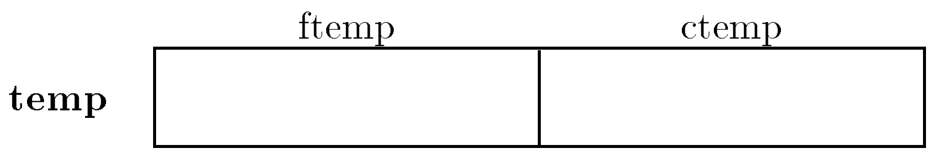

Figure 12.2: Structure Variable `temp` in Memory

to be of structure type with the declaration statement:

```
struct trecd {
     float ftemp;
     float ctemp;
} temp;
```

This statement consists of the keyword, `struct`, followed by the description of the template for the structure and then the variable name. The description of the template, in our example, consists of a *tag* (or name), `trecd` which names the template, followed by a bracketed list of field declarations. The tag is optional. Within its scope, the tag can be used to refer to this structure template without specifying the fields again, explicitly. The bracketed list declares the fields of the structure giving a type followed by an identifier. Our example shows that this structure has two fields: `ftemp` and `ctemp`, both of type `float`.

Figure 12.2 shows the memory cells allocated to the variable `temp`. Two `float` cells have been allocated, one referred to as the `ftemp` field and the other as `ctemp`. The entire block of memory is referred to by the variable name, `temp`. Otherwise, structure declarations are the same as any other variable declaration and have the same scope as would an `int` declaration, for example.

To access the information in a structure, the variable name (in our case, `temp`) is *qualified* using the "dot" operator (`.`) followed by the field name:

```
temp.ftemp
temp.ctemp
```

In general, the syntax for accessing a member of a structure is:

<variable_identifier>.<member_identifier>

In a program, members of a structure variable may be used just like other variables. In the function `main()` above, the argument to `scanf()` is `&temp.ftemp` which is the address of the float cell, `temp.ftemp`. (Precedence of the dot operator is higher than that of the address operator so no parentheses are needed in this case). The numeric value read by `scanf()` will be stored where the argument points — it will be stored in the cell `temp.ftemp`. The rest of the program is straight forward. We have passed a double value to the function `f_to_c` and get a double result which we assign to `temp.ctemp` and print the results.

Sample Session:

```
***Temperatures - Degrees F and C***
```

```
Enter temperature in degrees F : 78
Temp in degrees F = 78.0
Temp is degrees C = 25.6
```

As we have said, the members of a structure variable can be of different types. For example:

```
struct {
     char name[26];
     int id_number;
} student;
```

which defines a structure variable, `student`, with two fields: a string of characters called `name`, and an integer called `id_number`. Enough contiguous memory is allocated for the variable student to accommodate both fields. We can find the amount of storage allocated for a structure by using the `sizeof` operator. (Be aware that the total size of a structure variable may not be equal to the sum of the sizes for the fields because of rules about memory alignment which may vary from computer to computer. For example, memory allocation for an integer may have to start at a machine word boundary such as an even byte address. Such alignment requirements may make the size of a structure variable somewhat greater than the sum of the field sizes).

The identifiers used for the field names apply only to variables of that structure type. Different structure types may have fields specified by the same identifier, but these are distinct cells, uniquely accessed by an appropriate structure variable name qualified by a field names. In addition, only field names declared in the structure template can be used to qualify a variable name. And finally, a field name may not be used by itself — it must always qualify an appropriate structure variable. Consider the following examples of structure variable declarations:

```
struct {
     char f_name[10];
     char m_inits[3];
     char l_name[20];
     int id_no;
     int b_month;
     int b_day;
     int b_year;
} person, manager;

struct {
     int id_no;
     float cost;
     float price;
} part;
```

Here we have declared two variables, `person` and `manager`, to be structures with seven fields, some integers, some strings. In this case, two separate instances of the template are allocated, so person.id_no and manager.id_no are distinct storage cells. We have also defined a variable, `part`, whose template also has a `id_no` field name. But this is also a distinct storage location accessed by `part.id_no`. However, with these declarations, it is NOT legal to refer to the `cost` field of `person` (`person.cost`) or the _day of a `part` (`part.b_day`). Similarly, referring to `f_name` or `price` is not legal without a variable name of the appropriate type to be qualified. Here are some legal examples of structure usage:

```
part.id_no = 99;
part.cost = .2239;
if (strcmp(person.f_name, "Helen") == 0)
     printf("Last name is %s\n", person.l_name);
printf("This is the cost %d\n",part.cost);
part.price = part.cost * 2.0;
```

The only legal operations allowed on a structure variable are finding the address of the memory block using `&`, accessing its members, and copying or assigning it as a unit as long as the variables are of an identical structure type, for example:

```
manager = person;
```

12.1.2 Using Structure Tags

As we said above, declaring a structure variable requires two things — describing the template for the structure, and declaring variables of that structure type. It is also possible to perform these two steps in separate statements in a program. That is declare just a structure template with a tag without any variables declared; and later, declare variables of the structure type identified by the tag. For example, this declaration:

```
struct stdtype {
  char name[26];
  int id_number;
};
```

specifies a template with a tag for a structure type, `stdtype`. (Observe the semicolon after the declaration for the declaration). Such a declaration does NOT allocate memory, since no variables are declared; it merely defines a template for variables declared later. Within the scope of the tag declaration, we can then declare `stdtype` structure variables like:

```
struct stdtype x, y, z;
```

This declaration allocates memory for three variables, `x`, `y` and `z`, of type structure `stdtype`; i.e. fitting the template defined earlier. Some additional examples of structure tag and variable declarations are:

```
/* named structure template, no variables declared */
struct date {
     int month;
     int day;
     int year;
};

/* named structure template and a variable declared */
```

```
    struct stu_rec {
         char name[30];
         char class_id[3];
         int test[3];
         int project;
         int grade;
    } student;

    struct stu_rec ee_stu, me_stu;
    struct date today, birth_day;
```

The main advantage of splitting the template definition from variable declaration is that the template need be defined only once and may then be used to declare variables in many places. We will see the utility of this below when we pass structures to function.

In general, then, a structure declaration has the following syntax:

```
struct [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];
```

where the <tag_identifier> and variable identifiers are optional. Once a template has been defined, additional variables of the structure type may be declared by:

```
struct <tag_identifier> <identifier>[, <identifier>...];
```

The types of the fields of the structure may be any valid C type; a scalar data type (`int`, `float`, etc.), an array, or even a structure. This means that nested structure types can also be declared:

```
    struct inventory {
         int item_no;
         float cost;
         float price;
         struct date buy_date;
    };

    struct car_type{
         struct inventory part;
         struct date ship_date;
         int shipment;
    } car;
```

Here, the `ship_date` field of the `car_type` structure is itself a `date` structure (from above) and the `part` field is an `inventory` structure, with `item_no`, `cost`, etc. fields, including yet another `date structure` within. The members for nested structures may be accessed with dot operators applied successively from left to right (the grouping for the dot operator is from left to right). Thus:

```
    car.ship_date.month = 5;      /* Lvalue is (car.ship_date).month */
    car.part.buy_date.month = 12;
```

these assignments refer to the month field of the ship_date field of the variable car and the month field of the buy_date field of the part field of the variable, car, respectively.

Both the variables of structure type and the structure tags are frequently referred to as structures. Thus, we may refer to date as a structure, and we may say that the variable, today is a structure. It is usually clear from the context whether a structure tag or a variable of structure type is meant. However, for the most part, we will use the term structure for the templates themselves, i.e. tags; and, we will specify variables to be of structure type. Thus, date is a structure; and today is a structure variable or a variable of structure type, i.e. of type, struct date.

As with other data types, structures can be initialized in declarations by specifying constant values for the structure members within braces. The initializers for structure members are separated by commas as for an array. For example, a struct inventory item can be declared:

```
    struct inventory part = { 123, 10.00, 13.50 };
```

which initializes member, part_no to 123, member cost to 10.00, and member price to 13.50. As another example, a label item can be declared as:

```
    struct name {
         char f_name[10];
         char m_inits[3];
         char l_name[20];
    };

    struct address {
         char street[30];
         char city[15];
         char state[15];
         int zip};
    };

    struct label {
         struct name name;
         struct address address;
    };

    struct label person = { {"Jones", "John", "Paul"},
                   {"23 Dole Street", "Honolulu", "Hawaii", 96822} };
```

The structure, label has two members, each of which is a structure. The first member, name, has three members, and the second member, address, has four members. Initialization for each member structure is nested appropriately.

12.1.3 Structures and Functions

Structure variables may be passed as arguments and returned from functions just like scalar variables. Let us consider an example that reads and prints a data record for a part. The record consists of the part number, its cost and retail price. (In a later section, we will see how an inventory for a list of parts can be maintained). The code to read and print a single part structure is shown in Figure 12.3. Notice we have declared the structure template, `inventory`, at the head of the source file. This is called an *external* declaration and the scope is the entire file after the declaration. Since all the functions in the file use this structure tag, the template must be visible to each of them. The driver calls `read_part()` to read data into a structure and return it to be assigned to the variable, `item`. Next, it calls `print_part()` passing `item` which merely prints the values of the structure fields, one at a time. The program is straightforward. A sample session is shown below:

```
***Part Inventory Data***

Part Number:  2341
Cost:  12.5
Price:  15
Part no.  = 2341, Cost = 12.50, Retailprice =15.00
```

External declarations of structure templates and prototypes facilitate consistent usage of tags and functions. As a general practice, we will declare structure templates externally, usually at the head of the source file. Sometimes, external structure tag declarations will be placed in a separate header file, which is then made part of the source file by an include directive.

From this example, we can see that using structures with functions is no different than using any scalar data type like `int`. However, let us consider what really happens when the program runs. When the function `read_part()` is called, memory is allocated for all of its local variables, including the `struct inventory` variable, `part`. As each data item is read, it is placed in the corresponding field of `part`, accessed with the **dot** operator. The **value** of `part` is then returned to `main()` where it is assigned to the variable `item`. As would be the case for a scalar data type, the value of the return expression is **copied** back to the calling function. Since this is a structure, the entire structure (each of the fields) is copied. For our `inventory` structure, this isn't too bad — only two floats and an integer. If the structure where much larger, maybe including nested structures and arrays, many values would need to be copied.

Likewise with the call to `print_part()`. Here, an `inventory` structure is passed to the function. Recall that in C, all parameters are passed by value — the value of each argument expression is **copied** from the calling function into the cell allocated for the parameter of the called function. Again, for large structures, this may not be a very efficient way to pass data to functions. In the next section we see a way to remedy this problem.

12.1.4 Pointers to Structures

As we saw in the last section, passing and returning structures to functions may not be efficient, particularly if the structure is large. We can eliminate this excessive data movement by passing pointers to the structures to the function, and access them indirectly through the pointers. Figure 12.4 shows a modified version of our previous program which uses pointers instead of passing entire structures.

The code is very similar to Figure 12.3, but we have changed the prototypes and functions to work with pointers. The argument of `read_part()` is a pointer to the `inventory` structure, `item`

```
/*   File: part.c
     This program reads and prints inventory data for a part.
*/
#include <stdio.h>

struct inventory {
     int part_no;
     float cost;
     float price;
};

struct inventory read_part(void);
void print_part(struct inventory part);

main()
{    struct inventory item;

     printf("***Part Inventory Data***\n\n");
     item = read_part();
     print_part(item);
}

/*   Prints data for a single part. */
void print_part(struct inventory part)
{
     printf("Part no. = %d,  Cost = %5.2f, Retail price = %5.2f\n",
               part.part_no, part.cost, part.price);
}

/*   Reads data for a single part  structure and returns the
     structure.
*/
struct inventory read_part(void)
{    int n;
     float x;
     struct inventory part;
     printf("Part Number: ");
     scanf("%d", &n);
     part.part_no = n;
     printf("Cost: ");
     scanf("%f", &x);
     part.cost = x;
     printf("Price: ");
     scanf("%f", &x);
     part.price = x;
     return part;
}
```

Figure 12.3: Code for Reading and Printing a Single Part

```
/*   File: part.c
     This program reads and prints inventory data for a part.
*/
#include <stdio.h>

struct inventory {
     int part_no;
     float cost;
     float price;
};

void read_part(struct inventory * partptr);
void print_part(struct inventory * partptr);

main()
{    struct inventory item;

     printf("***Part Inventory Data***\n\n");
     read_part(&item);
     print_part(&item);
}

/*   Prints data for a single part pointed to by partptr. */
void print_part(struct inventory * partptr)
{
     printf("Part no. = %d,  Cost = %5.2f, Retail price = %5.2f\n",
               (* partptr).part_no, (* partptr).cost, (* partptr).price);
}

/*   Reads data for a single part into an object pointed to
     by partptr.
*/
void read_part(struct inventory * partptr)
{    int n;
     float x;
     struct inventory part;
     printf("Part Number: ");
     scanf("%d", &n);
     (* partptr).part_no = n;
     printf("Cost: ");
     scanf("%f", &x);
     (* partptr).cost = x;
     printf("Price: ");
     scanf("%f", &x);
     (* partptr).price = x;
}
```

Figure 12.4: Code for Reading and Printing a Part Using Pointers

declared in `main()`. The function accesses the object pointed to by `partptr`, and uses the dot operator to access a member of that object. Since `partptr` points to an object of type `struct inventory`, we dereference the pointer to access the members of the object:

```
(*partptr).part_no
(*partptr).cost
(*partptr).price
```

Similar changes have been made to `print_part()`. Note, the parentheses are necessary here because the `.` operator has higher precedence than the indirection operator, `*`. We must first dereference the pointer, and then select its appropriate member.

Since, for efficiency, pointers to structures are often passed to functions, and, within those functions, the members of the structures are accessed, the operation of dereferencing a structure pointer and a selecting a member is very common in programs. Therefore, C provides a special pointer operator, $->$, (called *arrow*) to access a member of a structure pointed to by a pointer variable. The operator is a minus symbol, `-`, followed by a greater-than symbol, >. This operator is exactly equivalent to a dereference operation followed by the `.` operator as shown below:

```
partptr- >part_no   <=>  (*partptr).part_no
partptr- >cost      <=>  (*partptr).cost
partptr- >retail    <=>  (*partptr).price
```

The left hand expressions are equivalent ways of writing expressions on the right hand side, e.g. `prtptr-` >*member* accesses the member of an object pointed to by `partptr`. Our code for `read_part()` could use the following alternative expressions:

```
partptr->part_no = n;
partptr->cost = x;
partptr->price = x;
```

The general syntax for using the arrow operator is:

<variable_identifier> – ><member_identifier>

which is equivalent to:

(* <variable_identifier>).<member_identifier>

We now consider an example using nested structures. The program reads and prints data for a single label consisting of members that are themselves structures. The first member is a structure for a name, the second is a structure for an address. This program is organized in several source and header files as shown in Figure 12.5. (We intend to use the functions in these files for other programs as well).

The driver calls the function `readlabel()` to read in the label data, and the function `printlabel()` to print the label data. Like the previous example, in both function calls, we assume that a pointer to a `struct label` variable is passed as an argument. In the functions, we will use the pointer operator, $->$, to access the members of the object pointed to by the pointer. The function prototypes are shown in the header file `lblutil.h`. The functions are shown in Figure 12.6

The formal parameter in the functions `readlabel()` and `printlabel()` are both a pointer, called `pptr`, which points to an object of type `struct label`. Each function accesses the `first` field of the `name` field of the object pointed to by `pptr` as follows:

```
/*   File: lbl.h
     This file contains structure tags for labels. Label has two members,
     name and address, each of which is a structure type.
*/
struct name_recd {
     char last[15];
     char first[15];
     char middle[15];
};

struct addr_recd {
     char street[25];
     char city[15];
     char state[15];
     long zip;
};

struct label {
     struct name_recd name;
     struct addr_recd address;
};

/* File: lblutil.h */
void printlabel(struct label * personptr);
int readlabel(struct label * personptr);

/*   File: lbl.c
     Other Source Files: lblutil.c
     Header Files: lbl.h, lblutil.h
     This program reads and prints data for one label.
*/
#include <stdio.h>
#include "lbl.h"
#include "lblutil.h"
main()
{    struct label person;

     printf("***Label Data for a Person***\n\n");
     readlabel(&person);
     printf("\nLabel Data:\n");
     printlabel(&person);
}
```

Figure 12.5: Driver and Header Files for Label Program

```
/*   File: lblutil.c */
#include <stdio.h>
#include "lbl.h"
#include "lblutil.h"
#define FALSE 0
#define TRUE 1

/* This routine prints the label data. */
void printlabel(struct label * pptr)
{
     printf("\n%s %s %s\n%s\n%s %s %ld\n",
               pptr->name.first,
               pptr->name.middle,
               pptr->name.last,
               pptr->address.street,
               pptr->address.city,
               pptr->address.state,
               pptr->address.zip);
}

/* This routine reads the label data. */
int readlabel(struct label * pptr)
{    int x;

     printf("Enter Name <First Middle Last>, EOF to quit: ");
     x = scanf("%s %s %s%*c",pptr->name.first,
                    pptr->name.middle,
                    pptr->name.last);
     if (x == EOF)
          return FALSE;
     printf("Enter Street Address: ");
     gets(pptr->address.street);
     printf("Enter City State Zip: ");
     scanf("%s %s %ld%*c",pptr->address.city,
               pptr->address.state,
               &(pptr->address.zip));
     return TRUE;
}
```

Figure 12.6: Code for Label Utility Functions

```
pptr->name.first
```

Remember, this is the same as:

```
(*pptr).name.first
```

which means `pptr` is first dereferenced; the `name` field of the dereferenced object is accessed next, and finally the `first` field of `name` is accessed (the dot operator groups from left to right). Similarly, other members of the object pointed to by `pptr` are accessed by:

```
pptr->name.middle
pptr->name.last
pptr->address.street
pptr->address.city
pptr->address.state
pptr->address.zip
```

All the above members, except `zip`, are strings. In `readlabel()`, `scanf()` expects to be passed pointers to objects to store the data read. Since all the string members are already pointers, we need to use the address operator only when we pass the pointer to `pptr->address.zip`. Notice, we use the suppression conversion, `%*c`, to discard the newline character at the end of each line. Thus, after the name is read, `gets()` reads the street address correctly. The function returns `TRUE` if a label is read successfully, and `FALSE` otherwise, i.e. when an EOF is entered by the user for the name, indicating that no label data is available. The function `printlabel()` could have been passed the structure variable itself since it merely needs to print the values of the members; however, as we discussed above, passing a pointer avoids the expense of copying the entire structure. Here is a sample session:

```
***Label Data for a Person***

Enter Name <First Middle Last>, EOF to quit:  John Paul Jones
Enter Street Address:  23 Dole Street
Enter City State Zip:  Honolulu Hawaii 96822

Label Data:

John Paul Jones
23 Dole Street
Honolulu Hawaii 96822
```

12.2 Arrays of Structures

The inventory and the label program examples of the last section handle only a single record. More realistically, a useful program may need to handle many such records. As in previous cases where we needed to store a list of items, we can use an array as our data structure. In this case, the elements of the array are structures of a specified type. For example:

	part_no	cost	price
table[0]—>	123	10.00	15.00
table[1]—>	.	.	.
table[2]—>	.	.	.
table[3]—>	.	.	.

Figure 12.7: A Table of Part Records

```
struct inventory {
     int part_no;
     float cost;
     float price;
};

struct inventory table[4];
```

which defines an array with four elements, each of which is of type `struct inventory`, i.e. each is an `inventory` structure.

We can think of such a data structure as a tabular representation of our data base of parts inventory with each row representing a part, and each column representing information about that part, i.e. the `part_no`, `cost`, and `price`, as shown in Figure 12.7. This is very similar to a two dimensional array, except that in an array, all data items must be of the same type, where an array of structures consists of columns, each of which may be of a distinct data type. As with any array, the array name used by itself in an expression is a pointer to the entire array of structures. Therefore, the following are equivalent ways of accessing the elements of the array.

*(table)	table[0]
(table + 1)	table[1]
(table + 2)	table[2]
(table + 3)	table[3]

With this in mind, let us extend out address label program from Section 12.1.4 to read and print a list of labels. The code is shown in Figure 12.8 and uses the same structures and functions used in program `lbl.c` included in files `lbl.h` and `lblutil.c`.

In the program, the reading of labels is still performed by `readlabel()` only now in a while loop. The loop terminates when either `MAX` number of labels have been read or `readlabel()` returns `FALSE` at end of file. In this case, a new label is not read, but the value of `n` is incremented anyway by the `++` operator. Thus, if the loop is terminated because of an end of file, the incremented value of `n` must be decremented to correctly indicate the number of entries in the array. Finally, labels are printed using `printlabel()` in a loop.

Sample Session:

```
***Labels - Input/Output***
```

```
/*   File: labels.c
     Other Source Files: lblutil.c
     Header Files: lbl.h, lblutil.h
     This program reads in a set of labels, and prints them out.
*/
#define MAX 100
#include <stdio.h>
#include "lbl.h"         /* declarations for the structures */
#include "lblutil.h"     /* prototypes for routines in file lblutil.c */
main()
{    struct label person[MAX];
     int i, n;

     printf("***Labels - Input/Output***\n\n");
     n = 0;
     /* read the labels */
     while (n < MAX && readlabel(&person[n++]))
          ;
     if (n == MAX)
          printf("Labels full - printing labels\n");
     else --n;      /* EOF encountered for last value of n */

     /* print out the labels */
     printf("\nLabel Data:\n");
     for (i = 0; i < n; i++)
          printlabel(&person[i]);
}
```

Figure 12.8: Driver for Address Label Program

```
Enter Name <First Middle Last>, EOF to quit:  John Paul Jones
Enter Street Address:  23 Dole Street
Enter City State Zip:  Honolulu Hawaii 96822
Enter Name <First Middle Last>, EOF to quit:  David Charles Smith
Enter Street Address:  52 University Ave
Enter City State Zip:  Honolulu Hawaii 96826
Enter Name <First Middle Last>, EOF to quit:  ^D

Label Data:

John Paul Jones
23 Dole Street
Honolulu Hawaii 96822

David Charles Smith
52 University Ave
Honolulu Hawaii 96826
```

Next, let us revise the payroll program so that a payroll data record is stored in a structure called `payrecord`. Let us also define a type called `payrecord` for the structure data type that houses a payroll data record:

```
typedef struct payrecord payrecord;
```

We may, thus, declare variables of type `payrecord` rather than `struct payrecord`. The name for the structure tag and the defined data type can be the same as shown. The structure definitions and typedef are placed in the file `payrec.h` and shown in Figure 12.9.

The program logic is simple enough — it reads input data, calculates payroll data, and prints payroll data as before. In this implementation, we will also include calculation of tax withheld. The result is that we have gross pay, net pay, and tax withheld as additional items in payroll data records as seen in the structure definitions. The program also keeps track of totals for gross and net pay disbursed as well as for taxes withheld. The totals are printed as a summary statement for the payroll. Figure 12.10 shows the main driver.

The function `readrecords()` reads the input data records into an array and returns the number of records read, `printrecords()` prints all payroll data records, and `printsummary()` prints the totals of gross pay and taxes withheld. Finally, we need `calcrecords()` to calculate pay for each of the records. These functions are shown in Figures 12.11 and 12.12.

In the code, we use functions `readname()` and `printname()` to read and print an individual name for each record. Finally, we must write `calcrecords()` which calculates the pay for each data record and the totals of gross pay and tax withheld. The tax is calculated on the following basis:

If the total pay is \$500 or less, the tax is 15%;

If the total is \$1000 or less, the tax is 28%;

If the total is over \$1000, the tax is 33%.

```
/*   File: payrec.h */
/*   This file contains structures and data type definitions needed for
     the program in file payrec.c.
*/
struct namerecd {
     char last[15];
     char first[15];
     char middle[15];
};

struct payrecord {
     int id;
     struct namerecd name;
     float hours;
     float rate;
     float regular;
     float overtime;
     float gross;
     float tax_withheld;
     float net;
};

typedef struct payrecord payrecord;
```

Figure 12.9: Data Structure Definitions for Payroll Program

```
/*   File: payrec.c
     Header Files: payrec.h
     This program computes payroll and prints it. Each data record is
     a structure, and the payroll is an array of structures. Tax is
     withheld 15% if weekly pay is below 500, 28% if pay is below 1000,
     and 33% otherwise. A summary report prints out the total gross
     pay and tax withheld.
*/
#include <stdio.h>
#include "payrec.h"
#define MAX 10

void printsummary(double gross, double tax);
int readrecords(payrecord payroll[], int lim);
void printrecords(payrecord payroll[], int n);
double calcrecords(payrecord payroll[], int n, double * taxptr);

main()
{    int i, n = 0;
     payrecord payroll[MAX];
     double gross, tax = 0;

     printf("***Payroll Program***\n\n");
     n = readrecords(payroll, MAX);
     gross = calcrecords(payroll, n, &tax);
     printrecords(payroll, n);
     printsummary(gross, tax);
}
```

Figure 12.10: Driver for Payroll Program

```
/* File: payrec.c - continued */
/* Function prints total gross pay and total tax withheld. */
void printsummary(double gross, double tax)
{
   printf("\n***SUMMARY***\n\n");
   printf("TOTAL GROSS PAY = $%8.2f; TOTAL TAX WITHHELD = $%8.2f\n",
         gross, tax);
}

/* Function reads payroll input data records until EOF or until lim
records have been read.
*/
int readrecords(payrecord payroll[], int lim)
{  int i, n, x;
   float z;
   void readname(payrecord payroll[], int i);

   for (i = 0; i < lim; i++) {
      printf("Id Number/EOF: ");
      x = scanf("%d%*c", &n);
      if (x == EOF) return i;
      payroll[i].id = n;
      readname(payroll, i);
      printf("Hours Worked: ");
      x = scanf("%f%*c", &z);
      payroll[i].hours = z;
      printf("Rate of Pay: ");
      x = scanf("%f%*c", &z);
      payroll[i].rate = z;
   }
   return i;
}

/* Function reads a single name. */
void readname(payrecord payroll[], int i)
{
   printf("Last Name: ");
   gets(payroll[i].name.last);
   printf("First Name: ");
   gets(payroll[i].name.first);
   printf("Middle Name: ");
   gets(payroll[i].name.middle);
}
```

Figure 12.11: Code for Payroll Program Functions

```
/* Prints a single name. */
void printname(payrecord payroll[], int i)
{
   printf("Name: %s %s %s\n", payroll[i].name.first,
                       payroll[i].name.middle,
                       payroll[i].name.last);
}

/* Function prints n payroll records. */
void printrecords(payrecord payroll[], int n)
{  int i, x;
   float z;
   void printname(payrecord payroll[], int i);

   printf("\n***PAYROLL REPORT***\n\n");
   for (i = 0; i < n; i++) {
      printf("\nId Number: %d\n", payroll[i].id);
      printname(payroll, i);
      printf("Hours Worked: %8.2f  ", payroll[i].hours);
      printf("Rate of Pay: %8.2f\n", payroll[i].rate);
      printf("PAY\n");
      printf("Regular: %8.2f, Overtime = %8.2f, ",
               payroll[i].regular, payroll[i].overtime);
      printf("Gross = %8.2f, Net = %8.2f\n",
               payroll[i].gross, payroll[i].net);
      printf("TAX Withheld = %8.2f\n", payroll[i].tax_withheld);
   }
}
```

Figure 12.12: Code for Payroll Program Functions — continued

The function also keeps a cumulative sum of total gross pay and total tax withheld. Finally, it returns total gross pay and indirectly returns the total tax withheld. The code is shown in Figure 12.13. Here is a sample interaction with the program:

```
***Payroll Program***

Id Number/EOF: 17
Last Name:  Young
First Name:  Cyrus
Middle Name:  Lee
Hours Worked:  38
Rate of Pay:  12
Id Number/EOF: 10
Last Name:  Jones
First Name:  John
Middle Name:  Paul
Hours Worked:  50
Rate of Pay:  16.5
Id Number/EOF: ^D

***PAYROLL REPORT***

Id Number:  17
Name:  Cyrus Lee Young
Hours Worked:  38.00 Rate of Pay:  12.00
PAY
Regular:  456.00, Overtime = 0.00, Gross = 456.00, Net = 387.60
TAX Withheld = 68.40

Id Number:  10
Name:  John Paul Jones
Hours Worked:  50.00 Rate of Pay:  16.50
PAY
Regular:  660.00, Overtime = 247.50, Gross = 907.50, Net = 653.40
TAX Withheld = 254.10

***SUMMARY***

TOTAL GROSS PAY = $ 1363.50; TOTAL TAX WITHHELD = $ 322.50
```

12.3 Sorting Arrays of Structures

We can make one more small improvement to our address label program. Often when we want to print labels, we would like to print them in some sorted order. In this section we will write a function to sort the array of `label` structures. As we saw in Chapter 10, an array is sorted by some *key*, that is, for an array of structures, by a specific member of the structure. A list of

```
/* File: payrec.c - continued */
/* This function computes regular and overtime pay, and the tax to be
   withheld. Tax withheld is 15% of gross pay if not over $500, 28% of
   gross if not over $1000, and 33% of gross otherwise. The function also
   cumulatively sums total gross pay and total tax withheld.
*/
double calcrecords(payrecord payroll[], int n, double * taxptr)
{  int i;
   double gross = 0;
   *taxptr = 0;

   for (i = 0; i < n; i++) {
      if (payroll[i].hours <= 40) {
         payroll[i].regular = payroll[i].gross =
            payroll[i].hours * payroll[i].rate;
         payroll[i].overtime = 0;
      }
      else {
         payroll[i].regular = 40 * payroll[i].rate;
         payroll[i].overtime = (payroll[i].hours - 40) * 1.5 *
                        payroll[i].rate;
      }
      payroll[i].gross = payroll[i].regular + payroll[i].overtime;
      if (payroll[i].gross <= 500)
         payroll[i].tax_withheld = 0.15 * payroll[i].gross;
      else if (payroll[i].gross <= 1000)
         payroll[i].tax_withheld = 0.28 * payroll[i].gross;
      else
         payroll[i].tax_withheld = 0.33 * payroll[i].gross;
      gross += payroll[i].gross;
      *taxptr += payroll[i].tax_withheld;
      payroll[i].net = payroll[i].gross - payroll[i].tax_withheld;
   }
   return gross;
}
```

Figure 12.13: Code for `calcrecords()`

```
/* File: lblutil.c - continued */
/* Sorts an array of labels person[], of size n, by last name
   using an array of pointers plabel[]. */
void sortlabels(struct label person[], struct label *plabel[], int n)
{    int i;

     for (i = 0; i < n; i++)
          plabel[i] = person + i;
     sortptrs(plabel, n);
}

/* Sorts pointers to labels by last name */
void sortptrs(struct label *plabel[], int n)
{    int j, maxpos, eff_size;
     struct label *ptemp;

     for (eff_size = n; eff_size > 1; eff_size--) {
          maxpos = 0;
          for (j = 0; j < eff_size; j++)
               if (strcmp(plabel[j]->name.last,
                         plabel[maxpos]->name.last) > 0)
                    maxpos = j;
          ptemp = plabel[maxpos];
          plabel[maxpos] = plabel[eff_size-1];
          plabel[eff_size-1] = ptemp;
     }
}
```

Figure 12.14: Utility Functions to Sort `label` Structures

labels may be sorted either by last name, or by zip code, or by street address, and so forth. Again, considering the sorting algorithms in Chapter 10, we saw that sorting involves swapping data items to place them in the correct order. However, like passing structures to functions, swapping entire structures can be inefficient if the structures are large. In addition, it is common that an array of structures needs to be sorted by different keys for different purposes. To solve these problems, we can use a technique we used in Chapter 9 for sorting two dimensional arrays — sorting the data using an array of pointers. In this way, the swapping operations while sorting involve only pointers, not entire records, and we can maintain several such pointer arrays, each sorted by a different key.

Figure 12.14 shows the code for the function `sortlabels()` added to the file `lblutil.c` which sorts labels by last name using pointers. (The function assumes the `label` structure defined in `lbl.h` — Section 12.1.4). The function `sortlabels()` is passed the array of labels, `person[]` and an array of pointers to `label` structures, `plabel[]`. This array should be declared in `main()` as:

```
struct label *plabel[MAX];
```

and passed to `sortlabels()` in the call:

```
    sortlabels(person,plabel,n);
```

after the `person[]` array is read. The function begins by initializing the elements of `plabel[]` to point to successive elements of the array of structures, `person[]`. It then calls `sortptrs()` to sort the array by last name using these pointers using a selection sort algorithm. The only thing to note is that for the comparison step of the sort, a structure element is accessed by:

```
    plabel[j]->name.last
```

which accesses the `last` field of the `name` field of the object pointed to by `plabel[j]`. In the swap step of the sort algorithm, only the pointers are swapped.

We can now write a function, `printsortedlabels()`, to print the labels in sorted order using the `plabel[]` array, modifying `main()` appropriately. We leave this as an exercise.

The utility functions in the file `lblutil.c` provide most of the tools needed to write a useful, interactive address label data base program. In the next chapter, we discuss the remaining piece — file storage for the data base, and write the entire application.

12.4 Unions

In some applications, we might want to maintain information of one of two alternate forms. For example, suppose, we wish to store information about a person, and the person may be identified **either** by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. (This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company). In addition, we wish to use the same member name to access identity the information for a person.

C provides a data structure which fits our needs in this case called a *union* data type. A union type variable can store objects of different types at different times; however, at any given moment it stores an object of only one of the specified types. The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure template. For example, we can declare a union variable, `person`, with two members, a string and an integer. If the name is entered, we will use person to store the string; if an identification number is entered, we will use person to store an integer. Here is the union declaration:

```
    union {
         int id;
         char name[25];
    } person;
```

This declaration differs from a structure in that, when memory is allocated for the variable `person`, only enough memory is allocated to accommodate the **largest** of the specified types. The memory allocated for `person` will be large enough to store the larger of an integer or an 25 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by name:

```
union human {
     int id;
     char name[25];
} person;
```

Likewise, it is possible to declare just a tag, and later, use the tag to declare variables:

```
union human {
     int id;
     char name[25];
};
union human person, *ppers;
```

The syntax for declaring a union type is basically the same as for a structure:

```
union [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];
```

The members of a union variable may be accessed in the same manner as are members of a structure variable:

```
<union_var>.<member>
<ptr_to_union_var> − ><member>
```

Examples include:

```
ppers = &person;
person.id = 12;
if (ppers->id == 12)
     ...
printf("Id = %d\n", person.id);
```

The type of data accessed is determined by the member name used to qualify the variable name. In our example, `person.id` will access an integer; while `person.name` will access a string (a character pointer).

Since at any given time, the contents of the union variable may be one of several types (`int` or string for `person`), we must keep track what type of data is stored in order to access the information correctly. Each time an object is stored in a union variable, it is the programmer's responsibility to keep track of the type stored. If an attempt is made to retrieve a type different from the type last stored, the result is sure to be strange and incorrect. The specific behavior is implementation dependent.

To remember the type of object last stored in a union variable, it is common to store `that` information in a variable. The best way is to declare a structure containing both the union variable as a field and another field that indicates the type of data stored in the union. For example, we can declare such a structure type and a structure array as follows:

```
/*   File: uniutil.c */
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "unidef.h"
#include "uniutil.h"
/*   Reads a list of items. Each item is either a string
     or an integer.
*/
int readlist(struct record list[], int lim)
{    int i;
     char s[SIZE];

     printf("Type Identifications For Persons on the List\n");
     printf("Either a Name or an Id Number, EOF to quit\n");
     for (i = 0; i < lim && gets(s); i++) {
          if (isdigit(*s)) {               /* Is it a number? */
               list[i].ptype = INT;        /* If so, store type, */
               list[i].person.id = atoi(s);  /* and the ID number. */
          }
          else {                           /* Otherwise, */
               list[i].ptype = NAME;       /* Store string type, */
               strcpy(list[i].person.name, s); /* and the NAME. */
          }
     }
     return i;                             /* Return no. of items. */
}
```

Figure 12.15: Reading Data into a Union Variable

```
#define NAME 0
#define ID 1
struct record {
     int ptype;
     union human person;
};
struct record list[MAX];
```

Now, as we read information about each element of list, if the information is numeric, we store it as `id`; otherwise, we store it as `name`. We also store the type, `ID` or `NAME` in the member, `ptype`.

Figure 12.15 shows a function that reads identifying information about each person and stores it in the union type member. Depending on the type of information read, it uses the appropriate union field name, and stores the type in the `ptype` field of the structure. The loop body in the function looks at the first character of the input string, `s`. If it is a digit, then the data is an id number so `INT` is stored in `ptype`, and the string is converted to an integer (using `atoi()`) and

```
/*   File: uniutil.c - continued */
/*   Prints out the list of items. Each item is either a string
     or an integer.
*/
void printlist(struct record list[], int n)
{    int i;

     printf("Identifications of People on the List\n");
     printf("Either a Name or an ID Number\n");
     for (i = 0; i < n; i++) {
          if (list[i].ptype == INT)
               printf("Id number: %d\n", list[i].person.id);
          else
               printf("Name: %s\n", list[i].person.name);
     }
}
```

Figure 12.16: Printing Information from a Union Variable

stored in the union `id` field. If the first character of `s` is not a digit, `NAME` is stored in `ptype`, and the string is copied into the union `name` field.

It is now easy to write a function that prints the identifying information stored in the list. Since each record includes the type of information stored in the union variable, it is easy to retrieve the information correctly as shown in Figure 12.16.

We now write a simple program that first reads a list of identifying information about a group of people, and later prints the list. The identifying information may be either a name or an id number. The structure and union declarations as well as constant definitions are included in the file `unidef.h` shown together with the code in Figure 12.16.

Sample Session:

```
***Union Variables - Lists***

Type Identifications For Persons on the List
Either a Name or an Id Number, EOF to quit
John Kent
345
Jane Ching
231
Mary Smith
^D
Identifications of People on the List
Either a Name or an ID Number
Name:  John Kent
Id number:  345
Name:  Jane Ching
```

```
/*   File: unidef.h */
#define INT 0
#define NAME 1
#define MAX 10
#define SIZE 25
union human {
     int id;
     char name[SIZE];
};
struct record {
     int ptype;
     union human person;
};

/*   File: uniutil.h */
int readlist(struct record list[], int lim);
void printlist(struct record list[], int n);

/*   File: union.c
     Other Source Files: uniutil.c
     Header Files: unidef.h, uniutil.h
     This program illustrates the use of union variables. It reads
     a list of items identifying people either by name or by id
     number. It then prints out the list. Each item is stored in
     a union variable either as a name or as an integer. The list
     is kept in an array of structure record. The structure record
     has two members, the union variable and a variable that stores
     the type of object stored in the union.
*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "unidef.h"
#include "uniutil.h"
main()
{    struct record list[MAX];
     int n;

     printf("***Union Variables - Lists***\n\n");
     n = readlist(list, MAX);
     printlist(list, n);
}
```

Figure 12.17: Header File and Driver Program for Union Example

```
Id number:  231
Name:  Mary Smith
```

The above program can be written in many alternate ways. We have written the program to illustrate the use of union variables.

12.5 Common Errors

Common errors occur when pointers are used to reference structures and their members. It is best to use parentheses around dereferenced pointers, `(*p).member`, or to use the operator, `->`, e.g. `p->member`, when referencing a member of a structure pointed to by a pointer.

12.6 Summary

In this chapter, we have described the last remaining data types provided by the C language: structures and unions. A structure allows the grouping of various pieces of related information of different types into one variable. It is declared by defining a *template* specifying the type of each data item stored in the structure and giving each *member* or *field* a name:

```
struct [<tag_identifier>] {
<type_specifier> <identifier>;
<type_specifier> <identifier>;
...
} [<identifier>[, <identifier>...]];
```

Variables may be declared when the template is defined or, if a *tag* is used to name the template, may be declared later using the tag:

```
struct <tag_identifier> <identifier>[, <identifier>...];
```

which allocates storage for all members. Fields of a structure variable may be accessed using the "dot" (.) operator:

```
<variable_identifier>.<member_identifier>
```

called *qualifying* the variable name. Such qualified structure variable expressions may be used like the corresponding field type in a program. Structure variables may be passed to and returned from functions, but it is more common to use pointers to structures to avoid excessive copying. Members of a structure can be accessed with a pointer using the $->$ operator:

```
<variable_identifier> - ><member_identifier>
```

which is equivalent to:

```
(* <variable_identifier>).<member_identifier>
```

We have illustrated the use of structures with various programming examples.

Finally, we have described the union data type, which is defined similar to structures; however, has the semantics of only one of the member types being resident in such a variable at one time. That is, a union allows several different types of information to be stored in the same physical space at different times. For a union variable, storage is allocated only for the largest of the data types which may reside in the variable.

Structures are a valuable tool for developing complex programs and data structures in an efficient and top down manner.

12.7 Exercises

1. Find and correct errors if any. What will be the output?

```
struct node {
     int id;
     int score;
}

#include <stdio.h>
main()
{    struct node *px, x, y;
     px = &x;
     while (scanf("%d %d", px.id, px.score) != EOF)
          printf("%d   %d\n", *px.id, *px.score);
}
```

2. Define a data structure, `intflt`, that will allow one to store either an integer or a float. Read strings and convert them to either integers or floats depending on whether there is a fractional part present. Store the resulting values in an `intflt` type array. When the input is terminated, print the stored values.

12.8 Problems

1. Write the function `printsortedlabels()` described in Section 12.3 and make the modifications to `main()` to read a list of address labels and print them in sorted order by last name.

2. Define a structure with the following members:

   ```
   social security number
   id number
   name (last, first, middle)
   exam score
   ```

   Use the above structure for the data record of one student in a class of 50 students maximum. Write a menu-driven program that allows the commands: read data from an input file into an array of the above structure; print data on screen; save data into an output file; sort the data by a specified primary key using pointers to the array; quit.

3. Modify Problem 2 to allow more than one exam up to a maximum of 5 exams. Use an array of exam scores in the structure. Assume that the first three lines of the input file include course title and headings. The actual data starts with the fourth line.

4. Modify Problem 3 to compute and store a weighted average of the exam scores for each student. Weighted average should be a member of the structure. Also allow for computation of an average of any one or all the exams.

5. Modify Problem 4 to allow deleting one or more records, changing one or more records, adding one or more records.

6. Modify Problem 5 so that it can read an input file which may or may not contain a column for the weighted average. Allow the user to output the data but specify which data fields are not to be output to a new file.

7. Modify the above program to include scores for a number of projects up to a maximum of 15. Weighted average must now include exam as well as project scores. Allow a structure member for a letter grade.

8. Modify the above program so it allows the user to perform the following functions: form a class grade list for a new class; enter grades for a project or an exam; change grades for a project or an exam; add or delete a student from a class list; calculate the average for a project or an exam; calculate the weighted average for each student over the projects and the exams; sort the data by a primary key, e.g. weighted average, exam2, proj3, etc.; sort the data by a primary key and a secondary key, i.e. if two records have the same primary key, then sort those records by a secondary key; plot a distribution of the weighted average grades.

9. Write a program that keeps a membership list for a private club. The data fields required are:

```
name
spouse name if any
address, business, residence
telephone, business, residence
hobby interests
membership date
dues outstanding
other charges outstanding
```

The club has a limit of 100 members. Write a program that allows the club manager to: maintain the club list and update it; send out a mailing list to all members with all data about the club members, except for financial data; send out reminders to members about the charges outstanding; post new charges and dues at regular intervals; post paid amounts upon receipt.

10. Assume that the above club maintains a library of at most 500 books. Data for each book consists of:

```
book number
title
author
co-authors
publisher
date published
subject
keywords
check out data:
     name, address, phone
     date checked out
     data returned
     charges, if any
```

Write a program to maintain the library including: search the library by book number, author, title, subject, keywords; add new books, remove outdated books (all books older than 5 years); check out books; late charges at $0.25 per day if a book is out by more than a month; write data to a file for books overdue and charges.

11. Write a macro processor assuming that the macros do not have arguments. Use a structure to keep a macro identifier and its replacement string. Read an input file which may have macros, and create an output file with macros replaced by replacement strings.

12. Write a macro processor. Assume that macros may have arguments. Use structures to keep data about a macro.

13. Use a structure to represent a rational number. Write functions for rational number arithmetic. Write a simple calculator program for rational numbers.

Chapter 13

Files and the Operating System

So far, our interaction with the Operating System of the computer has been limited to using the compiler and shell to create executable files from our programs and execute them, as well as limited interaction with the File System to provide input data to our programs and store the output of results. All of our file I/O, either redirected standard input and output or direct using library functions such as `fscanf()` and `fprintf()`, has been with ASCII files using the formatted I/O utilities provided in C. In this chapter we look at an alternate method of doing I/O — block I/O, where a binary image of a data structure can be stored or retrieved. We discuss the library routines for performing block I/O and managing access to such files. We then provide an example program, a small data base system, which makes use of this facility.

Finally, we discuss other facilities provided by the C library for interacting with the shell from within a program, such as executing a shell command and command line arguments for our program.

13.1 Block Input/Output

The file I/O functions discussed so far perform reading and writing of different types of data using *formatted* ASCII information stored in files. Each I/O operation acts on a stream of character bytes, and the appropriate conversions from characters to an internal representation is performed by the library routines. While it is convenient to have data stored in files in an ASCII form (such data can be read or written by other programs and devices such as text editors, printers, etc.), it can be tedious and inefficient to perform all that data conversion from ASCII to internal binary, and back to ASCII); particularly for structure type data with many members of different types.

C provides additional file I/O library functions which allow direct input or output of the binary, internal representation of data to files. This form of I/O is called *block I/O*, because data is transferred in blocks directly from the file to storage locations in memory with no conversion. It should be noted that the files that store such data are **binary** files and cannot be read or written directly by other operating system programs such as text editors or printer software. Only a program which knows the organization of binary information within the files can access them correctly.

The library functions provided for this type of I/O are `fwrite()` and `fread()`. The function `fwrite()` writes (i.e. appends) a block of data of specified size to a file. Similarly, `fread()` reads a block of data of specified size into a memory location. The prototypes for these functions are defined in `stdio.h` and they may be described as:

fread *Prototype:* `size_t fread(void *buffer, size_t size, size_t no_items, FILE *fp);` *in:* `<stdio.h>`

Returns: actual number of items read (may be less than `no_items`; `NULL` if error or end of file.

Description: This function reads `no_items`, each of size, `size`, bytes from stream, `fp`, into `buffer`.

fwrite *Prototype:* `int fwrite(const void *buffer, int size, int no_objs, FILE *stream);` *in:* `<stdio.h>`

Returns: the number of objects written if successful; less than `no_objs` on error.

Description: This function writes (appends) `no_objs` objects of size, `size`, from `buffer` to `stream`.

The function prototypes use the data type, `size_t`, defined in `stdio.h`, which is of an unsigned type. (This is the type actually returned by the `sizeof operator`. As stated above, `fread()` returns the number of items read. When an end of file is encountered before `no_items` items are read, the return value will be less than the number requested; so this may be used to indicate end of file. Also note that the first parameter of the prototype for `fwrite()` uses a `const` qualifier. This ensures that no attempt is made to change the object pointed to by `buffer` within the function. Many prototypes use `const` qualifiers in the parameter declarations to prevent unplanned changes.

To see how to use these functions, let us use them to copy a file. The program logic is straight forward: open the input and output files, read each block of characters from the input file into a buffer, write each block to the output file. When a short size block is read, terminate the loop, write the short block, and close the files. The code is shown in Figure 13.1. We have declared a buffer of type `signed char` so it can store any arbitrary bytes. We have also declared a pointer, `ptr`, with the qualifier `const`, since it is to remain unchanged, and initialized it to point to the buffer, `buf`. After the files are opened, the while loop reads a block of 100 items of `char` size from the input file, and writes the block to the output file. The loop is terminated when less than 100 items are read, indicating less than 100 items were remaining in the file to be read, so the end of file has been reached. The number of items read are assigned to n. At this point, the number of data bytes in the buffer is stored in `n`, so writing a block of 100 items would result in some garbage output. (The data from the previous block is still present in the rest of the buffer). Instead, the final block of `n` items is written.

This program simply copies blocks of 100 bytes at a time from the input file to the output file. These files can be any type of file, such as text files, program files, other ASCII files, even binary files, at least on Unix systems. However, on some non-Unix systems, the system routines may not be able to read or write arbitrary binary information unless the mode strings passed to `fopen()` explicitly indicates opening a file for binary I/O by appending the character `'b'` to the string. Thus, the mode strings must be `"rb"` or `"wb"` instead of `"r"` or `"w"`. A program using block I/O to copy binary files on an IBM PC using DOS operating system and a TURBO C compiler is shown in Figure 13.2. The program is the same as before except for the mode strings used in function calls to `fopen()`.