("Weight is %.2lf.\n", weight);
   printf("GPA is %.2f.\n", gpa);
   return 0;
}
```

```
Gender is m.
Number of children is 8.
Year of birth is 1945.
Salary is 88000.
Weight is 88.88.
GPA is 3.88.
```