The strcpy function

- 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.
- **String copy**: 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).

- EXAMPLE:

```
. . .
strcpy(str2, "abcd");//str2 now has "abcd"
strcpy(str1, str2); // str1 now has "abcd"
```

- 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.

- 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)-1);
str1[sizeof(str1)-1] = '\0';
```

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.

strlen returns the length of a string s, not including the null character.

Examples:

```
int len;
len = strlen("abc");  // len is now 3
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).

Examples:

```
strcpy(str1, "abc");
strcat(str1, "def");//has "abcdef"

strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);//has "abcdef"

strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
//str1 has "abcdefghi", str2 has "defghi"
```

- 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 **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.

- 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 less than s2.
- 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.
  - Upper-case letters are less than lower-case letters.
  - Digits are less than letters.
  - Spaces are less than all printing characters.
- Testing whether str1 is less than str2:

```
if(strcmp(str1, str2) < 0) ;
```

- Testing whether str1 is less than or equal to str2:

```
if(strcmp(str1, str2) <= 0) ;
```

- Choose the proper operator (<, <=, >, >=, ==, !=) to test any possible relationship between str1 and str2.

  Example: How strcat can be implemented using more primitive operations.

```
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;
}
```

Another code example:

```
char *strcat(char *s1, const char *s2){
  char *p = s1;
  while(*p) p++;
  while(*p++ = *s2++)
     ;
  return s1;
}
```

Arrays of Strings

- There is more than one way to store an array of strings.
- One option: 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.

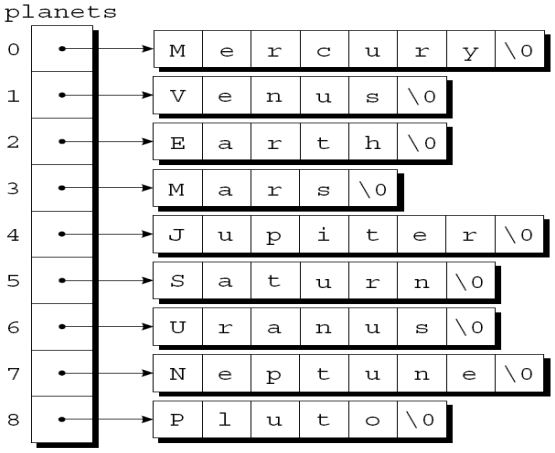
  Unfortunately, the planets array contains a fair bit of wasted space (extra null characters):

	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

- 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.
- Use an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

- This change has an effect on how planets is stored:



- 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

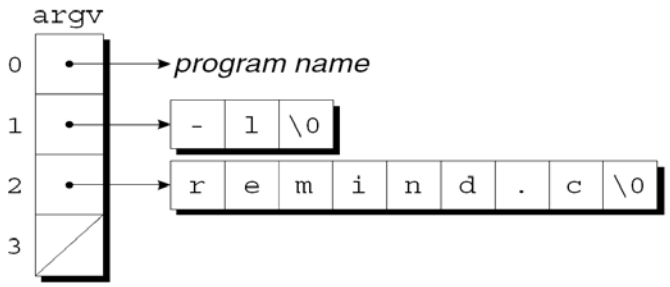
- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, main must have two parameters:

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

- Command-line arguments are called ***program parameters*** in the C standard.
  - 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.
- If the user enters the command line

```
ls -l remind.c
```

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



- 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]);
```

- 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);
```

- Example:

  The planet.c is a program 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
```

- The code:

```
#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

- Directives such as #define and #include are handled by the ***preprocessor,*** a piece of software that edits C programs just prior to compilation.

  The preprocessor looks for ***preprocessing directives,*** which begin with a # character.

- **#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.

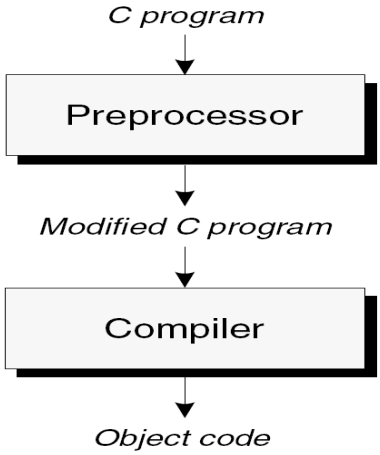
- **#include** tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.

  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 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.

- **Example:**

```
#include <stdio.h>

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

void main(void) {
  float f, c;

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

  celsius = (f-FREEZING_PT)*SCALE_FACTOR;
  printf("Celsius equivalent= %.1f\n", c);
}
```

- The program after preprocessing:

```
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line
void main(void) {
  float f, c;

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

  celsius = (f-32.0f)* (5.0f/9.0f);
  printf("Celsius equivalent= %.1f\n", c);
}
```

- The preprocessor does a bit more than just execute directives.
  -- 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.

- 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).

- 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.

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.

Rules:

- ***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
```
- ***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)
```
- ***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 */
```

Simple Macros

- Simple macros **are macros with no parameters**.

  Definition of a ***simple 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*.

- Any symbol in a macro definition will become part of the replacement list.

  For example, Putting the = symbol in a macro definition is a common error:
```
#define N = 100 /* Maybe not correct */
...
int a[N];    /* becomes int a[= 100];*/
```
- 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 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 EOS     '\0'
```
- 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 have additional uses.
  - ***Making minor changes to the syntax of C***
    Macros can serve as alternate names for C symbols:
    ```
    #define BEGIN {
    #define END   }
    ```
  - ***Renaming types***
    ```
    #define BOOL int
    ```
  - ***Controlling conditional compilation***
    ```
    #define DEBUG     1
    ```
    Later in program, you may use
    ```
    #if DEBUG
    . . . // e.g. some printf statements
    #endif
    ```
    to control the compilation.
- When macros are used as constants, C programmers customarily capitalize all letters in their names.

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, …, 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, …, x_n) as part of the replacement list.
- 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, …, 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.
- 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++;
  ```
- A more complicated function-like macro:
  ```
  #define TOUPPER(c) \
  ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
  ```
  <ctype.h> provides a similar function named toupper.
- 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.
- **Advantages:**
  - ***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.
- **Disadvantages:**
  - ***The compiled code will often be larger.***
    Each macro invocation increases the size of the source program (and hence the compiled code).
    For the macro
    ```
    #define MAX(x,y)   ((x)>(y)?(x):(y))
    ```
    The problem is compounded when macro invocations are nested:
    ```
    n = MAX(i, MAX(j, k));
    ```
    The statement after preprocessing: `n =`
  - ***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."
  - ***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.

- 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.

Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses 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.

  An example that illustrates the need to put parentheses around a macro's replacement list:

  ```
  #define TWO_PI 2*3.14159
  ```

  During preprocessing, the statement

  ```
  conversion_factor = 360/TWO_PI;
  ```

  becomes

  ```
  conversion_factor = 360/2*3.14159;
  ```

- Each occurrence of a parameter in a macro's replacement list needs parentheses. Example:

  ```
  #define SCALE(x) (x*10)
  ```

  During preprocessing, the statement

  ```
  j = SCALE(i+1);
  ```

  becomes

  ```
  j = (i+1*10);
  ```

  not equivalent to

  ```
  j = (i+1)*10;
  ```

The # Operator

- Macro definitions may contain the special operator #.

  The # operator is not recognized by the compiler; instead, it's 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."

- Example:

  ```
  #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.

- Example:

  ```
  #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;
  ```

- Another example:

  ```
  #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

- ***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.

- ***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");
```

- ***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.

- ***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.

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 want 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.

- Example: Define a macro DEBUG:

  ```
  #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
  ```

  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.

- 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 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.

- Conditional compilation has other uses besides debugging.

  Example: Writing programs that are portable to several machines or operating systems.

  ```
  #if defined(WIN32)
  …
  #elif defined(MAC_OS)
  …
  #elif defined(LINUX)
  …
  #endif
  ```