CMSC 246 Systems Programming

Spring 2018
Bryn Mawr College
Instructor: Deepak Kumar

Input

- `scanf()` is the C library’s counterpart to `printf`.
- Syntax for using `scanf()`

  scanf(<***format-string>***, <***variable-reference(s)***>)

- Example: read an integer value into an `int` variable `data`.

```
scanf("%d", &data); //read an integer; store into data
```

- The `&` is a reference operator. More on that later!

Reading Input

- Reading a `float`:

```
scanf("%f", &x);
```

- `"%f"` tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn't have to).

Standard Input & Output Devices

- In Linux the standard I/O devices are, by default, the keyboard for input, and the terminal console for output.
- Thus, input and output in C, if not specified, is always from the standard input and output devices. That is,

  `printf()` always outputs to the terminal console

  `scanf()` always inputs from the keyboard

- Later, you will see how these can be reassigned/redirected to other devices.

Program: Convert Fahrenheit to Celsius

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

- The program will allow temperatures that aren’t integers.

Program: Convert Fahrenheit to Celsius `ctof.c`

```
#include <stdio.h>

int main(void)
{
  float f, c;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &f);

  c = (f - 32) * 5.0/9.0;

  printf("Celsius equivalent: %.1f\n", c);

  return 0;
} // main()
```

Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

Improving ctof.c

Look at the following command:

```
c = (f - 32) * 5.0/9.0;
```

First, 32, 5.0, and 9.0 should be floating point values: 32.0, 5.0, 9.0

Second, by default, in C, they will be assumed to be of type `double`
Instead, we should write

```
c = (f - 32.0f) * 5.0f/9.0f;
```

What about using constants/magic numbers?

Defining constants - macros

```
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)
```

So we can write:

```
c = (f - FREEZING_PT) * SCALE_FACTOR;
```

When a program is compiled, the preprocessor replaces each macro by the value that it represents.

During preprocessing, the statement

```
c = (f - FREEZING_PT) * SCALE_FACTOR;
```

will become

```
c = (f - 32.f) * (5.0f/9.0f);
```

This is a safer programming practice.

Program: Convert Fahrenheit to Celsius `ctof.c`

```
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)

int main(void)
{
  float f, c;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &f);

  c = (f - FREEZING_PT) * SCALE_FACTOR;

  printf("Celsius equivalent: %.1f\n", c);

  return 0;
} // main()
```

Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

Identifiers

- Names for variables, functions, macros, etc. are called ***identifiers.***
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

  ```
  times10  get_next_char  _done
  ```

  It’s usually best to avoid identifiers that begin with an underscore.
- Examples of illegal identifiers:

  ```
  10times  get-next-char
  ```

Identifiers

- C is ***case-sensitive:*** it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

```
job  joB  jOb  jOB  Job  JoB  JOb  JOB
```

- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

```
symbol_table  current_page  name_and_address
```

- Other programmers use an upper-case letter to begin each word within an identifier:

```
symbolTable  currentPage  nameAndAddress
```

- C places no limit on the maximum length of an identifier.

Keywords

- The following ***keywords*** can't be used as identifiers:

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

- Keywords (with the exception of `_Bool`, `_Complex`, and `_Imaginary`) must be written using only lower-case letters.
- Names of library functions (e.g., `printf`) are also lower-case.

If and Switch statements in C

- A compound statement has the form

```
{ statements }
```

- In its simplest form, the `if` statement has the form

```
if ( expression ) compound/statement
```

- An `if` statement may have an `else` clause:

```
if ( expression ) compound/statement else compound/statement
```

- Most common form of the `switch` statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

Arithmetic Operators

- C provides five binary ***arithmetic operators:***

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	remainder

- An operator is ***binary*** if it has two operands.
- There are also two ***unary*** arithmetic operators:

`+`	unary plus
`-`	unary minus

Logical Expressions

- Several of C’s statements must test the value of an expression to see if it is “true” or “false.”
- In many programming languages, an expression such as `i < j` would have a special “Boolean” or “logical” type.
- In C, a comparison such as `i < j` yields an integer: either 0 (false) or 1 (true).

Relational Operators

- C’s ***relational operators:***

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

- C provides two ***equality operators:***

==	equal to
!=	not equal to

- More complicated logical expressions can be built from simpler ones by using the ***logical operators:***

!	logical negation
&&	logical *and*

These operators produce 0 (false) or 1 (true) when used in expressions.

Logical Operators

- Both `&&` and `||` perform “short-circuit” evaluation: they first evaluate the left operand, then the right one.
- If the value of the expression can be deduced from the left operand alone, the right operand isn’t evaluated.
- Example:

```
(i != 0) && (j / i > 0)
```

  `(i != 0)` is evaluated first. If `i` isn’t equal to 0, then `(j / i > 0)` is evaluated.
- If `i` is 0, the entire expression must be false, so there’s no need to evaluate `(j / i > 0)`. Without short-circuit evaluation, division by zero would have occurred.

Relational Operators & Lack of Boolean Watch out!!!

- The expression

```
i < j < k
```

  is legal, but does not test whether `j` lies between `i` and `k`.
- Since the `<` operator is left associative, this expression is equivalent to

```
(i < j) < k
```

  The 1 or 0 produced by `i < j` is then compared to `k`.
- The correct expression is `i < j && j < k`.

Loops

- The `while` statement has the form

  `while (` *expression* `)` *statement*
- General form of the `do` statement:

  `do` *statement* `while (` *expression* `) ;`
- General form of the `for` statement:

  `for (` *expr1* `;` *expr2* `;` *expr3* `)` *statement*

  expr1, *expr2*, and *expr3* are expressions.
- Example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

- In C99, the first expression in a `for` statement can be replaced by a declaration.
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
  …
```

The **`printf`** Function

- The `printf` function must be supplied with a ***format string,*** followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`,` *expr*`1,` *expr*`2,` …`);`
- The format string may contain both ordinary characters and ***conversion specifications,*** which begin with the `%` character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - `%d` is used for `int` values
  - `%f` is used for `float` values

The **printf** Function

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

The **printf** Function

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Too many conversion specifications:

```
printf("%d %d\n", i);    /*** WRONG ***/
```

- Too few conversion specifications:

```
printf("%d\n", i, j);    /*** WRONG ***/
```

- If the programmer uses an incorrect specification, the program will produce meaningless output:

```
printf("%f %d\n", i, x);   /*** WRONG ***/
```

tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence.***
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

Alert (bell)	`\a`
Backspace	`\b`
New line	`\n`
Horizontal tab	`\t`

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Escape Sequences

- Another common escape sequence is `\"`, which represents the `"` character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single `\` character, put two `\` characters in the string:

```
printf("\\");
  /* prints one \ character */
```

The **scanf** Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

The **scanf** Function

- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively.

How **scanf** Works

- As it searches for a number, `scanf` ignores ***white-space characters*** (space, horizontal and vertical tab, form-feed, and new-line).
- A call of `scanf` that reads four numbers:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over several lines:

```
   1
-20   .3
   -4.0e3
```

- `scanf` sees a stream of characters (¤ represents new-line):

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```
(`s` = skipped; `r` = read)

- `scanf` “peeks” at the final new-line without reading it.

How **scanf** Works

- Sample input:

  `1-20.3-4.0e3¤`

- The call of `scanf` is the same as before:

  `scanf("%d%d%f%f", &i, &j, &x, &y);`

- Here's how `scanf` would process the new input:
  - `%d`. Stores 1 into `i` and puts the − character back.
  - `%d`. Stores –20 into `j` and puts the `.` character back.
  - `%f`. Stores 0.3 into `x` and puts the − character back.
  - `%f`. Stores –4.0 × 103 into `y` and puts the new-line character back.

Ordinary Characters in Format Strings

- When it encounters one or more white-space characters in a format string, `scanf` reads white-space characters from the input until it reaches a non-white-space character (which is "put back").
- When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character.
  - If they match, `scanf` discards the input character and continues processing the format string.
  - If they don't match, `scanf` puts the offending character back into the input, then aborts.

Ordinary Characters in Format Strings

- Examples:
  - If the format string is `"%d/%d"` and the input is `•5/•96`, `scanf` succeeds.
  - If the input is `•5•/•96`, `scanf` fails, because the `/` in the format string doesn't match the space in the input.
- To allow spaces after the first number, use the format string `"%d /%d"` instead.

Program: Adding Fractions

- The `addfrac.c` program prompts the user to enter two fractions and then displays their sum.
- Sample program output:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n",result_num, result_denom)

  return 0;
}
```

Acknowledgements

Some content from these slides is based on the book, C Programming – A Modern Approach, By K. N. King, 2nd Edition, W. W. Norton 2008.

Some content is also included from the lecture slides provided by Prof. K. N. King. Thank You!