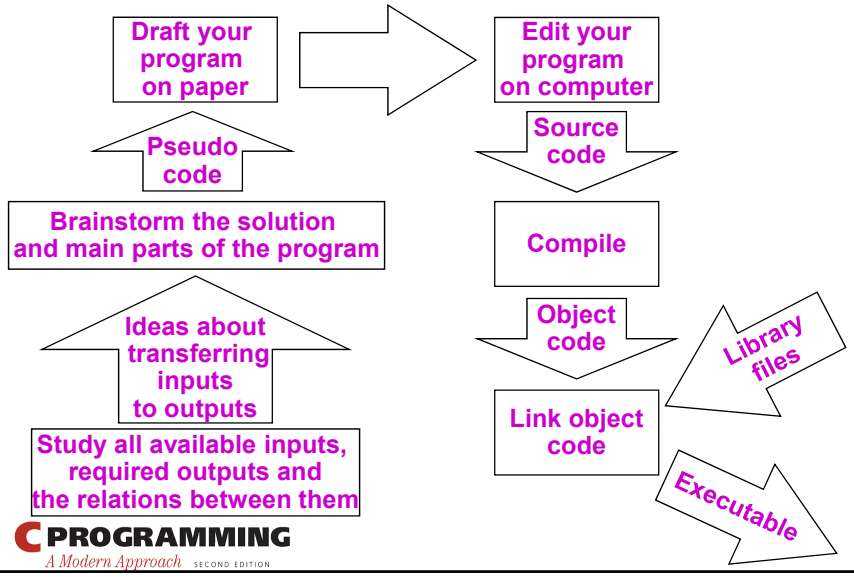
Chapter 2

C Fundamentals

The C Development Cycle



The General Form of a Simple Program

- Simple C programs have the form

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

- Every C programs rely on three key language features:
  - ***Directives***
  - ***Functions***
  - ***Statements***

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*** translates the program into machine instructions (***object code***).
  - ***Linking***
    - A ***linker*** combines the object code produced by the compiler with any additional code needed to yield a complete executable program
- *The preprocessor and linker are usually integrated within the compiler*

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 Using `cc`

- To compile and link the `pun.c` program under Unix, enter the following command in a terminal at command-line prompt:

```
cc pun.c
```

- Preprocessing and linking are automatic when using `cc`

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 and is available for many other platforms as well
- Using this compiler is similar to using `cc`:

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

Directives

- Before a C program is compiled, it is first edited by a preprocessor
- Commands intended for the preprocessor are called directives
  - Directives always begin with a `#` character
- Example:

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

- `<stdio.h>` is a ***header*** containing information about C's standard I/O library
- 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 a special function
  - it gets called automatically when the program is executed
- `main` returns a status code
  - Same as in Unix shell script
  - 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
- C requires that each statement end with a semicolon

Statements

```
#include <stdio.h>

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

- `pun.c` main function includes only two statements
  - one is the `printf` ***function call***
  - the other is the `return` statement
- Asking a function to perform its task is known as ***calling*** the function
- `pun.c` calls `printf` to display a string

Printing Strings

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

```
printf("To C, or not to C: that is the question.\n");
printf("To C, or not to C: ");
printf("that is the question.\n");
printf("Brevity is the soul of wit.\n  --Shakespeare\n");
```

Comments

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

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

- Comments may extend over more than one line

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

- *Warning*: Forgetting to terminate a comment may cause the compiler to ignore part of your program
- Comments ***can not*** be nested ***(why?)***
- In `C99`, comments can also be written in the following way: ***(why?)***

```
// This is a comment ended at the end of line
```


Variables

- Most programs need 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`
  - *Short for what?*
  - *Speed*
  - *Approximate nature of 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` function contains declarations, these must precede any statements

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

- In **`C99`**, declarations don't have to come before statements
  - Yet, variable must be ***declared*** before they are used

Assignment

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

```
height = 8;
```

  The number `8` is said to be a ***constant***
- 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

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable

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

- `%d` is a *placeholder* indicating where the value of the variable `height` is to be filled in
- `%d` works only for `int` variables
- To print a `float` variable, use `%f` instead

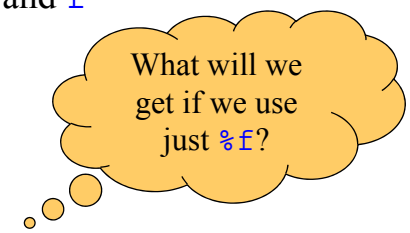
Printing the Value of a Variable

- 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:

```
profit = 2150.48f;
printf("Profit: $%.2f\n", profit);
```



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 12″×10″ ×8″ box

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

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;     /* Why did we use 165 here?*/

  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

- 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 */
  ```

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

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

`i` must be an `int` variable

- When using `scanf`, the `&` symbol is usually (but not always) required (*depends of the type of the used variable*)

Reading Input

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

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

`x` must be a `float` variable

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

  return 0;
}
```

There is no *placeholder* here. Hence, only a constant string will be printed.

There is no `"\n"`

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

- Sample output of the program:

```
Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Dimensions: 12x10x8
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
```

There is no "=" sign here

- Using only upper-case letters in macro names is a *common convention, but not a must*

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;
```


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*
- Examples of *legal* identifiers

```
times10   get_next_char   _done
```

- Examples of *illegal* identifiers

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

- It's usually best to avoid identifiers that begin with an *underscore*

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
```

Red means C99 only

Layout of a C Program

- A C program is a series of ***tokens***
- Tokens include:
  - Identifiers
  - Keywords
  - Operators
  - Punctuations
  - 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
- *Can we put any C program in one line? Why?*
- 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
  - *Statements can be divided* over any number of lines
  - *Space between tokens* (e.g., 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
- Spaces are not allowed within a token, since they will change its meaning

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 ***/

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