

Laboratory 7

Pointers and Passing Parameters by Reference

7.1 Overview

This laboratory begins by investigating how parameters are normally passed to functions. When a function is called, the value of an argument is copied into a parameter (which acts like a variable) inside the function. This is called *passing by value*.

You will then be introduced to a special kind of variable called *pointers* which, rather than store a value of their own, reference another variable. A pointer allows the program to find, and modify, another variable. In the next part of this lab you will use pointers when calling functions to allow them to modify the variables that have been passed as arguments. This is called *passing by reference*.

In the final part of the lab you will use pointers with structures to improve the way in which your programs from lab 6 are written.

7.2 More About Function Parameter Passing

This laboratory will introduce you to a useful and very powerful technique for working with variables. Before you can go on to this, we will take some time to look at the way in which variables work when functions are called. We will use very simple pieces of C code to understand important concepts.

The source code below is from the “lab7” project. Change the target to “lab7” and open “lab7.c” in the editor. Don’t execute the program yet.



```
void change(int i)
{
    printf("The number is now %d\n", i);

    i = 20;

    printf("The number is now %d\n", i);
}

int main(void)
{
    int j;

    j = 5;

    printf("The number is %d\n", j);

    change(j);

    printf("The number is now %d\n", j);

    return 0;
}
```

You will see that there are two functions in this program. The `main` function and another function called `change`.

Exercise 7.1: Investigating Function Parameters

Before you try running the program look at it carefully. Decide *exactly* what you think it will do. What will it display on the screen?

Try running the program. Does it do what you thought it would? Do you understand why it operates in the way it does?

You should remember some important facts about functions from lab 4:

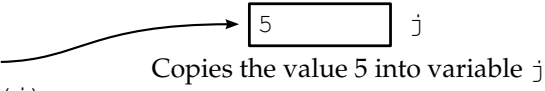
- parameters (the part in parentheses after the function name) are just like variables;
- all variables (and parameters) that are declared in a function are only accessible inside that function;
- when a function is called, it is common to specify some arguments in brackets after the function name;
- the values of these arguments are copied into the function parameter variables.

These points mean that if a parameter is changed inside a function the value of the variable that was used as an argument is **not** affected.

To emphasize this point, let's look at how the simple program "lab7" works. To begin with, a value is assigned to the variable `j`. This is shown below:

```
void change(int i)
{
    i = 20;
}
```

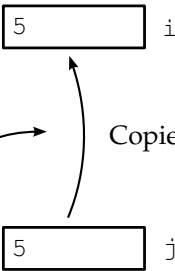
```
int main(void)
{
    int j;
    j = 5;
    change(j);
    return 0;
}
```



Copies the value 5 into variable j

When the function is called, the variable `j` is specified as an argument. Its value is copied into the `change` function's parameter `i` (see the diagram below).

```
void change(int i)
{
    i = 20;
}
```




Copies the value from argument variable `j` into parameter `i`

```
int main(void)
{
    int j;
    j = 5;
    change(j);
    return 0;
}
```

In the `change` function, the value 20 is assigned to the parameter `i`. Notice that the value of the variable `j`, which only exists inside `main`, is **not affected**.

```
void change(int i)
{
    i = 20;
}
```



Copies the value 20 into parameter `i`

```
int main(void)
{
    int j;
    j = 5;
    change(j);
    return 0;
}
```




main variable `j` unaffected

When the `change` function returns the program flow to the `main` function, the value of `j` has not changed.

```
void change(int i)
{
    i = 20;
}
```

```
int main(void)
{
    int j;
    j = 5;
    change(j);
    return 0;
}
```



Value of variable `j` has not changed

This system of copying the *value* of an argument into the parameter is called *passing by value*. It is the normal way in which C functions work. It has a very good side and a less good side:

- the good side is that functions are not able to affect the code that calls them. This stops them from damaging other parts of your program and ensures encapsulation (this was mentioned in laboratory 4).
- the less good side is that the only way to return a value is by using the `return` statement. This limits a function to only returning one value. Sometimes you might want to write code in which a function returns more than one value.

The next part of the lab introduces *pointers* which, amongst other things, allow us to get round this limitation.

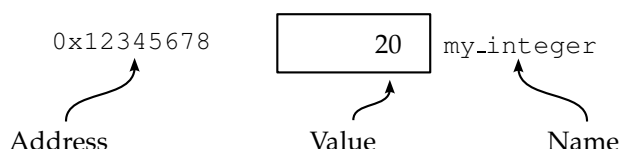
7.3 Introducing Pointers

When you store a number in a variable, the program represents that number as binary and places it in the computer's memory. In some diagrams in this lab script this process has been shown by depicting the memory location as a box containing the number. Like this:

`int my_integer = 20;` \equiv 20 `my_integer`

(This diagram appeared in lab 5). You can think of the computer's memory being entirely made up of boxes like this one. Some of these boxes are used for your program's variables, others are used for your program's code and more are used by other programs.

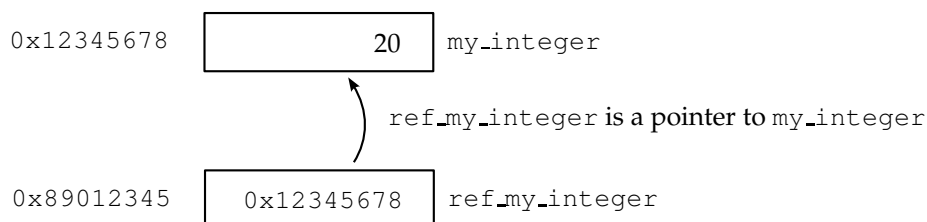
Every box or *location* in the computer's memory is numbered. The number for a location identifies it uniquely and is called an *address*. You can refer to any location in the computer's memory by specifying its address. Since addresses are just numbers it can be difficult to keep track of them so programming languages give us a way of representing location addresses with names. A variable name (like `my_integer`) is just an easier way of talking about a location in the computer's memory.



The address in this diagram is made up. It is difficult to know what address your variables will have before your program loads. Because C gives a name to each variable, you don't need to know the address.

C provides a way to use one variable to refer to the contents of another variable. This kind of variable is called a *reference*, because instead of holding a value of its own it references another memory location, which holds a value. In C, references are more commonly called *pointers*. The two names are interchangeable, we will most often use the name *pointer* in this lab script.

The diagram below shows a pointer variable called `ref_my_integer` which references the variable `my_integer`. The reference is shown with an arrow.



Notice that the contents of the memory location associated with the pointer variable `ref_my_integer` is `0x12345678`. This is the address of the memory location associated with the `my_integer` variable.

Pointers in C are declared like this:

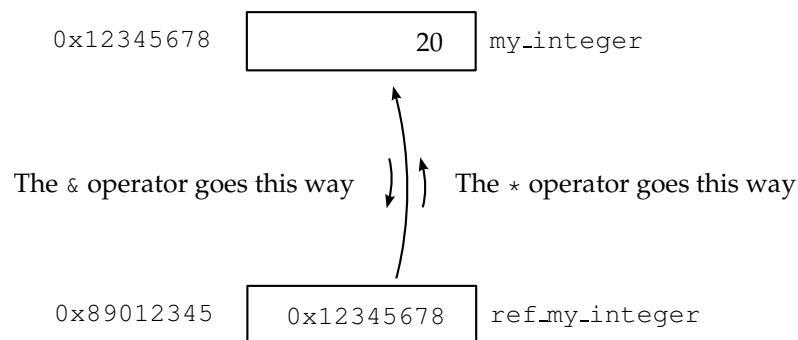
```
int *ref_my_integer;
```

This line declares a new variable called `ref_my_integer` which is a pointer, or a reference, to an `int`. The `*` symbol can be placed immediately after the `int` like this:

```
int* ref_my_integer;
```

These two statements are identical. It is a matter of stylistic taste whether you append the data type (i.e. `int`) with an asterisk or prepend the variable itself. Notice that we have specified what this point will reference: an integer (type `int`). This pointer cannot be used to reference a `double` or a `char` or anything else. This helps to prevent some simple programming mistakes.

The star (`*`) is one of two C operators which are used when working with pointers, the other is the ampersand (`&`). You can think of these operators as travelling along the arrow which goes from a pointer to the memory location it is pointing to.



To make this clearer, let's look at an example piece of code:

```
int main(void)
{
    int my_integer;
    int *ref_my_integer;

    ref_my_integer = &my_integer;

    my_integer = 20;
    printf("my_integer is %d\n", my_integer);

    *ref_my_integer = 5;
    printf("my_integer is now %d\n", my_integer);

    return 0;
}
```

This piece of code is in the project “lab7a”. Open it in the IDE and change the target to “lab7a”.



Exercise 7.2: Investigating Pointers

Build and execute the program “lab7a”. Look at the source code and have a go at trying to understand how the program works. Don't worry if you don't understand, the next part of the lab will explain this program.

We will now investigate this program a line at a time.

The first two lines of the program declare the integer variable `my_integer` and the pointer variable `ref_my_integer` which is specified as referencing integers. The program has not assigned values to these variables yet.

```
int main(void)
{
    0x12345678   my_integer

    int my_integer;
    int *ref_my_integer;

    0x89012345   ref_my_integer

    ref_my_integer = &my_integer;

    my_integer = 20;
    printf("my_integer is %d\n", my_integer);

    *ref_my_integer = 5;
    printf("my_integer is now %d\n", my_integer);

    return 0;
}
```

The next line uses the `&` operator to set up a pointer from the integer variable `my_integer`. The `&` operator is called the *reference operator* because it allows us to create a reference from a variable. A reference (pointer) to the variable `my_integer` is created and assigned to the pointer variable `ref_my_integer`.

```
int main(void)
{
    0x12345678   my_integer
                ↑
    0x89012345   ref_my_integer
                |
                0x12345678

    ref_my_integer = &my_integer;

    my_integer = 20;
    printf("my_integer is %d\n", my_integer);

    *ref_my_integer = 5;
    printf("my_integer is now %d\n", my_integer);

    return 0;
}
```

The next line assigns the value 20 to the variable `my_integer` and then uses a call to the `printf` function to display its value.

```
int main(void)
{
    0x12345678   my_integer
                ↑
    0x89012345   ref_my_integer
                |
                0x12345678

    ref_my_integer = &my_integer;

    my_integer = 20;
    printf("my_integer is %d\n", my_integer);

    *ref_my_integer = 5;
    printf("my_integer is now %d\n", my_integer);

    return 0;
}
```

The next line uses the `ref_my_integer` pointer variable with the `*` operator. The `*` operator is called the *dereference operator* and tells the program that we are not talking about the pointer variable, `ref_my_integer`, by what ever it references, in this case the `my_integer` variable.

```
int main(void)
{
    int my_integer;
    int *ref_my_integer;

    ref_my_integer = &my_integer;

    my_integer = 20;
    printf("my_integer is %d\n", my_integer);

    *ref_my_integer = 5;
    printf("my_integer is now %d\n", my_integer);

    return 0;
}
```



The assignment copies the value 5 into the location that is referenced by the pointer `ref_my_integer`, which changes the `my_integer` variable. The value that is displayed on the screen is therefore 5.

Exercise 7.3: More Pointer Investigation

Try stepping through the “lab7a” program. Watch the variables change in the watch window.

Make sure that you understand what is happening before you continue. If necessary, ask one of the demonstrators to explain it to you.

You can see that the reference (`&`) and dereference (`*`) operators do opposite things.

- The reference operator (`&`) creates a reference (a pointer) from a variable. It simply takes the address of the memory location the variable is using to store its information in.
- The dereference operator (`*`) uses a reference to find the original variable. It uses the address stored in the pointer variable to find the memory location containing the information.

Syntax: Pointer Reference and Dereference Operators

& *

The reference operator (&) operates on a variable to create a reference. It does this by obtaining the address of the memory location which is being used to store the information for that variable. For example:

```
&some_variable
```

is directly equivalent to the address of the variable `some_variable`.

The dereference operator * carries out the opposite operation to the reference operator: it takes an address in the form of a pointer and follows it to find the location that it refers to. So, if:

```
some_pointer = &some_variable;
```

`*some_pointer` is directly equivalent to `some_variable` i.e. the lines

```
*some_pointer = 20;  
some_variable = 20;
```

do the same thing.

The dereference operator is used when declaring pointer variables, as in:

```
type *name;
```

For example, the line:

```
int *some_pointer;
```

Declares a new variable called `some_pointer` which is a pointer to a variable of type `int`

You might have noticed that a pointer variable is declared using the dereference operator like this:

```
int *ref_my_integer;
```

This is saying “I would like to declare a variable with a type such that if you were to dereference it, you would get an `int`”. If you dereference the `ref_my_integer` variable, you get an integer.

7.4 Using Pointers to Pass Parameters by Reference

The reason we started looking at pointers was to change the way parameters are passed into functions. The next thing you will do is to change the simple program “lab7” you met at the beginning of this lab so that the `change` function can alter the value of the argument variable in the `main` function.

This can be done by changing what we pass into the `change` function. Instead of passing the value of the variable `j` from the `main` function, we could pass a reference to `j` using the reference operator &. The `change` function should then be adjusted to accept and use a pointer to an integer instead of an integer.

This technique of passing a pointer rather than a value to a function is called *passing by reference*.

Exercise 7.4: Changing the “lab7” Program to Pass by Reference

Change the “lab7” program to pass the parameter into the `change` function by reference, rather than by value. You will have to change:

- The way in which the function is called. You will have to use the reference operator to create pointer.
- The parameter declaration part of the `change` function head. At the moment the `change` function accepts a single integer parameter. You should change this so that it accepts a pointer to an integer instead.
- The assignment statement inside the `change` function to use the dereference operator. By using the dereference operator you will affect the value of the `j` variable in the `main` function.

Try building and executing your program. You should find that the `change` function is now able to affect the value of `j`.

You may have noticed that you have been using the reference operator (`&`) since laboratory 2 in `scanf` lines like this:

```
scanf("%d", &my_integer);
```

This means that you are passing the `scanf` function a pointer to the `my_integer` variable, which allows it to change the value. This is another example of passing by reference.

Exercise 7.5: Using `scanf` with a Pointer

Alter the `change` function so that it asks the user what value they wish to set the variable to. It should then set the value of the location that `i` references using a `scanf` function call.

Be Careful! Remember that the variable your are using (`i`) is a pointer. Do you need to use the reference operator when calling `scanf`? If you don't understand this, ask one of the demonstrators to explain it to you before continuing.

Exercise 7.6: Passing Multiple Values by Reference

Because a pointer is just another parameter, a function can accept a number of pointers as parameters, as well as other values. There are no limitations on this.

Change the “lab7” program so that the `change` function accepts two (or more!) pointers as parameters, and sets the values of them using `scanf` function calls.

7.5 Using Pointers with Structures

One of the most powerful ways to use pointers is in combination with structures, like those you met in lab 6. By passing a pointer to a function by reference it allows that function to change any of the members of that structure.

Suppose we have a `student_type` structure type just like the ones you looked at in lab 6:

```
typedef struct
{
    char family_name[40];
    char given_name[40];
    int year_of_birth;
    int course_code;
} student_type;
```

We could declare a function to accept one parameter, which was a pointer to a structure of this type:

```
void change_student(student_type *student);
```

So that parameter `student` is a pointer to a structure of type `student_type`.

So how would the `change_student` function set the values in the structure? It **can't** do it like this:

```
student.year_of_birth = 1986;
```

because `student` is a *pointer* to structure **not** a structure. We need to use the dereference operator to follow the reference to find the actual `student_type` structure. It has to be done like this:

```
(*student).year_of_birth = 1986;
```

This dereferences the pointer `student` to find the structure, then uses the dot operator (`.`) to access the member variable `year_of_birth`. The brackets are necessary because otherwise the `'.'` takes precedence over the `'*'` and C thinks that the star refers to both the parts before and after the `'.'`.

Writing `(*student).member` is very cumbersome and difficult to read. To alleviate this problem, C includes a special operator for accessing the members of a structure to which you have a pointer. It looks like this: `->`. It is a combination of a minus sign, `-`, and a greater than sign, `>`, and is meant to look like an arrow. The line:

```
student->year_of_birth = 1986;
```

is exactly equivalent to:

```
(*student).year_of_birth = 1986;
```

The first notation is easier to read and is almost always used in place of the second.

Syntax: Pointer to Structure Member Access Operator

`->`

The `->` is used for accessing the member variables of a structure to which you have a pointer. As in:

```
struct_pointer->member;
```

For example:

```
my_struct->some_value
```

Accesses the member variable `some_value` from the structure which is referenced by the pointer variable `my_struct`. This is directly equivalent to:

```
(*my_struct).some_value
```

Exercise 7.7: Using Pointers with Structures

Open the program that you were working on in lab 6 as part of your option. This will be “graphics3” if you are were working on the graphics option, or “music3” if you were working on the music option.

Both these programs have a loop in the `main` function which accepts values from the user to put into a structure in the array. Create a separate function which accepts a pointer to a structure. It should be this function which prompts the user for values and places them in the structure.

The function prototype for those doing the graphics option might look something like this:

```
void get_line_values(line *new_line);
```

For those doing the music option it might look something like this:

```
void get_note_values(note *new_note);
```

You will have to use the reference operator when you call the function, and the `->` operator.

Exercise 7.8: Improving Program Efficiency Using Pointers

When you pass a structure by value all of the member variables are copied into the parameter. This can make your programs inefficient, both in terms of their speed, and the amount of memory they use.

Change the `draw_line` (graphics option) or the `play_note` (music option) function so that the structure is passed by reference rather than by value.

7.6 Summary

Now that you have finished this laboratory you should know that the usual way in which functions are called involves a copy of the values of the arguments into the function parameters and is called *passing by value*.

You should understand what a pointer is and be able to use the reference (`&`) and dereference (`*`) operators to work with pointers. The technique of using pointers as parameters to functions is called *passing by reference*, you should know how to write and call functions which use this technique.

In the final part of the lab you used pointers with structures. You should have met the `->` operator which is designed for working with structures. You should also have altered one of the programs you created in lab 6 so that it passed structures by reference.

