1BEIT105/205 - Programming in C

Module-5

Structures, Unions, Enumerations, and typedef: ***Structures, Unions, Enumerations, and typedef: Structures, Arrays of Structures, Passing Structure to Functions, Structure Pointers, Arrays and Structures within Structures, Unions, Bit-Fields, Enumerations, Using sizeof to Ensure Portability, typedef.***

Textbook 1: Chapter 7

Chapter 7: Structures, Unions, Enumerations, and typedef (Page 173 – 204)

- The C language gives you ***five ways*** to create a ***custom data type***:
  1. The ***"structure"***, which is a grouping of variables under one name and is called an aggregate data type.
  2. The ***"union"***, which enables the same piece of memory to be defined as two or more different types of variables.
  3. The ***"bit-field"***, which is a special type of structure or union element that allows easy access to individual bits.
  4. The "***enumeration"***, which is a list of named integer constants.
  5. The ***"typedef"*** keyword, which defines a new name for an existing type.

Structures

- A ***structure*** is a ***collection of variables referenced under one name***, providing a convenient means of keeping related information together. A ***structure declaration forms a template*** that ***can be used to create structure objects*** (instances of a structure).
- The ***variables*** that ***make up the structure are called members***. (Structure members are also commonly referred to as elements or fields.)
- Usually, the members of a structure are logically related. The following code fragment shows how to declare a structure that defines the name and address fields. The keyword ***"struct"*** tells the ***compiler*** that ***a structure is being declared.***

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

- ***Notice that the declaration*** is ***terminated*** by a ***semicolon***. This is because a structure declaration is a statement. Also, the ***structure*** tag ***"addr" identifies this particular data structure*** and its type of ***specifier.***
- At this point, no variable has actually been created. Only the form of the data has been defined. When you declare a structure, you are defining an aggregate type, not a variable.
- To declare a variable (that is, a physical object) of type addr, write

```
struct addr addr_info;
```

- This declares a variable of type ***addr*** called ***addr_info***. Thus, ***addr*** describes the form of a structure (its type), and ***addr_info*** is an instance (an object) of the structure.

- When a structure variable (such as ***addr_info***) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members. ***Figure*** shows how ***addr_info*** appears in memory, assuming ***4-byte long*** integers.

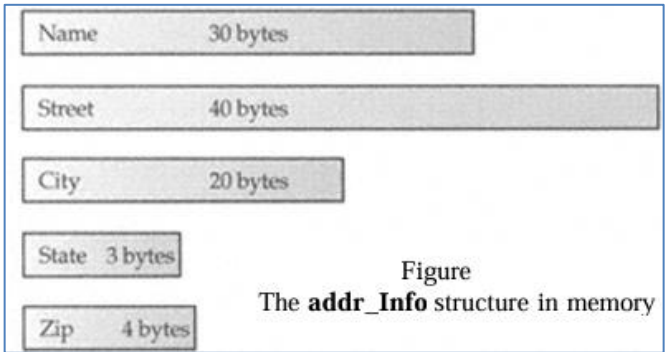


Figure
The **addr_Info** structure in memory

- You can also declare one or more objects when you declare a structure. For example,

```
struct
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

- Declares one variable named ***addr_info*** as defined by the structure preceding it. The general form of a structure declaration is

```
struct tag {
   type member-name;
   type member-name;
   type member-name;
   .
   .
   .
} structure-variables;
```

- where either tag or structure-variables may be omitted, but not both.

Accessing Structure Members

- Individual members of a structure are accessed through the use of the ***"."*** operator (usually called the ***dot operator***). For example, the following statement assigns the ZIP code 12345 to the zip field of the structure variable ***addr_info*** declared earlier: ***addr_info.zip = 12345;***
- The object name (in this case, ***addr_info***) followed by a period and the member name (in this case, zip) refers to that individual member. The ***general form*** for accessing a member of a structure is : ***object-name.member-name.***
- Therefore, to print the ZIP code on the screen, write ***printf("%lu", addr_info.zip);***
- This prints the ZIP code contained in the zip member of the structure variable ***addr_info.***
- In the same fashion, the character array addr_info.name can be used in a call to ***gets( ),*** as shown here: ***gets(addr_info.name);*** This passes a character pointer to the start of name.
- Since ***name*** is a ***character array***, you can ***access the individual characters*** of ***addr_info.name*** by indexing name. For example, you can print the contents of addr_info.name one character at a time by using the following code:

```
for(t=0; addr_info.name[t]; ++t)
  putchar(addr_info.name[t]);
```

- Notice that it is ***name*** (not addr_info) that is indexed. Remember, ***addr_info*** is the ***name of an entire structure object***; ***name*** is an element of that structure. Thus, if you want to index an element of a structure, you must put the subscript after the element's name.

Structure Assignments

- The information contained in one structure can be assigned to another structure of the same type using a single assignment statement. You do not need to assign the value of each member separately. The following program illustrates structure assignments:

```
#include <stdio.h>
int main(void)
 {
  struct {
    int a;
    int b;
  } x, y;
  x.a = 10;
  y = x;  /* assign one structure to another*/
  printf("%d", y.a);
  return 0;
 }
```

Output

```
10
```

- After the assignment, **y.a** will contain the value **10**.

Arrays of Structures

- Structures are often arrayed. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type ***"addr"*** defined earlier, write

```
struct addr addr_list[100];
```

- This creates 100 sets of variables that are organized as defined in the structure ***"addr"***.
- To access a specific structure, index the array name. For example, to print the ZIP code of structure 3, write

```
printf("%lu", addr_list[2].zip);
```

- ***Like all array variables, arrays of structures begin indexing at 0.***
- **To review:** When you want to refer to a specific structure within an array of structures, index the structure array name. When you want to index a specific element of a structure, index the element. Thus, the following statement assigns ***'X'*** to the first character of name in the third structure of ***addr_list.***

```
addr_list[2].name[0] = 'X';
```

A Mailing List Example

- To illustrate how structures and arrays of structures are used, this section develops a simple mailing list program that uses an array of structures to hold the address information.
- In this ***example***, the ***stored information*** includes ***name, street, city, state,*** and ***ZIP*** code.
- The address information is held in an array of ***"addr"*** structures, as shown here:

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_list[MAX];
```

- Notice that the zip field is an unsigned long integer. Frankly, it is more common to store postal codes using a character string because it accommodates postal codes that use letters as well as numbers. However, this example stores the ZIP code in an integer as a means of illustrating a numeric structure element.
- The first function needed for the program is ***main( ),*** shown here:

```
int main(void)
{
  char choice;
  init_list();
  /*initialize structure array*/
  for(;;) {
    choice = menu_select();
    switch(choice) {
      case 1: enter();
        break;
      case 2: delete();
        break;
      case 3: list();
        break;
      case 4: exit(0);
    }
  }
  return 0;
}
```

The function begins by initializing the structure array and then responds to menu selections.

- The function ***init_list( )*** prepares the structure array for use by putting a null character into the first byte of the name field for each structure in the array. The program assumes that an array element is not in use if name is empty. The ***init_list( )*** function is shown here:

```
/* Initialize the list. */
void initlist(void)
{
  register int t;
  for(t=0; t<MAX; ++t)
    addr_list[t].name[0] = '\0';
}
```

- The ***menu_select( )*** function displays the menu and returns the user's selection.---------->
- The ***enter( )*** function prompts the user for input and stores the information in the next free structure.
- If the array is full, the message List Full is displayed.
- ***find_free( )*** searches the structure array for an unused element.

```
 /* Get a menu selection. */
int menu_select(void)
{
  char s[80];
  int c;
  printf("1. Enter a name\n");
  printf("2. Delete a name\n");
  printf("3. List the file\n");
  printf("4. Quit\n");
  do {
    printf("\nEnter your choice: ");
    gets(s);
    c = atoi(s);
  } while(c<0 || c>4);
  return c;
 }
```

```
void enter(void) /*Input addresses into list.*/
{
  int slot;
  char s[80];
  slot = find_free();
  if(s1ot==-1) {
    printf("\nList Full");
    return;
  }
  printf("Enter name: ");
  gets(addr_list[slot].name);
  printf("Enter street: ");
  gets(addr_list[slot].street);
  printf("Enter city: ");
  gets(addr_list[slot].city);
  printf("Enter state: ");
  gets(addr_list[slot].state);
  printf("Enter zip: ");
  gets(s);
  addr_list[slot].zip = strtoul(s, '\0', 10);
 }

int find_free(void) /* Find unused structure.*/
{
  register int t;
  for(t=0; addr_list[t].name[0] && t<MAX; ++t) ;
    if(t==MAX) return -1; /* no slots free */
  return t;
}
```

- Notice that ***find_free( )*** returns a ***–1*** if every structure array variable is in use. This is a safe number because there ***cannot be a –1 element in an array***.
- The ***delete( )*** function asks the user to specify the index of the address that needs to be deleted. The function then puts a null character in the first character position of the name field.

```
/* Delete an address. */
void delete(void)
{
  register int slot;
  char s[80];
  printf("Enter record #: ");
  gets(s);
  slot = atoi(s);
  if(s1ot>=0 && slot < MAX)
    addr_list[slot].name[0] = '\0';
}
```

- The final function needed by the program is ***list( ),*** which prints the entire mailing list on the screen.
- C does not define a standard function that sends output to the printer because of the wide variation among computing environments.
- However, all C compilers provide some means to accomplish this. You might want to add printing capability to the mailing list program on your own.

```c
 /* Display the list on the screen. */
void list(void)
{
  register int t;
  for(t=0; t<MAX; ++t) {
    if(addr_list[t].name[0])  {
      printf("%s\n", addr_list[t].name);
      printf("%s\n", addr_list[t].street);
      printf("%s\n", addr_list[t].city);
      printf("%s\n", addr_list[t].state);
      printf("%lu\n\n", addr_list[t].zip);
    }
  }
  printf("\n\n");
}
```

Passing Structure to Functions (Page 186 - 188)

Topics covered: ***Passing Structure Members to Functions - Passing Entire Structures to Functions***

- This section discusses passing structures and their members to functions.

Passing Structure Members to Functions

- When you pass a member of a structure to a function, you are passing the value of that member to the function. It is irrelevant that the value is obtained from a member of a structure.
- For example, consider this structure:

struct fred	***Here are examples of each member being passed to a function:***

{
char x;
int y;
float z;
char s[10];
} ***mike;***
	`func(mike.x);  /* passes character value of x */`
	`func2(mike.y);  /* passes integer value of y */`
	`func3(mike.z);  /* passes float value of z */`
	`func4(mike.s);  /* passes address of string s */`
	`func(mike.s[2]); /* passes character value of s[2] */`

- ***In each case, it is the value of a specific element that is passed to the function***. It does not matter that the element is part of a larger unit.
- If you ***wish to pass the address of an individual structure member***, ***put the & operator before the structure name***. For example, to pass the address of the members of the structure ***"mike"***, write

```c
func(&mike.x);    /* passes address of character x */
func2(&mike.y);   /* passes address of integer y */
func3(&mike.z);   /* passes address of float z */
func4(mike.s);    /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

Note that the "&" operator precedes the structure name, not the individual member's name. Note also that "s" already signifies an address, so no "&" is required.

Passing Entire Structures to Functions

- When a ***structure is used as an argument to a function***, the ***entire structure is passed*** using the normal ***call-by-value method***.
- This means that ***any changes made to the contents*** of the parameter ***inside the function do not affect the structure passed as the argument***.

- When ***using a structure as a parameter***, remember that the ***type of the argument must match the type of the parameter***. For example, in the following program both the argument ***"arg"*** and the parameter ***"parm"*** are declared as the same type of structure.----------→
- As this program illustrates, if you will be ***declaring parameters that are structures***, you must make the ***declaration of the structure type global so that all parts of your program can use it***.
- For example, had ***struct_type*** been declared inside ***main(),*** it would not have been visible to ***f1().***
- As just stated, ***when passing structures***, the ***type of the argument must match the type of the parameter***. It is not sufficient for them simply to be physically similar; their type names must match.
- For example, the following version of the preceding program is incorrect and will not compile because the type name of the argument used to call ***f1()*** differs from the type name of its parameter.

```c
#include <stdio.h>

/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
};

void f1(struct struct_type parm);

int main(void)
{
  struct struct_type arg;
  arg.a = 1000;
  f1(arg);
  return 0;
 }

void f1(struct struct_type parm)
 {
  printf("%d", parm.a);
}
```

Output
1000

```c
/* This program is incorrect */
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
};

struct struct_type2 {
  int a, b;
  char ch;
};

void f1(struct struct_type2 parm);

int main(void)
{
  struct struct_type arg;
  arg.a = 1000;
  f1(arg); /* type mismatch */
  return 0;
 }

void f1(struct struct_type2 parm)
 {
  printf("%d", parm.a);
}
```

```c
#include <stdio.h>
/* Define two structure types */
struct struct_type {
    int a, b;
    char ch;
};
struct struct_type2 {
    int a, b;
    char ch;
};

void f1(struct struct_type2 parm);

int main(void)
{
    struct struct_type2 arg;
    /* changed type to match function */
    arg.a = 1000;

    f1(arg);   /* now correct */
    return 0;
}

void f1(struct struct_type2 parm)
{
    printf("%d", parm.a);
    /* fixed string literal */
}
```

Output
1000

Error	Fix
Passing **struct struct_type** to a function expecting **struct struct_type2**	Changed arg to type **struct_type2**
Incorrect **printf** syntax **printf("%d", parm.a);**	Changed to **printf("%d", parm.a);**
Unmatched structure types in function call	**Both now match correctly**

Structure Pointers (Page 188 - 192)

- C allows pointers to structures just as it allows pointers to any other type of object. However, there are some special aspects to structure pointers.

Topics covered: **Declaring a Structure Pointer - Using Structure Pointers**

Declaring a Structure Pointer

- Like other pointers, structure pointers are declared by placing * in front of a structure variable's name. For example, assuming the previously defined structure ***addr***, the following declares ***addr_pointer*** as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Using Structure Pointers

- There are ***two primary uses for structure pointers***:
  1. To ***pass a structure to a function*** using ***call by reference*** and to ***create linked lists*** and
  2. ***Dynamic data structures*** that rely on ***dynamic allocation***.
- There is one ***major drawback*** to passing all but the simplest structures to functions: the overhead needed to push the structure onto the stack when the function call is executed. (Recall that arguments are passed to functions on the stack.)
- For ***simple structures with few members, this overhead is not too great***.
- **Problem**: If the structure ***contains many members***, if ***some*** of its members ***are arrays***, run-time ***performance may degrade*** to unacceptable levels.
- **Solution**: ***To pass a pointer to the structure***.
- When a pointer to a structure is passed to a function, only the address of the structure is pushed on the stack. This makes for very fast function calls.
- A second advantage, in some cases, is that passing a pointer makes it possible for the function to modify the contents of the structure used as the argument.
- To find the address of a structure variable, place the & operator before the structure's name. For example, given the following fragment,

```
struct bal {
  float balance;
  char name[80];
} person;
struct bal *p;
/* declare a structure pointer */
```

  this places the address of the structure person into the pointer p: ***"p = &person;"***.
- To ***access the members of a structure*** using a pointer to that structure, you must use the ***–>*** operator. For example, this references the balance field: ***p–>balance.***
- The ***–>***, usually called the ***arrow operator***. The ***arrow is used in place of the dot operator*** when you are ***accessing a structure member through a pointer to the structure***.
- To see how a structure pointer can be used, examine this simple program.

```
#include <stdio.h>
/* Define a structure */
struct student {
    int roll;
    float marks;
};

int main() {
    struct student s1 = {101, 89.5};

    /* Declare a structure pointer */
    struct student *ptr;

    ptr = &s1;  /* Point to s1 */
/* Access structure members using -> operator */
    printf("Roll Number : %d\n", ptr->roll);
    printf("Marks       : %.2f\n", ptr->marks);

    return 0;
}
```

```
Output
Roll Number : 101
Marks       : 89.50
```

- Remember, use the dot operator to access structure elements when operating on the structure itself. When you have a pointer to a structure, use the arrow operator.

Arrays and Structures within Structures

- A member of a structure can be ***either a simple variable***, such as an ***int*** or ***double***, ***or an aggregate type***. In C, ***aggregate types are arrays and structures***. You have already seen one type of aggregate element: the character arrays used in ***addr***.
- A member of a structure that is an array is treated as you might expect from the earlier examples. For example, consider this structure:

```
struct x {
  int a[10] [10];
  float b;
} y;
```

- To ***reference integer 3,7 in a of structure y***, write ***y.a[3][7]***
- When a ***structure is a member of another structure,*** it is called a ***nested structure***. For example, the structure ***"address"*** is nested inside ***"emp"*** in this example:

```
struct emp {
  struct addr address; /* nested structure */
  float wage;
} worker;
```

- Here, structure emp has been defined as having two members. The first is a structure of type ***addr***, which contains an employee's address. The other is ***wage***, which holds the employee's wage. The following code fragment assigns ***560083*** to the ***zip*** element of address.

```
worker.address.zip = 560083;
```

- As you can see, the members of each structure are referenced from outermost to innermost. The ***C89 standard*** specifies that ***structures*** can be ***nested to at least 15 levels***.
- The ***C99 standard*** suggests that at least ***63 levels*** of nesting be allowed.

Unions

- A union is a memory location that is shared by two or more different types of variables. A union provides a way of interpreting the same bit pattern in two or more different ways. Declaring a union is similar to declaring a structure.

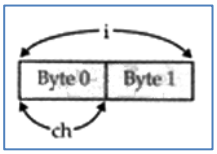
General form	Example

union tag {
type member-name;
type member-name;
.
.
} union-variables;
	union u_type {
	int i;
	char ch;
	};

- This declaration does not create any variables. You can declare a variable either by placing its name at the end of the declaration or by using a separate declaration statement. To declare a union variable called ***cnvt*** of type ***u_type*** using the definition just given, write

```
union u_type cnvt;
```

- In ***cnvt***, both integer ***"i"*** and character ***"ch"*** share the same memory location. Of course, ***"i"*** occupies 2 bytes (assuming 2-byte integers), and ***"ch"*** uses only 1. Figure shows how ***"i"*** and ***"ch"*** share the same address.



At any point in your program, you can refer to the data stored in a ***cnvt*** as ***either an integer*** or a ***character***.

- When a ***union variable*** is ***declared***, the ***compiler automatically allocates*** enough ***storage*** to hold the largest member of the union. For example, (assuming 2-byte integers) ***cnvt*** is 2 bytes long so that it can hold ***"i"***, even though ***"ch"*** requires only 1 byte.
- To ***access a member of a union***, use the ***same syntax that you would use for structures***: the ***dot(.)*** and ***arrow(->)*** operators.
- If you are ***operating on the union directly, use the dot operator***. If the ***union is accessed through a pointer***, ***use the arrow operator***. For example, to assign the integer 10 to element ***"i"*** of ***cnvt***, write ***cnvt.i = 10;***
- In the ***next example, a pointer to cnvt is passed to a function:***

```
void func1(union u_type *un)
{
 un-> = 10;
}
```

- Unions are used frequently when specialized type conversions are needed because you can refer to the data held in the union in fundamentally different ways. For example, you might use a union to manipulate the bytes that constitute a double in order to alter its precision or to perform some unusual type of rounding.
- To get an idea of the usefulness of a union when nonstandard type conversions are needed, consider the problem of writing a short integer to a disk file.
- The C standard library defines no function specifically designed to write a short integer to a file. Although you can write any type of data to a file using ***fwrite( ),*** using ***fwrite( )*** incurs excessive overhead for such a simple operation. However, using a union, you can easily create a function called ***putw( ),*** which writes the binary representation of a short integer to a file one byte at a time. (This example assumes that short integers are 2 bytes long.)

```c
#include <stdio.h>
#include <stdlib.h>
union pw {
  short int i;
  char ch[2];
};
int putw(short int num, FILE *fp);
 int main(void)
 {
  FILE *fp;
  fp = fopen("test.tmp", "wb+");
  if(fp == NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }
  putw(1025, fp);  /* write the value 1025 */
  fclose(fp);
  return 0;
 }
 int putw(short int num, FILE *fp)
 {
  union pw word;
  word.i = num;
  putc(word.ch[0], fp); /* write first half */
  return putc(word.ch[1], fp); /* write second half */
 }
```

- To see how, first create a union consisting of one short integer and a 2 byte character array:

```
union pw {
 short int i;
 char ch[2];
};
```

- Now, you can use ***"pw"*** to create the version of ***putw( )*** shown in the following program.---------->
- Although ***putw( )*** is called with a short integer, it can still use the standard function putc( ) to write each byte in the integer to a disk file one byte at a time.

Difference between Structure and Union in C

Feature	Structure (*struct*)	Union (*union*)
Memory allocation	Allocates **separate memory** for each member. Total size = sum of all members' sizes (plus padding).	Allocates **shared memory** for all members. Total size = size of the **largest** member.
Storage of values	All members can **store values simultaneously**.	Only **one member can store a value at a time**—updating one overwrites the others.
Usage	Used when you need to store **multiple independent values**.	Used when you need **memory-efficient** storage for variables that are **not used at the same time**.
Access	Each member can be accessed individually and safely.	Accessing a member different from the last written one gives **garbage/undefined** data.
Initialization	All members can be initialized together.	Only the **first member** can be initialized at declaration.
Example Use Cases	Records (student, employee, books), data structures.	Embedded systems, memory-critical applications, hardware registers.

Bit-Fields

- Unlike some other computer languages, C has a ***built-in feature***, called a ***bit-field***, that ***allows you to access a single bit***.
- Bit-fields can be ***useful*** for a number of reasons, such as:
  - If storage is limited, you can ***store several Boolean (true/false) variables in one byte***.
  - Certain ***devices transmit status information encoded*** into one or more bits within a byte.
  - Certain ***encryption routines need to access the bits within a byte***.
- Although these tasks can be performed using the bitwise operators, a bit-field can add more structure (and possibly efficiency) to your code.
- A bit-field must be a member of a structure or union. It defines how long, in bits, the field is to be. The general form of a bit-field definition is ***type name: length;***
  - Here, ***type*** is the type of the bit-field, and length is the number of bits in the field. The type of a bit field must be ***int***, ***signed***, or ***unsigned***.
  - C99 also allows a bit-field to be of type _Bool.
- ***Bit-fields are frequently used when analyzing input from a hardware device***. For example, the ***status port of a serial communications adapter*** might return a status byte organized like this:

Bit	Meaning When Set
0	Change in clear-to-send line
1	Change in data-set-ready
2	Trailing edge detected
3	Change in receive line
4	Clear-to-send
5	Data-set-ready
6	Telephone ringing
7	Received signal

- You can represent the information in a status byte using the following bit-field----------→

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```

- You might use statements like the ones shown here to enable a program to determine when it can send or receive data:

```
status = get_port_status();
if(status.cts) printf("clear to send");
if(status.dsr) printf("data ready");
```

- To assign a value to a bit-field, simply use the form you would use for any other type of structure element. For example, this code fragment clears the ring field: ***status.ring = 0;***
- As you can see from this example, each bit-field is accessed with the ***dot*** **operator**. However, if the structure is referenced through a pointer, you must use the **–> operator**.
- You do not have to name each bit-field. This makes it easy to reach the bit you want, bypassing unused ones.
- For example, if you only care about the ***cts*** and ***dsr*** bits, you could declare the ***status_type*** structure like this:

```
struct status_type {
  unsigned :    4;
  unsigned cts: 1;
  unsigned dsr: 1;
} status;
```

- Also, notice that the bits after ***dsr*** do not need to be specified if they are not used. It is valid to mix normal structure members with bit-fields. For example,

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:    1; /* lay off or active */
  unsigned hourly:     1; /* hourly pay or wage */
  unsigned deductions:3; /* IRS deductions */
};
```

  defines an employee record that uses only 1 byte to hold three pieces of information: the employee's status, whether the employee is salaried, and the number of deductions. Without the bit-field, this information would take 3 bytes.

- Bit-fields have certain restrictions. You cannot take the address of a bit-field. Bit-fields cannot be arrayed. You cannot know, from machine to machine, whether the fields will run from right to left or from left to right; this implies that any code using bit-fields may have some machine dependencies. Other restrictions may be imposed by various specific implementations.

Enumerations

- An enumeration is a set of named integer constants. Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is ***penny, nickel, dime, quarter, half-dollar, dollar***
- Enumerations are defined much like structures; the keyword enum signals the start of an enumeration type. The general form for enumerations is

  enum tag { enumeration list } variable_list;

- Here, both the tag and the variable list are optional. (But at least one must be present.) The following code fragment defines an enumeration called ***coin***:

```
enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};
```

- The enumeration tag name can be used to declare variables of its type. The following declares money to be a variable of type coin: ***enum coin money;***

- Given these declarations, the following types of statements are perfectly valid:

```
money = dime;
if(money==quarter) printf("Money is a quarter.\n");
```

- The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they can be used anywhere that an integer can be used. Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore, ***printf("%d %d", penny, dime); displays 0 2 on the screen.***
- You can specify the value of one or more of the symbols by using an initializer. Do this by following the symbol with an equal sign and an integer value. Symbols that appear after an initializer are assigned values greater than the preceding value.
- For example, the following code assigns the value of 100 to quarter:

```
enum coin  { penny, nickel, dime, quarter=100, half_dollar, dollar};
```

- Now, the values of these symbols are

Symbols	Values
penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

- One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case.
- For example, the following code fragment will not perform as desired:

```
/* this will not work */
money = dollar;
printf("%s", money);
```

- Remember, dollar is simply a name for an integer; it is not a string. Thus, attempting to output money as a string is inherently invalid. For the same reason, you cannot use this code to achieve the desired results: ***/* this code is wrong */ strcpy(money, "dime");***
- That is, a string that contains the name of a symbol is not automatically converted to that symbol.
- Actually, creating code to input and output enumeration symbols is quite tedious (unless you are willing to settle for their integer values).
- For example, you need the following code to display, in words, the kind of coin that money contains:

```
switch(money) {
  case penny: printf("penny");
    break;
  case nickel: printf("nickel");
    break;
  case dime: printf("dime");
    break;
  case quarter: printf("quarter");
    break;
  case half_dollar: printf("half_dollar");
    break;
  case dollar: printf("dollar");
}
```

- Sometimes, you can declare an array of strings and use the enumeration value as an index to translate that value into its corresponding string. For example, this code also outputs the proper string:

```
char name[][12]={
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[money]);
```

- Of course, this only works if no symbol is initialized, because the string array must be indexed starting at 0 in strictly ascending order using increments of 1.
- Since enumeration values must be converted manually to their human-readable string equivalents for I/O operations, they are most useful in routines that do not make such conversions. An enumeration is often used to define a compiler's symbol table, for example.

Using "sizeof" to Ensure Portability

- The ***"sizeof"*** operator computes the size of any variable or type and can help eliminate machine-dependent code from your programs. This operator is especially useful where structures or unions are concerned.
- For the following discussion, assume an implementation that has the sizes for the data types shown here:

Type	Size in Bytes
char	1
int	4
double	8

- Therefore, the following code will print the numbers 1, 4, and 8 on the screen:

```
char ch;
int i;
double f;

printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

- The size of a structure is equal to or greater than the sum of the sizes of its members.
- For example:

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

- Here, ***sizeof(s_var)*** is at least ***13 (8+4+1).*** However, the size of ***s_var*** might be greater because the compiler is allowed to pad a structure in order to achieve word or paragraph alignment.
- Since the size of a structure may be greater than the sum of the sizes of its members, you should always use ***sizeof*** when you need to know the size of a structure.

- For example, if you want to dynamically allocate memory for an object of ***type s,*** you should use a statement sequence like the one shown here (rather than manually adding up the lengths of its members):

```
struct s *p;
p = malloc(sizeof(struct s));
```

- Since ***sizeof*** is a ***compile-time operator***, all the information necessary to compute the size of any variable is known at compile time. This is especially meaningful for unions, because the size of a union is always equal to the size of its largest member.
- For example, consider

```
union u {
  char ch;
  int i;
  double f;
} u_var;
```

- Here, the ***sizeof(u_var)*** is ***8***. At run time, it does not matter what ***u_var*** is actually holding. All that matters is the size of its largest member, because any union must be as large as its largest element.

"typedef"

- You can define new data type names by using the keyword ***"typedef"***. You are ***not actually creating a new data type, but rather defining a new name for an existing type***.
- This process can ***help make machine-dependent programs more portable***. If you define your own type name for each machine dependent data type used by your program, then only the ***"typedef"*** statements have to be changed when compiling for a new environment.
- ***"typedef"*** also can aid in self-documenting your code by allowing descriptive names for the standard data types.
- The general form of the ***"typedef"*** statement is

```
typedef type newname;
```

where ***type*** is any valid data type, and ***newname*** is the new name for this type. The new name you define is in addition to, not a replacement for, the existing type name.

- For example, you could create a new name for ***float*** by using "***typedef float balance;***"
- This statement tells the compiler to recognize ***"balance"*** as another name for ***float***. Next, you could create a ***float*** variable using ***balance***: ***balance over_due;***
- Here, ***over_due*** is a ***floating-point variable*** of type ***balance***, which is ***another word for float***. Now that balance has been defined, it can be used in another ***"typedef"***.
- For example, ***typedef balance overdraft;*** tells the compiler to recognize overdraft as another name for ***balance***, which is another name for ***float***.
- Using ***"typedef"*** can make your code easier to read and easier to port to a new machine. But you are not creating a new physical type.
