Chapter 2

C Fundamentals

Program: Printing a Pun

```
#include <stdio.h>

int main(void)
{
  printf("To C, or not to C: that is the question.\n");
  return 0;
}
```

- This program might be stored in **a file named `pun.c`**.
- The file name doesn’t matter, but the `.c` extension is often required.

Compiling and Linking

- Before a program can be executed, three steps are usually necessary:
  - ***Preprocessing.*** The ***preprocessor*** obeys commands that begin with # (known as ***directives***)
  - ***Compiling.*** A ***compiler*** then translates the program into machine instructions (***object code***).
  - ***Linking.*** A ***linker*** combines the object code produced by the compiler with any additional codes needed to yield a complete executable program. Ex. /usr/lib/x86_64-linux-gnu/libc.so and libm.so
- Preprocessor is usually integrated with compiler.

- ldd --version
  - ldd (Debian EGLIBC 2.13-38+deb7u12) 2.13
- ldd /usr/bin/touch
  - In /lib/x86_64-linux-gnu/
    - libc.so.6 -> libc-2.13.so (-> logic link)
- gcc –lm
  - Linked with libm.so.6 -> libm-2.13.so

Compiling and Linking Using **cc**

- To compile and link the `pun.c` program under UNIX, enter the following command in a terminal or command-line window:

  ```
  % cc pun.c (gcc pun.c|gcc –o pun pun.c)
  ```

  The `%` character is the UNIX prompt.
- Linking is automatic when using `cc`; no separate link command is necessary.

Compiling and Linking Using **cc**

- After compiling and linking the program, `cc` leaves the executable program in a file named `a.out` by default.
- The `-o` option lets us choose the name of the file containing the executable program.
- The following command causes the executable version of `pun.c` to be named `pun`:

```
% cc -o pun pun.c
```

The GCC Compiler

- GCC is one of the most popular C compilers.
- GCC is supplied with Linux but is available for many other platforms as well.
- Using this compiler is similar to using `cc`:

```
% gcc -o pun pun.c
```

Integrated Development Environments

- An ***integrated development environment (IDE)*** is a software package that makes it possible to edit, compile, link, execute, and debug a program without leaving the environment.

The General Form of a Simple Program

- Simple C programs have the form

```
directives

int main(void)
{
   statements
}
```

The General Form of a Simple Program

- 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: (see p.351)

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

- `<stdio.h>` is a ***header*** containing information about C's standard I/O library.

Directives

- Directives always begin with a # character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.

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;
```

The `main` Function

- The `main` function is mandatory.
- `main` is special: 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.

Statements

- A ***statement*** is a command to be executed when the program runs.
- `pun.c` 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: that is the
question.\n");
```

Statements

- C requires that each statement end with a semicolon (;).
  - There's one exception: the compound statement.
- Directives are normally one line long, and they don't end with a semicolon.

Printing Strings

- 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.

Printing Strings

- The statement

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

could be replaced by two calls of `printf`:

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

- The new-line character can appear more than once in a string literal:

```
printf("Brevity is the soul of wit.\n
--Shakespeare\n");
```

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.
- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

Comments

- *Warning:* Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");    /* forgot to close this comment...
printf("cat ");
printf("has ");   /* so it ends here */
printf("fleas");
```

Comments in C99

- 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.***

Types

- Every variable must have a ***type.***
- C has a wide variety of types, including `int` and `float`.
- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553.
  - The largest `int` value is typically 2,147,483,647 but can be as small as 32,767.

Types

- A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable.
- Also, a `float` variable can store numbers with digits after the decimal point, like 379.125.
- Drawbacks of `float` variables:
  - Slower arithmetic
  - Approximate nature of `float` values

Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

Declarations

- When `main` contains declarations, these must precede statements:

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

- In C99, declarations don’t have to come before statements.

Assignment

- A variable can be given a value by means of ***assignment:***

  ```
  height = 8;
  ```

  The number `8` is said to be a ***constant.***
- Before a variable can be assigned a value—or used in any other way—it must first be declared.

Assignment

- A constant assigned to a `float` variable usually contains a decimal point:

```
profit = 2150.48;
```

- It's best to append the letter `f` to a floating-point constant if it is assigned to a `float` variable:

```
profit = 2150.48f;
```

  Failing to include the `f` may cause a warning from the compiler.

Assignment

- An `int` variable is normally assigned a value of type `int`, and a `float` variable is normally assigned a value of type `float`.
- Mixing types (such as assigning an `int` value to a `float` variable or assigning a `float` value to an `int` variable) is possible but not always safe.

Assignment

- 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.

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the message

  `Height:` *h*

  where *h* is the current value of the `height` variable, we'd use the following call of `printf`:

  ```
  printf("Height: %d\n", height);
  ```
- `%d` is a placeholder indicating where the value of `height` is to be filled in.

Printing the Value of a Variable

- `%d` works only for `int` variables; to print a `float` variable, use `%f` instead.
- By default, `%f` displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
- To print the line

  ```
  Profit: $2150.48
  ```

  use the following call of `printf`:

  ```
  printf("Profit: $%.2f\n", profit);
  ```

Printing the Value of a Variable

- There's no limit to the number of variables that can be printed by a single call of `printf`:

```
printf("Height: %d  Length: %d\n", height, length);
```

Program: Computing the Dimensional Weight of a Box

- Shipping companies often charge extra for boxes that are large but very light, basing the fee on volume instead of weight.
- The usual method to compute the "dimensional weight" is to divide the volume by 166 (the allowable number of cubic inches per pound).
- The `dweight.c` program computes the dimensional weight of a particular box:

```
Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

Program: Computing the Dimensional Weight of a Box

- Division is represented by `/` in C, so the obvious way to compute the dimensional weight would be

```
weight = volume / 166;
```

- In C, however, when one integer is divided by another, the answer is "truncated": all digits after the decimal point are lost.
  - The volume of a 12" × 10" × 8" box will be 960 cubic inches.
  - Dividing by 166 gives 5 instead of 5.783.

Program: Computing the Dimensional Weight of a Box

- One solution is to add 165 to the volume before dividing by 166:

```
weight = (volume + 165) / 166;
```

- A volume of 166 would give a weight of 331/166, or 1, while a volume of 167 would yield 332/166, or 2.

dweight.c

```
/* Computes the dimensional weight of a 12" x 10" x 8" box */

#include <stdio.h>

int main(void)
{
  int height, length, width, volume, weight;

  height = 8;
  length = 12;
  width = 10;
  volume = height * length * width;
  weight = (volume + 165) / 166;

  printf("Dimensions: %dx%dx%d\n", length, width, height);
  printf("Volume (cubic inches): %d\n", volume);
  printf("Dimensional weight (pounds): %d\n", weight);

  return 0;
}
```

Initialization

- Some variables are automatically set to zero when a program begins to execute, but most are not.
- A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be ***uninitialized.***
- Attempting to access the value of an uninitialized variable may yield an unpredictable result.
- With some compilers, worse behavior—even a program crash—may occur.

Initialization

- 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, width = 10;
  ```

- Each variable requires its own initializer.

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

Printing Expressions

- `printf` can display the value of any numeric expression.
- The statements

```
volume = height * length * width;
printf("%d\n", volume);
```

could be replaced by

```
printf("%d\n", height * length * width);
```

Reading Input

- `scanf` is the C library's counterpart to `printf`.
- `scanf` requires a ***format string*** to specify the appearance of the input data.
- Example of using `scanf` to read an `int` value:

```
scanf("%d", &i);
/* reads an integer; stores into i */
```

- The & symbol is usually (but not always) required when using `scanf`.

Reading Input

- Reading a `float` value requires a slightly different call of `scanf`:

```
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).

Program: Computing the Dimensional Weight of a Box (Revisited)

- `dweight2.c` is an improved version of the dimensional weight program in which the user enters the dimensions.
- Each call of `scanf` is immediately preceded by a call of `printf` that displays a ***prompt.***

dweight2.c

```
/* Computes the dimensional weight of a box from input provided by the user */

#include <stdio.h>

int main(void)
{
  int height, length, width, volume, weight;

  printf("Enter height of box: ");
  scanf("%d", &height);
  printf("Enter length of box: ");
  scanf("%d", &length);
  printf("Enter width of box: ");
  scanf("%d", &width);
  volume = height * length * width;
  weight = (volume + 165) / 166;

  printf("Volume (cubic inches): %d\n", volume);
  printf("Dimensional weight (pounds): %d\n", weight);

  return 0;
}
```

Program: Computing the Dimensional Weight of a Box (Revisited)

- Sample output of program:

```
Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

- Note that a prompt shouldn't end with a new-line character.

Defining Names for Constants

- `dweight.c` and `dweight2.c` rely on the constant 166, whose meaning may not be clear to someone reading the program.
- Using a feature known as ***macro definition,*** we can name this constant:

```
#define INCHES_PER_POUND 166
```

Defining Names for Constants

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

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

Defining Names for Constants

- The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

- If it contains operators, the expression should be enclosed in parentheses.
- Using only upper-case letters in macro names is a common convention.

Program: Converting from 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.

celsius.c

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

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

int main(void)
{
  float fahrenheit, celsius;

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

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

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

  return 0;
}
```

Program: Converting from Fahrenheit to Celsius

- Defining `SCALE_FACTOR` to be `(5.0f / 9.0f)` instead of `(5 / 9)` is important.
- Note the use of `%.1f` to display `celsius` with just one digit after the decimal point.

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
  ```

  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
```

Identifiers

- 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
```

*C99 only

Keywords

- 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.

Layout of a C Program

- A C program is a series of ***tokens.***
- Tokens include:
  - Identifiers
  - Keywords
  - Operators
  - Punctuation
  - Constants
  - String literals

Layout of a C Program

- The statement

  ```
  printf("Height: %d\n", height);
  ```

  consists of seven tokens:

`printf`	Identifier
`(`	Punctuation
`"Height: %d\n"`	String literal
`,`	Punctuation
`height`	Identifier
`)`	Punctuation
`;`	Punctuation

Layout of a C Program

- The amount of space between tokens usually isn’t critical.
- At one extreme, tokens can be crammed together with no space between them, except where this would cause two tokens to merge:

```
/* Converts a Fahrenheit temperature to Celsius */
#include <stdio.h>
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)
int main(void){float fahrenheit,celsius;printf(
"Enter Fahrenheit temperature: ");scanf("%f", &fahrenheit);
celsius=(fahrenheit-FREEZING_PT)*SCALE_FACTOR;
printf("Celsius equivalent: %.1f\n", celsius);return 0;}
```

Layout of a C Program

- The whole program can't be put on one line, because each preprocessing directive requires a separate line.
- Compressing programs in this fashion isn't a good idea.
- In fact, adding spaces and blank lines to a program can make it easier to read and understand.

Layout of a C Program

- C allows any amount of space—blanks, tabs, and new-line characters—between tokens.
- Consequences for program layout:
  - *Statements can be divided* over any number of lines.
  - *Space between tokens* (such as before and after each operator, and after each comma) makes it easier for the eye to separate them.
  - *Indentation* can make nesting easier to spot.
  - *Blank lines* can divide a program into logical units.

Layout of a C Program

- Although extra spaces can be added between tokens, it's not possible to add space within a token without changing the meaning of the program or causing an error.
- Writing

```
fl oat fahrenheit, celsius;  /*** WRONG ***/
```

or

```
fl
oat fahrenheit, celsius;     /*** WRONG ***/
```

produces an error when the program is compiled.

Layout of a C Program

- Putting a space inside a string literal is allowed, although it changes the meaning of the string.
- Putting a new-line character in a string (splitting the string over two lines) is illegal:

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

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- This may include a file name or a switch that modifies the program's behavior.
- Examples of the UNIX `ls` command:

```
ls
ls –l
ls -l remind.c
```

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
   …
}
```

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments

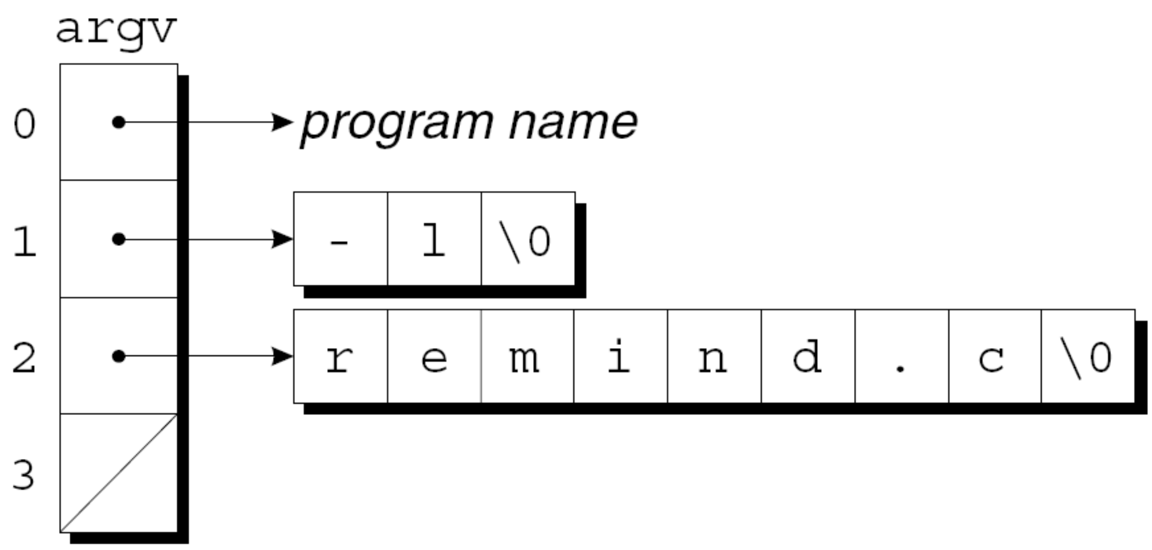
- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `argv[argc]` is always a ***null pointer***—a special pointer that points to nothing.
  - The macro `NULL` represents a null pointer.

Command-Line Arguments

- If the user enters the command line

  `ls -l remind.c`

  then `argc` will be 3, and `argv` will have the following appearance:



Command-Line Arguments

- Since `argv` is an array of pointers, accessing command-line arguments is easy.
- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn.
- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments.
- The program is designed to check a series of strings to see which ones are names of planets.
- The strings are put on the command line:

```
planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

planet.c

```
/* Checks planet names */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
```

```
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```