Chapter 2 The C Memory Model

In this chapter we will learn about how computer memory is organized, and how our programs use it. This is important because at the end of the day, our programs will be working with information stored in the computer's memory. Knowing what is stored where, and how to access and modify it, is essential for understanding what your program is doing.

2.1 Computer memory is like a room full of lockers

If you have been to the change room in a gym, or if you had a locker while you were in high school, you will have noticed the rows of lockers set up there for storage (Fig. 2.1). Row upon row of lockers, some empty, some full, the ones that are full are assigned to a person who has either a key or a combination that allows them to put things into, and take things out of their locker.

Figure 2.1: A locker room. Organized as rows and columns of numbered lockers, each with its own lock or key. *Image: Tiia Monto, Wikimedia Commons, CC-SA 3.0*

All locker rooms share a couple of important properties:

- Lockers are reserved - the owner of the locker has a key or combination that opens the door so only the owner can store things into, or take things out of the locker.
- Lockers are ordered by number in a way that makes it easy to find a specific one.

The process of reserving a locker changes from place to place and is not important to us right now. Neither is the size of the lockers, or the number of them in the locker room. What matters to us is that the properties listed above are very much the same properties of the main memory storage inside a computer. Indeed, at the lowest level, computer memory is just like a very large locker room:

- It consists of numbered boxes where we can store information
- The boxes are ordered by number in ascending order
- Boxes are reserved so that only the program using that box to store information can store or access the information in that box (we will see later what happens when we need to share)
- If you look at the microprocessor under a microscope, you can see the rows and columns of memory lockers (Fig. 2.2)

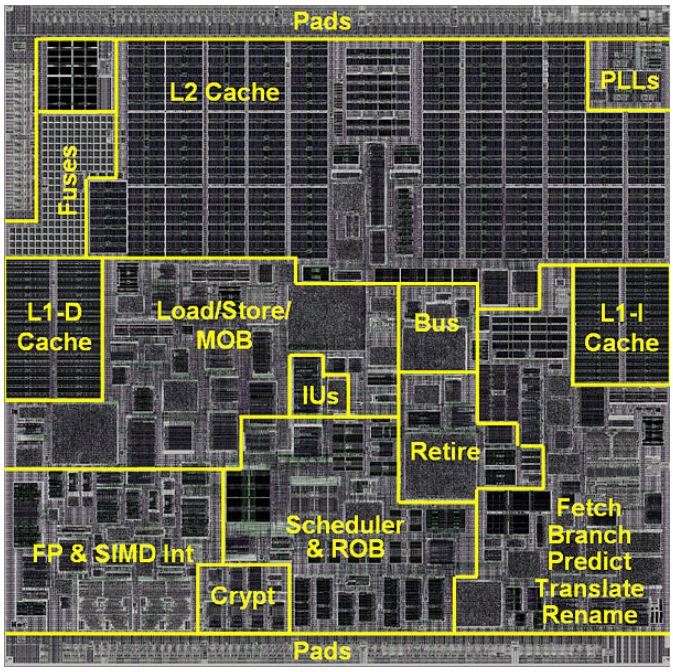

Figure 2.2: This is a micrograph of a CPU. Notice the rows and columns in the regions labeled L1 cache and L2 cache. These comprise this CPUs memory area, and they are indeed organized much like a regular locker room. You can learn all about how this works in any Computer Organization book. *Image: VIA Gallery, Wikimedia Commons, CC-SA 2.0*

For the purpose of this course, we will think of computer memory as nothing more than a large locker room where our programs keep their data. We will see that declaring variables, assigning values to these variables, and moving information between functions is very much the same thing as reserving a set of suitably sized lockers, storing things in them, and moving those things around.

2.2 What happens in memory when we declare a variable in C?

As we saw in the previous Chapter, **C** supports a small number of data types we can use to declare variables. Let's have a look at what happens in memory when we compile and run a program that does something very simple: Declaring a single integer variable and assigning a value to it.

Example 2.1

```
#include<stdio.h>
int main()
{
```

```
    int x;
    x=5;
}
```

Let's picture our computer's memory as that locker room we've been talking about - this could look like the image in Fig. 2.3.

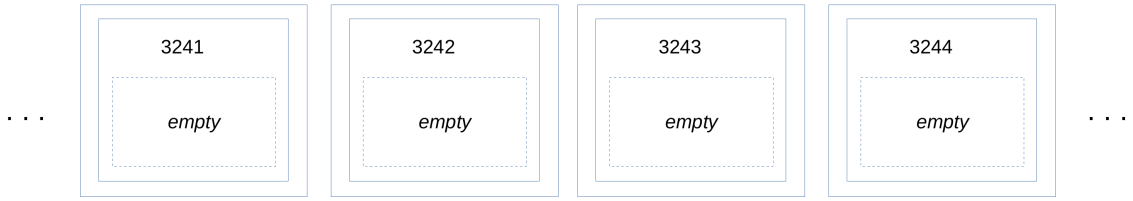

Figure 2.3: A section of the computer's memory where there's **empty** lockers - the specific locker numbers do not matter, they could be anything.

Figure 2.3 shows just a little section of memory, with 4 numbered boxes. The **numbers on them are not important**, just as the locker number you get when you go exercising doesn't matter. All that matters is that they are **empty**, and available for use by a program (in this case, our program!).

Note

What does **empty** mean? - The diagram above shows the four boxes as **empty**. This means that the computer knows these lockers are not reserved for use by any program. However, inside the actual computer memory **there may be junk left over by whatever program last used that box before**. So in practice, you should always assume that an **empty** locker contains junk until you change its contents to something useful. **Be careful with this**, it can cause bugs that are hard to fix because your program looks right – it's just using junk values left over inside a locker you haven't yet put anything meaningful into.

In **C**, **declaring** a variable (first line in Example 2.1):

```
int x;
```

means that we want to **reserve a locker**, to store **one integer value**, and we want to **tag that locker** with the name **x**. In the computer's memory, a locker suitable for storing one integer value is reserved for our program to use, and tagged with **x**, as shown in Fig. 2.4.

Figure 2.4: The same memory section from Fig. 2.3, after our program has requested space for an **integer** variable called **x**.

Note that:

- **'x' is not inside the box**, the box is still empty, the variable's name **x** is used as a tag so at any point when our program needs **x**, we can easily find the locker that contains the value of that variable
- **x** is also tagged as being of type **int**. Thereafter your **C** program will know what type of data is stored in that box
- Declaring a variable does not initialize it or set it to zero. As you see, the locker remains empty. Be careful with this, some compilers choose to reset variables to zero, and some do not. Or do so only in some cases and not in others. The **safe** practice is to assume your new variable contains **junk** after you initialize it, up until the point where you give it a meaningful value

In our example, locker #3241 is now reserved for our program's use, and it will store whatever values variable **x** will take during the program's execution. The next line in Example 2.1 contains an assignment operation:

```
x=5;
```

This literally means **go to the box tagged with x, and put a 5 inside it**. The end result is shown in Fig. 2.5.

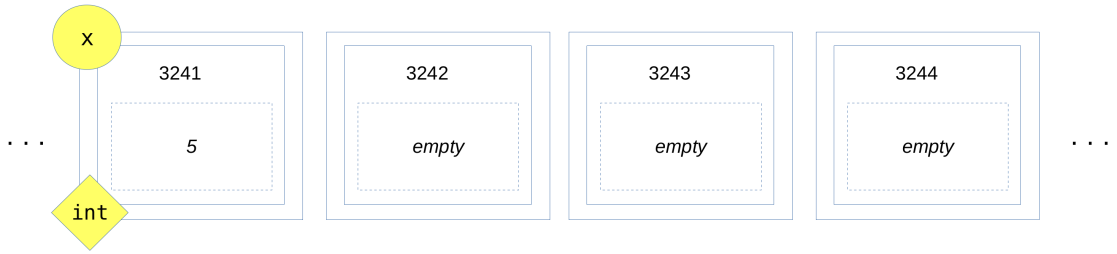

Figure 2.5: The same memory section from Fig. 2.4 after we perform an **assignment**, which puts data (an integer with value 5) inside the locker tagged **x**.

Of course, there could be a much more complicated expression right after the **equals** sign, in which case the expression is evaluated (it must result in a concrete data value) and the result is stored in the corresponding box.

Let's do a more interesting example to practice how all of this works. We will declare several variables, see what the corresponding situation is in memory after the declaration, and summarize the key ideas that explain how variables in **C** work in the computer's memory.

Example 2.2

```
main()
{
    int x;          // Reserve space for an integer variable called x
    float pi;       // Reserve space for a float variable called pi
    char c;         // Reserve space for a variable of type char called c

    x=5;            // Go to the box tagged 'x' and store a 5 there
    pi=3.141592;    // Go to the box tagged 'pi' and store 3.141592 there
    c='C';          // Go to the box tagged 'c' and store the character C in it.

    // The figure illustrates the situation in memory at *this* exact point.
}
```

The situation in memory after the code above is processed will look as shown in Fig. 2.6. We have three variables: **x**, **pi**, **c**, each with their own data type (int, float, and char respectively). We also have assigned values to these variables that make sense for their data type. In the memory model, we see that we have three reserved boxes, each of these corresponds to one of our variables **in the order in which they were declared**, each box is tagged with the variable's name, and the variable's data type, and each box contains the value that we assigned to the corresponding variable.

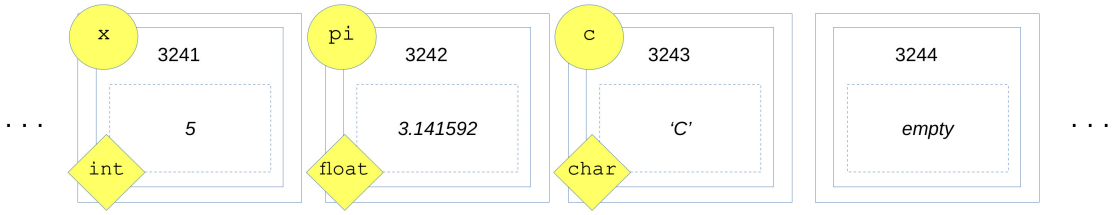

Figure 2.6: This shows the result of processing the code in example 2.2, we have three boxes reserved, and each of them contains a value as requested by the program.

Note

From the examples above, you should take home the following essential facts about variables:

- Variables are just boxes in memory where we can store information
- The variable's name is just a **tag** so the compiler knows which box we want to use
- Each variable **gets its own box**, and each box can have **only one tag**
- **Assignments** put values (data) inside boxes
- All your program (or anyone else's program for that matter) ever does is go into boxes, see or change what is in there, and move data around

Exercise 2.1

Draw a diagram like the one above showing what you'd expect would happen in memory if we compile and run the following program:

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

    x=5;
    y=x;        // What does this do?

    return 0;
}
```

2.3 Functions in C – how information moves about

In your introductory programming course, you learned that we break programs into small **functions** each of which has a specific task to perform, you learned that functions often take input **arguments**, and often **return** some

value that is the result of whatever processing the function does. The input arguments and the return value are the foundation of how information moves around inside a program.

Let's now take a look at how we **declare a function** in **C**, and discuss the important properties of how information moves around from one function to another as our program goes about its work. The example below declares a simple function. It has one input argument, and it computes and then **returns** the value corresponding to the square of the input argument.

Example 2.3

```
int square(int x)      // This is the function declaration
{
    int s;                 // The body of the function consists of anything contained
    s=x*x;                 // within the curly braces
    return s;
}
```

First off, let's spend a moment looking at the line that declares the function

```
   int square(int x)
//  ^     ^      ^
//  |     |      \------ Input parameter: an integer variable called 'x'
//  |     \------------- The name of the function
//  \------------------- The function's return type. This one returns an
//                       integer value
```

The declaration has 3 parts always:

- First off, the function's **return type** which tell the compiler the data type of whatever information the function will return. Functions in **C** can only return **a single data value**.
- The function's **name** - you should make sure to give each function a name that appropriately describes what it does.
- The function's **input arguments** - the function can have as many input arguments as needed, separated by commas. For each input argument, we need to provide the **data type**, and a **name**.

Note

Just like with variables, a function's **data type** is fixed. Once defined it can not be changed. The same goes for the argument list, once declared, the function will always expect the correct number of arguments and they all must have the correct data type. If you call a function with the wrong number or type of arguments you can expect the compiler to report an error or warning (depending on the severity of the issue).

After the function's declaration, we find the function's body which contains

- **Declarations for the variables** the function will use. Note that each input argument is **already a variable** usable by the function - for this reason, variables in a function must have a name that is different from any of the input arguments.
- The **function's code**, which can be as short or as long as needed. Good programming practice suggests you should keep functions on the shorter side as much as possible.

- The **return** statement, which **causes the function to send a result back** to whichever part of the program called the function. The function can return any of the variables or arguments it has available, but the value being returned must be of the correct data type.

That is all there is to function declarations in **C**. All the functions you will ever write in this language follow the rules stated above. Of course, a program will often contain many functions, so let's now look at an example where we have two functions, and we will use this example to understand what happens in memory when some part of a program uses a function to do some work.

Example 2.4

```
#include<stdio.h>

int square(int x)
{
    int s;
    s=x*x;
    return s;
}

int main()
{
    int input;
    int result;

    input=7;
    result=square(input);

    // Here we would normally print the result, we'll see how to do that very shortly!

    return 0;            // This is new! we'll explain below
}
```

Let's see what happens in memory when we compile and run the code above:

When the compiler is processing your program, it will note that **main()** declares two integer variables called **input** and **result**. These two variables will be assigned one locker each, next to each other, in the order in which they appear in the program. The compiler will also note that **main() is expected to return an int**. Therefore the compiler will make sure there is a **separate box reserved to store this return value**.

> **Note**
>
> Whatever value a function returns must be stored in its own box, and this box will be reserved right beside the last of the function's variables.

When we **run** the program, the instructions that are executed are the instructions in **main()**. In Example 2.4, the first thing the program does is to store a **7** in **input**. In the memory model, things will look as shown in Fig. 2.7 immediately after the **7** has been stored in the corresponding box.

The next line of the program is important to understand because **all function calls in C work in the same way**:

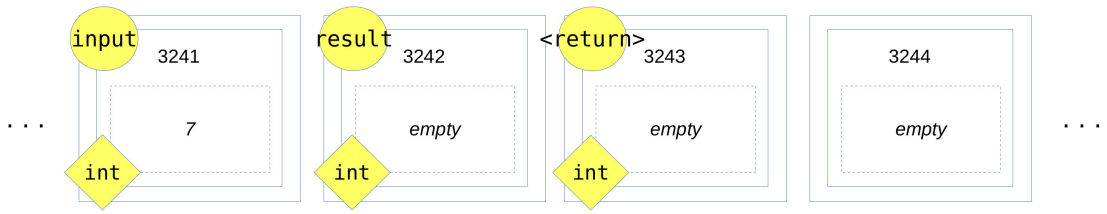

Figure 2.7: A memory model for the program in Example 2.4 up up to the line **input=7;**.

```
result=square(input);
```

It does the following:

1) It reserves memory for the function **square()** to use (notice that space for functions is only reserved at the time when they are called). From the function's declaration the compiler knows it needs space for one input argument **x**, one **local variable** *s*, and the function's **return value** which is of type **int**. This is shown in Fig. 2.8. Notice that the memory locations reserved for **square()** could be anywhere and not necessarily close to the lockers reserved for **main()**.

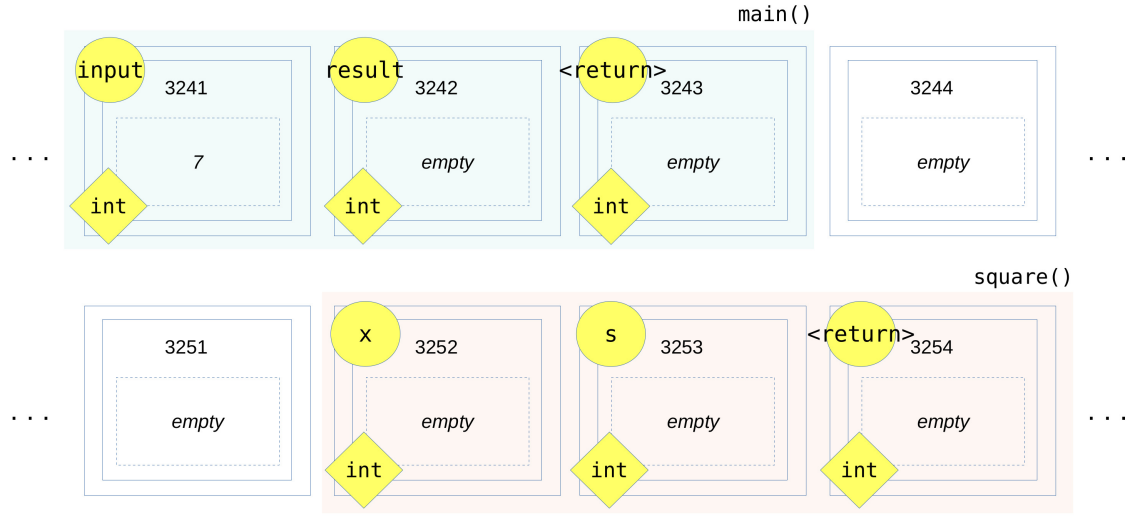

Figure 2.8: Memory model after lockers have been reserved for the **input argument**, **local variables**, and **return value** for function **square()**.

Note

The memory model has been annotated to show clearly which lockers were reserved by **main()** and which were reserved by **square()**. There is an essential property of the memory model that we should make sure to understand at this point: **variables and input arguments that belong to a function can only be accessed by the function that owns them**. This means that **main()** doesn't know anything about **square()**'s variables, their name, type, or where their lockers are. Similarly, **square()** doesn't know anything about **main()**'s variables, their type, or where they are. We say that the variables (and input arguments) declared by a function are **local** because **only the function that owns them can access/read/modify the corresponding boxes**.

Definition 2.1

The **scope** *of a variable is the region within a program's code where this variable can be referenced and manipulated.* ♣

The **scope** of all the **local variables** and **input arguments** declared within a function is the code contained within the curly brackets that delimit the start and end of the function. **There are other possible scopes** for variables in **C**, and we will learn about them as we progress through the course.

2) It **calls the function**, which means: It passes the required arguments to the function and then proceeds with whatever instructions are part of the function itself. In the example, the function was called from **main()** using the **local variable** *input* as the argument to **square()**. So the value of **input** is **copied onto the box reserved for** *x*. This is shown in Fig. 2.9.

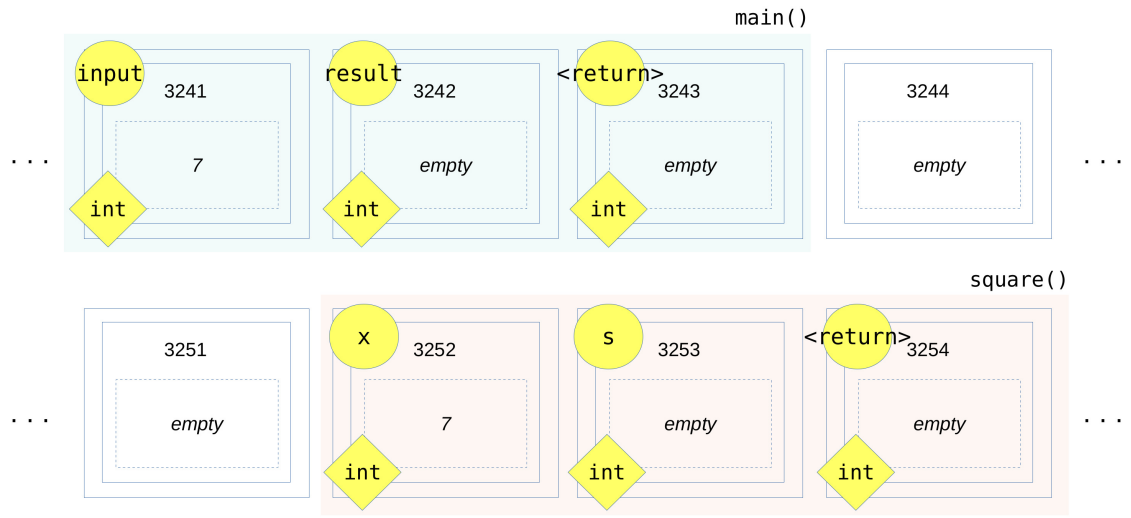

Figure 2.9: Calling the function means **making a copy** of values passed to the function into the corresponding **input arguments**.

3) The program then continues with the instructions that is are part of function **square()**. In this case, it evaluates the square of the input argument and stores it in the **local variable** *s*. The last line in **square()** contains

the **return** statement. In this case, it returns whatever the value of the **local variable** *s* is at that point. **This makes a copy of the value of** *s* **and stores it in the box reserved for the** *return value* **of square()**, as shown in Fig. 2.10.

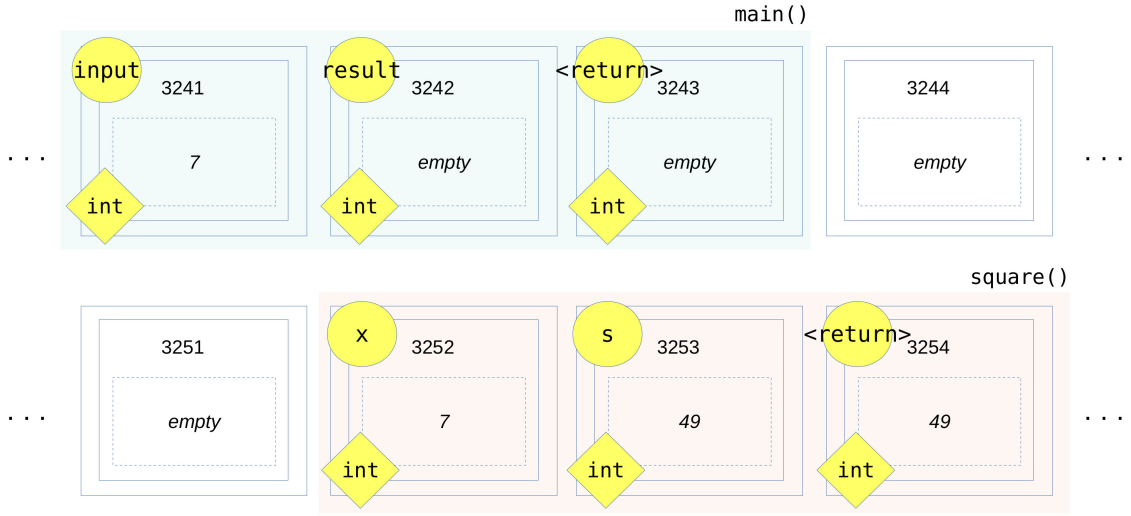

Figure 2.10: Memory model after the instructions in **square()** are carried out, the last line copies the value we want to return into the box reserved for the function's **return value**.

4) The program then returns to **main()** exactly to the line where **square()** was called, and copies the **return value of square()** onto the **local variable** *result*. This is illustrated in Fig. 2.11

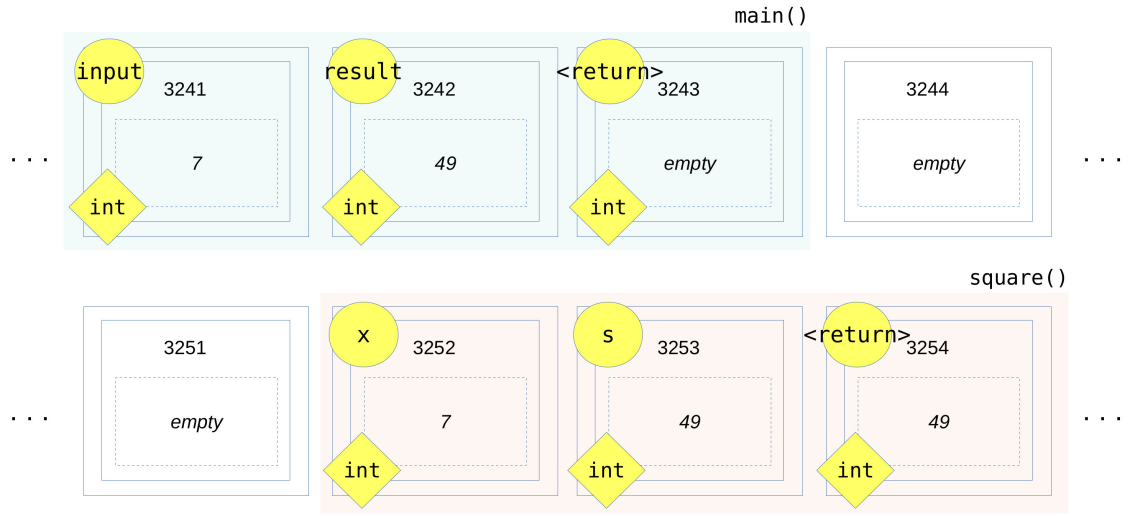

Figure 2.11: When the function **returns**, whatever is stored in its **return value** box will be **copied** into **main()'s local variable** *result*.

5) Because the work of **square()** is complete, the memory reserved for the function is **released**, which means these lockers become available for any other function or program to use for their own work. This is important, **memory reserved for a function is only reserved for as long as the function is doing work**. As soon as the

function returns, its reserved lockers are released. The final state of the memory for our program will look as shown in Fig. 2.12.

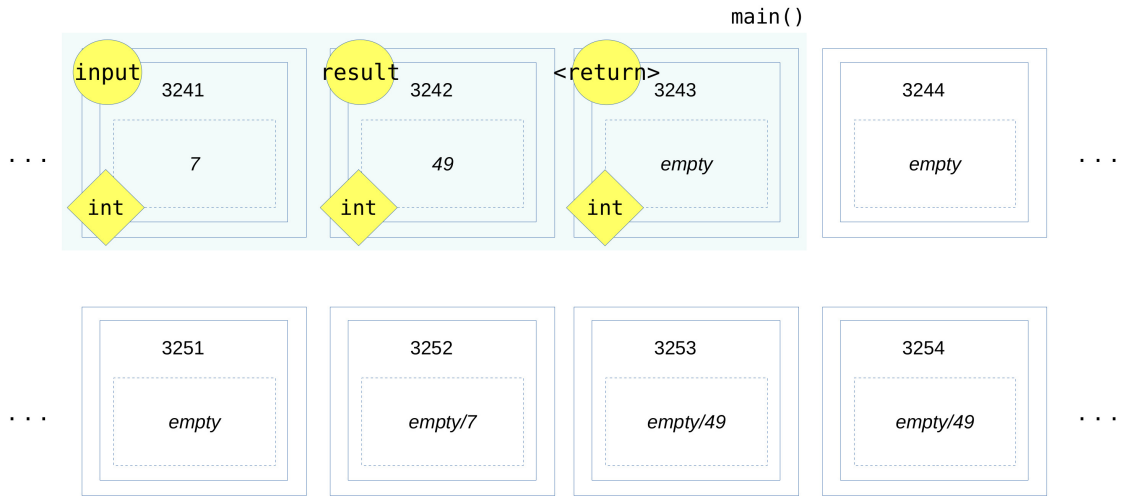

Figure 2.12: Memory model for the program after the function **square()** has returned, its work is done, and memory reserved for it has been released.

Note that **the computer does not clean up** the boxes that were used by **square**. They are marked as **empty**, and they can be reserved by any other program or function, but whatever was left there, is still there. This is why, as discussed previously, we should assume that lockers assigned to our program will contain **junk information** until we put something meaningful there.

After the call to **square()**, the program continues with any instructions left in **main()**, which could, for example, print the result, or there could be more function calls, etc. In the example above there's no more work for **main()** to do, and we find a **return** statement that signals the end of **main**. Why does **main()** have a return value? who gets that return value?.

Note

All programs have to be **started by something/someone**. The simple programs we have written so far are launched by us from a terminal, but the same thing happens when you double click on a program on your computer's desktop, or when another application launches a helper program to do some work. The point is, the program **is called by something outside itself, much the same way functions are called within a program**. So, by **convention**, a **C** program must have a **main()** function that returns an **integer value**. The point of this return value is to provide information to whatever user or application launched the program. Most of the time, the purpose of the return value is to **indicate whether the program completed its work correctly, or conversely, whether there was some error and the program did not finish properly**. The expected return values are defined as:

- **0** indicates the program completed correctly, with no errors
- **any positive integer greater than zero** indicates an error occurred, and the value of the integer can be used as an error code to indicate what the error was. There is no standard for what non-zero return values mean, it is up to the application to provide documentation that explains different possible error codes

2.3.1 Summary

From the examples above and the discussion on how information moves around between different functions in a program, you should make sure to remember the following principles that apply to **every function, and function call in C**.

- Each function will have **its own memory space**. Functions can only access/change information that is stored in boxes reserved for the function, no other part of the program has access to these boxes.
- Because each function has its own lockers, we say that variables and input arguments are **local to the function**.
- For each function, space is reserved for **each input argument** in the order in which they appear in the function declaration, then space is reserved for **each local variable** in the order in which they are declared, and finally a box is reserved for the function's **return value**.
- Multiple functions can have variables or input arguments with the same name, since **they each have their own box** within **the memory space reserved for their function**. There is no confusion about which variable some part of the program is using.
- Information is **passed into the function** by making copies of the **variables passed to the function in the function call** into the corresponding boxes for the function's **input arguments**. This is an essential property of how functions in **C** work. We say that arguments are **passed by value** into **C** functions because we provide the function with a copy of the values it is being passed as arguments.
- Information is **passed back outside the function** by copying the value stored in the function's **return value** box into the **specified local variable** outside of the function.
- A function can only return **a single data item**.
- Once the function's work is done, the space reserved for the function is **released** for use by other functions or programs. This means functions only use memory space when they are **actively doing work**.

✍ **Exercise 2.2** Show in in a memory model diagram what the contents of memory would look like **just before the space reserved for** *square()* is released. Your memory model diagram should clearly show **each function's memory space**, and for each function it should contain **the function's arguments if any**, the **local variables**, and the **return value**. These should be in the order discussed above. Do not forget to **tag each box with the variable's name and data type.**

```
#include<stdio.h>

int square(int x)
{
    x=x*x;
    return x;
}

int main()
{
    int x;
    x=9;
    square(x);

    // What would be the value of main()'s x variable if we printed it out in *this* line?

    return 0;
}
```

Question: What would be the final value for the local variable **x** declared by **main**()?

2.3.2 Input arguments and type conversions

You know by now that all variables, input arguments, and function return values have an associated data type. You can not change the data type at run-time. However, **certain type conversions are possible during the process of passing arguments to a function**, as well as during the **assignment of return values to variables**. Let's see an example:

```
#include<stdio.h>

float square(float v)
{
    // Now this function takes as input a floating point number,
    // and produces a corresponding floating point result.

    return v*v;    // In C, we can do a little bit of
                   // processing in the return statement!
}

int main()
{
    float pi;
    int x,y;
    int result;

    x=5;

    // Notice we will call square() with an int, not a float,
    // and we are assigning the result (which is float) to an int.
    y=square(x);
```

```
    printf("The first result is %d\n",y); // What does this print out?

    pi=3.14159265;
    y=square(pi);
    printf("The second result is %d\n",y); // What does this print out?

    return 0;
}
```

Note that we have changed the declaration of the function square(). It now takes as input argument a floating point value **v** and returns a floating point number that is the square of **v**. This time we shortened the function a bit so the calculation happens right at the return statement. Draw for yourself a diagram of what's going on in memory when you call this function.

Here's what happens when we compile and run the little program:

```
>./a.out
The first result is 25
The second result is 9
```

What's going on here?

- The first time we call **square()**, we are passing in **x** which is of type **int**. Because **square()** expects a float as an input parameter, the value of **x** which is **5** is converted into a **floating point** value **5.0**, and then stored in the corresponding input argument **v**.
- This process of converting information from one data type to another is called **type-casting**.
- After **square()** has done its work, the result which is the **floating point** value **25.0** is returned. However, it is being assigned to local variable **y** which is of type **int**. The floating point value **25.0** is type-cast to the integer value **25** and stored in **y** - so the first print statement prints out **25**.
- The second time we call **square()** we are passing in **pi** which is of type **float**. No conversion is necessary, so the value **3.14159265** is copied onto **v** and the square is computed and returned.
- However, the result, **9.869604** is being assigned to local variable **y** which is an **int**. Once more, the floating point value is converted to an integer. **This is done by chopping off the fractional part, the conversion does not round the value to the nearest integer**. The resulting value assigned to **y** is **9**, and this is what the second print statement outputs.

Note

The **conversion**, or **type casting**, is transparent to you but you must always be aware when a type conversion is taking place, as this is a place where bugs can easily creep into your program. In fact, good programming practice would have you make sure type casting only happens when the programmer explicitly intended for it to happen.

There is a consistent set of rules used to type-cast values in C. You can learn how these rules work, but it is a bad idea to write code that assumes you know these rules perfectly – worse, it makes your code hard to debug: Anyone reading the code would have to be an expert to identify and solve problems introduced by unintended type-casting. Here's an example of why this is important.

Example 2.5

```
#include<stdio.h>

int main()
{
    int x,y;
    float result;

    x=5;
    y=2;
    result=x/y;
    printf("The result is: %f\n",result);

    return 0;
}
```

Compiling and running the code above produces:

```
./a.out
The result is: 2.000000
```

The result is not correct despite the fact that **result** was declared as a **float** and should be able to represent the value of **5/2**. However, because the expression **x/y** must be evaluated first, and since both **x** and **y** are **int**, the division carried out is an **integer division** which gives the integer result of **2**. The integer **2** then is **type-cast** into a floating point value **2.0**. **To ensure things work out properly in your code you must explicitly indicate when type conversions should happen, and what data type the conversion should result in.** A correctly written version of the code above would look like this:

```
#include<stdio.h>

int main()
{
    int x,y;
    float result;

    x=5;
    y=2;
    result=(float)x/(float)y;
    printf("The result is: %f\n",result);

    return 0;
}
```

In the program above, the expression **(float)x** indicates we **are requesting the compiler perform a type conversion for us**. In this case, it tells the compiler to convert the value of **x** into a **float** before it's used. Likewise **(float)y** tells the compiler to convert the value of **y** into a **float** before using it. Since the division is now working with two floating point numbers, we should expect the result to come out right:

```
./a.out
The result is: 2.500000
```

The code above is an example of **explicit type casting**. We tell the compiler **when** to perform a type conversion, and **what data type** we want the result to have. It is essential that we do this whenever we are writing code that uses **mixed data types** such as **integers** and **floats** (this is quite common).

Note

You **must be very careful** when performing type conversions to ensure that the result makes sense and is valid. This is an issue because **floating point** variables can store values that are much, much larger (or much, much smaller) than any integer can handle. So for instance, if you tried to type-cast the floating point value **12345678987654321.0**, which has no fractional par, into an integer; you would get **4294967295**. This is clearly wrong! but happens because the original value is too large for an integer to hold. **This is unavoidable** and a property of these two data types. Similarly, any floating point value smaller than **1**, will give you **zero**.

Not only must you perform the type conversions explicitly in your program (as shown above), but you also **have to implement safeguards** to make sure the values that result from the conversion will be valid within the context of your program, and will not cause a bug. To summarize, you should always carry out **explicit type conversion** because

- It will help you keep in mind a complete and clear picture of what your code is doing, what the expected inputs and outputs to computations and functions should be, and ensures that data is being manipulated as you expected at all times.
- It greatly reduces the possibility that you will run into unexpected bugs caused by type conversions being performed without appropriate safeguards. This is no small thing, Fig. 2.13 shows an example of the kind of major fault that can result from not being careful with type conversion.

Figure 2.13: An Ariane-5 rocket self-destructed after takeoff due to software error caused by an incorrect type conversion. The cost of this mistake was over 370 million dollars. *Photo: ARIANE 5 Flight 501 Failure Report by the Inquiry Board (http://sunnyday.mit.edu/nasa-class/Ariane5-report.html), Public Domain*

Note

So why are data types such a big deal?

All the information in digital computer is stored in the form of bits, so every piece of information you will ever manipulate with a standard digital computer, from integers, to floats, to music videos must be represented, in some way, as a string of ones and zeros.

How the bits are interpreted to represent information depends on what type of information is being stored. Integers and floating point numbers have completely different binary representations. This means that the circuitry inside the CPU that carries out operations with these different types of binary data can not mix and match bit strings for integers with those for floating point numbers.

Therefore, all CPUs have separate instructions for doing arithmetic and logic operations on integers, and on floating point values. You can see that these circuits are even located on different parts of the CPU surface by looking at a micro photograph of the CPUs circuitry in Fig. 2.14.

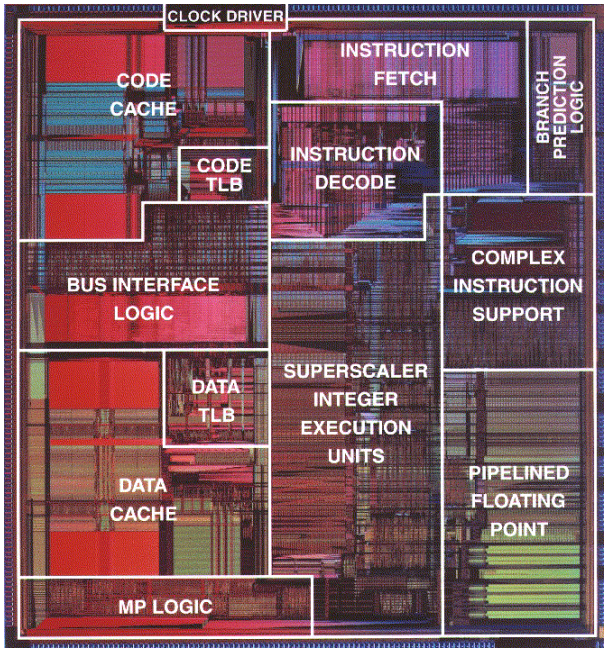

Figure 2.14: Micrograph of a Pentium CPU showing the separate circuits for the integer (labeled Superscalar Integer Execution Units), and floating point (labeled Pipelined Floating Point) arithmetic units. *Photo courtesy of: CPSC 3300 Computer Systems Organization, Clemson University*

2.4 Arrays

In **C**, we can reserve space for more than one data item of a given type. The resulting set of items is called an **array**, and it is the simplest data structure that will allow you to work with sets of data.

Here's how we declare an array in C:

```
#include<stdio.h>
```

```
int main()
{
    int my_array[10]; // This is an array of 10 integer values

    my_array[0]=10; // This is the first entry in the array
    my_array[9]=5;  // This is the last entry in the array

    return 0;
}
```

The syntax should look familiar to you, and you've likely used similar syntax to manipulate lists in Python. Be aware that **arrays are very different and much simpler than Python lists**. The only operations they support are reading and storing values in the different array locations.

In the memory model, the array declaration shown above causes **a group of 10 consecutive lockers, all of them able to store an integer** to be reserved for the program. The **first box, with the lowest locker number** is tagged with the **array name** and **array type**. This is shown in Fig. 2.15.

Figure 2.15: Memory model for an integer array with 10 entries, the first and last box were assigned values by the program, the rest remain **empty**.

All array declarations work the same way, the only thing that changes is the number of lockers reserved (which depends on the size specified in the array declaration), and the data type for the entries. You can create arrays with any valid data type supported by **C**.

Important facts about arrays that you must remember:

- The **size of the array is fixed**. Once you declare the array, you can not extend it. This is unlike the lists and dictionaries you are used to from Python which can grow whenever needed. You need to think carefully and in advance about how much space your program will need.
- The lockers reserved for the entries in an array **are always contiguous**, arrays can not have their entries scattered all over memory.
- Indexing is the same as in Python: **array[0]** is the first entry, **array[N-1]** is the last entry in an array with **N** entries. The only valid indexes are integers in **[0,N-1]**. Negative indexes are not allowed, neither are floating point values allowed as indexes.
- Array indexes are **offsets**, if you request to access **array[3]**, the compiler finds the first locker reserved for the array, and then looks for the box that is **3** lockers after it.
- All entries in the array have the same type, you can not mix data types within an array.
- **C** will not warn you or protect you from trying to access array entries outside valid index range. You must be extra careful with this, as it is a common source of bad bugs.

The last point in the list above is particularly important, as array indexing problems are among the most common, and often tricky to find sources of bugs in programs. Let's take for instance the sample array described above with **10** entries. We know this means that valid array indexes must be within **[0,9]**. But our program can request the following:

```
array[10]=5;     // Indexing beyond the array
```

This will in effect try to store an integer value of **5** in a location that is **10** lockers after the first box in the array. This box **has not been reserved for the array**, so our access request is **not valid**. At that point one of several things will happen (and we can not always predict which of them will occur, even for the same program, on the same computer, different things could happen at different times):

- The program will continue to run after the line that contains the invalid access. It will store a **5** at the locker you requested – **this is a bug**, we are overwriting something else. That box could be reserved for **another local variable or array**, it could be data from **another function in your code**, or it could be **empty space**. It will be hard to detect and fix this bug because the program continues as if there was no problem. The program will show unpredictable behaviour **and may or may not do the same thing at different times**.
- The code may **crash** (stop working before the normal end of its work) – this is obviously a bug, in this case the box we're trying to write to belongs to another program or is otherwise reserved. The computer's operating system terminates our program because it has tried to access data it doesn't own. Once more, this may happen unpredictably. **Sometimes the program may finish without crashing, other times it will crash**, or **it may work on one computer and not another**, or **it may work in Windows but not Mac or vice versa**.
- Another possibility is that the program continues past the line that has the invalid access, but the effect of changing a value in a box outside the array causes a problem later on, at a different point in the program. This can result in erroneous values being computed, unpredictable or unexpected behaviour from the program, or a **crash** at a different point in the code.

All of these are situations we must avoid. **Unpredictability** is not acceptable in a program, and array indexing

issues are a common source of unpredictable behaviour. Therefore, be very, very careful when indexing into arrays in **C**.

> **Note**
>
> If your program is behaving in unexpected ways, check your array indexing and make sure you don't have indexes out of bounds at any point.

✍ **Exercise 2.3** Write a small program that creates an array for **10 floating point values**. Then fills this array with multiples of **pi**. That is:

```
array[0]=1*pi;
array[1]=2*pi;
array[2]=3*pi;
// etc... maybe you want to use a loop...
```

Have your program print out the entries in the array in separate lines once the array has been filled with the correct values.

✍ **Exercise 2.4** Modify the program you wrote for Exercise 2.3 so that there is a second array, this array should be of **integer type**, and the entries of this array are filled by **type-casting to int** the corresponding entry of the floating point array that contains the multiples of **pi**.

Make sure your program prints out both the **floating point** value, and the corresponding **int** value for each index in the arrays.

✍ **Exercise 2.5** Modify the program you wrote for Exercise 2.4 so that instead of using **type-casting**, the entries in the **integer array** are obtained by **rounding** the corresponding floating point value to the nearest integer.

You should write your own separate function that takes as input a floating point value, and returns the corresponding nearest integer.

2.4.1 Strings in C

One of the most common uses of arrays in **C** is for storing and manipulating **strings**. Indeed, in **C** a **string** is nothing more than an array of individual characters. This makes it very different from the strings you may have used in Python (and which provided you with all kinds of functionality). Strings in **C** are incredibly simple, and working with them becomes a matter of knowing how to access and manipulate entries in arrays.

Here's what a typical declaration for a **string** looks like in **C**

```
char   a_string[1024];
```

Because **strings** are just arrays, they **follow exactly the same rules as arrays of integer, float, or any other data type**. Importantly, the size of the string is fixed and can not be changed after the string has been declared, and **initially the contents are junk**. The declaration above reserves space for up to **1024** characters, and we will not be able to store any strings longer than that (if we try, we will get into the same problems we described earlier regarding invalid indexing into arrays).

The string behaves exactly like any other array in **C**, so we can access and modify entries in the array as we normally would expect.

```
a_string[0]='H';
a_string[1]='e';
a_string[2]='l';
a_string[3]='l';
a_string[4]='o';
a_string[5]='\0';
```

The above code changes the contents of our string to contain the characters 'H','e','l','l','o','\0'.

Definition 2.2

The last entry is especially important, it is the **end-of-string delimiter***, in our program, we write it as a* **'\0'***. But it represents* **a single character***, not two. The* **'\0'** *is just a special sequence to tell the compiler we want to mark the end of a string.*

♣

Note

Why do strings need a delimiter?

Because we may want to store different text sequences within the same string array. The reason we declared the original array to have **1024** entries is that we want to have enough space that we can store a pretty large variety of text sequences without worrying they won't fit inside the string array.
Most of the text sequences we may be interested in **will be shorter than 1024 characters**. The example above contains only the text **Hello**, 6 characters altogether. But the string array contains **1024** entries and the remaining ones will be full of **junk**.
Any functions that need to work on the text of the string (for example, to print out what the string contains) need to know which entries contain **valid text, set by our program**, and **what entries are unassigned junk**. The end-of-string delimiter is used to tell these functions where the valid text stops. Anything stored in the array after the end-of-string delimiter is ignored.

A common source of bugs stems from using the end-of-string delimiter incorrectly (e.g. putting in the wrong place, or alternately, forgetting to put it where it should be). **All valid C strings must have an end-of-string delimiter** just after the valid text portion of the string.

Question: If the string array contains **1024** entries, what is the maximum number of regular text characters that we can store in this string?

Of course, updating strings character by character would be a very annoying process. Luckily, there is a standard **C** library that contains functions you can use to manipulate strings.

```
#include<stdio.h>
#include<string.h>    // - This is the C library for string management

int main()
{
```

```
    char  a_string[1024];

    // Let's initialize a string - we can do that with the strcpy() (string copy) function
    strcpy(a_string, "This is a message for those learning C.\n");

    // Let's print out what we have stored in the string at this point:
    printf("%s\n",a_string);

    // Let's now concatenate another string to what we already have using the strcat() (string
        concatenate) function
    strcat(a_string, "Don't forget to practice using strings!\n");

    // Let's print out the string after concatenation
    printf("%s\n",a_string);

    return 0;
}
```

Compiling and running the program above produces

```
>./a.out
This is a message for those learning C.

This is a message for those learning C.
Don't forget to practice using strings!
```

A couple of things worth noticing:

A single string can contain multiple lines of text (we can split text into separate lines by using the special end-of-line character **n**).

The functions in the **string library** automatically make sure the end-of-string delimiter is properly placed after each manipulation of the string.

In the examples above we used **constant string values** (the text sequences within double quotes in the program above) to **initialize** the string and to **concatenate** onto the string. But functions in the **string library** can use other **string arrays** as well, for instance, we can copy one string onto another with **strcpy()**, and we can concatenate the contents of two string arrays using **strcat()**.

There are many more functions in the **string library** that you will find useful. For example, **strlen()** returns the **length of the string** - this is the number of characters of valid text, not including the end-of-string delimiter. Another useful function **strcmp()** can compare the contents of two strings to tell you whether they are identical or different.

If you need to look up how to use these function, a handy resource is here: https://www.cs.bu.edu/teaching/cpp/string/cstring/. This is from the computer science department at Boston University. There are many other references and examples for using string library functions, so as ever, if you have a problem, Google is your friend!

Note

A final thought regarding strings: **it is easy to run intro trouble with index-out-of-bounds issues** when manipulating strings. In particular, if we are concatenating strings, it is easy to end up with a text sequence that is too long for the array that is meant to store it. This would create mayhem in memory as the string manipulating functions will try to write to parts of memory that are not part of the string. You **must be careful** to ensure all the text sequences your program will work with fit within the arrays reserved for the strings themselves.

2.5 What is a pointer, and why do we need them?

Let's review the original program we wrote to compute the square of a number:

```
#include<stdio.h>

int square(int y)
{
    return y*y;
}

int main()
{
    int x;
    x=9;
    x=square(x);

    return 0;
}
```

You have seen this program before, **main()** will reserve space in memory for one integer variable called **x**, and set it to **9**. When we call **square()**, it will reserve space for an integer parameter called **y** and for an integer **return value** which will be set to **y*y**. This is shown in Fig. 2.16.

The point we need to make here is that the only way for **square()** to change what is stored in **main()'s local variable** *x* is by returning a value. This is the consequence of variables being local to functions, as we have discussed in detail previously. This is perfectly fine for a wide range of functions we will ever need to implement, and it is a **safe** way to write code, because the locality of variables ensures functions can not go around changing data that they do not own.

However, there are situations where we want a function to be able to **directly access, read, or modify** information that is stored somewhere else, and that is owned by a different part of the program. For example, this is a common situation when working with **arrays**. Let's say you have a program in which **main()** declared and is using an array with one million floating point values (for example, this could be a simulation of weather patterns, or it could contain statistics about stock prices, or it could represent the location of vehicles in a city grid). Now imagine that every time **main()** calls another function to do some work that requires the array **we make copies of the array** so each function has its own **local copy** as dictated by the principles of **local variables** and **passing input arguments by value**.

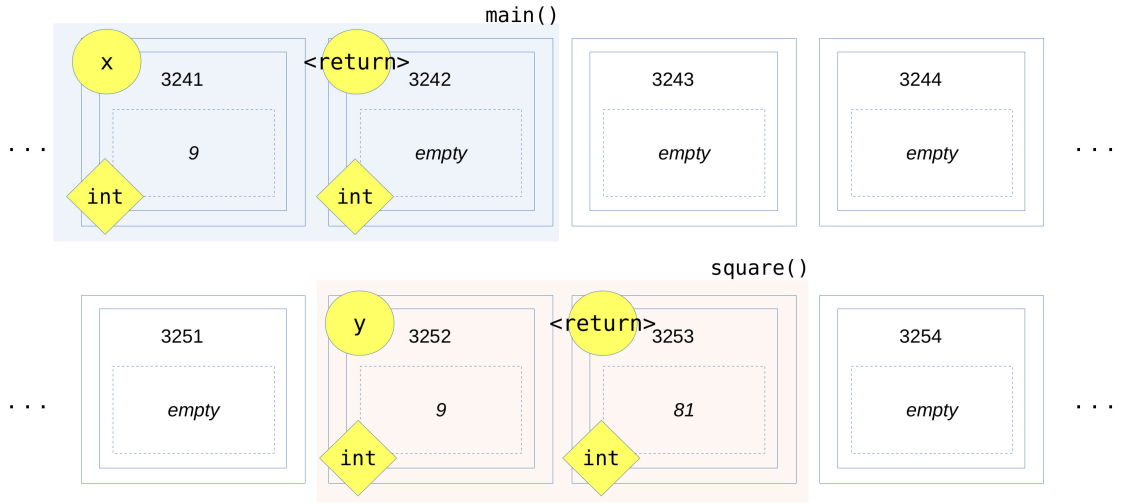

Figure 2.16: Memory model for the short program that computes the square of a number. This is what memory would look like **just before the memory used by** *square()* **is released**.

Clearly, doing this would result in a lot of space being used for copies of the array, and a lot of time and work would be spent simply creating and maintaining these copies so that they are **in sync** - all of these copies represent the same data, so they should contain the same values once work on them has been done.

Note

This is not a good way to handle large collections of data. An array with one million entries is actually fairly small, by today's standards. A single video file with about 30 minutes worth of video can easily require one thousand million array entries to fully load into memory.

In **C**, there is a **simple and intuitive way** to allow a function to access information it doesn't own, it's called **a pointer**.

To understand what a **pointer** is, let's start with an analogy to our locker room model for memory:

You reserve a locker at your gym to keep your clothes and toiletries. You know the locker number so you know where your locker is, and the gym **reserved the locker for you** *by giving you a key so only you can open the locker and access what's inside. As you are exercising (maybe playing basketball) with your friends, you realize you forgot your water bottle in the locker. Lucky for you, one of your friends is heading there to fetch their phone* **so you ask them to go to your locker and get your water bottle for you**. *You* **tell your friend your locker's number** *so they can find it -* **in effect you just gave your friend a pointer to your locker!**, *and you share your key so they can open the locker and get the bottle for you. One should always stay well hydrated.*

Definition 2.3

In **C**, *a pointer is simply a* **variable** *that contains a* **locker number** *that* **indicates where some information the program needs to use is stored**. *Like all other variables declared by a function, it* **will have its own box in the memory model**, *and has an* **associated data type** *that tells you* **the data type of the information stored in the locker** *that the pointer is referring to.*

There is nothing mysterious about **pointers**, they are nothing more than **locker numbers** we use to go find a

specific box whose data we need.

In **C**, pointers have their own particular (and, to be fair, somewhat clunky) syntax. There are three fundamental operations we need to learn to do with pointers: **declare a pointer**, **assign a locker number to the pointer**, and **use the pointer to access information**. Let's see how these work by way of an example:

Example 2.6

```
#include<stdio.h>

int main()
{
    int my_int;
    int *p=NULL;          // A pointer to an int!

    my_int=10;            // Change the value in 'my_int' directly
    printf("My int is: %d\n",my_int);

    p=&my_int;            // Put 'the address of my_int' in pointer p
    *(p)=21;              // Use the pointer to change the value of 'my_int'
    printf("My int is: %d\n",my_int);

    return 0;
}
```

Let's see what's happening here. First, **main()** reserves space for two variables, an integer variable called **my_int**, and a **pointer to an integer variable** called **p**.

The **'*'** in the line:

```
int   *p=NULL;
```

specifies that we are declaring a **pointer variable**, and the associated data type tells us **the pointer will keep track of the locker where we stored an int**. Importantly, the pointer is initialized to **NULL** to indicate it is initially **not assigned** to anything (we have not put a valid locker number in it just yet). In memory, this looks as shown in Fig. 2.17.

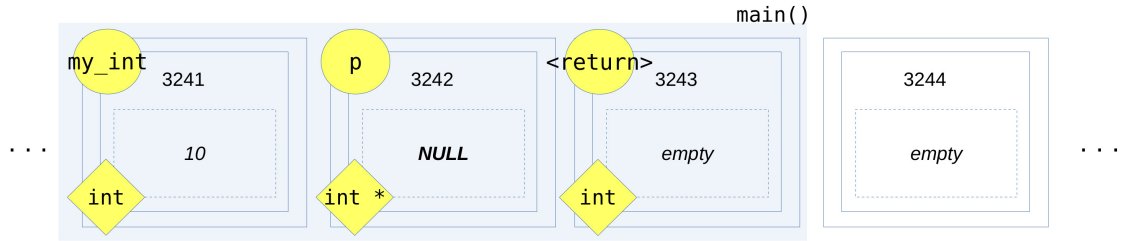

Figure 2.17: Memory model for the program in Example 2.6 up to the line **my_int=10;**.

The next line is something we haven't seen before:

```
p=&my_int;       // Put 'the address of my_int' in pointer p
```

this should be read as **get the address of the variable called** *my_int* **and store it in** *p*. Whenever you find a statement like this in **C** you should **translate it into the corresponding sentence in English** so that you clearly

understand what the program is doing. The **'&'** symbol is read as **'the address of'**. The effect of this line of code is that the compiler **looks up the locker number for the box where** *my_int* **is stored** and copies that locker number into **p**. This is exactly the same thing that happens when you give your friend the number of your locker at the gym. The result of this is shown in Fig 2.18.

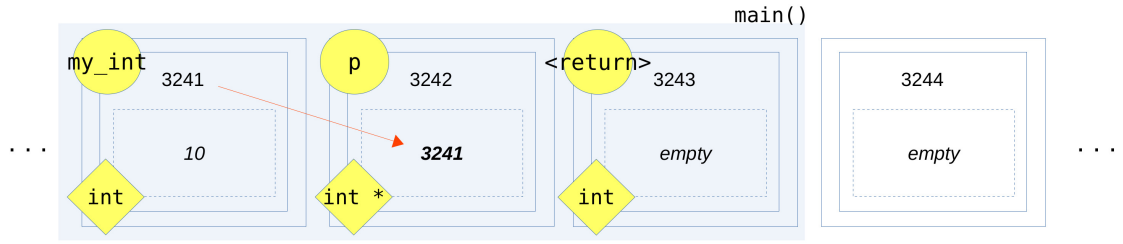

Figure 2.18: Memory model for the program in Example 2.6 up to the line **p=&my_int;**.

So now, our pointer **p** stores the value of the locker number reserved for **my_int**, and **we can use the pointer to access or even change what is stored in that locker**.

The next line does exactly that:

```
*(p)=21;
```

This should be read as **go to the locker number specified by (p) and store 21 in there**. Whenever you see a sentence like this:

```
*(a pointer or pointer expression) = expression
```

or

```
variable = *(a pointer or pointer expression)
```

You should read the **'*'** as **the contents of** (whatever is in the pointer expression within the parentheses).

So the instruction ***(p)=21;** becomes **make the contents of (p) equal to the value 21**. We know that **p** has the value **3241**, so the instruction really says **make the contents of box (3241) equal to the value 21**. We are telling the compiler to go to a specific locker, and put a particular value in that locker. This is shown in Fig. 2.19. Thereafter, if we print the value of **my_int**, we will see it is **21**.

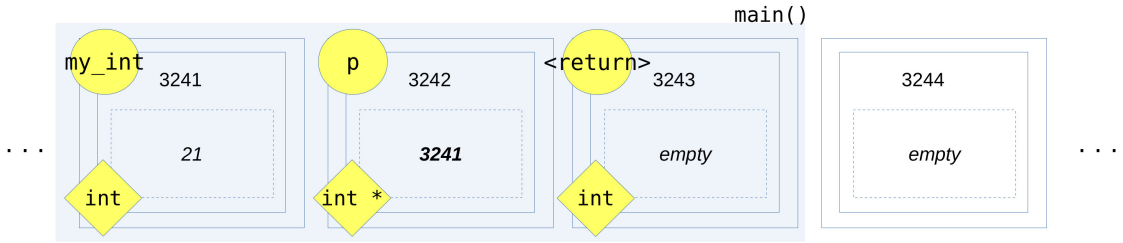

Figure 2.19: This is the result of using pointer **p** to change the value of **my_int**.

What we just did is: **we used a locker number to directly access a specific box in memory**. We did that without using the name of the corresponding variable which is important because we said before that each box in

memory can only have one tag. However, now we have two different ways to access box **3241**. One is by using the local variable **my_int**, and the second one is by using pointer **p**. There really is **no practical difference** in terms of using one or the other of these two methods, both will allow you to read or modify the value stored at box **3241**. However, the advantage of using a pointer is that we can have the same pointer variable **point to different lockers** at different points in the program. So, unlike the local variable **my_int** which is always attached to the same box, our pointer **p** can be used to access different data items at different points in the program.

Note

Using a **pointer** to access a locker directly will work **as long as the locker is reserved by some function in our program**. We can not go poking around memory that is **not reserved by our program** by using pointers to lockers the program doesn't own. If we try that, the computer's operating system will terminate (end/crash) our program. This is a common bug. **If your program does not complete correctly** (it ends without finishing its work, seemingly for no reason) or if the computer reports a **segmentation fault**, that is a clear sign that there is a problem with the way your program is accessing memory locations. This can be because the program is using pointers to lockers that it shouldn't access, or it can happen if the program is using invalid indexes into arrays. So, if you experience these problems, the first thing to do is to carefully check code that uses pointers or arrays and make sure it is not accessing memory locations that are not valid and correct for your program.

Note

As mentioned above, the syntax for **pointers** in **C** is a bit clunky. In particular, you have to be careful with the meaning of the * symbol. It is used in two very different ways in the context of pointers:

```
float  *fp;        // This tells the compiler you want to declare a pointer called 'fp'
                   // which will point to a floating point data item

*(fp)=3.1415926;   // This means 'assign to the contents of (fp) the value 3.1415926' here
                   // the '*' is used to access data in a particular locker
```

✍ **Exercise 2.6** In a memory model diagram, show what's happening in memory with the following small program that uses pointers. Indicate what the final value for **x** and **y** are.

```
#include<stdio.h>
int main()
{
    int   x,y;
    int   *p=NULL;

    x=5;
    p=&x;
    *(p)=3;
    y=*(p);
}
```

Note

One more note regarding good programming style when declaring pointers. You should use

```
int *p;
```

instead of

```
int* p;
```

both of which compile, and both of which you can find in textbooks and programming manuals. The reason for this is using **int*** (or any other data type with a * attached to it) is ambiguous. Here's why:

```
int a,b;
```

tells anyone reading the code that you are declaring two integer variables, **a** and **b**. So, a declaration that reads

```
int* p,q;
```

would reasonably be taken to mean you are declaring two integer pointers **p** and **q**. However, **this is not correct**. The preceding line will declare a single integer pointer **p**, and a regular integer variable **q**. This can easily be a source of misunderstanding. Conversely, any of the following declarations are completely unambiguous:

```
int *p, *q;
int *p, q;
int p, *q;
int p,q
```

in each case, it is perfectly clear which variable is a pointer and which is just an int. Make sure to write **code that is clean, easy to read, and contains no ambiguity about what you mean to do**.

2.5.1 Passing pointers to functions

As we mentioned before, a function can not directly use, access, or modify data that is external to itself. It can only receive data via its **input arguments**, and can only send data out to the rest of the program via its **return value**. This is the correct way for the function to work in many situations, and it is also good programming practice because it prevents the function from accidentally changing data external to itself in unexpected ways. However, we will find situations in which we need a function to directly access and/or modify data that is external to the function.

Let's update our example with the **square()** function to understand how this may work, and once we see the process in action, we will study the most common situation that requires us to allow functions to access information owned by a different part of the program. Consider the version of the **square()** function shown in the example below:

Example 2.7

```
#include<stdio.h>

void square(int *p)
{
```

```
    int x;          // This is unnecessarily long,
    x=*(p);         // but we want you to see every
    x=x*x;          // step of the process.
    *(p)=x;
}

int main()
{
    int my_int;
    my_int=7;

    square(&my_int);      // This reads 'take the address of my_int
                          // and pass it to square()'
                          // It's like telling your friend your locker
                          // number at the gym.

    printf("The final value for my_int is: %d\n",my_int);

    return 0;
}
```

In the memory model, the first thing that happens is **main()** reserves space for a local variable **my_int**, and for its **return value**. Then it sets **my_int** to 7, the result is shown in Fig. 2.20.

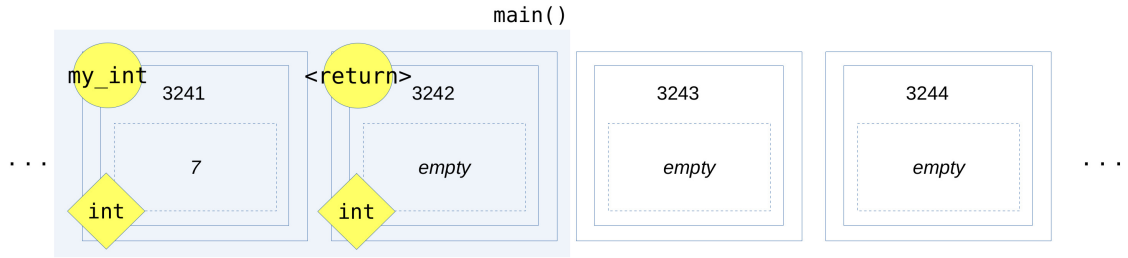

Figure 2.20: Memory model for the code in Example 2.7 after the line **my_int=7;** is processed.

The next line in the code: **square(&my_int);** is the call to the square function, it does the following:

- Reserve space for **square()**. One box for the **input argument p** which is a **pointer to int**, then a box for the local integer variable **x**. Now we notice something we haven't seen before. The function declaration states the **return value** is of type **void**. This effectively means **the function does not have a return value**. That means **there will be no box reserved for the return value of** *square()*.
- Pass the required input argument to **square()**. The function is being called with **(&my_int)** as argument. This means **get the address of** *my_int* **and pass it to** *square()*. The **input argument** *p* is a pointer, so we need to provide it with **an address, a locker number**.
- The program then continues with the code in **square()**.

The situation in the memory model will be as shown in Fig. 2.21. Pay close attention to the value stored in the box labeled **p**, it now contains **the locker number** for **main()'s** local variable **my_int**.

The next couple lines in **square()** do the actual work, let's look at them one by one:

```
x=*(p);
```

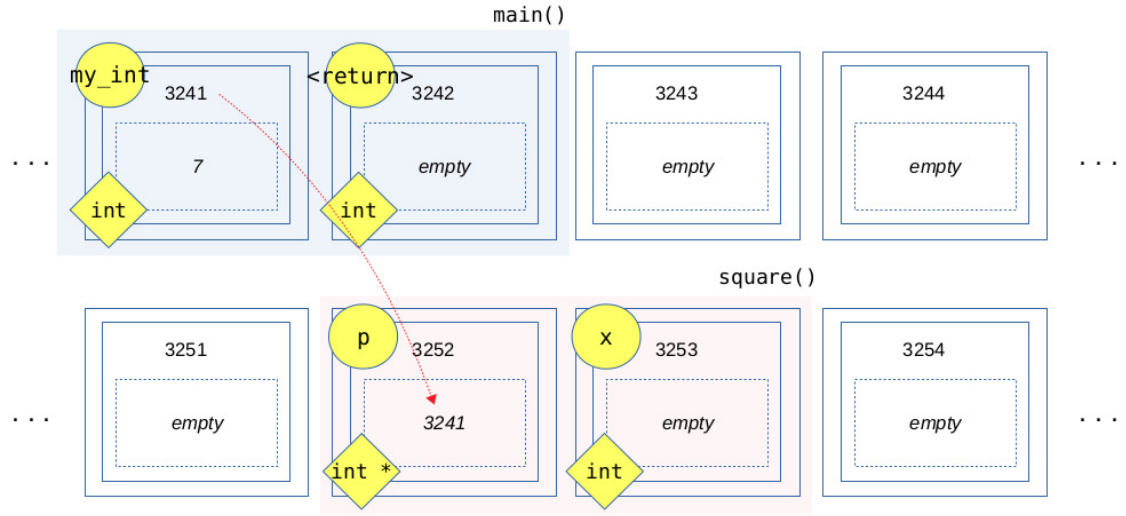

Figure 2.21: Memory model for the code in Example 2.7 once the function **square()** is called. Notice **there is no box for a return value for** *square()*.

this reads **get the contents of (p) and store that value in** *x*. Since **p** now has the locker number for **my_int**, this line effectively becomes **get the contents of locker (3241) and store that value in** *x*. This stores a **7** in variable **x**. The next line **x=x*x;** simply computes the square and makes the value of **x** equal to **49**. Finally, the last line

```
*(p)=x;
```

updates **my_int**. The line reads **make the contents of (p) equal to the value in** *x*, or, if we think of **p** as a locker number, **make the contents of box (3241) equal to the value of** *x*. The end result will be that **main()'s** local variable **my_int** will be updated to the value **49**. The situation in the memory model just before the memory reserved for **square()** is released is as shown in Fig. 2.22.

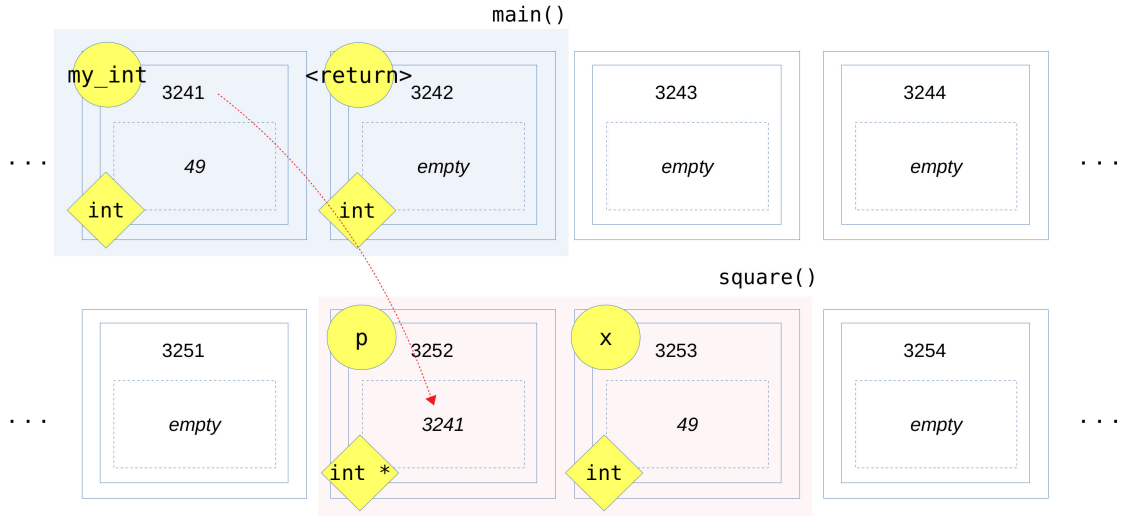

Figure 2.22: Memory model for the code in Example 2.7 after function **square()** has been processed, **right before the memory reserved for square() is released**.

Note

A small note regarding the code above. We could have written a much shorter, but also harder to understand version of **square()**. The short version doesn't use the **local variable x**. Instead, it contains a single line of code:

```
*(p)=(*(p))*(*(p));
```

Which should be read as **make the contents of (p) equal to the value of the contents of (p) times the contents of (p)**. This is correct, and shorter, but also hard to parse because the syntax is clunky. Notice the use of **parentheses** to clarify what is being multiplied. In particular, the parentheses show that **(*(p))** is one thing being multiplied with another **(*(p))** thing.

You could avoid the parentheses altogether and write:

```
*p=*p**p;
```

which will **compile and run** just fine, but is **very hard to read and understand** for anyone other than whoever wrote this **pretty ugly code**. So the message is this: Please write code using pointers **as clearly and unambiguously as possible**. The syntax is clunky, but you can use **parentheses** to help group things into **units that make sense** and can be more easily read and understood by a human going through the code.

From the example above, it's important to remember that **passing a pointer into a function** gives the function access to information that would otherwise be out of reach. This is both a powerful feature, and a possible source of all kinds of bugs. So **use this with care**. As noted earlier, most functions that perform simple computations and can return the result **should use a return value and not have access to external data**. However, handling larger amounts of information such as may be stored in arrays or strings is best done by using **pointers**. Let's now see how that works in **C**.

2.5.2 Pointers and Arrays

At the start of this section we briefly discussed that with arrays (or strings) we want to avoid having to make copies in order to provide a function with the information needed to do some work. Copying arrays would result in an often unnecessary waste of space as well as time. So, in **C**, by **design**, arrays will be passed onto the functions that use them as **pointers**.

Definition 2.4

In **C***, arrays are* **passed by reference***. This means that whenever we pass an array (or string, which is just an array of* **char***) to a function, the function receives* **a pointer to the first entry in the array***. The array* **is never copied***. The function thereafter can* **directly access/modify** *the contents of the array that was passed to it, regardless of which part of the program declared and owns the array.*

There are a couple of important ideas to keep firmly in mind regarding arrays passed on to functions:

- The function will only get **an address/locker number**. None of the values of the array are ever copied or passed on to the function.

- The function **must also receive** the **size of the array**. This is because arrays in **C** are only collections of boxes. They do not contain information about their size, this has to be provided separately so the function knows how many items of data it has to work with.
- The function will access the array normally, that is, it doesn't have to use **pointer notation** to access information in the array. This is done for the convenience of the programmer. What actually happens is that **the compiler translates all array accesses to the equivalent pointer expressions** without us needing to worry about it. This is also the case in the original function that declared and owns the array.

Let's have a look at an example of how this works:

Example 2.8

```
#include<stdio.h>

void square_array(int array[], int size)
{   // Square the value of each entry in the array
    int j;

    for (j=0; j<size; j=j+1)
    {
        array[j]=array[j]*array[j];
    }
}

int main()
{
    int i;
    int my_array[5];

    for (i=0; i<5; i=i+1)
    {
        my_array[i]=i;
    }

    square_array(my_array,5);

    for (i=0; i<5; i=i+1)
    {
        printf("%d squared equals %d\n",i,my_array[i]);
    }

    return 0;
}
```

Compiling and running the code above produces:

```
./a.out
0 squared equals 0
1 squared equals 1
2 squared equals 4
3 squared equals 9
4 squared equals 16
```

Let's see what's happening in memory when we execute this code. First, **main()** reserves space for a 5-entry **integer array** and fills it up with values from **0** to **4** (remember, array indexes for an array of length **k** go from **0** to **k-1**). This is shown in Fig. 2.23.

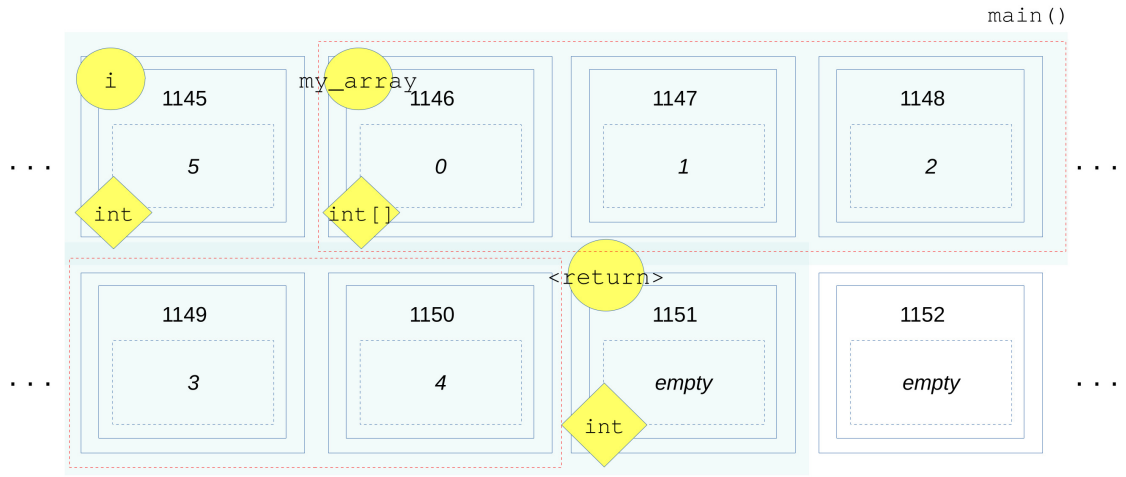

Figure 2.23: Memory model for the code in Example 2.8 after **main()** has declared and filled a small integer array.

The memory reserved for the array is marked by the red-dashed lines (as expected, it's contiguous, and only the first box is tagged with the name and type for the array). When the function **square_array()** is called, the following happens:

- The compiler reserves space for the input parameters. The first one is declared as an **int array** but notice **no size is given** because the function doesn't know what the size of the array was when it was declared, and also because we would want this function to work with arrays of any size. Because **arrays are passed by reference** the compiler will reserve **a single box** in which it will place **the address of the first entry of the array**.
- The compiler then reserves space for the integer input parameter **size**, which will provide the size of the array the function will be working on.
- Then the compiler reserves space for local variable **j**.
- Finally the program continues with the instructions in **square_array()**.

In the memory model this will have the effect shown in Fig. 2.24.

Things to note

- Function bf square_array() only reserved 3 boxes in memory, despite the fact that it will be working with a 5-entry array. The function would reserve only these 3 boxes even if the input array had millions of entries
- The box tagged **array** has a type **int[]** to indicate it is an array, however, what it actually contains **is the address (or locker number) of the original array in** *main()*

The last point above deserves comment. The box for **array** looks like an actual array would, and the code in the function **square_array** uses the **array** variable as you would use any other array. This **is done for the convenience of the programmer**. In reality what the **array** box contains is just a **pointer to the first entry in the array from** *main()*.

After the compiler has reserved space for **square_array**, the function goes about its work. The important thing to understand here is that the line

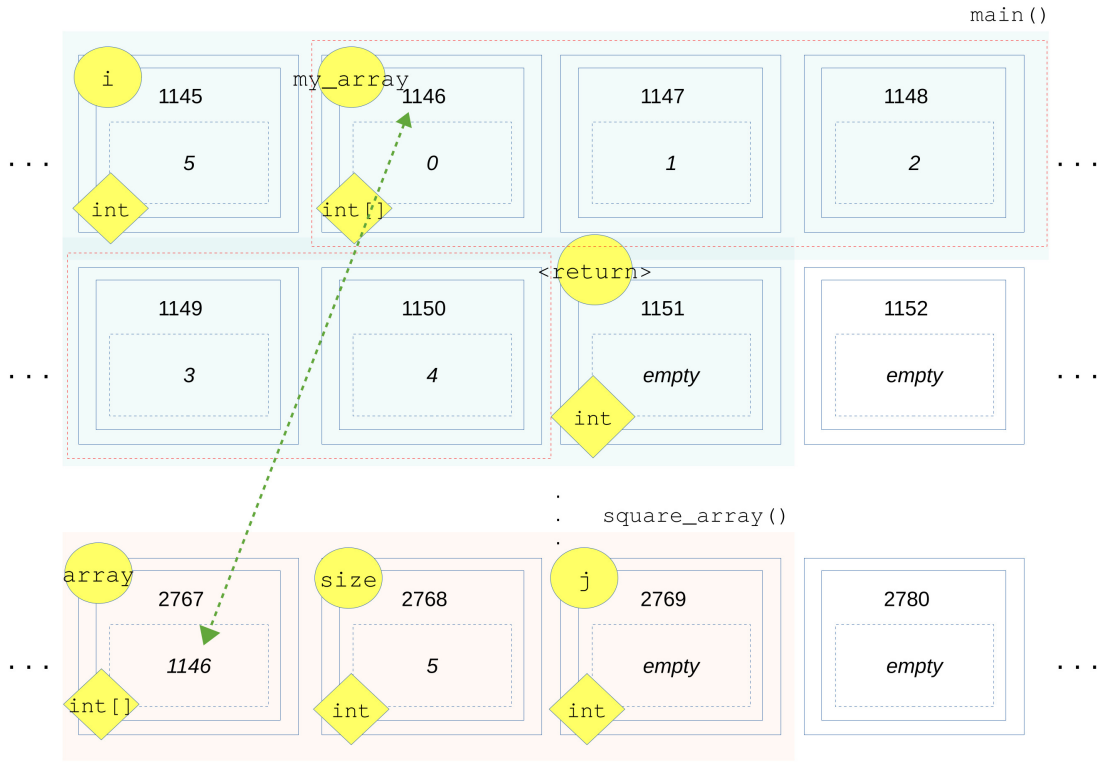

Figure 2.24: Memory model for the code in Example 2.8 after the call to **square_array()** has reserved all the space the function will need.

```
    array[j]=array[j]*array[j];
```

is **directly working on the data stored in** *my_array* **which was declared and is owned by** *main()*. Passing the array to the function is exactly the same thing as **giving the function a pointer** so it can work directly with the data in the original array.

Question: What memory location is being accessed if **square_array()** stores something in **array[3]**?

Question: What do you think will happen if **square_array()** attempts to store a value in **array[11]**?

After the function has completed its work, we expect the contents of memory to look as shown in Fig. 2.25.

The example above shows how useful pointers can be - without them working with arrays would be much more involved. The same is true for all work we will do using strings. Indeed, if you look at the definitions for functions in the string library, they all take as input parameters pointers to char arrays.

2.5.3 Using pointers and offsets to access array data

As noted above, when we pass an array into a function what the compiler does is provide a **pointer to the first entry in the array** so that the function can directly access array data. The **C** compiler then translates all the array instructions in the function code into the corresponding expressions using **pointers**. We can choose to write code

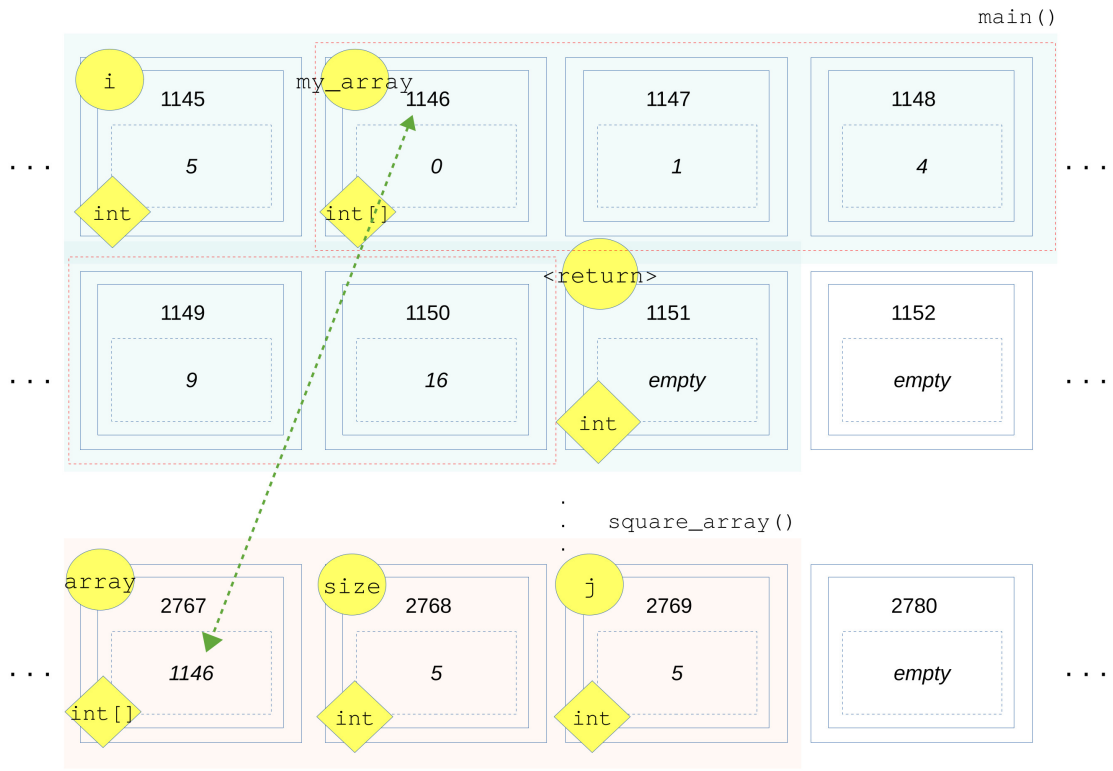

Figure 2.25: Memory model for the code in Example 2.8 after the call to **square_array()** has completed but **just before the memory reserved for** *square_array()* **is released**.

that uses **only pointers** instead of **array expressions**, and this is the more common way to work with arrays and strings in **C**. Let's see how that works, first by using pointers with a small example array, and then by rewriting the **square_array()** function in Example 2.8 to use pointer expressions instead of arrays.

Example 2.9 The program below uses pointers and offsets to change entries in an array without using array notation

```
#include<stdio.h>

int main()
{
    int  array[10];
    int  *p=NULL;

    p=&array[0];     // Get address of the first entry in the array

    *(p+0)=5;        // Set the contents of array[0] to 5
    *(p+4)=10;       // Set the contents of array[4] to 10fs

    printf("array[0] is %d, array[4] is %d\n",array[0],array[4]);

    return 0;
}
```

The example above initializes a pointer to the address of the first entry in **array** (the line **p=&array[0];** is read as **take the address of** *array[0]* **and store it in** *p*. Given a pointer into an array, we can **use an offset** to access any of the entries in the array. Offsets are exactly equivalent to **array indexes**, so **(p+4)** is referring to the data item

stored four boxes **after the first entry in the array**. The expression

```
*(p+4)=10;
```

is read as **make the contents of** *(p+4)* **equal to the value** *10*. Since **p** contains a locker number, **(p+4)** gives the location of a box four locations after the first entry in the array.

The instruction

```
*(p+7)=2;
```

is exactly equivalent to

```
array[7]=2;
```

in fact, as we have discussed above in the context of passing arrays to functions, the compiler will convert the expression that uses array notation to the corresponding expression that uses pointers when translating your program to **machine language**.

It's important to develop the ability to use pointers and offsets to access information, and we will have plenty of opportunity to practice this skill through the book. For now, please keep in mind the following important points that apply to the use of pointers and offsets to work with data in arrays:

- Similarly to array indexes, the compiler will not tell you if a **(pointer+offset)** expression results in your program trying to access a box it doesn't own, or a box that is not part of the array you meant to work with. You can easily get in trouble by using the wrong offset together with a pointer, **so be very careful** that your offsets and pointers are always correct and the resulting access is valid.
- Problems with accessing data via pointers and offsets can easily result in **unpredictable behaviour**, or your program may be **terminated by the computer** (e.g. you get a **segmentation fault**, or the program simply stops and nothing else happens, not even an error message).
- Negative offsets can be used, though they do not make sense if your pointer contains the address of the first entry in an array. We will come across situations where negative offsets have a role to play in our program, just be careful in the meantime.

Let's now do one more example and see how we would rewrite the **square_array()** function from Example 2.8 so that it uses pointers and offsets, instead of array notation.

Example 2.10

```
#include<stdio.h>

void square_array(int *array, int size)
{   // Square the value of each entry in the array
    int j;

    for (j=0; j<size; j=j+1)
    {
        *(array+j)=(*(array+j)) * (*(array+j));
    }
}

int main()
{
```

```
        int my_array[5];
        int i;

        for (i=0; i<5; i=i+1)
        {
            my_array[i]=i;
        }

        square_array(&my_array[0],5);

        for (i=0; i<5; i=i+1)
        {
            printf("%d squared equals %d\n",i,my_array[i]);
        }

        return 0;
    }
```

The code in Exercise 2.10 is very similar to the code in Exercise 2.8, but the differences are important and worth spending a moment on Firstly, the function declaration:

```
void square_array(int array[], int size)    // Using array notation

// becomes

void square_array(int *array, int size)     // Using pointers
```

The change is mostly a matter of syntax. We know that arrays are passed by reference, and that passing an array (first version of the function declaration) results in a single box being reserved, and in that box the compiler puts the address of the first entry of the array. The second version of the function declaration makes this explicit: The function expects to receive the address of an integer data item. Both functions will receive exactly the same thing, but the pointer version of the function makes explicit the fact that what the function is getting is just a locker number.

The second change is within the function itself:

```
array[j]=array[j] * array[j];           // Using array notation

// becomes

*(array+j)=(*(array+j)) * (*(array+j)); // Using pointer notation
```

which is exactly the same thing - the first version of the code (using array notation) will be converted by the compiler to an expression that looks like the second version, using pointers and offsets. The only difference is what the programmer would read, the array notation is perhaps easier to understand, but both do exactly the same thing.

Finally, there is a small change in the way the function is called:

```
square_array(my_array,5);        // Passing an array

// becomes

square_array(&my_array[0],5);    // Passing the address of
                                 // The first entry in the array
```

The first version (with array notation) takes advantage of the fact that the compiler knows to pass arrays **by reference**, so when we call a function using the array name, it provides the function with the address of the first entry in the array. The second version (with pointers) explicitly passed **the address of my_array[0]** - which is in effect the same thing but we are making explicit the fact that we are passing down a locker number. Because the compiler knows to pass arrays by reference, we could call the pointer version of **square_array()** exactly as we would call the version that uses arrays: **square_array(my_array,5);** works with both versions.

The notation can be a bit confusing, so whenever you're working with an array, remember the following:

```
my_array;      // The name of the array, same as the address of the first entry in the array
&my_array[0]; // 'The address of' my_array[0], also the first entry in the array
my_array[0];   // The value stored in the first entry of the array, *this is NOT an address!*
```

2.5.4 Summary

What are pointers? They are variables that hold the memory address (locker number!) of the contents of a variable, array, string, or data structure we want to share between functions in the program.

Why are they useful? Because they allow functions to access and modify data that has been declared outside of the function's code. Because they allow us to share information between functions without making extra copies of the data we are sharing.

What is the syntax for declaring a pointer variable?

```
type   *pointer_name=NULL;
```

for example,

```
int  *p=NULL;        // A pointer to an int
float *fp=NULL;      // A pointer to a float
```

What is the syntax for assigning a pointer to a variable?

```
pointer = &variable;
```

Which is read as **assign to** *pointer* **the address of** *variable*, for example

```
float pi;
float *fp=NULL;

pi=3.1415926535;
fp=&pi;
```

initializes a float called **pi**, and sets the pointer **fp** to the address of **pi**.

How do we access or update boxes in memory using pointers?

```
*(pointer+offset)=expression;  or  variable=*(pointer+offset);
```

the expression on the left is used to store a value at the locker specified by **(pointer+offset)**, the expression on the right is used to get the value stored at locker **(pointer+offset)** so we can assign it to a variable. Examples:

```
float pi;
float *fp=NULL;

fp=&pi;
*(fp)=3.1415926535;
```

The above uses the pointer to update the value of **pi**.

How do we obtain a pointer to an array? Get the address of the first entry in the array. Like so:

```
int array[10];
int *p=NULL;
p=&array[0];
```

or, equivalently

```
int array[10];
int *p=NULL;
p=array;
```

the above works because the compiler takes the array name to be equivalent to a pointer to the first entry in the array.

How do we update entries in an array using a pointer? Since the pointer has the location of the first entry in the array, we can use the pointer plus an offset to access and/or update any entry in the array like this:

```
int array[10];
int *p=NULL;

p=&array[0];

// Set the first entry in the array to 7
*(p)=7;

// Set the second entry in the array to 10
*(p+1)=10;

// Set the third entry in the array to 15
*(p+2)=15

// Set the last entry in the array to -15
*(p+9)=-15;

// The next two lines are an example of invalid
// (pointer+offset) combinations - they are a
// bug!
*(p+15)=25;
*(p-2)=0;
```

as noted earlier, we have to be careful that our **(pointer+offset)** expressions do not result in access to a box that is not valid. The last two lines would clearly attempt to store data in lockers not reserved for our program, so any of those lines would be expected to cause a problem.

Final notes on pointers:

We will be using pointers for the better part of the course. You should make yourself comfortable with them, practice using them for passing parameters to functions, for having functions manipulate and change data defined

outside of them, and to access and change the contents of arrays, both by passing them to functions that manipulate them, and by using pointers and offsets to access data in the arrays.

You should always make sure that

- Pointers are initialized to **NULL** when you declare them - this will save you no end of trouble as you can check whether the pointer has actually been assigned to something or not.
- Every function that receives a pointer as a parameter must check that the input pointer is not **NULL**. You can not access a **NULL** pointer and trying to do so will crash your program.
- When using pointers and offsets to access data, you must ensure the offsets used are within bounds for the array you're indexing into.

✍ **Exercise 2.7** Let's put everything you've learned in this section together. Write a program that:

- Declares a string in **main()**, you can fill that string with any text you wish, for example: **"Now I know how to program in C!"**.
- Write a function that **takes the string as input**, and reverses the string. Mind the fact that the length of the string may be different than the size of the array, and that the reversing process should not move the **end-of-string delimiter**.
- Try to implement the reverse function using **(pointer+offset) notation** only.
- Have your code print the reversed string.
- Write a function that takes takes as input **the string**, a **query character**, and a **target character**, then goes over the string and replaces any occurrence of the query character with the target character.
- Try to implement the replace function using **(pointer+offset) notation** only.
- Have your code print the string after replacing all occurrences of **a** with **i**.

For instance,

```
If the input string is "Hawdy! that strange pointer was NULL"

then the reverse function would print:

LLUN saw retniop egnarts taht !ydwaH

and after replacing 'a' with 'i' the program would print

LLUN siw retniop egnirts tiht !tdwiH
```

That's it! Have a go at it. You may want to read the last part of this Chapter on building code that works, and apply the advice there to solving this exercise.

And finally:

You've worked very hard in this Chapter to become familiar with C and how it works, so you deserve a moment to celebrate.

Go have a nice lunch, watch a movie, play sports, spend time with your friends, or something equally fun.

Congratulations! Now you know how to program in C!

Figure 2.26: Climbing a mountain is hard work, but the view from the top is worth it! *Photo: WolfmanSF, Wikipedia Commons, CC-SA 3.0.*

2.6 Model versus Reality

Through this Chapter, we developed a memory model to help us understand how information is being manipulated by our program, how variables, arrays, function arguments, return values, and pointers work. But,you should keep very clearly in your mind the following idea: The **memory model** we are studying here is exactly that - **just a model**. It is **an abstraction** intended to help you understand how information is organized and manipulated by a **C** program as it goes about its work. It is a good model in that it captures correctly the way in which memory is organized inside your computer, but by necessity, **it simplifies or removes some of the details** because they are neither relevant at this point, nor helpful in understanding how a C program works. However, it's important to make sure we understand what parts of the model are entirely accurate, and which are simplified.

Our **memory model** accurately describes:

- That memory is organized as a large collection of **sequentially numbered boxes** that can be used to contain data.
- That **each variable** our program declares **is assigned its own box**, its box can not be shared by any other variables, and it's tagged with the variable's name and data type.
- That **variables declared by the same function** in our program **will be assigned neighbouring boxes** in the memory model.
- That the **return value** of a function **will have its own reserved box**.
- That **variables and input arguments are local** to a function. Only the function that declared them knows their name, type, and can access or store values in their corresponding boxes by using the variable's name.
- That **arrays** are simply collections of **consecutive boxes** with the same **data type**.
- That **pointers are just variables** so they have **their own box** in memory, that box will store **the locker number** of another variable or array we want to work with.

- That memory reserved for a function **is released** after the function ends.

Some of the details that the **memory model** simplifies (removes or abstracts away):

- In our model neighbouring boxes have locker numbers that **increase by 1**, and **we assume boxes can hold any data type**. In reality each data type has a different size and the compiler needs to reserve the correct amount of space. This means that if we look at the locker number for two neighbouring variables **A** and **B** inside the computer, their locker numbers may be different by more than 1 unit. This is not relevant to us as it is handled automatically by the compiler.
- The order in which variables are placed in memory. As noted above, variables declared by a function will be placed in contiguous locations, but the order is up to the compiler and may be different from the order in which we declared them in our program. Assuming that the variables will be placed in the order in which they were declared is **reasonable and often accurate**, but keep in mind the compiler may choose to reorder things if it yields better code in **machine language**.
- That the **return value** is placed at the end, after all the input arguments and variables for the function. Once again this is often correct, but once again the compiler may choose to change things around if it yields better code after translating to **machine language**.

If you are interested, you can learn in detail how the parts of the memory model that were simplified actually work. This is part of what any good course in Computer Organization would cover in detail, so if you are curious, find a good book on that topic and read on. For the purpose of this book, we will work with the memory model as described above, and consistently follow the rules we described for how variables are ordered in memory.

2.7 Additional Exercises

Learning to program in a new language requires practice, so here are a few additional exercises to help you practice. Spend time working on them, and follow the advice contained in this Chapter's **How to Build Code that Works** section. The more thought you put into solving the exercise problems, and then thinking through the implementation of the corresponding program, the better you will develop your understanding of the material in this Chapter.

Note

For the exercises below where there are snippets of code and you're asked to figure out the output you should be able to do this without compiling/running the code. This is where you verify that you understood the part of the material that the exercise is about. If you can not figure out the output without compiling/running the code, that is a sign you should probably go back and review the relevant section of these notes.

✍ **Exercise 2.8** Draw a memory diagram that illustrates what happens in memory when you run the following snippet of code:

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

```
    int y;
    float pi;

    x=5;
    y=x+3;
    pi=y/2;

    printf("%f\n",pi);

    return 0;
}
```

What will be the final value printed by the program?

✍ **Exercise 2.9** Consider the following snippet of code:

```
float convert_to_degrees(float angle)
{
    // This function converts an angle given in radians to degrees
    return angle*360.0/(2.0*3.1415926535);
}

int main()
{
    int x;
    float ang;

    x=2;
    ang=convert_to_degrees(x);
    printf("%f\n",ang);

    return 0;
}
```

Draw a diagram that shows the memory model after we call **convert_to_degrees()** but **before the memory reserved for the function is released**.

What is the final value of **ang**?

✍ **Exercise 2.10** Consider the next little program:

```
void hard_working_function(int x)
{
    x=x*x;
    x=x/x;
    x=3*x;
    x=x+71;
    x=x/2;
}

int main()
{
    int x;
    x=1;
    hard_working_function(x);
    printf("%d\n",x);

    return 0;
```

```
}
```

What is the final value printed by the program?

✍ **Exercise 2.11** Write a program that

- Declares an array of 10 integer values
- Fills this array with equally spaced angles between 0 and 359 degrees
- Prints out the 10 angles both in degrees and in radians

✍ **Exercise 2.12** Remember that when we pass an array to a function, the function gets a pointer to the array (not a copy of the data), and recall that strings are arrays of chars. Now consider this program:

```
void start_a_story(char input_string[1014])
{
    char little_story[2048];

    strcpy(little_story,"Once upon a time...");
    strcat(little_story,input_string);
    printf("%s\n",little_story);
}

void main()
{
    start_a_story("there was a blue rhinoceros named Randolph.");
}
```

What is the output of this little program?

✍ **Exercise 2.13** Now consider the modified program from Exercise 2.12 shown below:

```
char *start_a_story(char input_string[1014])
{

    char little_story[2048];
    strcpy(little_story,"Once upon a time...");
    strcat(little_story,input_string);

    return little_story;
}

void main()
{
    char *story;
    story=start_a_story("There was a blue rhinoceros named Randolph.");
    printf("%s\n",story);
}
```

There is a major problem with the code above. What is it? (hint: when are boxes reserved for function calls?). Explain what the issue is, and what you expect would happen if we compile and run the code above.

✍ **Exercise 2.14** Let's consider the following little programs

```
void main()
{
    char little_string[1024];
```

```
        char *p=NULL;
        strcpy(little_string,"This is just a little harmless string");

        p=&little_string[0];

        printf("%c\n",*(p));
        printf("%c\n",*(p+3));

        p=p+5;
        *(p)='u';
        printf("%s\n",little_string);
    }
```

What is the output of the little program?

2.8 Building Programs That Work - Part 2

Writing programs that work, and writing good code, are habits that require work and discipline. Experience helps, so the more you practice, the better you will become at it. But it all begins by having a solid process for developing programs that solve a variety of problems. You will develop this process over the course of your career, and there are multiple disciplines that will contribute to it. For this introductory book, we will develop a skeleton on which you can later build a stronger, much more capable software development process.

Here is a short process that you should follow to solve any programming-related problems going forward.

Step 1 - Fully understand the problem you are expected to solve. In examples within the book, you will be working from a problem description of what your program should do, in more realistic contexts you may have a user with particular needs for what your software should accomplish. In every case, you **must fully understand the problem** as well as **any specified characteristics of the solution**. If the problem is being specified by a user, you will need to ask questions that help you fill-in any gaps in the original problem description.

Example 2.11 Description: You are asked to implement a program that will find all prime numbers between **2** and **N**, where **N** can be up to **10000**. The program should implement the method called **the sieve of Eratosthenes** to find these primes, and should print them out as a list separated by commas.

Understanding the problem: The description above is fairly short, and may not provide all of the information you need to solve this problem, but it does provide a good starting point. Reading the description carefully we note that:

- Our program will be tasked with finding **prime numbers** (what is a prime number?)
- These numbers have to be within a range of **[2,N]** where **N** can change but is at most **10000**
- To find the primes, we must use a method called **sieve of Eratosthenes** (what is that?)
- The primes that were found will be printed as a comma-separated list

The description of the problem was enough to give us the big picture, but now we need to fill-in the details. First, if we are not sure what a prime number is the first step is to learn about them and make sure that we understand

how to determine if a number is prime. We can't find prime numbers if we don't know how to check if a number is prime.

Secondly, if you, like me, have never heard of the **sieve of Eratosthenes**, we would have to go and find out what it is and how it works. This involves a bit of work, you can look this up online, or you can check it out in a book. The sieve works by **scanning the numbers that are in the specified range** and **progressively tagging** all numbers in this range that are multiples of the primes found so far. So, for instance, the first number scanned is **2** which is prime. The sieve then tags all multiples of **2** as not prime. The next number scanned is **3**, which was not tagged before so **it must be prime**, the sieve then tags all multiples of **3** in the range as not prime. Next up is **4** but it was tagged before because it's a multiple of **2** so it is not prime. Next is **5** which was not tagged previously so it must be prime and the sieve tags all multiples of 5, and so on.

By the time the sieve has scanned all numbers in the specified range, any non-tagged numbers left must be the primes.

Note that we have not said anything about programming, we haven't started implementing anything, we are just understanding what the problem is and what the task is that we need to solve. We should **not proceed** until we are certain we have **fully understood** the problem, the input our program will be required to work with, and the output it must produce, as well as any specified conditions on how the problem must be solved. Whenever you are not entirely sure about something, you should seek additional information. It is also possible you will come up with more questions as you work through step 2 of the process.

Step 2 - Think through the steps required to solve the problem *on paper*. You should **not start writing a program** until you have a well developed algorithm for what the solution should be. The solution should be detailed enough that you or anyone else implementing it can easily figure out what the different parts of the process are, how it can be broken up into functions, and how these functions will work together to produce the solution. You algorithm can be written in **pseudocode** or just as a list of steps, but it has to cover every important part of the solution.

Example 2.12 For the program that finds the prime numbers, given our understanding of the problem and the description of how the **sieve of Eratosthenes** works, we can come up with the sequence of steps that our program needs to carry out to solve the problem:

Requires: The value of **N Algorithm:**

- Declare an array with **N** integer entries. This array will be used to indicate if a number is prime or not. The value **1** will indicate a prime, **0** will indicate a non-prime
- Initialize all entries in the array to **1**. The process will then tag non-primes by setting their array entry to **0** as they are identified
- Loop over each value **i** from 2 to N
  - If the entry for **i** is **1**, then **i** is a **prime**. Loop through the array in increments of **i** tagging all multiples of **i** as non-primes by setting the corresponding array entry to **0**

- After the loop is complete, loop once again over the array entries from **2** to **N**, if the corresponding array entry has a value of **1** print the number

Note that the algorithm above states what information is **required** for the algorithm to work, and the description of steps is detailed enough that we could come up with a program that implements it. You should also note that **the algorithm is the solution**. Implementing the algorithm is really not the major issue when solving a problem - the key step is this one, figuring out what the algorithm is in enough detail that you can be confident that implementing that algorithm will solve the problem.

Needless to say, you can not expect a program to work if you skipped this step and simply tried to write some code without first thinking through the solution. That is a recipe for making a problem harder than it actually is: Writing a program from a complete algorithm is its own challenging task, but it is manageable. Similarly, solving a problem on paper is challenging but manageable. Trying to do both things at the same time is an easy way to turn two manageable tasks into a single unmanageable one.

Step 3 - Design the program that will implement your solution from Step 2. This means working out **how to break the algorithm into functions** that will carry out the required work. You need to think about **what data needs to be declared by each function**, and **how the different functions will communicate with each other** (input arguments and return values). You also need to **figure out the overall flow of the program**. You should be able to verify that your design has functions that take care of each of the steps in the algorithm you came up with, and you should be able to follow the program flow in your design and convince yourself it implements the algorithm correctly.

Example 2.13 For the prime finding program, and considering the algorithm proposed in Step 2, a reasonable design could look like so:

```
main()
    - Declares and owns the array of size N that will contain the 1s and 0s indicating
       which numbers are prime
    - Initializes this array to all 1s
    - Calls a function that implements the sieve of Erastosthenes on the array
    - Prints out the prime numbers after the sieve has completed its work

sieve()
    - Receives a pointer to the array with initially all 1s
    - Loops over entries starting from 2
       For each prime 'y' found, loop over the array tagging all multiples
       of 'y' as non-prime
```

For most problems and algorithms, there will be many possible designs, and there may be multiple reasonable ways to organize the program into functions. Later on in the course we will look at ways to decide whether one possible solution is better (in an objective way) than a competing one. For now, what matters is that you should come out of this step with a solid plan for your program, one which can be directly implemented.

Step 4 - Implement your solution. This is where you actually get to begin writing code. Notice that **by this point you have already solved the problem**, all that remains is to write code that carries out the algorithm, and implements the design that you already have. All your previous work will make implementing the actual program

much easier, and will help you later on with testing and with (if necessary) debugging. Once you start writing the code you may find the need to adjust or change some of the details in your design, or your algorithm. However, you should not find that major changes are needed - if you do, that would mean your original solution was not properly thought out, or your design was not carefully planned out. In either case, **if you find a major issue while implementing the program you should go back and revise your solution and program design**.

Importantly: Hacking at code until it works is not a good process, it is not a good problems solving technique, and will not work for complex problems and reasonably long programs. So **avoid doing this**. Instead, make sure you have a solid algorithm to solve the problem, make sure your design is sound and reasonable, and the result of this will be code that is good **by design**.

Example 2.14 Given the design above, an implementation of the program to find and print prime numbers could look as shown below:

```
#include<stdio.h>
#include<stdlib.h>

#define N 1000

void sieve(int *p, int size)
{
    /* Implements the sieve of Eratosthenes on an array
    of the given size, it receives a pointer to the
    first entry in the array */

    int j,k;

    for (j=2; j<=N; j++)
    {
        if (*(p+j)==1)
        {
            for (k=j+j; k<=N; k=k+j)
            {
                *(p+k)=0;
            }
        }
    }

}

int main()
{
    int primeTags[N+1];
    int i;

    for (i=0; i<=N; i++)
    {
           primeTags[i]=1;
    }

    sieve(&primeTags[0],N);

    printf("The list of primes from 2 to %d is:\n",N);
    for (i=2; i<=N; i++)
           if (primeTags[i]==1)
```

```
                printf("%d, ",i);
        printf("\n");

        return 0;
    }
```

As per the design, there are two functions: **main()**, and **sieve()**; **main()** declares and owns the integer array used to tag non-primes, and **sieve()** does the work of looping through the array tagging multiples of all primes found. The code follows directly from our design from Step 3, and did not require a large amount of work to write since the problem had already been solved, the algorithm had already been worked out, and the design had already determined how many functions there would be and what these functions would do.

Step 5 - Test your program. Of course, once you have written any program, you must ensure it works properly and the only way to do this is by **thorough testing**. Testing is a skill that requires time, practice, technique, and hard thought. **Testing is a topic that requires significant attention on its own so we will discuss it in detail in the next Chapter**. For now, we will simply **compile our code**, **deal with all compiler errors and warnings** as we learned in Chapter 1, and verify that **the program produces the correct list of primes** for a given value of N.

Compiling and running the code for **N=1000** as in the program above prints out:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
     101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
    193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
    293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401,
    409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509,
    521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631,
    641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751,
    757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877,
    881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
```

Should we be satisfied that the program didn't crash and printed out a bunch of numbers? **of course not!**. We have to check that **the program has correctly solved the problem**. Notice that this is often **very different from checking that the output is correct**.

A program can produce seemingly correct output, while having done the wrong thing internally. So whenever possible, we want to verify that the information computed or produced by the program is correct **internally**. Another way to think about this is: **we test for correctness, not output**.

In the case of this program, the task was to find the primes in the range **[2,N]**. From our program design we know the result of performing the sieve of Eratosthenes will be an integer array in which **primes will be indicated by a 1**, and **non-primes will be indicated by a 0**. Correctness of the results in this array requires us to check for two things:

- That all the numbers for which the corresponding entry is **1** are prime
- That all the numbers for which the corresponding entry is **0** are non-prime

We obviously do not want to do this by hand. Instead, we can write a test function, let's call it **check_result()**, that goes into the array after **sieve()** has done its work, and checks that both of the conditions noted above are true for the entire array.

Example 2.15 A sample function to check the correctness of the results produced by **sieve()** is shown below:

```
int check_result(int *p, int size)
{
    /* This function gets a pointer to the array with the results
    from sieve() and checks each entry starting from index 2.
    If the entry is a '1', it checks that the corresponding
    index is a prime number. If it's a '0', it checks that the
    number is not prime.

    If all entries in the array are correctly marked as prime
    or non-prime, it returns 1 (indicating everything is correct)
    else it returns a 0 (and also prints the index at which an
    erroneous value was found)
    */

    int i,k;
    int tst;

    for (i=2;i<=N;i++)          // For each entry in the array
    {
        tst=1;                  // Assume it is prime
        for (k=2;k<i;k++)       // Try to divide it exactly by ingeters
        {                       // from 2 to i-1
            if (i%k==0) tst=0;  // if we can do it, it's not prime
        }
        // Now we know if i is prime, check if it agrees with the array
        if (*(p+i)!=tst)
        {
            // Failed! something is inconsistent
            printf("Found a problem, integer %d is tagged with %d, prime test returns %d\n",
                i,*(p+i),tst);
            return 0;
        }
    }
    return 1;     // Everything looks good!
}
```

we also need to add a couple lines to **main()** to call this function and print a message depending on whether results check out or not (these lines would be added anywhere **after** the **sieve()** function has been called:

```
     // Call the test function to verify the result
    if (check_result(&primeTags[0],N))
    {
           printf("The results check out, everything appears correct\n");
    }
    else
    {
           printf("There was a problem, at least one entry was tagged erroneously by the
               sieve\n");
    }
```

compiling and running the code now gives the following output:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
    89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
     271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
     467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577,
    587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
     683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
    809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
     919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
The results check out, everything appears correct
```

so the test function has verified the work done by the **sieve()**, and no inconsistencies have been found. We can check that the test function will catch problems by **manually creating an inconsistency**. For example, we can add a couple lines of code that change some of the tags in the array to the wrong value - these lines are added **after** the call to **sieve()**:

```
primeTags[4]=1;   // Here we are saying that 4 is prime
primeTags[5]=0;   // and also that 5 is not prime
```

compiling and running the code now prints out:

```
./a.out
The list of primes from 2 to 1000 is:
2, 3, 4, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
    89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
    179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
     271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,
    373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
     467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577,
    587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677,
     683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
    809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
     919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997,
Found a problem, integer 4 is tagged with 1, prime test returns 0
There was a problem, at least one entry was tagged erroneously by the sieve
```

the test function correctly identified the first entry that was wrong in our array. If you look carefully, you can see that **4** was printed in the list of primes, and **5** was not. That may be hard to catch simply by inspecting the output, so please remember this: **write code to test your functions and program for correctness**, and **do not rely just on printed output** to decide your program is correct.

A final comment on the above: The example test shown above is a good start, but by no means does it constitute a full and comprehensive test for correctness for the program. We still don't know that the program works correctly **for different values of N**, or that **the output is actually printed correctly given the contents of the array**, or **that the code is actually implementing the sieve of Eratosthenes as specified in the problem description**. Testing is a thought and work intensive process, you should practice it as often and as thoroughly as you can, until you have built the ability to develop a sufficiently solid testing process for your programs that you can be highly confident that anything you ever make available for use is as likely to be completely correct as one could hope for. We will discuss testing at length in the next Chapter.