Introduction to Programming

Basics in C

Nguyen Dinh Quyen

CSC01.D16D17.W2 | 12-18.10.12

Content

- ✧ The C programming language
- ✧ Structure of a `.c` file
- ✧ Identifiers, keywords, variables, datatypes
- ✧ Operators, expressions
- ✧ Functions
- ✧ Console I/O

The C Programming Language

- Dennis Ritchie – AT&T Bell Laboratories – 1972
  - 16-bit DEC PDP-11 computer
- Widely used today
  - extends to newer system architectures
  - efficiency/performance
  - low-level access (to computer memory)

- http://en.wikipedia.org/wiki/C_(programming_language)

The C Programming Language

- Some characteristics of C
  - Imperative (procedural)
  - Few keywords
  - Large number of arithmetical and logical operators
  - Structures, unions – compound data types
  - Pointers – memory, arrays
  - External standard library – I/O, other facilities
  - Compiles to native code
  - Macro preprocessor

The C Programming Language

- Versions of C
  - 1972 – C invented
  - 1978 – *The C Programming Language* published; first specification of language (K & R C)
  - 1989 – C89 standard (known as ANSI C or Standard C)
  - 1990 – ANSI C adopted by ISO, known as C90, and in 1999 – C99 standard
    - mostly backward-compatible
    - not completely implemented in many compilers
  - 2011-12-08 – current new standard, known as C11

The C Programming Language

- Used for systems programming:
  - OSes, like Linux
  - microcontrollers: automobiles and airplanes
  - embedded processors: phones, portable electronics, etc.
  - DSP processors: digital audio and TV systems...
  - …

The C Programming Language

- C vs. related languages
  - More recent derivatives: C++, Objective C, C#
  - Influenced: Java, Perl, PHP, Python (quite different)…
  - C lacks:
    - exceptions
    - range-checking
    - garbage collection
    - object-oriented programming
    - polymorphism...
    - …
  - Low-level language ☞ faster code (usually)

The C Programming Language

- C → C++
  - C++ uses `new` and `delete` operators for memory management while C uses library functions.
  - C++ extends structs and unions, and also includes classes.
  - C++ has templates while C does not.
  - C++ has operator and function overloading.
  - C++ requires type casts conversions but C is less strict and C++ also provides alternative type conversion semantics.
  - `typedef` creates unique types in C++, and creates type aliasing in C.

Structure of a .c file

/ Begin with comments about file contents */*

Insert `#include` *statements and preprocessor definitions*

Function prototypes and variable declarations

Define `main()` *function*

{

Function body

}

Define other function

{

Function body

}

Comments

- Comments : `/* this is a simple comment */`
- In C++: `// this is another simple comment`
- Can span multiple lines

```
/* This comments
   pans multiple lines */
```

- Completely ignored by compiler
- Can appear almost anywhere

The `#include` macro

- Header files: constants, functions, other declarations
- `#include <stdio.h>` – read the contents of the header file stdio.h
  - stdio.h: standard I/O functions for console, files, part of the C Standard Library
  - other important header files: conio.h, ctype.h, math.h, stdlib.h, string.h, time.h
- included files must be on *include path*
  - User-defined header files typically in the same folder with *.c (using `#include “filename.h”`)

Definitions of variables and so on…

- A *variable* is a named memory location which temporarily stores data that can change while the program is running.
- A *constant* is a named memory location which temporarily stores data that remains the same throughout the execution of the program.
- The *type* of a variable indicates what kind of value it will store.
- The name of a variable is known as its *identifier*.
- A variable can be given a value through an *assignment statement*.

Character set

- The basic C source character set includes the following characters:
  - Letters: `a–z, A–Z, _`
  - Digits: `0–9`
  - Punctuation: `~ ! @ # % ^ & * ( ) - + = : ; " ' < > , . ? | / \ { } [ ]`
  - Whitespace characters: space, horizontal tab, vertical tab, form feed, newline

Keywords

- C89 has 32 keywords (reserved words with special meaning):

```
auto     break    case      char      const
continue default  do        double    else
enum     extern   float     for       goto
if       int      long      register  return
short    signed   sizeof    static    struct
switch   typedef  union     unsigned  void
volatile while
```

- C99 adds five more keywords:

```
_Bool    _Complex _Imaginary inlinere strict
```

- C11 adds seven more keywords:

```
_Alignas  _Alignof       _Atomic       _Generic
_Noreturn _Static_assert _Thread_local
```

Declaring variables

- Must declare variables before use
- Variable declaration: *`data-type identifier;`*

```
int n;
float phi;
```

- `int` - integer data type
- `float` - floating-point data type
- Variable names must be begin with a letter, and then continue to be letters, digits, and underscores.
- Variable names are case sensitive. `int x; int X;` declares two different variables

Variable names

- *Valid:*
  - `player`
  - `firstStudent`
  - `queryBuilder`
- *Invalid (no dollar signs, must begin with letter, no spaces):*
  - `zoo$`
  - `1player`
  - `query Builder`
- *Bad practice:*
  - `_player`

Assignment statements

- Because programming languages solve problems by directly manipulating data in memory, there must be a way to carry out operations on data.
  - assignment operators
  - extended and shorthand assignment operators (C++)
  - arithmetic operators
  - miscellaneous “built-in” operators
- Assignment syntax: *`variable = value;`*
- Common errors:
  - `x = 29 % 4.8;`
  - `p + 7 = x + 3;`

Initializing variables

- Uninitialized, variable assumes a default value
- Variables initialized via assignment operator:

```
n = 3;
```

- Can also initialize at declaration:

```
float phi = 1.6180339887;
```

- Can declare/initialize multiple variables at once:

```
int a, b, c = 0, d = 4;
```

Constants

- A constant variable must be initialized and can never change. The form for named constants is:

  `const` *`data-type identifier = value;`*

- *value* is a literal.

Preprocessor macros

- Preprocessor macros begin with `#` character such as `#include <stdio.h>`
- `#define msg "hello, 6.087 students"` defines *msg as "hello, 6.087 students"* throughout source file
- many constants specified this way
- `#define` can take arguments and be treated like a function `#define add3(x,y,z) ((x)+(y)+(z))`
- parentheses ensure order of operations
- compiler performs inline replacement; not suitable for recursion

Datatypes

- The *datatype* of an object in memory determines the set of values it can have and what operations that can be performed on it.
- C is a *weakly typed language*. It allows implicit conversions as well as forced (potentially dangerous) casting.

Datatypes

types		size
integers	`char, unsigned char, signed char`	1 byte
	`short, unsigned short`	2 bytes
	`int, unsigned int`	4 bytes
	`long, unsigned long`	4 bytes
Floating points	`float`	4 bytes
	`double`	8 bytes
	`long double`	8 bytes

- Unspecified: `void`
- Boolean in C99

Rules for assigning variables

- Assign `short`, `int` or `long` data types when you are sure a variable is an integer number (NO decimal points).
  - Here, for simple, we mostly use `int`.
- Assign `float` or `double` when decimals are needed.
  - Here primarily use `double`.
- Assign `char` if the variable will always contain ONE character of data. This means only one letter (or character).

Assignment statements

- An extended assignment statement is used to assign a value to multiple variables during a single executable statement.

  For example: `int x = y = i = a`

- Shorthand assignment operators (aka compound assignment operators):

  `n += expression;` // same as n = n + (expression)
  `n -= expression;` // same as n = n - (expression)
  `n *= expression;` // same as n = n * (expression)
  `n /= expression;` // same as n = n / (expression)
  `n %= expression;` // same as n = n%(expression)

Arithmetic operators

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulus) – for integers only

- Misc. “Built-In” operators
  - ++ (increment)
  - – (decrement)

Increment and decrement operators

- Increment and decrement are common arithmetic operation. C provides two short cuts for the same.
- *Postfix*
  - `x++` is a short cut for `x=x+1`
  - `x--` is a short cut for `x=x-1`
  - `y=x++` is a short cut for `y=x; x=x+1;`. *x* is evaluated *before it is incremented*.
  - `y=x--` is a short cut for `y=x; x=x-1;`. *x* is evaluated *before it is decremented*.

Increment and decrement operators

- *Prefix*:
  - `++x` is a short cut for `x=x+1`
  - `--x` is a short cut for `x=x-1`
  - `y=++x` is a short cut for `x=x+1;y=x;`. `x` is evaluated *after it is incremented*.
  - `y=--x` is a short cut for `x=x-1;y=x;`. `x` is evaluated *after it is decremented*.

Other operators

- Operator precedence: from highest (considered first) to lowest (considered last):
  - ( )
  - unary +, -
  - *, /, %
  - binary +, -

Relational Operators

- *Logical expressions*, or boolean expressions, are expressions that result in being true or false.
  - A value of 1 is true. A value of 0 is a false expression.
- Relational Operators
  - > : greater than
  - < : less than
  - >= : greater than or equal to
  - <= : less than or equal to
  - == : is equal to
  - != : is not equal to
- *Be careful about “=” and “==”!!!*

Logical expressions

- Logical operators: allow multiple relational expressions to be combined.
  - && : logical and
  - || : logical or
  - ! : logical not

Arithmetic expressions

Suppose x and y are variables

- `x+y`, `x-y`, `x*y`, `x/y`, `x%y`: binary arithmetic
- A simple statement:

  `y = x+3*x/(y-4);`

- Numeric literals like 3 or 4 valid in expressions
- Semicolon ends statement (not newline)
- `x += y`, `x -= y`, `x *= y`, `x /= y`, `x %= y`: arithmetic and assignment

Order of operations

- The operators have precedence (highest to lowest):
  - Arithmetic → relational → logical
- Exception is logical not (!) before arithmetic ops.
- Order of operations:

  1 : !, +, - (unary)
  2 : *, /, %
  3 : +, - (binary)
  4 : <, <=, >, >=
  5 : ==, !=
  6 : &&
  7 : ||
- Use parentheses to override order of evaluation

Order of operations

- Assume `x =2.0` and `y=6.0.` Evaluate the statement

  `float z = x+3*x/(y-4);`

  1. Evaluate expression in parentheses

  `float z = x+3*x/(y-4); → float z = x+3*x/2.0;`

  2. Evaluate multiplies and divides, from left-to-right

  `float z = x+3*x/2.0; → float z = x+6.0/2.0;`
  `→ float z = x+3.0;`

  3. Evaluate addition `float z=x+3.0; → float z=5.0;`

  4. Perform initialization with assignment. Now, `z =5.0`.

- How do I insert parentheses to get z =4.0?

  `float z = (x+3*x)/(y-4);`

Expressions

Common Errors:

- ✧ `if (midTerm = 'A') { ... }`

  → `if (midTerm == 'A') { ... }`

- ✧ `midTerm || final == 'A'`

  → `midTerm == 'A' || final == 'A'`

- ✧ `a < b > c`

  → `a < b && b > c`

- ✧ `if (x != -1 || x != -99) { ... }`

  → `if (x != -1 && x != -99) { ... }`

Type conversion

- Implicit conversion. Example:

```
int a;
float b = 3.5;
float c = 5.0;
int d = 2; a = b * c / d; // ☞ 8
```

- *d* was implicitly converted to 2.0 during *b* * *c* / *d*. The result of that arithmetic is 8.75, but *a* was declared *int*, so *a* was implicitly converted to 8 during the assignment statement.

Type conversion

- Explicit conversion: also referred to as typecasting
  - (type)variable
  - type(variable)
- *Example 1:*

```
double d; int k;
d = 12.78;
k = (int)d; // k = 12
d = double(k); // d = 12.0
```

- *Example 2:*

```
int m=10, n=4, r; double q;
q = m / n; // 2.0
r = (double)m / n; // 2
r = m / n; // 2
q = (double)m / n; // 2.5
```

Summary of operators

- arithmetic: +, -, *, /, %
- assignment: =
- augmented assignment: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- bitwise logic: ~, &, |, ^
- bitwise shifts: <<, >>
- boolean logic: !, &&, ||
- conditional evaluation: ? :
- equality testing: ==, !=
- calling functions: ( )
- increment and decrement: ++, --
- member selection: ., ->
- object size: sizeof
- order relations: <, <=, >, >=
- reference and dereference: &, *, [ ]
- sequencing: ,
- subexpression grouping: ( )
- type conversion: (*typename*)

✧ http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Function prototypes

- Functions also must be declared before use
- Declaration called *function prototype*
- Function prototypes:

```
int factorial(int);
```

or

```
int factorial(int n);
```

- Prototypes for many common functions in header files for C Standard Library

Function prototypes

- General form:

  return_type func_name(arg1,arg2,...);

- Arguments: local variables, values passed from caller
- Return value: single value returned to caller when function exits
- void – signifies no return value/arguments

```
int rand(void);
```

The `main()` function

- `main()`: entry point for C program
- Simplest version: no inputs, outputs 0 when successful, and nonzero to signal some error

```
int main(void);
```

- Two-argument form of `main()`: access command-line arguments

```
int main(int argc, char **argv);
```

Function definitions

Function declaration
{
    declare variables;
    program statements;
}

- Must match prototype (if there is one)
  - variable names don't have to match
  - no semicolon at end
- Curly braces define a *block – region of code*
  - Variables declared in a block exist only in that block
- Variable declarations before any other statements

Console I/O

- stdout, stdin: console output and input streams
- printf, scanf: print output and scan input
- puts(*string):* print string to stdout
- putchar(*char):* print character to stdout
- *char* = getchar()*:* return character from stdin
- *string* = gets*(string):* read line from stdin into string
- *…*

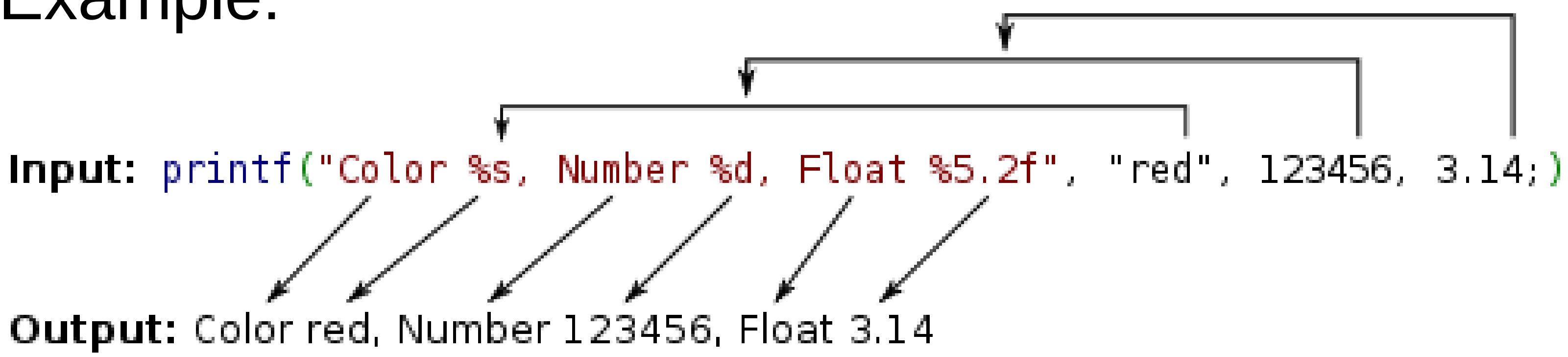
Strings

- Strings stored as character array
- Null-terminated (last character in array is ’\0’ null)
  - Not written explicitly in string literals
- Special characters specified using \ (escape character):
  - \\ – backslash, \’ – apostrophe, \” – quotation mark
  - \b, \t, \r, \n – backspace, tab, carriage return, linefeed
  - \ooo, \xhh – octal and hexadecimal ASCII character codes, e.g. \x41 – ’A’, \060 – ’0’

printf

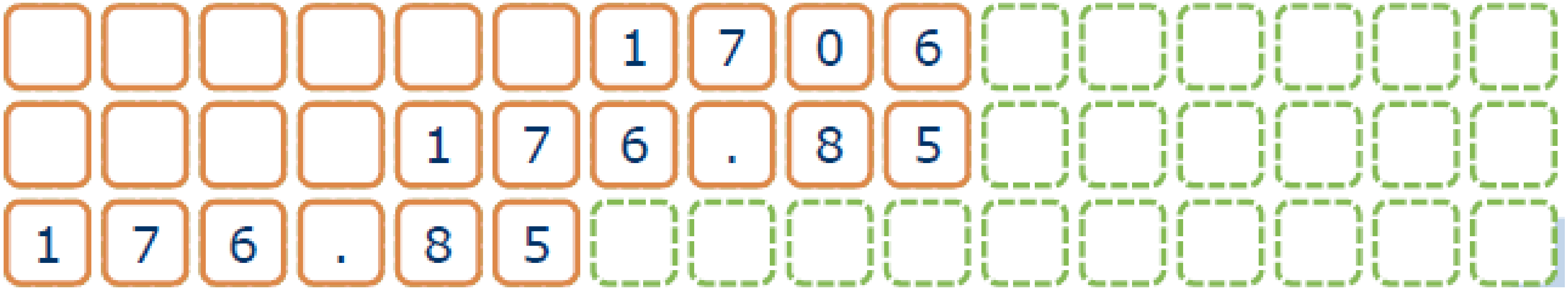
```
int printf(const char * format[, arg]...);
```

- `format` contains in literal text, escape sequences, and specifers
- A *format specifier* follows this prototype: %[flags][width][.precision][length]specifier
- Example:



printf

```c
int a = 1706;
float x = 176.85;
printf(“%10d”, a); printf(“\n”);
printf(“%10.2f”, x); printf(“\n”);
printf(“%.2f”, x); printf(“\n”);
```



printf

Specifier	Meaning	Types
%c	Character	char
%d, %ld	Signed integers	char, int, short, long
%f, %lf	Floating point numbers	float, double
%s	String	char[], char*
%u	Unsigned integers	unsigned int/short/long

scanf

```
int scanf(const char *format [, arg]... );
```

- Specifying variables though memory addresses.
- Examples:
  - scanf("%d", &a); // scan integer value for a
  - scanf("%d%d", &a, &b); // scan values for a and b
- Mistakes:
  - scanf("%d", a); // lacking &
  - scanf("%d", &a, &b); // lacking %d for b
  - scanf("%f", &a); // a is an integer
  - scanf("%9d", &a); // wrong specifier
  - scanf("a = %d, b = %d", &a, &b); // wrong specifier