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:

  ```
  #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 uses other than debugging.

Another use of directive

- 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
  ```

The #elif and #else Directives

#if *expr1*

Lines to be included if expr1 is nonzero

#elif *expr2*

Lines to be included if expr1 is 0 but expr2 is 1

#else

Lines to be included otherwise

#endif

Any number of #elif directives—but at most one #else—may appear between #if and #endif.

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

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

The defined Operator

- 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.
- 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)
```

- *#ifndef id* or *#if !defined(id)* can be used to protecting header files against multiple inclusion.

- #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 */
```

Chp 16 Structures, Enumerations Unions, and Abstract Data Type

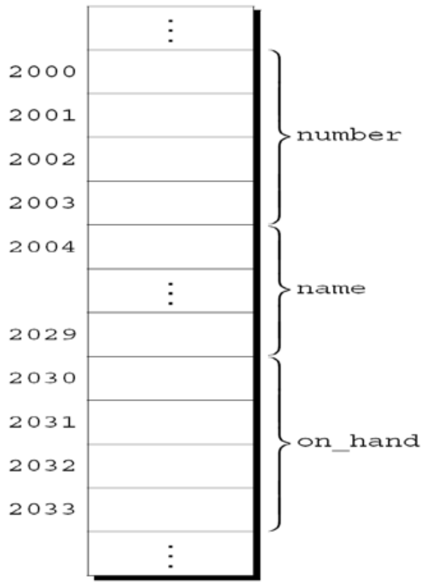
- C has two kinds of ***aggregate*** variables:
  - arrays and structures.
  - An array has same data type for all elements
  - A structure may have different data types for different elements (or members).
- The members of a structure have names; to select a particular member, we specify its name, not its position.

Declaring Structure Variables

- Example 1: 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;
```

- The members of a structure are stored in memory in the order in which they're declared.
- For the example above, assume that
  - part1 is located at address 2000.
  - Integers occupy 4 bytes.
  - NAME_LEN has the value 25.
  - There are no gaps between the members.



- Each structure represents a new scope.

  Any names declared in that scope won't conflict with other names in a program. In C, each structure has a separate *name space* for its members.

- The following declaration and the declaration in Example 1 can appear in the same program :

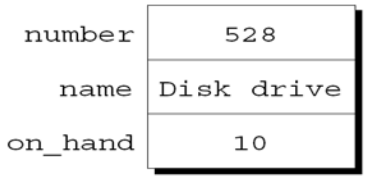
```
struct {
   char name[NAME_LEN+1];
   int number;
   char gender;
}  employee1, employee2;
```

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:



- 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)

- 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.***
- Members in a designated initializer don't have to be in the same order they are listed in the structure.
  - The programmer doesn't have to remember the order the members were originally declared.
  - The order of the members can be changed in the future without affecting designated initializers.
- 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 does not 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.
- Example: For the structure

```
struct {
   int number;
   char name[NAME_LEN+1];
   int on_hand;
} part1, part2;
```

  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);
```

- The members of a structure can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
part1.on_hand++;
```

- 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`.

- The assignment operation:

```
part2 = part1;
```

The effect of this statement is to copy part1.number into part2.number, part1.name into part2.name, and so on.

- Arrays can't be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.

- People can exploit this property by creating “dummy” structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
```

- 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 Tags

- 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”

- 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.

- 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.
- 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;
```

- All structures declared to have same struct tag are compatible with one another:

```
 struct part part1={528,"Disk drive",10};
 struct part part2;
 part2 = part1; //legal; the same type
```

Abstract Data 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_type;
```

  Part can be used in the same way as the built-in types:

```
     Part_type part1, part2;
```

- 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 defined using itself, e.g. in linked lists.

```
   struct node {
      int value;
      struct node *next;
   };
```

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values. Example:

```
void print_part(Part_type 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);`

- A function that returns a part structure:

```
Part_type build_part(int number,
                     const char *name,
                     int on_hand)
{
       Part_type 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);
```

- 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.

- 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.

Nested Arrays and Structures

- Arrays may have arrays or structures as elements, structures may have arrays and structures as members.
- In the example, te structure

```
typedef struct {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
} Name_type;
```

  has two arrays as its members. Structures of type `Name_type` can also be used by other structures as their members:

```
struct student {
   Name_type name;
   int id, age;
   char gender;
} student1, student2;
```

- Accessing student1's first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

- 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 Name_type argument instead of three arguments:

- Copying the information from a Name_type structure to the name member of a student structure would take one assignment instead of three:

```
Name_type new_name;
. . .
student1.name = new_name;
```

Arrays of Structures

- We have defined a new type

```
typedef struct {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
} Part_type;
```

- An example of an array whose elements are structure:

```
Part_type inventory[100] ;
```

- Arrays of structures can be useful for representing simple databases.
- Accessing an array element is by using subscripting:

```
print_part(inventory[i]);
```

  where the function `print_part` is defined by

```
void print_part(Part_type p) {
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n",
          p.on_hand);
}
```

- 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] = 'A';
```

Initializing an Array of Structures

- Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.

  Example:

```
struct dialing_code {
  char *country;
  int code;
};
```

  Then an array of structures can be initialized as below:

```
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}};
```

C99's designated initializers

- C99's designated initializers allow an item to have more than one designator. For example:

```
    typedef struct {
      int number;
      char name[NAME_LEN+1];
      int on_hand;
    } Part_type;
```

A declaration of the inventory[100] array that uses a designated initializer to create a single part:

```
Part_type inventory[100] =
{ [0].number = 528,
  [0].on_hand = 10,
  [0].name[0] = 'A'
} ;
```

The first two items in the initializer use two designators; the last item uses three.

Compound Literals (C99)

- We saw in Chapter 9 ***compound literals*** for arrays**.**

  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.
- A compound literal can be used to create a structure that will be passed to a function:

  print_part(**(Part_type) {528, "Disk drive", 10}**);

  The compound literal is shown in **bold**.
- A compound literal can also be assigned to a variable:

  part1 = (Part_type) {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.
- 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.

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.

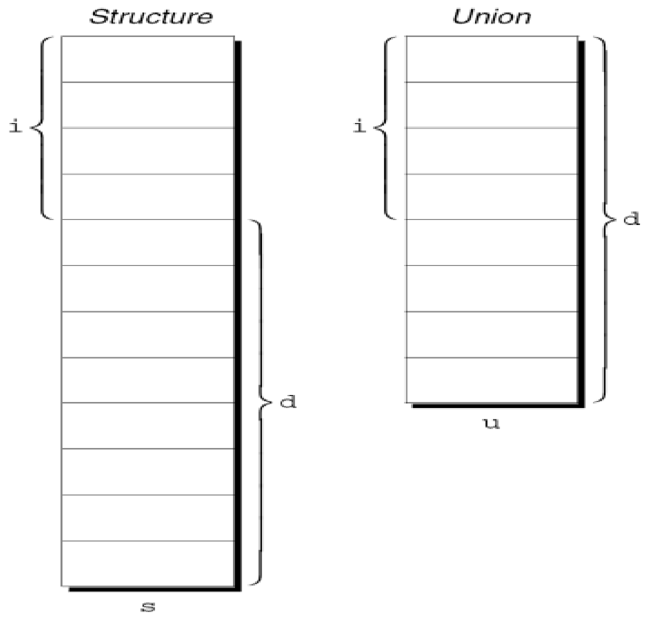
  Assigning a new value to one member alters the values of the other members as well.
- An example of a union variable:

```
    union {
      int i;
      double d;
    } u;
```

  The declaration of a union closely resembles a structure declaration:

```
    struct {
      int i;
      double d;
    } s;
```

- The structure s and the union u differ in one way.
  - The members of s are stored at different addresses in memory.
  - The members of u are stored at the same address.



- 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.

Updating in `u.d` causes any value previously stored in `u.i` to be lost. Changing `u.i` also changes `u.d`.

- 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.

- **Initialization**: Only the first member of a union can be given an initial value.

  Example: 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.

- **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};
```

  <u>Only one member can be initialized</u>, but it doesn't have to be the first one.

- **An application**: Unions can be used to create data structures that contain data of different types.

  Suppose that we need an array whose elements are a mixture of *int* and *double* values.

  <u>First</u>, define a union type whose members represent the different types of data to be stored in the array:

```
typedef union {
  int i;
  double d;
} Number_type;
```

  <u>Next</u>, create an array whose elements are Number_type values:

```
Number_type number_array[1000];
```

- 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_type 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.

- 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, as in the example:

  The `MixedNumber_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;
} MixedNumber_type;
```

  The value of kind will be either `INT_KIND` or `DOUBLE_KIND`.

- Then, each time we assign a value to a member of u, we should 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.

- 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.
- 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***).
- For example, a variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."

  enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.
- 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 can be declared in the following way:

```
enum suit s1, s2;
```

- As an alternative, we could use typedef to make Suit a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS,
              SPADES} Suit_type;
```

```
Suit_type s1, s2;
```

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.
- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- 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.
- Enumerations are perfect for determining which member of a union was the last to be assigned a value.

  In the Number_type 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_type;
```