C Programming

1. Introduction to C
2. C Fundamentals
3. Formatted Input/Output
4. Expression
5. Selection Statement
6. Loops
7. Basic Types
8. Arrays
9. Functions
10. Pointers
11. Pointers and Arrays
Introduction to C

• C is a “low-level” language
  • suitable language for systems programming

• C is a small language
  • relies on a “library” of standard function

• C is a permissive language
  • it assumes that you know what you’re doing, so it allows you a wider degree of latitude than many languages. It doesn’t mandate the detailed error-checking found in other language.
Introduction to C

• Strengths:
  • Efficiency: intended for applications where assembly language had traditionally been used.
  • Portability: hasn’t splintered into incompatible dialects; small and easily written.
  • Power: large collection of data types and operators
  • Flexibility: not only for system but also for embedded system commercial date processing
  • Standard library
  • Integration with UNIX
Introduction to C

• Weaknesses
  • error-prone
  • difficult to understand
  • difficult to modify
Similarities of C to Java

• /*Comments*/
• Variable declarations
• if else statements
• for loops
• while loops
• function definition (like methods)
• Main function starts program
Differences between C and Java

- C does not have objects. There are “struct”ures.
- C allows pointer manipulation
- Input/Output with C
  - Output with `printf` function
  - Input with `scanf` function
C Fundamentals

• First Program

```c
#include<stdio.h>
int main()
{
    printf("Hello World!");
}
```
C Fundamentals

• Compiling and Linking

• Preprocessing: the program is given to a preprocessor, which obeys commands that begin #\texttt{(directives)} add things to the program and make modifications

• Compiling: modified program -> compile -> object code

• Linking: add library functions to yield a complete executable program
C Fundamentals

- Compiler
- $ gcc -Wall -o Hello Hello.c
# C Fundamentals

<table>
<thead>
<tr>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>do</td>
</tr>
</tbody>
</table>
Variable Type

• C has the four basic data types:
  • char, int, float, and double

• signed, unsigned, short, and long
  • char, signed char, unsigned char
  • short, short int, signed short, signed short int, unsigned short, unsigned short int
  • int, signed int, unsigned int
  • long, long int, signed long, signed long int, unsigned long, unsigned long int
  • long long, long long int, signed long long, signed long long int, unsigned long long, unsigned long long int

• float
• double
• long double
Variable Type

• `sizeof()` function retrieve the size of one type in bytes
  • `sizeof(float)`
# Type Format Specifier

<table>
<thead>
<tr>
<th>Type</th>
<th>Scanf</th>
<th>Printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>char (signed and unsigned)</td>
<td>%c</td>
<td>%c</td>
</tr>
<tr>
<td>short (signed)</td>
<td>%hi</td>
<td>%hi</td>
</tr>
<tr>
<td>unsigned short</td>
<td>%hu</td>
<td>%hu</td>
</tr>
<tr>
<td>int (signed)</td>
<td>%d or %i</td>
<td>%d or %i</td>
</tr>
<tr>
<td>unsigned int</td>
<td>%u</td>
<td>%u</td>
</tr>
<tr>
<td>long (signed)</td>
<td>%li</td>
<td>%li</td>
</tr>
<tr>
<td>unsigned long</td>
<td>%lu</td>
<td>%lu</td>
</tr>
<tr>
<td>long long (signed)</td>
<td>%lli</td>
<td>%lli</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>%llu</td>
<td>%llu</td>
</tr>
<tr>
<td>float</td>
<td>%f</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>%lf</td>
<td>%f</td>
</tr>
<tr>
<td>long double</td>
<td>%Lf</td>
<td>%Lf</td>
</tr>
</tbody>
</table>
unsigned int u;
scanf("%u",&u);    // read and write decimal
printf("%u",u);

scanf("%o",&u);  // read and write octal
printf("%o",u);

scanf("%x",&u);  // read and write hex
printf("%x",u);
Floating Types

- float: single-precision floating-point
- double: double-precision floating-point
- long double: extended-precision floating-point

double x;
scanf("%lf",&x);
printf("%f",x);

double
scanf("%Lf",&x);
printf("%Lf",x);
Character Types

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’

if (‘a’ <= ch && ch <= ‘z’)

for (ch = ‘A’; ch <= ‘Z’; ch++)
Char Type

- ‘a’, ‘\t’, ‘\n’, ‘\0’, etc. are character constants
- strings: character arrays
  - see <string.h> for string functions
  - “I am a string”
  - always null (‘\0’) terminated.
- ‘x’ is different from “x”
Type Conversion

• narrower types are converted into wider types
  • `<ctype.h>` library contains conversion functions

• Boolean values:
  • `true` : `>= 1`
  • `false`: `0`
Type Conversion

- float -> double -> long double
- int -> unsigned int -> long int -> unsigned long int
Type Conversion

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 + i;  //i is converted to float
d = d + f;  //f is converted to double
ld = ld + d;  //d is converted to long double
Casting

• (type name) expression

```c
float f1, f2;
f2 = f1 - (int) f1;

float d;
int i,j;
f = (float) i/j;

short int i;
int j = 1000;
i = j * j;  //wrong
```
typedef int BOOL
BOOL flag; // same as int flag

typedef short int Int16;
typedef long int Int32;
typedef unsigned char Byte;

typedef struct {int age; char * name;} person;
person people;
Formatted Input/Output

- printf function

- printf(string, expr1, expr2, ...)
  - string: ordinary characters and conversion specification (%)
    - %d: int
    - %s: String
    - %f: float

- printf(“i=%d, j=%d, x=%f
”, i, j, x)
Formatted Input/Output

• Conversion Specification

  • %[-]m.pX

    • m: specifies the minimum number of characters to print

    • %4d: _123

  • p: depends on the choice of X

    • d: decimal form

    • e: floating-point number in exponential format

    • f: floating-point number in “fixed decimal” format

    • g: either exponential format or fixed decimal format, depending on the number’s size
Formatted Input/Output

```c
#include<stdio.h>
int main(){
    int i = 40;
    float x = 839.21;
    printf("%d\t%5d\t%-5d\t%5.3d\n",i,i,i,i);
    printf("%10.3f\t%10.3e\t%-10g\n",x,x,x);
}
```

```
<table>
<thead>
<tr>
<th>40</th>
<th>40</th>
<th>40</th>
<th>040</th>
</tr>
</thead>
<tbody>
<tr>
<td>839.210</td>
<td>8.392e+02</td>
<td>839.21</td>
<td></td>
</tr>
</tbody>
</table>
```
Formatted Input/Output

• Escape Sequence

• Enable strings to contain characters that would otherwise cause problem for the compiler

  • e.g.: \n \t " "

• e.g.: \n \t " "

• e.g.: \n \t " "

• e.g.: \n \t " "
How scarf works: is controlled by the conversion specification in the format string starting from left to right. When called, it tries to locate an item of the appropriate type in the input data, skipping whitespace characters (the space, Horizontal and vertical tab, form-feed, and new-line character)

- `scanf("%d%f",&i,&j);`
Ordinary Characters in Format String

• White-space characters: One white-space character in the format string will match any number of white-space character in the input

• Other character: when it encounters a non-white-space character in a format string, scanf compares it with the next input character. If the two characters match, scanf discards the input character and continues processing the format string. Otherwise, scanf puts the offending character back into the input, then aborts without further processing.
Ordinary Characters in Format String

- `%d/%d` will match _5/_96, but not _5_/_96
- `%d_/%d` will match _5_/_96

```c
#include<stdio.h>
int main(){
    int i,j;
    i=100;
    j=100;
    scanf("%d/%d", &i, &j);
    printf("%d\n%d\n", i, j);
}
```
Expressions

- Arithmetic operator: +, -, *, /, %, ++, --, ......
- Relational operator: <, >, <=, >=, !=
- Logical operator: &&, ||
Operator Precedence and Associativity

- \(- i \ast - j = (- i) \ast (- j)\)

- \(+ i + j / k = (+ i) + (j / k)\)

- left/right associative: it groups from left/right to right/left

- The binary arithmetic operators(*,/,%,+ and -) are all left associative

- \(i - j - k = (i - j) - k\) \hspace{50pt} \(i \ast j / k = (i \ast j) / k\)

- The unary arithmetic operators (+ and -) are both right associative

- \(- + i = - (+ i)\)
## Expression Evaluation

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Name</th>
<th>Symbol(s)</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X++/X- -</td>
<td></td>
<td>left</td>
</tr>
<tr>
<td>2</td>
<td>++X/- -X</td>
<td></td>
<td>right</td>
</tr>
<tr>
<td>3</td>
<td>unary +/-</td>
<td>* , / , %</td>
<td>left</td>
</tr>
<tr>
<td>4</td>
<td>additive</td>
<td>+ , -</td>
<td>left</td>
</tr>
<tr>
<td>5</td>
<td>assignment</td>
<td>*= , /= , += , -= , =</td>
<td>right</td>
</tr>
</tbody>
</table>
Expression Evaluation

• \[ a = b += c++ - d + - - e / - f \]
• \[ a = b += (c++) - d + - - e / - f \]
• \[ a = b += (c++) - d + (- - e) / - f \]
• \[ a = b += (c++) - d + (- - e) / (- f) \]
• \[ a = b += (c++) - d + ((- - e) / (- f)) \]
• \[ a = b += ((c++) - d) + ((- - e) / (- f)) \]
• \[ a = b += (((c++) - d) + ((- - e) / (- f))) \]
• \[ a = (b += (((c++) - d) + ((- - e) / (- f)))) \]
• \[ (a = (b += (((c++) - d) + ((- - e) / (- f)))) \]}
Bitwise Operations

- Applied to char, int, short, and long
  - And &
  - Or |
  - Exclusive Or ^
  - Left-shift <<
  - Right-shift >>
  - one’s complement ~
Example: Bit Count

```c
/*
 * count the 1 bits in a number
 * e.g. bitcount (0x45)(01000111 binary) returns 3
 */

int bitcount (unsigned int x) {
    int b;
    for(b = 0;x != 0;x = x >> 1)
        if (x & 01) b++;
    return b;
}
```
Conditional Expressions

- Conditional expressions
  - `expr1?expr2:expr3;`
  - `if expert is true then expert else expr3`
  - `for (i=0;i<n;i++) printf("%c",(i%10==9||i==(n-1))?\'\n\': '));`
Control Flow

- blocks:{...}
- **if** (expr) statement;
- **if** (expr) statement1 **else** statement2;
- **switch** (expr) {**case**…**default**}
- **while** (expr) statement;
- **for** (expr1;expr2;expr3) statement;
- **do** statement **while** (expr);
- **break**; **continue**; (only for loops)
- goto label; (don’t use it)
Scope Rules

• Automatic/Local Variable
  • Declared at the beginning of functions
  • Scope is the function body

• External/Global Variables
  • Declared outside functions
  • Scope is from the point where they are declared until the end of file (unless prefixed by extern)
Scope Rules

• Variables can be declared within blocks too

• scope is until end of the block

```c
{
    int block_variable;
}
block_variable = 9; (wrong  )
```
Scope Rules

- Static Variables: use static prefix on functions and variable declarations to limit scope

  - static prefix on external variables will limit scope to the rest of the source file (not accessible in other files).

  - static prefix on functions will make them invisible to other files.

  - static prefix on internal variables will create permanent private storage; retained even upon function exit.