Chapter 6

Pointers

In the preceding chapters, our programs have been written to access objects directly, i.e. using the variable names. We have postponed until now a discussion of the concept of **indirect access**, i.e. access of objects using their address. As we have seen, variables local to a function may be accessed using their name *only* within that function. When arguments are passed to another function, only the values are passed, and the *called* function may use these values, but cannot affect the variable cells in the *calling* function. Sometimes, however, a function needs to have direct access to the cells in another function. This can be done in C through indirect access, using the address of the cell, called a **pointer**.

In this chapter, we will introduce the concepts of indirect access, pointer types, and dereferenced pointer variables. We will use these concepts to write functions that indirectly access objects in a calling function.

6.1 What is a Pointer?

Frequently, a *called* function needs to make changes to objects declared in the *calling* function. For example, the function, `scanf()`, needs to access objects in the calling function to store the data read and converted into an object defined there. Therefore, we supply `scanf()` with the *address* of objects rather than their values. Here, we will see how any function can indirectly access an object by its address.

Another common use of pointers is to write functions that "return" more than one value. As we have seen, every function in C returns a value as the value of the function; however, if a function's *meaning* includes the return of several pieces of information, this single return value is not sufficient. In these cases, we can have the function return multiple data values indirectly, using pointers.

6.1.1 Data vs Address

Before we discuss passing pointers and indirectly accessing data between functions, let us look at how we can declare pointer variables and access data using them. Consider the following simple program:

```
main()
{    int x;
     int iptr;

     printf("***Testing Pointer Variables***\n");
     x = 10;
     iptr = &x;
     printf("%d\n",iptr);

}
```

We have declared two integers, `x`, intended to hold an integer value, and `iptr` which is intended to hold a pointer to an integer, i.e. and address of an integer. We then assign a value to `x`, and the address of `x` to the variable `iptr` using the `&` (address of) operator. The address of a variable is simply the byte address of the cell which was allocated by the declaration. An address is an integer (actually and unsigned integer) so may be stored in an `int` type variable. The situation is shown in Figure 6.1a). When we compile and execute this program the result is:

```
***Testing Pointer Variables***
1000
```

What if we had wanted to print the value of the cell pointed to by `iptr` and not the value of `iptr` itself? The **indirection operator**, `*`, accesses the object pointed to by its operand. In our example, the value of `iptr` is 1000 which is an address of some object; i.e. `iptr` points to *some* object located at address 1000. So we should be able to access that object with an expression like:

```
*iptr
```

However, there is no way to know how many bytes to access at address 1000, nor how to interpret the data, unless the type of object at address 1000 is known: is it an `int`? a `float`? a `char`? etc. In order for the compiler to know how to access an object indirectly, it must know the type of that object. We specify the type of object to access by indicating to the compiler the type of objects a pointer refers to when we declare the pointer. So, in our example, we should declare the variable, `iptr` as a "pointer to an integer" as follows:

```
int *iptr;
```

or,

```
int * iptr;
```

(white space may separate the operator, `*`, and the variable name, `iptr`). The declaration specifies a variable, `iptr`, of type `int *`, i.e. integer pointer (the type is read directly from the declaration). So, `int *` is the type of `iptr`, and `int` is the type of `*iptr` — the thing it points to. This statement declares an integer pointer variable, `iptr`, and allocates memory for a pointer variable. Similarly, we can declare `float` pointers or character pointers:

```
float * pa, * pb;
char  * pc;
```

These statements declare variables, `pa` and `pb`, which can point to `float` type objects, and `pc` which can point to a `char` type object. All pointer variables store addresses, which are unsigned integers, and so need the same amount of memory space regardless of the pointer types.

Since the compiler now knows that `iptr` points to an integer object, it can access the object correctly. Our simple program becomes:

```
main()
{    int x;
     int *iptr;

     printf("***Testing Pointer Variables***\n");
     x = 10;
     iptr = &x;
     printf("Address %d holds value %d\n",iptr,*iptr);

}
```

which produces the output:

```
***Testing Pointer Variables***
Address 1000 holds value 10
```

We are generally not interested in the *value* of the pointer variable itself; it may even be different each time a program is run. Instead, we are interested in the cell the pointer is pointing to, so we indicate the *value* of a pointer variable in diagrams and program traces using an arrow ($\leftarrow$) as shown in Figure 6.1b.

In summary, the address of an object is called a **pointer** to that object since the address tells one where to go in order to access the object. The address by itself does not provide sufficient

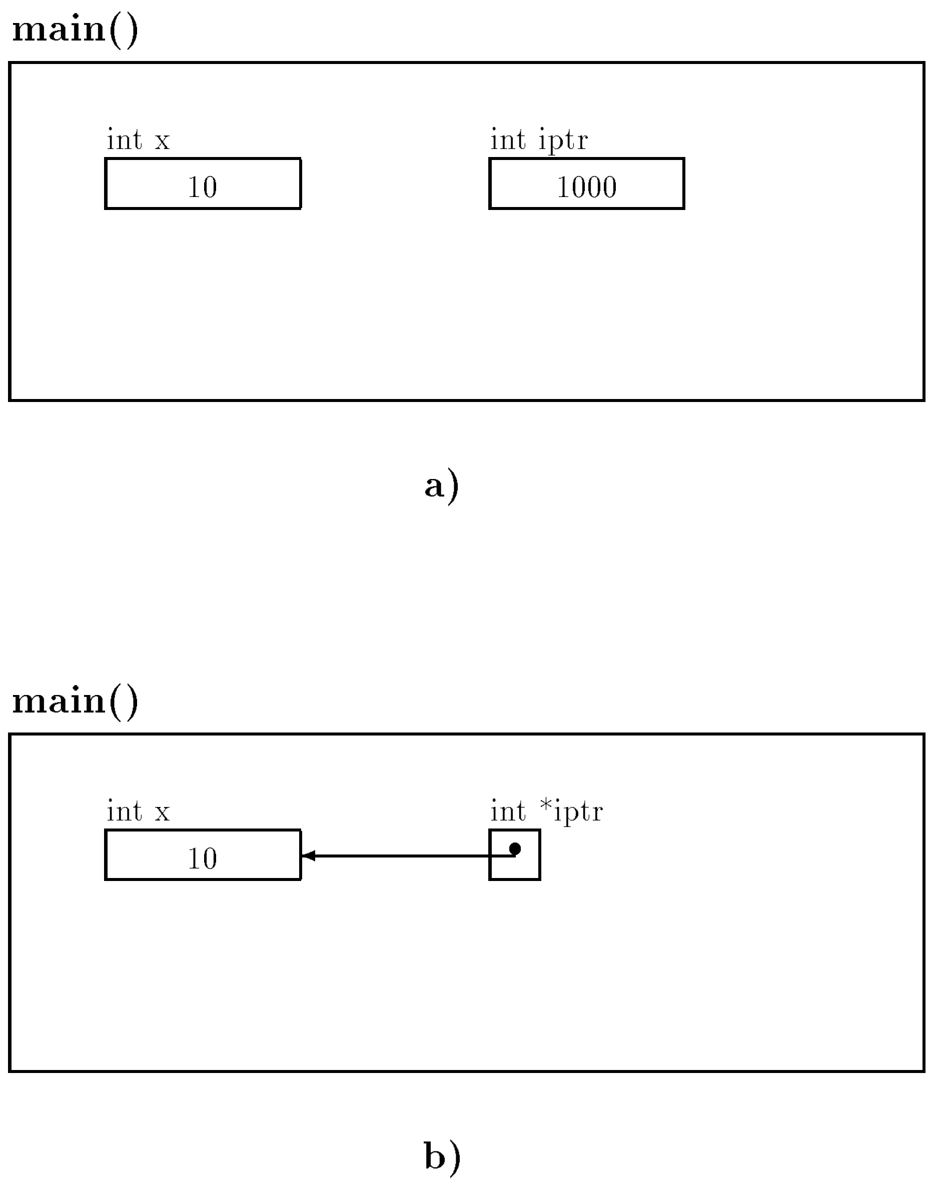

Figure 6.1: Declaring Pointer Variables

information to access an object; we must know what type of object the address is pointing to. If the pointer (address) value and the data type of the object that it points to are both known, then it is possible to access the object correctly. In other words, pointers must be specified to be `int` pointers, pointing to an integer type object, `float` pointers, pointing to a floating point type object, `char` pointers, etc.

6.1.2 Indirect Access of Values

The indirection operator, `*`, accesses an object of a specified type at an address. Accessing an object by its address is called **indirect access**. Thus, `*iptr` indirectly accesses the object that `iptr` points to, i.e. `*iptr` *accesses* `x`. The indirection operator is also called the **contents of operator** or the **dereference operator**. Applying the indirection operator to a pointer variable is referred to as **dereferencing** the pointer variable, i.e. `*iptr` *dereferences* `iptr`. The address of operator, `&`, is used to get the address of an object. We have already used it in calls to `scanf()`. We can also use it to assign a value to a pointer variable.

Let us consider some examples using the following declarations:

```
int x, z;
float y;
char ch, * pch;
int * pi, *pi2;
float * pf;
```

When these declarations are encountered, memory cells are allocated for these variables at some addresses as shown in Figure 6.2. Variables `x` and `z` are `int` types, `y` is `float`, and `ch` is `char`. Pointer variables `pi` and `pi2` are variables that can point to integers, `pf` is a `float` pointer, and `pch` is a character pointer. Note that the initial values of all variables, including pointer variables, are unknown. Just as we must initialize `int` and `float` variables, we must also initialize pointer variables. Here are some examples:

```
x = 100;
y = 20.0
z = 50;

pi = &x;        /* pi points to x */
pi2 = &z;       /* pi2 points to z */
pch = &ch;      /* pch points to ch */
```

The result of executing these statements is shown in Figure 6.3: `pi` points to the cell for the variable `x`, `pi2` points to `z`, `pch` points to `ch`, and `pf` still contains garbage. Remember, the *value* of a pointer variable is stored as an address in the cell; however, we do not need to be concerned with the value itself. Instead, our figure simply shows what the initialized pointer variables point

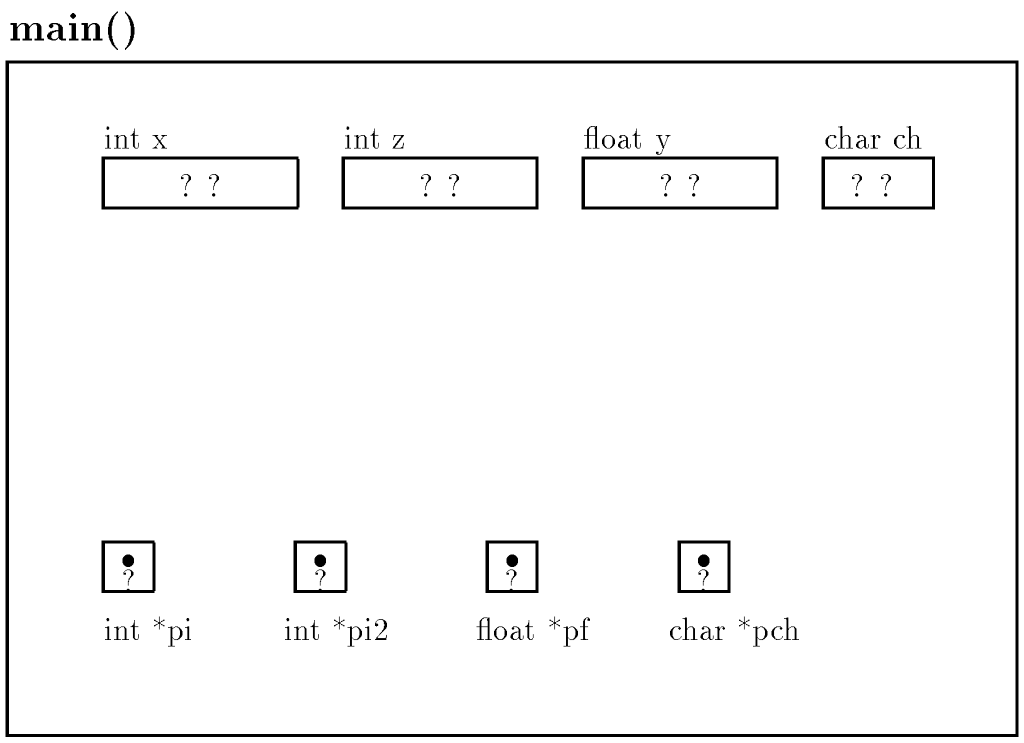

Figure 6.2: Declaration of Pointer Variables

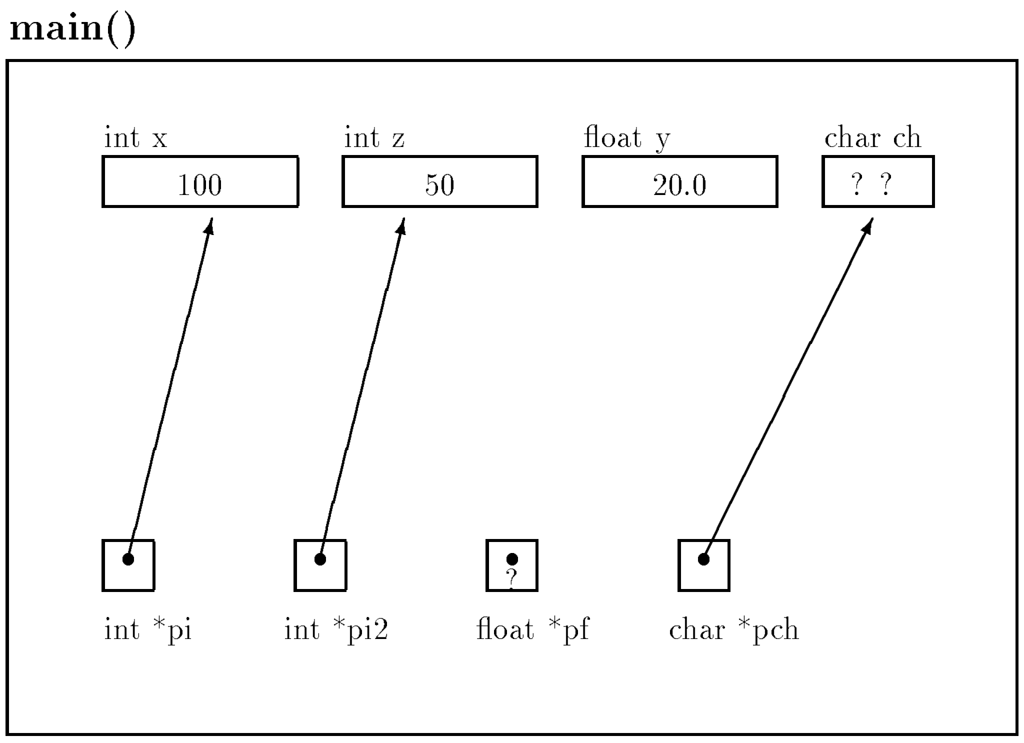

Figure 6.3: Assignments of pointers

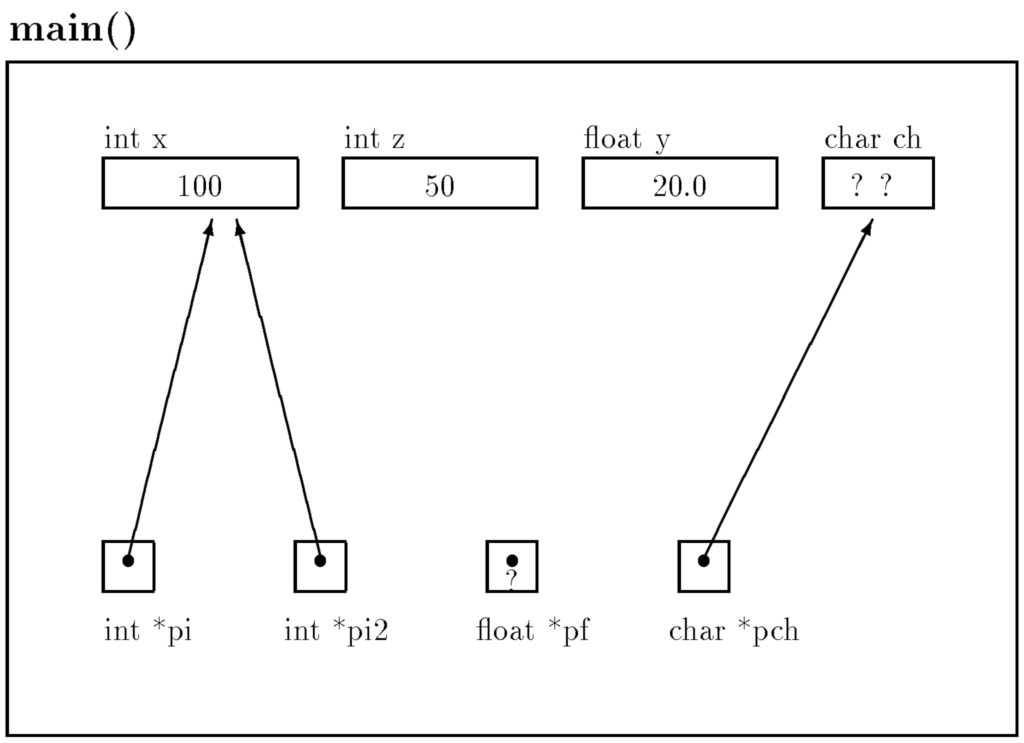

Figure 6.4: Effect of Pointer to Pointer Assignment — Statement 1.

to. These initialized pointers may now be used to indirectly access the objects they point to, or they be may be changed by new assignments. Here are some examples of statements and how they change things for the above memory organization. (The statements are numbered in order to reference them; the numbers are not part of the code).

```
1:   pi2 = pi;                /* pi2 points to where pi points            */
                              /* i.e. pi2 ==> x                           */
2:   pi = &z;                 /* pi now points to z, pi2 still points to x */
                              /* i.e. pi ==> z, pi2 ==> x                 */
3:   *pi = *pi2;              /* z = x, i.e, z = 100                      */
4:   *pi = 200;               /* z = 200, x is unchanged                  */
5:   *pi2 = *pi2 + 200;       /* x = 300, z is unchanged                  */
```

Statement 1: Assigns value of `pi` to `pi2`, so `pi2` now also points to `x` (see Figure 6.4). Since both of the variables are type `int *` this assignment is allowed.

Statement 2: Makes `pi` point to `z` (see Figure 6.5). The expression `&z` evaluates to the address of `z`; i.e. an `int` pointer.

Statement 3: Since `pi2` points to `x`, the value of the right hand side, `*pi2`, dereferences the pointer and evaluates to the value in the cell, i.e. 100. This value is assigned to the object accessed by the left hand side, `*pi`, i.e. the place pointed to by `pi` or the

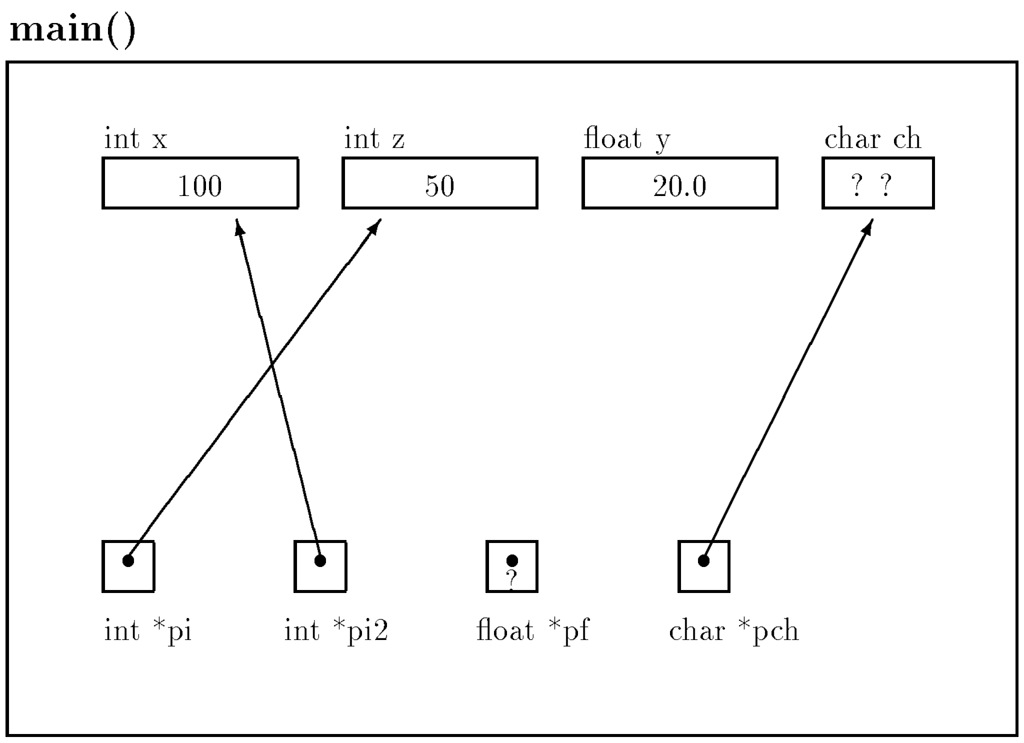

Figure 6.5: Effect of Pointer Reassignment — Statement 2.

object `z` (see Figure 6.6). This has the same effect as the assignment `z = x`. Note, we have used a dereferenced pointer variable as the Lvalue on the left hand side of an assignment operator. The semantics is to access the object indirectly and store the value of the expression on the right hand side.

Statement 4: The value, 200, is assigned to `*pi`, i.e. `z` (see Figure 6.7). Again, we have used an indirect access for the Lvalue of the assignment.

Statement 5: The right hand side evaluates to 300, since 200 is added to `*pi2`; so 300 is assigned to `*pi2`, i.e. `x` (see Figure 6.8). Again, we have used an indirect access on both the left and right hand sides.

We see that the left hand side of an assignment operator, the Lvalue, can be a reference to an object either by direct access (i.e. a variable name) or by indirect access (i.e. a dereferenced pointer variable). Also notice that we were very careful about the *type* of the objects on the left and right hand side of the assignment operators. We have assigned an integer value to a cell pointed to by an integer pointer, and when assigning pointers, we have assigned an integer pointer to a cell declared as an `int *`. An assignment statement such as:

```
pi = x;
```

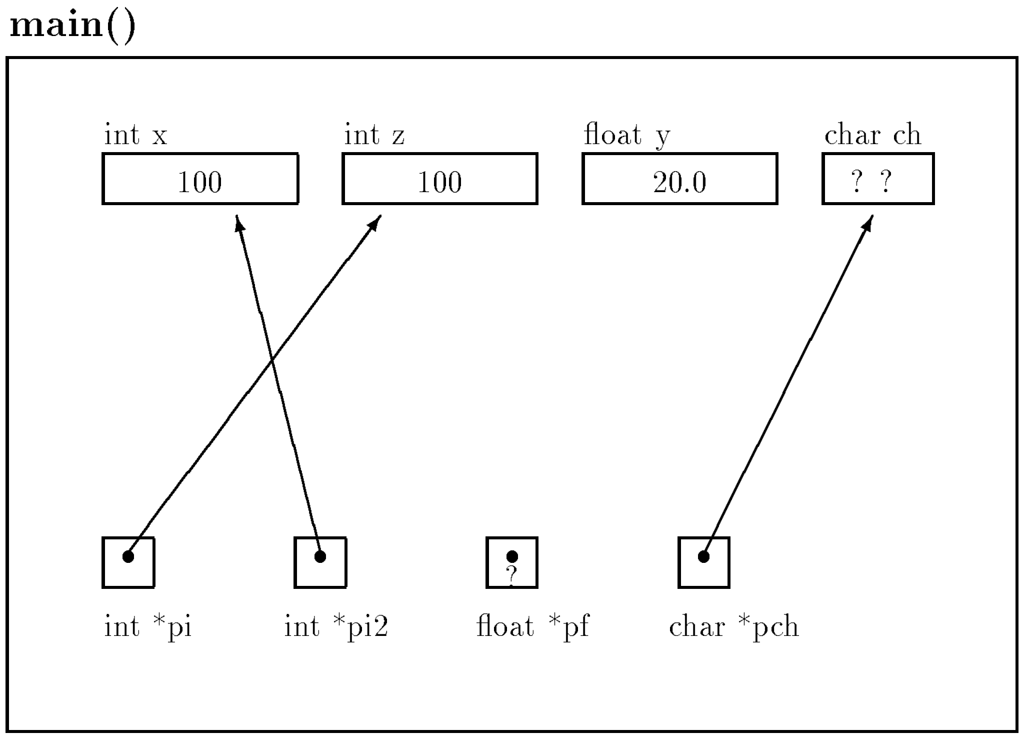

Figure 6.6: Effect of Indirect Pointer Access and Assignment — Statement 3

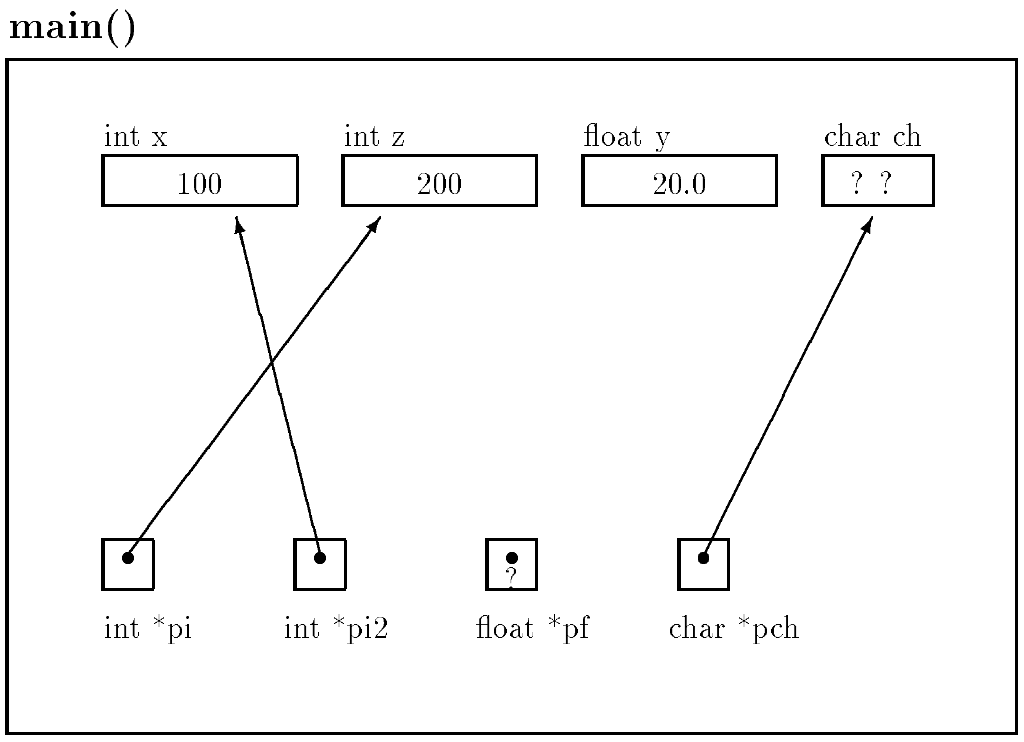

Figure 6.7: Effect of Indirect Assignment — Statement 4

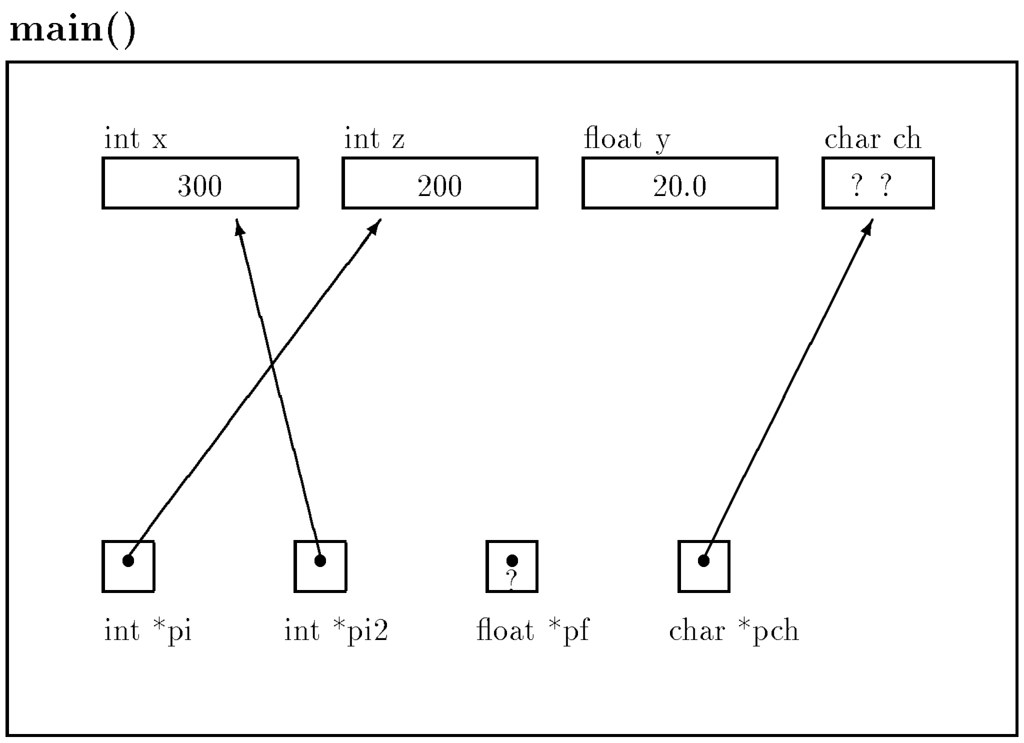

Figure 6.8: Effect of Indirect Pointer Access and Assignment — Statement 5

is a legal statement in C: assigning an integer value to a pointer cell. However, the effect may not be as we would expect. The value of `x` will be placed in the pointer cell, `pi`, and subsequent dereferencing of `pi`, (`*pi`), will use that value as a pointer (an address) to find the cell to indirectly access. This is almost *never* what we intend to do in this statement. Most C compilers will generate a *warning* at compile time stating that an illegal integer-pointer combination in an assignment was encountered to indicate that something is *possibly* wrong here. A warning is not an error; it does not prevent the compiler from generating a functional object file. However, it is an indication that the statement may not be what the programmer intended. Such a statement is probably correctly written as:

```
*pi = x;                 or                 pi = &x;
```

which assign a value to the cell pointed to by `pi` or to assign an address to `pi` itself, respectively. (In the RARE instance where such an assignment of an integer to a pointer cell is intended, the syntax:

```
pi = (int *)x;
```

i.e. casting the integer to an integer pointer, should be used).

Likewise, an attempt to use the uninitialized variable, `pf` will be a disaster. Suppose we write:

```
printf("%f\n", *pf);
```

The value of pf is garbage so *pf will attempt to access the garbage address for a float object. The garbage value of pf may be an invalid memory address, in which case, the program will be aborted due to a *memory fault*; a run time error. This is bad news; however, we may be even more unfortunate if the value in pf is a valid memory address. In this case, we would access a value from some unknown place in memory. The situation is even worse when an uninitialized pointer is used indirectly as an Lvalue:

```
*pf = 3.5;
```

Since we do not know where pf is pointing, if it happens to be a legal address, we have just placed the value, 3.5, in some unknown location in memory, possible a cell belonging to a variable in another part of the program. Finding this type of bug is very difficult. The lesson here is that care should be taken when using pointers, particularly ensuring that pointers are properly initialized.

On the other hand, the character variable, ch, is not initialized, but the pointer variable, pch is initialized to point to ch so the expression, *pch, will access the object, ch, correctly. If the value of *pch is accessed, it will be garbage; but a value can be stored in *pch correctly.

With proper care, the value of an initialized pointer variable (the address of some object) allows us to indirectly access the object by dereferencing the pointer variable. An example program, shown in Figure 6.9, illustrates the *value of a pointer variable* and the *value of the object indirectly accessed* by it.

Figure 6.10 shows program trace graphically. The program first declares an int and an int * variables (Figure 6.10a)). The first printf() statement prints the program title followed by the initialization of i1 and iptr (Figure 6.10b)). The next printf() gives the hexadecimal value of iptr, which is the address of i1. The next statement prints the value of the same object indirectly accessed by *iptr and directly accessed by i1. Then, the value of *iptr is changed (Figure 6.10c)); and the last statement prints the changed value of the object, accessed first indirectly and then directly.

The output for a sample run is:

```
Pointers:  Direct and Indirect Access

iptr = 65490
*iptr = 10, i1 = 10
*iptr = 100, i1 = 100
```

```
/*   File: access.c
     This program prints out the values of pointers and values of
     dereferenced pointer variables.
*/
#include <stdio.h>
main()
{    int *iptr,     /* integer pointer */
          i1;

     printf("Pointers: Direct and Indirect Access\n\n");
     /* initializations */
     i1 = 10;
     iptr = &i1;    /* iptr points to the object whose name is i1 */

     /* print value of iptr, i.e., address of i1 */
     printf("iptr = %u\n", iptr);
     /* print value of the object accessed indirectly and directly */
     printf("*iptr = %d,  i1 = %d\n", *iptr, i1);

     *iptr = *iptr * 10;      /* value of *iptr changed */
     /* print values of the object again */
     printf("*iptr = %d,  i1 = %d\n", *iptr, i1);
}
```

Figure 6.9: Example Code with Direct and Indirect Access

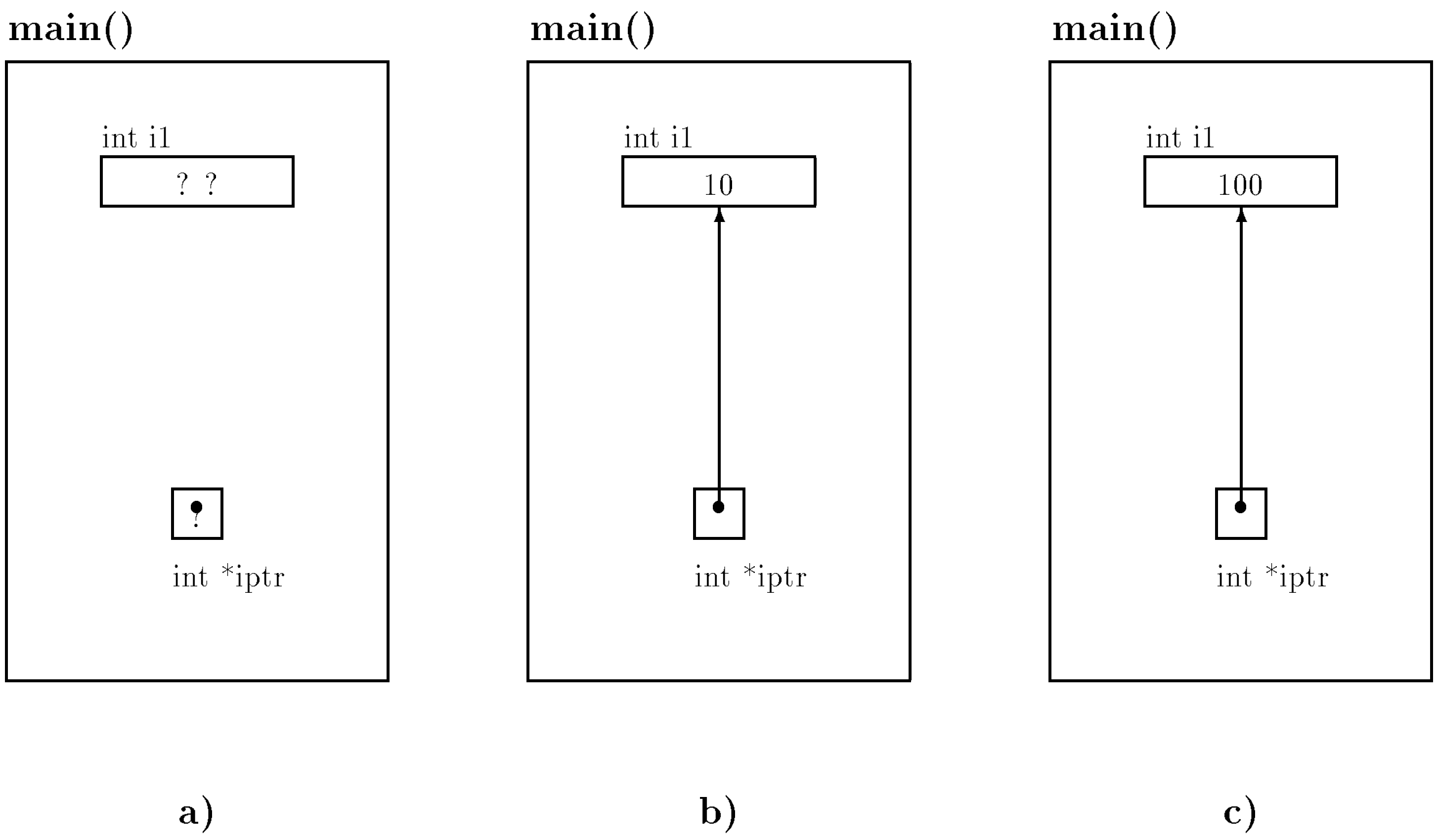

Figure 6.10: Trace for Direct and Indirect Access

6.2 Passing Pointers to Functions

As we have seen, in C, arguments are passed to functions *by value*; i.e. only the *values* of argument expressions are passed to the called function. Some programming languages allow arguments passed *by reference*, which allows the called function to make changes in argument objects. C allows only call by value, not call by reference; however, if a called function is to change the value of an object defined in the calling function, it can be passed a value which is a *pointer* to the object. The called function can then dereference the pointer to access the object indirectly. We have also seen that a C function can return a single value as the value of the function. However, by indirect access, a called function can effectively "return" several values. Only one value is actually returned as the value of the function, all other values may be indirectly stored in objects in the calling function. This use of pointer variables is one of the most common in C. Let us look at some simple examples that use indirect access.

6.2.1 Indirectly Incrementing a Variable

We will first write a program which uses a function to increment the value of an object defined in `main()`. As explained above, the called function must indirectly access the object defined in `main()`, i.e. it must use a pointer to access the desired object. Therefore, the calling function must pass an argument which is a pointer to the object which the called function can indirectly access.

Figure 6.11 shows the code for the program and the program trace is shown graphically in Figure 6.12. The function, `main()` declares a single integer variable and initializes it to 7 (see Figure 6.12a)). When `main()` calls `indirect_incr()`, it passes the pointer, `&x` (the address of `x`). The formal parameter, `p`, is defined in `indirect_incr()` as a pointer variable of type `int *`. When `indirect_incr()` is called, the variable, `p` gets the value of a pointer the the cell named `x` in `main()` (see Figure 6.12b)). The function, `indirect_incr()`, indirectly accesses the object pointed to by `p`, i.e. the `int` object, `x`, defined in `main()`. The assignment statement indirectly accesses the value, 7, in this cell, and increments it to 8, storing it indirectly in the cell, `x`, in `main()` (see Figure 6.12c)).

Sample Session:

```
***Indirect Access***
Original value of x is 7
The value of x is 8
```

6.2.2 Computing the Square and Cube

Sometimes, whether a value should be returned as the value of a called function or indirectly stored in an object is a matter of choice. For example, consider a function which is required to "return"

```
/*   File: indincr.c
     Program illustrates indirect access
     to x by a function indirect_incr().
     Function increments x by 1.
*/
#include <stdio.h>
void indirect_incr(int * p);

main()
{    int x;

     printf("***Indirect Access***\n");
     x = 7;
     printf("Original value of x is %d\n", x);
     indirect_incr(&x);

     printf("The value of x is %d\n", x);
}


/*   Function indirectly accesses object in calling function.  */
void indirect_incr(int * p)
{
     *p = *p + 1;
}
```

Figure 6.11: Code for Indirect Access by a Function

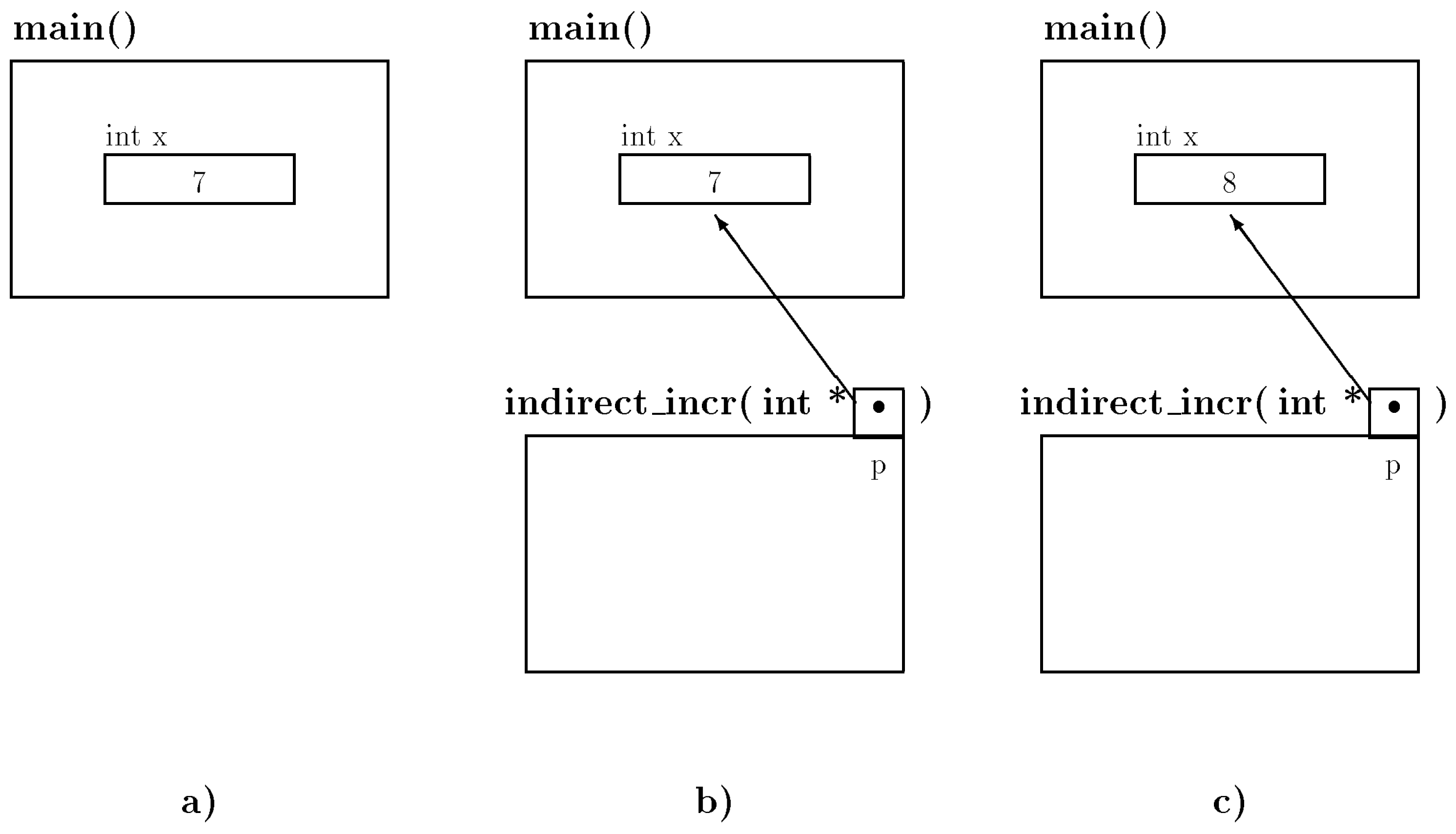

Figure 6.12: Trace for Indirect Access by a Function

```
/*   File: sqcube.c
     Program uses a function that returns a square of its argument and
     indirectly stores the cube.
*/
#include <stdio.h>
double sqcube(double x, double * pcube);

main()
{    double x, square, cube;

     printf("***Directly and Indirectly Returned Values***\n");
     x = 3;

     square = sqcube(x, &cube);
     printf("x = %f, square = %f, cube = %f\n",
               x, square, cube);
}

/* Function return square of x, and indirectly stores cube of x */
double sqcube(double x, double * pcube)
{
     *pcube = x * x * x;
     return (x * x);
}
```

Figure 6.13: Code for Indirectly Returned Values

two values to the calling function. We know that only one value can be returned as the value of the function, so we can decide to write the function with one of the two values *formally* returned by a `return` statement, and the other value stored, by indirect access, in an object defined in the calling function. The two values are "returned" to the calling function, one formally and one by indirection.

Let us write a function to return the square and the cube of a value. We decide that the function returns the square as its value, and "returns" the cube by indirection. We need two parameters; one to pass the value to be squared and cubed to the function, and one pointer type parameter which will be used to indirectly access an appropriate object in the calling function to store the cube of the value. We assume all objects are of type `double`.

The code is shown in Figure 6.13. The prototype for `sqcube()` is defined to have two parameters, a `double` and a pointer to `double`, and it returns a `double` value. The `printf()` prints the value of `x`; the value of `square` which is the value returned by `sqcube()` (the square of `x`); and, the value of `cube` (the cube of `x`) which is indirectly stored by `sqcube()`.

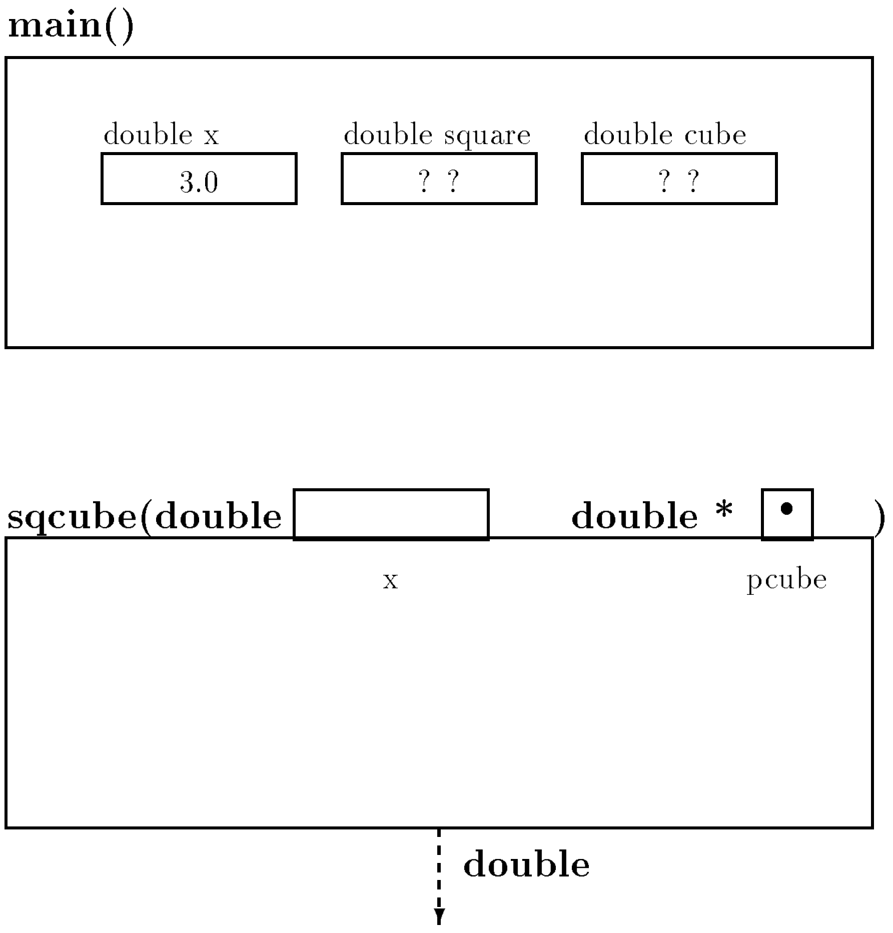

Figure 6.14: Trace for `sqcube` — Step 1

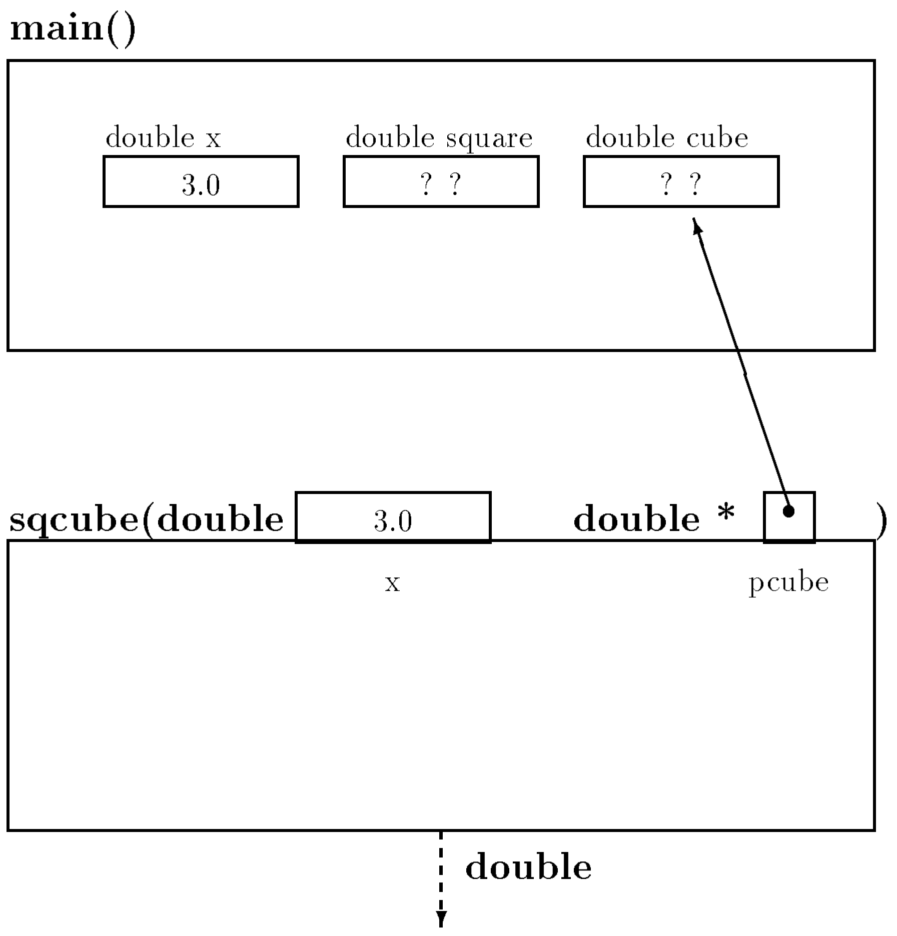

Figure 6.15: Trace for `sqcube` — Step 2

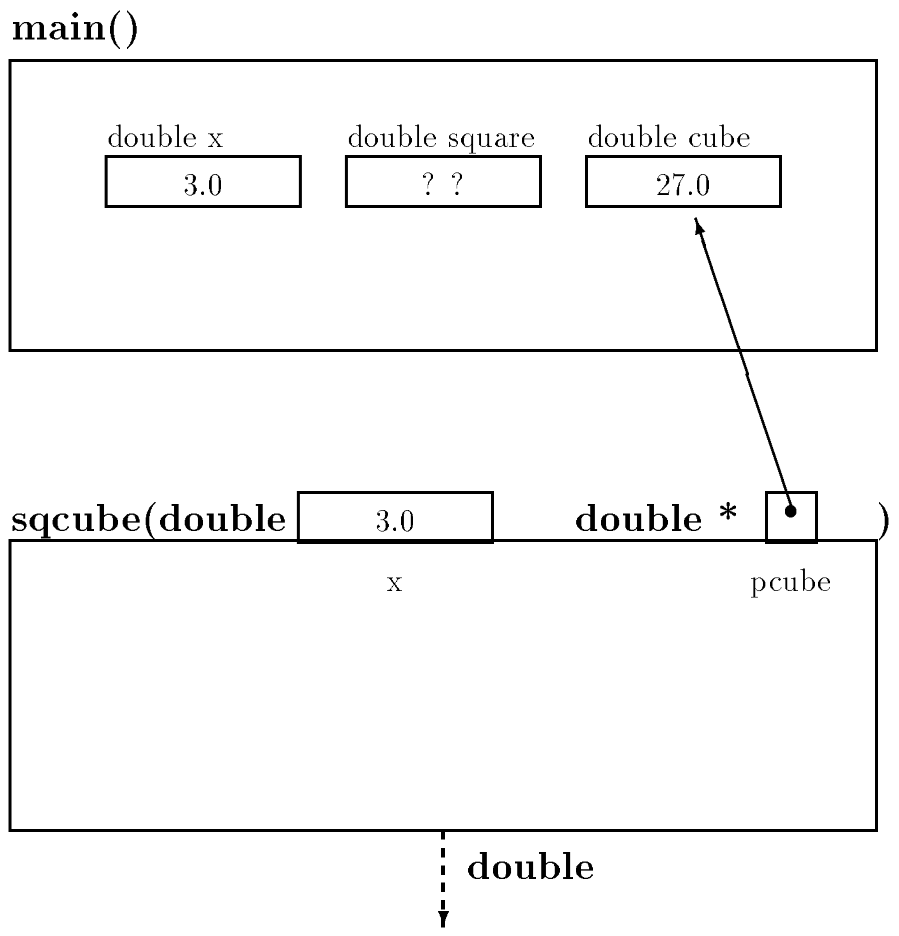

Figure 6.16: Trace for `sqcube` — Step 3

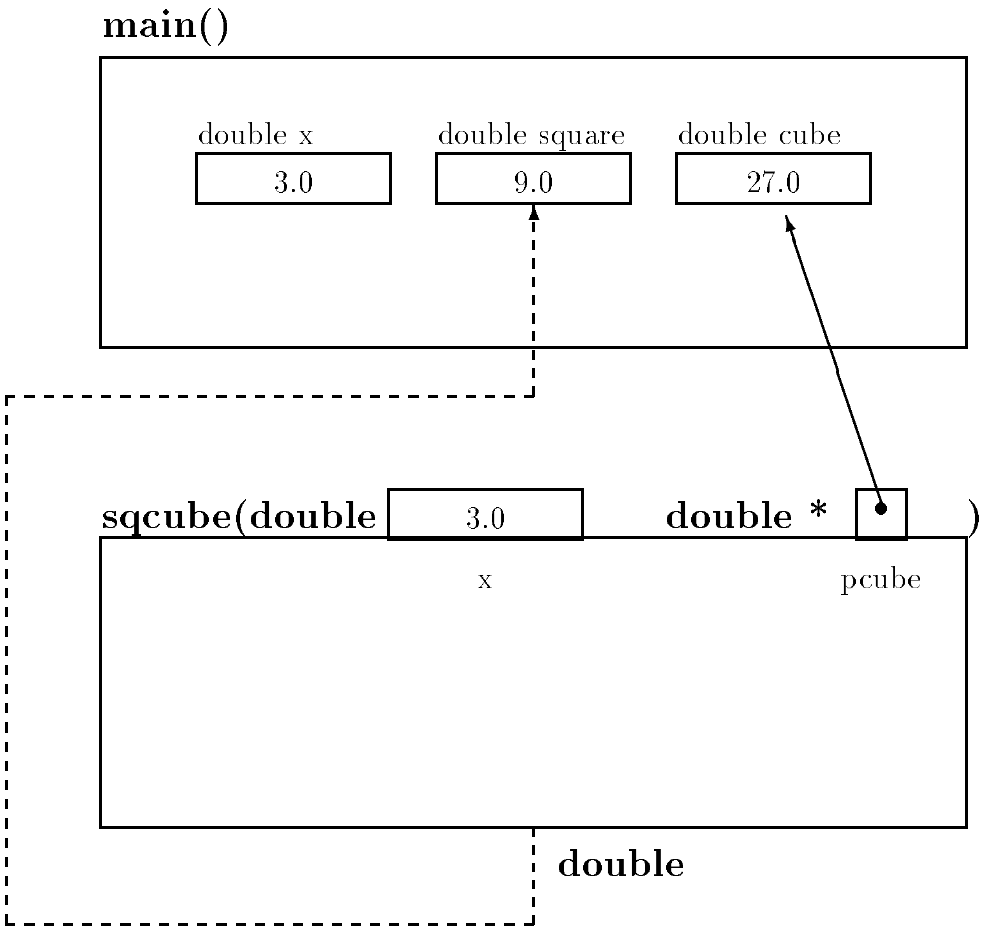

Figure 6.17: Trace for `sqcube` — Step 4

Figures 6.14 — 6.17 show a step-by-step trace of the changes in objects, both in the calling function and in the called function. In the first step (Figure 6.14), the declarations for the function, `main()` and the template for the function, `sqcube()` are shown with the initialization of the variable, `x`, in `main()`. In the second step (Figure 6.15), the function, `sqcube()` is called from `main()` passing the *value* of `x` (3.0) to the first parameter, (called `x` in `sqcube()`), and the *value* of `&cube`, namely a pointer to `cube`, as the second argument to the parameter, `pcube`. In the third step (Figure 6.16), the first statement in `sqcube()` has been executed, computing the cube of the local variable, `x`, and storing the value indirectly in the cell pointed to by `pcube`. Finally, Figure 6.17 shows the situation just as `sqcube()` is returning, computing the square of `x` and returning the value which is assigned to the variable, `square`, by the assignment in `main()`.

While only one value can be returned as the value of a function, we loosely say that this function "returns" two values: the square and the cube of `x`. The distinction between a formally returned value and an indirectly or loosely "returned" value will be clear from the context.

Sample Session:

```
***Directly and Indirectly Returned Values***
x = 3.000000, square = 9.000000, cube = 27.000000
```

6.2.3 A function to Swap Values

We have already seen how values of two objects can be swapped directly in the code in `main()`. We now write a function, `swap()`, that swaps values of two objects defined in `main()` (or any other function) by accessing them indirectly, i.e. through pointers. The function `main()` calls the function, `swap()`, passing pointers to the two variables. The code is shown in Figure 6.18. (We assume integer type objects in `main()`).

The function, `swap()`, has two formal parameters, integer pointers, `ptr1` and `ptr2`. A temporary variable is needed in the function body to save the value of one of the objects. The objects are accessed indirectly and swapped. Figures 6.19 — 6.22 show the process of function call, passed values, and steps in the swap.

Sample Session:

```
Original values:  dat1 = 100, dat2 = 200
Swapped values:  dat1 = 200, dat2 = 100
```

6.3 Returning to the Payroll Task with Pointers

We will now modify our pay calculation program so that the driver calls upon other functions to perform *all* subtasks. The driver, `main()`, represents only the overall logic of the program; the

```
/*   File: swapfnc.c
     Program uses a function to swap values of two objects.
*/
#include <stdio.h>
/* arguments of swap() are integer pointers */
void swap(int * p1, int * p2);

main()
{    int dat1 = 100, dat2 = 200;

     printf("Original values: dat1 = %d, dat2 = %d\n", dat1, dat2);
     swap(&dat1, &dat2);
     printf("Swapped values: dat1 = %d, dat2 = %d\n", dat1, dat2);
}

/*  Function swaps values of objects pointed to by ptr1 and ptr2 */
void swap(int * ptr1, int * ptr2)
{    int temp;

     temp = *ptr1;
     *ptr1 = *ptr2;
     *ptr2 = temp;
}
```

Figure 6.18: Code for a Function, `swap()`

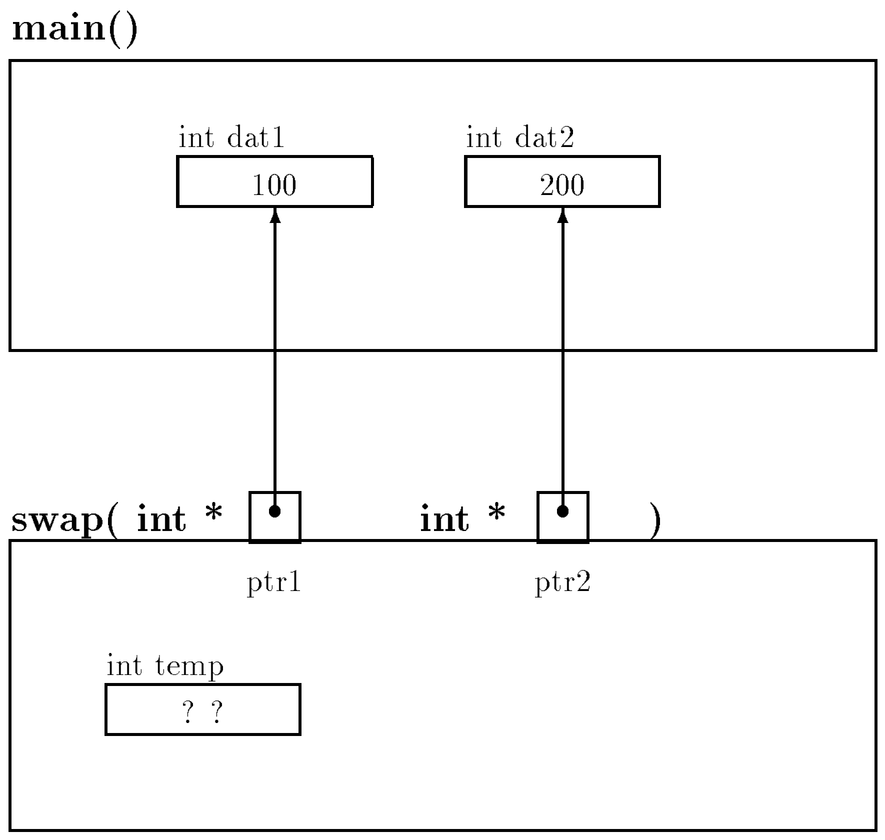

Figure 6.19: Trace for `swap()` — Step 1

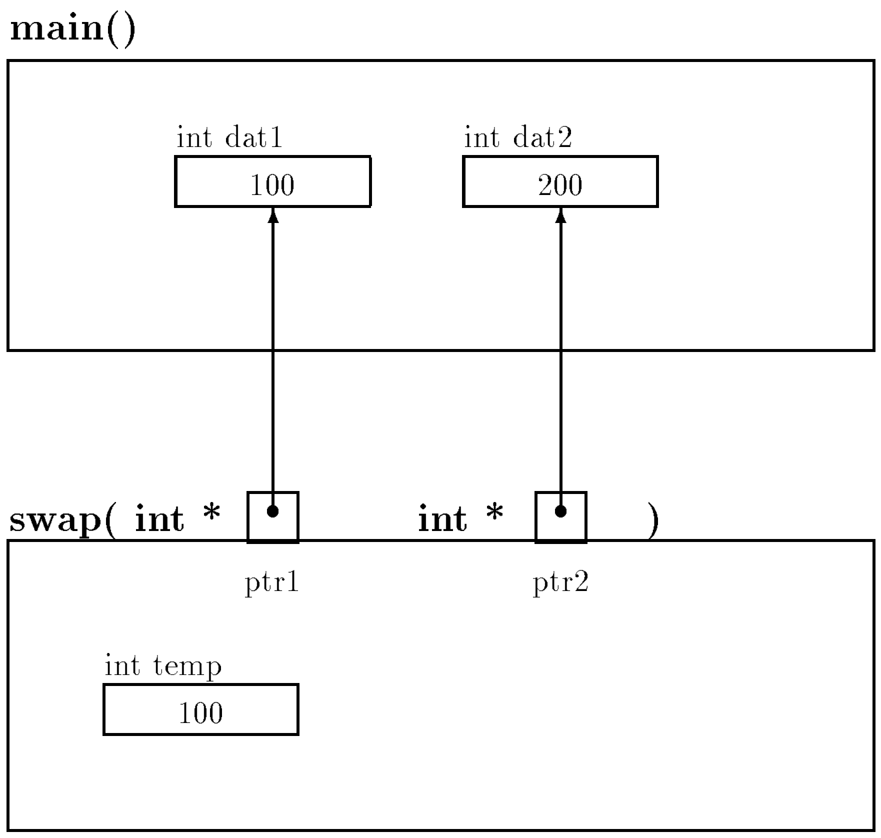

Figure 6.20: Trace for `swap()` — Step 2

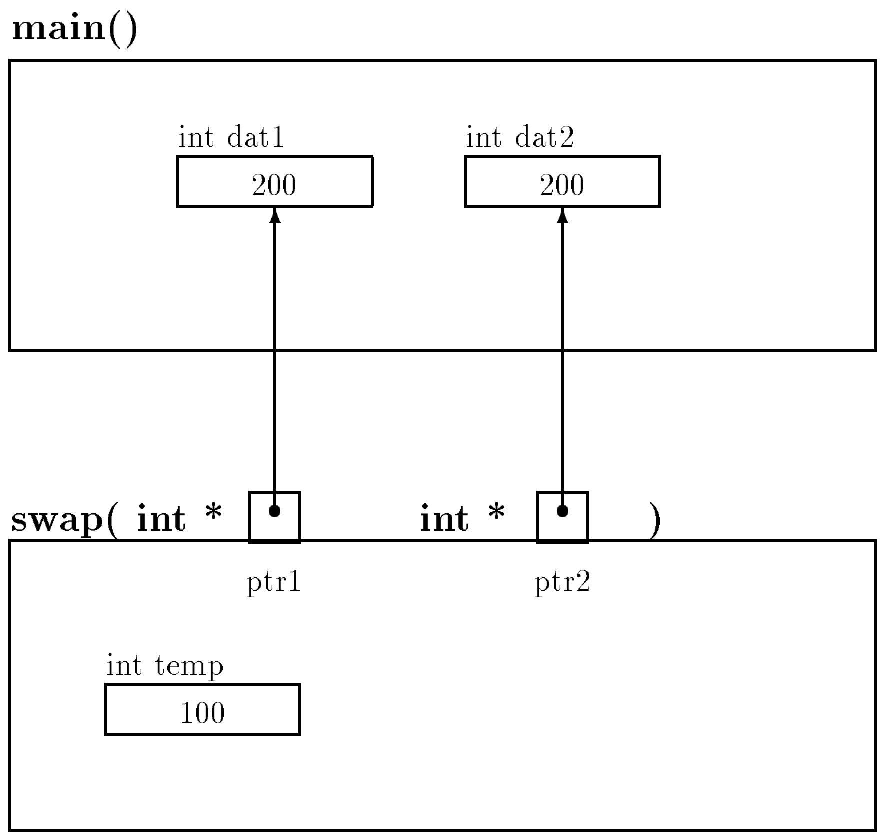

Figure 6.21: Trace for `swap()` — Step 3

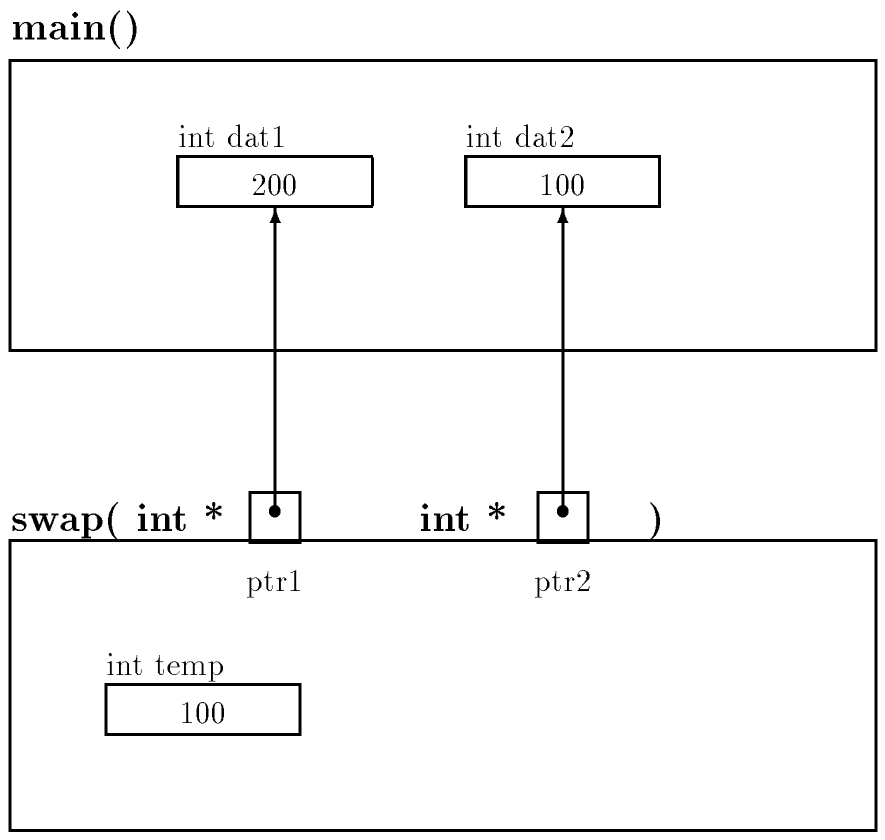

Figure 6.22: Trace for `swap()` — Step 4

details are hidden in the functions that perform the various subtasks. The algorithm for the driver is:

```
get data
repeat the following while there is more data
     calculate pay
     print data and results
     get data
```

For each step of the algorithm, we will use functions to do the tasks of getting data, printing data and results, and calculating pay. We have already written functions in Chapters 3 and 4 to calculate pay and to print data and results, and will repeat them here for easy reference, making some modifications and improvements. We have postponed until now writing a function to read data as such a function would require returning more than one value. By using pointers, we now have the tool at our disposal to implement such a function.

Before we *write* these functions, we should *design* them by describing what the functions do and specifying the **interface** to these functions; i.e. by indicating the arguments and their types to be passed to the functions (the information *given* to the functions) and the meaning and type of the return values (the information returned from the function). Here are our choices:

get_data(): This function reads the id number, hours worked, and rate of pay for one employee and stores their values indirectly using pointers. Since these values are returned indirectly, the arguments must be pointers to appropriate objects in the calling function (`main()` in our case). The function returns True, if it found new data in the input; it returns False otherwise. Here is the prototype:

```
int get_data(int * pid, float * phrs, float * prate);
```

We use names `pid`, `phrs`, and `prate`, to indicate that they are pointers to cells for the id, hours and rate, respectively. It is a good habit to distinguish between object names and pointer names whenever there is a possibility of confusion.

print_data(): This function writes the id number, hours worked, and rate of pay passed to it. It has no useful information to return so returns a `void` type. Here is the prototype:

```
void print_data(int id, float hrs, float rate, float pay);
```

print_pay(): This function is given values for the regular pay, overtime pay, and total pay and writes them to the output. It also returns a `void` type.

```
void print_pay(float regular, float overtime, float total);
```

calc_pay(): Given the necessary information (hours and rate), this function calculates and returns the total pay, and indirectly returns the regular and overtime pay. In addition to the values of hours worked and rate of pay, pointers to regular pay and overtime pay are passed to the function. The prototype is:

```
/* File: payutil.h */
#define REG_LIMIT 40
#define OT_FACTOR 1.5
int get_data(int *pid, float *phrs, float *prate);
void print_data(int id, float hrs, float rate);
void print_pay(float regular, float overtime, float total);
float calc_pay(float hours, float rate, float * pregular,
                    float * povertime);
```

Figure 6.23: Header file `payutil.h`

```
float calc_pay(float hours, float rate, float * pregular,
                    float * povertime);
```

Here, `pregular` and `povertime` are pointers to cells for regular and overtime pay objects in the calling function.

All of these functions will be defined in a file, `payutil.c` and their prototypes are included in `payutil.h`. Figure 6.23 shows the header file. We have also included the definitions for symbolic constants `REG_LIMIT` and `OT_FACTOR` in the header file. This header file will be included in all relevant source files.

With the information in this file (and the preceding discussion of the function) we have sufficient information to write the driver for the program *using* the functions prior to writing the actual code for them. Figure 6.24 shows the driver. It also includes the file, `tfdef.h` which defines the macros, `TRUE` and `FALSE`.

The logic of the driver is as follows. After the program title is printed, the first statement calls `get_data()` to get the `id_number`, `hours_worked`, and `rate_of_pay`. As indicated in the prototype, pointers to these objects are passed as arguments so that `get_data()` can indirectly access them and store values. The function, `get_data()`, returns True or False depending on whether there is new data. The True/False value is assigned to the variable, `moredata`. The `while` loop is executed as long as there is more data; i.e. `moredata` is True. The loop body calls on `calc_pay()` to calculate the pay, `print_data()` to print the input data, `print_pay()` to print the results, and `get_data()` again to get more data. Since `calc_pay()` returns the values of overtime and total pay indirectly, `main()` passes pointers to objects which will hold these values.

The overall logic in the driver is easy to read and understand; at this top level of logic, the details of the computations are not important and would only complicate understanding the program. The driver will remain the same no matter how the various functions are defined. The actual details in one or more functions may be changed at a later time without disturbing the driver or the other functions. This program is implemented in functional modules. Such a modular programming style makes program development, debugging and maintenance much easier.

```
/*   File: pay6.c
     Other Files: payutil.c
     Header Files; tfdef.h, payutil.h
   The program gets payroll data, calculates pay, and prints out
   the results for a number of people. Modular functions are used
   to get data, calculate total pay, print data, and print results.
*/
#include <stdio.h>
#include "tfdef.h"
#include "payutil.h"
main()
{
     /* declarations */
     int id_number, moredata;
     float hours_worked, rate_of_pay, regular_pay, overtime_pay, total_pay;


     /* print title */
     printf("***Pay Calculation***\n\n");

     /* get data and initialize loop  variable */
     moredata = get_data(&id_number, &hours_worked,
                         &rate_of_pay);
     /* process while moredata */
     while (moredata) {
          total_pay = calc_pay(hours_worked, rate_of_pay, &regular_pay,
                                   &overtime_pay);
          print_data(id_number, hours_worked, rate_of_pay);
          print_pay(regular_pay, overtime_pay, total_pay);
          moredata = get_data(&id_number, &hours_worked,
                                &rate_of_pay);
     }
}
```

Figure 6.24: Code for the Driver for pay6.c

```
/* File: payutil.c */
#include <stdio.h>
#include "tfdef.h"
#include "payutil.h"
/* Function prints out the input data */
void print_data(int id, float hours, float rate)
{
     printf("\nID Number = %d\n", id);
     printf("Hours Worked = %f, Rate of Pay = %f\n",
                hours, rate);
}

/* Function prints pay data */
void print_pay(float regular, float overtime, float pay)
{
     printf("Regular Pay = %f, Overtime Pay = %f\n",
                regular, overtime);
     printf("Total Pay = %f\n", pay);
}
```

Figure 6.25: Code for `print_data()` and `print_pay()`

Of course, we still have to write the various functions used in the above driver. We write each of these functions in turn. Figure 6.25 shows the code for `print_data()` and `print_pay()` in the file `payutil.c` which are simple enough.

The next two functions require indirect access. The function, `calc_pay()`, must indirectly store the regular and overtime pay so the formal parameters include two pointers: `preg` (pointing to the cell for regular pay) and `pover` (pointing to the cell for overtime pay). The function returns the value of the total pay. It is shown in Figure 6.26. Finally, `get_data()` must indirectly store the values of the id number, hours worked, and rate of pay, and return True if id number is positive, and False otherwise. Figure 6.27 shows the code. The formal parameters `pid`, `phrs`, and `prate` are pointers to objects in the calling function (`main()` in our case). Recall, when `scanf()` is called to read data, it requires arguments that are pointers to the objects where the data is to be placed so that it can indirectly store the values. Therefore, when `get_data()` calls `scanf()`, it must pass pointers to relevant objects as arguments, i.e. it passes the pointers, `pid`, `phrs`, and `prate`. These pointer variables point to the objects where values are to be stored. We do NOT want to pass `&pid`, `&phrs`, `&prate` as these are the *addresses of the pointers*, `pid`, `phrs`, and `prate`; they are NOT the addresses cells to hold the data. If the id number stored in `*pid` is not positive, i.e. (`*pid` $<=$ `0`), `get_data()` returns `FALSE` to indicate that there is no more data. If `*pid` is positive, the rest of the function is executed, in which case the rest of the input data is read. The value, `TRUE` is returned to indicate that more data is present.

The above functions are in the source file, `payutil.c` which must be compiled and linked with the source program file, `pay6.c`. A sample session would be similar to the ones for similar previous

```
/* File: payutil.c - continued */
/* Function calculates and returns total pay */
float calc_pay(float hours, float rate, float * preg, float * pover)
{    float total;

     if (hours > REG_LIMIT) {
          *preg = REG_LIMIT * rate;
          *pover = OT_FACTOR * rate * (hours - REG_LIMIT);
     }
     else {
          *preg = hours * rate;
          *pover = 0;
     }
     total = *preg + *pover;
     return total;
}
```

Figure 6.26: Code for `calc_pay()`

```
/* File: payutil.c - continued */
/* Function reads in the payroll data */
int get_data(int * pid, float * phrs, float * prate)
{
     printf("Type ID Number, zero to quit: ");
     scanf("%d", pid);
     if (*pid <= 0)            /* if ID number is <= 0, */
          return FALSE;        /* return 0 to calling function */
     printf("Hours Worked: "); /* ID number is valid, get data */
     scanf("%f", phrs);
     printf("Hourly Rate: ");
     scanf("%f", prate);
     return TRUE;              /* valid data entered, return 1 */
}
```

Figure 6.27: Code for `get_data()`

programs and is not shown here.

6.4 Common Errors

1. Using an uninitialized pointer. Remember, declaring a pointer variable simply allocates a cell that can hold a pointer — it does not place a value in the cell. So, for example, a code fragment like:

```
{  int * iptr;

   *iptr = 2;
   . . .
}
```

   will attempt to place the value, 2, in the cell pointed to by `iptr`; however, `iptr` has not been initialized, so some garbage value will be used as the address of there to place the value. On some systems this may result in an attempt to access an illegal address, and a memory violation. Avoid this error by remembering to initialize all pointer variables before they are used.

2. Instead of using a pointer to an object, a pointer to a pointer is used. Consider a function, `read_int()`. It reads an integer and stores it where its argument points. The correct version is:

```
void read_int(int * pn)
{
     scanf("%d", pn);
}
```

   `pn` is a pointer to the object where the integer is to be stored. When passing the argument to `scanf()`, we pass the pointer, `pn`, NOT `&pn`.

3. Confusion between the address of operator and the dereference operator.

```
... calling_func(...)
{    int x;
     called_func(*x);         /* should be &x   */
     ...
}
... called_func(int &px)      /* should be * px */
{
     ...
}
```

   A useful mnemonic aid is that the "address of" operator is the "and" symbol, `&` — both start with letter, *a*.

6.5 Summary

In this Chapter we have introduced a new data type, a **pointer**. We have seen how we can declare variables of this type using the * and indicating the type of object this variable can point to, for example:

```
int   * iptr;
float * fptr;
char  * cptr;
```

declare three pointer variables, `iptr` which can point to an integer cell, `fptr` which can point to a cell holding a floating point variable, and `cptr` which can point to a character cell.

We have seen how we can assign values to pointer variables using the "address of" operator, `&` as well as from other pointer variables. For example,

```
{   int x;
    int * ip;
    int * iptr;

    iptr = &x;
    ip = iptr;

    . . .
}
```

declares an integer variable, `x`, and two integer pointers, `ip` and `iptr`, which can point to integers (we can read this last declaration from right to left, as saying that " `iptr` points to an `int`"). We then assign the *address* of `x` to the pointer variable, `iptr`, and the *pointer* in `iptr` to the variable, `ip`.

We have also shown how pointer variables may be used to *indirectly* access the value in a cell using the **dereference** operator, `*`:

```
y = *iptr;
```

which assigns the value of the cell pointed to by `iptr` to the variable, `y`. Values may also be stored indirectly using pointer variables:

```
*iptr = y;
```

which assigns the value in the variable, `y`, to the cell pointed to by `iptr`.

We have also seen that we can pass pointers to functions and use them to modify the values of cells in the *calling* function. For example:

```
main()
{  int x, y, z;

   z = set_em( &x, &y};
   . . .
}

int set_em( int *a, int *b)
{
   *a = 1;
   *b = 2;
   return 3;
}
```

Here the function, `set_em` will set the values 1, 2, and 3 into the variables `x`, `y`, and `z` respectively. The first two values are assigned indirectly using the pointers passed to the function, and the third is returned as the value of the function and assigned to `z` by the assignment statement in `main()`. This, the function, `set_em()`, has "effectively" returned three values.

Finally, we have used this new indirect access mechanism to write several programs, including an update to our payroll program. As we will see in succeeding chapters, pointers are very useful in developing complex programs. The concept of pointers may be confusing at first, however, a useful tool for understanding the behavior of a program using pointers is to draw the memory picture showing which to cells each pointer is pointing.

6.6 Exercises

1. What is the output of the following code?

```
int x, y, z, w;
int * pa, * pb, * pc, * pd;

x = 10; y = 20; z = 30;
pa = &x;
pb = &y;

printf("%d, %d, %d\n", *pa, *pb, *pc);
pc = pb;
printf("%d, %d, %d\n", *pa, *pb, *pc);
pb = pa;
printf("%d, %d, %d\n", *pa, *pb, *pc);
pa = &z;
printf("%d, %d, %d\n", *pa, *pb, *pc);
*pa = *pb;
printf("%d, %d, %d\n", *pa, *pb, *pc);
```

What is the output for each of the following programs:

2.

```
#define SWAP(x, y)  {int temp; temp = x; x = y; y = temp; )
main()
{    int data1 = 10, data2= 20;
     SWAP(data1, data2);
     printf("Data1 = %d, data2 = %d\n", data1, data2);
}
```

3.

```
#define SWAP(x, y)  {int *temp; temp = x; x = y; y = temp; )
main()
{    int data1 = 10, data2= 20;
     int *p1, *p2;
     p1 = &data1; p2 = &data2;
     SWAP(p1, p2);
     printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
}
```

Correct the code in the following problems:

4.

```
main()
{    int x, *p;

     x = 13;
     ind_square(*p);
}
```

```
ind_square(int *p)
{
     *p = *p * *p;
}
```

5.
```
main()
{    int x, *p;

     x = 13; p = &x;
     ind_square(&p);
}

ind_square(int &p)
{
     *p = *p * *p;
}
```

6.
```
main()
{    int x, *p;

     x = 13;
     ind_square(x);
}

ind_square(int *p)
{
     *p = *p * *p;
}
```

7.
```
main()
{    int x, *p;

     x = 13;
     ind_square(p);
}

ind_square(int *p)
{
     *p = *p * *p;
}
```

6.7 Problems

1. Write a program that initializes integer type variables, `data1` and `data2` to the values 122 and 312. Declare pointers, `ptr1` and `ptr2`; initialize `ptr1` to point to `data1` and `ptr2` to point to `data2`. Swap the values of `data1` and `data2` values using direct access and using indirect access. Next, swap the values of the pointers, `ptr1` and `ptr2` and print the values indirectly accessed by the swapped pointers.

2. Write a function (that returns `void`) which reads and indirectly stores three values in the calling function. The types of the three data items are an integer, a character, and a float.

3. Write a function `maxmin(float x, float * pmax, float * pmin)` where `x` is a new value which is to be compared with the largest and the smallest values pointed to by `pmax` and `pmin`, respectively. The function should indirectly update the largest and the smallest values appropriately. Write a program that reads a sequence of numbers and uses the above function to update the maximum and the minimum until end of file, when the maximum and the minimum should be printed.

4. Repeat Problem 2.10 using functions `get_course_data()`, `calc_gpr()`, and `print_gpr()`.

5. Rewrite Problem 5.1 as a function that finds the roots of a quadratic and returns them indirectly.

6. Rewrite the program to solve simultaneous equations (Problem 5.10). The program should use a function, `solve_eqns()` to solve for the unknowns. The function must indirectly access objects in main() to store the solution values.

7. Write a menu-driven program that uses the function, `solve_eqns()`, of Problem 6. The commands are: get data, display data, solve equations, print solution, verify solution, help, and quit. Use functions to implemnent the code for each command.

8. A rational number is maintained as a ratio of two integers, e.g., 20/23, 35/46, etc. Rational number arithmetic adds, subtracts, multiplies and divides two rational numbers. Write a function to add two rational numbers.

9. Write a function to subtract two rational numbers.

10. Write a function to multiply two rational numbers.

11. Write a function to divide two rational numbers.

12. Write a function to reduce a rational number. A reduced rational number is one in which all common factors in the numerator and the denominator have been cancelled out. For example, 20/30 is reduce to 2/3, 24/18 is reduced to 4/3, and so forth.

13. Use the function, `reduce()`, of Problem 12 to implement the functions in Problems 8 through 11.

14. Rewrite the program of Problem 5.13 to calculate the current and the power in a resistor using a function instead to perform the calculations. One value may be returned as a function value, the other must be indirectly stored in the calling function.