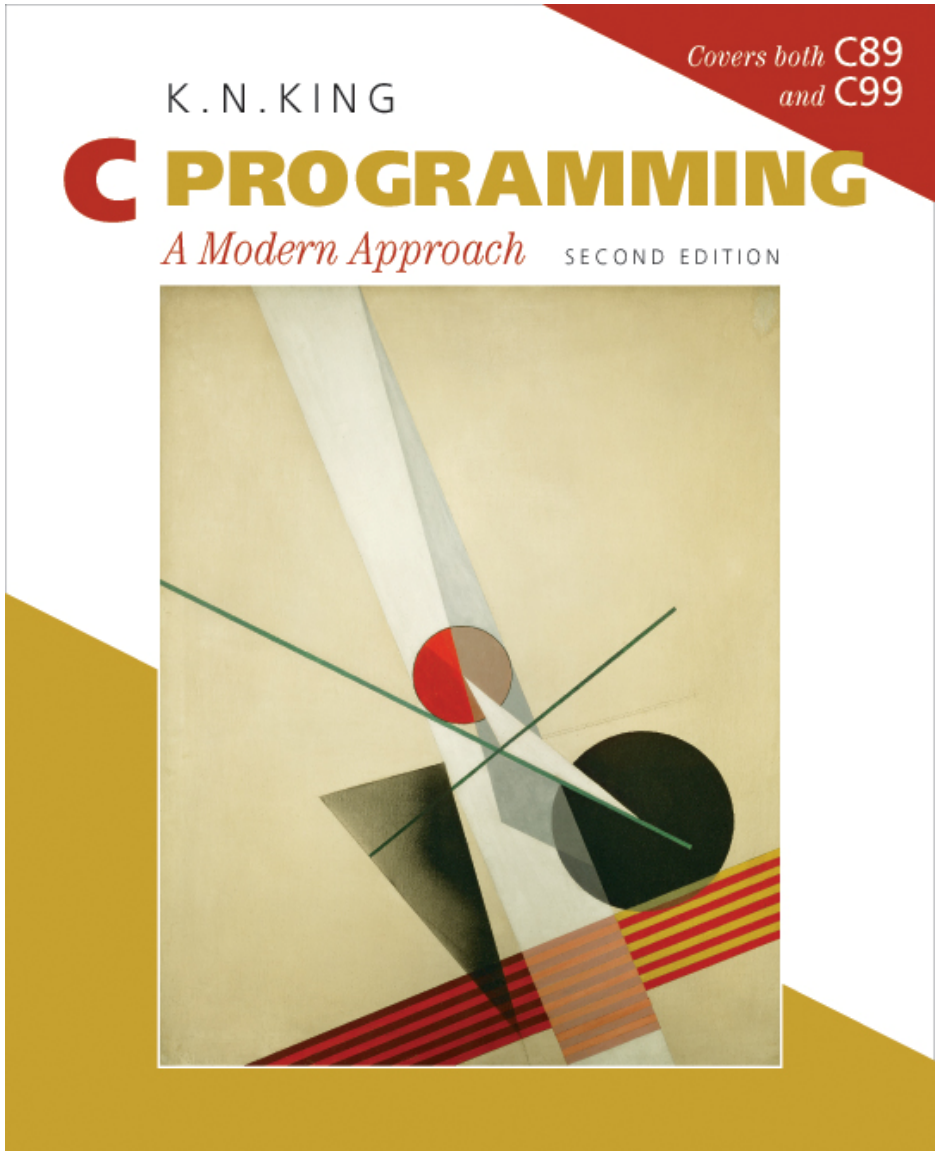


程式設計

csie.ncyu@2009

Chapter 1

Introducing C

Origins of C

- C is a by-product of UNIX, developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others.
- Thompson designed a small language named B.
- B was based on BCPL, a systems programming language developed in the mid-1960s.

Origins of C

- By 1971, Ritchie began to develop an extended version of B.
- He called his language NB ("New B") at first.
- As the language began to diverge more from B, he changed its name to C.
- The language was stable enough by 1973 that UNIX could be rewritten in C.

Standardization of C

- *K&R C*
  - Described in Kernighan and Ritchie, *The C Programming Language* (1978)
  - De facto standard
- *C89/C90*
  - ANSI standard X3.159-1989 (completed in 1988; formally approved in December 1989)
  - International standard ISO/IEC 9899:1990
- *C99*
  - International standard ISO/IEC 9899:1999
  - Incorporates changes from Amendment 1 (1995)

C-Based Languages

- ***C++*** includes all the features of C, but adds classes and other features to support object-oriented programming.
- ***Java*** is based on C++ and therefore inherits many C features.
- ***C#*** is a more recent language derived from C++ and Java.
- ***Perl*** has adopted many of the features of C.

Properties of C

- Low-level
- Small
- Permissive

Strengths of C

- Efficiency
- Portability
- Power
- Flexibility
- Standard library
- Integration with UNIX

Weaknesses of C

- Programs can be error-prone.
- Programs can be difficult to understand.
- Programs can be difficult to modify.

Effective Use of C

- Learn how to avoid pitfalls.
- Use software tools (`lint`, debuggers) to make programs more reliable.
- Take advantage of existing code libraries.
- Adopt a sensible set of coding conventions.
- Avoid "tricks" and overly complex code.
- Stick to the standard.

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*** translates then 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 is usually integrated with the compiler.

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
  ```

  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:

```
#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" x 10" x 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 ***/
```

Chapter 3

Formatted Input/Output

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 ***/
```

The `printf` Function

- 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

- A conversion specification can have the form `%`*m*`.`*p*`X` or `%-`*m*`.`*p*`X`, where *m* and *p* are integer constants and *X* is a letter.
- Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped.
- In the conversion specification `%10.2f`, *m* is 10, *p* is 2, and *X* is `f`.
- In the specification `%10f`, *m* is 10 and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is 2 and *m* is missing.

Conversion Specifications

- The ***minimum field width,*** *m*, specifies the minimum number of characters to print.
- If the value to be printed requires fewer than *m* characters, it is right-justified within the field.
  - `%4d` displays the number 123 as •`123`. (• represents the space character.)
- If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size.
- Putting a minus sign in front of *m* causes left justification.
  - The specification `%-4d` would display 123 as `123•`.

Conversion Specifications

- The meaning of the ***precision,*** *p*, depends on the choice of *X*, the ***conversion specifier.***
- The `d` specifier is used to display an integer in decimal form.
  - *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary).
  - If *p* is omitted, it is assumed to be 1.

Conversion Specifications

- Conversion specifiers for floating-point numbers:

  `e` — Exponential format. *p* indicates how many digits should appear after the decimal point (the default is 6). If *p* is 0, no decimal point is displayed.

  `f` — "Fixed decimal" format. *p* has the same meaning as for the `e` specifier.

  `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.

Program: Using `printf` to Format Numbers

- The `tprintf.c` program uses `printf` to display integers and floating-point numbers in various formats.

`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`

Escape Sequences

- A string may contain any number of escape sequences:

```
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.

The `scanf` Function

- When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable.
- Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call.
- The `&` is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

How `scanf` Works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.
- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, skipping blank space if necessary.
- `scanf` then reads the item, stopping when it reaches a character that can't belong to the item.
  - If the item was read successfully, `scanf` continues processing the rest of the format string.
  - If not, `scanf` returns immediately.

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

- When asked to read an integer, `scanf` first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit.
- When asked to read a floating-point number, `scanf` looks for
  - a plus or minus sign (optional), followed by
  - digits (possibly containing a decimal point), followed by
  - an exponent (optional). An exponent consists of the letter `e` (or `E`), an optional sign, and one or more digits.
- `%e`, `%f`, and `%g` are interchangeable when used with `scanf`.

How `scanf` Works

- When `scanf` encounters a character that can't be part of the current item, the character is "put back" to be read again during the scanning of the next input item or during the next call of `scanf`.

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 x 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.

Confusing `printf` with `scanf`

- Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two.
- One common mistake is to put `&` in front of variables in a call of `printf`:

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


Confusing `printf` with `scanf`

- Incorrectly assuming that `scanf` format strings should resemble `printf` format strings is another common error.
- Consider the following call of `scanf`:

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

  - `scanf` will first look for an integer in the input, which it stores in the variable `i`.
  - `scanf` will then try to match a comma with the next input character.
  - If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`.

Confusing `printf` with `scanf`

- Putting a new-line character at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character.
- If the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character.
- A format string like this can cause an interactive program to "hang."

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;
}
```

Chapter 4

Expressions

Operators

- C emphasizes expressions rather than statements.
- Expressions are built from variables, constants, and operators.
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators

  and many others

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

Unary Arithmetic Operators

- The unary operators require one operand:

```
i = +1;
j = -i;
```

- The unary + operator does nothing. It's used primarily to emphasize that a numeric constant is positive.

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`.
  `10 % 3` has the value 1, and `12 % 4` has the value 0.
- Binary arithmetic operators—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed.
- When `int` and `float` operands are mixed, the result has type `float`.
  `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The / and % Operators

- The `/` and `%` operators require special care:
  - When both operands are integers, `/` "truncates" the result. The value of `1 / 2` is 0, not 0.5.
  - The `%` operator requires integer operands; if either operand is not an integer, the program won't compile.
  - Using zero as the right operand of either `/` or `%` causes undefined behavior.
  - The behavior when `/` and `%` are used with negative operands is ***implementation-defined*** in C89.
  - In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`.

Implementation-Defined Behavior

- The C standard deliberately leaves parts of the language unspecified.
- Leaving parts of the language unspecified reflects C's emphasis on efficiency, which often means matching the way that hardware behaves.
- It's best to avoid writing programs that depend on implementation-defined behavior.

Operator Precedence

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.

Operator Precedence

- The arithmetic operators have the following relative precedence:

  Highest: `+ -` (unary)
  `* / %`
  Lowest: `+ -` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`
  `-i * -j` is equivalent to `(-i) * (-j)`
  `+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

  `i - j - k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`

Operator Associativity

- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `-`) are both right associative, so

  `- + i` is equivalent to `-(+i)`

Program: Computing a UPC Check Digit

- Most goods sold in U.S. and Canadian stores are marked with a Universal Product Code (UPC):

- Meaning of the digits underneath the bar code:
  First digit: Type of item
  First group of five digits: Manufacturer
  Second group of five digits: Product (including package size)
  Final digit: Check digit, used to help identify an error in the preceding digits

Program: Computing a UPC Check Digit

- How to compute the check digit:

  Add the first, third, fifth, seventh, ninth, and eleventh digits.
  Add the second, fourth, sixth, eighth, and tenth digits.
  Multiply the first sum by 3 and add it to the second sum.
  Subtract 1 from the total.
  Compute the remainder when the adjusted total is divided by 10.
  Subtract the remainder from 9.

Program: Computing a UPC Check Digit

- Example for UPC 0 13800 15173 5:

  First sum: 0 + 3 + 0 + 1 + 1 + 3 = 8.
  Second sum: 1 + 8 + 0 + 5 + 7 = 21.
  Multiplying the first sum by 3 and adding the second yields 45.
  Subtracting 1 gives 44.
  Remainder upon dividing by 10 is 4.
  Remainder is subtracted from 9.
  Result is 5.

Program: Computing a UPC Check Digit

- The `upc.c` program asks the user to enter the first 11 digits of a UPC, then displays the corresponding check digit:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

- The program reads each digit group as five one-digit numbers.
- To read single digits, we'll use `scanf` with the `%1d` conversion specification.

upc.c

```
/* Computes a Universal Product Code check digit */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

Assignment Operators

- ***Simple assignment:*** used for storing a value into a variable
- ***Compound assignment:*** used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment *v* = *e* is to evaluate the expression *e* and copy its value into *v*.
- *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;              /* i is now 5  */
j = i;              /* j is now 5  */
k = 10 * i + j;     /* k is now 55 */
```


Simple Assignment

- If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;

i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0 */
```


Simple Assignment

- In many programming languages, assignment is a statement; in C, however, assignment is an operator, just like +.
- The value of an assignment *v* = *e* is the value of *v* after the assignment.
  - The value of `i = 72.99f` is 72 (not 72.99).

Side Effects

- An operators that modifies one of its operands is said to have a ***side effect.***
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.

Side Effects

- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The = operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```


Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

- `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

Side Effects

- An assignment of the form *v* = *e* is allowed wherever a value of type *v* would be permitted:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

- "Embedded assignments" can make programs hard to read.
- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand.
- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as `10` or `2 * i` are not.

Lvalues

- Since the assignment operator requires an lvalue as its left operand, it's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;       /*** WRONG ***/
i + j = 0;    /*** WRONG ***/
-i = j;       /*** WRONG ***/
```

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.
- Example:

```
i = i + 2;
```

- Using the += compound assignment operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```


Compound Assignment

- There are nine other compound assignment operators, including the following:

```
-=   *=   /=   %=
```

- All compound assignment operators work in much the same way:

  v `+=` *e* adds *v* to *e*, storing the result in *v*

  v `-=` *e* subtracts *e* from *v*, storing the result in *v*

  v `*=` *e* multiplies *v* by *e*, storing the result in *v*

  v `/=` *e* divides *v* by *e*, storing the result in *v*

  v `%=` *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

Compound Assignment

- *v* += *e* isn't "equivalent" to *v* = *v* + *e*.
- One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`.
- There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect.
- Similar remarks apply to the other compound assignment operators.

Compound Assignment

- When using the compound assignment operators, be careful not to switch the two characters that make up the operator.
- Although `i =+ j` will compile, it is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

Increment and Decrement Operators

- Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1):

```
i = i + 1;
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators:

```
i += 1;
j -= 1;
```


Increment and Decrement Operators

- C provides special `++` (***increment***) and `--` (***decrement***) operators.
- The `++` operator adds 1 to its operand. The `--` operator subtracts 1.
- The increment and decrement operators are tricky to use:
  - They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).
  - They have side effects: they modify the values of their operands.

Increment and Decrement Operators

- Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```


Increment and Decrement Operators

- `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later."
- How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators

- The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);    /* prints "i is 0" */
printf("i is %d\n", i);      /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 0" */
```


Increment and Decrement Operators

- When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.
- Example:

```
i = 1;
j = 2;
k = ++i + j++;
```

The last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators

- In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

Expression Evaluation

- Table of operators discussed so far:

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix)	`++`	left
	decrement (postfix)	`--`	
2	increment (prefix)	`++`	right
	decrement (prefix)	`--`	
	unary plus	`+`	
	unary minus	`-`	
3	multiplicative	`* / %`	left
4	additive	`+ -`	left
5	assignment	`= *= /= %= += -=`	right

Expression Evaluation

- The table can be used to add parentheses to an expression that lacks them.
- Starting with the operator with highest precedence, put parentheses around the operator and its operands.
- Example:

```
a = b += c++ - d + --e / -f
```

	Precedence level
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

Order of Subexpression Evaluation

- The value of an expression may depend on the order in which its subexpressions are evaluated.
- C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical and, logical or, conditional, and comma operators).
- In the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Order of Subexpression Evaluation

- Most expressions have the same value regardless of the order in which their subexpressions are evaluated.
- However, this may not be true when a subexpression modifies one of its operands:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- The effect of executing the second statement is undefined.

Order of Subexpression Evaluation

- Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression.
- Some compilers may produce a warning message such as *"operation on 'a' may be undefined"* when they encounter such an expression.

Order of Subexpression Evaluation

- To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions.
- Instead, use a series of separate assignments:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6.

Order of Subexpression Evaluation

- Besides the assignment operators, the only operators that modify their operands are increment and decrement.
- When using these operators, be careful that an expression doesn't depend on a particular order of evaluation.

Order of Subexpression Evaluation

- Example:

```
i = 2;
j = i * i++;
```

- It's natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:
  1. The second operand (the original value of i) is fetched, then `i` is incremented.
  2. The first operand (the new value of `i`) is fetched.
  3. The new and old values of `i` are multiplied, yielding 6.

Undefined Behavior

- Statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior.***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers.
  - The program may not compile in the first place.
  - If it compiles it may not run.
  - If it does run, the program may crash, behave erratically, or produce meaningless results.
- Undefined behavior should be avoided.

Expression Statements

- C has the unusual rule that any expression can be used as a statement.
- Example:

```
++i;
```

`i` is first incremented, then the new value of `i` is fetched but then discarded.

Expression Statements

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* not useful */
```

Expression Statements

- A slip of the finger can easily create a "do-nothing" expression statement.
- For example, instead of entering

  ```
  i = j;
  ```

  we might accidentally type

  ```
  i + j;
  ```

- Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

Chapter 5

Selection Statements

Statements

- So far, we've used `return` statements and expression statements.
- Most of C's remaining statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break`, `continue`, and `goto`. (`return` also belongs in this category.)
- Other C statements:
  - Compound statement
  - Null statement

Logical Expressions

- Several of C's statements must test the value of an expression to see if it is "true" or "false."
- For example, an `if` statement might need to test the expression `i < j`; a true value would indicate that `i` is less than `j`.
- 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
- These operators produce 0 (false) or 1 (true) when used in expressions.
- The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed.

Relational Operators

- The precedence of the relational operators is lower than that of the arithmetic operators.
  - For example, `i + j < k - 1` means `(i + j) < (k - 1)`.
- The relational operators are left associative.

Relational Operators

- 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`.

Equality Operators

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

  `==` equal to
  `!=` not equal to
- The equality operators are left associative and produce either 0 (false) or 1 (true) as their result.
- The equality operators have lower precedence than the relational operators, so the expression

  ```
  i < j == j < k
  ```

  is equivalent to

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


Logical Operators

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

  `!` logical negation
  `&&` logical *and*
  `||` logical *or*
- The `!` operator is unary, while `&&` and `||` are binary.
- The logical operators produce 0 or 1 as their result.
- The logical operators treat any nonzero operand as a true value and any zero operand as a false value.

Logical Operators

- Behavior of the logical operators:

  !*expr* has the value 1 if *expr* has the value 0.

  expr1 && *expr2* has the value 1 if the values of *expr1* and *expr2* are both nonzero.

  expr1 || *expr2* has the value 1 if either *expr1* or *expr2* (or both) has a nonzero value.
- In all other cases, these operators produce the value 0.

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.

Logical Operators

- Thanks to the short-circuit nature of the `&&` and `||` operators, side effects in logical expressions may not always occur.
- Example:

  ```
  i > 0 && ++j > 0
  ```

  If `i > 0` is false, then `++j > 0` is not evaluated, so j isn't incremented.
- The problem can be fixed by changing the condition to `++j > 0 && i > 0` or, even better, by incrementing `j` separately.

Logical Operators

- The `!` operator has the same precedence as the unary plus and minus operators.
- The precedence of `&&` and `||` is lower than that of the relational and equality operators.
  - For example, `i < j && k == m` means `(i < j) && (k == m)`.
- The `!` operator is right associative; `&&` and `||` are left associative.

The `if` Statement

- The `if` statement allows a program to choose between two alternatives by testing an expression.
- In its simplest form, the `if` statement has the form

  `if ( `*expression*` ) `*statement*
- When an `if` statement is executed, *expression* is evaluated; if its value is nonzero, *statement* is executed.
- Example:

```
if (line_num == MAX_LINES)
  line_num = 0;
```


The `if` Statement

- Confusing == (equality) with = (assignment) is perhaps the most common C programming error.
- The statement

  `if (i == 0) …`

  tests whether `i` is equal to 0.
- The statement

  `if (i = 0) …`

  assigns 0 to `i`, then tests whether the result is nonzero.

The `if` Statement

- Often the expression in an `if` statement will test whether a variable falls within a range of values.
- To test whether $0 \le i < n$:

  `if (0 <= i && i < n) …`
- To test the opposite condition (`i` is outside the range):

  `if (i < 0 || i >= n) …`

Compound Statements

- In the `if` statement template, notice that *statement* is singular, not plural:

  `if ( `*expression*` ) `*statement*
- To make an `if` statement control two or more statements, use a ***compound statement.***
- A compound statement has the form

  `{ `*statements*` }`
- Putting braces around a group of statements forces the compiler to treat it as a single statement.

Compound Statements

- Example:

```
{ line_num = 0; page_num++; }
```

- A compound statement is usually put on multiple lines, with one statement per line:

```
{
  line_num = 0;
  page_num++;
}
```

- Each inner statement still ends with a semicolon, but the compound statement itself does not.

Compound Statements

- Example of a compound statement used inside an `if` statement:

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

- Compound statements are also common in loops and other places where the syntax of C requires a single statement.

The `else` Clause

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

```
if ( expression ) statement else statement
```

- The statement that follows the word `else` is executed if the expression has the value 0.
- Example:

```
if (i > j)
  max = i;
else
  max = j;
```


The `else` Clause

- When an `if` statement contains an `else` clause, where should the `else` be placed?
- Many C programmers align it with the `if` at the beginning of the statement.
- Inner statements are usually indented, but if they're short they can be put on the same line as the `if` and `else`:

```
if (i > j) max = i;
else max = j;
```

The `else` Clause

- It's not unusual for `if` statements to be nested inside other `if` statements:

```
if (i > j)
  if (i > k)
    max = i;
  else
    max = k;
else
  if (j > k)
    max = j;
  else
    max = k;
```

- Aligning each `else` with the matching `if` makes the nesting easier to see.

The `else` Clause

- To avoid confusion, don't hesitate to add braces:

```
if (i > j) {
  if (i > k)
    max = i;
  else
    max = k;
} else {
  if (j > k)
    max = j;
  else
    max = k;
}
```


The `else` Clause

- Some programmers use as many braces as possible inside `if` statements:

```
if (i > j) {
  if (i > k) {
    max = i;
  } else {
    max = k;
  }
} else {
  if (j > k) {
    max = j;
  } else {
    max = k;
  }
}
```


The `else` Clause

- Advantages of using braces even when they're not required:
  - Makes programs easier to modify, because more statements can easily be added to any `if` or `else` clause.
  - Helps avoid errors that can result from forgetting to use braces when adding statements to an `if` or `else` clause.

Cascaded `if` Statements

- A "cascaded" `if` statement is often the best way to test a series of conditions, stopping as soon as one of them is true.
- Example:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```


Cascaded `if` Statements

- Although the second `if` statement is nested inside the first, C programmers don't usually indent it.
- Instead, they align each `else` with the original `if`:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```


Cascaded `if` Statements

- This layout avoids the problem of excessive indentation when the number of tests is large:

```
if ( expression )
  statement
else if ( expression )
  statement
...
else if ( expression )
  statement
else
  statement
```


Program: Calculating a Broker's Commission

- When stocks are sold or purchased through a broker, the broker's commission often depends upon the value of the stocks traded.
- Suppose that a broker charges the amounts shown in the following table:

Transaction size	*Commission rate*
Under $2,500	$30 + 1.7%
$2,500–$6,250	$56 + 0.66%
$6,250–$20,000	$76 + 0.34%
$20,000–$50,000	$100 + 0.22%
$50,000–$500,000	$155 + 0.11%
Over $500,000	$255 + 0.09%

- The minimum charge is $39.

Program: Calculating a Broker's Commission

- The `broker.c` program asks the user to enter the amount of the trade, then displays the amount of the commission:

```
Enter value of trade: 30000
Commission: $166.00
```

- The heart of the program is a cascaded `if` statement that determines which range the trade falls into.

broker.c

```
/* Calculates a broker's commission */

#include <stdio.h>

int main(void)
{
  float commission, value;

  printf("Enter value of trade: ");
  scanf("%f", &value);

  if (value < 2500.00f)
    commission = 30.00f + .017f * value;
  else if (value < 6250.00f)
    commission = 56.00f + .0066f * value;
  else if (value < 20000.00f)
    commission = 76.00f + .0034f * value;
  else if (value < 50000.00f)
    commission = 100.00f + .0022f * value;
  else if (value < 500000.00f)
    commission = 155.00f + .0011f * value;
  else
    commission = 255.00f + .0009f * value;
```



```
  if (commission < 39.00f)
    commission = 39.00f;

  printf("Commission: $%.2f\n", commission);

  return 0;
}
```


The "Dangling `else`" Problem

- When if statements are nested, the "dangling `else`" problem may occur:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

- The indentation suggests that the `else` clause belongs to the outer `if` statement.
- However, C follows the rule that an `else` clause belongs to the nearest `if` statement that hasn't already been paired with an `else`.

The "Dangling `else`" Problem

- A correctly indented version would look like this:

```
if (y != 0)
  if (x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0\n");
```


The "Dangling `else`" Problem

- To make the `else` clause part of the outer `if` statement, we can enclose the inner `if` statement in braces:

```
if (y != 0) {
  if (x != 0)
    result = x / y;
} else
  printf("Error: y is equal to 0\n");
```

- Using braces in the original `if` statement would have avoided the problem in the first place.

Conditional Expressions

- C's ***conditional operator*** allows an expression to produce one of two values depending on the value of a condition.
- The conditional operator consists of two symbols (`?` and `:`), which must be used together:

  expr1 `?` *expr2* `:` *expr3*
- The operands can be of any type.
- The resulting expression is said to be a ***conditional expression.***

Conditional Expressions

- The conditional operator requires three operands, so it is often referred to as a ***ternary*** operator.
- The conditional expression *expr1* `?` *expr2* `:` *expr3* should be read "if *expr1* then *expr2* else *expr3*."
- The expression is evaluated in stages: *expr1* is evaluated first; if its value isn't zero, then *expr2* is evaluated, and its value is the value of the entire conditional expression.
- If the value of *expr1* is zero, then the value of *expr3* is the value of the conditional.

Conditional Expressions

- Example:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;          /* k is now 2 */
k = (i >= 0 ? i : 0) + j;   /* k is now 3 */
```

- The parentheses are necessary, because the precedence of the conditional operator is less than that of the other operators discussed so far, with the exception of the assignment operators.

Conditional Expressions

- Conditional expressions tend to make programs shorter but harder to understand, so it's probably best to use them sparingly.
- Conditional expressions are often used in `return` statements:

```
return i > j ? i : j;
```


Conditional Expressions

- Calls of `printf` can sometimes benefit from condition expressions. Instead of

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", j);
```

  we could simply write

```
printf("%d\n", i > j ? i : j);
```

- Conditional expressions are also common in certain kinds of macro definitions.

Boolean Values in C89

- For many years, the C language lacked a proper Boolean type, and there is none defined in the C89 standard.
- One way to work around this limitation is to declare an `int` variable and then assign it either 0 or 1:

```
int flag;

flag = 0;
…
flag = 1;
```

- Although this scheme works, it doesn't contribute much to program readability.

Boolean Values in C89

- To make programs more understandable, C89 programmers often define macros with names such as `TRUE` and `FALSE`:

```
#define TRUE 1
#define FALSE 0
```

- Assignments to `flag` now have a more natural appearance:

```
flag = FALSE;
…
flag = TRUE;
```


Boolean Values in C89

- To test whether `flag` is true, we can write

```
if (flag == TRUE) …
```

or just

```
if (flag) …
```

- The latter form is more concise. It also works correctly if `flag` has a value other than 0 or 1.
- To test whether `flag` is false, we can write

```
if (flag == FALSE) …
```

or

```
if (!flag) …
```


Boolean Values in C89

- Carrying this idea one step further, we might even define a macro that can be used as a type:

```
#define BOOL int
```

- `BOOL` can take the place of `int` when declaring Boolean variables:

```
BOOL flag;
```

- It's now clear that `flag` isn't an ordinary integer variable, but instead represents a Boolean condition.

Boolean Values in C99

- C99 provides the `_Bool` type.
- A Boolean variable can be declared by writing

```
_Bool flag;
```

- `_Bool` is an integer type, so a `_Bool` variable is really just an integer variable in disguise.
- Unlike an ordinary integer variable, however, a `_Bool` variable can only be assigned 0 or 1.
- Attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1:

```
flag = 5;   /* flag is assigned 1 */
```

Boolean Values in C99

- It's legal (although not advisable) to perform arithmetic on `_Bool` variables.
- It's also legal to print a `_Bool` variable (either 0 or 1 will be displayed).
- And, of course, a `_Bool` variable can be tested in an `if` statement:

```
if (flag)   /* tests whether flag is 1 */
  …
```


Boolean Values in C99

- C99's `<stdbool.h>` header makes it easier to work with Boolean values.
- It defines a macro, `bool`, that stands for `_Bool`.
- If `<stdbool.h>` is included, we can write

```
bool flag;   /* same as _Bool flag; */
```

- `<stdbool.h>` also supplies macros named `true` and `false`, which stand for 1 and 0, respectively, making it possible to write

```
flag = false;
…
flag = true;
```


The `switch` Statement

- A cascaded `if` statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```


The `switch` Statement

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

The `switch` Statement

- A `switch` statement may be easier to read than a cascaded `if` statement.
- `switch` statements are often faster than `if` statements.
- Most common form of the `switch` statement:

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


The `switch` Statement

- The word `switch` must be followed by an integer expression—the ***controlling expression***—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don't qualify, however.

The `switch` Statement

- Each case begins with a label of the form

  `case` *constant-expression* `:`
- A ***constant expression*** is much like an ordinary expression except that it can't contain variables or function calls.
  - `5` is a constant expression, and `5 + 10` is a constant expression, but `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant).
- The constant expression in a case label must evaluate to an integer (characters are acceptable).

The `switch` Statement

- After each case label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The `switch` Statement

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the `default` case doesn't need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```


The `switch` Statement

- To save space, several case labels can be put on the same line:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
           printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

- If the `default` case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the `switch`.

The Role of the `break` Statement

- Executing a `break` statement causes the program to "break" out of the `switch` statement; execution continues at the next statement after the `switch`.
- The `switch` statement is really a form of "computed jump."
- When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression.
- A case label is nothing more than a marker indicating a position within the `switch`.

The Role of the `break` Statement

- Without `break` (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Illegal grade");
}
```

- If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal grade
```

The Role of the `break` Statement

- Omitting `break` is sometimes done intentionally, but it's usually just an oversight.
- It's a good idea to point out deliberate omissions of `break`:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

- Although the last case never needs a `break` statement, including one makes it easy to add cases in the future.

Program: Printing a Date in Legal Form

- Contracts and other legal documents are often dated in the following way:

```
Dated this __________ day of __________ , 20__ .
```

- The `date.c` program will display a date in this form after the user enters the date in month/day/year form:

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

- The program uses `switch` statements to add "th" (or "st" or "nd" or "rd") to the day, and to print the month as a word instead of a number.

`date.c`

```
/* Prints a date in legal form */

#include <stdio.h>

int main(void)
{
  int month, day, year;

  printf("Enter date (mm/dd/yy): ");
  scanf("%d /%d /%d", &month, &day, &year);

  printf("Dated this %d", day);
  switch (day) {
    case 1: case 21: case 31:
      printf("st"); break;
    case 2: case 22:
      printf("nd"); break;
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");
```



```
  switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```

Chapter 6

Loops

Iteration Statements

- C's iteration statements are used to set up loops.
- A ***loop*** is a statement whose job is to repeatedly execute some other statement (the ***loop body***).
- In C, every loop has a ***controlling expression***.
- Each time the loop body is executed (an ***iteration*** of the loop), the controlling expression is evaluated.
  - If the expression is true (has a value that's not zero) the loop continues to execute.

Iteration Statements

- C provides three iteration statements:
  - The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed.
  - The `do` statement is used if the expression is tested *after* the loop body is executed.
  - The `for` statement is convenient for loops that increment or decrement a counting variable.

The `while` Statement

- Using a `while` statement is the easiest way to set up a loop.
- The `while` statement has the form

  `while ( `*expression*` )` *statement*
- *expression* is the controlling expression; *statement* is the loop body.

The `while` Statement

- Example of a `while` statement:

```
while (i < n)  /* controlling expression */
  i = i * 2;   /* loop body */
```

- When a `while` statement is executed, the controlling expression is evaluated first.
- If its value is nonzero (true), the loop body is executed and the expression is tested again.
- The process continues until the controlling expression eventually has the value zero.

The `while` Statement

- A `while` statement that computes the smallest power of 2 that is greater than or equal to a number `n`:

```
i = 1;
while (i < n)
  i = i * 2;
```

- A trace of the loop when `n` has the value 10:

`i = 1;`	`i` is now 1.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 2.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 4.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 8.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 16.
Is `i < n`?	No; exit from loop.

The `while` Statement

- Although the loop body must be a single statement, that's merely a technicality.
- If multiple statements are needed, use braces to create a single compound statement:

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {
  i = i * 2;
}
```


The `while` Statement

- The following statements display a series of "countdown" messages:

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

- The final message printed is `T minus 1 and counting`.

The `while` Statement

- Observations about the `while` statement:
  - The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by `i > 0` terminates, `i` must be less than or equal to 0.
  - The body of a `while` loop may not be executed at all, because the controlling expression is tested *before* the body is executed.
  - A `while` statement can often be written in a variety of ways. A more concise version of the countdown loop:

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


Infinite Loops

- A `while` statement won't terminate if the controlling expression always has a nonzero value.
- C programmers sometimes deliberately create an ***infinite loop*** by using a nonzero constant as the controlling expression:

```
while (1) …
```

- A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

Program: Printing a Table of Squares

- The `square.c` program uses a while statement to print a table of squares.
- The user specifies the number of entries in the table:

```
This program prints a table of squares.
Enter number of entries in table: 5
          1          1
          2          4
          3          9
          4         16
          5         25
```


`square.c`

```
/* Prints a table of squares using a while statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  while (i <= n) {
    printf("%10d%10d\n", i, i * i);
    i++;
  }

  return 0;
}
```

Program: Summing a Series of Numbers

- The `sum.c` program sums a series of integers entered by the user:

```
This program sums a series of integers.
Enter integers (0 to terminate): 8 23 71 5 0
The sum is: 107
```

- The program will need a loop that uses `scanf` to read a number and then adds the number to a running total.

sum.c

```
/* Sums a series of numbers */

#include <stdio.h>

int main(void)
{
  int n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%d", &n);
  while (n != 0) {
    sum += n;
    scanf("%d", &n);
  }
  printf("The sum is: %d\n", sum);

  return 0;
}
```


The do Statement

- General form of the `do` statement:

  `do` *statement* `while (` *expression* `) ;`

- When a `do` statement is executed, the loop body is executed first, then the controlling expression is evaluated.
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.

The do Statement

- The countdown example rewritten as a `do` statement:

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

- The `do` statement is often indistinguishable from the `while` statement.
- The only difference is that the body of a `do` statement is always executed at least once.

The do Statement

- It's a good idea to use braces in *all* `do` statements, whether or not they're needed, because a `do` statement without braces can easily be mistaken for a `while` statement:

```
do
  printf("T minus %d and counting\n", i--);
while (i > 0);
```

- A careless reader might think that the word `while` was the beginning of a `while` statement.

Program: Calculating the Number of Digits in an Integer

- The `numdigits.c` program calculates the number of digits in an integer entered by the user:

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

- The program will divide the user's input by 10 repeatedly until it becomes 0; the number of divisions performed is the number of digits.
- Writing this loop as a `do` statement is better than using a `while` statement, because every integer—even 0—has at least one digit.

numdigits.c

```
/* Calculates the number of digits in an integer */

#include <stdio.h>

int main(void)
{
  int digits = 0, n;

  printf("Enter a nonnegative integer: ");
  scanf("%d", &n);

  do {
    n /= 10;
    digits++;
  } while (n > 0);

  printf("The number has %d digit(s).\n", digits);

  return 0;
}
```


The for Statement

- The `for` statement is ideal for loops that have a "counting" variable, but it's versatile enough to be used for other kinds of loops as well.
- 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);
```

The `for` Statement

- The `for` statement is closely related to the `while` statement.
- Except in a few rare cases, a `for` loop can always be replaced by an equivalent `while` loop:

```
expr1;
while ( expr2 ) {
  statement
  expr3;
}
```

- *expr1* is an initialization step that's performed only once, before the loop begins to execute.

The `for` Statement

- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
- *expr3* is an operation to be performed at the end of each loop iteration.
- The result when this pattern is applied to the previous `for` loop:

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


The `for` Statement

- Studying the equivalent `while` statement can help clarify the fine points of a `for` statement.
- For example, what if `i--` is replaced by `--i`?

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

- The equivalent `while` loop shows that the change has no effect on the behavior of the loop:

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


The `for` Statement

- Since the first and third expressions in a `for` statement are executed as statements, their values are irrelevant—they're useful only for their side effects.
- Consequently, these two expressions are usually assignments or increment/decrement expressions.

`for` Statement Idioms

- The `for` statement is usually the best choice for loops that "count up" (increment a variable) or "count down" (decrement a variable).
- A `for` statement that counts up or down a total of `n` times will usually have one of the following forms:

***Counting up from* 0 *to* n–1:**
```
for (i = 0; i < n; i++) …
```

***Counting up from* 1 *to* n:**
```
for (i = 1; i <= n; i++) …
```

***Counting down from* n–1 *to* 0:**
```
for (i = n - 1; i >= 0; i--) …
```

***Counting down from* n *to* 1:**
```
for (i = n; i > 0; i--) …
```


`for` Statement Idioms

- Common `for` statement errors:
  - Using < instead of > (or vice versa) in the controlling expression. "Counting up" loops should use the < or <= operator. "Counting down" loops should use > or >=.
  - Using == in the controlling expression instead of <, <=, >, or >=.
  - "Off-by-one" errors such as writing the controlling expression as `i <= n` instead of `i < n`.

Omitting Expressions in a `for` Statement

- C allows any or all of the expressions that control a `for` statement to be omitted.
- If the *first* expression is omitted, no initialization is performed before the loop is executed:

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

- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false:

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


Omitting Expressions in a `for` Statement

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a while statement in disguise:

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

is the same as

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

- The `while` version is clearer and therefore preferable.

Omitting Expressions in a `for` Statement

- If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion).
- For example, some programmers use the following `for` statement to establish an infinite loop:

```
for (;;) …
```


`for` Statements in C99

- 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 variable `i` need not have been declared prior to this statement.

`for` Statements in C99

- A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);
  /* legal; i is visible inside loop */
  …
}
printf("%d", i);    /*** WRONG ***/
```


`for` Statements in C99

- Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand.
- However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.
- A `for` statement may declare more than one variable, provided that all variables have the same type:

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

The Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
- This effect can be accomplished by using a ***comma expression*** as the first or third expression in the `for` statement.
- A comma expression has the form

  expr1 `,` *expr2*

  where *expr1* and *expr2* are any two expressions.

The Comma Operator

- A comma expression is evaluated in two steps:
  - First, *expr1* is evaluated and its value discarded.
  - Second, *expr2* is evaluated; its value is the value of the entire expression.
- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
- When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated.
  - If `i` and `j` have the values 1 and 5, respectively, the value of the expression will be 7, and `i` will be incremented to 2.

The Comma Operator

- The comma operator is left associative, so the compiler interprets

  ```
  i = 1, j = 2, k = i + j
  ```

  as

  ```
  ((i = 1), (j = 2)), (k = (i + j))
  ```

- Since the left operand in a comma expression is evaluated before the right operand, the assignments `i = 1`, `j = 2`, and `k = i + j` will be performed from left to right.

The Comma Operator

- The comma operator makes it possible to "glue" two expressions together to form a single expression.
- Certain macro definitions can benefit from the comma operator.
- The `for` statement is the only other place where the comma operator is likely to be found.
- Example:

  ```
  for (sum = 0, i = 1; i <= N; i++)
    sum += i;
  ```

- With additional commas, the `for` statement could initialize more than two variables.

Program: Printing a Table of Squares (Revisited)

- The `square.c` program (Section 6.1) can be improved by converting its `while` loop to a `for` loop.

square2.c

```
/* Prints a table of squares using a for statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  for (i = 1; i <= n; i++)
    printf("%10d%10d\n", i, i * i);

  return 0;
}
```


Program: Printing a Table of Squares (Revisited)

- C places no restrictions on the three expressions that control the behavior of a `for` statement.
- Although these expressions usually initialize, test, and update the same variable, there's no requirement that they be related in any way.
- The `square3.c` program is equivalent to `square2.c`, but contains a `for` statement that initializes one variable (`square`), tests another (`i`), and increments a third (`odd`).
- The flexibility of the `for` statement can sometimes be useful, but in this case the original program was clearer.

square3.c

```
/* Prints a table of squares using an odd method */

#include <stdio.h>

int main(void)
{
  int i, n, odd, square;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  odd = 3;
  for (square = 1; i <= n; odd += 2) {
    printf("%10d%10d\n", i, square);
    ++i;
    square += odd;
  }

  return 0;
}
```

Exiting from a Loop

- The normal exit point for a loop is at the beginning (as in a `while` or `for` statement) or at the end (the `do` statement).
- Using the `break` statement, it's possible to write a loop with an exit point in the middle or a loop with more than one exit point.

The `break` Statement

- The `break` statement can transfer control out of a switch statement, but it can also be used to jump out of a `while`, `do`, or `for` loop.
- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```


The `break` Statement

- After the loop has terminated, an `if` statement can be use to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```


The `break` Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end.
- Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape only one level of nesting.
- Example:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop.

The `continue` Statement

- The `continue` statement is similar to `break`:
  - `break` transfers control just past the end of a loop.
  - `continue` transfers control to a point just before the end of the loop body.
- With `break`, control leaves the loop; with `continue`, control remains inside the loop.
- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

The `continue` Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```


The `continue` Statement

- The same loop written without using `continue`:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```

The goto Statement

- The goto statement is capable of jumping to any statement in a function, provided that the statement has a ***label.***
- A label is just an identifier placed at the beginning of a statement:

  identifier : *statement*
- A statement may have more than one label.
- The goto statement itself has the form

  goto *identifier* ;
- Executing the statement goto *L*; transfers control to the statement that follows the label *L*, which must be in the same function as the goto statement itself.

The goto Statement

- If C didn't have a break statement, a goto statement could be used to exit from a loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;
done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```


The goto Statement

- The goto statement is rarely needed in everyday C programming.
- The break, continue, and return statements—which are essentially restricted goto statements—and the exit function are sufficient to handle most situations that might require a goto in other languages.
- Nonetheless, the goto statement can be helpful once in a while.

The goto Statement

- Consider the problem of exiting a loop from within a switch statement.
- The break statement doesn't have the desired effect: it exits from the switch, but not from the loop.
- A goto statement solves the problem:

```
while (…) {
  switch (…) {
    …
    goto loop_done;    /* break won't work here */
    …
  }
}
loop_done: …
```

- The goto statement is also useful for exiting from nested loops.

Program: Balancing a Checkbook

- Many simple interactive programs present the user with a list of commands to choose from.
- Once a command is entered, the program performs the desired action, then prompts the user for another command.
- This process continues until the user selects an "exit" or "quit" command.
- The heart of such a program will be a loop:

```
for (;;) {
  prompt user to enter command;
  read command;
  execute command;
}
```


Program: Balancing a Checkbook

- Executing the command will require a `switch` statement (or cascaded `if` statement):

```
for (;;) {
  prompt user to enter command;
  read command;
  switch (command) {
    case command1: perform operation1; break;
    case command2: perform operation2; break;
      .
      .
      .
    case commandn: perform operationn; break;
    default: print error message; break;
  }
}
```


Program: Balancing a Checkbook

- The `checking.c` program, which maintains a checkbook balance, uses a loop of this type.
- The user is allowed to clear the account balance, credit money to the account, debit money from the account, display the current balance, and exit the program.

Program: Balancing a Checkbook

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

checking.c

```
/* Balances a checkbook */

#include <stdio.h>

int main(void)
{
  int cmd;
  float balance = 0.0f, credit, debit;

  printf("*** ACME checkbook-balancing program ***\n");
  printf("Commands: 0=clear, 1=credit, 2=debit, ");
  printf("3=balance, 4=exit\n\n");
  for (;;) {
    printf("Enter command: ");
    scanf("%d", &cmd);
    switch (cmd) {
      case 0:
        balance = 0.0f;
        break;
```



```
      case 1:
        printf("Enter amount of credit: ");
        scanf("%f", &credit);
        balance += credit;
        break;
      case 2:
        printf("Enter amount of debit: ");
        scanf("%f", &debit);
        balance -= debit;
        break;
      case 3:
        printf("Current balance: $%.2f\n", balance);
        break;
      case 4:
        return 0;
      default:
        printf("Commands: 0=clear, 1=credit, 2=debit, ");
        printf("3=balance, 4=exit\n\n");
        break;
    }
  }
}
```


The Null Statement

- A statement can be ***null***—devoid of symbols except for the semicolon at the end.
- The following line contains three statements:

```
i = 0; ; j = 1;
```

- The null statement is primarily good for one thing: writing loops whose bodies are empty.

The Null Statement

- Consider the following prime-finding loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

- If the `n % d == 0` condition is moved into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)
  /* empty loop body */ ;
```

- To avoid confusion, C programmers customarily put the null statement on a line by itself.

The Null Statement

- Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement.
- Example 1:

```
if (d == 0);                          /*** WRONG ***/
  printf("Error: Division by zero\n");
```

  The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

- Example 2:

```
i = 10;
while (i > 0);                        /*** WRONG ***/
{
  printf("T minus %d and counting\n", i);
  --i;
}
```

  The extra semicolon creates an infinite loop.

The Null Statement

- Example 3:

```
i = 11;
while (--i > 0);                      /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

  The loop body is executed only once; the message printed is:

```
T minus 0 and counting
```

- Example 4:

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

  Again, the loop body is executed only once, and the same message is printed as in Example 3.

Chapter 7

Basic Types

Basic Types

- C's ***basic*** (built-in) ***types:***
  - Integer types, including long integers, short integers, and unsigned integers
  - Floating types (`float`, `double`, and `long double`)
  - `char`
  - `_Bool` (C99)

Integer Types

- C supports two fundamentally different kinds of numeric types: integer types and floating types.
- Values of an ***integer type*** are whole numbers.
- Values of a floating type can have a fractional part as well.
- The integer types, in turn, are divided into two categories: signed and unsigned.

Signed and Unsigned Integers

- The leftmost bit of a ***signed*** integer (known as the ***sign bit***) is 0 if the number is positive or zero, 1 if it's negative.
- The largest 16-bit integer has the binary representation 0111111111111111, which has the value 32,767 ($2^{15} - 1$).
- The largest 32-bit integer is
  01111111111111111111111111111111
  which has the value 2,147,483,647 ($2^{31} - 1$).
- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be ***unsigned.***
- The largest 16-bit unsigned integer is 65,535 ($2^{16} - 1$).
- The largest 32-bit unsigned integer is 4,294,967,295 ($2^{32} - 1$).

Signed and Unsigned Integers

- By default, integer variables are signed in C—the leftmost bit is reserved for the sign.
- To tell the compiler that a variable has no sign bit, declare it to be `unsigned`.
- Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications.

Integer Types

- The `int` type is usually 32 bits, but may be 16 bits on older CPUs.
- ***Long*** integers may have more bits than ordinary integers; ***short*** integers may have fewer bits.
- The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form integer types.
- Only six combinations produce different types:

```
short int       unsigned short int
int             unsigned int
long int        unsigned long int
```

- The order of the specifiers doesn't matter. Also, the word `int` can be dropped (`long int` can be abbreviated to just `long`).

Integer Types

- The range of values represented by each of the six integer types varies from one machine to another.
- However, the C standard requires that `short int`, `int`, and `long int` must each cover a certain minimum range of values.
- Also, `int` must not be shorter than `short int`, and `long int` must not be shorter than `int`.

Integer Types

- Typical ranges of values for the integer types on a 16-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–32,768	32,767
`unsigned int`	0	65,535
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 32-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	–2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	–32,768	32,767
`unsigned short int`	0	65,535
`int`	–2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	-2^{63}	$2^{63}-1$
`unsigned long int`	0	$2^{64}-1$

- The `<limits.h>` header defines macros that represent the smallest and largest values of each integer type.

Integer Types in C99

- C99 provides two additional standard integer types, `long long int` and `unsigned long long int`.
- Both `long long` types are required to be at least 64 bits wide.
- The range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807).
- The range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

Integer Types in C99

- The `short int`, `int`, `long int`, and `long long int` types (along with the `signed char` type) are called ***standard signed integer types*** in C99.
- The `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` types (along with the `unsigned char` type and the `_Bool` type) are called ***standard unsigned integer types.***
- In addition to the standard integer types, the C99 standard allows implementation-defined ***extended integer types,*** both signed and unsigned.

Integer Constants

- ***Constants*** are numbers that appear in the text of a program.
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

- Octal numbers use only the digits 0 through 7.
- Each position in an octal number represents a power of 8.
  - The octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.
- A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  - The hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

Integer Constants

- ***Decimal*** constants contain digits between 0 and 9, but must not begin with a zero:

```
15   255   32767
```

- ***Octal*** constants contain only digits between 0 and 7, and must begin with a zero:

```
017   0377   077777
```

- ***Hexadecimal*** constants contain digits between 0 and 9 and letters between `a` and `f`, and always begin with `0x`:

```
0xf   0xff   0x7fff
```

- The letters in a hexadecimal constant may be either upper or lower case:

```
0xff   0xfF   0xFf   0xFF   0Xff   0XfF   0XFf   0XFF
```


Integer Constants

- The type of a *decimal* integer constant is normally `int`.
- If the value of the constant is too large to store as an `int`, the constant has type `long int` instead.
- If the constant is too large to store as a `long int`, the compiler will try `unsigned long int` as a last resort.
- For an *octal* or *hexadecimal* constant, the rules are slightly different: the compiler will go through the types `int`, `unsigned int`, `long int`, and `unsigned long int` until it finds one capable of representing the constant.

Integer Constants

- To force the compiler to treat a constant as a long integer, just follow it with the letter `L` (or `l`):

  ```
  15L   0377L   0x7fffL
  ```

- To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

  ```
  15U   0377U   0x7fffU
  ```

- `L` and `U` may be used in combination:

  ```
  0xffffffffUL
  ```

  The order of the `L` and `U` doesn't matter, nor does their case.

Integer Constants in C99

- In C99, integer constants that end with either `LL` or `ll` (the case of the two letters must match) have type `long long int`.
- Adding the letter `U` (or `u`) before or after the `LL` or `ll` denotes a constant of type `unsigned long long int`.
- C99's general rules for determining the type of an integer constant are a bit different from those in C89.
- The type of a decimal constant with no suffix (`U`, `u`, `L`, `l`, `LL`, or `ll`) is the "smallest" of the types `int`, `long int`, or `long long int` that can represent the value of that constant.

Integer Constants in C99

- For an octal or hexadecimal constant, the list of possible types is `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, and `unsigned long long int`, in that order.
- Any suffix at the end of a constant changes the list of possible types.
  - A constant that ends with `U` (or `u`) must have one of the types `unsigned int`, `unsigned long int`, or `unsigned long long int`.
  - A decimal constant that ends with `L` (or `l`) must have one of the types `long int` or `long long int`.

Integer Overflow

- When arithmetic operations are performed on integers, it's possible that the result will be too large to represent.
- For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`.
- If the result can't be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred.

Integer Overflow

- The behavior when integer overflow occurs depends on whether the operands were signed or unsigned.
  - When overflow occurs during an operation on *signed* integers, the program's behavior is undefined.
  - When overflow occurs during an operation on *unsigned* integers, the result *is* defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result.

Reading and Writing Integers

- Reading and writing unsigned, short, and long integers requires new conversion specifiers.
- When reading or writing an *unsigned* integer, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification.

```
unsigned int u;

scanf("%u", &u);  /* reads  u in base 10 */
printf("%u", u);  /* writes u in base 10 */
scanf("%o", &u);  /* reads  u in base  8 */
printf("%o", u);  /* writes u in base  8 */
scanf("%x", &u);  /* reads  u in base 16 */
printf("%x", u);  /* writes u in base 16 */
```


Reading and Writing Integers

- When reading or writing a *short* integer, put the letter `h` in front of `d`, `o`, `u`, or `x`:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter `l` ("ell," not "one") in front of `d`, `o`, `u`, or `x`.
- When reading or writing a *long long* integer (C99 only), put the letters `ll` in front of `d`, `o`, `u`, or `x`.

Program: Summing a Series of Numbers (Revisited)

- The `sum.c` program (Chapter 6) sums a series of integers.
- One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable.
- Here's what might happen if the program is run on a machine whose integers are 16 bits long:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

- When overflow occurs with signed numbers, the outcome is undefined.
- The program can be improved by using `long` variables.

sum2.c

```
/* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
  long n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%ld", &n);
  while (n != 0) {
    sum += n;
    scanf("%ld", &n);
  }
  printf("The sum is: %ld\n", sum);

  return 0;
}
```


Floating Types

- C provides three ***floating types,*** corresponding to different floating-point formats:
  - `float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).

The IEEE Floating-Point Standard

- IEEE Standard 754 was developed by the Institute of Electrical and Electronics Engineers.
- It has two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits).
- Numbers are stored in a form of scientific notation, with each number having a ***sign,*** an ***exponent,*** and a ***fraction.***
- In single-precision format, the exponent is 8 bits long, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.

Floating Types

- Characteristics of `float` and `double` when implemented according to the IEEE standard:

Type	*Smallest Positive Value*	*Largest Value*	*Precision*
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	2.22507×10^{-308}	1.79769×10^{308}	15 digits

- On computers that don't follow the IEEE standard, this table won't be valid.
- In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`.

Floating Types

- Macros that define the characteristics of the floating types can be found in the `<float.h>` header.
- In C99, the floating types are divided into two categories.
  - ***Real floating types*** (`float`, `double`, `long double`)
  - ***Complex types*** (`float _Complex`, `double _Complex`, `long double _Complex`)

Floating Constants

- Floating constants can be written in a variety of ways.
- Valid ways of writing the number 57.0:

```
57.0   57.   57.0e0   57E0   5.7e1   5.7e+1
.57e2   570.e-1
```

- A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled.
- If an exponent is present, it must be preceded by the letter `E` (or `e`). An optional `+` or `-` sign may appear after the `E` (or `e`).

Floating Constants

- By default, floating constants are stored as double-precision numbers.
- To indicate that only single precision is desired, put the letter `F` (or `f`) at the end of the constant (for example, `57.0F`).
- To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

Reading and Writing Floating-Point Numbers

- The conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers.
- When reading a value of type `double`, put the letter `l` in front of `e`, `f`, or `g`:

```
double d;

scanf("%lf", &d);
```

- *Note:* Use `l` only in a `scanf` format string, not a `printf` string.
- In a `printf` format string, the `e`, `f`, and `g` conversions can be used to write either `float` or `double` values.
- When reading or writing a value of type `long double`, put the letter `L` in front of `e`, `f`, or `g`.

Character Types

- The only remaining basic type is `char`, the character type.
- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character Sets

- Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.
- ASCII is often extended to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

Character Sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';   /* lower-case a */
ch = 'A';   /* upper-case A */
ch = '0';   /* zero         */
ch = ' ';   /* space        */
```

- Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers.*
- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.
- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.
- Character constants actually have `int` type rather than `char` type.

Operations on Characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
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' */
```


Operations on Characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.
- These values depend on the character set in use, so programs that use <, <=, >, and >= to compare characters may not be portable.

Operations on Characters

- The fact that characters have the same properties as numbers has advantages.
- For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

- Disadvantages of treating characters as numbers:
  - Can lead to errors that won't be caught by the compiler.
  - Allows meaningless expressions such as `'a' * 'b' / 'c'`.
  - Can hamper portability, since programs may rely on assumptions about the underlying character set.

Signed and Unsigned Characters

- The `char` type—like the integer types—exists in both signed and unsigned versions.
- Signed characters normally have values between –128 and 127. Unsigned characters have values between 0 and 255.
- Some compilers treat `char` as a signed type, while others treat it as an unsigned type. Most of the time, it doesn't matter.
- C allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;
```


Signed and Unsigned Characters

- C89 uses the term ***integral types*** to refer to both the integer types and the character types.
- Enumerated types are also integral types.
- C99 doesn't use the term "integral types."
- Instead, it expands the meaning of "integer types" to include the character types and the enumerated types.
- C99's `_Bool` type is considered to be an unsigned integer type.

Arithmetic Types

- The integer types and floating types are collectively known as ***arithmetic types.***
- A summary of the arithmetic types in C89, divided into categories and subcategories:
  - Integral types
    - `char`
    - Signed integer types (`signed char`, `short int`, `int`, `long int`)
    - Unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`)
    - Enumerated types
  - Floating types (`float`, `double`, `long double`)

Arithmetic Types

- C99 has a more complicated hierarchy:
  - Integer types
    - `char`
    - Signed integer types, both standard (`signed char`, `short int`, `int`, `long int`, `long long int`) and extended
    - Unsigned integer types, both standard (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `_Bool`) and extended
    - Enumerated types
  - Floating types
    - Real floating types (`float`, `double`, `long double`)
    - Complex types (`float _Complex`, `double _Complex`, `long double _Complex`)

Escape Sequences

- A character constant is usually one character enclosed in single quotes.
- However, certain special characters—including the new-line character—can't be written in this way, because they're invisible (nonprinting) or because they can't be entered from the keyboard.
- ***Escape sequences*** provide a way to represent these characters.
- There are two kinds of escape sequences: ***character escapes*** and ***numeric escapes.***

Escape Sequences

- A complete list of character escapes:

Name	*Escape Sequence*
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Question mark	`\?`
Single quote	`\'`
Double quote	`\"`

Escape Sequences

- Character escapes are handy, but they don't exist for all nonprinting ASCII characters.
- Character escapes are also useless for representing characters beyond the basic 128 ASCII characters.
- Numeric escapes, which can represent any character, are the solution to this problem.
- A numeric escape for a particular character uses the character's octal or hexadecimal value.
- For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex.

Escape Sequences

- An ***octal escape sequence*** consists of the `\` character followed by an octal number with at most three digits, such as `\33` or `\033`.
- A ***hexadecimal escape sequence*** consists of `\x` followed by a hexadecimal number, such as `\x1b` or `\x1B`.
- The `x` must be in lower case, but the hex digits can be upper or lower case.

Escape Sequences

- When used as a character constant, an escape sequence must be enclosed in single quotes.
- For example, a constant representing the escape character would be written `'\33'` (or `'\x1b'`).
- Escape sequences tend to get a bit cryptic, so it's often a good idea to use `#define` to give them names:

```
#define ESC '\33'
```

- Escape sequences can be embedded in strings as well.

Character-Handling Functions

- Calling C's `toupper` library function is a fast and portable way to convert case:

```
ch = toupper(ch);
```

- `toupper` returns the upper-case version of its argument.
- Programs that call `toupper` need to have the following `#include` directive at the top:

```
#include <ctype.h>
```

- The C library provides many other useful character-handling functions.

Reading and Writing Characters Using `scanf` and `printf`

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);  /* reads one character */
printf("%c", ch);  /* writes one character */
```

- `scanf` doesn't skip white-space characters.
- To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:

```
scanf(" %c", &ch);
```


Reading and Writing Characters Using `scanf` and `printf`

- Since `scanf` doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character.
- A loop that reads and ignores all remaining characters in the current input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters Using `getchar` and `putchar`

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- `getchar` returns an `int` value rather than a `char` value, so `ch` will often have type `int`.
- Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Reading and Writing Characters Using `getchar` and `putchar`

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves execution time.
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats.
  - They are usually implemented as macros for additional speed.
- `getchar` has another advantage. Because it returns the character that it reads, `getchar` lends itself to various C idioms.

Reading and Writing Characters Using `getchar` and `putchar`

- Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\n');
```


Reading and Writing Characters Using `getchar` and `putchar`

- Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
  ;
```

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character:

```
while (getchar() != '\n')
  ;
```

Reading and Writing Characters Using `getchar` and `putchar`

- `getchar` is useful in loops that skip characters as well as loops that search for characters.
- A statement that uses `getchar` to skip an indefinite number of blank characters:

```
while ((ch = getchar()) == ' ')
  ;
```

- When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.

Reading and Writing Characters Using `getchar` and `putchar`

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has "peeked" at but not read, including the new-line character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

  `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character.
- `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`.

`length.c`

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  char ch;
  int len = 0;

  printf("Enter a message: ");
  ch = getchar();
  while (ch != '\n') {
    len++;
    ch = getchar();
  }
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```

length2.c

```
/* Determines the length of a message */

#include <stdio.h>

int main(void)
{
  int len = 0;

  printf("Enter a message: ");
  while (getchar() != '\n')
    len++;
  printf("Your message was %d character(s) long.\n", len);

  return 0;
}
```


Type Conversion

- For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way.
- When operands of different types are mixed in expressions, the C compiler may have to generate instructions that change the types of some operands so that hardware will be able to evaluate the expression.
  - If we add a 16-bit `short` and a 32-bit `int`, the compiler will arrange for the `short` value to be converted to 32 bits.
  - If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format.

Type Conversion

- Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions.***
- C also allows the programmer to perform ***explicit conversions,*** using the cast operator.
- The rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Type Conversion

- Implicit conversions are performed:
  - When the operands in an arithmetic or logical expression don't have the same type. (C performs what are known as the ***usual arithmetic conversions.***)
  - When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
  - When the type of an argument in a function call doesn't match the type of the corresponding parameter.
  - When the type of the expression in a `return` statement doesn't match the function's return type.
- Chapter 9 discusses the last two cases.

The Usual Arithmetic Conversions

- The usual arithmetic conversions are applied to the operands of most binary operators.
- If `f` has type `float` and `i` has type `int`, the usual arithmetic conversions will be applied to the operands in the expression `f + i`.
- Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type).
  - When an integer is converted to `float`, the worst that can happen is a minor loss of precision.
  - Converting a floating-point number to `int`, on the other hand, causes the fractional part of the number to be lost. Worse still, the result will be meaningless if the original number is larger than the largest possible integer or smaller than the smallest integer.

The Usual Arithmetic Conversions

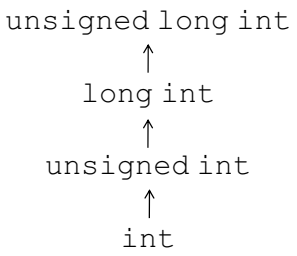
- Strategy behind the usual arithmetic conversions: convert operands to the "narrowest" type that will safely accommodate both values.
- Operand types can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***).
- Common promotions include the ***integral promotions,*** which convert a character or short integer to type `int` (or to `unsigned int` in some cases).
- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.

The Usual Arithmetic Conversions

- ***The type of either operand is a floating type.***
  - If one operand has type `long double`, then convert the other operand to type `long double`.
  - Otherwise, if one operand has type `double`, convert the other operand to type `double`.
  - Otherwise, if one operand has type `float`, convert the other operand to type `float`.
- Example: If one operand has type `long int` and the other has type `double`, the `long int` operand is converted to `double`.

The Usual Arithmetic Conversions

- ***Neither operand type is a floating type.*** First perform integral promotion on both operands.
- Then use the following diagram to promote the operand whose type is narrower:



The Usual Arithmetic Conversions

- When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value.
- This rule can cause obscure programming errors.
- It's best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

The Usual Arithmetic Conversions

- Example of the usual arithmetic conversions:

```
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 + ul;    /* ul is converted to float            */
d = d + f;     /* f is converted to double            */
ld = ld + d;   /* d is converted to long double       */
```


Conversion During Assignment

- The usual arithmetic conversions don't apply to assignment.
- Instead, the expression on the right side of the assignment is converted to the type of the variable on the left side:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int   */
f = i;   /* i is converted to float */
d = f;   /* f is converted to double */
```


Conversion During Assignment

- Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;     /* i is now 842 */
i = -842.97;    /* i is now -842 */
```

- Assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;     /*** WRONG ***/
i = 1.0e20;    /*** WRONG ***/
f = 1.0e100;   /*** WRONG ***/
```

Conversion During Assignment

- It's a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable:

```
f = 3.14159f;
```

- Without the suffix, the constant `3.14159` would have type `double`, possibly causing a warning message.

Implicit Conversions in C99

- C99's rules for implicit conversions are somewhat different.
- Each integer type has an "integer conversion rank."
- Ranks from highest to lowest:
  1. `long long int`, `unsigned long long int`
  2. `long int`, `unsigned long int`
  3. `int`, `unsigned int`
  4. `short int`, `unsigned short int`
  5. `char`, `signed char`, `unsigned char`
  6. `_Bool`
- C99's "integer promotions" involve converting any type whose rank is less than `int` and `unsigned int` to `int` (provided that all values of the type can be represented using `int`) or else to `unsigned int`.

Implicit Conversions in C99

- C99's rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.
- ***The type of either operand is a floating type.*** As long as neither operand has a complex type, the rules are the same as before. (The conversion rules for complex types are discussed in Chapter 27.)

Implicit Conversions in C99

- ***Neither operand type is a floating type.*** Perform integer promotion on both operands. Stop if the types of the operands are now the same. Otherwise, use the following rules:
  - If both operands have signed types or both have unsigned types, convert the operand whose type has lesser integer conversion rank to the type of the operand with greater rank.
  - If the unsigned operand has rank greater or equal to the rank of the type of the signed operand, convert the signed operand to the type of the unsigned operand.
  - If the type of the signed operand can represent all of the values of the type of the unsigned operand, convert the unsigned operand to the type of the signed operand.
  - Otherwise, convert both operands to the unsigned type corresponding to the type of the signed operand.

Implicit Conversions in C99

- All arithmetic types can be converted to `_Bool` type. The result of the conversion is 0 if the original value is 0; otherwise, the result is 1.

Casting

- Although C's implicit conversions are convenient, we sometimes need a greater degree of control over type conversion.
- For this reason, C provides ***casts.***
- A cast expression has the form

  (*type-name*) *expression*

  type-name specifies the type to which the expression should be converted.

Casting

- Using a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;

frac_part = f - (int) f;
```

- The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.
- Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f;  /* f is converted to int */
```


Casting

- Cast expressions also let us force the compiler to perform conversions.
- Example:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

- To avoid truncation during division, we need to cast one of the operands:

```
quotient = (float) dividend / divisor;
```

- Casting `dividend` to `float` causes the compiler to convert `divisor` to `float` also.

Casting

- C regards (*type-name*) as a unary operator.
- Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

- Other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
quotient = (float) dividend / (float) divisor;
```


Casting

- Casts are sometimes necessary to avoid overflow:

```
long i;
int j = 1000;

i = j * j;   /* overflow may occur */
```

- Using a cast avoids the problem:

```
i = (long) j * j;
```

- The statement

```
i = (long) (j * j);   /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.

Type Definitions

- The `#define` directive can be used to create a "Boolean type" macro:

```
#define BOOL int
```

- There's a better way using a feature known as a ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names.
- Example:

```
Bool flag;   /* same as int flag; */
```


Advantages of Type Definitions

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as

```
typedef float Dollars;
```

and then writing

```
Dollars cash_in, cash_out;
```

is more informative than just writing

```
float cash_in, cash_out;
```

Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine `Dollars` as `double`, only the type definition need be changed:

```
typedef double Dollars;
```

- Without the type definition, we would need to locate all `float` variables that store dollar amounts and change their declarations.

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

```
i = 100000;
```

is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

Type Definitions and Portability

- For greater portability, consider using `typedef` to define new names for integer types.
- Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000.
- We could use `long` variables for this purpose, but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values. Also, `int` variables may take up less space.

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

```
typedef int Quantity;
```

and use this type to declare variables:

```
Quantity q;
```

- When we transport the program to a machine with shorter integers, we'll change the type definition:

```
typedef long Quantity;
```

- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

- In C99, the `<stdint.h>` header uses `typedef` to define names for integer types with a particular number of bits.

The `sizeof` Operator

- The value of the expression

  `sizeof ( type-name )`

  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*.
- `sizeof(char)` is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, `sizeof(int)` is normally 4.

The `sizeof` Operator

- The `sizeof` operator can also be applied to constants, variables, and expressions in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.
- When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses.
  - We could write `sizeof i` instead of `sizeof(i)`.
- Parentheses may be needed anyway because of operator precedence.
  - The compiler interprets `sizeof i + j` as `(sizeof i) + j`, because `sizeof` takes precedence over binary `+`.

The `sizeof` Operator

- Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named `size_t`.
- In C89, it's best to convert the value of the expression to a known type before printing it:

```
printf("Size of int: %lu\n",
       (unsigned long) sizeof(int));
```

- The `printf` function in C99 can display a `size_t` value directly if the letter `z` is included in the conversion specification:

```
printf("Size of int: %zu\n", sizeof(int));
```

Chapter 8

Arrays

Scalar Variables versus Aggregate Variables

- So far, the only variables we've seen are ***scalar:*** capable of holding a single data item.
- C also supports ***aggregate*** variables, which can store collections of values.
- There are two kinds of aggregates in C: arrays and structures.
- The focus of the chapter is on one-dimensional arrays, which play a much bigger role in C than do multidimensional arrays.

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the same type.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array `a` are conceptually arranged one after another in a single row (or column):

One-Dimensional Arrays

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

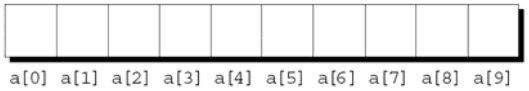
```
int a[10];
```

- The elements may be of any type; the length of the array can be any (integer) constant expression.
- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

- To access an array element, write the array name followed by an integer value in square brackets.
- This is referred to as ***subscripting*** or ***indexing*** the array.
- The elements of an array of length *n* are indexed from 0 to *n* – 1.
- If `a` is an array of length 10, its elements are designated by `a[0]`, `a[1]`, …, `a[9]`:



Array Subscripting

- Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

- In general, if an array contains elements of type *T*, then each element of the array is treated as if it were a variable of type *T*.

Array Subscripting

- Many programs contain `for` loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array `a` of length `N`:

```
for (i = 0; i < N; i++)
  a[i] = 0;             /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);   /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];          /* sums the elements of a */
```


Array Subscripting

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

With some compilers, this innocent-looking `for` statement causes an infinite loop.

Array Subscripting

- An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

- The expression can even have side effects:

```
i = 0;
while (i < N)
  a[i++] = 0;
```


Array Subscripting

- Be careful when an array subscript has a side effect:

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i`, causing undefined behavior.
- The problem can be avoided by removing the increment from the subscript:

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```


Program: Reversing a Series of Numbers

- The `reverse.c` program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

- The program stores the numbers in an array as they're read, then goes through the array backwards, printing the elements one by one.

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

Array Initialization

- An array, like any other variable, can be given an initial value at the time it's declared.
- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```


Array Initialization

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

  There's a single 0 inside the braces because it's illegal for an initializer to be completely empty.
- It's also illegal for an initializer to be longer than the array it initializes.

Array Initialization

- If an initializer is present, the length of the array may be omitted:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- The compiler uses the length of the initializer to determine how long the array is.

Designated Initializers (C99)

- It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values.
- An example:

```
int a[15] =
  {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

- For a large array, writing an initializer in this fashion is tedious and error-prone.

Designated Initializers (C99)

- C99's ***designated initializers*** can be used to solve this problem.
- Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- Each number in brackets is said to be a ***designator.***

Designated Initializers (C99)

- Designated initializers are shorter and easier to read (at least for some arrays).
- Also, the order in which the elements are listed no longer matters.
- Another way to write the previous example:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```


Designated Initializers (C99)

- Designators must be integer constant expressions.
- If the array being initialized has length n, each designator must be between 0 and $n - 1$.
- If the length of the array is omitted, a designator can be any nonnegative integer.
  - The compiler will deduce the length of the array from the largest designator.
- The following array will have 24 elements:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```


Designated Initializers (C99)

- An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};
```

Program: Checking a Number for Repeated Digits

- The `repdigit.c` program checks whether any of the digits in a number appear more than once.
- After the user enters a number, the program prints either `Repeated digit` or `No repeated digit`:

```
Enter a number: 28212
Repeated digit
```

- The number 28212 has a repeated digit (2); a number like 9357 doesn't.

Program: Checking a Number for Repeated Digits

- The program uses an array of 10 Boolean values to keep track of which digits appear in a number.
- Initially, every element of the `digit_seen` array is false.
- When given a number `n`, the program examines `n`'s digits one at a time, storing the current digit in a variable named `digit`.
  - If `digit_seen[digit]` is true, then `digit` appears at least twice in `n`.
  - If `digit_seen[digit]` is false, then `digit` has not been seen before, so the program sets `digit_seen[digit]` to `true` and keeps going.

repdigit.c

```
/* Checks numbers for repeated digits */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);
  while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
      break;
    digit_seen[digit] = true;
    n /= 10;
  }
```



```
  if (n > 0)
    printf("Repeated digit\n");
  else
    printf("No repeated digit\n");

  return 0;
}
```

Using the `sizeof` Operator with Arrays

- The `sizeof` operator can determine the size of an array (in bytes).
- If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).
- We can also use `sizeof` to measure the size of an array element, such as `a[0]`.
- Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```


Using the `sizeof` Operator with Arrays

- Some programmers use this expression when the length of the array is needed.
- A loop that clears the array `a`:

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
  a[i] = 0;
```

Note that the loop doesn't have to be modified if the array length should change at a later date.

Using the `sizeof` Operator with Arrays

- Some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`.
- The variable `i` probably has type `int` (a signed type), whereas `sizeof` produces a value of type `size_t` (an unsigned type).
- Comparing a signed integer with an unsigned integer can be dangerous, but in this case it's safe.

Using the `sizeof` Operator with Arrays

- To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

- Defining a macro for the size calculation is often helpful:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
  a[i] = 0;
```

Program: Computing Interest

- The `interest.c` program prints a table showing the value of $100 invested at different rates of interest over a period of years.
- The user will enter an interest rate and the number of years the money will be invested.
- The table will show the value of the money at one-year intervals—at that interest rate and the next four higher rates—assuming that interest is compounded once a year.

Program: Computing Interest

- Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5

Years     6%      7%      8%      9%     10%
  1     106.00  107.00  108.00  109.00  110.00
  2     112.36  114.49  116.64  118.81  121.00
  3     119.10  122.50  125.97  129.50  133.10
  4     126.25  131.08  136.05  141.16  146.41
  5     133.82  140.26  146.93  153.86  161.05
```


Program: Computing Interest

- The numbers in the second row depend on the numbers in the first row, so it makes sense to store the first row in an array.
  - The values in the array are then used to compute the second row.
  - This process can be repeated for the third and later rows.
- The program uses nested `for` statements.
  - The outer loop counts from 1 to the number of years requested by the user.
  - The inner loop increments the interest rate from its lowest value to its highest value.

interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
  int i, low_rate, num_years, year;
  double value[5];

  printf("Enter interest rate: ");
  scanf("%d", &low_rate);
  printf("Enter number of years: ");
  scanf("%d", &num_years);
```

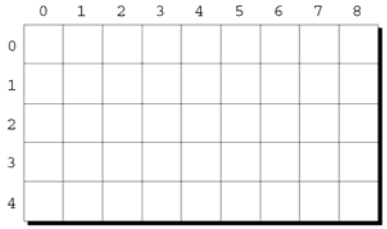
```
  printf("\nYears");
  for (i = 0; i < NUM_RATES; i++) {
    printf("%6d%%", low_rate + i);
    value[i] = INITIAL_BALANCE;
  }
  printf("\n");

  for (year = 1; year <= num_years; year++) {
    printf("%3d    ", year);
    for (i = 0; i < NUM_RATES; i++) {
      value[i] += (low_rate + i) / 100.0 * value[i];
      printf("%7.2f", value[i]);
    }
    printf("\n");
  }

  return 0;
}
```


Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

```
int m[5][9];
```

- `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

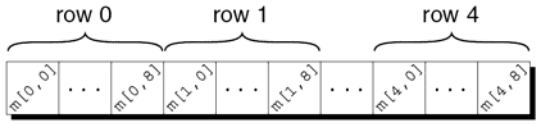


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.
- Resist the temptation to write `m[i,j]` instead of `m[i][j]`.
- C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```


Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.
- C provides a variety of ways to abbreviate initializers for multidimensional arrays

Initializing a Multidimensional Array

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0.
- The following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```


Initializing a Multidimensional Array

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Initializing a Multidimensional Array

- We can even omit the inner braces:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

  Once the compiler has seen enough values to fill one row, it begins filling the next.
- Omitting the inner braces can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer.

Initializing a Multidimensional Array

- C99's designated initializers work with multidimensional arrays.
- How to create 2 × 2 identity matrix:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

  As usual, all elements for which no value is specified will default to zero.

Constant Arrays

- An array can be made "constant" by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that's been declared `const` should not be modified by the program.

Constant Arrays

- Advantages of declaring an array to be `const`:
  - Documents that the program won't change the array.
  - Helps the compiler catch errors.
- `const` isn't limited to arrays, but it's particularly useful in array declarations.

Program: Dealing a Hand of Cards

- The `deal.c` program illustrates both two-dimensional arrays and constant arrays.
- The program deals a random hand from a standard deck of playing cards.
- Each card in a standard deck has a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace).

Program: Dealing a Hand of Cards

- The user will specify how many cards should be in the hand:

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

- Problems to be solved:
  - How do we pick cards randomly from the deck?
  - How do we avoid picking the same card twice?

Program: Dealing a Hand of Cards

- To pick cards randomly, we'll use several C library functions:
  - `time` (from `<time.h>`) – returns the current time, encoded in a single number.
  - `srand` (from `<stdlib.h>`) – initializes C's random number generator.
  - `rand` (from `<stdlib.h>`) – produces an apparently random number each time it's called.
- By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks).

Program: Dealing a Hand of Cards

- The `in_hand` array is used to keep track of which cards have already been chosen.
- The array has 4 rows and 13 columns; each element corresponds to one of the 52 cards in the deck.
- All elements of the array will be false to start with.
- Each time we pick a card at random, we'll check whether the element of `in_hand` corresponding to that card is true or false.
  - If it's true, we'll have to pick another card.
  - If it's false, we'll store `true` in that element to remind us later that this card has already been picked.

Program: Dealing a Hand of Cards

- Once we've verified that a card is "new," we'll need to translate its numerical rank and suit into characters and then display the card.
- To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays.
- These arrays won't change during program execution, so they are declared to be `const`.

deal.c

```
/* Deals a random hand of cards */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
  bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
  int num_cards, rank, suit;
  const char rank_code[] = {'2','3','4','5','6','7','8',
                            '9','t','j','q','k','a'};
  const char suit_code[] = {'c','d','h','s'};
```



```
  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand:");
  while (num_cards > 0) {
    suit = rand() % NUM_SUITS;     /* picks a random suit */
    rank = rand() % NUM_RANKS;     /* picks a random rank */
    if (!in_hand[suit][rank]) {
      in_hand[suit][rank] = true;
      num_cards--;
      printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
  }
  printf("\n");

  return 0;
}
```


Variable-Length Arrays (C99)

- In C89, the length of an array variable must be specified by a constant expression.
- In C99, however, it's sometimes possible to use an expression that's *not* constant.
- The `reverse2.c` program—a modification of `reverse.c`—illustrates this ability.

reverse2.c

```
/* Reverses a series of numbers using a variable-length
   array - C99 only */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
```



```
  printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```


Variable-Length Arrays (C99)

- The array `a` in the `reverse2.c` program is an example of a ***variable-length array*** (or ***VLA***).
- The length of a VLA is computed when the program is executed.
- The chief advantage of a VLA is that a program can calculate exactly how many elements are needed.
- If the programmer makes the choice, it's likely that the array will be too long (wasting memory) or too short (causing the program to fail).

Variable-Length Arrays (C99)

- The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions are legal:

```
int a[3*i+5];
int b[j+k];
```

- Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

- Restrictions on VLAs:
  - Can't have static storage duration (discussed in Chapter 18).
  - Can't have an initializer.

Chapter 9

Functions

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Defining and Calling Functions

- Before we go over the formal rules for defining a function, let's look at three simple programs that define functions.

Program: Computing Averages

- A function named `average` that computes the average of two `double` values:

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- The word `double` at the beginning is the ***return type*** of `average`.
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.

Program: Computing Averages

- Every function has an executable part, called the ***body,*** which is enclosed in braces.
- The body of `average` consists of a single `return` statement.
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.

Program: Computing Averages

- A function call consists of a function name followed by a list of ***arguments.***
  - `average(x, y)` is a call of the `average` function.
- Arguments are used to supply information to a function.
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program: Computing Averages

- We'll put the call of `average` in the place where we need to use the return value.
- A statement that prints the average of `x` and `y`:

```
printf("Average: %g\n", average(x, y));
```

  The return value of `average` isn't saved; the program prints it and then discards it.
- If we had needed the return value later in the program, we could have captured it in a variable:

```
avg = average(x, y);
```


Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```


countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```


Program: Printing a Countdown

- To indicate that a function has no return value, we specify that its return type is `void`:

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `void` is a type with no values.
- A call of `print_count` must appear in a statement by itself:

```
print_count(i);
```

- The `countdown.c` program calls `print_count` 10 times inside a loop.

Program: Printing a Pun (Revisited)

- When a function has no parameters, the word `void` is placed in parentheses after the function's name:

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

- To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

  The parentheses *must* be present.
- The `pun2.c` program tests the `print_pun` function.

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```


Function Definitions

- General form of a ***function definition:***

  return-type function-name (*parameters*)
  {
    declarations
    statements
  }

Function Definitions

- The return type of a function is the type of value that the function returns.
- Rules governing the return type:
  - Functions may not return arrays.
  - Specifying that the return type is `void` indicates that the function doesn't return a value.
- If the return type is omitted in C89, the function is presumed to return a value of type `int`.
- In C99, omitting the return type is illegal.

Function Definitions

- As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
  return (a + b) / 2;
}
```

- Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

Function Definitions

- After the function name comes a list of parameters.
- Each parameter is preceded by a specification of its type; parameters are separated by commas.
- If the function has no parameters, the word `void` should appear between the parentheses.

Function Definitions

- The body of a function may include both declarations and statements.
- An alternative version of the `average` function:

```
double average(double a, double b)
{
  double sum;        /* declaration */

  sum = a + b;       /* statement */
  return sum / 2;    /* statement */
}
```


Function Definitions

- Variables declared in the body of a function can't be examined or modified by other functions.
- In C89, variable declarations must come first, before all statements in the body of a function.
- In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Function Definitions

- The body of a function whose return type is `void` (a "`void` function") can be empty:

```
void print_pun(void)
{
}
```

- Leaving the body empty may make sense as a temporary step during program development.

Function Calls

- A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

- If the parentheses are missing, the function won't be called:

```
print_pun;    /*** WRONG ***/
```

This statement is legal but has no effect.

Function Calls

- A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

- A call of a non-`void` function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
  printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```


Function Calls

- The value returned by a non-`void` function can always be discarded if it's not needed:

```
average(x, y);  /* discards return value */
```

This call is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Function Calls

- Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense.
- `printf` returns the number of characters that it prints.
- After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

- We'll normally discard `printf`'s return value:

```
printf("Hi, Mom!\n");
  /* discards return value */
```

Function Calls

- To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

```
(void) printf("Hi, Mom!\n");
```

- Using `(void)` makes it clear to others that you deliberately discarded the return value, not just forgot that there was one.

Program: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:

```
Enter a number: 34
Not prime
```

- The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.
- `is_prime` divides its parameter `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, `n` isn't prime.

prime.c

```
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```



```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Function Declarations

- C doesn't require that the definition of a function precede its calls.
- Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`.

Function Declarations

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```


Function Declarations

- When the compiler encounters the first call of `average` in `main`, it has no information about the function.
- Instead of producing an error message, the compiler assumes that `average` returns an `int` value.
- We say that the compiler has created an ***implicit declaration*** of the function.

Function Declarations

- The compiler is unable to check that we're passing `average` the right number of arguments and that the arguments have the proper type.
- Instead, it performs the default argument promotions and hopes for the best.
- When it encounters the definition of `average` later in the program, the compiler notices that the function's return type is actually `double`, not `int`, and so we get an error message.

Function Declarations

- One way to avoid the problem of call-before-definition is to arrange the program so that the definition of each function precedes all its calls.
- Unfortunately, such an arrangement doesn't always exist.
- Even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

Function Declarations

- Fortunately, C offers a better solution: declare each function before calling it.
- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name (*parameters*) ;
- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations

```
#include <stdio.h>

double average(double a, double b);   /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)    /* DEFINITION */
{
  return (a + b) / 2;
}
```


Function Declarations

- Function declarations of the kind we're discussing are known as ***function prototypes.***
- C also has an older style of function declaration in which the parentheses are left empty.
- A function prototype doesn't have to specify the names of the function's parameters, as long as their types are present:

  ```
  double average(double, double);
  ```
- It's usually best not to omit parameter names.

Function Declarations

- C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function.
- Calling a function for which the compiler has not yet seen a declaration or definition is an error.

Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value assigned to the corresponding parameter.
- Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument.

Arguments

- The fact that arguments are passed by value has both advantages and disadvantages.
- Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, reducing the number of genuine variables needed.

Arguments

- Consider the following function, which raises a number `x` to a power `n`:

```
int power(int x, int n)
{
  int i, result = 1;

  for (i = 1; i <= n; i++)
    result = result * x;

  return result;
}
```

Arguments

- Since `n` is a *copy* of the original exponent, the function can safely modify it, removing the need for `i`:

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result = result * x;

  return result;
}
```


Arguments

- C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions.
- Suppose that we need a function that will decompose a `double` value into an integer part and a fractional part.
- Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

```
void decompose(double x, long int_part,
               double frac_part)
{
  int_part = (long) x;
  frac_part = x - int_part;
}
```


Arguments

- A call of the function:

```
decompose(3.14159, i, d);
```

- Unfortunately, `i` and `d` won't be affected by the assignments to `int_part` and `frac_part`.
- Chapter 11 shows how to make `decompose` work correctly.

Argument Conversions

- C allows function calls in which the types of the arguments don't match the types of the parameters.
- The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call.

Argument Conversions

- ***The compiler has encountered a prototype prior to the call.***
- The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment.
- Example: If an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.

Argument Conversions

- ***The compiler has not encountered a prototype prior to the call.***
- The compiler performs the ***default argument promotions:***
  - `float` arguments are converted to `double`.
  - The integral promotions are performed, causing `char` and `short` arguments to be converted to `int`. (In C99, the integer promotions are performed.)

Argument Conversions

- Relying on the default argument promotions is dangerous.
- Example:

```
#include <stdio.h>

int main(void)
{
  double x = 3.0;
  printf("Square: %d\n", square(x));

  return 0;
}

int square(int n)
{
  return n * n;
}
```

- At the time `square` is called, the compiler doesn't know that it expects an argument of type `int`.

Argument Conversions

- Instead, the compiler performs the default argument promotions on `x`, with no effect.
- Since it's expecting an argument of type `int` but has been given a `double` value instead, the effect of calling `square` is undefined.
- The problem can be fixed by casting `square`'s argument to the proper type:

```
printf("Square: %d\n", square((int) x));
```

- A much better solution is to provide a prototype for `square` before calling it.
- In C99, calling `square` without first providing a declaration or definition of the function is an error.

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[])  /* no length specified */
{
  …
}
```

- C doesn't provide any easy way for a function to determine the length of an array passed to it.
- Instead, we'll have to supply the length—if the function needs it—as an additional argument.

Array Arguments

- Example:

```
int sum_array(int a[], int n)
{
  int i, sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments

- The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

- As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```


Array Arguments

- When `sum_array` is called, the first argument will be the name of an array, and the second will be its length:

```
#define LEN 100

int main(void)
{
  int b[LEN], total;
  …
  total = sum_array(b, LEN);
  …
}
```

- Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);   /*** WRONG ***/
```

Array Arguments

- A function has no way to check that we've passed it the correct array length.
- We can exploit this fact by telling the function that the array is smaller than it really is.
- Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100.
- We can sum just the first 50 elements by writing

```
total = sum_array(b, 50);
```


Array Arguments

- Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150);    /*** WRONG ***/
```

`sum_array` will go past the end of the array, causing undefined behavior.

Array Arguments

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.
- A function that modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```


Array Arguments

- A call of `store_zeros`:

```
store_zeros(b, 100);
```

- The ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value.
- Chapter 12 explains why there's actually no contradiction.

Array Arguments

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```


Array Arguments

- Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance.
- We can often work around this difficulty by using arrays of pointers.
- C99's variable-length array parameters provide an even better solution.

Variable-Length Array Parameters (C99)

- C99 allows the use of variable-length arrays as parameters.
- Consider the `sum_array` function:

```
int sum_array(int a[], int n)
{
  …
}
```

  As it stands now, there's no direct link between `n` and the length of the array `a`.
- Although the function body treats `n` as `a`'s length, the actual length of the array could be larger or smaller than `n`.

Variable-Length Array Parameters (C99)

- Using a variable-length array parameter, we can explicitly state that `a`'s length is `n`:

```
int sum_array(int n, int a[n])
{
  …
}
```

- The value of the first parameter (`n`) specifies the length of the second parameter (`a`).
- Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.

Variable-Length Array Parameters (C99)

- There are several ways to write the prototype for the new version of `sum_array`.
- One possibility is to make it look exactly like the function definition:

```
int sum_array(int n, int a[n]);  /* Version 1 */
```

- Another possibility is to replace the array length by an asterisk (`*`):

```
int sum_array(int n, int a[*]);  /* Version 2a */
```


Variable-Length Array Parameters (C99)

- The reason for using the `*` notation is that parameter names are optional in function declarations.
- If the name of the first parameter is omitted, it wouldn't be possible to specify that the length of the array is `n`, but the `*` provides a clue that the length of the array is related to parameters that come earlier in the list:

```
int sum_array(int, int [*]);     /* Version 2b */
```


Variable-Length Array Parameters (C99)

- It's also legal to leave the brackets empty, as we normally do when declaring an array parameter:

```
int sum_array(int n, int a[]);   /* Version 3a */
int sum_array(int, int []);      /* Version 3b */
```

- Leaving the brackets empty isn't a good choice, because it doesn't expose the relationship between `n` and `a`.

Variable-Length Array Parameters (C99)

- In general, the length of a variable-length array parameter can be any expression.
- A function that concatenates two arrays `a` and `b`, storing the result into a third array named `c`:

```
int concatenate(int m, int n, int a[m], int b[n],
                int c[m+n])
{
  …
}
```

- The expression used to specify the length of `c` involves two other parameters, but in general it could refer to variables outside the function or even call other functions.

Variable-Length Array Parameters (C99)

- Variable-length array parameters with a single dimension have limited usefulness.
- They make a function declaration or definition more descriptive by stating the desired length of an array argument.
- However, no additional error-checking is performed; it's still possible for an array argument to be too long or too short.

Variable-Length Array Parameters (C99)

- Variable-length array parameters are most useful for multidimensional arrays.
- By using a variable-length array parameter, we can generalize the `sum_two_dimensional_array` function to any number of columns:

```
int sum_two_dimensional_array(int n, int m, int a[n][m])
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      sum += a[i][j];

  return sum;
}
```


Variable-Length Array Parameters (C99)

- Prototypes for this function include:

```
int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);
```


Using `static` in Array Parameter Declarations (C99)

- C99 allows the use of the keyword `static` in the declaration of array parameters.
- The following example uses `static` to indicate that the length of `a` is guaranteed to be at least 3:

```
int sum_array(int a[static 3], int n)
{
  …
}
```

Using `static` in Array Parameter Declarations (C99)

- Using `static` has no effect on program behavior.
- The presence of `static` is merely a "hint" that may allow a C compiler to generate faster instructions for accessing the array.
- If an array parameter has more than one dimension, `static` can be used only in the first dimension.

Compound Literals (C99)

- Let's return to the original `sum_array` function.
- When `sum_array` is called, the first argument is usually the name of an array.
- Example:

```
int b[] = {3, 0, 3, 4, 1};
total = sum_array(b, 5);
```

- `b` must be declared as a variable and then initialized prior to the call.
- If `b` isn't needed for any other purpose, it can be annoying to create it solely for the purpose of calling `sum_array`.

Compound Literals (C99)

- In C99, we can avoid this annoyance by using a ***compound literal:*** an unnamed array that's created "on the fly" by simply specifying which elements it contains.
- A call of `sum_array` with a compound literal (shown in **bold**) as its first argument:

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

- We didn't specify the length of the array, so it's determined by the number of elements in the literal.
- We also have the option of specifying a length explicitly:

```
(int [4]){1, 9, 2, 1}
```

is equivalent to

```
(int []){1, 9, 2, 1}
```


Compound Literals (C99)

- A compound literal resembles a cast applied to an initializer.
- In fact, compound literals and initializers obey the same rules.
- A compound literal may contain designators, just like a designated initializer, and it may fail to provide full initialization (in which case any uninitialized elements default to zero).
- For example, the literal `(int [10]){8, 6}` has 10 elements; the first two have the values 8 and 6, and the remaining elements have the value 0.

Compound Literals (C99)

- Compound literals created inside a function may contain arbitrary expressions, not just constants:

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

- A compound literal is an lvalue, so the values of its elements can be changed.
- If desired, a compound literal can be made "read-only" by adding the word `const` to its type:

```
(const int []){5, 4}
```


The `return` Statement

- A non-`void` function must use the `return` statement to specify what value it will return.
- The `return` statement has the form

```
return expression ;
```

- The expression is often just a constant or variable:

```
return 0;
return status;
```

- More complex expressions are possible:

```
return n >= 0 ? n : 0;
```


The `return` Statement

- If the type of the expression in a `return` statement doesn't match the function's return type, the expression will be implicitly converted to the return type.
  - If a function returns an `int`, but the `return` statement contains a `double` expression, the value of the expression is converted to `int`.

The `return` Statement

- `return` statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return;  /* return in a void function */
```

- Example:

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

The `return` Statement

- A `return` statement may appear at the end of a `void` function:

```
void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
  return;   /* OK, but not needed */
}
```

  Using `return` here is unnecessary.
- If a non-`void` function fails to execute a `return` statement, the behavior of the program is undefined if it attempts to use the function's return value.

Program Termination

- Normally, the return type of `main` is `int`:

```
int main(void)
{
  …
}
```

- Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
  …
}
```


Program Termination

- Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice.
- Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to include it.

Program Termination

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return 0 if the program terminates normally.
- To indicate abnormal termination, `main` should return a value other than 0.
- It's good practice to make sure that every C program returns a status code.

The `exit` Function

- Executing a `return` statement in `main` is one way to terminate a program.
- Another is calling the `exit` function, which belongs to `<stdlib.h>`.
- The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination.
- To indicate normal termination, we'd pass 0:

```
exit(0);    /* normal termination */
```


The `exit` Function

- Since 0 is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS);
```

- Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE);
```

- `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`.
- The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

The `exit` Function

- The statement

  `return` *expression*`;`

  in `main` is equivalent to

  `exit(`*expression*`);`
- The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it.
- The `return` statement causes program termination only when it appears in the `main` function.

Recursion

- A function is ***recursive*** if it calls itself.
- The following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Recursion

- To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

`fact(3)` finds that 3 is not less than or equal to 1, so it calls
`fact(2)`, which finds that 2 is not less than or equal to 1, so it calls
`fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing
`fact(2)` to return 2 x 1 = 2, causing
`fact(3)` to return 3 x 2 = 6.

Recursion

- To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

`fact(3)` finds that 3 is not less than or equal to 1, so it calls
`fact(2)`, which finds that 2 is not less than or equal to 1, so it calls
`fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing
`fact(2)` to return 2 x 1 = 2, causing
`fact(3)` to return 3 x 2 = 6.

Recursion

- We can condense the `power` function by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

- Both `fact` and `power` are careful to test a "termination condition" as soon as they're called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Recursion

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * power(x, n - 1);
}
```


The Quicksort Algorithm

- Recursion is most helpful for sophisticated algorithms that require a function to call itself two or more times.
- Recursion often arises as a result of an algorithm design technique known as ***divide-and-conquer,*** in which a large problem is divided into smaller pieces that are then tackled by the same algorithm.

The Quicksort Algorithm

- A classic example of divide-and-conquer can be found in the popular ***Quicksort*** algorithm.
- Assume that the array to be sorted is indexed from 1 to *n*.

Quicksort algorithm

1. Choose an array element *e* (the "partitioning element"), then rearrange the array so that elements 1, …, *i* – 1 are less than or equal to *e*, element *i* contains e, and elements *i* + 1, …, *n* are greater than or equal to *e*.
2. Sort elements 1, …, *i* – 1 by using Quicksort recursively.
3. Sort elements *i* + 1, …, *n* by using Quicksort recursively.

The Quicksort Algorithm

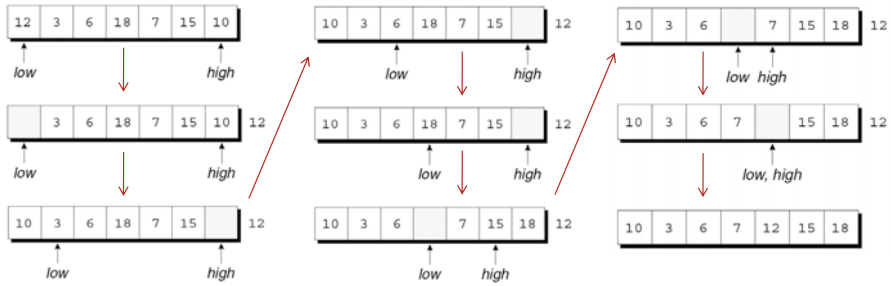
- Step 1 of the Quicksort algorithm is obviously critical.
- There are various methods to partition an array.
- We'll use a technique that's easy to understand but not particularly efficient.
- The algorithm relies on two "markers" named *low* and *high*, which keep track of positions within the array.

The Quicksort Algorithm

- Initially, *low* points to the first element; *high* points to the last.
- We copy the first element (the partitioning element) into a temporary location, leaving a "hole" in the array.
- Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element.
- We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*).
- We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to.
- The process repeats until *low* and *high* meet at a hole.
- Finally, we copy the partitioning element into the hole.

The Quicksort Algorithm

- Example of partitioning an array:



The Quicksort Algorithm

- By the final figure, all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12.
- Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).

Program: Quicksort

- Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers.
- The `qsort.c` program reads 10 numbers into an array, calls `quicksort` to sort the array, then prints the elements in the array:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

- The code for partitioning the array is in a separate function named `split`.

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

int main(void)
{
  int a[N], i;

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```



```
void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}
```

```
int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```


Program: Quicksort

- Ways to improve the program's performance:
  - Improve the partitioning algorithm.
  - Use a different method to sort small arrays.
  - Make Quicksort nonrecursive.

Chapter 10

Program Organization

Local Variables

- A variable declared in the body of a function is said to be ***local*** to the function:

```
int sum_digits(int n)
{
  int sum = 0;   /* local variable */

  while (n > 0) {
    sum += n % 10;
    n /= 10;
  }

  return sum;
}
```


Local Variables

- Default properties of local variables:
  - ***Automatic storage duration.*** Storage is "automatically" allocated when the enclosing function is called and deallocated when the function returns.
  - ***Block scope.*** A local variable is visible from its point of declaration to the end of the enclosing function body.

Local Variables

- Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope:

```
void f(void)
{
  …
  int i;
  …
}
```

scope of i

Static Local Variables

- Including `static` in the declaration of a local variable causes it to have ***static storage duration.***
- A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program.
- Example:

```
void f(void)
{
  static int i;    /* static local variable */
  …
}
```

- A static local variable still has block scope, so it's not visible to other functions.

Parameters

- Parameters have the same properties—automatic storage duration and block scope—as local variables.
- Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

External Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through ***external variables***—variables that are declared outside the body of any function.
- External variables are sometimes known as ***global variables.***

External Variables

- Properties of external variables:
  - Static storage duration
  - File scope
- Having ***file scope*** means that an external variable is visible from its point of declaration to the end of the enclosing file.

Example: Using External Variables to Implement a Stack

- To illustrate how external variables might be used, let's look at a data structure known as a ***stack.***
- A stack, like an array, can store multiple data items of the same type.
- The operations on a stack are limited:
  - ***Push*** an item (add it to one end—the "stack top")
  - ***Pop*** an item (remove it from the same end)
- Examining or modifying an item that's not at the top of the stack is forbidden.

Example: Using External Variables to Implement a Stack

- One way to implement a stack in C is to store its items in an array, which we'll call `contents`.
- A separate integer variable named `top` marks the position of the stack top.
  - When the stack is empty, `top` has the value 0.
- To *push* an item: Store it in `contents` at the position indicated by `top`, then increment `top`.
- To *pop* an item: Decrement `top`, then use it as an index into `contents` to fetch the item that's being popped.

Example: Using External Variables to Implement a Stack

- The following program fragment declares the `contents` and `top` variables for a stack.
- It also provides a set of functions that represent stack operations.
- All five functions need access to the `top` variable, and two functions need access to `contents`, so `contents` and `top` will be external.

Example: Using External Variables to Implement a Stack

```
#include <stdbool.h>   /* C99 only */

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
  top = 0;
}

bool is_empty(void)
{
  return top == 0;
}
```

Example: Using External Variables to Implement a Stack

```
bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full()
    stack_overflow();
  else
    contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
```


Pros and Cons of External Variables

- External variables are convenient when many functions must share a variable or when a few functions share a large number of variables.
- In most cases, it's better for functions to communicate through parameters rather than by sharing variables:
  - If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.
  - If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function.
  - Functions that rely on external variables are hard to reuse in other programs.

Pros and Cons of External Variables

- Don't use the same external variable for different purposes in different functions.
- Suppose that several functions need a variable named `i` to control a `for` statement.
- Instead of declaring `i` in each function that uses it, some programmers declare it just once at the top of the program.
- This practice is misleading; someone reading the program later may think that the uses of `i` are related, when in fact they're not.

Pros and Cons of External Variables

- Make sure that external variables have meaningful names.
- Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.

Pros and Cons of External Variables

- Making variables external when they should be local can lead to some rather frustrating bugs.
- Code that is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
  for (i = 1; i <= 10; i++)
    printf("*");
}

void print_all_rows(void)
{
  for (i = 1; i <= 10; i++) {
    print_one_row();
    printf("\n");
  }
}
```

- Instead of printing 10 rows, `print_all_rows` prints only one.

Program: Guessing a Number

- The `guess.c` program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible:

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!
```


Program: Guessing a Number

```
Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

- Tasks to be carried out by the program:
  - Initialize the random number generator
  - Choose a secret number
  - Interact with the user until the correct number is picked
- Each task can be handled by a separate function.

guess.c

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);
```

```
int main(void)
{
  char command;
  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```



```
/*************************************************************
 * initialize_number_generator: Initializes the random       *
 *                              number generator using       *
 *                              the time of day.             *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/*************************************************************
 * choose_new_secret_number: Randomly selects a number       *
 *                           between 1 and MAX_NUMBER and    *
 *                           stores it in secret_number.     *
 *************************************************************/
void choose_new_secret_number(void)
{
  secret_number = rand() % MAX_NUMBER + 1;
}
```



```
/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells     *
 *               the user whether each guess is too low,     *
 *               too high, or correct. When the guess is     *
 *               correct, prints the total number of         *
 *               guesses and returns.                        *
 *************************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```


Program: Guessing a Number

- Although `guess.c` works fine, it relies on the external variable `secret_number`.
- By altering `choose_new_secret_number` and `read_guesses` slightly, we can move `secret_number` into the `main` function.
- The new version of `guess.c` follows, with changes in **bold.**

guess2.c

```
/* Asks user to guess a hidden number */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);
```



```
int main(void)
{
  char command;
  int secret_number;

  printf("Guess the secret number between 1 and %d.\n\n",
         MAX_NUMBER);
  initialize_number_generator();
  do {
    secret_number = new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses(secret_number);
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}
```



```
/*************************************************************
 * initialize_number_generator: Initializes the random       *
 *                              number generator using       *
 *                              the time of day.             *
 *************************************************************/
void initialize_number_generator(void)
{
  srand((unsigned) time(NULL));
}

/*************************************************************
 * new_secret_number: Returns a randomly chosen number       *
 *                    between 1 and MAX_NUMBER.              *
 *************************************************************/
int new_secret_number(void)
{
  return rand() % MAX_NUMBER + 1;
}
```



```
/*************************************************************
 * read_guesses: Repeatedly reads user guesses and tells     *
 *               the user whether each guess is too low,     *
 *               too high, or correct. When the guess is     *
 *               correct, prints the total number of         *
 *               guesses and returns.                        *
 *************************************************************/
void read_guesses(int secret_number)
{
  int guess, num_guesses = 0;

  for (;;) {
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d", &guess);
    if (guess == secret_number) {
      printf("You won in %d guesses!\n\n", num_guesses);
      return;
    } else if (guess < secret_number)
      printf("Too low; try again.\n");
    else
      printf("Too high; try again.\n");
  }
}
```

Blocks

- In Section 5.2, we encountered compound statements of the form

  { *statements* }
- C allows compound statements to contain declarations as well as statements:

  { *declarations statements* }
- This kind of compound statement is called a ***block.***

Blocks

- Example of a block:

```
if (i > j) {
  /* swap values of i and j */
  int temp = i;
  i = j;
  j = temp;
}
```


Blocks

- By default, the storage duration of a variable declared in a block is automatic: storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- The variable has block scope; it can't be referenced outside the block.
- A variable that belongs to a block can be declared `static` to give it static storage duration.

Blocks

- The body of a function is a block.
- Blocks are also useful inside a function body when we need variables for temporary use.
- Advantages of declaring temporary variables in blocks:
  - Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly.
  - Reduces name conflicts.
- C99 allows variables to be declared anywhere within a block.

Scope

- In a C program, the same identifier may have several different meanings.
- C's scope rules enable the programmer (and the compiler) to determine which meaning is relevant at a given point in the program.
- The most important scope rule: When a declaration inside a block names an identifier that's already visible, the new declaration temporarily "hides" the old one, and the identifier takes on a new meaning.
- At the end of the block, the identifier regains its old meaning.

Scope

- In the example on the next slide, the identifier `i` has four different meanings:
  - In Declaration 1, `i` is a variable with static storage duration and file scope.
  - In Declaration 2, `i` is a parameter with block scope.
  - In Declaration 3, `i` is an automatic variable with block scope.
  - In Declaration 4, `i` is also automatic and has block scope.
- C's scope rules allow us to determine the meaning of `i` each time it's used (indicated by arrows).


```
int i;              /* Declaration 1 */

void f(int i)       /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;        /* Declaration 3 */

  if (i > 0) {
    int i;          /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```


Organizing a C Program

- Major elements of a C program:
  - Preprocessing directives such as `#include` and `#define`
  - Type definitions
  - Declarations of external variables
  - Function prototypes
  - Function definitions

Organizing a C Program

- C imposes only a few rules on the order of these items:
  - A preprocessing directive doesn't take effect until the line on which it appears.
  - A type name can't be used until it's been defined.
  - A variable can't be used until it's declared.
- It's a good idea to define or declare every function prior to its first call.
  - C99 makes this a requirement.

Organizing a C Program

- There are several ways to organize a program so that these rules are obeyed.
- One possible ordering:
  - `#include` directives
  - `#define` directives
  - Type definitions
  - Declarations of external variables
  - Prototypes for functions other than `main`
  - Definition of `main`
  - Definitions of other functions

Organizing a C Program

- It's a good idea to have a boxed comment preceding each function definition.
- Information to include in the comment:
  - Name of the function
  - Purpose of the function
  - Meaning of each parameter
  - Description of return value (if any)
  - Description of side effects (such as modifying external variables)

Program: Classifying a Poker Hand

- The `poker.c` program will classify a poker hand.
- Each card in the hand has a *suit* and a *rank*.
  - Suits: clubs, diamonds, hearts, spades
  - Ranks: two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace
- Jokers are not allowed, and aces are high.
- After reading a hand of five cards, the program will classify the hand using the categories on the next slide.
- If a hand falls into two or more categories, the program will choose the best one.

Program: Classifying a Poker Hand

- Categories (listed from best to worst):
  - straight flush (both a straight and a flush)
  - four-of-a-kind (four cards of the same rank)
  - full house (a three-of-a-kind and a pair)
  - flush (five cards of the same suit)
  - straight (five cards with consecutive ranks)
  - three-of-a-kind (three cards of the same rank)
  - two pairs
  - pair (two cards of the same rank)
  - high card (any other hand)

Program: Classifying a Poker Hand

- For input purposes, ranks and suits will be single letters (upper- or lower-case):

  Ranks: `2 3 4 5 6 7 8 9 t j q k a`
  Suits: `c d h s`
- Actions to be taken if the user enters an illegal card or tries to enter the same card twice:
  - Ignore the card
  - Issue an error message
  - Request another card
- Entering the number 0 instead of a card will cause the program to terminate.

Program: Classifying a Poker Hand

- A sample session with the program:

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush
```


Program: Classifying a Poker Hand

```
Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair
```

Program: Classifying a Poker Hand

```
Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
High card

Enter a card: 0
```


Program: Classifying a Poker Hand

- The program has three tasks:
  - Read a hand of five cards
  - Analyze the hand for pairs, straights, and so forth
  - Print the classification of the hand
- The functions `read_cards`, `analyze_hand`, and `print_result` will perform these tasks.
- `main` does nothing but call these functions inside an endless loop.

Program: Classifying a Poker Hand

- The functions will need to share a fairly large amount of information, so we'll have them communicate through external variables.
- `read_cards` will store information about the hand into several external variables.
- `analyze_hand` will then examine these variables, storing its findings into other external variables for the benefit of `print_result`.

Program: Classifying a Poker Hand

- Program outline:

```
/* #include directives go here */

/* #define directives go here */

/* declarations of external variables go here */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);
```

Program: Classifying a Poker Hand

```
/*************************************************************
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly.                                          *
 *************************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}

/*************************************************************
 * read_cards:  Reads the cards into external variables;  *
 *              checks for bad cards and duplicate cards. *
 *************************************************************/
void read_cards(void)
{
  …
}
```


Program: Classifying a Poker Hand

```
/*************************************************************
 * analyze_hand: Determines whether the hand contains a   *
 *               straight, a flush, four-of-a-kind,       *
 *               and/or three-of-a-kind; determines the   *
 *               number of pairs; stores the results into *
 *               external variables.                      *
 *************************************************************/
void analyze_hand(void)
{
  …
}

/*************************************************************
 * print_result: Notifies the user of the result, using   *
 *               the external variables set by            *
 *               analyze_hand.                            *
 *************************************************************/
void print_result(void)
{
  …
}
```


Program: Classifying a Poker Hand

- How should we represent the hand of cards?
- `analyze_hand` will need to know how many cards are in each rank and each suit.
- This suggests that we use two arrays, `num_in_rank` and `num_in_suit`.
  - `num_in_rank[r]` will be the number of cards with rank `r`.
  - `num_in_suit[s]` will be the number of cards with suit `s`.
- We'll encode ranks as numbers between 0 and 12.
- Suits will be numbers between 0 and 3.

Program: Classifying a Poker Hand

- We'll also need a third array, `card_exists`, so that `read_cards` can detect duplicate cards.
- Each time `read_cards` reads a card with rank `r` and suit `s`, it checks whether the value of `card_exists[r][s]` is true.
  - If so, the card was previously entered.
  - If not, `read_cards` assigns `true` to `card_exists[r][s]`.

Program: Classifying a Poker Hand

- Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so they will be external variables.
- The `card_exists` array is used only by `read_cards`, so it can be local to that function.
- As a rule, variables should be made external only if necessary.

poker.c

```
/* Classifies a poker hand */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5

/* external variables */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs;   /* can be 0, 1, or 2 */
```



```
/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*************************************************************
 * main: Calls read_cards, analyze_hand, and print_result    *
 *       repeatedly.                                         *
 *************************************************************/
int main(void)
{
  for (;;) {
    read_cards();
    analyze_hand();
    print_result();
  }
}
```



```
/*************************************************************
 * read_cards: Reads the cards into the external             *
 *             variables num_in_rank and num_in_suit;        *
 *             checks for bad cards and duplicate cards.     *
 *************************************************************/
void read_cards(void)
{
  bool card_exists[NUM_RANKS][NUM_SUITS];
  char ch, rank_ch, suit_ch;
  int rank, suit;
  bool bad_card;
  int cards_read = 0;

  for (rank = 0; rank < NUM_RANKS; rank++) {
    num_in_rank[rank] = 0;
    for (suit = 0; suit < NUM_SUITS; suit++)
      card_exists[rank][suit] = false;
  }

  for (suit = 0; suit < NUM_SUITS; suit++)
    num_in_suit[suit] = 0;
```

```
while (cards_read < NUM_CARDS) {
  bad_card = false;

  printf("Enter a card: ");

  rank_ch = getchar();
  switch (rank_ch) {
    case '0':           exit(EXIT_SUCCESS);
    case '2':           rank = 0; break;
    case '3':           rank = 1; break;
    case '4':           rank = 2; break;
    case '5':           rank = 3; break;
    case '6':           rank = 4; break;
    case '7':           rank = 5; break;
    case '8':           rank = 6; break;
    case '9':           rank = 7; break;
    case 't': case 'T': rank = 8; break;
    case 'j': case 'J': rank = 9; break;
    case 'q': case 'Q': rank = 10; break;
    case 'k': case 'K': rank = 11; break;
    case 'a': case 'A': rank = 12; break;
    default:            bad_card = true;
  }
```



```
  suit_ch = getchar();
  switch (suit_ch) {
    case 'c': case 'C': suit = 0; break;
    case 'd': case 'D': suit = 1; break;
    case 'h': case 'H': suit = 2; break;
    case 's': case 'S': suit = 3; break;
    default:            bad_card = true;
  }

  while ((ch = getchar()) != '\n')
    if (ch != ' ') bad_card = true;

  if (bad_card)
    printf("Bad card; ignored.\n");
  else if (card_exists[rank][suit])
    printf("Duplicate card; ignored.\n");
  else {
    num_in_rank[rank]++;
    num_in_suit[suit]++;
    card_exists[rank][suit] = true;
    cards_read++;
  }
 }
}
```



```
/*************************************************************
 * analyze_hand: Determines whether the hand contains a      *
 *               straight, a flush, four-of-a-kind,          *
 *               and/or three-of-a-kind; determines the      *
 *               number of pairs; stores the results into    *
 *               the external variables straight, flush,     *
 *               four, three, and pairs.                     *
 *************************************************************/
void analyze_hand(void)
{
  int num_consec = 0;
  int rank, suit;
  straight = false;
  flush = false;
  four = false;
  three = false;
  pairs = 0;
```



```
  /* check for flush */
  for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
      flush = true;

  /* check for straight */
  rank = 0;
  while (num_in_rank[rank] == 0) rank++;
  for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
  if (num_consec == NUM_CARDS) {
    straight = true;
    return;
  }

  /* check for 4-of-a-kind, 3-of-a-kind, and pairs */
  for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = true;
    if (num_in_rank[rank] == 3) three = true;
    if (num_in_rank[rank] == 2) pairs++;
  }
}
```

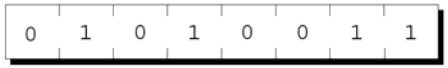
```
/*********************************************************
 * print_result: Prints the classification of the hand,   *
 *               based on the values of the external      *
 *               variables straight, flush, four, three,  *
 *               and pairs.                               *
 *********************************************************/
void print_result(void)
{
  if (straight && flush) printf("Straight flush");
  else if (four)         printf("Four of a kind");
  else if (three &&
           pairs == 1)   printf("Full house");
  else if (flush)        printf("Flush");
  else if (straight)     printf("Straight");
  else if (three)        printf("Three of a kind");
  else if (pairs == 2)   printf("Two pairs");
  else if (pairs == 1)   printf("Pair");
  else                   printf("High card");

  printf("\n\n");
}
```

Chapter 11

Pointers

Pointer Variables

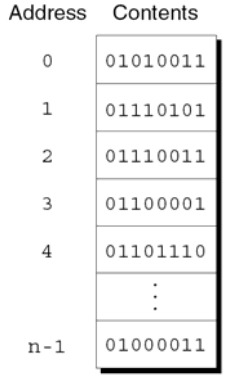
- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:



- Each byte has a unique ***address.***

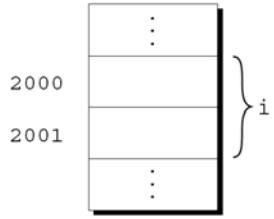
Pointer Variables

- If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$:



Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
- A graphical representation:

Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- `p` is a pointer variable capable of pointing to ***objects*** of type `int`.
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;     /* points only to integers    */
double *q;  /* points only to doubles     */
char *r;    /* points only to characters */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
  - To find the address of a variable, we use the `&` (address) operator.
  - To gain access to the object that a pointer points to, we use the `*` (***indirection***) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

- It's crucial to initialize `p` before we use it.

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
…
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:

The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:

```
int i;
int *p = &i;
```

- The declaration of `i` can even be combined with the declaration of `p`:

```
int i, *p = &i;
```


The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.
- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

- Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

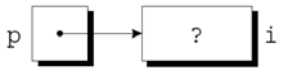
```
j = *&i;    /* same as j = i; */
```

The Indirection Operator

- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  - `*p` has the same value as `i`.
  - Changing the value of `*p` changes the value of `i`.
- The example on the next slide illustrates the equivalence of `*p` and `i`.

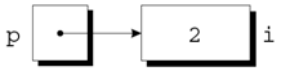
The Indirection Operator

```
p = &i;
```




```
i = 1;
```


```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```


The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```


Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

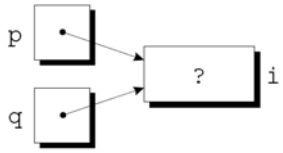
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:

  `q = p;`

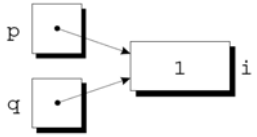
  `q` now points to the same place as `p`:



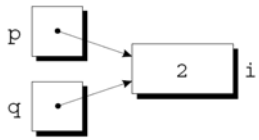
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:

  `*p = 1;`



  `*q = 2;`



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

  `q = p;`

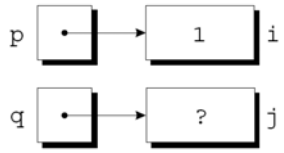
  with

  `*q = *p;`

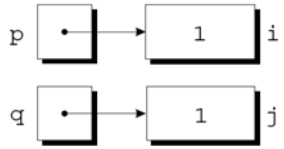
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```



`*q = *p;`



Pointers as Arguments

- In Chapter 9, we tried—and failed—to write a `decompose` function that could modify its arguments.
- By passing a *pointer* to a variable instead of the *value* of the variable, `decompose` can be fixed.

Pointers as Arguments

- New definition of `decompose`:

```
void decompose(double x, long *int_part,
               double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

- Possible prototypes for `decompose`:

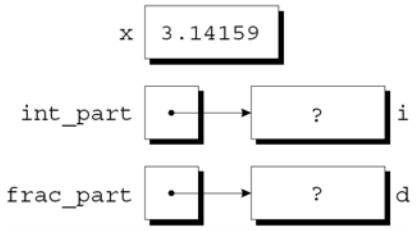
```
void decompose(double x, long *int_part,
               double *frac_part);
void decompose(double, long *, double *);
```


Pointers as Arguments

- A call of `decompose`:

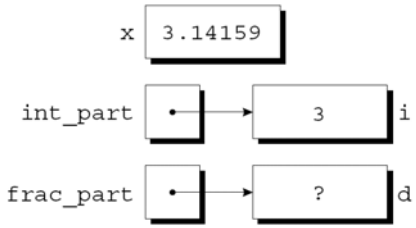
```
decompose(3.14159, &i, &d);
```

- As a result of the call, `int_part` points to `i` and `frac_part` points to `d`:



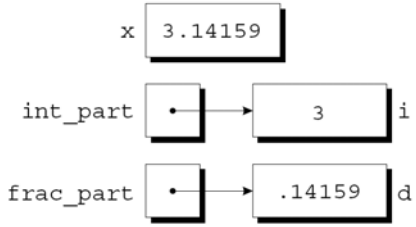
Pointers as Arguments

- The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`:



Pointers as Arguments

- The second assignment stores `x - *int_part` into the object that `frac_part` points to:



Pointers as Arguments

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

```
int i, *p;
…
p = &i;
scanf("%d", p);
```

- Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /*** WRONG ***/
```


Pointers as Arguments

- Arguments in calls of `scanf` are pointers:

```
int i;
…
scanf("%d", &i);
```

Without the `&`, `scanf` would be supplied with the *value* of `i`.

Pointers as Arguments

- Failing to pass a pointer to a function when one is expected can have disastrous results.
- A call of `decompose` in which the `&` operator is missing:

```
decompose(3.14159, i, d);
```

- When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.
- If we've provided a prototype for `decompose`, the compiler will detect the error.
- In the case of `scanf`, however, failing to pass pointers may go undetected.

Program: Finding the Largest and Smallest Elements in an Array

- The `max_min.c` program uses a function named `max_min` to find the largest and smallest elements in an array.
- Prototype for `max_min`:

```
void max_min(int a[], int n, int *max, int *min);
```

- Example call of `max_min`:

```
max_min(b, N, &big, &small);
```

- When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`.
- `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

Program: Finding the Largest and Smallest Elements in an Array

- `max_min.c` will read 10 numbers into an array, pass it to the `max_min` function, and print the results:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```


maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);
```



```
  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Using `const` to Protect Arguments

- When an argument is a pointer to a variable `x`, we normally assume that `x` will be modified:

```
f(&x);
```

- It's possible, though, that `f` merely needs to examine the value of `x`, not change it.
- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.

Using `const` to Protect Arguments

- We can use `const` to document that a function won't change an object whose address is passed to the function.
- `const` goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

Attempting to modify `*p` is an error that the compiler will detect.

Pointers as Return Values

- Functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A call of the `max` function:

```
int *p, i, j;
…
p = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

Pointers as Return Values

- Although `max` returns one of the pointers passed to it as an argument, that's not the only possibility.
- A function could also return a pointer to an external variable or to a static local variable.
- Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

The variable `i` won't exist after `f` returns.

Pointers as Return Values

- Pointers can point to array elements.
- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Chapter 12

Pointers and Arrays

Introduction

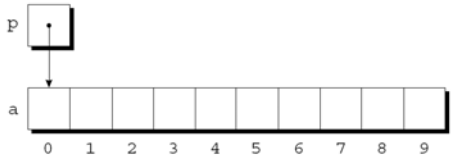
- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- The relationship between pointers and arrays in C is a close one.
- Understanding this relationship is critical for mastering C.

Pointer Arithmetic

- Chapter 11 showed that pointers can point to array elements:

```
int a[10], *p;
p = &a[0];
```

- A graphical representation:

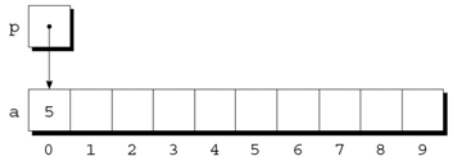


Pointer Arithmetic

- We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```

- An updated picture:



Pointer Arithmetic

- If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

Adding an Integer to a Pointer

- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
- Assume that the following declarations are in effect:

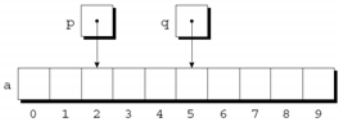
```
int a[10], *p, *q, i;
```


Adding an Integer to a Pointer

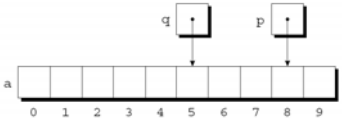
- Example of pointer addition:

```
p = &a[2];
```


```
q = p + 3;
```



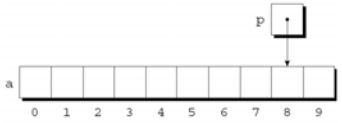

```
p += 6;
```



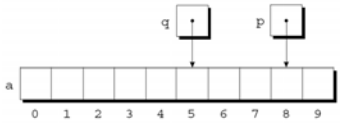
Subtracting an Integer from a Pointer

- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.
- Example:

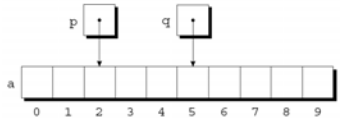
```
p = &a[8];
```




```
q = p - 3;
```



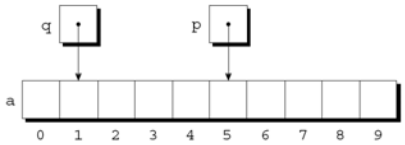

```
p -= 6;
```



Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.
- Example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```


Subtracting One Pointer from Another

- Operations that cause undefined behavior:
  - Performing arithmetic on a pointer that doesn't point to an array element
  - Subtracting pointers unless both point to elements of the same array

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
  - Using relational operators is meaningful only for pointers to elements of the same array.
- The outcome of the comparison depends on the relative positions of the two elements in the array.
- After the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

Pointers to Compound Literals (C99)

- It's legal for a pointer to point to an element within an array created by a compound literal:

```
int *p = (int []){3, 0, 3, 4, 1};
```

- Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

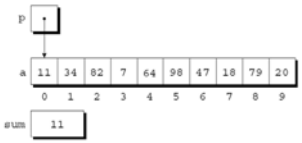
Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array `a`:

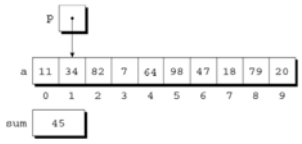
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```


Using Pointers for Array Processing

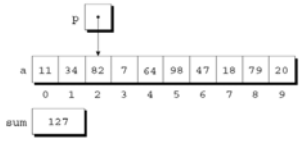
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



Using Pointers for Array Processing

- The condition `p < &a[N]` in the `for` statement deserves special mention.
- It's legal to apply the address operator to `a[N]`, even though this element doesn't exist.
- Pointer arithmetic may save execution time.
- However, some C compilers produce better code for loops that rely on subscripting.

Combining the * and ++ Operators

- C programmers often combine the `*` (indirection) and `++` operators.
- A statement that modifies an array element and then advances to the next element:

```
a[i++] = j;
```

- The corresponding pointer version:

```
*p++ = j;
```

- Because the postfix version of `++` takes precedence over `*`, the compiler sees this as

```
*(p++) = j;
```

Combining the * and ++ Operators

- Possible combinations of * and ++:

Expression	*Meaning*
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment

Combining the * and ++ Operators

- Most common combination of * and ++ is `*p++`, which is handy in loops.
- Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array `a`, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```


Combining the * and ++ Operators

- The * and -- operators mix in the same way as * and ++.
- For an application that combines * and --, let's return to the stack example of Chapter 10.
- The original version of the stack relied on an integer variable named `top` to keep track of the "top-of-stack" position in the contents array.
- Let's replace `top` by a pointer variable that points initially to element 0 of the `contents` array:

```
int *top_ptr = &contents[0];
```


Combining the * and ++ Operators

- The new `push` and `pop` functions:

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
- Another key relationship:

  The name of an array can be used as a pointer to the first element in the array.
- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

Using an Array Name as a Pointer

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of using `a` as a pointer:

```
*a = 7;   /* stores 7 in a[0] */
*(a+1) = 12;   /* stores 12 in a[1] */
```

- In general, `a + i` is the same as `&a[i]`.
  - Both represent a pointer to element `i` of `a`.
- Also, `*(a+i)` is equivalent to `a[i]`.
  - Both represent element `i` itself.

Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.
- Original loop:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

- Simplified version:

```
for (p = a; p < a + N; p++)
  sum += *p;
```


Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it's not possible to assign it a new value.
- Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;            /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

Program: Reversing a Series of Numbers (Revisited)

- The `reverse.c` program of Chapter 8 reads 10 numbers, then writes the numbers in reverse order.
- The original program stores the numbers in an array, with subscripting used to access elements of the array.
- `reverse3.c` is a new version of the program in which subscripting has been replaced with pointer arithmetic.

reverse3.c

```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```


Array Arguments (Revisited)

- When passed to a function, an array name is treated as a pointer.
- Example:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

- A call of `find_largest`:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

Array Arguments (Revisited)

- The fact that an array argument is treated as a pointer has some important consequences.
- *Consequence 1:* When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.
- In contrast, an array used as an argument isn't protected against change.

Array Arguments (Revisited)

- For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    a[i] = 0;
}
```


Array Arguments (Revisited)

- To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
  …
}
```

- If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

Array Arguments (Revisited)

- *Consequence 2:* The time required to pass an array to a function doesn't depend on the size of the array.
- There's no penalty for passing a large array, since no copy of the array is made.

Array Arguments (Revisited)

- *Consequence 3:* An array parameter can be declared as a pointer if desired.
- `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
  …
}
```

- Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Array Arguments (Revisited)

- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*.
- The following declaration causes the compiler to set aside space for 10 integers:

```
int a[10];
```

- The following declaration causes the compiler to allocate space for a pointer variable:

```
int *a;
```


Array Arguments (Revisited)

- In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results.
- For example, the assignment

```
*a = 0;   /*** WRONG ***/
```

will store 0 where `a` is pointing.

- Since we don't know where `a` is pointing, the effect on the program is undefined.

Array Arguments (Revisited)

- *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:

```
largest = find_largest(&b[5], 10);
```


Using a Pointer as an Array Name

- C allows us to subscript a pointer as though it were an array name:

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

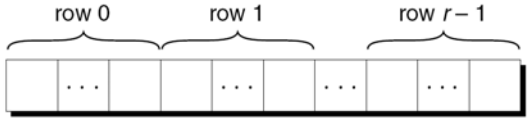
The compiler treats `p[i]` as `*(p+i)`.

Pointers and Multidimensional Arrays

- Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays.
- This section explores common techniques for using pointers to process the elements of multidimensional arrays.

Processing the Elements of a Multidimensional Array

- Chapter 8 showed that C stores two-dimensional arrays in row-major order.
- Layout of an array with *r* rows:



- If `p` initially points to the element in row 0, column 0, we can visit every element in the array by incrementing `p` repeatedly.

Processing the Elements of a Multidimensional Array

- Consider the problem of initializing all elements of the following array to zero:

```
int a[NUM_ROWS][NUM_COLS];
```

- The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

- If we view `a` as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```


Processing the Elements of a Multidimensional Array

- Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers.
- Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency.
- With many modern compilers, though, there's often little or no speed advantage.

Processing the Rows of a Multidimensional Array

- A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.
- To visit the elements of row `i`, we'd initialize `p` to point to element 0 in row `i` in the array `a`:

```
p = &a[i][0];
```

or we could simply write

```
p = a[i];
```


Processing the Rows of a Multidimensional Array

- For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first element in row `i`.
- To see why this works, recall that `a[i]` is equivalent to `*(a + i)`.
- Thus, `&a[i][0]` is the same as `&(*(a[i] + 0))`, which is equivalent to `&*a[i]`.
- This is the same as `a[i]`, since the `&` and `*` operators cancel.

Processing the Rows of a Multidimensional Array

- A loop that clears row `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

- Since `a[i]` is a pointer to row `i` of the array `a`, we can pass `a[i]` to a function that's expecting a one-dimensional array as its argument.
- In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.

Processing the Rows of a Multidimensional Array

- Consider `find_largest`, which was originally designed to find the largest element of a one-dimensional array.
- We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the Columns of a Multidimensional Array

- Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.
- A loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```


Using the Name of a Multidimensional Array as a Pointer

- The name of *any* array can be used as a pointer, regardless of how many dimensions it has, but some care is required.
- Example:

```
int a[NUM_ROWS][NUM_COLS];
```

`a` is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`.

- C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.
- When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Using the Name of a Multidimensional Array as a Pointer

- Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array.
- Instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```


Using the Name of a Multidimensional Array as a Pointer

- We can "trick" a function into thinking that a multidimensional array is really one-dimensional.
- A first attempt at using using `find_largest` to find the largest element in `a`:

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);
  /* WRONG */
```

This an error, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`.

- The correct call:

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

`a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler).

Pointers and Variable-Length Arrays (C99)

- Pointers are allowed to point to elements of variable-length arrays (VLAs).
- An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```


Pointers and Variable-Length Arrays (C99)

- When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first.
- A two-dimensional example:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.***

Pointers and Variable-Length Arrays (C99)

- The validity of an assignment such as `p = a` can't always be determined by the compiler.
- The following code will compile but is correct only if `m` and `n` are equal:

```
int a[m][n], (*p)[m];
p = a;
```

- If `m` is not equal to `n`, any subsequent use of `p` will cause undefined behavior.

Pointers and Variable-Length Arrays (C99)

- Variably modified types are subject to certain restrictions.
- The most important restriction: the declaration of a variably modified type must be inside the body of a function or in a function prototype.

Pointers and Variable-Length Arrays (C99)

- Pointer arithmetic works with VLAs.
- A two-dimensional VLA:

```
int a[m][n];
```

- A pointer capable of pointing to a row of `a`:

```
int (*p)[n];
```

- A loop that clears column `i` of `a`:

```
for (p = a; p < a + m; p++)
  (*p)[i] = 0;
```

Chapter 13

Strings

Introduction

- This chapter covers both string *constants* (or *literals,* as they're called in the C standard) and string *variables*.
- Strings are arrays of characters in which a special character—the null character—marks the end.
- The C library provides a collection of functions for working with strings.

String Literals

- A ***string literal*** is a sequence of characters enclosed within double quotes:

```
"When you come to a fork in the road, take it."
```

- String literals may contain escape sequences.
- Character escapes often appear in `printf` and `scanf` format strings.
- For example, each `\n` character in the string

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

causes the cursor to advance to the next line:

```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```


Continuing a String Literal

- The backslash character (`\`) can be used to continue a string literal from one line to the next:

```
printf("When you come to a fork in the road, take it. \
--Yogi Berra");
```

- In general, the `\` character can be used to join two or more lines of a program into a single line.

Continuing a String Literal

- There's a better way to deal with long string literals.
- When two or more string literals are adjacent, the compiler will join them into a single string.
- This rule allows us to split a string literal over two or more lines:

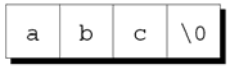
```
printf("When you come to a fork in the road, take it. "
       "--Yogi Berra");
```


How String Literals Are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.
- This memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.
- The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

How String Literals Are Stored

- The string literal `"abc"` is stored as an array of four characters:



- The string `""` is stored as a single null character:

How String Literals Are Stored

- Since a string literal is stored as an array, the compiler treats it as a pointer of type `char *`.
- Both `printf` and `scanf` expect a value of type `char *` as their first argument.
- The following call of `printf` passes the address of `"abc"` (a pointer to where the letter `a` is stored in memory):

```
printf("abc");
```

Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer:

```
char *p;

p = "abc";
```

- This assignment makes `p` point to the first character of the string.

Operations on String Literals

- String literals can be subscripted:

```
char ch;

ch = "abc"[1];
```

The new value of `ch` will be the letter `b`.

- A function that converts a number between 0 and 15 into the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```


Operations on String Literals

- Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';   /*** WRONG ***/
```

- A program that tries to change a string literal may crash or behave erratically.

String Literals versus Character Constants

- A string literal containing a single character isn't the same as a character constant.
  - `"a"` is represented by a *pointer*.
  - `'a'` is represented by an *integer*.
- A legal call of `printf`:

```
printf("\n");
```

- An illegal call:

```
printf('\n');   /*** WRONG ***/
```

String Variables

- Any one-dimensional array of characters can be used to store a string.
- A string must be terminated by a null character.
- Difficulties with this approach:
  - It can be hard to tell whether an array of characters is being used as a string.
  - String-handling functions must be careful to deal properly with the null character.
  - Finding the length of a string requires searching for the null character.

String Variables

- If a string variable needs to hold 80 characters, it must be declared to have length 81:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

- Adding 1 to the desired length allows room for the null character at the end of the string.
- Defining a macro that represents 80 and then adding 1 separately is a common practice.

String Variables

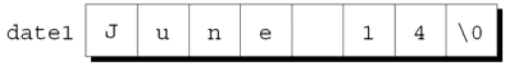
- Be sure to leave room for the null character when declaring a string variable.
- Failing to do so may cause unpredictable results when the program is executed.
- The actual length of a string depends on the position of the terminating null character.
- An array of `STR_LEN` + 1 characters can hold strings with lengths between 0 and `STR_LEN`.

Initializing a String Variable

- A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a null character so that `date1` can be used as a string:



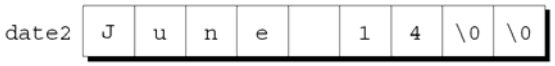
- `"June 14"` is not a string literal in this context.
- Instead, C views it as an abbreviation for an array initializer.

Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

Appearance of `date2`:

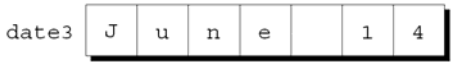


Initializing a String Variable

- An initializer for a string variable can't be longer than the variable, but it can be the same length:

```
char date3[7] = "June 14";
```

- There's no room for the null character, so the compiler makes no attempt to store one:



Initializing a String Variable

- The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

- The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character.
- Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Character Arrays versus Character Pointers

- The declaration

```
char date[] = "June 14";
```

declares `date` to be an *array,*

- The similar-looking

```
char *date = "June 14";
```

declares `date` to be a *pointer.*

- Thanks to the close relationship between arrays and pointers, either version can be used as a string.

Character Arrays versus Character Pointers

- However, there are significant differences between the two versions of `date`.
  - In the array version, the characters stored in `date` can be modified. In the pointer version, `date` points to a string literal that shouldn't be modified.
  - In the array version, `date` is an array name. In the pointer version, `date` is a variable that can point to other strings.

Character Arrays versus Character Pointers

- The declaration

  ```
  char *p;
  ```

  does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- One possibility is to make `p` point to a string variable:

  ```
  char str[STR_LEN+1], *p;

  p = str;
  ```
- Another possibility is to make `p` point to a dynamically allocated string.

Character Arrays versus Character Pointers

- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string `"abc"`:

  ```
  char *p;

  p[0] = 'a';     /*** WRONG ***/
  p[1] = 'b';     /*** WRONG ***/
  p[2] = 'c';     /*** WRONG ***/
  p[3] = '\0';    /*** WRONG ***/
  ```
- Since `p` hasn't been initialized, this causes undefined behavior.

Reading and Writing Strings

- Writing a string is easy using either `printf` or `puts`.
- Reading a string is a bit harder, because the input may be longer than the string variable into which it's being stored.
- To read a string in a single step, we can use either `scanf` or `gets`.
- As an alternative, we can read strings one character at a time.

Writing Strings Using `printf` and `puts`

- The `%s` conversion specification allows `printf` to write a string:

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

The output will be

```
Are we having fun yet?
```

- `printf` writes the characters in a string one by one until it encounters a null character.

Writing Strings Using `printf` and `puts`

- To print part of a string, use the conversion specification `%.ps`.
- *p* is the number of characters to be displayed.
- The statement

```
printf("%.6s\n", str);
```

will print

```
Are we
```


Writing Strings Using `printf` and `puts`

- The `%ms` conversion will display a string in a field of size *m*.
- If the string has fewer than *m* characters, it will be right-justified within the field.
- To force left justification instead, we can put a minus sign in front of *m*.
- The *m* and *p* values can be used in combination.
- A conversion specification of the form `%m.ps` causes the first *p* characters of a string to be displayed in a field of size *m*.

Writing Strings Using `printf` and `puts`

- `printf` isn't the only function that can write strings.
- The C library also provides `puts`:

```
puts(str);
```

- After writing a string, `puts` always writes an additional new-line character.

Reading Strings Using `scanf` and `gets`

- The `%s` conversion specification allows `scanf` to read a string into a character array:

```
scanf("%s", str);
```

- `str` is treated as a pointer, so there's no need to put the `&` operator in front of `str`.
- When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character.
- `scanf` always stores a null character at the end of the string.

Reading Strings Using `scanf` and `gets`

- `scanf` won't usually read a full line of input.
- A new-line character will cause `scanf` to stop reading, but so will a space or tab character.
- To read an entire line of input, we can use `gets`.
- Properties of `gets`:
  - Doesn't skip white space before starting to read input.
  - Reads until it finds a new-line character.
  - Discards the new-line character instead of storing it; the null character takes its place.

Reading Strings Using `scanf` and `gets`

- Consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

- Suppose that after the prompt

```
Enter a sentence:
```

the user enters the line

```
To C, or not to C: that is the question.
```

- `scanf` will store the string `"To"` in `sentence`.

Reading Strings Using `scanf` and `gets`

- Suppose that we replace `scanf` by `gets`:

```
gets(sentence);
```

- When the user enters the same input as before, `gets` will store the string

```
" To C, or not to C: that is the question."
```

in `sentence`.

Reading Strings Using `scanf` and `gets`

- As they read characters into an array, `scanf` and `gets` have no way to detect when it's full.
- Consequently, they may store characters past the end of the array, causing undefined behavior.
- `scanf` can be made safer by using the conversion specification `%`*n*`s` instead of `%s`.
- *n* is an integer indicating the maximum number of characters to be stored.
- `gets` is inherently unsafe; `fgets` is a much better alternative.

Reading Strings Character by Character

- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?
  - What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
  - What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Reading Strings Character by Character

- Suppose we need a function that (1) doesn't skip white-space characters, (2) stops reading at the first new-line character (which isn't stored in the string), and (3) discards extra characters.
- A prototype for the function:

```
int read_line(char str[], int n);
```

- If the input line contains more than `n` characters, `read_line` will discard the additional characters.
- `read_line` will return the number of characters it stores in `str`.

Reading Strings Character by Character

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left:

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';    /* terminates string */
  return i;         /* number of characters stored */
}
```

- `ch` has `int` type rather than `char` type because `getchar` returns an `int` value.

Reading Strings Character by Character

- Before returning, `read_line` puts a null character at the end of the string.
- Standard functions such as `scanf` and `gets` automatically put a null character at the end of an input string.
- If we're writing our own input function, we must take on that responsibility.

Accessing the Characters in a String

- Since strings are stored as arrays, we can use subscripting to access the characters in a string.
- To process every character in a string `s`, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

Accessing the Characters in a String

- A function that counts the number of spaces in a string:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```


Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripting :

```
int count_spaces(const char *s)
{
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

Accessing the Characters in a String

- Questions raised by the `count_spaces` example:
  - *Is it better to use array operations or pointer operations to access the characters in a string?* We can use either or both. Traditionally, C programmers lean toward using pointer operations.
  - *Should a string parameter be declared as an array or as a pointer?* There's no difference between the two.
  - *Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?* No.

Using the C String Library

- Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like.
- C's operators, in contrast, are essentially useless for working with strings.
- Strings are treated as arrays in C, so they're restricted in the same ways as arrays.
- In particular, they can't be copied or compared using operators.

Using the C String Library

- Direct attempts to copy or compare strings will fail.
- Copying a string into a character array using the = operator is not possible:

```
char str1[10], str2[10];
…
str1 = "abc";  /*** WRONG ***/
str2 = str1;   /*** WRONG ***/
```

  Using an array name as the left operand of = is illegal.
- *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

  In this context, = is not the assignment operator.

Using the C String Library

- Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) …   /*** WRONG ***/
```

- This statement compares `str1` and `str2` as *pointers*.
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Using the C String Library

- The C library provides a rich set of functions for performing operations on strings.
- Programs that need string operations should contain the following line:

```
#include <string.h>
```

- In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.

The `strcpy` (String Copy) Function

- Prototype for the `strcpy` function:

```
char *strcpy(char *s1, const char *s2);
```

- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

The `strcpy` (String Copy) Function

- A call of `strcpy` that stores the string `"abcd"` in `str2`:

```
strcpy(str2, "abcd");
  /* str2 now contains "abcd" */
```

- A call that copies the contents of `str2` into `str1`:

```
strcpy(str1, str2);
  /* str1 now contains "abcd" */
```


The `strcpy` (String Copy) Function

- In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the `str2` string will fit in the array pointed to by `str1`.
- If it doesn't, undefined behavior occurs.

The `strcpy` (String Copy) Function

- Calling the `strncpy` function is a safer, albeit slower, way to copy a string.
- `strncpy` has a third argument that limits the number of characters that will be copied.
- A call of `strncpy` that copies `str2` into `str1`:

```
strncpy(str1, str2, sizeof(str1));
```


The `strcpy` (String Copy) Function

- `strncpy` will leave `str1` without a terminating null character if the length of `str2` is greater than or equal to the size of the `str1` array.
- A safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- The second statement guarantees that `str1` is always null-terminated.

The `strlen` (String Length) Function

- Prototype for the `strlen` function:

```
size_t strlen(const char *s);
```

- `size_t` is a `typedef` name that represents one of C's unsigned integer types.

The `strlen` (String Length) Function

- `strlen` returns the length of a string `s`, not including the null character.
- Examples:

```
int len;

len = strlen("abc");  /* len is now 3 */
len = strlen("");     /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);   /* len is now 3 */
```

The `strcat` (String Concatenation) Function

- Prototype for the `strcat` function:

```
char *strcat(char *s1, const char *s2);
```

- `strcat` appends the contents of the string `s2` to the end of the string `s1`.
- It returns `s1` (a pointer to the resulting string).
- `strcat` examples:

```
strcpy(str1, "abc");
strcat(str1, "def");
  /* str1 now contains "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);
  /* str1 now contains "abcdef" */
```


The `strcat` (String Concatenation) Function

- As with `strcpy`, the value returned by `strcat` is normally discarded.
- The following example shows how the return value might be used:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
  /* str1 now contains "abcdefghi";
     str2 contains "defghi" */
```


The `strcat` (String Concatenation) Function

- `strcat(str1, str2)` causes undefined behavior if the `str1` array isn't long enough to accommodate the characters from `str2`.
- Example:

```
char str1[6] = "abc";

strcat(str1, "def");   /*** WRONG ***/
```

- `str1` is limited to six characters, causing `strcat` to write past the end of the array.

The `strcat` (String Concatenation) Function

- The `strncat` function is a safer but slower version of `strcat`.
- Like `strncpy`, it has a third argument that limits the number of characters it will copy.
- A call of `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- `strncat` will terminate `str1` with a null character, which isn't included in the third argument.

The strcmp (String Comparison) Function

- Prototype for the `strcmp` function:

```
int strcmp(const char *s1, const char *s2);
```

- `strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`.

The strcmp (String Comparison) Function

- Testing whether `str1` is less than `str2`:

```
if (strcmp(str1, str2) < 0)   /* is str1 < str2? */
  …
```

- Testing whether `str1` is less than or equal to `str2`:

```
if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */
  …
```

- By choosing the proper operator (<, <=, >, >=, ==, !=), we can test any possible relationship between `str1` and `str2`.

The strcmp (String Comparison) Function

- `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:
  - The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`.
  - All characters of `s1` match `s2`, but `s1` is shorter than `s2`.

The strcmp (String Comparison) Function

- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings.
- Some knowledge of the underlying character set is helpful to predict what `strcmp` will do.
- Important properties of ASCII:
  - A–Z, a–z, and 0–9 have consecutive codes.
  - All upper-case letters are less than all lower-case letters.
  - Digits are less than letters.
  - Spaces are less than all printing characters.

Program: Printing a One-Month Reminder List

- The `remind.c` program prints a one-month list of daily reminders.
- The user will enter a series of reminders, with each prefixed by a day of the month.
- When the user enters 0 instead of a valid day, the program will print a list of all reminders entered, sorted by day.
- The next slide shows a session with the program.

Program: Printing a One-Month Reminder List

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused"
 24 Susan's birthday
 26 Movie - "Chinatown"
```


Program: Printing a One-Month Reminder List

- Overall strategy:
  - Read a series of day-and-reminder combinations.
  - Store them in order (sorted by day).
  - Display them.
- `scanf` will be used to read the days.
- `read_line` will be used to read the reminders.

Program: Printing a One-Month Reminder List

- The strings will be stored in a two-dimensional array of characters.
- Each row of the array contains one string.
- Actions taken after the program reads a day and its associated reminder:
  - Search the array to determine where the day belongs, using `strcmp` to do comparisons.
  - Use `strcpy` to move all strings below that point down one position.
  - Copy the day into the array and call `strcat` to append the reminder to the day.

Program: Printing a One-Month Reminder List

- One complication: how to right-justify the days in a two-character field.
- A solution: use `scanf` to read the day into an integer variable, than call `sprintf` to convert the day back into string form.
- `sprintf` is similar to `printf`, except that it writes output into a string.
- The call

  ```
  sprintf(day_str, "%2d", day);
  ```

  writes the value of `day` into `day_str`.

Program: Printing a One-Month Reminder List

- The following call of `scanf` ensures that the user doesn't enter more than two digits:

  ```
  scanf("%2d", &day);
  ```


remind.c

```
/* Prints a one-month reminder list */

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

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);

int main(void)
{
  char reminders[MAX_REMIND][MSG_LEN+3];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }
```



```
    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      strcpy(reminders[j], reminders[j-1]);

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```


String Idioms

- Functions that manipulate strings are a rich source of idioms.
- We'll explore some of the most famous idioms by using them to write the `strlen` and `strcat` functions.

Searching for the End of a String

- A version of `strlen` that searches for the end of a string, using a variable to keep track of the string's length:

```
size_t strlen(const char *s)
{
  size_t n;

  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```


Searching for the End of a String

- To condense the function, we can move the initialization of `n` to its declaration:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

Searching for the End of a String

- The condition `*s != '\0'` is the same as `*s != 0`, which in turn is the same as `*s`.
- A version of `strlen` that uses these observations:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s; s++)
    n++;
  return n;
}
```


Searching for the End of a String

- The next version increments `s` and tests `*s` in the same expression:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s++;)
    n++;
  return n;
}
```


Searching for the End of a String

- Replacing the `for` statement with a `while` statement gives the following version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while (*s++)
    n++;
  return n;
}
```


Searching for the End of a String

- Although we've condensed `strlen` quite a bit, it's likely that we haven't increased its speed.
- A version that *does* run faster, at least with some compilers:

```
size_t strlen(const char *s)
{
  const char *p = s;

  while (*s)
    s++;
  return s - p;
}
```

Searching for the End of a String

- Idioms for "search for the null character at the end of a string":

```
while (*s)      while (*s++)
  s++;            ;
```

- The first version leaves `s` pointing to the null character.
- The second version is more concise, but leaves `s` pointing just past the null character.

Copying a String

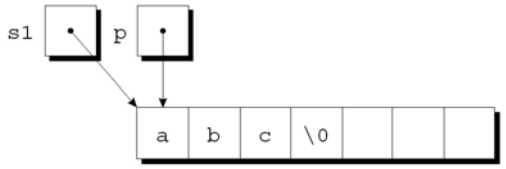
- Copying a string is another common operation.
- To introduce C's "string copy" idiom, we'll develop two versions of the `strcat` function.
- The first version of `strcat` (next slide) uses a two-step algorithm:
  - Locate the null character at the end of the string `s1` and make `p` point to it.
  - Copy characters one by one from `s2` to where `p` is pointing.

Copying a String

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

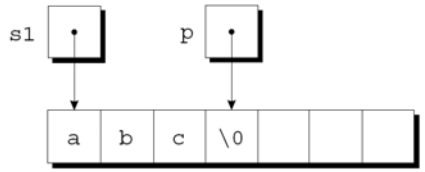

Copying a String

- `p` initially points to the first character in the `s1` string:



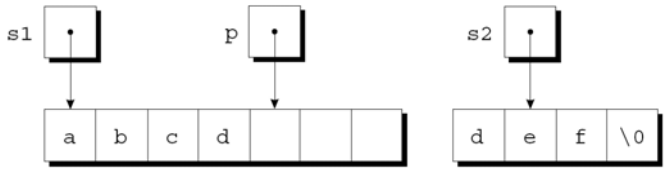
Copying a String

- The first `while` statement locates the null character at the end of `s1` and makes `p` point to it:



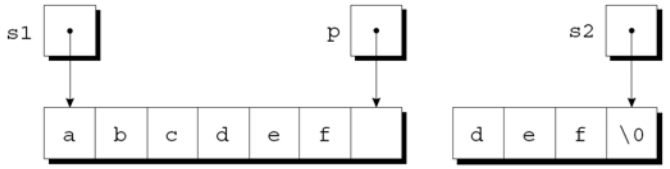
Copying a String

- The second `while` statement repeatedly copies one character from where `s2` points to where `p` points, then increments both `p` and `s2`.
- Assume that `s2` originally points to the string `"def"`.
- The strings after the first loop iteration:



Copying a String

- The loop terminates when `s2` points to the null character:



- After putting a null character where `p` is pointing, `strcat` returns.

Copying a String

- Condensed version of `strcat`:

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
}
```

Copying a String

- The heart of the streamlined `strcat` function is the "string copy" idiom:

```
while (*p++ = *s2++)
  ;
```

- Ignoring the two ++ operators, the expression inside the parentheses is an assignment:

```
*p = *s2
```

- After the assignment, `p` and `s2` are incremented.
- Repeatedly evaluating this expression copies characters from where `s2` points to where `p` points.

Copying a String

- But what causes the loop to terminate?
- The `while` statement tests the character that was copied by the assignment `*p = *s2`.
- All characters except the null character test true.
- The loop terminates *after* the assignment, so the null character will be copied.

Arrays of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

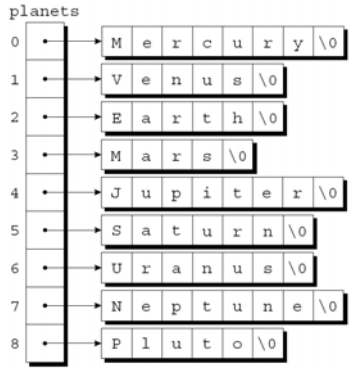
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- What we need is a ***ragged array,*** whose rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```


Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```


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

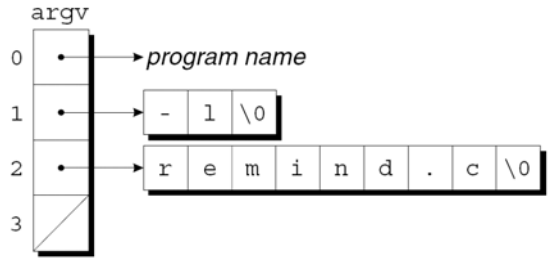
- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `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;
}
```

Chapter 14

The Preprocessor

Introduction

- Directives such as `#define` and `#include` are handled by the ***preprocessor,*** a piece of software that edits C programs just prior to compilation.
- Its reliance on a preprocessor makes C (along with C++) unique among major programming languages.
- The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs.

How the Preprocessor Works

- The preprocessor looks for ***preprocessing directives,*** which begin with a `#` character.
- We've encountered the `#define` and `#include` directives before.
- `#define` defines a ***macro***—a name that represents something else, such as a constant.
- The preprocessor responds to a `#define` directive by storing the name of the macro along with its definition.
- When the macro is used later, the preprocessor "expands" the macro, replacing it by its defined value.

How the Preprocessor Works

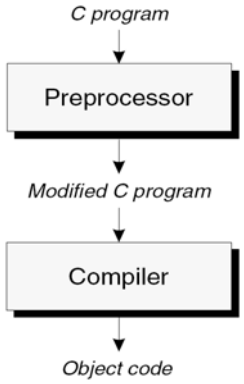
- `#include` tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.
- For example, the line

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

  instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program.

How the Preprocessor Works

- The preprocessor's role in the compilation process:



How the Preprocessor Works

- The input to the preprocessor is a C program, possibly containing directives.
- The preprocessor executes these directives, removing them in the process.
- The preprocessor's output goes directly into the compiler.

How the Preprocessor Works

- The `celsius.c` program of Chapter 2:

```
/* 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 is: %.1f\n", celsius);

  return 0;
}
```


How the Preprocessor Works

- The program after preprocessing:

```
Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line
int main(void)
{
  float fahrenheit, celsius;

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

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

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

  return 0;
}
```

How the Preprocessor Works

- The preprocessor does a bit more than just execute directives.
- In particular, it replaces each comment with a single space character.
- Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

How the Preprocessor Works

- In the early days of C, the preprocessor was a separate program.
- Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code.
- Still, it's useful to think of the preprocessor as separate from the compiler.

How the Preprocessor Works

- Most C compilers provide a way to view the output of the preprocessor.
- Some compilers generate preprocessor output when a certain option is specified (GCC will do so when the `-E` option is used).
- Others come with a separate program that behaves like the integrated preprocessor.

How the Preprocessor Works

- A word of caution: The preprocessor has only a limited knowledge of C.
- As a result, it's quite capable of creating illegal programs as it executes directives.
- In complicated programs, examining the output of the preprocessor may prove useful for locating this kind of error.

Preprocessing Directives

- Most preprocessing directives fall into one of three categories:
  - ***Macro definition.*** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
  - ***File inclusion.*** The `#include` directive causes the contents of a specified file to be included in a program.
  - ***Conditional compilation.*** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program.

Preprocessing Directives

- Several rules apply to all directives.
- ***Directives always begin with the # symbol.***

  The # symbol need not be at the beginning of a line, as long as only white space precedes it.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.***

  Example:

```
#    define    N    100
```


Preprocessing Directives

- ***Directives always end at the first new-line character, unless explicitly continued.***

  To continue a directive to the next line, end the current line with a \ character:

```
#define DISK_CAPACITY (SIDES *              \
                       TRACKS_PER_SIDE *    \
                       SECTORS_PER_TRACK * \
                       BYTES_PER_SECTOR)
```


Preprocessing Directives

- ***Directives can appear anywhere in a program.***

  Although `#define` and `#include` directives usually appear at the beginning of a file, other directives are more likely to show up later.
- ***Comments may appear on the same line as a directive.***

  It's good practice to put a comment at the end of a macro definition:

```
#define FREEZING_PT 32.0f /* freezing point of water */
```

Macro Definitions

- The macros that we've been using since Chapter 2 are known as *simple* macros, because they have no parameters.
- The preprocessor also supports *parameterized* macros.

Simple Macros

- Definition of a ***simple macro*** (or ***object-like macro***):

  `#define` *identifier replacement-list*

  replacement-list is any sequence of ***preprocessing tokens.***
- The replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation.
- Wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

Simple Macros

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:

```
#define N = 100   /*** WRONG ***/
…
int a[N];         /* becomes int a[= 100]; */
```


Simple Macros

- Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;   /*** WRONG ***/
…
int a[N];        /* becomes int a[100;]; */
```

- The compiler will detect most errors caused by extra symbols in a macro definition.
- Unfortunately, the compiler will flag each use of the macro as incorrect, rather than identifying the actual culprit: the macro's definition.

Simple Macros

- Simple macros are primarily used for defining "manifest constants"—names that represent numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```


Simple Macros

- Advantages of using `#define` to create names for constants:
  - ***It makes programs easier to read.*** The name of the macro can help the reader understand the meaning of the constant.
  - ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition.
  - ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like 3.14159 appears many times in a program, chances are it will occasionally be written 3.1416 or 3.14195 by accident.

Simple Macros

- Simple macros have additional uses.
- ***Making minor changes to the syntax of C***

  Macros can serve as alternate names for C symbols:

```
#define BEGIN {
#define END   }
#define LOOP for (;;)
```

  Changing the syntax of C usually isn't a good idea, since it can make programs harder for others to understand.

Simple Macros

- ***Renaming types***

  An example from Chapter 5:

```
#define BOOL int
```

  Type definitions are a better alternative.
- ***Controlling conditional compilation***

  Macros play an important role in controlling conditional compilation.

  A macro that might indicate "debugging mode":

```
#define DEBUG
```

Simple Macros

- When macros are used as constants, C programmers customarily capitalize all letters in their names.
- However, there's no consensus as to how to capitalize macros used for other purposes.
  - Some programmers like to draw attention to macros by using all upper-case letters in their names.
  - Others prefer lower-case names, following the style of K&R.

Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):

  `#define` *identifier* `(` x_1 `,` x_2 `,` … `,` x_n `)` *replacement-list*

  $x_1, x_2, \ldots, x_n$ are identifiers (the macro's ***parameters***).
- The parameters may appear as many times as desired in the replacement list.
- There must be *no space* between the macro name and the left parenthesis.
- If space is left, the preprocessor will treat $(x_1, x_2, \ldots, x_n)$ as part of the replacement list.

Parameterized Macros

- When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use.
- Wherever a macro ***invocation*** of the form *identifier* $(y_1, y_2, \ldots, y_n)$ appears later in the program, the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.
- Parameterized macros often serve as simple functions.

Parameterized Macros

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Parameterized Macros

- A more complicated function-like macro:

```
#define TOUPPER(c) \
  ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

- The <ctype.h> header provides a similar function named `toupper` that's more portable.
- A parameterized macro may have an empty parameter list:

```
#define getchar() getc(stdin)
```

- The empty parameter list isn't really needed, but it makes `getchar` resemble a function.

Parameterized Macros

- Using a parameterized macro instead of a true function has a couple of advantages:
  - ***The program may be slightly faster.*** A function call usually requires some overhead during program execution, but a macro invocation does not.
  - ***Macros are "generic."*** A macro can accept arguments of any type, provided that the resulting program is valid.

Parameterized Macros

- Parameterized macros also have disadvantages.
- ***The compiled code will often be larger.***

  Each macro invocation increases the size of the source program (and hence the compiled code).

  The problem is compounded when macro invocations are nested:

```
n = MAX(i, MAX(j, k));
```

  The statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```


Parameterized Macros

- ***Arguments aren't type-checked.***

  When a function is called, the compiler checks each argument to see if it has the appropriate type.

  Macro arguments aren't checked by the preprocessor, nor are they converted.
- ***It's not possible to have a pointer to a macro.***

  C allows pointers to functions, a useful concept.

  Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro."

Parameterized Macros

- ***A macro may evaluate its arguments more than once.***

  Unexpected behavior may occur if an argument has side effects:

  ```
  n = MAX(i++, j);
  ```

  The same line after preprocessing:

  ```
  n = ((i++)>(j)?(i++):(j));
  ```

  If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Parameterized Macros

- Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call.
- To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects.
- For self-protection, it's a good idea to avoid side effects in arguments.

Parameterized Macros

- Parameterized macros can be used as patterns for segments of code that are often repeated.
- A macro that makes it easier to display integers:

  ```
  #define PRINT_INT(n) printf("%d\n", n)
  ```

- The preprocessor will turn the line

  ```
  PRINT_INT(i/j);
  ```

  into

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


The # Operator

- Macro definitions may contain two special operators, `#` and `##`.
- Neither operator is recognized by the compiler; instead, they're executed during preprocessing.
- The `#` operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro.
- The operation performed by `#` is known as "stringization."

The # Operator

- There are a number of uses for #; let's consider just one.
- Suppose that we decide to use the PRINT_INT macro during debugging as a convenient way to print the values of integer variables and expressions.
- The # operator makes it possible for PRINT_INT to label each value that it prints.

The # Operator

- Our new version of PRINT_INT:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

- The invocation

```
PRINT_INT(i/j);
```

  will become

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

- The compiler automatically joins adjacent string literals, so this statement is equivalent to

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


The ## Operator

- The ## operator can "paste" two tokens together to form a single token.
- If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument.

The ## Operator

- A macro that uses the ## operator:

```
#define MK_ID(n) i##n
```

- A declaration that invokes MK_ID three times:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

- The declaration after preprocessing:

```
int i1, i2, i3;
```

The ## Operator

- The ## operator has a variety of uses.
- Consider the problem of defining a `max` function that behaves like the `MAX` macro described earlier.
- A single `max` function usually isn't enough, because it will only work for arguments of one type.
- Instead, we can write a macro that expands into the definition of a `max` function.
- The macro's parameter will specify the type of the arguments and the return value.

The ## Operator

- There's just one snag: if we use the macro to create more than one function named `max`, the program won't compile.
- To solve this problem, we'll use the ## operator to create a different name for each version of `max`:

```
#define GENERIC_MAX(type)       \
type type##_max(type x, type y) \
{                               \
  return x > y ? x : y;         \
}
```

- An invocation of this macro:

```
GENERIC_MAX(float)
```

- The resulting function definition:

```
float float_max(float x, float y) { return x > y ? x : y; }
```


General Properties of Macros

- Several rules apply to both simple and parameterized macros.
- ***A macro's replacement list may contain invocations of other macros.***

  Example:

```
#define PI      3.14159
#define TWO_PI (2*PI)
```

  When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.

  The preprocessor then ***rescans*** the replacement list to see if it contains invocations of other macros.

General Properties of Macros

- ***The preprocessor replaces only entire tokens.***

  Macro names embedded in identifiers, character constants, and string literals are ignored.

  Example:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

  Appearance after preprocessing:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

General Properties of Macros

- ***A macro definition normally remains in effect until the end of the file in which it appears.***

  Macros don't obey normal scope rules.

  A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.
- ***A macro may not be defined twice unless the new definition is identical to the old one.***

  Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.

General Properties of Macros

- ***Macros may be "undefined" by the `#undef` directive.***

  The `#undef` directive has the form

  `#undef` *identifier*

  where *identifier* is a macro name.

  One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses in order to avoid unexpected results.
- If the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

- Also, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

- Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions.

Parentheses in Macro Definitions

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

  becomes

```
conversion_factor = 360/2*3.14159;
```

  The division will be performed before the multiplication.

Parentheses in Macro Definitions

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

This statement is equivalent to

```
j = i+10;
```


Creating Longer Macros

- The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions.
- A macro that reads a string and then prints it:

```
#define ECHO(s) (gets(s), puts(s))
```

- Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator.
- We can invoke `ECHO` as though it were a function:

```
ECHO(str); /* becomes (gets(str), puts(str)); */
```


Creating Longer Macros

- An alternative definition of `ECHO` that uses braces:

```
#define ECHO(s) { gets(s); puts(s); }
```

- Suppose that we use `ECHO` in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

- Replacing `ECHO` gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```


Creating Longer Macros

- The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
  { gets(str); puts(str); }
```

- It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`.
- We could solve the problem by remembering not to put a semicolon after each invocation of `ECHO`, but then the program would look odd.

Creating Longer Macros

- The comma operator solves this problem for ECHO, but not for all macros.
- If a macro needs to contain a series of *statements,* not just a series of *expressions,* the comma operator is of no help.
- The solution is to wrap the statements in a do loop whose condition is false:

```
do { … } while (0)
```

- Notice that the do statement needs a semicolon at the end.

Creating Longer Macros

- A modified version of the ECHO macro:

```
#define ECHO(s)     \
        do {        \
          gets(s);  \
          puts(s);  \
        } while (0)
```

- When ECHO is used, it must be followed by a semicolon, which completes the do statement:

```
ECHO(str);
/* becomes
   do { gets(str); puts(str); } while (0); */
```


Predefined Macros

- C has several predefined macros, each of which represents an integer constant or string literal.
- The __DATE__ and __TIME__ macros identify when a program was compiled.
- Example of using __DATE__ and __TIME__:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

- Output produced by these statements:

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

- This information can be helpful for distinguishing among different versions of the same program.

Predefined Macros

- We can use the __LINE__ and __FILE__ macros to help locate errors.
- A macro that can help pinpoint the location of a division by zero:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

- The CHECK_ZERO macro would be invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

Predefined Macros

- If `j` happens to be zero, a message of the following form will be printed:

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```

- Error-detecting macros like this one are quite useful.
- In fact, the C library has a general-purpose error-detecting macro named `assert`.
- The remaining predefined macro is named `__STDC__`.
- This macro exists and has the value 1 if the compiler conforms to the C standard (either C89 or C99).

Additional Predefined Macros in C99

- C99 provides a few additional predefined macros.
- The `__STDC_HOSTED__` macro represents the constant 1 if the compiler is a hosted implementation. Otherwise, the macro has the value 0.
- An ***implementation*** of C consists of the compiler plus other software necessary to execute C programs.
- A ***hosted implementation*** must accept any program that conforms to the C99 standard.
- A ***freestanding implementation*** doesn't have to compile programs that use complex types or standard headers beyond a few of the most basic.

Additional Predefined Macros in C99

- The `__STDC_VERSION__` macro provides a way to check which version of the C standard is recognized by the compiler.
  - If a compiler conforms to the C89 standard, including Amendment 1, the value is `199409L`.
  - If a compiler conforms to the C99 standard, the value is `199901L`.

Additional Predefined Macros in C99

- A C99 compiler will define up to three additional macros, but only if the compiler meets certain requirements:

`__STDC_IEC_559__` is defined (and has the value 1) if the compiler performs floating-point arithmetic according to IEC 60559.

`__STDC_IEC_559_COMPLEX__` is defined (and has the value 1) if the compiler performs complex arithmetic according to IEC 60559.

`__STDC_ISO_10646__` is defined as *yyyymm*`L` if wide characters are represented by the codes in ISO/IEC 10646 (with revisions as of the specified year and month).

Empty Macro Arguments (C99)

- C99 allows any or all of the arguments in a macro call to be empty.
- Such a call will contain the same number of commas as a normal call.
- Wherever the corresponding parameter name appears in the replacement list, it's replaced by nothing.

Empty Macro Arguments (C99)

- Example:

```
#define ADD(x,y) (x+y)
```

- After preprocessing, the statement

```
i = ADD(j,k);
```

becomes

```
i = (j+k);
```

whereas the statement

```
i = ADD(,k);
```

becomes

```
i = (+k);
```


Empty Macro Arguments (C99)

- When an empty argument is an operand of the # or ## operators, special rules apply.
- If an empty argument is "stringized" by the # operator, the result is "" (the empty string):

```
#define MK_STR(x) #x
…
char empty_string[] = MK_STR();
```

- The declaration after preprocessing:

```
char empty_string[] = "";
```


Empty Macro Arguments (C99)

- If one of the arguments of the ## operator is empty, it's replaced by an invisible "placemarker" token.
- Concatenating an ordinary token with a placemarker token yields the original token (the placemarker disappears).
- If two placemarker tokens are concatenated, the result is a single placemarker.
- Once macro expansion has been completed, placemarker tokens disappear from the program.

Empty Macro Arguments (C99)

- Example:

```
#define JOIN(x,y,z) x##y##z
…
int JOIN(a,b,c), JOIN(a,b,), JOIN(a,,c), JOIN(,,c);
```

- The declaration after preprocessing:

```
int abc, ab, ac, c;
```

- The missing arguments were replaced by placemarker tokens, which then disappeared when concatenated with any nonempty arguments.
- All three arguments to the `JOIN` macro could even be missing, which would yield an empty result.

Macros with a Variable Number of Arguments (C99)

- C99 allows macros that take an unlimited number of arguments.
- A macro of this kind can pass its arguments to a function that accepts a variable number of arguments.
- Example:

```
#define TEST(condition, ...) ((condition)? \
  printf("Passed test: %s\n", #condition): \
  printf(__VA_ARGS__))
```

- The `...` token (***ellipsis***) goes at the end of the parameter list, preceded by ordinary parameters, if any.
- `__VA_ARGS__` is a special identifier that represents all the arguments that correspond to the ellipsis.

Macros with a Variable Number of Arguments (C99)

- An example that uses the `TEST` macro:

```
TEST(voltage <= max_voltage,
     "Voltage %d exceeds %d\n", voltage, max_voltage);
```

- Preprocessor output (reformatted for readability):

```
((voltage <= max_voltage)?
  printf("Passed test: %s\n", "voltage <= max_voltage"):
  printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

- The program will display the message

```
Passed test: voltage <= max_voltage
```

if `voltage` is no more than `max_voltage`.

- Otherwise, it will display the values of `voltage` and `max_voltage`:

```
Voltage 125 exceeds 120
```


The `__func__` Identifier (C99)

- The `__func__` identifier behaves like a string variable that stores the name of the currently executing function.
- The effect is the same as if each function contains the following declaration at the beginning of its body:

```
static const char __func__[] = "function-name";
```

where *function-name* is the name of the function.

The `__func__` Identifier (C99)

- Debugging macros that rely on the `__func__` identifier:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```

- These macros can used to trace function calls:

```
void f(void)
{
  FUNCTION_CALLED();   /* displays "f called" */
  …
  FUNCTION_RETURNS();  /* displays "f returns" */
}
```

- Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation.***
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The `#if` and `#endif` Directives

- Suppose we're in the process of debugging a program.
- We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program.
- Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later.
- Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

The `#if` and `#endif` Directives

- During preprocessing, the `#if` directive will test the value of `DEBUG`.
- Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program.
- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program.
- The `#if`-`#endif` blocks can be left in the final program, allowing diagnostic information to be produced later if any problems turn up.

The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:

  `#if` *constant-expression*
  `#endif`
- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise, the lines between `#if` and `#endif` will remain.

The `#if` and `#endif` Directives

- The `#if` directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define `DEBUG`, the test

  ```
  #if DEBUG
  ```

  will fail (but not generate an error message).
- The test

  ```
  #if !DEBUG
  ```

  will succeed.

The `defined` Operator

- The preprocessor supports three operators: `#`, `##`, and `defined`.
- When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise.
- The `defined` operator is normally used in conjunction with the `#if` directive.

The `defined` Operator

- Example:

```
#if defined(DEBUG)
…
#endif
```

- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro.
- The parentheses around `DEBUG` aren't required:

```
#if defined DEBUG
```

- It's not necessary to give `DEBUG` a value:

```
#define DEBUG
```


The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:

```
#ifdef identifier
```

- The effect is the same as

```
#if defined(identifier)
```

- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:

```
#ifndef identifier
```

- The effect is the same as

```
#if !defined(identifier)
```


The `#elif` and `#else` Directives

- `#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements.
- When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows.
- Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
…
#endif /* DEBUG */
```


The `#elif` and `#else` Directives

- `#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

```
#if expr1
Lines to be included if expr1 is nonzero
#elif expr2
Lines to be included if expr1 is zero but expr2 is nonzero
#else
Lines to be included otherwise
#endif
```

- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

  Example:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#endif
```


Uses of Conditional Compilation

- ***Writing programs that can be compiled with different compilers.***

  An example that uses the `__STDC__` macro:

```
#if __STDC__
Function prototypes
#else
Old-style function declarations
#endif
```

  If the compiler does not conform to the C standard, old-style function declarations are used instead of function prototypes.

Uses of Conditional Compilation

- ***Providing a default definition for a macro.***

  Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```


Uses of Conditional Compilation

- ***Temporarily disabling code that contains comments.***

  A `/*…*/` comment can't be used to "comment out" code that already contains `/*…*/` comments.

  An `#if` directive can be used instead:

```
#if 0
Lines containing comments
#endif
```

Uses of Conditional Compilation

- Chapter 15 discusses another common use of conditional compilation: protecting header files against multiple inclusion.

Miscellaneous Directives

- The `#error`, `#line`, and `#pragma` directives are more specialized than the ones we've already examined.
- These directives are used much less frequently.

The `#error` Directive

- Form of the `#error` directive:

  `#error` *message*

  message is any sequence of tokens.
- If the preprocessor encounters an `#error` directive, it prints an error message which must include *message*.
- If an `#error` directive is processed, some compilers immediately terminate compilation without attempting to find other errors.

The `#error` Directive

- `#error` directives are frequently used in conjunction with conditional compilation.
- Example that uses an `#error` directive to test the maximum value of the `int` type:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

The `#error` Directive

- The `#error` directive is often found in the `#else` part of an `#if`-`#elif`-`#else` series:

```
#if defined(WIN32)
...
#elif defined(MAC_OS)
...
#elif defined(LINUX)
...
#else
#error No operating system specified
#endif
```


The `#line` Directive

- The `#line` directive is used to alter the way program lines are numbered.
- First form of the `#line` directive:

  `#line` *n*

  Subsequent lines in the program will be numbered *n*, *n* + 1, *n* + 2, and so forth.
- Second form of the `#line` directive:

  `#line` *n* "*file*"

  Subsequent lines are assumed to come from *file*, with line numbers starting at *n*.

The `#line` Directive

- The `#line` directive changes the value of the `__LINE__` macro (and possibly `__FILE__`).
- Most compilers will use the information from the `#line` directive when generating error messages.
- Suppose that the following directive appears at the beginning of `foo.c`:

  ```
  #line 10 "bar.c"
  ```

  If the compiler detects an error on line 5 of `foo.c`, the message will refer to line 13 of file `bar.c`.
- The `#line` directive is used primarily by programs that generate C code as output.

The `#line` Directive

- The most famous example is `yacc` (Yet Another Compiler-Compiler), a UNIX utility that automatically generates part of a compiler.
- The programmer prepares a file that contains information for `yacc` as well as fragments of C code.
- From this file, `yacc` generates a C program, `y.tab.c`, that incorporates the code supplied by the programmer.
- By inserting `#line` directives, `yacc` tricks the compiler into believing that the code comes from the original file.
- Error messages produced during the compilation of `y.tab.c` will refer to lines in the original file.

The #pragma Directive

- The #pragma directive provides a way to request special behavior from the compiler.
- Form of a #pragma directive:

  #pragma *tokens*
- #pragma directives can be very simple (a single token) or they can be much more elaborate:

```
#pragma data(heap_size => 1000, stack_size => 2000)
```


The #pragma Directive

- The set of commands that can appear in #pragma directives is different for each compiler.
- The preprocessor must ignore any #pragma directive that contains an unrecognized command; it's not permitted to give an error message.
- In C89, there are no standard pragmas—they're all implementation-defined.
- C99 has three standard pragmas, all of which use STDC as the first token following #pragma.

The _Pragma Operator (C99)

- C99 introduces the _Pragma operator, which is used in conjunction with the #pragma directive.
- A _Pragma expression has the form

  _Pragma (*string-literal*)
- When it encounters such an expression, the preprocessor "destringizes" the string literal:
  - Double quotes around the string are removed.
  - \" is replaced by ".
  - \\ is replaced by \.

The _Pragma Operator (C99)

- The resulting tokens are then treated as though they appear in a #pragma directive.
- For example, writing

```
_Pragma("data(heap_size => 1000, stack_size => 2000)")
```

  is the same as writing

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

The `_Pragma` Operator (C99)

- The `_Pragma` operator lets us work around the fact that a preprocessing directive can't generate another directive.
- `_Pragma`, however, is an operator, not a directive, and can therefore appear in a macro definition.
- This makes it possible for a macro expansion to leave behind a `#pragma` directive.

The `_Pragma` Operator (C99)

- A macro that uses the `_Pragma` operator:

```
#define DO_PRAGMA(x) _Pragma(#x)
```

- An invocation of the macro:

```
DO_PRAGMA(GCC dependency "parse.y")
```

- The result after expansion:

```
#pragma GCC dependency "parse.y"
```

- The tokens passed to `DO_PRAGMA` are stringized into `"GCC dependency \"parse.y\""`.
- The `_Pragma` operator destringizes this string, producing a `#pragma` directive.

Chapter 15

Writing Large Programs

Source Files

- A C program may be divided among any number of ***source files.***
- By convention, source files have the extension `.c`.
- Each source file contains part of the program, primarily definitions of functions and variables.
- One source file must contain a function named `main`, which serves as the starting point for the program.

Source Files

- Consider the problem of writing a simple calculator program.
- The program will evaluate integer expressions entered in Reverse Polish notation (RPN), in which operators follow operands.
- If the user enters an expression such as

  ```
  30 5 - 7 *
  ```

  the program should print its value (175, in this case).

Source Files

- The program will read operands and operators, one by one, using a stack to keep track of intermediate results.
  - If the program reads a number, it will push the number onto the stack.
  - If the program reads an operator, it will pop two numbers from the stack, perform the operation, and then push the result back onto the stack.
- When the program reaches the end of the user's input, the value of the expression will be on the stack.

Source Files

- How the expression `30 5 - 7 *`: will be evaluated:
  1. Push 30 onto the stack.
  2. Push 5 onto the stack.
  3. Pop the top two numbers from the stack, subtract 5 from 30, giving 25, and then push the result back onto the stack.
  4. Push 7 onto the stack.
  5. Pop the top two numbers from the stack, multiply them, and then push the result back onto the stack.
- The stack will now contain 175, the value of the expression.

Source Files

- The program's `main` function will contain a loop that performs the following actions:
  - Read a "token" (a number or an operator).
  - If the token is a number, push it onto the stack.
  - If the token is an operator, pop its operands from the stack, perform the operation, and then push the result back onto the stack.
- When dividing a program like this one into files, it makes sense to put related functions and variables into the same file.

Source Files

- The function that reads tokens could go into one source file (`token.c`, say), together with any functions that have to do with tokens.
- Stack-related functions such as `push`, `pop`, `make_empty`, `is_empty`, and `is_full` could go into a different file, `stack.c`.
- The variables that represent the stack would also go into `stack.c`.
- The `main` function would go into yet another file, `calc.c`.

Source Files

- Splitting a program into multiple source files has significant advantages:
  - Grouping related functions and variables into a single file helps clarify the structure of the program.
  - Each source file can be compiled separately, which saves time.
  - Functions are more easily reused in other programs when grouped in separate source files.

Header Files

- Problems that arise when a program is divided into several source files:
  - How can a function in one file call a function that's defined in another file?
  - How can a function access an external variable in another file?
  - How can two files share the same macro definition or type definition?
- The answer lies with the `#include` directive, which makes it possible to share information among any number of source files.

Header Files

- The `#include` directive tells the preprocessor to insert the contents of a specified file.
- Information to be shared among several source files can be put into such a file.
- `#include` can then be used to bring the file's contents into each of the source files.
- Files that are included in this fashion are called ***header files*** (or sometimes ***include files***).
- By convention, header files have the extension `.h`.

The `#include` Directive

- The `#include` directive has two primary forms.
- The first is used for header files that belong to C's own library:

  `#include <`*filename*`>`
- The second is used for all other header files:

  `#include "`*filename*`"`
- The difference between the two has to do with how the compiler locates the header file.

The `#include` Directive

- Typical rules for locating header files:
  - `#include <`*filename*`>`: Search the directory (or directories) in which system header files reside.
  - `#include "`*filename*`"`: Search the current directory, then search the directory (or directories) in which system header files reside.
- The places to be searched for header files can usually be altered, often by a command-line option such as `-I`*path*.

The `#include` Directive

- Don't use brackets when including header files that you have written:

```
#include <myheader.h>   /*** WRONG ***/
```

- The preprocessor will probably look for `myheader.h` where the system header files are kept.

The `#include` Directive

- The file name in an `#include` directive may include information that helps locate the file, such as a directory path or drive specifier:

```
#include "c:\cprogs\utils.h"
  /* Windows path */

#include "/cprogs/utils.h"
  /* UNIX path */
```

- Although the quotation marks in the `#include` directive make file names look like string literals, the preprocessor doesn't treat them that way.

The `#include` Directive

- It's usually best not to include path or drive information in `#include` directives.
- Bad examples of Windows `#include` directives:

```
#include "d:utils.h"
#include "\cprogs\include\utils.h"
#include "d:\cprogs\include\utils.h"
```

- Better versions:

```
#include "utils.h"
#include "..\include\utils.h"
```


The `#include` Directive

- The `#include` directive has a third form:

```
#include tokens
```

tokens is any sequence of preprocessing tokens.

- The preprocessor will scan the tokens and replace any macros that it finds.
- After macro replacement, the resulting directive must match one of the other forms of `#include`.
- The advantage of the third kind of `#include` is that the file name can be defined by a macro rather than being "hard-coded" into the directive itself.

The `#include` Directive

- Example:

```
#if defined(IA32)
  #define CPU_FILE "ia32.h"
#elif defined(IA64)
  #define CPU_FILE "ia64.h"
#elif defined(AMD64)
  #define CPU_FILE "amd64.h"
#endif

#include CPU_FILE
```


Sharing Macro Definitions and Type Definitions

- Most large programs contain macro definitions and type definitions that need to be shared by several source files.
- These definitions should go into header files.

Sharing Macro Definitions and Type Definitions

- Suppose that a program uses macros named `BOOL`, `TRUE`, and `FALSE`.
- Their definitions can be put in a header file with a name like `boolean.h`:

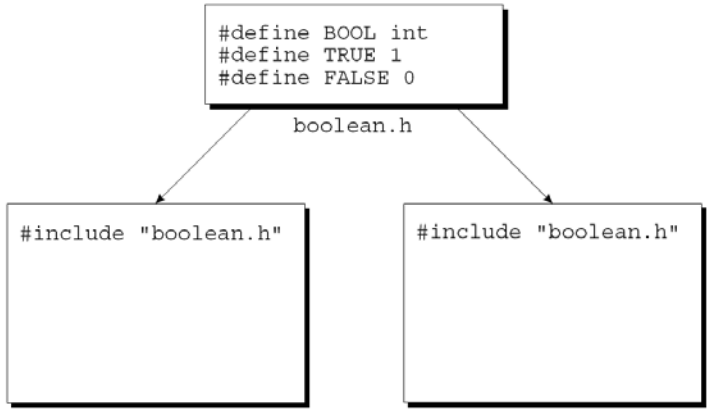
```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

- Any source file that requires these macros will simply contain the line

```
#include "boolean.h"
```


Sharing Macro Definitions and Type Definitions

- A program in which two files include `boolean.h`:



Sharing Macro Definitions and Type Definitions

- Type definitions are also common in header files.
- For example, instead of defining a `BOOL` macro, we might use `typedef` to create a `Bool` type.
- If we do, the `boolean.h` file will have the following appearance:

```
#define TRUE 1
#define FALSE 0
typedef int Bool;
```


Sharing Macro Definitions and Type Definitions

- Advantages of putting definitions of macros and types in header files:
  - Saves time. We don't have to copy the definitions into the source files where they're needed.
  - Makes the program easier to modify. Changing the definition of a macro or type requires editing a single header file.
  - Avoids inconsistencies caused by source files containing different definitions of the same macro or type.

Sharing Function Prototypes

- Suppose that a source file contains a call of a function `f` that's defined in another file, `foo.c`.
- Calling `f` without declaring it first is risky.
  - The compiler assumes that `f`'s return type is `int`.
  - It also assumes that the number of parameters matches the number of arguments in the call of `f`.
  - The arguments themselves are converted automatically by the default argument promotions.

Sharing Function Prototypes

- Declaring `f` in the file where it's called solves the problem but can create a maintenance nightmare.
- A better solution is to put `f`'s prototype in a header file (`foo.h`), then include the header file in all the places where `f` is called.
- We'll also need to include `foo.h` in `foo.c`, enabling the compiler to check that `f`'s prototype in `foo.h` matches its definition in `foo.c`.

Sharing Function Prototypes

- If `foo.c` contains other functions, most of them should be declared in `foo.h`.
- Functions that are intended for use only within `foo.c` shouldn't be declared in a header file, however; to do so would be misleading.

Sharing Function Prototypes

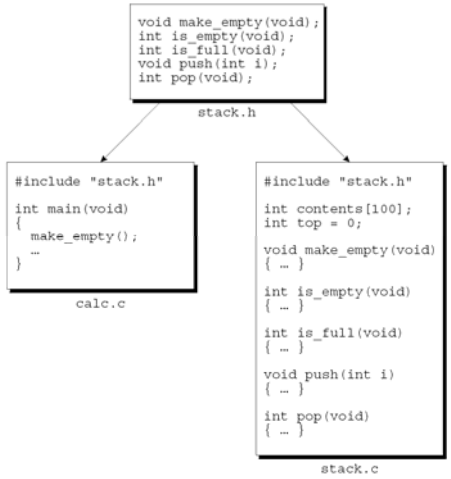
- The RPN calculator example can be used to illustrate the use of function prototypes in header files.
- The `stack.c` file will contain definitions of the `make_empty`, `is_empty`, `is_full`, `push`, and `pop` functions.
- Prototypes for these functions should go in the `stack.h` header file:

```
void make_empty(void);
int is_empty(void);
int is_full(void);
void push(int i);
int pop(void);
```


Sharing Function Prototypes

- We'll include `stack.h` in `calc.c` to allow the compiler to check any calls of stack functions that appear in the latter file.
- We'll also include `stack.h` in `stack.c` so the compiler can verify that the prototypes in `stack.h` match the definitions in `stack.c`.

Sharing Function Prototypes



Sharing Variable Declarations

- To share a function among files, we put its *definition* in one source file, then put *declarations* in other files that need to call the function.
- Sharing an external variable is done in much the same way.

Sharing Variable Declarations

- An example that both declares and defines `i` (causing the compiler to set aside space):

```
int i;
```

- The keyword `extern` is used to declare a variable without defining it:

```
extern int i;
```

- `extern` informs the compiler that `i` is defined elsewhere in the program, so there's no need to allocate space for it.

Sharing Variable Declarations

- When we use `extern` in the declaration of an array, we can omit the length of the array:

```
extern int a[];
```

- Since the compiler doesn't allocate space for `a` at this time, there's no need for it to know `a`'s length.

Sharing Variable Declarations

- To share a variable `i` among several source files, we first put a definition of `i` in one file:

```
int i;
```

- If `i` needs to be initialized, the initializer would go here.
- The other files will contain declarations of `i`:

```
extern int i;
```

- By declaring `i` in each file, it becomes possible to access and/or modify `i` within those files.

Sharing Variable Declarations

- When declarations of the same variable appear in different files, the compiler can't check that the declarations match the variable's definition.
- For example, one file may contain the definition

```
int i;
```

  while another file contains the declaration

```
extern long i;
```

- An error of this kind can cause the program to behave unpredictably.

Sharing Variable Declarations

- To avoid inconsistency, declarations of shared variables are usually put in header files.
- A source file that needs access to a particular variable can then include the appropriate header file.
- In addition, each header file that contains a variable declaration is included in the source file that contains the variable's definition, enabling the compiler to check that the two match.

Nested Includes

- A header file may contain `#include` directives.
- `stack.h` contains the following prototypes:

```
int is_empty(void);
int is_full(void);
```

- Since these functions return only 0 or 1, it's a good idea to declare their return type to be `Bool`:

```
Bool is_empty(void);
Bool is_full(void);
```

- We'll need to include the `boolean.h` file in `stack.h` so that the definition of `Bool` is available when `stack.h` is compiled.

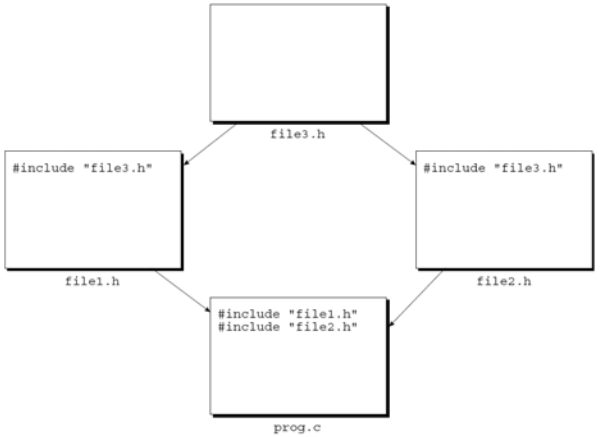
Nested Includes

- Traditionally, C programmers shun nested includes.
- However, the bias against nested includes has largely faded away, in part because nested includes are common practice in C++.

Protecting Header Files

- If a source file includes the same header file twice, compilation errors may result.
- This problem is common when header files include other header files.
- Suppose that `file1.h` includes `file3.h`, `file2.h` includes `file3.h`, and `prog.c` includes both `file1.h` and `file2.h`.

Protecting Header Files



- When `prog.c` is compiled, `file3.h` will be compiled twice.

Protecting Header Files

- Including the same header file twice doesn't always cause a compilation error.
- If the file contains only macro definitions, function prototypes, and/or variable declarations, there won't be any difficulty.
- If the file contains a type definition, however, we'll get a compilation error.

Protecting Header Files

- Just to be safe, it's probably a good idea to protect all header files against multiple inclusion.
- That way, we can add type definitions to a file later without the risk that we might forget to protect the file.
- In addition, we might save some time during program development by avoiding unnecessary recompilation of the same header file.

Protecting Header Files

- To protect a header file, we'll enclose the contents of the file in an `#ifndef`-`#endif` pair.
- How to protect the `boolean.h` file:

```
#ifndef BOOLEAN_H
#define BOOLEAN_H

#define TRUE 1
#define FALSE 0
typedef int Bool;

#endif
```


Protecting Header Files

- Making name of the macro resemble the name of the header file is a good way to avoid conflicts with other macros.
- Since we can't name the macro `BOOLEAN.H`, a name such as `BOOLEAN_H` is a good alternative.

`#error` Directives in Header Files

- `#error` directives are often put in header files to check for conditions under which the header file shouldn't be included.
- Suppose that a header file uses a feature that didn't exist prior to the original C89 standard.
- An `#ifndef` directive that tests for the existence of the `__STDC__` macro:

```
#ifndef __STDC__
#error This header requires a Standard C compiler
#endif
```


Dividing a Program into Files

- Designing a program involves determining what functions it will need and arranging the functions into logically related groups.
- Once a program has been designed, there is a simple technique for dividing it into files.

Dividing a Program into Files

- Each set of functions will go into a separate source file (`foo.c`).
- Each source file will have a matching header file (`foo.h`).
  - `foo.h` will contain prototypes for the functions defined in `foo.c`.
  - Functions to be used only within `foo.c` should not be declared in `foo.h`.
- `foo.h` will be included in each source file that needs to call a function defined in `foo.c`.
- `foo.h` will also be included in `foo.c` so the compiler can check that the prototypes in `foo.h` match the definitions in `foo.c`.

Dividing a Program into Files

- The `main` function will go in a file whose name matches the name of the program.
- It's possible that there are other functions in the same file as `main`, so long as they're not called from other files in the program.

Program: Text Formatting

- Let's apply this technique to a small text-formatting program named `justify`.
- Assume that a file named `quote` contains the following sample input:

```
   C     is quirky,  flawed,    and  an
enormous   success.      Although accidents of
   history
 surely  helped,   it evidently    satisfied   a   need

    for  a   system  implementation    language
   efficient
 enough   to  displace         assembly   language,
   yet sufficiently   abstract   and fluent    to
   describe
  algorithms   and     interactions    in a   wide
   variety
of environments.
```


Program: Text Formatting

- To run the program from a UNIX or Windows prompt, we'd enter the command

  ```
  justify <quote
  ```

- The < symbol informs the operating system that justify will read from the file `quote` instead of accepting input from the keyboard.
- This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection.***

Program: Text Formatting

- Output of `justify`:

```
C is quirky, flawed, and an enormous success. Although
accidents of history surely helped, it evidently satisfied a
need for a system implementation language efficient enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and interactions in a wide
variety of environments. -- Dennis M. Ritchie
```

- The output of `justify` will normally appear on the screen, but we can save it in a file by using ***output redirection:***

```
justify <quote >newquote
```


Program: Text Formatting

- `justify` will delete extra spaces and blank lines as well as filling and justifying lines.
  - "Filling" a line means adding words until one more word would cause the line to overflow.
  - "Justifying" a line means adding extra spaces between words so that each line has exactly the same length (60 characters).
- Justification must be done so that the space between words in a line is equal (or nearly equal).
- The last line of the output won't be justified.

Program: Text Formatting

- We assume that no word is longer than 20 characters, including any adjacent punctuation.
- If the program encounters a longer word, it must ignore all characters after the first 20, replacing them with a single asterisk.
- For example, the word

  `antidisestablishmentarianism`

  would be printed as

  `antidisestablishment*`

Program: Text Formatting

- The program can't write words one by one as they're read.
- Instead, it will have to store them in a "line buffer" until there are enough to fill a line.

Program: Text Formatting

- The heart of the program will be a loop:

```
for (;;) {
  read word;
  if (can't read word) {
    write contents of line buffer without justification;
    terminate program;
  }
  if (word doesn't fit in line buffer) {
    write contents of line buffer with justification;
    clear line buffer;
  }
  add word to line buffer;
}
```


Program: Text Formatting

- The program will be split into three source files:
  - `word.c`: functions related to words
  - `line.c`: functions related to the line buffer
  - `justify.c`: contains the `main` function
- We'll also need two header files:
  - `word.h`: prototypes for the functions in `word.c`
  - `line.h`: prototypes for the functions in `line.c`
- `word.h` will contain the prototype for a function that reads a word.

word.h

```
#ifndef WORD_H
#define WORD_H

/*************************************************************
 * read_word: Reads the next word from the input and         *
 *            stores it in word. Makes word empty if no      *
 *            word could be read because of end-of-file.     *
 *            Truncates the word if its length exceeds       *
 *            len.                                           *
 *************************************************************/
void read_word(char *word, int len);

#endif
```


Program: Text Formatting

- The outline of the main loop reveals the need for functions that perform the following operations:
  - Write contents of line buffer without justification
  - Determine how many characters are left in line buffer
  - Write contents of line buffer with justification
  - Clear line buffer
  - Add word to line buffer
- We'll call these functions `flush_line`, `space_remaining`, `write_line`, `clear_line`, and `add_word`.

line.h

```
#ifndef LINE_H
#define LINE_H

/*************************************************************
 * clear_line: Clears the current line.                      *
 *************************************************************/
void clear_line(void);

/*************************************************************
 * add_word: Adds word to the end of the current line.       *
 *           If this is not the first word on the line,      *
 *           puts one space before word.                     *
 *************************************************************/
void add_word(const char *word);
```



```
/*************************************************************
 * space_remaining: Returns the number of characters left *
 *                  in the current line.                     *
 *************************************************************/
int space_remaining(void);

/*************************************************************
 * write_line: Writes the current line with                  *
 *             justification.                                *
 *************************************************************/
void write_line(void);

/*************************************************************
 * flush_line: Writes the current line without               *
 *             justification. If the line is empty, does     *
 *             nothing.                                      *
 *************************************************************/
void flush_line(void);

#endif
```


Program: Text Formatting

- Before we write the `word.c` and `line.c` files, we can use the functions declared in `word.h` and `line.h` to write `justify.c`, the main program.
- Writing this file is mostly a matter of translating the original loop design into C.

justify.c

```
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;
```

```
  clear_line();
  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }
    if (word_len > MAX_WORD_LEN)
      word[MAX_WORD_LEN] = '*';
    if (word_len + 1 > space_remaining()) {
      write_line();
      clear_line();
    }
    add_word(word);
  }
}
```


Program: Text Formatting

- `main` uses a trick to handle words that exceed 20 characters.
- When it calls `read_word`, `main` tells it to truncate any word that exceeds 21 characters.
- After `read_word` returns, `main` checks whether `word` contains a string that's longer than 20 characters.
- If so, the word must have been at least 21 characters long (before truncation), so `main` replaces its 21st character by an asterisk.

Program: Text Formatting

- The `word.h` header file has a prototype for only one function, `read_word`.
- `read_word` is easier to write if we add a small "helper" function, `read_char`.
- `read_char`'s job is to read a single character and, if it's a new-line character or tab, convert it to a space.
- Having `read_word` call `read_char` instead of `getchar` solves the problem of treating new-line characters and tabs as spaces.

word.c

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

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
    return ' ';
  return ch;
}
```

```
void read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
}
```


Program: Text Formatting

- `line.c` supplies definitions of the functions declared in `line.h`.
- `line.c` will also need variables to keep track of the state of the line buffer:
  - `line`: characters in the current line
  - `line_len`: number of characters in the current line
  - `num_words`: number of words in the current line

`line.c`

```
#include <stdio.h>
#include <string.h>
#include "line.h"
#define MAX_LINE_LEN 60

char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;

void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}
```



```
void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}
```

```
void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
}

void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```


Building a Multiple-File Program

- Building a large program requires the same basic steps as building a small one:
  - Compiling
  - Linking

Building a Multiple-File Program

- Each source file in the program must be compiled separately.
- Header files don't need to be compiled.
- The contents of a header file are automatically compiled whenever a source file that includes it is compiled.
- For each source file, the compiler generates a file containing object code.
- These files—known as ***object files***—have the extension `.o` in UNIX and `.obj` in Windows.

Building a Multiple-File Program

- The linker combines the object files created in the previous step—along with code for library functions—to produce an executable file.
- Among other duties, the linker is responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.

Building a Multiple-File Program

- Most compilers allow us to build a program in a single step.
- A GCC command that builds `justify`:

```
gcc -o justify justify.c line.c word.c
```

- The three source files are first compiled into object code.
- The object files are then automatically passed to the linker, which combines them into a single file.
- The `-o` option specifies that we want the executable file to be named `justify`.

Makefiles

- To make it easier to build large programs, UNIX originated the concept of the ***makefile.***
- A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.
- Suppose that the file `foo.c` includes the file `bar.h`.
- We say that `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

Makefiles

- A UNIX makefile for the `justify` program:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
        gcc -c justify.c

word.o: word.c word.h
        gcc -c word.c

line.o: line.c line.h
        gcc -c line.c
```


Makefiles

- There are four groups of lines; each group is known as a ***rule.***
- The first line in each rule gives a ***target*** file, followed by the files on which it depends.
- The second line is a ***command*** to be executed if the target should need to be rebuilt because of a change to one of its dependent files.

Makefiles

- In the first rule, `justify` (the executable file) is the target:

```
justify: justify.o word.o line.o
        gcc -o justify justify.o word.o line.o
```

- The first line states that `justify` depends on the files `justify.o`, `word.o`, and `line.o`.
- If any of these files have changed since the program was last built, `justify` needs to be rebuilt.
- The command on the following line shows how the rebuilding is to be done.

Makefiles

- In the second rule, `justify.o` is the target:

```
justify.o: justify.c word.h line.h
        gcc -c justify.c
```

- The first line indicates that `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h`.
- The next line shows how to update `justify.o` (by recompiling `justify.c`).
- The `-c` option tells the compiler to compile `justify.c` but not attempt to link it.

Makefiles

- Once we've created a makefile for a program, we can use the `make` utility to build (or rebuild) the program.
- By checking the time and date associated with each file in the program, `make` can determine which files are out of date.
- It then invokes the commands necessary to rebuild the program.

Makefiles

- Each command in a makefile must be preceded by a tab character, not a series of spaces.
- A makefile is normally stored in a file named `Makefile` (or `makefile`).
- When the `make` utility is used, it automatically checks the current directory for a file with one of these names.

Makefiles

- To invoke `make`, use the command

  `make` *target*

  where *target* is one of the targets listed in the makefile.
- If no target is specified when `make` is invoked, it will build the target of the first rule.
- Except for this special property of the first rule, the order of rules in a makefile is arbitrary.

Makefiles

- Real makefiles aren't always easy to understand.
- There are numerous techniques that reduce the amount of redundancy in makefiles and make them easier to modify.
- These techniques greatly reduce the readability of makefiles.
- Alternatives to makefiles include the "project files" supported by some integrated development environments.

Errors During Linking

- Some errors that can't be detected during compilation will be found during linking.
- If the definition of a function or variable is missing from a program, the linker will be unable to resolve external references to it
- The result is a message such as *"undefined symbol"* or *"undefined reference."*

Errors During Linking

- Common causes of errors during linking:
  - ***Misspellings.*** If the name of a variable or function is misspelled, the linker will report it as missing.
  - ***Missing files.*** If the linker can't find the functions that are in file `foo.c`, it may not know about the file.
  - ***Missing libraries.*** The linker may not be able to find all library functions used in the program.
- In UNIX, the `-lm` option may need to be specified when a program that uses `<math.h>` is linked.

Rebuilding a Program

- During the development of a program, it's rare that we'll need to compile all its files.
- To save time, the rebuilding process should recompile only those files that might be affected by the latest change.
- Assume that a program has been designed with a header file for each source file.
- To see how many files will need to be recompiled after a change, we need to consider two possibilities.

Rebuilding a Program

- If the change affects a single source file, only that file must be recompiled.
- Suppose that we decide to condense the `read_char` function in `word.c`:

```
int read_char(void)
{
  int ch = getchar();

  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

- This modification doesn't affect `word.h`, so we need only recompile `word.c` and relink the program.

Rebuilding a Program

- The second possibility is that the change affects a header file.
- In that case, we should recompile all files that include the header file, since they could potentially be affected by the change.

Rebuilding a Program

- Suppose that we modify `read_word` so that it returns the length of the word that it reads.
- First, we change the prototype of `read_word` in `word.h`:

```
/*************************************************************
 * read_word: Reads the next word from the input and         *
 *            stores it in word. Makes word empty if no      *
 *            word could be read because of end-of-file.     *
 *            Truncates the word if its length exceeds       *
 *            len. Returns the number of characters          *
 *            stored.                                        *
 *************************************************************/
int read_word(char *word, int len);
```

Rebuilding a Program

- Next, we change the definition of `read_word`:

```
int read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
    ;
  while (ch != ' ' && ch != EOF) {
    if (pos < len)
      word[pos++] = ch;
    ch = read_char();
  }
  word[pos] = '\0';
  return pos;
}
```


Rebuilding a Program

- Finally, we modify `justify.c` by removing the include of `<string.h>` and changing `main`:

```
int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();
  for (;;) {
    word_len = read_word(word, MAX_WORD_LEN+1);
    ...
  }
}
```


Rebuilding a Program

- Once we've made these changes, we'll rebuild `justify` by recompiling `word.c` and `justify.c` and then relinking.
- A GCC command that rebuilds the program:

```
gcc -o justify justify.c word.c line.o
```


Rebuilding a Program

- One of the advantages of using makefiles is that rebuilding is handled automatically.
- By examining the date of each file, `make` can determine which files have changed since the program was last built.
- It then recompiles these files, together with all files that depend on them, either directly or indirectly.

Rebuilding a Program

- Suppose that we make the indicated changes to `word.h`, `word.c`, and `justify.c`.
- When the `justify` program is rebuilt, `make` will perform the following actions:
  1. Build `justify.o` by compiling `justify.c` (because `justify.c` and `word.h` were changed).
  2. Build `word.o` by compiling `word.c` (because `word.c` and `word.h` were changed).
  3. Build `justify` by linking `justify.o`, `word.o`, and `line.o` (because `justify.o` and `word.o` were changed).

Defining Macros Outside a Program

- C compilers usually provide some method of specifying the value of a macro at the time a program is compiled.
- This ability makes it easy to change the value of a macro without editing any of the program's files.
- It's especially valuable when programs are built automatically using makefiles.

Defining Macros Outside a Program

- Most compilers (including GCC) support the `-D` option, which allows the value of a macro to be specified on the command line:

```
gcc -DDEBUG=1 foo.c
```

- In this example, the `DEBUG` macro is defined to have the value `1` in the program `foo.c`.
- If the `-D` option names a macro without specifying its value, the value is taken to be `1`.

Defining Macros Outside a Program

- Many compilers also support the `-U` option, which "undefines" a macro as if by using `#undef`.
- We can use `-U` to undefine a predefined macro or one that was defined earlier in the command line using `-D`.

Chapter 16

Structures, Unions, and Enumerations

Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
- In some languages, structures are called ***records,*** and members are known as ***fields.***

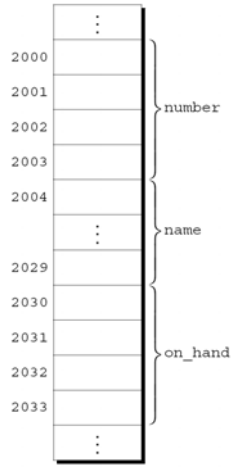
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

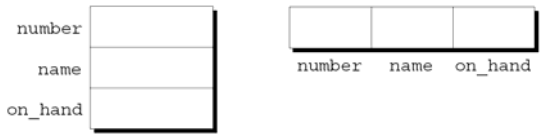

Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1` ⟶
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure:



- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

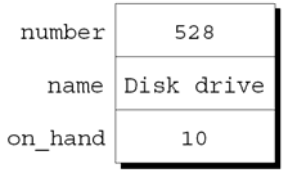
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```


Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- Expressions used in a structure initializer must be constant. (This restriction is relaxed in C99.)
- An initializer can have fewer members than the structure it's initializing.
- Any "leftover" members are given 0 as their initial value.

Designated Initializers (C99)

- C99's designated initializers can be used with structures.
- The initializer for `part1` shown in the previous example:

```
{528, "Disk drive", 10}
```

- In a designated initializer, each value would be labeled by the name of the member that it initializes:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the period and the member name is called a ***designator.***

Designated Initializers (C99)

- Designated initializers are easier to read and check for correctness.
- Also, values in a designated initializer don't have to be placed in the same order that the members are listed in the structure.
  - The programmer doesn't have to remember the order in which the members were originally declared.
  - The order of the members can be changed in the future without affecting designated initializers.

Designated Initializers (C99)

- Not all values listed in a designated initializer need be prefixed by a designator.
- Example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

  The compiler assumes that `"Disk drive"` initializes the member that follows `number` in the structure.
- Any members that the initializer fails to account for are set to zero.

Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```


Operations on Structures

- The members of a structure are lvalues.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```


Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Operations on Structures

- Arrays can't be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.
- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```


Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
- Two structures declared at the same time (as `part1` and `part2` were) are compatible.
- Structures declared using the same "structure tag" or the same type name are also compatible.
- Other than assignment, C provides no operations on entire structures.
- In particular, the == and != operators can't be used with structures.

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- We need a name that represents a *type* of structure, not a particular structure *variable*.
- Ways to name a structure:
  - Declare a "structure tag"
  - Use `typedef` to define a type name

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Declaring a Structure Tag

- The `part` tag can be used to declare variables:

```
struct part part1, part2;
```

- We can't drop the word `struct`:

```
part part1, part2;   /*** WRONG ***/
```

`part` isn't a type name; without the word `struct`, it is meaningless.

- Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program.

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables:*

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```


Declaring a Structure Tag

- All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```


Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Defining a Structure Type

- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`.
- However, declaring a structure tag is mandatory when the structure is to be used in a linked list (Chapter 17).

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```


Structures as Arguments and Return Values

- A function that returns a `part` structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```


Structures as Arguments and Return Values

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
- To avoid this overhead, it's sometimes advisable to pass a pointer to a structure or return a pointer to a structure.
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

Structures as Arguments and Return Values

- There are other reasons to avoid copying structures.
- For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure.
- Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program.
- Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure.
- Every function that performs an operation on an open file requires a `FILE` pointer as an argument.

Structures as Arguments and Return Values

- Within a function, the initializer for a structure variable can be another structure:

```
void f(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The structure being initialized must have automatic storage duration.

Compound Literals (C99)

- Chapter 9 introduced the C99 feature known as the ***compound literal.***
- A compound literal can be used to create a structure "on the fly," without first storing it in a variable.
- The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable.

Compound Literals (C99)

- A compound literal can be used to create a structure that will be passed to a function:

```
print_part((struct part) {528, "Disk drive", 10});
```

  The compound literal is shown in **bold**.
- A compound literal can also be assigned to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```

- A compound literal consists of a type name within parentheses, followed by a set of values in braces.
- When a compound literal represents a structure, the type name can be a structure tag preceded by the word `struct` or a `typedef` name.

Compound Literals (C99)

- A compound literal may contain designators, just like a designated initializer:

```
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

- A compound literal may fail to provide full initialization, in which case any uninitialized members default to zero.

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.

Nested Structures

- Nesting one structure inside another is often useful.
- Suppose that `person_name` is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```


Nested Structures

- We can use `person_name` as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Nested Structures

- Having `name` be a structure makes it easier to treat names as units of data.
- A function that displays a name could be passed one `person_name` argument instead of three arguments:

```
display_name(student1.name);
```

- Copying the information from a `person_name` structure to the `name` member of a student structure would take one assignment instead of three:

```
struct person_name new_name;
…
student1.name = new_name;
```


Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```


Arrays of Structures

- Accessing a part in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```


Initializing an Array of Structures

- Initializing an array of structures is done in much the same way as initializing a multidimensional array.
- Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.

Initializing an Array of Structures

- One reason for initializing an array of structures is that it contains information that won't change during program execution.
- Example: an array that contains country codes used when making international telephone calls.
- The elements of the array will be structures that store the name of a country along with its code:

```
struct dialing_code {
  char *country;
  int code;
};
```


Initializing an Array of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",            54}, {"Bangladesh",       880},
   {"Brazil",               55}, {"Burma (Myanmar)",   95},
   {"China",                86}, {"Colombia",          57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",             20},
   {"Ethiopia",            251}, {"France",            33},
   {"Germany",              49}, {"India",             91},
   {"Indonesia",            62}, {"Iran",              98},
   {"Italy",                39}, {"Japan",             81},
   {"Mexico",               52}, {"Nigeria",          234},
   {"Pakistan",             92}, {"Philippines",       63},
   {"Poland",               48}, {"Russia",             7},
   {"South Africa",         27}, {"South Korea",       82},
   {"Spain",                34}, {"Sudan",            249},
   {"Thailand",             66}, {"Turkey",            90},
   {"Ukraine",             380}, {"United Kingdom",    44},
   {"United States",         1}, {"Vietnam",           84}};
```

- The inner braces around each structure value are optional.

Initializing an Array of Structures

- C99's designated initializers allow an item to have more than one designator.
- A declaration of the `inventory` array that uses a designated initializer to create a single part:

```
struct part inventory[100] =
  {[0].number = 528, [0].on_hand = 10,
   [0].name[0] = '\0'};
```

The first two items in the initializer use two designators; the last item uses three.

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates how nested arrays and structures are used in practice.
- The program tracks parts stored in a warehouse.
- Information about the parts is stored in an array of structures.
- Contents of each structure:
  - Part number
  - Name
  - Quantity

Program: Maintaining a Parts Database

- Operations supported by the program:
  - Add a new part number, part name, and initial quantity on hand
  - Given a part number, print the name of the part and the current quantity on hand
  - Given a part number, change the quantity on hand
  - Print a table showing all information in the database
  - Terminate program execution

Program: Maintaining a Parts Database

- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.
- A session with the program:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```


Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```


Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name              Quantity on Hand
    528       Disk drive                    8
    914       Printer cable                 5

Enter operation code: q
```

Program: Maintaining a Parts Database

- The program will store information about each part in a structure.
- The structures will be stored in an array named `inventory`.
- A variable named `num_parts` will keep track of the number of parts currently stored in the array.

Program: Maintaining a Parts Database

- An outline of the program's main loop:

```
for (;;) {
  prompt user to enter operation code;
  read code;
  switch (code) {
    case 'i': perform insert operation; break;
    case 's': perform search operation; break;
    case 'u': perform update operation; break;
    case 'p': perform print operation; break;
    case 'q': terminate program;
    default:  print error message;
  }
}
```


Program: Maintaining a Parts Database

- Separate functions will perform the insert, search, update, and print operations.
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
- The program is split into three files:
  - `inventory.c` (the bulk of the program)
  - `readline.h` (contains the prototype for the `read_line` function)
  - `readline.c` (contains the definition of `read_line`)

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/*************************************************************
 * main: Prompts the user to enter an operation code,        *
 *       then calls a function to perform the requested      *
 *       action. Repeats until the user enters the           *
 *       command 'q'. Prints an error message if the user    *
 *       enters an illegal code.                             *
 *************************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```



```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```



```
/*************************************************************
 * find_part: Looks up a part number in the inventory        *
 *            array. Returns the array index if the part     *
 *            number is found; otherwise, returns -1.        *
 *************************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```



```
/*************************************************************
 * insert: Prompts the user for information about a new      *
 *         part and then inserts the part into the           *
 *         database. Prints an error message and returns     *
 *         prematurely if the part already exists or the     *
 *         database is full.                                 *
 *************************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

```
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```



```
/*************************************************************
 * search: Prompts the user to enter a part number, then     *
 *         looks up the part in the database. If the part    *
 *         exists, prints the name and quantity on hand;     *
 *         if not, prints an error message.                  *
 *************************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```



```
/*************************************************************
 * update: Prompts the user to enter a part number.          *
 *         Prints an error message if the part doesn't       *
 *         exist; otherwise, prompts the user to enter       *
 *         change in quantity on hand and updates the        *
 *         database.                                         *
 *************************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```



```
/*************************************************************
 * print: Prints a listing of all parts in the database,     *
 *        showing the part number, part name, and            *
 *        quantity on hand. Parts are printed in the         *
 *        order in which they were entered into the          *
 *        database.                                          *
 *************************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

Program: Maintaining a Parts Database

- The version of `read_line` in Chapter 13 won't work properly in the current program.
- Consider what happens when the user inserts a part:

```
Enter part number: 528
Enter part name: Disk drive
```

- The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read.
- When `scanf` reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread.

Program: Maintaining a Parts Database

- If we try to read the part name using the original `read_line` function, it will encounter the new-line character immediately and stop reading.
- This problem is common when numerical input is followed by character input.
- One solution is to write a version of `read_line` that skips white-space characters before it begins storing characters.
- This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**********************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **********************************************************/
int read_line(char str[], int n);

#endif
```


readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

Unions

- A ***union,*** like a structure, consists of one or more members, possibly of different types.
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space.
- Assigning a new value to one member alters the values of the other members as well.

Unions

- An example of a union variable:

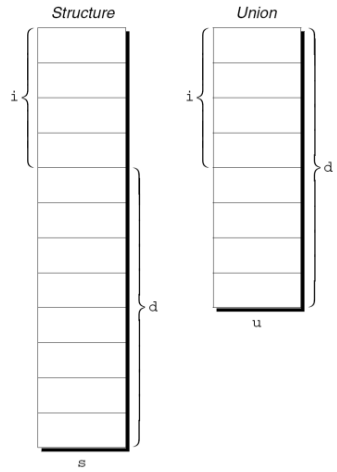
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```


Unions

- The structure `s` and the union `u` differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Unions

- Members of a union are accessed in the same way as members of a structure:

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members.
  - Storing a value in `u.d` causes any value previously stored in `u.i` to be lost.
  - Changing `u.i` corrupts `u.d`.

Unions

- The properties of unions are almost identical to the properties of structures.
- We can declare union tags and union types in the same way we declare structure tags and types.
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.

Unions

- Only the first member of a union can be given an initial value.
- How to initialize the `i` member of `u` to 0:

```
union {
  int i;
  double d;
} u = {0};
```

- The expression inside the braces must be constant. (The rules are slightly different in C99.)

Unions

- Designated initializers can also be used with unions.
- A designated initializer allows us to specify which member of a union should be initialized:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one.

Unions

- Applications for unions:
  - Saving space
  - Building mixed data structures
  - Viewing storage in different ways (discussed in Chapter 20)

Using Unions to Save Space

- Unions can be used to save space in structures.
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.
- Each item has a stock number and a price, as well as other information that depends on the type of the item:

  Books: Title, author, number of pages

  Mugs: Design

  Shirts: Design, colors available, sizes available

Using Unions to Save Space

- A first attempt at designing the `catalog_item` structure:

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```


Using Unions to Save Space

- The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`.
- The `colors` and `sizes` members would store encoded combinations of colors and sizes.
- This structure wastes space, since only part of the information in the structure is common to all items in the catalog.
- By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure.

Using Unions to Save Space

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Using Unions to Save Space

- If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:

```
printf("%s", c.item.book.title);
```

- As this example shows, accessing a union that's nested inside a structure can be awkward.

Using Unions to Save Space

- The `catalog_item` structure can be used to illustrate an interesting aspect of unions.
- Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member.
- However, there is a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members.
- If one of the structures is currently valid, then the matching members in the other structures will also be valid.

Using Unions to Save Space

- The union embedded in the `catalog_item` structure contains three structures as members.
- Two of these (`mug` and `shirt`) begin with a matching member (`design`).
- Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

- The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design);
  /* prints "Cats" */
```


Using Unions to Build Mixed Data Structures

- Unions can be used to create data structures that contain a mixture of data of different types.
- Suppose that we need an array whose elements are a mixture of `int` and `double` values.
- First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union {
  int i;
  double d;
} Number;
```

Using Unions to Build Mixed Data Structures

- Next, we create an array whose elements are `Number` values:

```
Number number_array[1000];
```

- A `Number` union can store either an `int` value or a `double` value.
- This makes it possible to store a mixture of `int` and `double` values in `number_array`:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```


Adding a "Tag Field" to a Union

- There's no easy way to tell which member of a union was last changed and therefore contains a meaningful value.
- Consider the problem of writing a function that displays the value stored in a `Number` union:

```
void print_number(Number n)
{
  if (n contains an integer)
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

There's no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

Adding a "Tag Field" to a Union

- In order to keep track of this information, we can embed the union within a structure that has one other member: a "tag field" or "discriminant."
- The purpose of a tag field is to remind us what's currently stored in the union.
- `item_type` served this purpose in the `catalog_item` structure.

Adding a "Tag Field" to a Union

- The `Number` type as a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;   /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

- The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Adding a "Tag Field" to a Union

- Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified.
- An example that assigns a value to the `i` member of `u`:

```
n.kind = INT_KIND;
n.u.i = 82;
```

`n` is assumed to be a `Number` variable.

Adding a "Tag Field" to a Union

- When the number stored in a `Number` variable is retrieved, `kind` will tell us which member of the union was the last to be assigned a value.
- A function that takes advantage of this capability:

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```


Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values.
- A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."

Enumerations

- A "suit" variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

```
int s;    /* s will store a suit */
…
s = 2;    /* 2 represents "hearts" */
```

- Problems with this technique:
  - We can't tell that `s` has only four possible values.
  - The significance of 2 isn't apparent.

Enumerations

- Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

- An updated version of the previous example:

```
SUIT s;
…
s = HEARTS;
```


Enumerations

- Problems with this technique:
  - There's no indication to someone reading the program that the macros represent values of the same "type."
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious.
  - The names `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` will be removed by the preprocessor, so they won't be available during debugging.

Enumerations

- C provides a special kind of type designed specifically for variables that have a small number of possible values.
- An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer.
- Each value must have a name (an ***enumeration constant***).

Enumerations

- Although enumerations have little in common with structures and unions, they're declared in a similar way:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumerations

- Enumeration constants are similar to constants created with the `#define` directive, but they're not equivalent.
- If an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

- As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.
- Enumeration tags resemble structure and union tags:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- `suit` variables would be declared in the following way:

```
enum suit s1, s2;
```


Enumeration Tags and Type Names

- As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

- In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```


Enumerations as Integers

- Behind the scenes, C treats enumeration variables and constants as integers.
- By default, the compiler assigns the integers 0, 1, 2, … to the constants in a particular enumeration.
- In the `suit` enumeration, `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` represent 0, 1, 2, and 3, respectively.

Enumerations as Integers

- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20,
           PRODUCTION = 10, SALES = 25};
```

- It's even legal for two or more enumeration constants to have the same value.

Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.
- The first enumeration constant has the value 0 by default.
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7,
                 DK_GRAY, WHITE = 15};
```

`BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15.

Enumerations as Integers

- Enumeration values can be mixed with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1            */
s = 0;           /* s is now 0 (CLUBS)    */
s++;             /* s is now 1 (DIAMONDS) */
i = s + 2;       /* i is now 3            */
```

- `s` is treated as a variable of some integer type.
- `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` are names for the integers 0, 1, 2, and 3.

Enumerations as Integers

- Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value.
- For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.

Using Enumerations to Declare "Tag Fields"

- Enumerations are perfect for determining which member of a union was the last to be assigned a value.
- In the `Number` structure, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```


Using Enumerations to Declare "Tag Fields"

- The new structure is used in exactly the same way as the old one.
- Advantages of the new structure:
  - Does away with the `INT_KIND` and `DOUBLE_KIND` macros
  - Makes it obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`

Chapter 17

Advanced Uses of Pointers

Dynamic Storage Allocation

- C's data structures, including arrays, are normally fixed in size.
- Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program.
- Fortunately, C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution.
- Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Dynamic Storage Allocation

- Dynamic storage allocation is used most often for strings, arrays, and structures.
- Dynamically allocated structures can be linked together to form lists, trees, and other data structures.
- Dynamic storage allocation is done by calling a memory allocation function.

Memory Allocation Functions

- The `<stdlib.h>` header declares three memory allocation functions:

  `malloc`—Allocates a block of memory but doesn't initialize it.

  `calloc`—Allocates a block of memory and clears it.

  `realloc`—Resizes a previously allocated block of memory.
- These functions return a value of type `void *` (a "generic" pointer).

Null Pointers

- If a memory allocation function can't locate a memory block of the requested size, it returns a ***null pointer.***
- A null pointer is a special value that can be distinguished from all valid pointers.
- After we've stored the function's return value in a pointer variable, we must test to see if it's a null pointer.

Null Pointers

- An example of testing `malloc`'s return value:

```
p = malloc(10000);
if (p == NULL) {
  /* allocation failed; take appropriate action */
}
```

- `NULL` is a macro (defined in various library headers) that represents the null pointer.
- Some programmers combine the call of `malloc` with the `NULL` test:

```
if ((p = malloc(10000)) == NULL) {
  /* allocation failed; take appropriate action */
}
```


Null Pointers

- Pointers test true or false in the same way as numbers.
- All non-null pointers test true; only null pointers are false.
- Instead of writing

```
if (p == NULL) …
```

we could write

```
if (!p) …
```

- Instead of writing

```
if (p != NULL) …
```

we could write

```
if (p) …
```


Dynamically Allocated Strings

- Dynamic storage allocation is often useful for working with strings.
- Strings are stored in character arrays, and it can be hard to anticipate how long these arrays need to be.
- By allocating strings dynamically, we can postpone the decision until the program is running.

Using `malloc` to Allocate Memory for a String

- Prototype for the `malloc` function:

  ```
  void *malloc(size_t size);
  ```

- `malloc` allocates a block of `size` bytes and returns a pointer to it.
- `size_t` is an unsigned integer type defined in the library.

Using `malloc` to Allocate Memory for a String

- A call of `malloc` that allocates memory for a string of `n` characters:

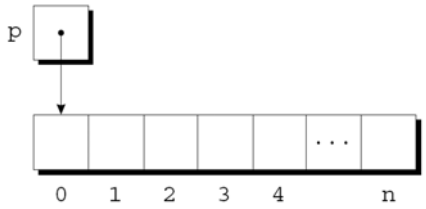
  ```
  p = malloc(n + 1);
  ```

  `p` is a `char *` variable.
- Each character requires one byte of memory; adding 1 to `n` leaves room for the null character.
- Some programmers prefer to cast `malloc`'s return value, although the cast is not required:

  ```
  p = (char *) malloc(n + 1);
  ```


Using `malloc` to Allocate Memory for a String

- Memory allocated using `malloc` isn't cleared, so `p` will point to an uninitialized array of `n` + 1 characters:

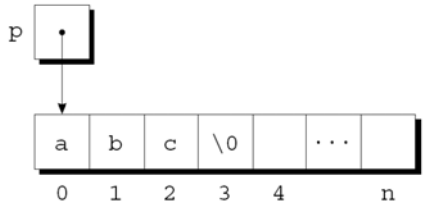


Using `malloc` to Allocate Memory for a String

- Calling `strcpy` is one way to initialize this array:

  ```
  strcpy(p, "abc");
  ```

- The first four characters in the array will now be `a`, `b`, `c`, and `\0`:



Using Dynamic Storage Allocation in String Functions

- Dynamic storage allocation makes it possible to write functions that return a pointer to a "new" string.
- Consider the problem of writing a function that concatenates two strings without changing either one.
- The function will measure the lengths of the two strings to be concatenated, then call `malloc` to allocate the right amount of space for the result.

Using Dynamic Storage Allocation in String Functions

```
char *concat(const char *s1, const char *s2)
{
  char *result;

  result = malloc(strlen(s1) + strlen(s2) + 1);
  if (result == NULL) {
    printf("Error: malloc failed in concat\n");
    exit(EXIT_FAILURE);
  }
  strcpy(result, s1);
  strcat(result, s2);
  return result;
}
```


Using Dynamic Storage Allocation in String Functions

- A call of the `concat` function:

```
p = concat("abc", "def");
```

- After the call, `p` will point to the string `"abcdef"`, which is stored in a dynamically allocated array.

Using Dynamic Storage Allocation in String Functions

- Functions such as `concat` that dynamically allocate storage must be used with care.
- When the string that `concat` returns is no longer needed, we'll want to call the `free` function to release the space that the string occupies.
- If we don't, the program may eventually run out of memory.

Program: Printing a One-Month Reminder List (Revisited)

- The `remind2.c` program is based on the `remind.c` program of Chapter 13, which prints a one-month list of daily reminders.
- The original `remind.c` program stores reminder strings in a two-dimensional array of characters.
- In the new program, the array will be one-dimensional; its elements will be pointers to dynamically allocated strings.

Program: Printing a One-Month Reminder List (Revisited)

- Advantages of switching to dynamically allocated strings:
  - Uses space more efficiently by allocating the exact number of characters needed to store a reminder.
  - Avoids calling `strcpy` to move existing reminder strings in order to make room for a new reminder.
- Switching from a two-dimensional array to an array of pointers requires changing only eight lines of the program (shown in **bold**).

remind2.c

```
/* Prints a one-month reminder list (dynamic string version) */

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

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);
int main(void)
{
  char *reminders[MAX_REMIND];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;
```



```
  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }

    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      reminders[j] = reminders[j-1];
```

```
    reminders[i] = malloc(2 + strlen(msg_str) + 1);
    if (reminders[i] == NULL) {
      printf("-- No space left --\n");
      break;
    }

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```



```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```


Dynamically Allocated Arrays

- Dynamically allocated arrays have the same advantages as dynamically allocated strings.
- The close relationship between arrays and pointers makes a dynamically allocated array as easy to use as an ordinary array.
- Although `malloc` can allocate space for an array, the `calloc` function is sometimes used instead, since it initializes the memory that it allocates.
- The `realloc` function allows us to make an array "grow" or "shrink" as needed.

Using `malloc` to Allocate Storage for an Array

- Suppose a program needs an array of `n` integers, where `n` is computed during program execution.
- We'll first declare a pointer variable:

  ```
  int *a;
  ```

- Once the value of `n` is known, the program can call `malloc` to allocate space for the array:

  ```
  a = malloc(n * sizeof(int));
  ```

- Always use the `sizeof` operator to calculate the amount of space required for each element.

Using `malloc` to Allocate Storage for an Array

- We can now ignore the fact that `a` is a pointer and use it instead as an array name, thanks to the relationship between arrays and pointers in C.
- For example, we could use the following loop to initialize the array that `a` points to:

```
for (i = 0; i < n; i++)
  a[i] = 0;
```

- We also have the option of using pointer arithmetic instead of subscripting to access the elements of the array.

The `calloc` Function

- The `calloc` function is an alternative to `malloc`.
- Prototype for `calloc`:

```
void *calloc(size_t nmemb, size_t size);
```

- Properties of `calloc`:
  - Allocates space for an array with `nmemb` elements, each of which is `size` bytes long.
  - Returns a null pointer if the requested space isn't available.
  - Initializes allocated memory by setting all bits to 0.

The `calloc` Function

- A call of `calloc` that allocates space for an array of `n` integers:

```
a = calloc(n, sizeof(int));
```

- By calling `calloc` with 1 as its first argument, we can allocate space for a data item of any type:

```
struct point { int x, y; } *p;

p = calloc(1, sizeof(struct point));
```


The `realloc` Function

- The `realloc` function can resize a dynamically allocated array.
- Prototype for `realloc`:

```
void *realloc(void *ptr, size_t size);
```

- `ptr` must point to a memory block obtained by a previous call of `malloc`, `calloc`, or `realloc`.
- `size` represents the new size of the block, which may be larger or smaller than the original size.

The `realloc` Function

- Properties of `realloc`:
  - When it expands a memory block, `realloc` doesn't initialize the bytes that are added to the block.
  - If `realloc` can't enlarge the memory block as requested, it returns a null pointer; the data in the old memory block is unchanged.
  - If `realloc` is called with a null pointer as its first argument, it behaves like `malloc`.
  - If `realloc` is called with 0 as its second argument, it frees the memory block.

The `realloc` Function

- We expect `realloc` to be reasonably efficient:
  - When asked to reduce the size of a memory block, `realloc` should shrink the block "in place."
  - `realloc` should always attempt to expand a memory block without moving it.
- If it can't enlarge a block, `realloc` will allocate a new block elsewhere, then copy the contents of the old block into the new one.
- Once `realloc` has returned, be sure to update all pointers to the memory block in case it has been moved.

Deallocating Storage

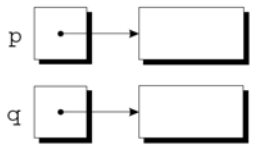
- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as the ***heap.***
- Calling these functions too often—or asking them for large blocks of memory—can exhaust the heap, causing the functions to return a null pointer.
- To make matters worse, a program may allocate blocks of memory and then lose track of them, thereby wasting space.

Deallocating Storage

- Example:

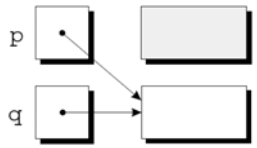
```
p = malloc(…);
q = malloc(…);
p = q;
```

- A snapshot after the first two statements have been executed:



Deallocating Storage

- After `q` is assigned to `p`, both variables now point to the second memory block:



- There are no pointers to the first block, so we'll never be able to use it again.

Deallocating Storage

- A block of memory that's no longer accessible to a program is said to be ***garbage.***
- A program that leaves garbage behind has a ***memory leak.***
- Some languages provide a ***garbage collector*** that automatically locates and recycles garbage, but C doesn't.
- Instead, each C program is responsible for recycling its own garbage by calling the `free` function to release unneeded memory.

The `free` Function

- Prototype for `free`:

```
void free(void *ptr);
```

- `free` will be passed a pointer to an unneeded memory block:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

- Calling `free` releases the block of memory that `p` points to.

The "Dangling Pointer" Problem

- Using `free` leads to a new problem: ***dangling pointers.***
- `free(p)` deallocates the memory block that `p` points to, but doesn't change `p` itself.
- If we forget that `p` no longer points to a valid memory block, chaos may ensue:

```
char *p = malloc(4);
…
free(p);
…
strcpy(p, "abc");   /*** WRONG ***/
```

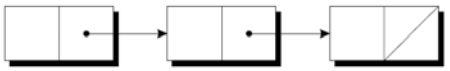
- Modifying the memory that `p` points to is a serious error.

The "Dangling Pointer" Problem

- Dangling pointers can be hard to spot, since several pointers may point to the same block of memory.
- When the block is freed, all the pointers are left dangling.

Linked Lists

- Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures.
- A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain:



- The last node in the list contains a null pointer.

Linked Lists

- A linked list is more flexible than an array: we can easily insert and delete nodes in a linked list, allowing the list to grow and shrink as needed.
- On the other hand, we lose the "random access" capability of an array:
  - Any element of an array can be accessed in the same amount of time.
  - Accessing a node in a linked list is fast if the node is close to the beginning of the list, slow if it's near the end.

Declaring a Node Type

- To set up a linked list, we'll need a structure that represents a single node.
- A node structure will contain data (an integer in this example) plus a pointer to the next node in the list:

```
struct node {
  int value;          /* data stored in the node  */
  struct node *next;  /* pointer to the next node */
};
```

- `node` must be a tag, not a `typedef` name, or there would be no way to declare the type of `next`.

Declaring a Node Type

- Next, we'll need a variable that always points to the first node in the list:

```
struct node *first = NULL;
```

- Setting `first` to `NULL` indicates that the list is initially empty.

Creating a Node

- As we construct a linked list, we'll create nodes one by one, adding each to the list.
- Steps involved in creating a node:
  1. Allocate memory for the node.
  2. Store data in the node.
  3. Insert the node into the list.
- We'll concentrate on the first two steps for now.

Creating a Node

- When we create a node, we'll need a variable that can point to the node temporarily:

```
struct node *new_node;
```

- We'll use `malloc` to allocate memory for the new node, saving the return value in `new_node`:

```
new_node = malloc(sizeof(struct node));
```

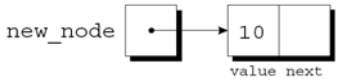
- `new_node` now points to a block of memory just large enough to hold a `node` structure:

Creating a Node

- Next, we'll store data in the `value` member of the new node:

```
(*new_node).value = 10;
```

- The resulting picture:



- The parentheses around `*new_node` are mandatory because the `.` operator would otherwise take precedence over the `*` operator.

The -> Operator

- Accessing a member of a structure using a pointer is so common that C provides a special operator for this purpose.
- This operator, known as ***right arrow selection,*** is a minus sign followed by >.
- Using the -> operator, we can write

```
new_node->value = 10;
```

instead of

```
(*new_node).value = 10;
```


The -> Operator

- The -> operator produces an lvalue, so we can use it wherever an ordinary variable would be allowed.
- A `scanf` example:

```
scanf("%d", &new_node->value);
```

- The `&` operator is still required, even though `new_node` is a pointer.

Inserting a Node at the Beginning of a Linked List

- One of the advantages of a linked list is that nodes can be added at any point in the list.
- However, the beginning of a list is the easiest place to insert a node.
- Suppose that `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list.

Inserting a Node at the Beginning of a Linked List

- It takes two statements to insert the node into the list.
- The first step is to modify the new node's `next` member to point to the node that was previously at the beginning of the list:

```
new_node->next = first;
```

- The second step is to make `first` point to the new node:

```
first = new_node;
```

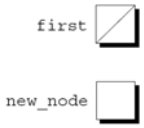
- These statements work even if the list is empty.

Inserting a Node at the Beginning of a Linked List

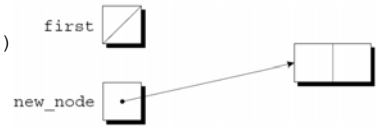
- Let's trace the process of inserting two nodes into an empty list.
- We'll insert a node containing the number 10 first, followed by a node containing 20.

Inserting a Node at the Beginning of a Linked List

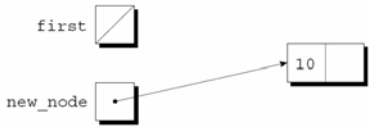
```
first = NULL;
```




```
new_node =
  malloc(sizeof(struct node));
```

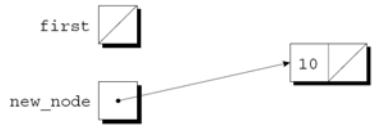



```
new_node->value = 10;
```

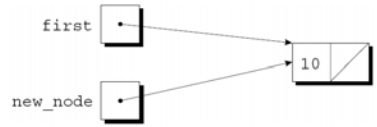


Inserting a Node at the Beginning of a Linked List

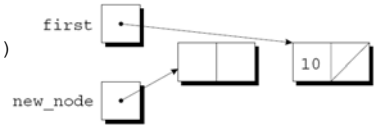
```
new_node->next = first;
```




```
first = new_node;
```

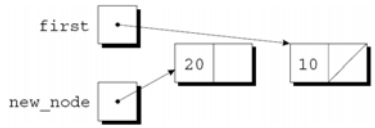



```
new_node =
  malloc(sizeof(struct node))
```

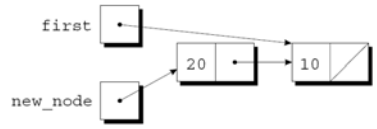


Inserting a Node at the Beginning of a Linked List

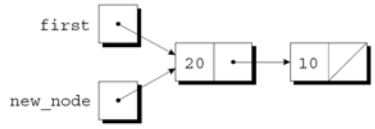
```
new_node->value = 20;
```




```
new_node->next = first;
```




```
first = new_node;
```



Inserting a Node at the Beginning of a Linked List

- A function that inserts a node containing `n` into a linked list, which pointed to by `list`:

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```


Inserting a Node at the Beginning of a Linked List

- Note that `add_to_list` returns a pointer to the newly created node (now at the beginning of the list).
- When we call `add_to_list`, we'll need to store its return value into `first`:

```
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

- Getting `add_to_list` to update `first` directly, rather than return a new value for `first`, turns out to be tricky.

Inserting a Node at the Beginning of a Linked List

- A function that uses `add_to_list` to create a linked list containing numbers entered by the user:

```
struct node *read_numbers(void)
{
  struct node *first = NULL;
  int n;

  printf("Enter a series of integers (0 to terminate): ");
  for (;;) {
    scanf("%d", &n);
    if (n == 0)
      return first;
    first = add_to_list(first, n);
  }
}
```

- The numbers will be in reverse order within the list.

Searching a Linked List

- Although a `while` loop can be used to search a list, the `for` statement is often superior.
- A loop that visits the nodes in a linked list, using a pointer variable `p` to keep track of the "current" node:

```
for (p = first; p != NULL; p = p->next)
  …
```

- A loop of this form can be used in a function that searches a list for an integer `n`.

Searching a Linked List

- If it finds `n`, the function will return a pointer to the node containing `n`; otherwise, it will return a null pointer.
- An initial version of the function:

```
struct node *search_list(struct node *list, int n)
{
  struct node *p;

  for (p = list; p != NULL; p = p->next)
    if (p->value == n)
      return p;
  return NULL;
}
```


Searching a Linked List

- There are many other ways to write `search_list`.
- One alternative is to eliminate the `p` variable, instead using `list` itself to keep track of the current node:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL; list = list->next)
    if (list->value == n)
      return list;
  return NULL;
}
```

- Since `list` is a copy of the original list pointer, there's no harm in changing it within the function.

Searching a Linked List

- Another alternative:

```
struct node *search_list(struct node *list, int n)
{
  for (; list != NULL && list->value != n;
         list = list->next)
    ;
  return list;
}
```

- Since list is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`.

Searching a Linked List

- This version of `search_list` might be a bit clearer if we used a `while` statement:

```
struct node *search_list(struct node *list, int n)
{
  while (list != NULL && list->value != n)
    list = list->next;
  return list;
}
```

Deleting a Node from a Linked List

- A big advantage of storing data in a linked list is that we can easily delete nodes.
- Deleting a node involves three steps:
  1. Locate the node to be deleted.
  2. Alter the previous node so that it "bypasses" the deleted node.
  3. Call `free` to reclaim the space occupied by the deleted node.
- Step 1 is harder than it looks, because step 2 requires changing the *previous* node.
- There are various solutions to this problem.

Deleting a Node from a Linked List

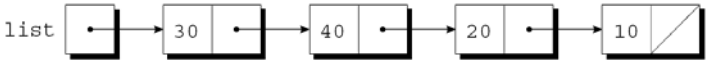
- The "trailing pointer" technique involves keeping a pointer to the previous node (`prev`) as well as a pointer to the current node (`cur`).
- Assume that `list` points to the list to be searched and `n` is the integer to be deleted.
- A loop that implements step 1:

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next)
  ;
```

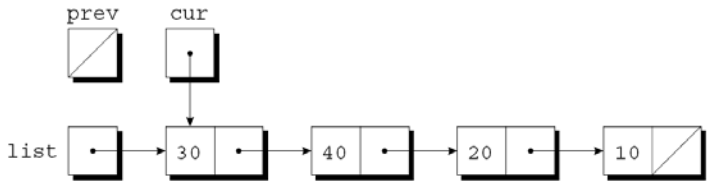
- When the loop terminates, `cur` points to the node to be deleted and `prev` points to the previous node.

Deleting a Node from a Linked List

- Assume that `list` has the following appearance and `n` is 20:

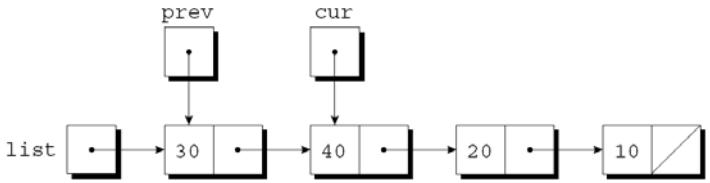


- After `cur = list, prev = NULL` has been executed:



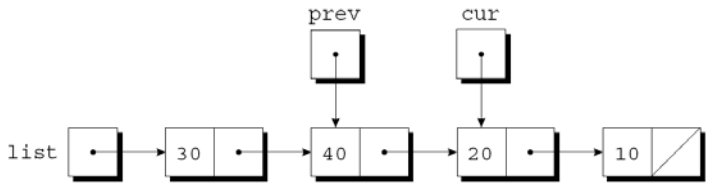
Deleting a Node from a Linked List

- The test `cur != NULL && cur->value != n` is true, since `cur` is pointing to a node and the node doesn't contain 20.
- After `prev = cur, cur = cur->next` has been executed:



Deleting a Node from a Linked List

- The test `cur != NULL && cur->value != n` is again true, so `prev = cur, cur = cur->next` is executed once more:



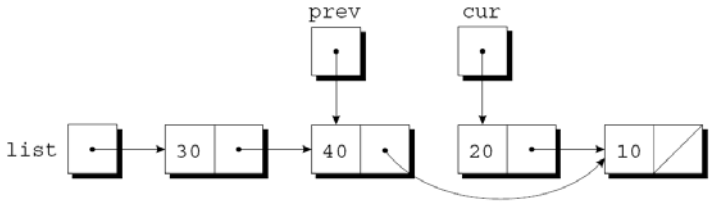
- Since `cur` now points to the node containing 20, the condition `cur->value != n` is false and the loop terminates.

Deleting a Node from a Linked List

- Next, we'll perform the bypass required by step 2.
- The statement

  `prev->next = cur->next;`

  makes the pointer in the previous node point to the node *after* the current node:



Deleting a Node from a Linked List

- Step 3 is to release the memory occupied by the current node:

  `free(cur);`

Deleting a Node from a Linked List

- The `delete_from_list` function uses the strategy just outlined.
- When given a list and an integer `n`, the function deletes the first node containing `n`.
- If no node contains `n`, `delete_from_list` does nothing.
- In either case, the function returns a pointer to the list.
- Deleting the first node in the list is a special case that requires a different bypass step.

Deleting a Node from a Linked List

```
struct node *delete_from_list(struct node *list, int n)
{
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next)
    ;
  if (cur == NULL)
    return list;              /* n was not found */
  if (prev == NULL)
    list = list->next;        /* n is in the first node */
  else
    prev->next = cur->next;   /* n is in some other node */
  free(cur);
  return list;
}
```


Ordered Lists

- When the nodes of a list are kept in order—sorted by the data stored inside the nodes—we say that the list is ***ordered.***
- Inserting a node into an ordered list is more difficult, because the node won't always be put at the beginning of the list.
- However, searching is faster: we can stop looking after reaching the point at which the desired node would have been located.

Program: Maintaining a Parts Database (Revisited)

- The `inventory2.c` program is a modification of the parts database program of Chapter 16, with the database stored in a linked list this time.
- Advantages of using a linked list:
  - No need to put a limit on the size of the database.
  - Database can easily be kept sorted by part number.
- In the original program, the database wasn't sorted.

Program: Maintaining a Parts Database (Revisited)

- The `part` structure will contain an additional member (a pointer to the next node):

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};
```

- `inventory` will point to the first node in the list:

```
struct part *inventory = NULL;
```

Program: Maintaining a Parts Database (Revisited)

- Most of the functions in the new program will closely resemble their counterparts in the original program.
- `find_part` and `insert` will be more complex, however, since we'll keep the nodes in the inventory list sorted by part number.

Program: Maintaining a Parts Database (Revisited)

- In the original program, `find_part` returns an index into the `inventory` array.
- In the new program, `find_part` will return a pointer to the node that contains the desired part number.
- If it doesn't find the part number, `find_part` will return a null pointer.

Program: Maintaining a Parts Database (Revisited)

- Since the list of parts is sorted, `find_part` can stop when it finds a node containing a part number that's greater than or equal to the desired part number.
- `find_part`'s search loop:

```
for (p = inventory;
     p != NULL && number > p->number;
     p = p->next)
  ;
```

- When the loop terminates, we'll need to test whether the part was found:

```
if (p != NULL && number == p->number)
  return p;
```


Program: Maintaining a Parts Database (Revisited)

- The original version of `insert` stores a new part in the next available array element.
- The new version must determine where the new part belongs in the list and insert it there.
- It will also check whether the part number is already present in the list.
- A loop that accomplishes both tasks:

```
for (cur = inventory, prev = NULL;
     cur != NULL && new_node->number > cur->number;
     prev = cur, cur = cur->next)
  ;
```

Program: Maintaining a Parts Database (Revisited)

- Once the loop terminates, `insert` will check whether `cur` isn't `NULL` and whether `new_node->number` equals `cur->number`.
  - If both are true, the part number is already in the list.
  - Otherwise, `insert` will insert a new node between the nodes pointed to by `prev` and `cur`.
- This strategy works even if the new part number is larger than any in the list.
- Like the original program, this version requires the `read_line` function of Chapter 16.

inventory2.c

```
/* Maintains a parts database (linked list version) */

#include <stdio.h>
#include <stdlib.h>
#include "readline.h"
#define NAME_LEN 25

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
  struct part *next;
};

struct part *inventory = NULL;   /* points to first part */

struct part *find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```



```
/*************************************************************
 * main: Prompts the user to enter an operation code,        *
 *       then calls a function to perform the requested      *
 *       action. Repeats until the user enters the           *
 *       command 'q'. Prints an error message if the user    *
 *       enters an illegal code.                             *
 *************************************************************/
int main(void)
{
  char code;

  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```



```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/*************************************************************
 * find_part: Looks up a part number in the inventory        *
 *            list. Returns a pointer to the node            *
 *            containing the part number; if the part        *
 *            number is not found, returns NULL.             *
 *************************************************************/
struct part *find_part(int number)
{
  struct part *p;

  for (p = inventory;
       p != NULL && number > p->number;
       p = p->next)
    ;
  if (p != NULL && number == p->number)
    return p;
  return NULL;
}
```



```
/*************************************************************
 * insert: Prompts the user for information about a new      *
 *         part and then inserts the part into the           *
 *         inventory list; the list remains sorted by        *
 *         part number. Prints an error message and          *
 *         returns prematurely if the part already exists    *
 *         or space could not be allocated for the part.     *
 *************************************************************/
void insert(void)
{
  struct part *cur, *prev, *new_node;

  new_node = malloc(sizeof(struct part));
  if (new_node == NULL) {
    printf("Database is full; can't add more parts.\n");
    return;
  }

  printf("Enter part number: ");
  scanf("%d", &new_node->number);
```



```
  for (cur = inventory, prev = NULL;
       cur != NULL && new_node->number > cur->number;
       prev = cur, cur = cur->next)
    ;
  if (cur != NULL && new_node->number == cur->number) {
    printf("Part already exists.\n");
    free(new_node);
    return;
  }

  printf("Enter part name: ");
  read_line(new_node->name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &new_node->on_hand);

  new_node->next = cur;
  if (prev == NULL)
    inventory = new_node;
  else
    prev->next = new_node;
}
```



```
/*************************************************************
 * search: Prompts the user to enter a part number, then     *
 *         looks up the part in the database. If the part    *
 *         exists, prints the name and quantity on hand;     *
 *         if not, prints an error message.                  *
 *************************************************************/
void search(void)
{
  int number;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Part name: %s\n", p->name);
    printf("Quantity on hand: %d\n", p->on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/*************************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 *************************************************************/
void update(void)
{
  int number, change;
  struct part *p;

  printf("Enter part number: ");
  scanf("%d", &number);
  p = find_part(number);
  if (p != NULL) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    p->on_hand += change;
  } else
    printf("Part not found.\n");
}
```



```
/*************************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Part numbers will appear in   *
 *        ascending order.                                *
 *************************************************************/
void print(void)
{
  struct part *p;
  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (p = inventory; p != NULL; p = p->next)
    printf("%7d       %-25s%11d\n", p->number, p->name,
           p->on_hand);
}
```


Pointers to Pointers

- Chapter 13 introduced the idea of a *pointer* to a *pointer*.
- The concept of "pointers to pointers" also pops up frequently in the context of linked data structures.
- In particular, when an argument to a function is a pointer variable, we may want the function to be able to modify the variable.
- Doing so requires the use of a pointer to a pointer.

Pointers to Pointers

- The `add_to_list` function is passed a pointer to the first node in a list; it returns a pointer to the first node in the updated list:

```
struct node *add_to_list(struct node *list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = list;
  return new_node;
}
```

Pointers to Pointers

- Modifying `add_to_list` so that it assigns `new_node` to `list` instead of returning `new_node` doesn't work.
- Example:

```
add_to_list(first, 10);
```

- At the point of the call, `first` is copied into `list`.
- If the function changes the value of `list`, making it point to the new node, `first` is not affected.

Pointers to Pointers

- Getting `add_to_list` to modify `first` requires passing `add_to_list` a *pointer* to `first`:

```
void add_to_list(struct node **list, int n)
{
  struct node *new_node;

  new_node = malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Error: malloc failed in add_to_list\n");
    exit(EXIT_FAILURE);
  }
  new_node->value = n;
  new_node->next = *list;
  *list = new_node;
}
```


Pointers to Pointers

- When the new version of `add_to_list` is called, the first argument will be the address of `first`:

```
add_to_list(&first, 10);
```

- Since `list` is assigned the address of `first`, we can use `*list` as an alias for `first`.
- In particular, assigning `new_node` to `*list` will modify `first`.

Pointers to Functions

- C doesn't require that pointers point only to *data;* it's also possible to have pointers to *functions.*
- Functions occupy memory locations, so every function has an address.
- We can use function pointers in much the same way we use pointers to data.
- Passing a function pointer as an argument is fairly common.

Function Pointers as Arguments

- A function named `integrate` that integrates a mathematical function `f` can be made as general as possible by passing `f` as an argument.
- Prototype for `integrate`:

```
double integrate(double (*f)(double),
                 double a, double b);
```

The parentheses around `*f` indicate that `f` is a pointer to a function.

- An alternative prototype:

```
double integrate(double f(double),
                 double a, double b);
```


Function Pointers as Arguments

- A call of `integrate` that integrates the `sin` (sine) function from 0 to π/2:

```
result = integrate(sin, 0.0, PI / 2);
```

- When a function name isn't followed by parentheses, the C compiler produces a pointer to the function.
- Within the body of `integrate`, we can call the function that `f` points to:

```
y = (*f)(x);
```

- Writing `f(x)` instead of `(*f)(x)` is allowed.

The `qsort` Function

- Some of the most useful functions in the C library require a function pointer as an argument.
- One of these is `qsort`, which belongs to the `<stdlib.h>` header.
- `qsort` is a general-purpose sorting function that's capable of sorting any array.

The `qsort` Function

- `qsort` must be told how to determine which of two array elements is "smaller."
- This is done by passing `qsort` a pointer to a ***comparison function.***
- When given two pointers `p` and `q` to array elements, the comparison function must return an integer that is:
  - *Negative* if `*p` is "less than" `*q`
  - *Zero* if `*p` is "equal to" `*q`
  - *Positive* if `*p` is "greater than" `*q`

The qsort Function

- Prototype for `qsort`:

```
void qsort(void *base, size_t nmemb, size_t size,
  int (*compar)(const void *, const void *));
```

- `base` must point to the first element in the array (or the first element in the portion to be sorted).
- `nmemb` is the number of elements to be sorted.
- `size` is the size of each array element, measured in bytes.
- `compar` is a pointer to the comparison function.

The qsort Function

- When `qsort` is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.
- A call of `qsort` that sorts the `inventory` array of Chapter 16:

```
qsort(inventory, num_parts,
      sizeof(struct part), compare_parts);
```

- `compare_parts` is a function that compares two `part` structures.

The qsort Function

- Writing the `compare_parts` function is tricky.
- `qsort` requires that its parameters have type `void *`, but we can't access the members of a `part` structure through a `void *` pointer.
- To solve the problem, `compare_parts` will assign its parameters, `p` and `q`, to variables of type `struct part *`.

The qsort Function

- A version of `compare_parts` that can be used to sort the `inventory` array into ascending order by part number:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

The qsort Function

- Most C programmers would write the function more concisely:

```
int compare_parts(const void *p, const void *q)
{
  if (((struct part *) p)->number <
      ((struct part *) q)->number)
    return -1;
  else if (((struct part *) p)->number ==
           ((struct part *) q)->number)
    return 0;
  else
    return 1;
}
```


The qsort Function

- `compare_parts` can be made even shorter by removing the `if` statements:

```
int compare_parts(const void *p, const void *q)
{
  return ((struct part *) p)->number -
         ((struct part *) q)->number;
}
```


The qsort Function

- A version of `compare_parts` that can be used to sort the `inventory` array by part name instead of part number:

```
int compare_parts(const void *p, const void *q)
{
  return strcmp(((struct part *) p)->name,
                ((struct part *) q)->name);
}
```


Other Uses of Function Pointers

- Although function pointers are often used as arguments, that's not all they're good for.
- C treats pointers to functions just like pointers to data.
- They can be stored in variables or used as elements of an array or as members of a structure or union.
- It's even possible for functions to return function pointers.

Other Uses of Function Pointers

- A variable that can store a pointer to a function with an `int` parameter and a return type of `void`:

```
void (*pf)(int);
```

- If `f` is such a function, we can make `pf` point to `f` in the following way:

```
pf = f;
```

- We can now call `f` by writing either

```
(*pf)(i);
```

or

```
pf(i);
```


Other Uses of Function Pointers

- An array whose elements are function pointers:

```
void (*file_cmd[])(void) = {new_cmd,
                            open_cmd,
                            close_cmd,
                            close_all_cmd,
                            save_cmd,
                            save_as_cmd,
                            save_all_cmd,
                            print_cmd,
                            exit_cmd
                           };
```


Other Uses of Function Pointers

- A call of the function stored in position `n` of the `file_cmd` array:

```
(*file_cmd[n])();  /* or file_cmd[n](); */
```

- We could get a similar effect with a `switch` statement, but using an array of function pointers provides more flexibility.

Program: Tabulating the Trigonometric Functions

- The `tabulate.c` program prints tables showing the values of the `cos`, `sin`, and `tan` functions.
- The program is built around a function named `tabulate` that, when passed a function pointer `f`, prints a table showing the values of `f`.
- `tabulate` uses the `ceil` function.
- When given an argument `x` of `double` type, `ceil` returns the smallest integer that's greater than or equal to `x`.

Program: Tabulating the Trigonometric Functions

- A session with `tabulate.c`:

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1
```

```
      x         cos(x)
-------     -------
0.00000     1.00000
0.10000     0.99500
0.20000     0.98007
0.30000     0.95534
0.40000     0.92106
0.50000     0.87758
```


Program: Tabulating the Trigonometric Functions

```
      x         sin(x)
-------     -------
0.00000     0.00000
0.10000     0.09983
0.20000     0.19867
0.30000     0.29552
0.40000     0.38942
0.50000     0.47943

      x         tan(x)
-------     -------
0.00000     0.00000
0.10000     0.10033
0.20000     0.20271
0.30000     0.30934
0.40000     0.42279
0.50000     0.54630
```


tabulate.c

```
/* Tabulates values of trigonometric functions */

#include <math.h>
#include <stdio.h>

void tabulate(double (*f)(double), double first,
              double last, double incr);

int main(void)
{
  double final, increment, initial;

  printf("Enter initial value: ");
  scanf("%lf", &initial);

  printf("Enter final value: ");
  scanf("%lf", &final);

  printf("Enter increment: ");
  scanf("%lf", &increment);
```



```
  printf("\n      x         cos(x)"
         "\n   -------    -------\n");
  tabulate(cos, initial, final, increment);

  printf("\n      x         sin(x)"
         "\n   -------    -------\n");
  tabulate(sin, initial, final, increment);

  printf("\n      x         tan(x)"
         "\n   -------    -------\n");
  tabulate(tan, initial, final, increment);

  return 0;
}

void tabulate(double (*f)(double), double first,
              double last, double incr)
{
  double x;
  int i, num_intervals;

  num_intervals = ceil((last - first) / incr);
  for (i = 0; i <= num_intervals; i++) {
    x = first + i * incr;
    printf("%10.5f %10.5f\n", x, (*f)(x));
  }
}
```

Restricted Pointers (C99)

- In C99, the keyword `restrict` may appear in the declaration of a pointer:

```
int * restrict p;
```

`p` is said to be a ***restricted pointer.***

- The intent is that if `p` points to an object that is later modified, then that object is not accessed in any way other than through `p`.
- Having more than one way to access an object is often called ***aliasing.***

Restricted Pointers (C99)

- Consider the following code:

```
int * restrict p;
int * restrict q;
p = malloc(sizeof(int));
```

- Normally it would be legal to copy `p` into `q` and then modify the integer through `q`:

```
q = p;
*q = 0;  /* causes undefined behavior */
```

- Because `p` is a restricted pointer, the effect of executing the statement `*q = 0;` is undefined.

Restricted Pointers (C99)

- To illustrate the use of `restrict`, consider the `memcpy` and `memmove` functions.
- The C99 prototype for `memcpy`, which copies bytes from one object (pointed to by `s2`) to another (pointed to by `s1`):

```
void *memcpy(void * restrict s1,
             const void * restrict s2,
             size_t n);
```

- The use of `restrict` with both `s1` and `s2` indicates that the objects to which they point shouldn't overlap.

Restricted Pointers (C99)

- In contrast, `restrict` doesn't appear in the prototype for `memmove`:

```
void *memmove(void *s1, const void *s2,
              size_t n);
```

- `memmove` is similar to `memcpy`, but is guaranteed to work even if the source and destination overlap.
- Example of using `memmove` to shift the elements of an array:

```
int a[100];
...
memmove(&a[0], &a[1], 99 * sizeof(int));
```

Restricted Pointers (C99)

- Prior to C99, there was no way to document the difference between `memcpy` and `memmove`.
- The prototypes for the two functions were nearly identical:

```
void *memcpy(void *s1, const void *s2,
             size_t n);
void *memmove(void *s1, const void *s2,
              size_t n);
```

- The use of `restrict` in the C99 version of `memcpy`'s prototype is a warning that the `s1` and `s2` objects should not overlap.

Restricted Pointers (C99)

- `restrict` provides information to the compiler that may enable it to produce more efficient code—a process known as ***optimization.***
- The C99 standard guarantees that `restrict` has no effect on the behavior of a program that conforms to the standard.
- Most programmers won't use `restrict` unless they're fine-tuning a program to achieve the best possible performance.

Flexible Array Members (C99)

- Occasionally, we'll need to define a structure that contains an array of an unknown size.
- For example, we might want a structure that stores the characters in a string together with the string's length:

```
struct vstring {
  int len;
  char chars[N];
};
```

- Using a fixed-length array is undesirable: it limits the length of the string and wastes memory.

Flexible Array Members (C99)

- C programmers traditionally solve this problem by declaring the length of `chars` to be 1 and then dynamically allocating each string:

```
struct vstring {
  int len;
  char chars[1];
};
…
struct vstring *str =
  malloc(sizeof(struct vstring) + n - 1);
str->len = n;
```

- This technique is known as the "struct hack."

Flexible Array Members (C99)

- The struct hack is supported by many compilers.
- Some (including GCC) even allow the `chars` array to have zero length.
- The C89 standard doesn't guarantee that the struct hack will work, but a C99 feature known as the ***flexible array member*** serves the same purpose.

Flexible Array Members (C99)

- When the last member of a structure is an array, its length may be omitted:

```
struct vstring {
  int len;
  char chars[]; /* flexible array member - C99 only */
};
```

- The length of the array isn't determined until memory is allocated for a `vstring` structure:

```
struct vstring *str =
  malloc(sizeof(struct vstring) + n);
str->len = n;
```

`sizeof` ignores the `chars` member when computing the size of the structure.

Flexible Array Members (C99)

- Special rules for structures that contain a flexible array member:
  - The flexible array must be the last member.
  - The structure must have at least one other member.
- Copying a structure that contains a flexible array member will copy the other members but not the flexible array itself.

Flexible Array Members (C99)

- A structure that contains a flexible array member is an ***incomplete type.***
- An incomplete type is missing part of the information needed to determine how much memory it requires.
- Incomplete types are subject to various restrictions.
- In particular, an incomplete type can't be a member of another structure or an element of an array.
- However, an array may contain pointers to structures that have a flexible array member.

Chapter 18

Declarations

Declaration Syntax

- Declarations furnish information to the compiler about the meaning of identifiers.
- Examples:

```
int i;
float f(float);
```

- General form of a declaration:

declaration-specifiers declarators ;

- ***Declaration specifiers*** describe the properties of the variables or functions being declared.
- ***Declarators*** give their names and may provide additional information about their properties.

Declaration Syntax

- Declaration specifiers fall into three categories:
  - Storage classes
  - Type qualifiers
  - Type specifiers
- C99 has a fourth category, ***function specifiers,*** which are used only in function declarations.
  - This category has one member, the keyword `inline`.
- Type qualifiers and type specifiers should follow the storage class, but there are no other restrictions on their order.

Declaration Syntax

- There are four ***storage classes:*** `auto`, `static`, `extern`, and `register`.
- At most one storage class may appear in a declaration; if present, it should come first.
- In C89, there are only two ***type qualifiers:*** `const` and `volatile`.
- C99 has a third type qualifier, `restrict`.
- A declaration may contain zero or more type qualifiers.

Declaration Syntax

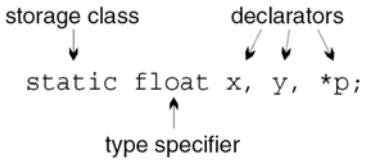
- The keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all ***type specifiers.***
- The order in which they are combined doesn't matter.
  - `int unsigned long` is the same as `long unsigned int`.
- Type specifiers also include specifications of structures, unions, and enumerations.
  - Examples: `struct point { int x, y; }`, `struct { int x, y; }`, `struct point`.
- `typedef` names are also type specifiers.

Declaration Syntax

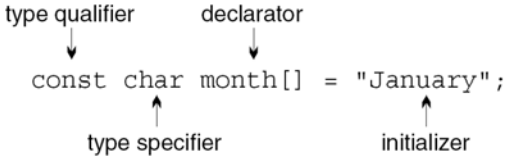
- Declarators include:
  - Identifiers (names of simple variables)
  - Identifiers followed by `[]` (array names)
  - Identifiers preceded by `*` (pointer names)
  - Identifiers followed by `()` (function names)
- Declarators are separated by commas.
- A declarator that represents a variable may be followed by an initializer.

Declaration Syntax

- A declaration with a storage class and three declarators:

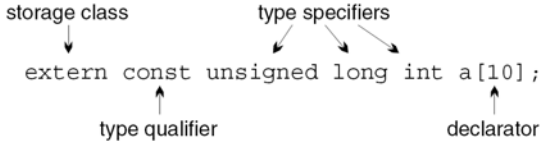


- A declaration with a type qualifier and initializer but no storage class:

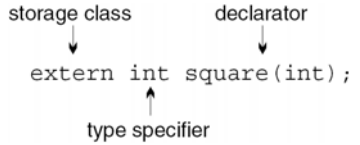


Declaration Syntax

- A declaration with a storage class, a type qualifier, and three type specifiers:



- Function declarations may have a storage class, type qualifiers, and type specifiers:



Storage Classes

- Storage classes can be specified for variables and—to a lesser extent—functions and parameters.
- Recall that the term *block* refers to the body of a function (the part in braces) or a compound statement, possibly containing declarations.

Properties of Variables

- Every variable in a C program has three properties:
  - Storage duration
  - Scope
  - Linkage

Properties of Variables

- The ***storage duration*** of a variable determines when memory is set aside for the variable and when that memory is released.
  - ***Automatic storage duration:*** Memory for variable is allocated when the surrounding block is executed and deallocated when the block terminates.
  - ***Static storage duration:*** Variable stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely.

Properties of Variables

- The ***scope*** of a variable is the portion of the program text in which the variable can be referenced.
  - ***Block scope:*** Variable is visible from its point of declaration to the end of the enclosing block.
  - ***File scope:*** Variable is visible from its point of declaration to the end of the enclosing file.

Properties of Variables

- The ***linkage*** of a variable determines the extent to which it can be shared.
  - ***External linkage:*** Variable may be shared by several (perhaps all) files in a program.
  - ***Internal linkage:*** Variable is restricted to a single file but may be shared by the functions in that file.
  - ***No linkage:*** Variable belongs to a single function and can't be shared at all.

Properties of Variables

- Example:

```
int i;        /* static storage duration, file scope, external linkage */

void f(void)
{
  int j;      /* automatic storage duration, block scope, no linkage */
}
```

- We can alter these properties by specifying an explicit storage class: `auto`, `static`, `extern`, or `register`.

Properties of Variables

- The default storage duration, scope, and linkage of a variable depend on where it's declared:
  - Variables declared *inside* a block (including a function body) have *automatic* storage duration, *block* scope, and *no* linkage.
  - Variables declared *outside* any block, at the outermost level of a program, have *static* storage duration, *file* scope, and *external* linkage.

The `auto` Storage Class

- The `auto` storage class is legal only for variables that belong to a block.
- An `auto` variable has automatic storage duration, block scope, and no linkage.
- The `auto` storage class is almost never specified explicitly.

The `static` Storage Class

- The `static` storage class can be used with all variables, regardless of where they're declared.
  - When used *outside* a block, `static` specifies that a variable has internal linkage.
  - When used *inside* a block, `static` changes the variable's storage duration from automatic to static.

The `static` Storage Class

- Example:

```
static int i;

void f(void)
{
  static int j;
}
```

static storage duration
file scope
internal linkage

static storage duration
block scope
no linkage

The `static` Storage Class

- When used outside a block, `static` hides a variable within a file:

```
static int i;  /* no access to i in other files */

void f1(void)
{
  /* has access to i */
}

void f2(void)
{
  /* has access to i */
}
```

- This use of `static` is helpful for implementing information hiding.

The `static` Storage Class

- A `static` variable declared within a block resides at the same storage location throughout program execution.
- A `static` variable retains its value indefinitely.
- Properties of `static` variables:
  - A `static` variable is initialized only once, prior to program execution.
  - A `static` variable declared inside a function is shared by all calls of the function, including recursive calls.
  - A function may return a pointer to a `static` variable.

The `static` Storage Class

- Declaring a local variable to be `static` allows a function to retain information between calls.
- More often, we'll use `static` for reasons of efficiency:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] =
    "0123456789ABCDEF";

  return hex_chars[digit];
}
```

- Declaring `hex_chars` to be `static` saves time, because `static` variables are initialized only once.

The `extern` Storage Class

- The `extern` storage class enables several source files to share the same variable.
- A variable declaration that uses `extern` doesn't cause memory to be allocated for the variable:

  `extern int i;`

  In C terminology, this is not a *definition* of `i`.
- An `extern` declaration tells the compiler that we need access to a variable that's defined elsewhere.
- A variable can have many *declarations* in a program but should have only one *definition*.

The `extern` Storage Class

- There's one exception to the rule that an `extern` declaration of a variable isn't a definition.
- An `extern` declaration that initializes a variable serves as a definition of the variable.
- For example, the declaration

  `extern int i = 0;`

  is effectively the same as

  `int i = 0;`
- This rule prevents multiple `extern` declarations from initializing a variable in different ways.

The `extern` Storage Class

- A variable in an `extern` declaration always has static storage duration.
- If the declaration is inside a block, the variable has block scope; otherwise, it has file scope:

```
extern int i;      static storage duration
                   file scope
                   ? linkage

void f(void)
{
  extern int j;    static storage duration
                   block scope
                   ? linkage
}
```

The `extern` Storage Class

- Determining the linkage of an `extern` variable is a bit harder.
  - If the variable was declared `static` earlier in the file (outside of any function definition), then it has internal linkage.
  - Otherwise (the normal case), the variable has external linkage.

The `register` Storage Class

- Using the `register` storage class in the declaration of a variable asks the compiler to store the variable in a register.
- A ***register*** is a high-speed storage area located in a computer's CPU.
- Specifying the storage class of a variable to be `register` is a request, not a command.
- The compiler is free to store a `register` variable in memory if it chooses.

The `register` Storage Class

- The `register` storage class is legal only for variables declared in a block.
- A `register` variable has the same storage duration, scope, and linkage as an `auto` variable.
- Since registers don't have addresses, it's illegal to use the `&` operator to take the address of a `register` variable.
- This restriction applies even if the compiler has elected to store the variable in memory.

The `register` Storage Class

- `register` is best used for variables that are accessed and/or updated frequently.
- The loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;

  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

The `register` Storage Class

- `register` isn't as popular as it once was.
- Many of today's compilers can determine automatically which variables would benefit from being kept in registers.
- Still, using `register` provides useful information that can help the compiler optimize the performance of a program.
- In particular, the compiler knows that a `register` variable can't have its address taken, and therefore can't be modified through a pointer.

The Storage Class of a Function

- Function declarations (and definitions) may include a storage class.
- The only options are `extern` and `static`:
  - `extern` specifies that the function has external linkage, allowing it to be called from other files.
  - `static` indicates internal linkage, limiting use of the function's name to the file in which it's defined.
- If no storage class is specified, the function is assumed to have external linkage.

The Storage Class of a Function

- Examples:

```
extern int f(int i);
static int g(int i);
int h(int i);
```

- Using `extern` is unnecessary, but `static` has benefits:
  - ***Easier maintenance.*** A `static` function isn't visible outside the file in which its definition appears, so future modifications to the function won't affect other files.
  - ***Reduced "name space pollution."*** Names of `static` functions don't conflict with names used in other files.

The Storage Class of a Function

- Function parameters have the same properties as `auto` variables: automatic storage duration, block scope, and no linkage.
- The only storage class that can be specified for parameters is `register`.

Summary

- A program fragment that shows all possible ways to include—or omit—storage classes in declarations of variables and parameters:

```
int a;
extern int b;
static int c;

void f(int d, register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```


Summary

Name	*Storage Duration*	*Scope*	*Linkage*
a	static	file	external
b	static	file	†
c	static	file	internal
d	automatic	block	none
e	automatic	block	none
g	automatic	block	none
h	automatic	block	none
i	static	block	none
j	static	block	†
k	automatic	block	none

†In most cases, `b` and `j` will be defined in another file and will have external linkage.

Summary

- Of the four storage classes, the most important are `static` and `extern`.
- `auto` has no effect, and modern compilers have made `register` less important.

Type Qualifiers

- There are two type qualifiers: `const` and `volatile`.
  - C99 has a third type qualifier, `restrict`, which is used only with pointers.
- `volatile` is discussed in Chapter 20.
- `const` is used to declare "read-only" objects.
- Examples:

```
const int n = 10;
const int tax_brackets[] =
  {750, 2250, 3750, 5250, 7000};
```

Type Qualifiers

- Advantages of declaring an object to be `const`:
  - Serves as a form of documentation.
  - Allows the compiler to check that the value of the object isn't changed.
  - Alerts the compiler that the object can be stored in ROM (read-only memory).

Type Qualifiers

- It might appear that `const` serves the same role as the `#define` directive, but there are significant differences between the two features.
- `#define` can be used to create a name for a numerical, character, or string constant, but `const` can create read-only objects of *any* type.
- `const` objects are subject to the same scope rules as variables; constants created using `#define` aren't.

Type Qualifiers

- The value of a `const` object, unlike the value of a macro, can be viewed in a debugger.
- Unlike macros, `const` objects can't be used in constant expressions:

```
const int n = 10;
int a[n];           /*** WRONG ***/
```

- It's legal to apply the address operator (`&`) to a `const` object, since it has an address; a macro doesn't have an address.

Type Qualifiers

- There are no absolute rules that dictate when to use `#define` and when to use `const`.
- `#define` is good for constants that represent numbers or characters.

Declarators

- In the simplest case, a declarator is just an identifier:

```
int i;
```

- Declarators may also contain the symbols `*`, `[]`, and `()`.
- A declarator that begins with `*` represents a pointer:

```
int *p;
```


Declarators

- A declarator that ends with `[]` represents an array:

```
int a[10];
```

- The brackets may be left empty if the array is a parameter, if it has an initializer, or if its storage class is `extern`:

```
extern int a[];
```

- In the case of a multidimensional array, only the first set of brackets can be empty.

Declarators

- C99 provides two additional options for what goes between the brackets in the declaration of an array parameter:
  - The keyword `static`, followed by an expression that specifies the array's minimum length.
  - The `*` symbol, which can be used in a function prototype to indicate a variable-length array argument.
- Chapter 9 discusses both features.

Declarators

- A declarator that ends with `()` represents a function:

```
int abs(int i);
void swap(int *a, int *b);
int find_largest(int a[], int n);
```

- C allows parameter names to be omitted in a function declaration:

```
int abs(int);
void swap(int *, int *);
int find_largest(int [], int);
```

Declarators

- The parentheses can even be left empty:

```
int abs();
void swap();
int find_largest();
```

  This provides no information about the arguments.
- Putting the word `void` between the parentheses is different: it indicates that there are no arguments.
- The empty-parentheses style doesn't let the compiler check whether function calls have the right arguments.

Declarators

- Declarators in actual programs often combine the `*`, `[]`, and `()` notations.
- An array of 10 pointers to integers:

```
int *ap[10];
```

- A function that has a `float` argument and returns a pointer to a `float`:

```
float *fp(float);
```

- A pointer to a function with an `int` argument and a `void` return type:

```
void (*pf)(int);
```


Deciphering Complex Declarations

- But what about declarators like the one in the following declaration?

```
int *(*x[10])(void);
```

- It's not obvious whether `x` is a pointer, an array, or a function.

Deciphering Complex Declarations

- Rules for understanding declarations:
  - ***Always read declarators from the inside out.*** Locate the identifier that's being declared, and start deciphering the declaration from there.
  - ***When there's a choice, always favor `[]` and `()` over `*`.*** Parentheses can be used to override the normal priority of `[]` and `()` over `*`.

Deciphering Complex Declarations

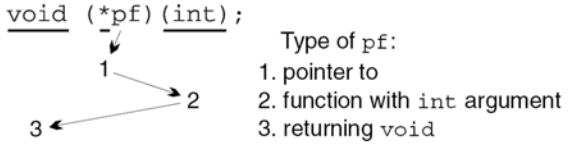
- Example 1:
  `int *ap[10];`
  `ap` is an *array* of *pointers*.
- Example 2:
  `float *fp(float);`
  `fp` is a *function* that returns a *pointer*.

Deciphering Complex Declarations

- Example 3:
  `void (*pf)(int);`
- Since `*pf` is enclosed in parentheses, `pf` must be a pointer.
- But `(*pf)` is followed by `(int)`, so `pf` must point to a function with an `int` argument.
- The word `void` represents the return type of this function.

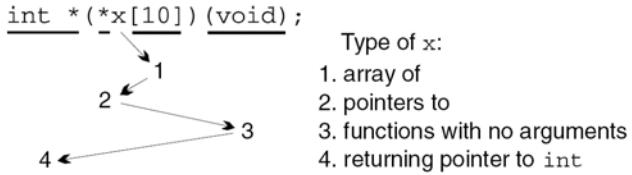
Deciphering Complex Declarations

- Understanding a complex declarator often involves zigzagging from one side of the identifier to the other:



Deciphering Complex Declarations

- A second example of "zigzagging":



Deciphering Complex Declarations

- Certain things can't be declared in C.
- Functions can't return arrays:

```
int f(int)[];       /*** WRONG ***/
```

- Functions can't return functions:

```
int g(int)(int);    /*** WRONG ***/
```

- Arrays of functions aren't possible, either:

```
int a[10](int);     /*** WRONG ***/
```

- In each case, pointers can be used to get the desired effect.
- For example, a function can't return an array, but it can return a *pointer* to an array.

Using Type Definitions to Simplify Declarations

- Some programmers use type definitions to help simplify complex declarations.
- Suppose that `x` is declared as follows:

```
int *(*x[10])(void);
```

- The following type definitions make `x`'s type easier to understand:

```
typedef int *Fcn(void);
typedef Fcn *Fcn_ptr;
typedef Fcn_ptr Fcn_ptr_array[10];
Fcn_ptr_array x;
```


Initializers

- For convenience, C allows us to specify initial values for variables as we're declaring them.
- To initialize a variable, we write the = symbol after its declarator, then follow that with an initializer.

Initializers

- The initializer for a simple variable is an expression of the same type as the variable:

```
int i = 5 / 2;    /* i is initially 2 */
```

- If the types don't match, C converts the initializer using the same rules as for assignment:

```
int j = 5.5;      /* converted to 5 */
```

- The initializer for a pointer variable must be an expression of the same type or of type `void *`:

```
int *p = &i;
```

Initializers

- The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

- In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

Initializers

- An initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

- If `LAST` and `FIRST` had been variables, the initializer would be illegal.

Initializers

- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n)
{
  int last = n - 1;
  …
}
```


Initializers

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions:

```
#define N 2

int powers[5] =
  {1, N, N * N, N * N * N, N * N * N * N};
```

- If `N` were a variable, the initializer would be illegal.
- In C99, this restriction applies only if the variable has static storage duration.

Initializers

- The initializer for an automatic structure or union can be another structure or union:

```
void g(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type.

Uninitialized Variables

- The initial value of a variable depends on its storage duration:
  - Variables with *automatic* storage duration have no default initial value.
  - Variables with *static* storage duration have the value zero by default.
- A static variable is correctly initialized based on its type, not simply set to zero bits.
- It's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero.

Inline Functions (C99)

- C99 function declarations may contain the keyword `inline`.
- `inline` is related to the concept of the "overhead" of a function call—the work required to call a function and later return from it.
- Although the overhead of a function call slows the program by only a tiny amount, it may add up in certain situations.

Inline Functions (C99)

- In C89, the only way to avoid the overhead of a function call is to use a parameterized macro.
- C99 offers a better solution to this problem: create an ***inline function.***
- The word "inline" suggests that the compiler replaces each call of the function by the machine instructions for the function.
- This technique may cause a minor increase in the size of the compiled program.

Inline Functions (C99)

- Declaring a function to be `inline` doesn't actually force the compiler to "inline" the function.
- It suggests that the compiler should try to make calls of the function as fast as possible, but the compiler is free to ignore the suggestion.

Inline Definitions (C99)

- An inline function has the keyword `inline` as one of its declaration specifiers:

```
inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

- `average` has external linkage, so other source files may contain calls of `average`.
- However, the definition of `average` isn't an external definition (it's an ***inline definition*** instead).
- Attempting to call `average` from another file will be considered an error.

Inline Definitions (C99)

- There are two ways to avoid this error.
- One option is to add the word `static` to the function definition:

```
static inline double average(double a, double b)
{
  return (a + b) / 2;
}
```

- `average` now has internal linkage, so it can't be called from other files.
- Other files may contain their own definitions of `average`, which might be the same or different.

Inline Definitions (C99)

- The other option is to provide an external definition for `average` so that calls are permitted from other files.
- One way to do this is to write the `average` function a second time (without using `inline`) and put this definition in a different source file.
- However, it's not a good idea to have two versions of a function: we can't guarantee that they'll remain consistent when the program is modified.

Inline Definitions (C99)

- A better approach is to put the inline definition of `average` in a header file:

```
#ifndef AVERAGE_H
#define AVERAGE_H

inline double average(double a, double b)
{
  return (a + b) / 2;
}

#endif
```

- Let's name this file `average.h`.

Inline Definitions (C99)

- Next, we'll create a matching source file, `average.c`:

```
#include "average.h"

extern double average(double a, double b);
```

- Any file that needs to call the `average` function can include `average.h`.
- The definition of `average` included from `average.h` will be treated as an external definition in `average.c`.

Inline Definitions (C99)

- A general rule: If all top-level declarations of a function in a file include `inline` but not `extern`, then the definition of the function in that file is inline.
- If the function is used anywhere in the program, an external definition of the function will need to be provided by some other file.

Inline Definitions (C99)

- When an inline function is called, the compiler has a choice:
  - Perform an ordinary call (using the function's external definition).
  - Perform inline expansion (using the function's inline definition).
- Because the choice is left to the compiler, it's crucial that the two definitions be consistent.
- The technique just discussed (using the `average.h` and `average.c` files) guarantees that the definitions are the same.

Restrictions on Inline Functions (C99)

- Restrictions on inline functions with external linkage:
  - May not define a modifiable `static` variable.
  - May not contain references to variables with internal linkage.
- Such a function is allowed to define a variable that is both `static` and `const`.
- However, each inline definition of the function may create its own copy of the variable.

Using Inline Functions with GCC (C99)

- Some compilers, including GCC, supported inline functions prior to the C99 standard.
- Their rules for using inline functions may vary from the standard.
- The scheme described earlier (using the `average.h` and `average.c` files) may not work with these compilers.
- Version 4.3 of GCC is expected to support inline functions in the way described in the C99 standard.

Using Inline Functions with GCC (C99)

- Functions that are specified to be both `static` and `inline` should work fine, regardless of the version of GCC.
- This strategy is legal in C99 as well, so it's the safest bet.
- A `static inline` function can be used within a single file or placed in a header file and included into any source file that needs to call the function.

Using Inline Functions with GCC (C99)

- A technique for sharing an inline function among multiple files that works with older versions of GCC but conflicts with C99:
  - Put a definition of the function in a header file.
  - Specify that the function is both `extern` and `inline`.
  - Include the header file into any source file that contains a call of the function.
  - Put a second copy of the definition—without the words `extern` and `inline`—in one of the source files.
- A final note about GCC: Functions are "inlined" only when the `-O` command-line option is used.

Chapter 19

Program Design

Introduction

- Most full-featured programs are at least 100,000 lines long.
- Although C wasn't designed for writing large programs, many large programs have been written in C.
- Writing large programs is quite different from writing small ones.

Introduction

- Issues that arise when writing a large program:
  - Style
  - Documentation
  - Maintenance
  - Design
- This chapter focuses on design techniques that can make C programs readable and maintainable.

Modules

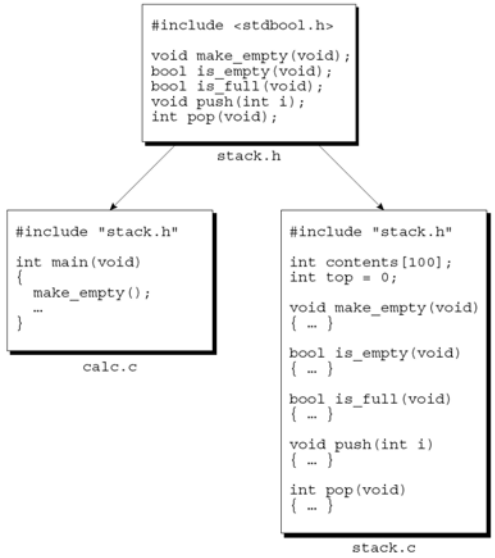
- It's often useful to view a program as a number of independent ***modules.***
- A module is a collection of services, some of which are made available to other parts of the program (the ***clients***).
- Each module has an ***interface*** that describes the available services.
- The details of the module—including the source code for the services themselves—are stored in the module's ***implementation.***

Modules

- In the context of C, “services” are functions.
- The interface of a module is a header file containing prototypes for the functions that will be made available to clients (source files).
- The implementation of a module is a source file that contains definitions of the module’s functions.

Modules

- The calculator program sketched in Chapter 15 consists of:
  - `calc.c`, which contains the `main` function
  - A stack module, stored in `stack.h` and `stack.c`
- `calc.c` is a *client* of the stack module.
- `stack.h` is the *interface* of the stack module.
- `stack.c` is the *implementation* of the module.



Modules

- The C library is itself a collection of modules.
- Each header in the library serves as the interface to a module.
  - `<stdio.h>` is the interface to a module containing I/O functions.
  - `<string.h>` is the interface to a module containing string-handling functions.

Modules

- Advantages of dividing a program into modules:
  - Abstraction
  - Reusability
  - Maintainability

Modules

- ***Abstraction.*** A properly designed module can be treated as an ***abstraction;*** we know what it does, but we don't worry about how it works.
- Thanks to abstraction, it's not necessary to understand how the entire program works in order to make changes to one part of it.
- Abstraction also makes it easier for several members of a team to work on the same program.

Modules

- ***Reusability.*** Any module that provides services is potentially reusable in other programs.
- Since it's often hard to anticipate the future uses of a module, it's a good idea to design modules for reusability.

Modules

- ***Maintainability.*** A small bug will usually affect only a single module implementation, making the bug easier to locate and fix.
- Rebuilding the program requires only a recompilation of the module implementation (followed by linking the entire program).
- An entire module implementation can be replaced if necessary.

Modules

- Maintainability is the most critical advantage.
- Most real-world programs are in service over a period of years
- During this period, bugs are discovered, enhancements are made, and modifications are made to meet changing requirements.
- Designing a program in a modular fashion makes maintenance much easier.

Modules

- Decisions to be made during modular design:
  - What modules should a program have?
  - What services should each module provide?
  - How should the modules be interrelated?

Cohesion and Coupling

- In a well-designed program, modules should have two properties.
- ***High cohesion.*** The elements of each module should be closely related to one another.
  - High cohesion makes modules easier to use and makes the entire program easier to understand.
- ***Low coupling.*** Modules should be as independent of each other as possible.
  - Low coupling makes it easier to modify the program and reuse modules.

Types of Modules

- Modules tend to fall into certain categories:
  - Data pools
  - Libraries
  - Abstract objects
  - Abstract data types

Types of Modules

- A ***data pool*** is a collection of related variables and/or constants.
  - In C, a module of this type is often just a header file.
  - `<float.h>` and `<limits.h>` are both data pools.
- A ***library*** is a collection of related functions.
  - `<string.h>` is the interface to a library of string-handling functions.

Types of Modules

- An ***abstract object*** is a collection of functions that operate on a hidden data structure.
- An ***abstract data type (ADT)*** is a type whose representation is hidden.
  - Client modules can use the type to declare variables but have no knowledge of the structure of those variables.
  - To perform an operation on such a variable, a client must call a function provided by the ADT.

Information Hiding

- A well-designed module often keeps some information secret from its clients.
  - Clients of the stack module have no need to know whether the stack is stored in an array, in a linked list, or in some other form.
- Deliberately concealing information from the clients of a module is known as ***information hiding.***

Information Hiding

- Primary advantages of information hiding:
  - ***Security.*** If clients don't know how a module stores its data, they won't be able to corrupt it by tampering with its internal workings.
  - ***Flexibility.*** Making changes—no matter how large—to a module's internals won't be difficult.

Information Hiding

- In C, the major tool for enforcing information hiding is the `static` storage class.
  - A `static` variable with file scope has internal linkage, preventing it from being accessed from other files, including clients of the module.
  - A `static` function can be directly called only by other functions in the same file.

A Stack Module

- To see the benefits of information hiding, let's look at two implementations of a stack module, one using an array and the other a linked list.
- `stack.h` is the module's header file.
- `stack1.c` uses an array to implement the stack.

stack.h

```
#ifndef STACK_H
#define STACK_H

#include <stdbool.h>   /* C99 only */

void make_empty(void);
bool is_empty(void);
bool is_full(void);
void push(int i);
int pop(void);

#endif
```


stack1.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

#define STACK_SIZE 100

static int contents[STACK_SIZE];
static int top = 0;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  top = 0;
}
```

```
bool is_empty(void)
{
  return top == 0;
}

bool is_full(void)
{
  return top == STACK_SIZE;
}

void push(int i)
{
  if (is_full())
    terminate("Error in push: stack is full.");
  contents[top++] = i;
}

int pop(void)
{
  if (is_empty())
    terminate("Error in pop: stack is empty.");
  return contents[--top];
}
```


A Stack Module

- Macros can be used to indicate whether a function or variable is "public" (accessible elsewhere in the program) or "private" (limited to a single file):

```
#define PUBLIC  /* empty */
#define PRIVATE static
```

- The word `static` has more than one use in C; `PRIVATE` makes it clear that we're using it to enforce information hiding.

A Stack Module

- The stack implementation redone using `PUBLIC` and `PRIVATE`:

```
PRIVATE int contents[STACK_SIZE];
PRIVATE int top = 0;

PRIVATE void terminate(const char *message) { … }

PUBLIC void make_empty(void) { … }

PUBLIC bool is_empty(void) { … }

PUBLIC bool is_full(void) { … }

PUBLIC void push(int i) { … }

PUBLIC int pop(void) { … }
```


A Stack Module

- `stack2.c` is a linked-list implementation of the stack module.

stack2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct node {
  int data;
  struct node *next;
};

static struct node *top = NULL;

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

void make_empty(void)
{
  while (!is_empty())
    pop();
}
```



```
bool is_empty(void)
{
  return top == NULL;
}

bool is_full(void)
{
  return false;
}

void push(int i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = top;
  top = new_node;
}
```



```
int pop(void)
{
  struct node *old_top;
  int i;

  if (is_empty())
    terminate("Error in pop: stack is empty.");

  old_top = top;
  i = top->data;
  top = top->next;
  free(old_top);
  return i;
}
```


A Stack Module

- Thanks to information hiding, it doesn't matter whether we use `stack1.c` or `stack2.c` to implement the stack module.
- Both versions match the module's interface, so we can switch from one to the other without having to make changes elsewhere in the program.

Abstract Data Types

- A module that serves as an abstract object has a serious disadvantage: there's no way to have multiple instances of the object.
- To accomplish this, we'll need to create a new *type*.
- For example, a `Stack` type can be used to create any number of stacks.

Abstract Data Types

- A program fragment that uses two stacks:

```
Stack s1, s2;

make_empty(&s1);
make_empty(&s2);
push(&s1, 1);
push(&s2, 2);
if (!is_empty(&s1))
  printf("%d\n", pop(&s1));  /* prints "1" */
```

- To clients, `s1` and `s2` are *abstractions* that respond to certain operations (`make_empty`, `is_empty`, `is_full`, `push`, and `pop`).

Abstract Data Types

- Converting the `stack.h` header so that it provides a `Stack` type requires adding a `Stack` (or `Stack *`) parameter to each function.

Abstract Data Types

- Changes to `stack.h` are shown in **bold**:

```
#define STACK_SIZE 100

typedef struct {
  int contents[STACK_SIZE];
  int top;
} Stack;

void make_empty(Stack *s);
bool is_empty(const Stack *s);
bool is_full(const Stack *s);
void push(Stack *s, int i);
int pop(Stack *s);
```

Abstract Data Types

- The stack parameters to `make_empty`, `push`, and `pop` need to be pointers, since these functions modify the stack.
- The parameter to `is_empty` and `is_full` doesn't need to be a pointer.
- Passing these functions a `Stack` *pointer* instead of a `Stack` *value* is done for efficiency, since the latter would result in a structure being copied.

Encapsulation

- Unfortunately, `Stack` isn't an *abstract* data type, since `stack.h` reveals what the `Stack` type really is.
- Nothing prevents clients from using a `Stack` variable as a structure:

```
Stack s1;

s1.top = 0;
s1.contents[top++] = 1;
```

- Providing access to the `top` and `contents` members allows clients to corrupt the stack.

Encapsulation

- Worse still, we can't change the way stacks are stored without having to assess the effect of the change on clients.
- What we need is a way to prevent clients from knowing how the `Stack` type is represented.
- C has only limited support for ***encapsulating*** types in this way.
- Newer C-based languages, including C++, Java, and C#, are better equipped for this purpose.

Incomplete Types

- The only tool that C gives us for encapsulation is the ***incomplete type.***
- Incomplete types are "types that describe objects but lack information needed to determine their sizes."
- Example:

```
struct t;  /* incomplete declaration of t */
```

- The intent is that an incomplete type will be completed elsewhere in the program.

Incomplete Types

- An incomplete type can't be used to declare a variable:

```
struct t s;   /*** WRONG ***/
```

- However, it's legal to define a pointer type that references an incomplete type:

```
typedef struct t *T;
```

- We can now declare variables of type `T`, pass them as arguments to functions, and perform other operations that are legal for pointers.

A Stack Abstract Data Type

- The following stack ADT will illustrate how abstract data types can be encapsulated using incomplete types.
- The stack will be implemented in three different ways.

Defining the Interface for the Stack ADT

- `stackADT.h` defines the stack ADT type and gives prototypes for the functions that represent stack operations.
- The `Stack` type will be a pointer to a `stack_type` structure (an incomplete type).
- The members of this structure will depend on how the stack is implemented.

stackADT.h
(version 1)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, int i);
int pop(Stack s);

#endif
```

Defining the Interface for the Stack ADT

- Clients that include `stackADT.h` will be able to declare variables of type `Stack`, each of which is capable of pointing to a `stack_type` structure.
- Clients can then call the functions declared in `stackADT.h` to perform operations on stack variables.
- However, clients can't access the members of the `stack_type` structure, since that structure will be defined in a separate file.

Defining the Interface for the Stack ADT

- A module generally doesn't need `create` and `destroy` functions, but an ADT does.
  - `create` dynamically allocates memory for a stack and initializes the stack to its "empty" state.
  - `destroy` releases the stack's dynamically allocated memory.

Defining the Interface for the Stack ADT

- `stackclient.c` can be used to test the stack ADT.
- It creates two stacks and performs a variety of operations on them.

stackclient.c

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

int main(void)
{
  Stack s1, s2;
  int n;

  s1 = create();
  s2 = create();

  push(s1, 1);
  push(s1, 2);

  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);
```

```
  n = pop(s1);
  printf("Popped %d from s1\n", n);
  push(s2, n);

  destroy(s1);

  while (!is_empty(s2))
    printf("Popped %d from s2\n", pop(s2));

  push(s2, 3);
  make_empty(s2);
  if (is_empty(s2))
    printf("s2 is empty\n");
  else
    printf("s2 is not empty\n");

  destroy(s2);

  return 0;
}
```


Defining the Interface for the Stack ADT

- Output if the stack ADT is implemented correctly:

```
Popped 2 from s1
Popped 1 from s1
Popped 1 from s2
Popped 2 from s2
s2 is empty
```


Implementing the Stack ADT Using a Fixed-Length Array

- There are several ways to implement the stack ADT.
- The simplest is to have the `stack_type` structure contain a fixed-length array:

```
struct stack_type {
  int contents[STACK_SIZE];
  int top;
};
```


stackADT.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

#define STACK_SIZE 100

struct stack_type {
  int contents[STACK_SIZE];
  int top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = 0;
  return s;
}

void destroy(Stack s)
{
  free(s);
}

void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}
```



```
bool is_full(Stack s)
{
  return s->top == STACK_SIZE;
}

void push(Stack s, int i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

int pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```


Changing the Item Type in the Stack ADT

- `stackADT.c` requires that stack items be integers, which is too restrictive.
- To make the stack ADT easier to modify for different item types, let's add a type definition to the `stackADT.h` header.
- It will define a type named `Item`, representing the type of data to be stored on the stack.

stackADT.h
(version 2)

```
#ifndef STACKADT_H
#define STACKADT_H

#include <stdbool.h>   /* C99 only */

typedef int Item;

typedef struct stack_type *Stack;

Stack create(void);
void destroy(Stack s);
void make_empty(Stack s);
bool is_empty(Stack s);
bool is_full(Stack s);
void push(Stack s, Item i);
Item pop(Stack s);

#endif
```

Changing the Item Type in the Stack ADT

- The `stackADT.c` file will need to be modified, but the changes are minimal.
- The updated `stack_type` structure:

```
struct stack_type {
  Item contents[STACK_SIZE];
  int top;
};
```

- The second parameter of `push` will now have type `Item`.
- `pop` now returns a value of type `Item`.

Changing the Item Type in the Stack ADT

- The `stackclient.c` file can be used to test the new `stackADT.h` and `stackADT.c` to verify that the `Stack` type still works.
- The item type can be changed by modifying the definition of `Item` in `stackADT.h`.

Implementing the Stack ADT Using a Dynamic Array

- Another problem with the stack ADT: each stack has a fixed maximum size.
- There's no way to have stacks with different capacities or to set the stack size as the program is running.
- Possible solutions to this problem:
  - Implement the stack as a linked list.
  - Store stack items in a dynamically allocated array.

Implementing the Stack ADT Using a Dynamic Array

- The latter approach involves modifying the `stack_type` structure.
- The `contents` member becomes a *pointer* to the array in which the items are stored:

```
struct stack_type {
  Item *contents;
  int top;
  int size;
};
```

- The `size` member stores the stack's maximum size.

Implementing the Stack ADT Using a Dynamic Array

- The `create` function will now have a parameter that specifies the desired maximum stack size:
  `Stack create(int size);`
- When `create` is called, it will create a `stack_type` structure plus an array of length `size`.
- The `contents` member of the structure will point to this array.

Implementing the Stack ADT Using a Dynamic Array

- `stackADT.h` will be the same as before, except that `create` will have a `size` parameter.
- The new version will be named `stackADT2.h`.
- `stackADT.c` will need more extensive modification, yielding `stackADT2.c`.

stackADT2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT2.h"

struct stack_type {
  Item *contents;
  int top;
  int size;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```



```
Stack create(int size)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->contents = malloc(size * sizeof(Item));
  if (s->contents == NULL) {
    free(s);
    terminate("Error in create: stack could not be created.");
  }
  s->top = 0;
  s->size = size;
  return s;
}

void destroy(Stack s)
{
  free(s->contents);
  free(s);
}
```

```
void make_empty(Stack s)
{
  s->top = 0;
}

bool is_empty(Stack s)
{
  return s->top == 0;
}

bool is_full(Stack s)
{
  return s->top == s->size;
}
```



```
void push(Stack s, Item i)
{
  if (is_full(s))
    terminate("Error in push: stack is full.");
  s->contents[s->top++] = i;
}

Item pop(Stack s)
{
  if (is_empty(s))
    terminate("Error in pop: stack is empty.");
  return s->contents[--s->top];
}
```


Implementing the Stack ADT Using a Dynamic Array

- The `stackclient.c` file can again be used to test the stack ADT.
- The calls of `create` will need to be changed, since `create` now requires an argument.
- Example:

```
s1 = create(100);
s2 = create(200);
```


Implementing the Stack ADT Using a Linked List

- Implementing the stack ADT using a dynamically allocated array provides more flexibility than using a fixed-size array.
- However, the client is still required to specify a maximum size for a stack at the time it's created.
- With a linked-list implementation, there won't be any preset limit on the size of a stack.

Implementing the Stack ADT Using a Linked List

- The linked list will consist of nodes, represented by the following structure:

```
struct node {
  Item data;
  struct node *next;
};
```

- The `stack_type` structure will contain a pointer to the first node in the list:

```
struct stack_type {
  struct node *top;
};
```


Implementing the Stack ADT Using a Linked List

- The `stack_type` structure seems superfluous, since `Stack` could be defined to be `struct node *`.
- However, `stack_type` is needed so that the interface to the stack remains unchanged.
- Moreover, having the `stack_type` structure will make it easier to change the implementation in the future.

Implementing the Stack ADT Using a Linked List

- Implementing the stack ADT using a linked list involves modifying the `stackADT.c` file to create a new version named `stackADT3.c`.
- The `stackADT.h` header is unchanged.
- The original `stackclient.c` file can be used for testing.

stackADT3.c

```
#include <stdio.h>
#include <stdlib.h>
#include "stackADT.h"

struct node {
  Item data;
  struct node *next;
};

struct stack_type {
  struct node *top;
};

static void terminate(const char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
Stack create(void)
{
  Stack s = malloc(sizeof(struct stack_type));
  if (s == NULL)
    terminate("Error in create: stack could not be created.");
  s->top = NULL;
  return s;
}

void destroy(Stack s)
{
  make_empty(s);
  free(s);
}

void make_empty(Stack s)
{
  while (!is_empty(s))
    pop(s);
}
```



```
bool is_empty(Stack s)
{
  return s->top == NULL;
}

bool is_full(Stack s)
{
  return false;
}

void push(Stack s, Item i)
{
  struct node *new_node = malloc(sizeof(struct node));
  if (new_node == NULL)
    terminate("Error in push: stack is full.");

  new_node->data = i;
  new_node->next = s->top;
  s->top = new_node;
}
```



```
Item pop(Stack s)
{
  struct node *old_top;
  Item i;

  if (is_empty(s))
    terminate("Error in pop: stack is empty.");

  old_top = s->top;
  i = old_top->data;
  s->top = old_top->next;
  free(old_top);
  return i;
}
```


Design Issues for Abstract Data Types

- The stack ADT suffers from several problems that prevent it from being industrial-strength.

Naming Conventions

- The stack ADT functions currently have short, easy-to-understand names, such as `create`.
- If a program has more than one ADT, name clashes are likely.
- It will probably be necessary for function names to incorporate the ADT name (`stack_create`).

Error Handling

- The stack ADT deals with errors by displaying an error message and terminating the program.
- It might be better to provide a way for a program to recover from errors rather than terminating.
- An alternative is to have the `push` and `pop` functions return a `bool` value to indicate whether or not they succeeded.

Error Handling

- The C standard library contains a parameterized macro named `assert` that can terminate a program if a specified condition isn't satisfied.
- We could use calls of this macro as replacements for the `if` statements and calls of `terminate` that currently appear in the stack ADT.

Generic ADTs

- Other problems with the stack ADT:
  - Changing the type of items stored in a stack requires modifying the definition of the `Item` type.
  - A program can't create two stacks whose items have different types.
- We'd like to have a single "generic" stack type.
- There's no completely satisfactory way to create such a type in C.

Generic ADTs

- The most common approach uses `void *` as the item type:

```
void push(Stack s, void *p);
void *pop(Stack s);
```

`pop` returns a null pointer if the stack is empty.

- Disadvantages of using `void *` as the item type:
  - Doesn't work for data that can't be represented in pointer form, including basic types such as `int` and `double`.
  - Error checking is no longer possible, because stack items can be a mixture of pointers of different types.

ADTs in Newer Languages

- These problems are dealt with much more cleanly in newer C-based languages.
  - Name clashes are prevented by defining function names within a ***class.***
  - ***Exception handling*** allows functions to "throw" an exception when they detect an error condition.
  - Some languages provide special features for defining generic ADTs. (C++ ***templates*** are an example.)

Chapter 20

Low-Level Programming

Introduction

- Previous chapters have described C's high-level, machine-independent features.
- However, some kinds of programs need to perform operations at the bit level:
  - Systems programs (including compilers and operating systems)
  - Encryption programs
  - Graphics programs
  - Programs for which fast execution and/or efficient use of space is critical

Bitwise Operators

- C provides six ***bitwise operators,*** which operate on integer data at the bit level.
- Two of these operators perform shift operations.
- The other four perform bitwise complement, bitwise *and*, bitwise exclusive *or*, and bitwise inclusive *or* operations.

Bitwise Shift Operators

- The bitwise shift operators shift the bits in an integer to the left or right:
  - << left shift
  - >> right shift
- The operands for << and >> may be of any integer type (including `char`).
- The integer promotions are performed on both operands; the result has the type of the left operand after promotion.

Bitwise Shift Operators

- The value of `i << j` is the result when the bits in `i` are shifted left by `j` places.
  - For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right.
- The value of `i >> j` is the result when `i` is shifted right by `j` places.
  - If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed.
  - If `i` is negative, the result is implementation-defined.

Bitwise Shift Operators

- Examples illustrating the effect of applying the shift operators to the number 13:

```
unsigned short i, j;

i = 13;
  /* i is now 13 (binary 0000000000001101) */
j = i << 2;
  /* j is now 52 (binary 0000000000110100) */
j = i >> 2;
  /* j is now  3 (binary 0000000000000011) */
```


Bitwise Shift Operators

- To modify a variable by shifting its bits, use the compound assignment operators <<= and >>=:

```
i = 13;
  /* i is now 13 (binary 0000000000001101) */
i <<= 2;
  /* i is now 52 (binary 0000000000110100) */
i >>= 2;
  /* i is now 13 (binary 0000000000001101) */
```


Bitwise Shift Operators

- The bitwise shift operators have lower precedence than the arithmetic operators, which can cause surprises:

  `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`

Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- There are four additional bitwise operators:
  - `~` bitwise complement
  - `&` bitwise *and*
  - `^` bitwise exclusive *or*
  - `|` bitwise inclusive *or*
- The `~` operator is unary; the integer promotions are performed on its operand.
- The other operators are binary; the usual arithmetic conversions are performed on their operands.

Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- The `~`, `&`, `^`, and `|` operators perform Boolean operations on all bits in their operands.
- The `^` operator produces 0 whenever both operands have a 1 bit, whereas `|` produces 1.

Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- Examples of the `~`, `&`, `^`, and `|` operators:

```
unsigned short i, j, k;

i = 21;
  /* i is now    21 (binary 0000000000010101) */
j = 56;
  /* j is now    56 (binary 0000000000111000) */
k = ~i;
  /* k is now 65514 (binary 1111111111101010) */
k = i & j;
  /* k is now    16 (binary 0000000000010000) */
k = i ^ j;
  /* k is now    45 (binary 0000000000101101) */
k = i | j;
  /* k is now    61 (binary 0000000000111101) */
```


Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- The `~` operator can be used to help make low-level programs more portable.
  - An integer whose bits are all 1: `~0`
  - An integer whose bits are all 1 except for the last five: `~0x1f`

Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- Each of the ~, &, ^, and | operators has a different precedence:

  Highest: ~
  &
  ^
  Lowest: |

- Examples:

  `i & ~j | k` means `(i & (~j)) | k`

  `i ^ j & ~k` means `i ^ (j & (~k))`

- Using parentheses helps avoid confusion.

Bitwise Complement, *And,* Exclusive *Or,* and Inclusive *Or*

- The compound assignment operators &=, ^=, and |= correspond to the bitwise operators &, ^, and |:

```
i = 21;
  /* i is now 21 (binary 0000000000010101) */
j = 56;
  /* j is now 56 (binary 0000000000111000) */
i &= j;
  /* i is now 16 (binary 0000000000010000) */
i ^= j;
  /* i is now 40 (binary 0000000000101000) */
i |= j;
  /* i is now 56 (binary 0000000000111000) */
```


Using the Bitwise Operators to Access Bits

- The bitwise operators can be used to extract or modify data stored in a small number of bits.
- Common single-bit operations:
  - Setting a bit
  - Clearing a bit
  - Testing a bit
- Assumptions:
  - `i` is a 16-bit `unsigned short` variable.
  - The leftmost—or ***most significant***—bit is numbered 15 and the least significant is numbered 0.

Using the Bitwise Operators to Access Bits

- ***Setting a bit.*** The easiest way to set bit 4 of `i` is to *or* the value of `i` with the constant `0x0010`:

```
i = 0x0000;
  /* i is now 0000000000000000 */
i |= 0x0010;
  /* i is now 0000000000010000 */
```

- If the position of the bit is stored in the variable `j`, a shift operator can be used to create the mask:

```
i |= 1 << j;          /* sets bit j */
```

- Example: If `j` has the value 3, then `1 << j` is `0x0008`.

Using the Bitwise Operators to Access Bits

- ***Clearing a bit.*** Clearing bit 4 of `i` requires a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;
  /* i is now 0000000011111111 */
i &= ~0x0010;
  /* i is now 0000000011101111 */
```

- A statement that clears a bit whose position is stored in a variable:

```
i &= ~(1 << j);      /* clears bit j */
```


Using the Bitwise Operators to Access Bits

- ***Testing a bit.*** An `if` statement that tests whether bit 4 of `i` is set:

```
if (i & 0x0010) …   /* tests bit 4 */
```

- A statement that tests whether bit `j` is set:

```
if (i & 1 << j) …   /* tests bit j */
```


Using the Bitwise Operators to Access Bits

- Working with bits is easier if they are given names.
- Suppose that bits 0, 1, and 2 of a number correspond to the colors blue, green, and red, respectively.
- Names that represent the three bit positions:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```


Using the Bitwise Operators to Access Bits

- Examples of setting, clearing, and testing the `BLUE` bit:

```
i |= BLUE;          /* sets BLUE bit   */
i &= ~BLUE;         /* clears BLUE bit */
if (i & BLUE) …     /* tests BLUE bit  */
```

Using the Bitwise Operators to Access Bits

- It's also easy to set, clear, or test several bits at time:

```
i |= BLUE | GREEN;
  /* sets BLUE and GREEN bits   */

i &= ~(BLUE | GREEN);
  /* clears BLUE and GREEN bits */

if (i & (BLUE | GREEN)) …
  /* tests BLUE and GREEN bits  */
```

- The `if` statement tests whether either the `BLUE` bit or the `GREEN` bit is set.

Using the Bitwise Operators to Access Bit-Fields

- ***Modifying a bit-field.*** Modifying a bit-field requires two operations:
  - A bitwise *and* (to clear the bit-field)
  - A bitwise *or* (to store new bits in the bit-field)
- Example:

```
i = i & ~0x0070 | 0x0050;
  /* stores 101 in bits 4-6 */
```

- The `&` operator clears bits 4–6 of `i`; the `|` operator then sets bits 6 and 4.

Using the Bitwise Operators to Access Bit-Fields

- Dealing with a group of several consecutive bits (a ***bit-field***) is slightly more complicated than working with single bits.
- Common bit-field operations:
  - Modifying a bit-field
  - Retrieving a bit-field

Using the Bitwise Operators to Access Bit-Fields

- To generalize the example, assume that `j` contains the value to be stored in bits 4–6 of `i`.
- `j` will need to be shifted into position before the bitwise *or* is performed:

```
i = (i & ~0x0070) | (j << 4);
  /* stores j in bits 4-6 */
```

- The `|` operator has lower precedence than `&` and <<, so the parentheses can be dropped:

```
i = i & ~0x0070 | j << 4;
```

Using the Bitwise Operators to Access Bit-Fields

- ***Retrieving a bit-field.*** Fetching a bit-field at the right end of a number (in the least significant bits) is easy:

```
j = i & 0x0007;
  /* retrieves bits 0-2 */
```

- If the bit-field isn't at the right end of `i`, we can first shift the bit-field to the end before extracting the field using the `&` operator:

```
j = (i >> 4) & 0x0007;
  /* retrieves bits 4-6 */
```


Program: XOR Encryption

- One of the simplest ways to encrypt data is to exclusive-*or* (XOR) each character with a secret key.
- Suppose that the key is the `&` character.
- XORing this key with the character `z` yields the `\` character:

```
      00100110  (ASCII code for &)
XOR   01111010  (ASCII code for z)
      01011100  (ASCII code for \)
```


Program: XOR Encryption

- Decrypting a message is done by applying the same algorithm:

```
      00100110  (ASCII code for &)
XOR   01011100  (ASCII code for \)
      01111010  (ASCII code for z)
```


Program: XOR Encryption

- The `xor.c` program encrypts a message by XORing each character with the `&` character.
- The original message can be entered by the user or read from a file using input redirection.
- The encrypted message can be viewed on the screen or saved in a file using output redirection.

Program: XOR Encryption

- A sample file named `msg`:

```
Trust not him with your secrets, who, when left
alone in your room, turns over your papers.
            --Johann Kaspar Lavater (1741-1801)
```

- A command that encrypts `msg`, saving the encrypted message in `newmsg`:

```
xor <msg >newmsg
```

- Contents of `newmsg`:

```
rTSUR HIR NOK QORN _IST UCETCRU, QNI, QNCH JC@R
GJIHC OH _IST TIIK, RSTHU IPCT _IST VGVCTU.
            --lINGHH mGUVGT jGPGRCT (1741-1801)
```


Program: XOR Encryption

- A command that recovers the original message and displays it on the screen:

```
xor <newmsg
```


Program: XOR Encryption

- The `xor.c` program won't change some characters, including digits.
- XORing these characters with `&` would produce invisible control characters, which could cause problems with some operating systems.
- The program checks whether both the original character and the new (encrypted) character are printing characters.
- If not, the program will write the original character instead of the new character.

xor.c

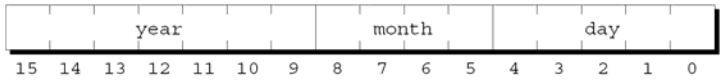
```
/* Performs XOR encryption */

#include <ctype.h>
#include <stdio.h>

#define KEY '&'

int main(void)
{
  int orig_char, new_char;

  while ((orig_char = getchar()) != EOF) {
    new_char = orig_char ^ KEY;
    if (isprint(orig_char) && isprint(new_char))
      putchar(new_char);
    else
      putchar(orig_char);
  }

  return 0;
}
```

Bit-Fields in Structures

- The bit-field techniques discussed previously can be tricky to use and potentially confusing.
- Fortunately, C provides an alternative: declaring structures whose members represent bit-fields.

Bit-Fields in Structures

- Example: How DOS stores the date at which a file was created or last modified.
- Since days, months, and years are small numbers, storing them as normal integers would waste space.
- Instead, DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



Bit-Fields in Structures

- A C structure that uses bit-fields to create an identical layout:

```
struct file_date {
  unsigned int day: 5;
  unsigned int month: 4;
  unsigned int year: 7;
};
```

- A condensed version:

```
struct file_date {
  unsigned int day: 5, month: 4, year: 7;
};
```


Bit-Fields in Structures

- The type of a bit-field must be either `int`, `unsigned int`, or `signed int`.
- Using `int` is ambiguous; some compilers treat the field's high-order bit as a sign bit, but others don't.
- In C99, bit-fields may also have type `_Bool`.
- C99 compilers may allow additional bit-field types.

Bit-Fields in Structures

- A bit-field can be used in the same way as any other member of a structure:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;      /* represents 1988 */
```

- Appearance of the `fd` variable after these assignments:

Bit-Fields in Structures

- The address operator (`&`) can't be applied to a bit-field.
- Because of this rule, functions such as `scanf` can't store data directly in a bit-field:

```
scanf("%d", &fd.day);   /*** WRONG ***/
```

- We can still use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

How Bit-Fields Are Stored

- The C standard allows the compiler considerable latitude in choosing how it stores bit-fields.
- The rules for handling bit-fields depend on the notion of "storage units."
- The size of a storage unit is implementation-defined.
  - Typical values are 8 bits, 16 bits, and 32 bits.

How Bit-Fields Are Stored

- The compiler packs bit-fields one by one into a storage unit, with no gaps between the fields, until there's not enough room for the next field.
- At that point, some compilers skip to the beginning of the next storage unit, while others split the bit-field across the storage units.
- The order in which bit-fields are allocated (left to right or right to left) is also implementation-defined.

How Bit-Fields Are Stored

- Assumptions in the `file_date` example:
  - Storage units are 16 bits long.
  - Bit-fields are allocated from right to left (the first bit-field occupies the low-order bits).
- An 8-bit storage unit is also acceptable if the compiler splits the `month` field across two storage units.

How Bit-Fields Are Stored

- The name of a bit-field can be omitted.
- Unnamed bit-fields are useful as "padding" to ensure that other bit-fields are properly positioned.
- A structure that stores the time associated with a DOS file:

```
struct file_time {
  unsigned int seconds: 5;
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```


How Bit-Fields Are Stored

- The same structure with the name of the `seconds` field omitted:

```
struct file_time {
  unsigned int : 5;       /* not used */
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

- The remaining bit-fields will be aligned as if `seconds` were still present.

How Bit-Fields Are Stored

- The length of an unnamed bit-field can be 0:

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;     /* 0-length bit-field */
  unsigned int b: 8;
};
```

- A 0-length bit-field tells the compiler to align the following bit-field at the beginning of a storage unit.
  - If storage units are 8 bits long, the compiler will allocate 4 bits for `a`, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`.
  - If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

Other Low-Level Techniques

- Some features covered in previous chapters are used often in low-level programming.
- Examples:
  - Defining types that represent units of storage
  - Using unions to bypass normal type-checking
  - Using pointers as addresses
- The `volatile` type qualifier was mentioned in Chapter 18 but not discussed because of its low-level nature.

Defining Machine-Dependent Types

- The `char` type occupies one byte, so characters can be treated as bytes.
- It's a good idea to define a `BYTE` type:

```
typedef unsigned char BYTE;
```

- Depending on the machine, additional types may be needed.
- A useful type for the x86 platform:

```
typedef unsigned short WORD;
```


Using Unions to Provide Multiple Views of Data

- Unions can be used in a portable way, as shown in Chapter 16.
- However, they're often used in C for an entirely different purpose: viewing a block of memory in two or more different ways.
- Consider the `file_date` structure described earlier.
- A `file_date` structure fits into two bytes, so any two-byte value can be thought of as a `file_date` structure.

Using Unions to Provide Multiple Views of Data

- In particular, an `unsigned short` value can be viewed as a `file_date` structure.
- A union that can be used to convert a short integer to a file date or vice versa:

```
union int_date {
  unsigned short i;
  struct file_date fd;
};
```

Using Unions to Provide Multiple Views of Data

- A function that prints an `unsigned short` argument as a file date:

```
void print_date(unsigned short n)
{
  union int_date u;

  u.i = n;
  printf("%d/%d/%d\n", u.fd.month,
         u.fd.day, u.fd.year + 1980);
}
```


Using Unions to Provide Multiple Views of Data

- Using unions to allow multiple views of data is especially useful when working with registers, which are often divided into smaller units.
- x86 processors have 16-bit registers named AX, BX, CX, and DX.
- Each register can be treated as two 8-bit registers.
  - AX is divided into registers named AH and AL.

Using Unions to Provide Multiple Views of Data

- Writing low-level applications for x86-based computers may require variables that represent AX, BX, CX, and DX.
- The goal is to access both the 16- and 8-bit registers, taking their relationships into account.
  - A change to AX affects both AH and AL; changing AH or AL modifies AX.
- The solution is to set up two structures:
  - The members of one correspond to the 16-bit registers.
  - The members of the other match the 8-bit registers.

Using Unions to Provide Multiple Views of Data

- A union that encloses the two structures:

```
union {
  struct {
    WORD ax, bx, cx, dx;
  } word;
  struct {
    BYTE al, ah, bl, bh, cl, ch, dl, dh;
  } byte;
} regs;
```

Using Unions to Provide Multiple Views of Data

- The members of the `word` structure will be overlaid with the members of the `byte` structure.
  - `ax` will occupy the same memory as `al` and `ah`.
- An example showing how the `regs` union might be used:

```
regs.byte.ah = 0x12;
regs.byte.al = 0x34;
printf("AX: %hx\n", regs.word.ax);
```

- Output:

```
AX: 1234
```


Using Unions to Provide Multiple Views of Data

- Note that the `byte` structure lists `al` before `ah`.
- When a data item consists of more than one byte, there are two logical ways to store it in memory:
  - ***Big-endian:*** Bytes are stored in "natural" order (the leftmost byte comes first).
  - ***Little-endian:*** Bytes are stored in reverse order (the leftmost byte comes last).
- x86 processors use little-endian order.

Using Unions to Provide Multiple Views of Data

- We don't normally need to worry about byte ordering.
- However, programs that deal with memory at a low level must be aware of the order in which bytes are stored.
- It's also relevant when working with files that contain non-character data.

Using Pointers as Addresses

- An address often has the same number of bits as an integer (or long integer).
- Creating a pointer that represents a specific address is done by casting an integer to a pointer:

```
BYTE *p;

p = (BYTE *) 0x1000;
  /* p contains address 0x1000 */
```

Program: Viewing Memory Locations

- The `viewmemory.c` program allows the user to view segments of computer memory.
- The program first displays the address of its own `main` function as well as the address of one of its variables.
- The program next prompts the user to enter an address (as a hexadecimal integer) plus the number of bytes to view.
- The program then displays a block of bytes of the chosen length, starting at the specified address.

Program: Viewing Memory Locations

- Bytes are displayed in groups of 10 (except for the last group).
- Bytes are shown both as hexadecimal numbers and as characters.
- Only printing characters are displayed; other characters are shown as periods.
- The program assumes that `int` values and addresses are stored using 32 bits.
- Addresses are displayed in hexadecimal.

viewmemory.c

```
/* Allows the user to view regions of computer memory */

#include <ctype.h>
#include <stdio.h>

typedef unsigned char BYTE;

int main(void)
{
  unsigned int addr;
  int i, n;
  BYTE *ptr;

  printf("Address of main function: %x\n", (unsigned int) main);
  printf("Address of addr variable: %x\n", (unsigned int) &addr);
```



```
  printf("\nEnter a (hex) address: ");
  scanf("%x", &addr);
  printf("Enter number of bytes to view: ");
  scanf("%d", &n);

  printf("\n");
  printf(" Address             Bytes             Characters\n");
  printf(" ------- ------------------------------ ----------\n");
```

```
  ptr = (BYTE *) addr;
  for (; n > 0; n -= 10) {
    printf("%8X  ", (unsigned int) ptr);
    for (i = 0; i < 10 && i < n; i++)
      printf("%.2X ", *(ptr + i));
    for (; i < 10; i++)
      printf("   ");
    printf(" ");
    for (i = 0;  i < 10 && i < n; i++) {
      BYTE ch = *(ptr + i);
      if (!isprint(ch))
        ch = '.';
      printf("%c", ch);
    }
    printf("\n");
    ptr += 10;
  }

  return 0;
}
```


Program: Viewing Memory Locations

- Sample output using GCC on an x86 system running Linux:

```
Address of main function: 804847c
Address of addr variable: bff41154

Enter a (hex) address: 8048000
Enter number of bytes to view: 40

 Address            Bytes              Characters
 -------  -----------------------------  ----------
 8048000  7F 45 4C 46 01 01 01 00 00 00  .ELF......
 804800A  00 00 00 00 00 00 02 00 03 00  ..........
 8048014  01 00 00 00 C0 83 04 08 34 00  ........4.
 804801E  00 00 C0 0A 00 00 00 00 00 00  ..........
```

- The 7F byte followed by the letters E, L, and F identify the format (ELF) in which the executable file was stored.

Program: Viewing Memory Locations

- A sample that displays bytes starting at the address of `addr`:

```
Address of main function: 804847c
Address of addr variable: bfec5484

Enter a (hex) address: bfec5484
Enter number of bytes to view: 64

 Address            Bytes              Characters
 -------  -----------------------------  ----------
BFEC5484  84 54 EC BF B0 54 EC BF F4 6F  .T...T...o
BFEC548E  68 00 34 55 EC BF C0 54 EC BF  h.4U...T..
BFEC5498  08 55 EC BF E3 3D 57 00 00 00  .U...=W...
BFEC54A2  00 00 A0 BC 55 00 08 55 EC BF  ....U..U..
BFEC54AC  E3 3D 57 00 01 00 00 00 34 55  .=W.....4U
BFEC54B6  EC BF 3C 55 EC BF 56 11 55 00  ..<U..V.U.
BFEC54C0  F4 6F 68 00                    .oh.
```

- When reversed, the first four bytes form the number BFEC5484, the address entered by the user.

The `volatile` Type Qualifier

- On some computers, certain memory locations are "volatile."
- The value stored at such a location can change as a program is running, even though the program itself isn't storing new values there.
- For example, some memory locations might hold data coming directly from input devices.

The `volatile` Type Qualifier

- The `volatile` type qualifier allows us to inform the compiler if any of the data used in a program is volatile.
- `volatile` typically appears in the declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p;
  /* p will point to a volatile byte */
```


The `volatile` Type Qualifier

- Suppose that `p` points to a memory location that contains the most recent character typed at the user's keyboard.
- A loop that obtains characters from the keyboard and stores them in a buffer array:

```
while (buffer not full) {
  wait for input;
  buffer[i] = *p;
  if (buffer[i++] == '\n')
    break;
}
```


The `volatile` Type Qualifier

- A sophisticated compiler might notice that this loop changes neither `p` nor `*p`.
- It could optimize the program by altering it so that `*p` is fetched just once:

```
store *p in a register;
while (buffer not full) {
  wait for input;
  buffer[i] = value stored in register;
  if (buffer[i++] == '\n')
    break;
}
```


The `volatile` Type Qualifier

- The optimized program will fill the buffer with many copies of the same character.
- Declaring that `p` points to volatile data avoids this problem by telling the compiler that `*p` must be fetched from memory each time it's needed.

Chapter 21

The Standard Library

Using the Library

- The C89 standard library is divided into 15 parts, with each part described by a header.
- C99 has an additional nine headers.

`<assert.h>`	`<inttypes.h>`†	`<signal.h>`	`<stdlib.h>`
`<complex.h>`†	`<iso646.h>`†	`<stdarg.h>`	`<string.h>`
`<ctype.h>`	`<limits.h>`	`<stdbool.h>`†	`<tgmath.h>`†
`<errno.h>`	`<locale.h>`	`<stddef.h>`	`<time.h>`
`<fenv.h>`†	`<math.h>`	`<stdint.h>`†	`<wchar.h>`†
`<float.h>`	`<setjmp.h>`	`<stdio.h>`	`<wctype.h>`†

†C99 only

Using the Library

- Most compilers come with a more extensive library that has additional (nonstandard) headers.
- Nonstandard headers often provide:
  - Functions that are specific to a particular computer or operating system
  - Functions that allow more control over the screen and keyboard
  - Support for graphics or a window-based user interface

Using the Library

- The standard headers consist primarily of function prototypes, type definitions, and macro definitions.
- When a file includes several standard headers, the order of `#include` directives doesn't matter.
- It's also legal to include a standard header more than once.

Restrictions on Names Used in the Library

- Any file that includes a standard header must obey two rules:
  - The names of macros defined in that header can't be used for any other purpose.
  - Library names with file scope (`typedef` names, in particular) can't be redefined at the file level.

Restrictions on Names Used in the Library

- These rules apply to *every* file in a program, regardless of which headers the file includes.
- Moreover, they apply not just to names that are currently used in the library, but also to names that are set aside for future use.
- For example, C reserves identifiers that begin with `str` followed by a lower-case letter.

Restrictions on Names Used in the Library

- Other restrictions are less obvious:
  - ***Identifiers that begin with an underscore followed by an upper-case letter or a second underscore*** are reserved for use within the library.
  - ***Identifiers that begin with an underscore*** are reserved for use as identifiers and tags with file scope.
  - ***Every identifier with external linkage in the standard library*** is reserved for use as an identifier with external linkage. In particular, the names of all standard library functions are reserved.

Functions Hidden by Macros

- The C standard allows headers to define macros that have the same names as library functions, but requires that a true function be available as well.
- It's not unusual for a library header to declare a function *and* define a macro with the same name.

Functions Hidden by Macros

- `getchar` is a library function declared in the `<stdio.h>` header:

```
int getchar(void);
```

- `<stdio.h>` usually defines `getchar` as a macro as well:

```
#define getchar() getc(stdin)
```

- By default, a call of `getchar` will be treated as a macro invocation.

Functions Hidden by Macros

- A macro is usually preferable to a true function, because it will probably improve the speed of a program.
- Occasionally, a genuine function is needed, perhaps to minimize the size of the executable code.

Functions Hidden by Macros

- A macro definition can be removed (thus gaining access to the true function) by using `#undef`:

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

- `#undef` has no effect when given a name that's not defined as a macro.

Functions Hidden by Macros

- Individual uses of a macro can be disabled by putting parentheses around its name:

```
ch = (getchar)();
  /* instead of ch = getchar(); */
```

- The preprocessor can't spot a parameterized macro unless its name is followed by a left parenthesis.
- However, the compiler can still recognize `getchar` as a function.

C89 Library Overview

<assert.h> ***Diagnostics***

Contains only the `assert` macro, which can be used to insert self-checks into a program. If any check fails, the program terminates.

<ctype.h> ***Character Handling***

Provides functions for classifying characters and for converting letters from lower to upper case or vice versa.

C89 Library Overview

<errno.h> ***Errors***

Provides `errno` ("error number"), an lvalue that can be tested after a call of certain library functions to see if an error occurred.

<float.h> ***Characteristics of Floating Types***

Provides macros that describe the characteristics of floating types, including their range and accuracy.

C89 Library Overview

<limits.h> ***Sizes of Integer Types***

Provides macros that describe the characteristics of integer types (including character types), including their maximum and minimum values.

<locale.h> ***Localization***

Provides functions to help a program adapt its behavior to a country or other geographic region.

C89 Library Overview

<math.h> ***Mathematics***

Provides common mathematical functions.

<setjmp.h> ***Nonlocal Jumps***

Provides the `setjmp` and `longjmp` functions. `setjmp` "marks" a place in a program; `longjmp` can then be used to return to that place later.

C89 Library Overview

<signal.h> ***Signal Handling***

Provides functions that deal with exceptional conditions (signals).

- The `signal` function installs a function to be called if a given signal should occur later.
- The `raise` function causes a signal to occur.

<stdarg.h> ***Variable Arguments***

Provides tools for writing functions that can have a variable number of arguments.

C89 Library Overview

<stddef.h> ***Common Definitions***

Provides definitions of frequently used types and macros.

<stdio.h> ***Input/Output***

Provides a large assortment of input/output functions, including operations on both sequential and random-access files.

C89 Library Overview

<stdlib.h> ***General Utilities***

Provides functions that perform the following operations:

- Converting strings to numbers
- Generating pseudo-random numbers
- Performing memory management tasks
- Communicating with the operating system
- Searching and sorting
- Performing conversions between multibyte characters and wide characters

C89 Library Overview

<string.h> ***String Handling***

Provides functions that perform string operations, as well as functions that operate on arbitrary blocks of memory.

<time.h> ***Date and Time***

Provides functions for determining the time (and date), manipulating times, and formatting times for display.

C99 Library Changes

- Some of the biggest changes in C99 affect the standard library:
  - ***Additional headers.*** The C99 standard library has nine headers that don't exist in C89.
  - ***Additional macros and functions.*** C99 adds macros and functions to several existing headers (especially `<math.h>`).
  - ***Enhanced versions of existing functions.*** Some existing functions, including `printf` and `scanf`, have additional capabilities in C99.

C99 Library Changes

`<complex.h>` ***Complex Arithmetic***

Defines the `complex` and `I` macros.

Provides functions for performing mathematical operations on complex numbers.

`<fenv.h>` ***Floating-Point Environment***

Provides access to floating-point status flags and control modes.

C99 Library Changes

`<inttypes.h>` ***Format Conversion of Integer Types***

Defines macros that can be used in format strings for input/output of the integer types declared in `<stdint.h>`.

Provides functions for working with greatest-width integers.

`<iso646.h>` ***Alternative Spellings***

Defines macros representing the operators whose symbols contain the characters `&`, `|`, `~`, `!`, and `^`.

C99 Library Changes

`<stdbool.h>` ***Boolean Type and Values***

Defines the `bool`, `true`, and `false` macros, as well as a macro that can be used to test whether these macros have been defined.

`<stdint.h>` ***Integer Types***

Declares integer types with specified widths and defines related macros.

Defines parameterized macros that construct integer constants with specific types.

C99 Library Changes

<tgmath.h> ***Type-Generic Math***

Provides "type-generic" macros that can detect argument types and substitute a call of a <math.h> or <complex.h> function.

<wchar.h> ***Extended Multibyte and Wide-Character Utilities***

Provides functions for wide-character input/output and wide string manipulation.

C99 Library Changes

<wctype.h> ***Wide-Character Classification and Mapping Utilities***

The wide-character version of <ctype.h>.

Provides functions for classifying and changing the case of wide characters.

The <stddef.h> Header: Common Definitions

- Types defined in <stddef.h>:
  - ptrdiff_t. The type of the result when two pointers are subtracted.
  - size_t. The type returned by the sizeof operator.
  - wchar_t. A type large enough to represent all possible characters in all supported locales.
- Macros defined in <stddef.h>:
  - NULL. Represents the null pointer.
  - offsetof. Computes the number of bytes between the beginning of a structure and one of its members.

The <stddef.h> Header: Common Definitions

- An example structure:

```
struct s {
  char a;
  int b[2];
  float c;
};
```

- The value of offsetof(struct s, a) must be 0, but the offsets of b and c depend on the compiler.
- One possibility is that offsetof(struct s, b) is 1, and offsetof(struct s, c) is 9.
- If a compiler should leave a three-byte hole after a, the offsets of b and c would be 4 and 12.

The <stdbool.h> Header (C99): Boolean Type and Values

- Macros defined in <stdbool.h>:

  `bool` (defined to be `_Bool`)

  `true` (defined to be `1`)

  `false` (defined to be `0`)

  `__bool_true_false_are_defined` (defined to be `1`)

- A program could use a preprocessing directive to test the last of these before attempting to define its own version of `bool`, `true`, or `false`.

Chapter 22

Input/Output

Introduction

- C's input/output library is the biggest and most important part of the standard library.
- The `<stdio.h>` header is the primary repository of input/output functions, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`.
- This chapter provides more information about these six functions.
- It also introduces many new functions, most of which deal with files.

Introduction

- Topics to be covered:
  - Streams, the `FILE` type, input and output redirection, and the difference between text files and binary files
  - Functions designed specifically for use with files, including functions that open and close files
  - Functions that perform "formatted" input/output
  - Functions that read and write unformatted data (characters, lines, and blocks)
  - Random access operations on files
  - Functions that write to a string or read from a string

Introduction

- In C99, some I/O functions belong to the `<wchar.h>` header.
- The `<wchar.h>` functions deal with wide characters rather than ordinary characters.
- Functions in `<stdio.h>` that read or write data are known as ***byte input/output functions.***
- Similar functions in `<wchar.h>` are called ***wide-character input/output functions.***

Streams

- In C, the term ***stream*** means any source of input or any destination for output.
- Many small programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen).
- Larger programs may need additional streams.
- Streams often represent files stored on various media.
- However, they could just as easily be associated with devices such as network ports and printers.

File Pointers

- Accessing a stream is done through a ***file pointer,*** which has type `FILE *`.
- The `FILE` type is declared in `<stdio.h>`.
- Certain streams are represented by file pointers with standard names.
- Additional file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```


Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

File Pointer	*Stream*	*Default Meaning*
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.

Standard Streams and Redirection

- The I/O functions discussed in previous chapters obtain input from `stdin` and send output to `stdout`.
- Many operating systems allow these default meanings to be changed via a mechanism known as ***redirection.***

Standard Streams and Redirection

- A typical technique for forcing a program to obtain its input from a file instead of from the keyboard:

  ```
  demo <in.dat
  ```

  This technique is known as ***input redirection.***
- ***Output redirection*** is similar:

  ```
  demo >out.dat
  ```

  All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen.

Standard Streams and Redirection

- Input redirection and output redirection can be combined:

  ```
  demo <in.dat >out.dat
  ```
- The < and > characters don't have to be adjacent to file names, and the order in which the redirected files are listed doesn't matter:

  ```
  demo < in.dat > out.dat
  demo >out.dat <in.dat
  ```


Standard Streams and Redirection

- One problem with output redirection is that *everything* written to `stdout` is put into a file.
- Writing error messages to `stderr` instead of `stdout` guarantees that they will appear on the screen even when `stdout` has been redirected.

Text Files versus Binary Files

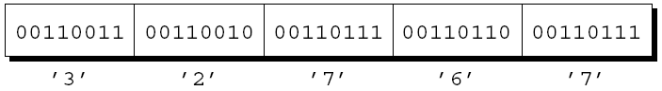
- `<stdio.h>` supports two kinds of files: text and binary.
- The bytes in a ***text file*** represent characters, allowing humans to examine or edit the file.
  - The source code for a C program is stored in a text file.
- In a ***binary file,*** bytes don't necessarily represent characters.
  - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
  - An executable C program is stored in a binary file.

Text Files versus Binary Files

- Text files have two characteristics that binary files don't possess.
- ***Text files are divided into lines.*** Each line in a text file normally ends with one or two special characters.
  - Windows: carriage-return character (`'\x0d'`) followed by line-feed character (`'\x0a'`)
  - UNIX and newer versions of Mac OS: line-feed character
  - Older versions of Mac OS: carriage-return character

Text Files versus Binary Files

- When data is written to a file, it can be stored in text form or in binary form.
- One way to store the number 32767 in a file would be to write it in text form as the characters 3, 2, 7, 6, and 7:

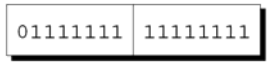


Text Files versus Binary Files

- ***Text files may contain a special "end-of-file" marker.***
  - In Windows, the marker is `'\x1a'` (Ctrl-Z), but it is not required.
  - Most other operating systems, including UNIX, have no special end-of-file character.
- In a binary file, there are no end-of-line or end-of-file markers; all bytes are treated equally.

Text Files versus Binary Files

- The other option is to store the number in binary, which would take as few as two bytes:



- Storing numbers in binary can often save space.

Text Files versus Binary Files

- Programs that read from a file or write to a file must take into account whether it's text or binary.
- A program that displays the contents of a file on the screen will probably assume it's a text file.
- A file-copying program, on the other hand, can't assume that the file to be copied is a text file.
  - If it does, binary files containing an end-of-file character won't be copied completely.
- When we can't say for sure whether a file is text or binary, it's safer to assume that it's binary.

File Operations

- Simplicity is one of the attractions of input and output redirection.
- Unfortunately, redirection is too limited for many applications.
  - When a program relies on redirection, it has no control over its files; it doesn't even know their names.
  - Redirection doesn't help if the program needs to read from two files or write to two files at the same time.
- When redirection isn't enough, we'll use the file operations that `<stdio.h>` provides.

Opening a File

- Opening a file for use as a stream requires a call of the `fopen` function.
- Prototype for `fopen`:

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

- `filename` is the name of the file to be opened.
  - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- The word `restrict` appears twice in the prototype for `fopen`.
- `restrict`, which is a C99 keyword, indicates that `filename` and `mode` should point to strings that don't share memory locations.
- The C89 prototype for `fopen` doesn't contain `restrict` but is otherwise identical.
- `restrict` has no effect on the behavior of `fopen`, so it can usually be ignored.

Opening a File

- In Windows, be careful when the file name in a call of `fopen` includes the `\` character.
- The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because `\t` is treated as a character escape.

- One way to avoid the problem is to use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```

- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```


Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

```
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- When it can't open a file, `fopen` returns a null pointer.

Modes

- Factors that determine which mode string to pass to `fopen`:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

Modes

- Mode strings for text files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists)
`"a+"`	Open for reading and writing (append if file exists)

Modes

- Mode strings for binary files:

String	*Meaning*
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

Modes

- Note that there are different mode strings for *writing* data and *appending* data.
- When data is written to a file, it normally overwrites what was previously there.
- When a file is opened for appending, data written to the file is added at the end.

Modes

- Special rules apply when a file is opened for both reading and writing.
  - Can't switch from reading to writing without first calling a file-positioning function unless the reading operation encountered the end of the file.
  - Can't switch from writing to reading without either calling `fflush` or calling a file-positioning function.

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen` or `freopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```


Closing a File

- It's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against `NULL`:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) …
```


Attaching a File to an Open Stream

- `freopen` attaches a different file to a stream that's already open.
- The most common use of `freopen` is to associate a file with one of the standard streams (`stdin`, `stdout`, or `stderr`).
- A call of `freopen` that causes a program to begin writing to the file `foo`:

```
if (freopen("foo", "w", stdout) == NULL)
{
  /* error; foo can't be opened */
}
```


Attaching a File to an Open Stream

- `freopen`'s normal return value is its third argument (a file pointer).
- If it can't open the new file, `freopen` returns a null pointer.

Attaching a File to an Open Stream

- C99 adds a new twist: if `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by the `mode` parameter.
- Implementations aren't required to support this feature.
- If they do, they may place restrictions on which mode changes are permitted.

Obtaining File Names from the Command Line

- There are several ways to supply file names to a program.
  - Building file names into the program doesn't provide much flexibility.
  - Prompting the user to enter file names can be awkward.
  - Having the program obtain file names from the command line is often the best solution.
- An example that uses the command line to supply two file names to a program named `demo`:

```
demo names.dat dates.dat
```


Obtaining File Names from the Command Line

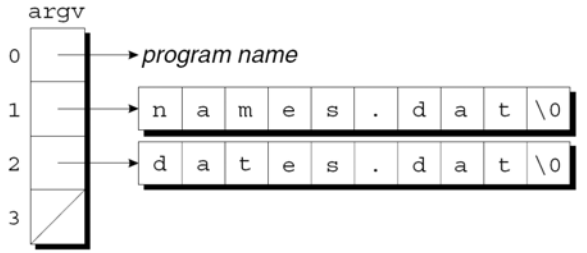
- Chapter 13 showed how to access command-line arguments by defining `main` as a function with two parameters:

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

- `argc` is the number of command-line arguments.
- `argv` is an array of pointers to the argument strings.

Obtaining File Names from the Command Line

- `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer.
- In the `demo` example, `argc` is 3 and `argv` has the following appearance:



Program: Checking Whether a File Can Be Opened

- The `canopen.c` program determines if a file exists and can be opened for reading.
- The user will give the program a file name to check:

  `canopen` *file*
- The program will then print either *file* `can be opened` or *file* `can't be opened`.
- If the user enters the wrong number of arguments on the command line, the program will print the message `usage: canopen filename`.

canopen.c

```
/* Checks whether a file can be opened for reading */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```


Temporary Files

- Programs often need to create temporary files—files that exist only as long as the program is running.
- `<stdio.h>` provides two functions, `tmpfile` and `tmpnam`, for working with temporary files.

Temporary Files

- `tmpfile` creates a temporary file (opened in `"wb+"` mode) that will exist until it's closed or the program ends.
- A call of `tmpfile` returns a file pointer that can be used to access the file later:

```
FILE *tempptr;
…
tempptr = tmpfile();
  /* creates a temporary file */
```

- If it fails to create a file, `tmpfile` returns a null pointer.

Temporary Files

- Drawbacks of using `tmpfile`:
  - Don't know the name of the file that `tmpfile` creates.
  - Can't decide later to make the file permanent.
- The alternative is to create a temporary file using `fopen`.
- The `tmpnam` function is useful for ensuring that this file doesn't have the same name as an existing file.

Temporary Files

- `tmpnam` generates a name for a temporary file.
- If its argument is a null pointer, `tmpnam` stores the file name in a static variable and returns a pointer to it:

```
char *filename;
…
filename = tmpnam(NULL);
  /* creates a temporary file name */
```


Temporary Files

- Otherwise, `tmpnam` copies the file name into a character array provided by the programmer:

```
char filename[L_tmpnam];
…
tmpnam(filename);
  /* creates a temporary file name */
```

- In this case, `tmpnam` also returns a pointer to the first character of this array.
- `L_tmpnam` is a macro in `<stdio.h>` that specifies how long to make a character array that will hold a temporary file name.

Temporary Files

- The `TMP_MAX` macro (defined in `<stdio.h>`) specifies the maximum number of temporary file names that can be generated by `tmpnam`.
- If it fails to generate a file name, `tmpnam` returns a null pointer.

File Buffering

- Transferring data to or from a disk drive is a relatively slow operation.
- The secret to achieving acceptable performance is ***buffering.***
- Data written to a stream is actually stored in a buffer area in memory; when it's full (or the stream is closed), the buffer is "flushed."
- Input streams can be buffered in a similar way: the buffer contains data from the input device; input is read from this buffer instead of the device itself.

File Buffering

- Buffering can result in enormous gains in efficiency, since reading a byte from a buffer or storing a byte in a buffer is very fast.
- It takes time to transfer the buffer contents to or from disk, but one large "block move" is much faster than many tiny byte moves.
- The functions in `<stdio.h>` perform buffering automatically when it seems advantageous.
- On rare occasions, we may need to use the functions `fflush`, `setbuf`, and `setvbuf`.

File Buffering

- By calling `fflush`, a program can flush a file's buffer as often as it wishes.
- A call that flushes the buffer for the file associated with `fp`:

```
fflush(fp);   /* flushes buffer for fp */
```

- A call that flushes *all* output streams:

```
fflush(NULL);  /* flushes all buffers */
```

- `fflush` returns zero if it's successful and `EOF` if an error occurs.

File Buffering

- `setvbuf` allows us to change the way a stream is buffered and to control the size and location of the buffer.
- The function's third argument specifies the kind of buffering desired:
  - `_IOFBF` (full buffering)
  - `_IOLBF` (line buffering)
  - `_IONBF` (no buffering)
- Full buffering is the default for streams that aren't connected to interactive devices.

File Buffering

- `setvbuf`'s second argument (if it's not a null pointer) is the address of the desired buffer.
- The buffer might have static storage duration, automatic storage duration, or even be allocated dynamically.
- `setvbuf`'s last argument is the number of bytes in the buffer.

File Buffering

- A call of `setvbuf` that changes the buffering of `stream` to full buffering, using the `N` bytes in the `buffer` array as the buffer:

```
char buffer[N];
…
setvbuf(stream, buffer, _IOFBF, N);
```

- `setvbuf` must be called after `stream` is opened but before any other operations are performed on it.

File Buffering

- It's also legal to call `setvbuf` with a null pointer as the second argument, which requests that `setvbuf` create a buffer with the specified size.
- `setvbuf` returns zero if it's successful.
- It returns a nonzero value if the `mode` argument is invalid or the request can't be honored.

File Buffering

- `setbuf` is an older function that assumes default values for the buffering mode and buffer size.
- If `buf` is a null pointer, the call `setbuf(stream, buf)` is equivalent to

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

- Otherwise, it's equivalent to

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

where `BUFSIZ` is a macro defined in `<stdio.h>`.

- `setbuf` is considered to be obsolete.

Miscellaneous File Operations

- The `remove` and `rename` functions allow a program to perform basic file management operations.
- Unlike most other functions in this section, `remove` and `rename` work with file *names* instead of file *pointers*.
- Both functions return zero if they succeed and a nonzero value if they fail.

Miscellaneous File Operations

- `remove` deletes a file:

```
remove("foo");
  /* deletes the file named "foo" */
```

- If a program uses `fopen` (instead of `tmpfile`) to create a temporary file, it can use `remove` to delete the file before the program terminates.
- The effect of removing a file that's currently open is implementation-defined.

Miscellaneous File Operations

- `rename` changes the name of a file:

```
rename("foo", "bar");
  /* renames "foo" to "bar" */
```

- `rename` is handy for renaming a temporary file created using `fopen` if a program should decide to make it permanent.
  - If a file with the new name already exists, the effect is implementation-defined.
- `rename` may fail if asked to rename an open file.

Formatted I/O

- The next group of library functions use format strings to control reading and writing.
- `printf` and related functions are able to convert data from numeric form to character form during output.
- `scanf` and related functions are able to convert data from character form to numeric form during input.

The ...printf Functions

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.
- The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments:

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

- Both functions return the number of characters written; a negative return value indicates that an error occurred.

The ...printf Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument.

The ...printf Functions

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

The ...printf Functions

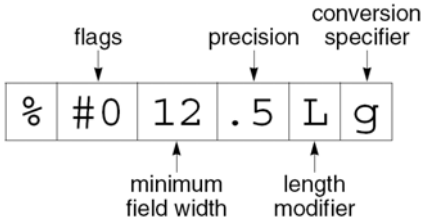
- Two other functions in `<stdio.h>` can write formatted output to a stream.
- These functions, named `vfprintf` and `vprintf`, are fairly obscure.
- Both rely on the `va_list` type, which is declared in `<stdarg.h>`, so they're discussed along with that header.

...printf Conversion Specifications

- Both `printf` and `fprintf` require a format string containing ordinary characters and/or conversion specifications.
  - Ordinary characters are printed as is.
  - Conversion specifications describe how the remaining arguments are to be converted to character form for display.

...printf Conversion Specifications

- A ...`printf` conversion specification consists of the `%` character, followed by as many as five distinct items:



...printf Conversion Specifications

- ***Flags*** (optional; more than one permitted):

Flag	*Meaning*
-	Left-justify within field.
+	Numbers produced by signed conversions always begin with + or -.
space	Nonnegative numbers produced by signed conversions are preceded by a space.
#	Octal numbers begin with `0`, nonzero hexadecimal numbers with `0x` or `0X`. Floating-point numbers always have a decimal point. Trailing zeros aren't removed from numbers printed with the `g` or `G` conversions.
`0` *(zero)*	Numbers are padded with leading zeros up to the field width.

...printf Conversion Specifications

- ***Minimum field width*** (optional). An item that's too small to occupy the field will be padded.
  - By default, spaces are added to the left of the item.
- An item that's too large for the field width will still be displayed in its entirety.
- The field width is either an integer or the character `*`.
  - If `*` is present, the field width is obtained from the next argument.

...printf Conversion Specifications

- ***Precision*** (optional). The meaning of the precision depends on the conversion:
  - `d`, `i`, `o`, `u`, `x`, `X`: minimum number of digits (leading zeros are added if the number has fewer digits)
  - `a`, `A`, `e`, `E`, `f`, `F`: number of digits after the decimal point
  - `g`, `G`: number of significant digits
  - `s`: maximum number of bytes
- The precision is a period (`.`) followed by an integer or the character `*`.
  - If `*` is present, the precision is obtained from the next argument.

...printf Conversion Specifications

- ***Length modifier*** (optional). Indicates that the item to be displayed has a type that's longer or shorter than normal.
  - `%d` normally refers to an `int` value; `%hd` is used to display a `short int` and `%ld` is used to display a `long int`.

...printf Conversion Specifications

Length Modifier	*Conversion Specifiers*	*Meaning*
`hh`†	`d`, `i`, `o`, `u`, `x`, `X`	`signed char`, `unsigned char`
	`n`	`signed char *`
`h`	`d`, `i`, `o`, `u`, `x`, `X`	`short int`, `unsigned short int`
	`n`	`short int *`
`l` *(ell)*	`d`, `i`, `o`, `u`, `x`, `X`	`long int`, `unsigned long int`
	`n`	`long int *`
	`c`	`wint_t`
	`s`	`wchar_t *`
	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	no effect

†C99 only

...printf Conversion Specifications

Length Modifier	*Conversion Specifiers*	*Meaning*
`ll`† *(ell-ell)*	`d`, `i`, `o`, `u`, `x`, `X`	`long long int`, `unsigned long long int`
	`n`	`long long int *`
`j`†	`d`, `i`, `o`, `u`, `x`, `X`	`intmax_t`, `uintmax_t`
	`n`	`intmax_t *`
`z`†	`d`, `i`, `o`, `u`, `x`, `X`	`size_t`
	`n`	`size_t *`
`t`†	`d`, `i`, `o`, `u`, `x`, `X`	`ptrdiff_t`
	`n`	`ptrdiff_t *`
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	`long double`

†C99 only

...printf Conversion Specifications

- ***Conversion specifier.*** Must be one of the characters in the following table.

Conversion Specifier	*Meaning*
`d`, `i`	Converts an `int` value to decimal form.
`o`, `u`, `x`, `X`	Converts an `unsigned int` value to base 8 (`o`), base 10 (`u`), or base 16 (`x`, `X`). `x` displays the hexadecimal digits `a`–`f` in lower case; `X` displays them in upper case.
`f`, `F`†	Converts a `double` value to decimal form, putting the decimal point in the correct position. If no precision is specified, displays six digits after the decimal point.

†C99 only

...printf Conversion Specifications

Conversion Specifier	*Meaning*
`e`, `E`	Converts a `double` value to scientific notation. If no precision is specified, displays six digits after the decimal point. If `e` is chosen, the exponent is preceded by the letter `e`; if `E` is chosen, the exponent is preceded by `E`.
`g`, `G`	`g` converts a `double` value to either `f` form or `e` form. `G` chooses between `F` and `E` forms.
`a`†, `A`†	Converts a `double` value to hexadecimal scientific notation using the form `[-]0x`*h*`.`*hhhh*`p±`*d*. `a` displays the hex digits `a`–`f` in lower case; `A` displays them in upper case. The choice of `a` or `A` also affects the case of the letters `x` and `p`.

†C99 only

...printf Conversion Specifications

Conversion Specifier	*Meaning*
`c`	Displays an `int` value as an unsigned character.
`s`	Writes the characters pointed to by the argument. Stops writing when the number of bytes specified by the precision (if present) is reached or a null character is encountered.
`p`	Converts a `void *` value to printable form.
`n`	The corresponding argument must point to an object of type `int`. Stores in this object the number of characters written so far by this call of `...printf`; produces no output.
`%`	Writes the character `%`.

C99 Changes to ...printf Conversion Specifications

- C99 changes to the conversion specifications for `printf` and `fprintf`:
  - Additional length modifiers
  - Additional conversion specifiers
  - Ability to write infinity and NaN
  - Support for wide characters
  - Previously undefined conversion specifications now allowed

Examples of ...printf Conversion Specifications

- Examples showing the effect of flags on the %d conversion:

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
%8d	•••••123	••••-123
%-8d	123•••••	-123••••
%+8d	••••+123	••••-123
% 8d	•••••123	••••-123
%08d	00000123	-0000123
%-+8d	+123••••	-123••••
%- 8d	•123••••	-123••••
%+08d	+0000123	-0000123
% 08d	•0000123	-0000123

Examples of ...printf Conversion Specifications

- Examples showing the effect of the # flag on the o, x, X, g, and G conversions:

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to 123.0*
%8o	•••••173	
%#8o	••••0173	
%8x	••••••7b	
%#8x	••••0x7b	
%8X	••••••7B	
%#8X	••••0X7B	
%8g		•••••123
%#8g		•123.000
%8G		•••••123
%#8G		•123.000

Examples of ...printf Conversion Specifications

- Examples showing the effect of the minimum field width and precision on the %s conversion:

Conversion Specification	*Result of Applying Conversion to* **"bogus"**	*Result of Applying Conversion to* **"buzzword"**
%6s	•bogus	buzzword
%-6s	bogus•	buzzword
%.4s	bogu	buzz
%6.4s	••bogu	••buzz
%-6.4s	bogu••	buzz••

Examples of ...printf Conversion Specifications

- Examples showing how the %g conversion displays some numbers in %e form and others in %f form:

Number	*Result of Applying* **%.4g** *Conversion to Number*
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

Examples of ...printf Conversion Specifications

- The minimum field width and precision are usually embedded in the format string.
- Putting the * character where either number would normally go allows us to specify it as an argument *after* the format string.
- Calls of printf that produce the same output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```


Examples of ...printf Conversion Specifications

- A major advantage of * is that it allows us to use a macro to specify the width or precision:

```
printf("%*d", WIDTH, i);
```

- The width or precision can even be computed during program execution:

```
printf("%*d", page_width / num_cols, i);
```


Examples of ...printf Conversion Specifications

- The %p conversion is used to print the value of a pointer:

```
printf("%p", (void *) ptr);
  /* displays value of ptr */
```

- The pointer is likely to be shown as an octal or hexadecimal number.

Examples of ...printf Conversion Specifications

- The %n conversion is used to find out how many characters have been printed so far by a call of ...printf.
- After the following call, the value of len will be 3:

```
printf("%d%n\n", 123, &len);
```

The ...scanf Functions

- fscanf and scanf read data items from an input stream, using a format string to indicate the layout of the input.
- After the format string, any number of pointers—each pointing to an object—follow as additional arguments.
- Input items are converted (according to conversion specifications in the format string) and stored in these objects.

The ...scanf Functions

- scanf always reads from stdin, whereas fscanf reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);
  /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
  /* reads from fp */
```

- A call of scanf is equivalent to a call of fscanf with stdin as the first argument.

The ...scanf Functions

- Errors that cause the ...scanf functions to return prematurely:
  - ***Input failure*** (no more input characters could be read)
  - ***Matching failure*** (the input characters didn't match the format string)
- In C99, an input failure can also occur because of an ***encoding error.***

The ...scanf Functions

- The ...scanf functions return the number of data items that were read and assigned to objects.
- They return EOF if an input failure occurs before any data items can be read.
- Loops that test scanf's return value are common.
- A loop that reads a series of integers one by one, stopping at the first sign of trouble:

```
while (scanf("%d", &i) == 1) {
  ...
}
```

...scanf Format Strings

- Calls of the ...scanf functions resemble those of the ...printf functions.
- However, the ...scanf functions work differently.
- The format string represents a pattern that a ...scanf function attempts to match as it reads input.
  - If the input doesn't match the format string, the function returns.
  - The input character that didn't match is "pushed back" to be read in the future.

...scanf Format Strings

- A ...scanf format string may contain three things:
  - Conversion specifications
  - White-space characters
  - Non-white-space characters

...scanf Format Strings

- ***Conversion specifications.*** Conversion specifications in a ...scanf format string resemble those in a ...printf format string.
- Most conversion specifications skip white-space characters at the beginning of an input item (the exceptions are %[, %c, and %n).
- Conversion specifications never skip *trailing* white-space characters, however.

...scanf Format Strings

- ***White-space characters.*** One or more white-space characters in a format string match zero or more white-space characters in the input stream.
- ***Non-white-space characters.*** A non-white-space character other than % matches the same character in the input stream.

...scanf Format Strings

- The format string "ISBN %d-%d-%ld-%d" specifies that the input will consist of:
  - the letters ISBN
  - possibly some white-space characters
  - an integer
  - the - character
  - an integer (possibly preceded by white-space characters)
  - the - character
  - a long integer (possibly preceded by white-space characters)
  - the - character
  - an integer (possibly preceded by white-space characters)

...scanf Conversion Specifications

- A ...scanf conversion specification consists of the character % followed by:
  - *
  - Maximum field width
  - Length modifier
  - Conversion specifier
- * (optional). Signifies ***assignment suppression:*** an input item is read but not assigned to an object.
  - Items matched using * aren't included in the count that ...scanf returns.

...scanf Conversion Specifications

- ***Maximum field width*** (optional). Limits the number of characters in an input item.
  - White-space characters skipped at the beginning of a conversion don't count.
- ***Length modifier*** (optional). Indicates that the object in which the input item will be stored has a type that's longer or shorter than normal.
- The table on the next slide lists each length modifier and the type indicated when it is combined with a conversion specifier.

...scanf Conversion Specifications

Length Modifier	*Conversion Specifiers*	*Meaning*
hh†	d, i, o, u, x, X, n	signed char *, unsigned char *
h	d, i, o, u, x, X, n	short int *, unsigned short int *
l	d, i, o, u, x, X, n	long int *, unsigned long int *
(*ell*)	a, A, e, E, f, F, g, G	double *
	c, s, or [	wchar_t *
ll†	d, i, o, u, x, X, n	long long int *,
(*ell-ell*)		unsigned long long int *
j†	d, i, o, u, x, X, n	intmax_t *, uintmax_t *
z†	d, i, o, u, x, X, n	size_t *
t†	d, i, o, u, x, X, n	ptrdiff_t *
L	a, A, e, E, f, F, g, G	long double *

†C99 only

...scanf Conversion Specifications

- ***Conversion specifier***. Must be one of the characters in the following table.

Conversion Specifier	*Meaning*
d	Matches a decimal integer; the corresponding argument is assumed to have type `int *`.
i	Matches an integer; the corresponding argument is assumed to have type `int *`. The integer is assumed to be in base 10 unless it begins with `0` (indicating octal) or with `0x` or `0X` (hexadecimal).
o	Matches an octal integer; the corresponding argument is assumed to have type `unsigned int *`.
u	Matches a decimal integer; the corresponding argument is assumed to have type `unsigned int *`.

...scanf Conversion Specifications

Conversion Specifier	*Meaning*
x, X	Matches a hexadecimal integer; the corresponding argument is assumed to have type unsigned `int *`.
a†, A†, e, E, f, F†, g, G	Matches a floating-point number; the corresponding argument is assumed to have type `float *`.
c	Matches *n* characters, where *n* is the maximum field width, or one character if no field width is specified. The corresponding argument is assumed to be a pointer to a character array (or a character object, if no field width is specified). Doesn't add a null character at the end.
s	Matches a sequence of non-white-space characters, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.

†C99 only

...scanf Conversion Specifications

Conversion Specifier	*Meaning*
[	Matches a nonempty sequence of characters from a scanset, then adds a null character at the end. The corresponding argument is assumed to be a pointer to a character array.
p	Matches a pointer value in the form that ...`printf` would have written it. The corresponding argument is assumed to be a pointer to a `void *` object.
n	The corresponding argument must point to an object of type `int`. Stores in this object the number of characters read so far by this call of ...`scanf`. No input is consumed and the return value of ...`scanf` isn't affected.
%	Matches the character `%`.

...scanf Conversion Specifications

- Numeric data items can always begin with a sign (+ or −).
- The `o`, `u`, `x`, and `X` specifiers convert the item to unsigned form, however, so they're not normally used to read negative numbers.

...scanf Conversion Specifications

- The [specifier is a more complicated (and more flexible) version of the s specifier.
- A conversion specification using [has the form %[*set*] or %[^*set*], where *set* can be any set of characters.
- %[*set*] matches any sequence of characters in *set* (the ***scanset***).
- %[^*set*] matches any sequence of characters not in *set*.
- Examples:
  %[abc] matches any string containing only a, b, and c.
  %[^abc] matches any string that doesn't contain a, b, or c.

...scanf Conversion Specifications

- Many of the ...scanf conversion specifiers are closely related to the numeric conversion functions in <stdlib.h>.
- These functions convert strings (like "-297") to their equivalent numeric values (–297).
- The d specifier, for example, looks for an optional + or − sign, followed by decimal digits; this is the same form that the strtol function requires.

...scanf Conversion Specifications

- Correspondence between ...scanf conversion specifiers and numeric conversion functions:

Conversion Specifier	*Numeric Conversion Function*
d	strtol with 10 as the base
i	strtol with 0 as the base
o	strtoul with 8 as the base
u	strtoul with 10 as the base
x, X	strtoul with 16 as the base
a, A, e, E, f, F, g, G	strtod

C99 Changes to ...scanf Conversion Specifications

- C99 changes to the conversion specifications for scanf and fscanf:
  - Additional length modifiers
  - Additional conversion specifiers
  - Ability to read infinity and NaN
  - Support for wide characters

scanf Examples

- The next three tables contain sample calls of `scanf`.
- Characters printed in ~~strikeout~~ are consumed by the call.

scanf Examples

- Examples that combine conversion specifications, white-space characters, and non-white-space characters:

scanf Call	***Input***	***Variables***
`n = scanf("%d%d", &i, &j);`	~~`12`~~`•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged
`n = scanf("%d,%d", &i, &j);`	~~`12`~~`•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged
`n = scanf("%d ,%d", &i, &j);`	~~`12•,•34`~~`¤`	`n`: 2 `i`: 12 `j`: 34
`n = scanf("%d, %d", &i, &j);`	~~`12`~~`•,•34¤`	`n`: 1 `i`: 12 `j`: unchanged

scanf Examples

- Examples showing the effect of assignment suppression and specifying a field width:

scanf Call	***Input***	***Variables***
`n = scanf("%*d%d", &i);`	~~`12•34`~~`¤`	`n`: 1 `i`: 34
`n = scanf("%*s%s", str);`	~~`My•Fair`~~`•Lady¤`	`n`: 1 `str`: `"Fair"`
`n = scanf("%1d%2d%3d", &i, &j, &k);`	~~`12345`~~`¤`	`n`: 3 `i`: 1 `j`: 23 `k`: 45
`n = scanf("%2d%2s%2d", &i, str, &j);`	~~`123456`~~`¤`	`n`: 3 `i`: 12 `str`: `"34"` `j`: 56

scanf Examples

- Examples that illustrate the `i`, `[`, and `n` conversion specifiers:

scanf Call	***Input***	***Variables***
`n = scanf("%i%i%i", &i, &j, &k);`	~~`12•012•0x12`~~`¤`	`n`: 3 `i`: 12 `j`: 10 `k`: 18
`n = scanf("%[0123456789]", str);`	~~`123`~~`abc¤`	`n`: 1 `str`: `"123"`
`n = scanf("%[0123456789]", str);`	`abc123¤`	`n`: 0 `str`: unchanged
`n = scanf("%[^0123456789]", str);`	~~`abc`~~`123¤`	`n`: 1 `str`: `"abc"`
`n = scanf("%*d%d%n", &i, &j);`	~~`10•20`~~`•30¤`	`n`: 1 `i`: 20 `j`: 5

Detecting End-of-File and Error Conditions

- If we ask a ...`scanf` function to read and store *n* data items, we expect its return value to be *n*.
- If the return value is less than *n*, something went wrong:
  - ***End-of-file.*** The function encountered end-of-file before matching the format string completely.
  - ***Read error.*** The function was unable to read characters from the stream.
  - ***Matching failure.*** A data item was in the wrong format.

Detecting End-of-File and Error Conditions

- Every stream has two indicators associated with it: an ***error indicator*** and an ***end-of-file indicator.***
- These indicators are cleared when the stream is opened.
- Encountering end-of-file sets the end-of-file indicator, and a read error sets the error indicator.
  - The error indicator is also set when a write error occurs on an output stream.
- A matching failure doesn't change either indicator.

Detecting End-of-File and Error Conditions

- Once the error or end-of-file indicator is set, it remains in that state until it's explicitly cleared, perhaps by a call of the `clearerr` function.
- `clearerr` clears both the end-of-file and error indicators:

```
clearerr(fp);
  /* clears eof and error indicators for fp */
```

- `clearerr` isn't needed often, since some of the other library functions clear one or both indicators as a side effect.

Detecting End-of-File and Error Conditions

- The `feof` and `ferror` functions can be used to test a stream's indicators to determine why a prior operation on the stream failed.
- The call `feof(fp)` returns a nonzero value if the end-of-file indicator is set for the stream associated with `fp`.
- The call `ferror(fp)` returns a nonzero value if the error indicator is set.

Detecting End-of-File and Error Conditions

- When `scanf` returns a smaller-than-expected value, `feof` and `ferror` can be used to determine the reason.
  - If `feof` returns a nonzero value, the end of the input file has been reached.
  - If `ferror` returns a nonzero value, a read error occurred during input.
  - If neither returns a nonzero value, a matching failure must have occurred.
- The return value of `scanf` indicates how many data items were read before the problem occurred.

Detecting End-of-File and Error Conditions

- The `find_int` function is an example that shows how `feof` and `ferror` might be used.
- `find_int` searches a file for a line that begins with an integer:

```
n = find_int("foo");
```

- `find_int` returns the value of the integer that it finds or an error code:
  - –1 File can't be opened
  - –2 Read error
  - –3 No line begins with an integer


```
int find_int(const char *filename)
{
  FILE *fp = fopen(filename, "r");
  int n;

  if (fp == NULL)
    return -1;                /* can't open file */

  while (fscanf(fp, "%d", &n) != 1) {
    if (ferror(fp)) {
      fclose(fp);
      return -2;              /* read error */
    }
    if (feof(fp)) {
      fclose(fp);
      return -3;              /* integer not found */
    }
    fscanf(fp, "%*[^\n]");    /* skips rest of line */
  }

  fclose(fp);
  return n;
}
```


Character I/O

- The next group of library functions can read and write single characters.
- These functions work equally well with text streams and binary streams.
- The functions treat characters as values of type `int`, not `char`.
- One reason is that the input functions indicate an end-of-file (or error) condition by returning `EOF`, which is a negative integer constant.

Output Functions

- `putchar` writes one character to the `stdout` stream:

```
putchar(ch);    /* writes ch to stdout */
```

- `fputc` and `putc` write a character to an arbitrary stream:

```
fputc(ch, fp);  /* writes ch to fp */
putc(ch, fp);   /* writes ch to fp */
```

- `putc` is usually implemented as a macro (as well as a function), while `fputc` is implemented only as a function.

Output Functions

- `putchar` itself is usually a macro:

```
#define putchar(c) putc((c), stdout)
```

- The C standard allows the `putc` macro to evaluate the stream argument more than once, which `fputc` isn't permitted to do.
- Programmers usually prefer `putc`, which gives a faster program.
- If a write error occurs, all three functions set the error indicator for the stream and return `EOF`.
- Otherwise, they return the character that was written.

Input Functions

- `getchar` reads a character from `stdin`:

```
ch = getchar();
```

- `fgetc` and `getc` read a character from an arbitrary stream:

```
ch = fgetc(fp);
ch = getc(fp);
```

- All three functions treat the character as an `unsigned char` value (which is then converted to `int` type before it's returned).
- As a result, they never return a negative value other than `EOF`.

Input Functions

- `getc` is usually implemented as a macro (as well as a function), while `fgetc` is implemented only as a function.
- `getchar` is normally a macro as well:

```
#define getchar() getc(stdin)
```

- Programmers usually prefer `getc` over `fgetc`.

Input Functions

- The `fgetc`, `getc`, and `getchar` functions behave the same if a problem occurs.
- At end-of-file, they set the stream's end-of-file indicator and return `EOF`.
- If a read error occurs, they set the stream's error indicator and return `EOF`.
- To differentiate between the two situations, we can call either `feof` or `ferror`.

Input Functions

- One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file.
- A typical `while` loop for that purpose:

```
while ((ch = getc(fp)) != EOF) {
  …
}
```

- Always store the return value in an `int` variable, not a `char` variable.
- Testing a `char` variable against `EOF` may give the wrong result.

Input Functions

- The `ungetc` function "pushes back" a character read from a stream and clears the stream's end-of-file indicator.
- A loop that reads a series of digits, stopping at the first nondigit:

```
while (isdigit(ch = getc(fp))) {
  …
}
ungetc(ch, fp);
  /* pushes back last character read */
```


Input Functions

- The number of characters that can be pushed back by consecutive calls of `ungetc` varies; only the first call is guaranteed to succeed.
- Calling a file-positioning function (`fseek`, `fsetpos`, or `rewind`) causes the pushed-back characters to be lost.
- `ungetc` returns the character it was asked to push back.
  - It returns `EOF` if an attempt is made to push back `EOF` or to push back more characters than allowed.

Program: Copying a File

- The `fcopy.c` program makes a copy of a file.
- The names of the original file and the new file will be specified on the command line when the program is executed.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

Program: Copying a File

- Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files.
- If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

fcopy.c

```
/* Copies a file */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }
```



```
  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Line I/O

- Library functions in the next group are able to read and write lines.
- These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

Output Functions

- The `puts` function writes a string of characters to `stdout`:

```
puts("Hi, there!");  /* writes to stdout */
```

- After it writes the characters in the string, `puts` always adds a new-line character.

Output Functions

- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);  /* writes to fp */
```

- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.
- Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

Input Functions

- The `gets` function reads a line of input from `stdin`:

```
gets(str); /* reads a line from stdin */
```

- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).
- `fgets` is a more general version of `gets` that can read from any stream.
- `fgets` is also safer than `gets`, since it limits the number of characters that it will store.

Input Functions

- A call of `fgets` that reads a line into a character array named `str`:

```
fgets(str, sizeof(str), fp);
```

- `fgets` will read characters until it reaches the first new-line character or `sizeof(str)` – 1 characters have been read.
- If it reads the new-line character, `fgets` stores it along with the other characters.

Input Functions

- Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters.
- Otherwise, both return their first argument, which points to the array in which the input was stored.
- Both functions store a null character at the end of the string.

Input Functions

- `fgets` should be used instead of `gets` in most situations.
- `gets` is safe to use only when the string being read is *guaranteed* to fit into the array.
- When there's no guarantee (and there usually isn't), it's much safer to use `fgets`.
- `fgets` will read from the standard input stream if passed `stdin` as its third argument:

```
fgets(str, sizeof(str), stdin);
```


Block I/O

- The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step.
- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Block I/O

- `fwrite` is designed to copy an array from memory to a stream.
- Arguments in a call of `fwrite`:
  - Address of array
  - Size of each array element (in bytes)
  - Number of elements to write
  - File pointer
- A call of `fwrite` that writes the entire contents of the array `a`:

```
fwrite(a, sizeof(a[0]),
       sizeof(a) / sizeof(a[0]), fp);
```


Block I/O

- `fwrite` returns the number of elements actually written.
- This number will be less than the third argument if a write error occurs.

Block I/O

- `fread` will read the elements of an array from a stream.
- A call of `fread` that reads the contents of a file into the array `a`:

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

- `fread`'s return value indicates the actual number of elements read.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

Block I/O

- `fwrite` is convenient for a program that needs to store data in a file before terminating.
- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't need to be in array form.
- A call of `fwrite` that writes a structure variable `s` to a file:

```
fwrite(&s, sizeof(s), 1, fp);
```

File Positioning

- Every stream has an associated ***file position.***
- When a file is opened, the file position is set at the beginning of the file.
  - In "append" mode, the initial file position may be at the beginning or end, depending on the implementation.
- When a read or write operation is performed, the file position advances automatically, providing sequential access to data.

File Positioning

- Although sequential access is fine for many applications, some programs need the ability to jump around within a file.
- If a file contains a series of records, we might want to jump directly to a particular record.
- `<stdio.h>` provides five functions that allow a program to determine the current file position or to change it.

File Positioning

- The `fseek` function changes the file position associated with the first argument (a file pointer).
- The third argument is one of three macros:

`SEEK_SET`	Beginning of file
`SEEK_CUR`	Current file position
`SEEK_END`	End of file

- The second argument, which has type `long int`, is a (possibly negative) byte count.

File Positioning

- Using `fseek` to move to the beginning of a file:

```
fseek(fp, 0L, SEEK_SET);
```

- Using `fseek` to move to the end of a file:

```
fseek(fp, 0L, SEEK_END);
```

- Using `fseek` to move back 10 bytes:

```
fseek(fp, -10L, SEEK_CUR);
```

- If an error occurs (the requested position doesn't exist, for example), `fseek` returns a nonzero value.

File Positioning

- The file-positioning functions are best used with binary streams.
- C doesn't prohibit programs from using them with text streams, but certain restrictions apply.
- For text streams, `fseek` can be used only to move to the beginning or end of a text stream or to return to a place that was visited previously.
- For binary streams, `fseek` isn't required to support calls in which the third argument is `SEEK_END`.

File Positioning

- The `ftell` function returns the current file position as a long integer.
- The value returned by `ftell` may be saved and later supplied to a call of `fseek`:

```
long file_pos;
…
file_pos = ftell(fp);
  /* saves current position */
…
fseek(fp, file_pos, SEEK_SET);
  /* returns to old position */
```


File Positioning

- If `fp` is a binary stream, the call `ftell(fp)` returns the current file position as a byte count, where zero represents the beginning of the file.
- If `fp` is a text stream, `ftell(fp)` isn't necessarily a byte count.
- As a result, it's best not to perform arithmetic on values returned by `ftell`.

File Positioning

- The `rewind` function sets the file position at the beginning.
- The call `rewind(fp)` is nearly equivalent to `fseek(fp, 0L, SEEK_SET)`.
  - The difference? `rewind` doesn't return a value but does clear the error indicator for `fp`.

File Positioning

- `fseek` and `ftell` are limited to files whose positions can be stored in a long integer.
- For working with very large files, C provides two additional functions: `fgetpos` and `fsetpos`.
- These functions can handle large files because they use values of type `fpos_t` to represent file positions.
  - An `fpos_t` value isn't necessarily an integer; it could be a structure, for instance.

File Positioning

- The call `fgetpos(fp, &file_pos)` stores the file position associated with `fp` in the `file_pos` variable.
- The call `fsetpos(fp, &file_pos)` sets the file position for `fp` to be the value stored in `file_pos`.
- If a call of `fgetpos` or `fsetpos` fails, it stores an error code in `errno`.
- Both functions return zero when they succeed and a nonzero value when they fail.

File Positioning

- An example that uses `fgetpos` and `fsetpos` to save a file position and return to it later:

```
fpos_t file_pos;
…
fgetpos(fp, &file_pos);
  /* saves current position */
…
fsetpos(fp, &file_pos);
  /* returns to old position */
```


Program: Modifying a File of Part Records

- Actions performed by the `invclear.c` program:
  - Opens a binary file containing part structures.
  - Reads the structures into an array.
  - Sets the `on_hand` member of each structure to 0.
  - Writes the structures back to the file.
- The program opens the file in `"rb+"` mode, allowing both reading and writing.

invclear.c

```
/* Modifies a file of part records by setting the quantity
   on hand to zero for all records */

#include <stdio.h>
#include <stdlib.h>

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts;
```



```
int main(void)
{
  FILE *fp;
  int i;

  if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
    fprintf(stderr, "Can't open inventory file\n");
    exit(EXIT_FAILURE);
  }

  num_parts = fread(inventory, sizeof(struct part),
                    MAX_PARTS, fp);

  for (i = 0; i < num_parts; i++)
    inventory[i].on_hand = 0;

  rewind(fp);
  fwrite(inventory, sizeof(struct part), num_parts, fp);
  fclose(fp);

  return 0;
}
```


String I/O

- The functions described in this section can read and write data using a string as though it were a stream.
- `sprintf` and `snprintf` write characters into a string.
- `sscanf` reads characters from a string.

String I/O

- Three similar functions (`vsprintf`, `vsnprintf`, and `vsscanf`) also belong to `<stdio.h>`.
- These functions rely on the `va_list` type, which is declared in `<stdarg.h>`, so they are discussed in Chapter 26.

Output Functions

- The `sprintf` function writes output into a character array (pointed to by its first argument) instead of a stream.
- A call that writes `"9/20/2010"` into `date`:

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

- `sprintf` adds a null character at the end of the string.
- It returns the number of characters stored (not counting the null character).

Output Functions

- `sprintf` can be used to format data, with the result saved in a string until it's time to produce output.
- `sprintf` is also convenient for converting numbers to character form.

Output Functions

- The `snprintf` function (new in C99) is the same as `sprintf`, except for an additional second parameter named `n`.
- No more than `n` – 1 characters will be written to the string, not counting the terminating null character, which is always written unless `n` is zero.
- Example:

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

The string `"Einstein, Al"` is written into `name`.

Output Functions

- `snprintf` returns the number of characters that would have been written (not including the null character) had there been no length restriction.
- If an encoding error occurs, `snprintf` returns a negative number.
- To see if `snprintf` had room to write all the requested characters, we can test whether its return value was nonnegative and less than `n`.

Input Functions

- The `sscanf` function is similar to `scanf` and `fscanf`.
- `sscanf` reads from a string (pointed to by its first argument) instead of reading from a stream.
- `sscanf`'s second argument is a format string identical to that used by `scanf` and `fscanf`.

Input Functions

- One advantage of using `sscanf` is that we can examine an input line as many times as needed.
- This makes it easier to recognize alternate input forms and to recover from errors.
- Consider the problem of reading a date that's written either in the form *month*/*day*/*year* or *month*-*day*-*year*:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf(str, "%d -%d -%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else
  printf("Date not in the proper form\n");
```


Input Functions

- `sscanf` is handy for extracting data from a string that was read by another input function.
- An example that uses `fgets` to obtain a line of input, then passes the line to `sscanf` for further processing:

```
fgets(str, sizeof(str), stdin);
  /* reads a line of input */
sscanf(str, "%d%d", &i, &j);
  /* extracts two integers */
```


Input Functions

- Like the `scanf` and `fscanf` functions, `sscanf` returns the number of data items successfully read and stored.
- `sscanf` returns `EOF` if it reaches the end of the string (marked by a null character) before finding the first item.