Review of earlier lectures

The General Form of a Simple Program

```
directives

int main(void)
{
    statements
}
```

- C uses { and } in much the same way that some other languages use words like begin and end.
- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements

Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

  ```
  #include <stdio.h>
  ```

- `<stdio.h>` is a *header* containing information about C's standard I/O library.
- Directives always begin with a # character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

Statements

- A ***statement*** is a command to be executed when the program runs.
- The program pun.c

  ```
  #include <stdio.h>
  int main(void)
  {
    printf("To C, or not to C.\n");
    return 0;
  }
  ```

  uses only two kinds of statements. One is the return statement; the other is the *function call.*
- Asking a function to perform its assigned task is known as *calling* the function.
- pun.c calls printf to display a string:

  ```
  printf("To C, or not to C.\n");
  ```

- C requires that each statement end with a semicolon.
- Directives are normally one line long, and they don't end with a semicolon.

In the example code given earlier

- When the `printf` function displays a *string literal*—characters enclosed in double quotation marks—it doesn't show the quotation marks.
- `printf` doesn't automatically advance to the next output line when it finishes printing.
- To make `printf` advance one line, include `\n` (the *new-line character*) in the string to be printed.
- The statement

  ```
  printf("To C, or not to C:
         that is the question.\n");
  ```

  is incorrect. But could be corrected by two calls of printf:

  ```
  printf("To C, or not to C: ");
  printf("that is the question.\n");
  ```

Week Two

Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a return statement to specify what value it "returns":

  ```
  return x + 1;
  ```

- **main** is special and mandatory: it gets called automatically when the program is executed.
- main returns a status code; the value 0 indicates normal program termination.
- If there's no return statement at the end of the main function, many compilers will produce a warning message.

Comments

- A *comment* begins with /* and end with */.

  ```
  /* This is a comment */
  ```

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- In C99, comments can also be written in the following way:

  ```
  // This is a comment
  ```

- This style of comment ends automatically at the end of a line.
- Advantages of // comments:
  - Safer: there's no chance that an unterminated comment will accidentally consume part of a program.
  - Multiline comments stand out better.

Variables and Assignment

- Most programs need to a way to store data temporarily during program execution.
- These storage locations are called ***variables.***
- Every variable must have a ***type.***
- C has a wide variety of types, including int and float.
- Variables must be declared before they can be used.
- When main contains declarations, these must precede statements:

  ```
  int main(void)
  {
   declarations
   statements
  }
  ```

- In C99, declarations don't have to come before statements.
- A variable can be given a value by means of ***assignment***, e.g.

  ```
  height = 8;
  ```

- An int variable is normally assigned a value of type int, and a float variable is normally assigned a value of type float.
- Once a variable has been assigned a value, it can be used to help compute the value of another variable:

  ```
  height = 8;
  length = 12;
  width = 10;
  volume = height * length * width;
  /* volume is now 960 */
  ```

- The right side of an assignment can be a formula (or ***expression,*** in C terminology) involving constants, variables, and operators.

- The initial value of a variable may be included in its declaration:

  ```
  int height = 8;
  ```

  The value 8 is said to be an ***initializer.***

- Any number of variables can be initialized in the same declaration:

  ```
  int height = 8, length = 12;
  ```

- Each variable requires its own initializer.

  ```
  int height, length, width = 10;
    /* initializes only width */
  ```

Identifiers

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

  ```
  times10  get_next_char  _done
  ```

- Examples of illegal identifiers:

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

- 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
  ```

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

- 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
```

* indicates C99 only

Chapter 3 Formatted Input/Output

- 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(*format string, expr1, expr2,*…);

- 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
- 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
```

- 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 **/
```

- Compilers aren’t required to check that a conversion specification is appropriate.
- If the programmer uses an incorrect specification, the program will produce meaningless output:

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

Conversion Specifications

An example C program that 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    |
```

- The d specifier is used to display an integer in decimal form.
- Conversion specifiers for floating-point numbers:

  e — Exponential format.

  f — “Fixed decimal” format.

  g — Either exponential format or fixed decimal format, depending on the number’s size. *p* indicates the maximum number of significant digits to be displayed. The g conversion won’t show trailing zeros. If the number has no digits after the decimal point, g doesn’t display the decimal point.

- A conversion specification can have the form %*m*.*pX* or %-*m*.*pX*, where *m* and *p* are integer constants and *X* is a letter.

  m --- the ***minimum field width***, specifies the minimum number of characters to print.

  If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size.

  p --- for int, *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary)

  --- for float with %e or %f, indicates how many digits should appear after the decimal point (the default is 6).

  If the value to be printed requires more than *m* characters, the ***minimum field width***, the field width automatically expands to the necessary size.

- In the conversion specification %10.2f, *m* is 10, *p* is 2, and *X* is f. It means that it is in a format for a float number with space of 10 characters and 2 digits after the decimal point.

  %4d displays the number 123 as _123. (_ represents the space character.)

- Putting a minus sign in front of *m* causes left justification.

  The specification %-4d would display 123 as 123_

Escape Sequences

- Escape sequences enable strings to contain control characters and special characters.

- A partial list of escape sequences:

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

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.

- 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.

- When using scanf, must check that
  - the number of conversion specifications matches the number of input variables and
  - each conversion is appropriate for the corresponding variable.

- Another trap involves the & symbol, which normally precedes each variable in a scanf call.