Learning Outcomes

This unit introduces the C memory model, a way of thinking about and tracing memory usage by C programs. It also covers fundamental data types and how to allocate memory for variables and arrays. By the end of this unit, you will be able to:

- Draw the C memory model and trace the effect of an executing C program on that model
- Understand how C allocates memory for variables and functions
- Trace code using functions and how C deals with parameters and return values
- Understand how variable scope works in C
- Write code that allocates, manipulates and uses arrays
- Use pointers to access and reference data

1 The Memory Model

Computer memory is very much like a large room full of lockers

If you have been to the PanAm Centre, or if you've walked on the lower levels of the BV building, you will have noticed the rows of lockers set up there for storage. 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 1: Lockers at the Munich train station. Photo: Tiia Monto - Wikimedia Commons, CC-SA 3.0

All locker rooms share a couple of important properties:

- Lockers have numbers (so that users can find their stuff!)
- Lockers are reserved - the *owner* of the locker has the key or combination that opens the door so only the owner can store 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 three 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!)

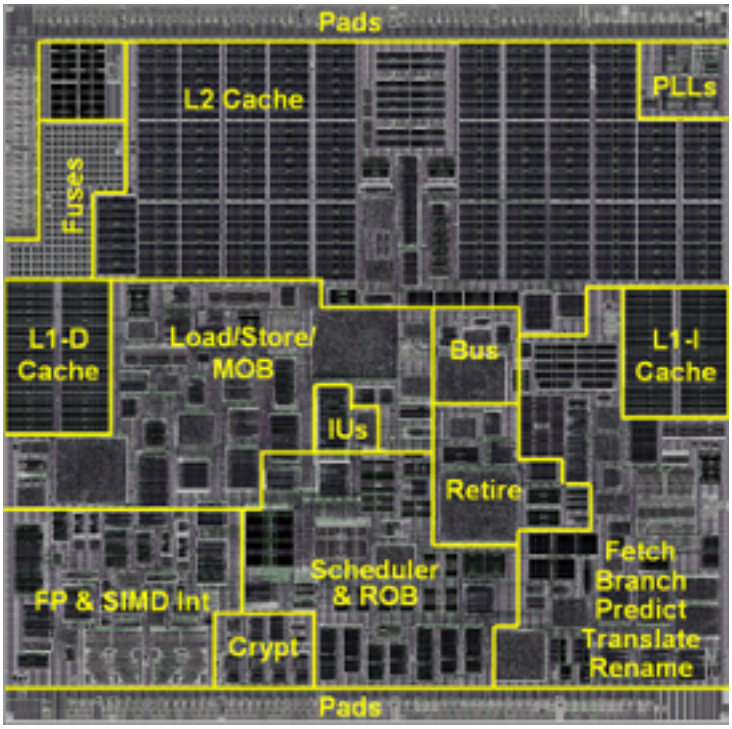


Figure 2: This is a micrograph of a CPU. Notice the rows and columns in the regions labeled L1 Cache and L2 Cache. These comprise this CPU's memory area, and they are indeed organized much like a regular locker room. You will learn all about how this works in your Computer Organization course (B58). Photo: VIA Gallery, Wikimedia Commons, CC-SA 2.0

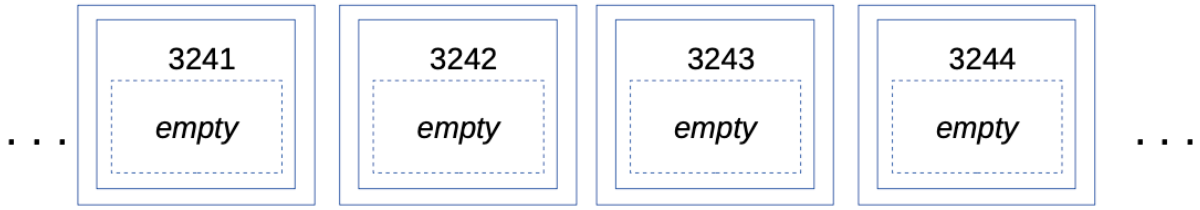
For the purpose of this course, we will think of computer memory as nothing more than a large locker room where our programs keep their data. We will see that declaring variables, assigning values to these variables, and moving information between functions is very much the same thing as reserving a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

2 What happens when we declare a variable in C?

As we saw last week, 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.

```c
#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 diagram above 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 the program (in this case, our program!).

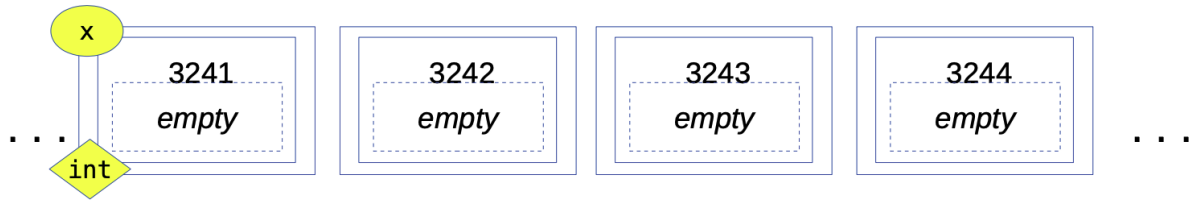
IMPORTANT NOTE

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!*. So in practice, you *should always assume a variable you just declared contains **junk** until you set its value to something.* 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 an un-initialized variable!

In C, **declaring** a variable:

```
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**.



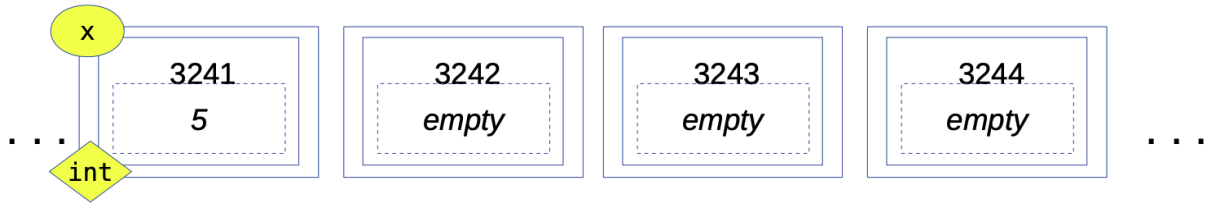
Note that:

- 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

That locker, #3241, is now reserved for our program's use to contain whatever values the variable x will take on during the program's execution. The next line contains an assignment:

```
x=5;
```

it literally means *go to the locker tagged* **x**, *and store a value of 5 there.*



Of course we could have a much more complex expression after **x=**, the expression would be evaluated and the result stored in the locker for **x**.

Suppose we changed the program a little bit to declare and assign values to a few extra variables:

```c
#include <stdio.h>
int main()
{
    int x;
    float pi;
    char c;

    x=5;
    pi=3.141592;
    c='C';
}
```

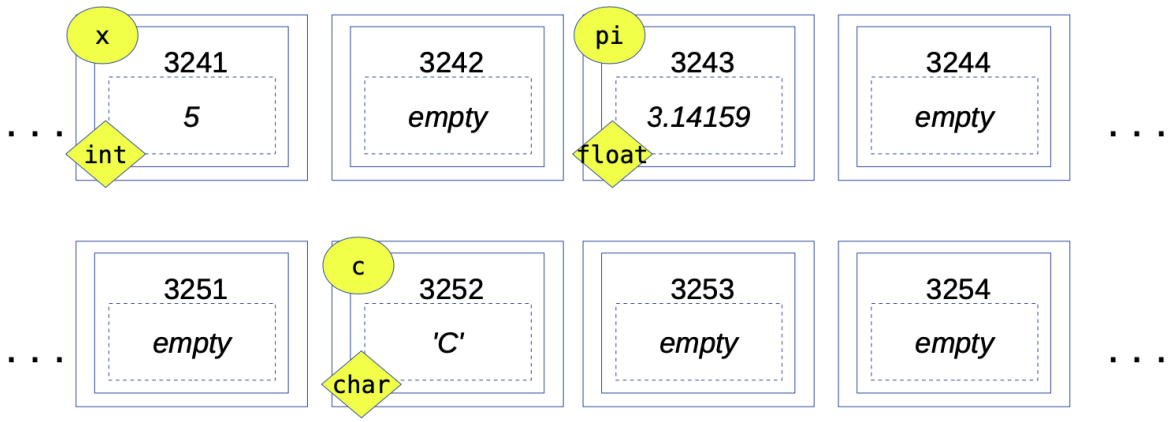
We would expect the following to happen when run:

- A box of suitable size is reserved to store an integer value, and tagged with **x**
- A box of suitable size is reserved to store a floating point value, and tagged **pi**
- A box of suitable size is reserved to store a character, and tagged **c**

Then

- We store a value of 5 in the box tagged **x**
- We store a value of 3.141592 in the box tagged **pi**
- We store the character 'C' in the box tagged **c**

So our locker room may now look like this:



Things to note:

- Lockers are assigned based on where there is a box of the right size that is empty (*not reserved for other variables*).

- Different data types use different amounts of space (we don't need to worry about this now)
- *Remember*: The particular locker numbers in the example are *not important*. What matters is each variable gets a box of the right size, and we know its number. It is also tagged with the variable's name and data type.

EXERCISE

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

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

    x=5;
    y=x;
}
```

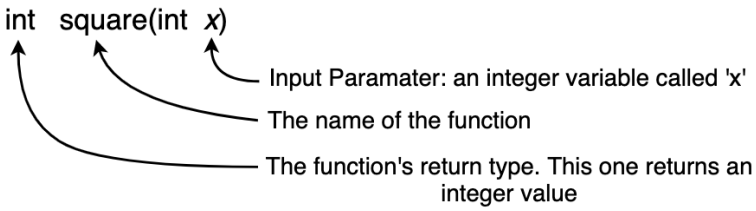
3 Functions in C - How information moves about

In your first computer science course, A08, you learned how functions work, and how to break your program into a number of functions that work together to get things done. Designing and using functions is the same in C, but the way information moves about is a bit different.

Let's see what happens in C when we declare, and then use a simple function to compute the square of an input integer value.

```
1 #include <stdio.h>
2
3 int square(int x)
4 {
5     int s;
6     s=x*x;
7     return s;
8 }
```

First let's study the function declaration:



The first part is the function's *return value type*. It specifies the data type of the value the function returns after it's done its work. A C function has a *fixed* return type, once defined it must *always return a result of that type*.

The second component is just the function's name.

The last part, in the parentheses, is the list of parameters and their types. Each one is in effect a *local variable* the function can use. You can have as many input parameters as the function needs, separated by commas.

Within the function, we have *local variable* declarations (in the above case, we are declaring an integer variable **s** to hold the square of our input), and the code for the function itself.

The function ends with a *return* statement that transfers the result of our function's work back to the part of the program that calls square(). Note that the data type for the variable **s** matches the *return type* for the function.

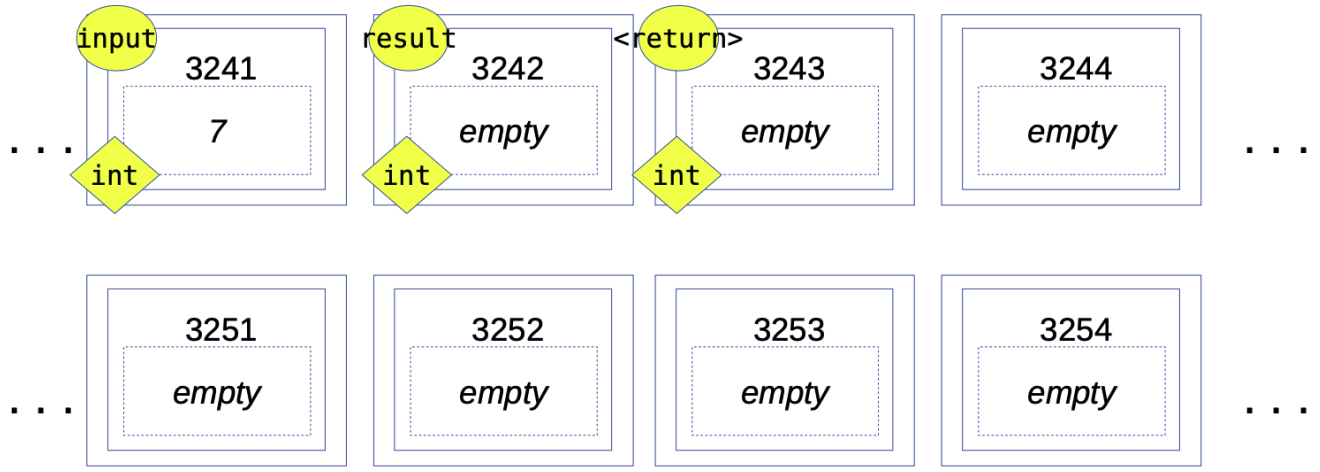
Let's see now what happens when we use this little function in a very short program:

```
1  #include <stdio.h>
2
3  int square(int x)
4  {
5      int s;
6      s=x*x;
7      return s;
8  }
9
10 int main()
11 {
12     int input;
13     int result;
14
15     input=7;
16     result=square(input);
17 }
```

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

First, main() declares two integer variables called **input** and **result**, and assigns the value of **7** to **input**. This will reserve a locker for an integer, and tag it as **input**, a locker for an int, tagged

result, and will store the value **7** in the box tagged**input**. There is also a locker reserved for the *return value* since we declared main() to be returning an *int*.



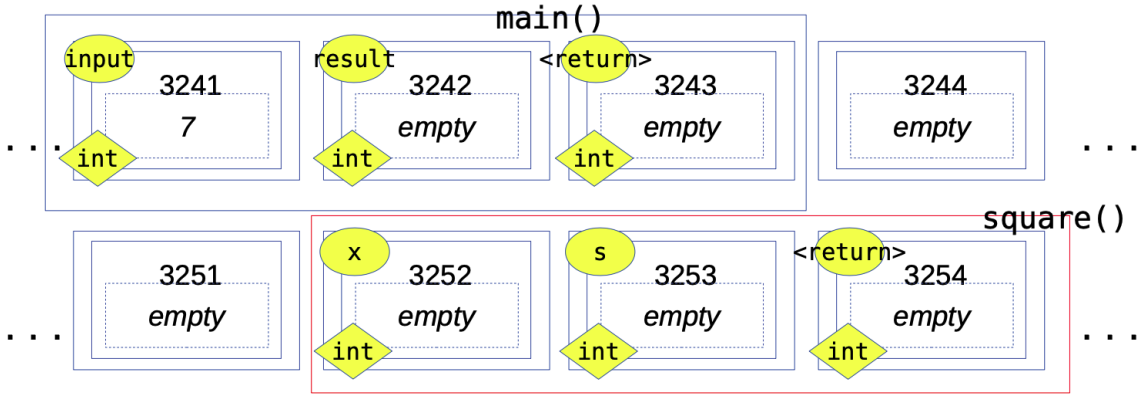
Let's see what happens when we call the square() function

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

The compiler will use our function's declarations

```
int square(int x);
```

to figure out what it needs to store in memory to keep the input parameters and the return value. In this case we need space for one integer input parameter called **x**, and an integer return type. The first line in square() declares an integer variable called **x**, so the program will reserve space for that as well.



Things to note:

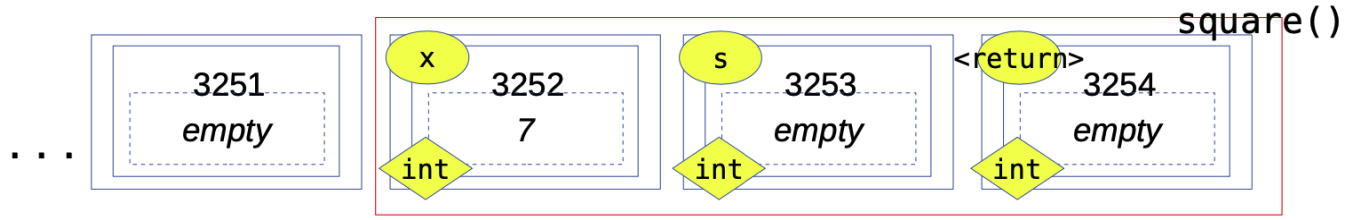
- Each of the input parameters for a function *gets its own locker!*

- Variables are *local* to functions, this is denoted by a blue box around the variables that main() is using, and a red box around the variables square() is using
- square() can't directly use or change **input** or **result**, main() can't directly access or use **x**, **s**, or the box reserved for the *return value*.

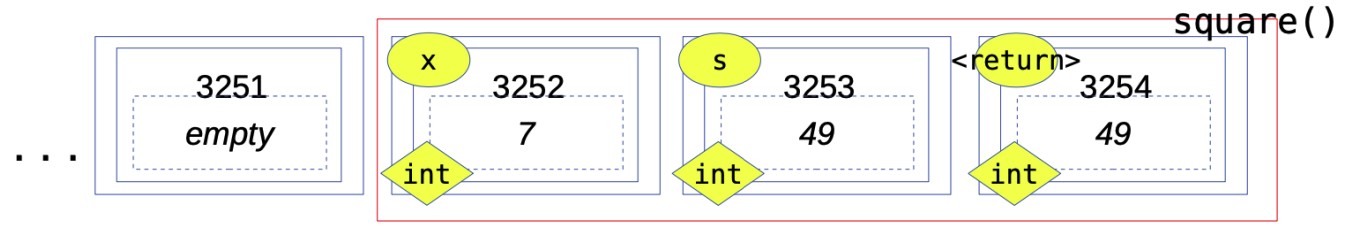
Once all the space needed for this function is reserved, the function call *passes* the input parameters to the function and the function's code is executed. In this case,

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

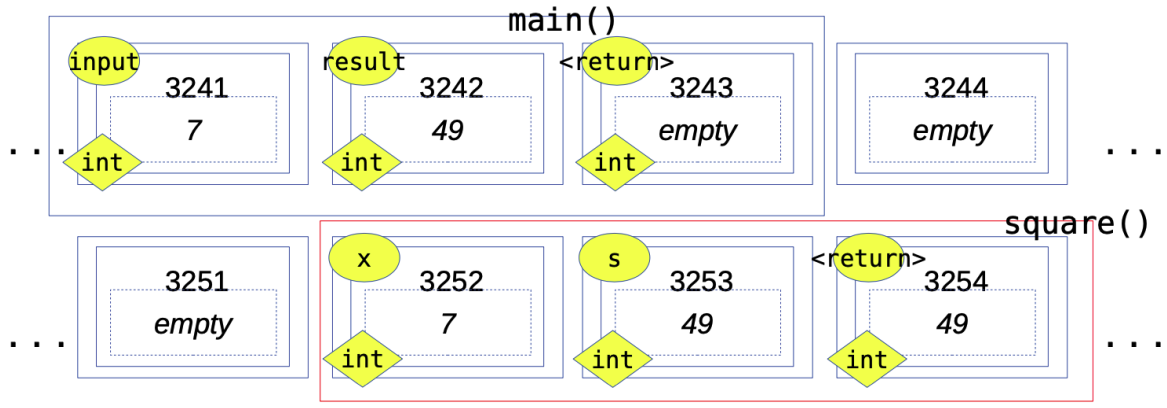
will cause the program to copy the value stored in 'input' to the box reserved for **x** and then the function's code will be executed



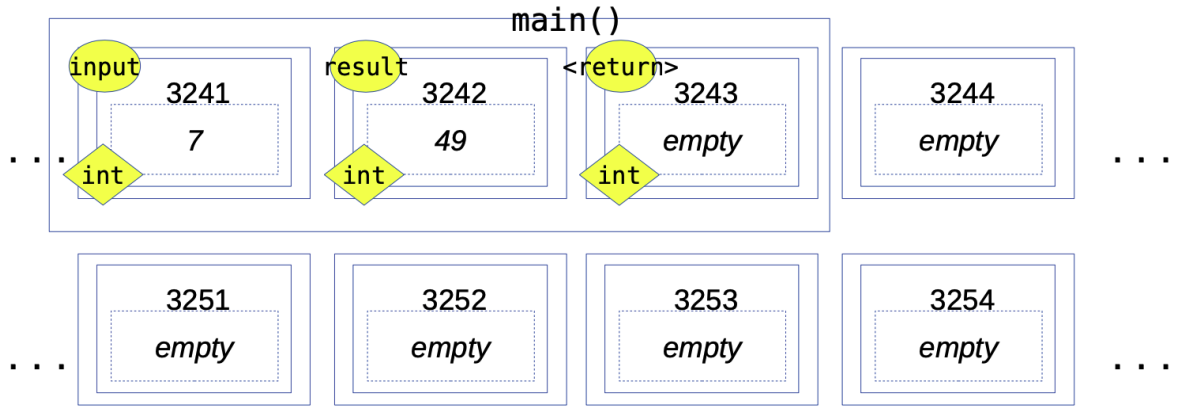
the function's code then computes the square of the value stored in **x**, and puts the result in the box reserved for **s**, and then it returns the value stored in **s** (for which it must copy the value in **s** to the space reserved for the return value).



Finally, on the same line, the return value from square() is put into the 'result' variable in main(). So the contents of memory *right at the point where the result is returned* looks like:



There's one last detail to keep in mind: Once square() is done its job, all the space reserved for its input arguments, local variables, and return type is *released* so it can be used by other parts of the program.



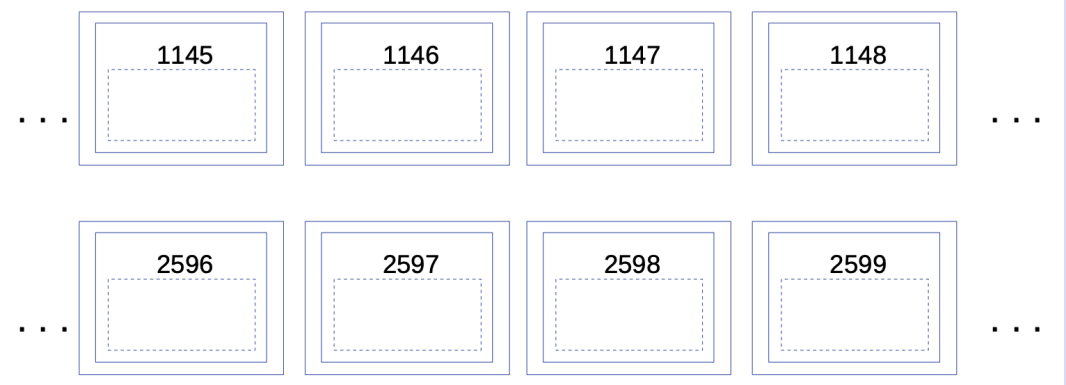
That's it for our little program. It uses a function to compute the square of an input value. Things you should take away from this example:

- Each function in your code will have its own space for variables, input parameters, and return value. These can not be directly accessed by any other part of the code.
- An important implication of this is that a function can not directly modify the value of variables that were declared outside of it.
- *Passing* arguments from one function to another involves *making a copy* of the value for the input arguments. This is called *pass-by-value*.
- Once a function is done its work, the space in memory reserved for its variables, input args., and return value is released.

EXERCISE

Show in the diagram below what the contents of memory would look like *right at the point where the result is returned* (i.e. before the space reserved for the function is released) for the following modified program – be sure to indicate which boxes are reserved for main() and which for square():

```
#include <stdio.h>
int square(int x)
{
    x=x*x;
    return x;
}
int main()
{
    int x;
    x=9;
    square(x);      // Does this change the value of x here?
}
```



Things to note in the previous exercise:

- Both main() and square() use a variable called **x**. However, if we see what's going on in memory, it should be clear that these are two separate variables, assigned to different boxes in memory. They just happen to have the same name.
- This in turn leads us to see that whatever square() does to its **x** variable, nothing happens to the one declared in main().
- Always remember: A function's parameters and variables are local and do not exist outside outside of the function.
- Similarly, a function can not directly change any variables declared outside.

4 Variable scope and type conversions

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

For variables declared within a function, and for the function's input parameters, the scope is the code contained within the curly brackets that define the start and end of the function. This type of variable is called a ***local variable***.

Additionally, we can declare variables that are outside any of the functions in the program. Usually this is done at or close to the top, before any actual code. Such variables are visible to all functions declared in the program and are called ***global variables***.

Example:

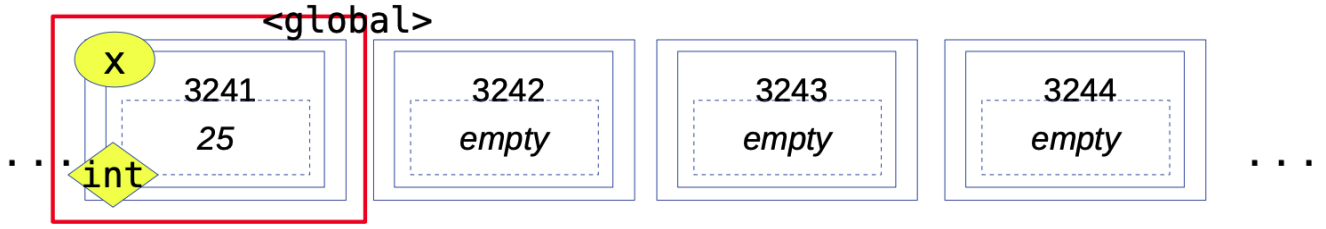
```
1  #include<stdio.h>
2
3  int x;      // This is a global variable!
4  int square()
5  {
6      x=x*x;
7  }
8
9  int main()
10 {
11     x=5;
12     square();
13     printf("The final value of x is %d\n",x);
14 }
```

Compiling and running the code above produces:

```
The final value of x is 25
```

The code above creates a global variable called **x**. When the program runs, a box will be reserved in memory for an integer value, and tagged with **x**. Except this time the variable can be used anywhere in the program, and can therefore be changed at any time, anywhere. Thus, main() first sets the value of **x** to be 5, and when we call square(), it updates that value to 25. This is completely different to what happened when **x** was local to square() and main().

Here's what memory would look like after we run the above program.



Notice that **x** is marked as *global*. It doesn't belong to a particular function.

Because of the fact that the value of global variables could change anywhere, anytime, their use is discouraged – it leads to programs that are much harder to test/debug. When implementing a program you should make every effort to rely on local variables. However, like every other general rule in programming, there are specific programming situations that may call for the use of global variables. You will learn to identify these situations with practice.

4.1 Notes on input arguments and return values – 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 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:

```
1  #include <stdio.h>
2  float square(float v)
3  {
4      // Now this function takes as input a floating point number,
5      // and produces a corresponding floating point result.
6
7      return v*v;      // In C, we can do a little bit of
8                       // processing in the return statement!
9  }
10
11 int main()
12 {
13     float pi;
14     int x,y;
15
16     int result;
17
18     x=5;
19     y=square(x);
20     printf("The result is %d\n",y);
21
22     pi=3.14159265;
23     y=square(pi);
24     printf("The result is %d\n",y);
25 }
```

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.exe
```

```
The result is 25
The 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 *type-cast* (converted!) into a floating point value 5.0, and then stored in **v**.
- The result, which is the floating point value 25.0 is returned. However, it is being assigned to **y** which is of type *int*. The floating point value 25.0 is *type-cast* to the integer value 25 and stored in **y**.
- The first use of square() produces the expected result. However, the second time we call square() we are passing in **pi** which is of type *float*. There is no casting, the value 3.14159265 is copied onto **v** and the square is computed and returned.
- However, the result, 9.869604 is being assigned to **y** which is *int*. The result is *cast* onto an *int* with value 9, which is not correct! The conversion does not *round* to the nearest integer value, it simply drops the fractional part. Hence, the floating point values 3.1, 3.45, and 3.99 will all be converted to 3.

IMPORTANT NOTE

The conversion, or *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:

```c
#include <stdio.h>

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

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

Compiling and running the above produces:

```
\a.exe
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 result 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 an 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.

```c
#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);
}
```

In the code above, the expression

```
(float)x
```

indicates an explicit type conversion – the integer value **x** must be converted to a float before it is used in the division. Similarly, **(float)y** ensures the value of **y** is converted to float before the division takes place. The result of compiling and running the code above is:

```
\a.exe
The result is:  2.500000
```

This time the code does the right thing and produces the correct result.

4.2 Why you should be very careful with type conversion – a cautionary tale

Figure 3: 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

ASIDE

So why are data types such a big deal?

All the information on 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

ASIDE (CONTINUED)

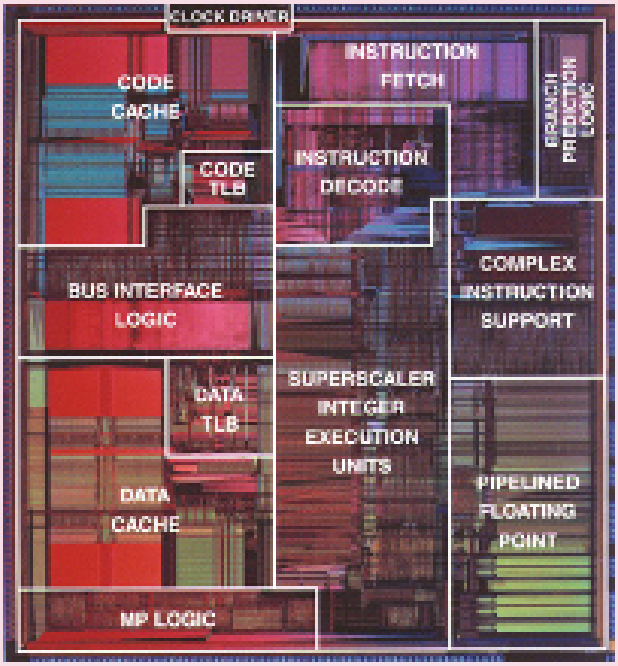


Figure 4: Micrograph of a Pentium CPU showing the separate circuits for the integer (labeled Superscalar Integer Execution Units), and floating point (labeled Pipelined Floating Point) arithmetic units. Photo link from: CPSC 3300 Computer Systems Organization, Clemson University

For those of you who want to learn more about how digital computers work (and it *is* a fascinating topic) we have a *Computer Organization* course CSC B58 that will cover everything from how to encode information using bits, to how to design and build circuits that take these bits and manipulate them to do math, to how a microprocessor works.

5 Arrays

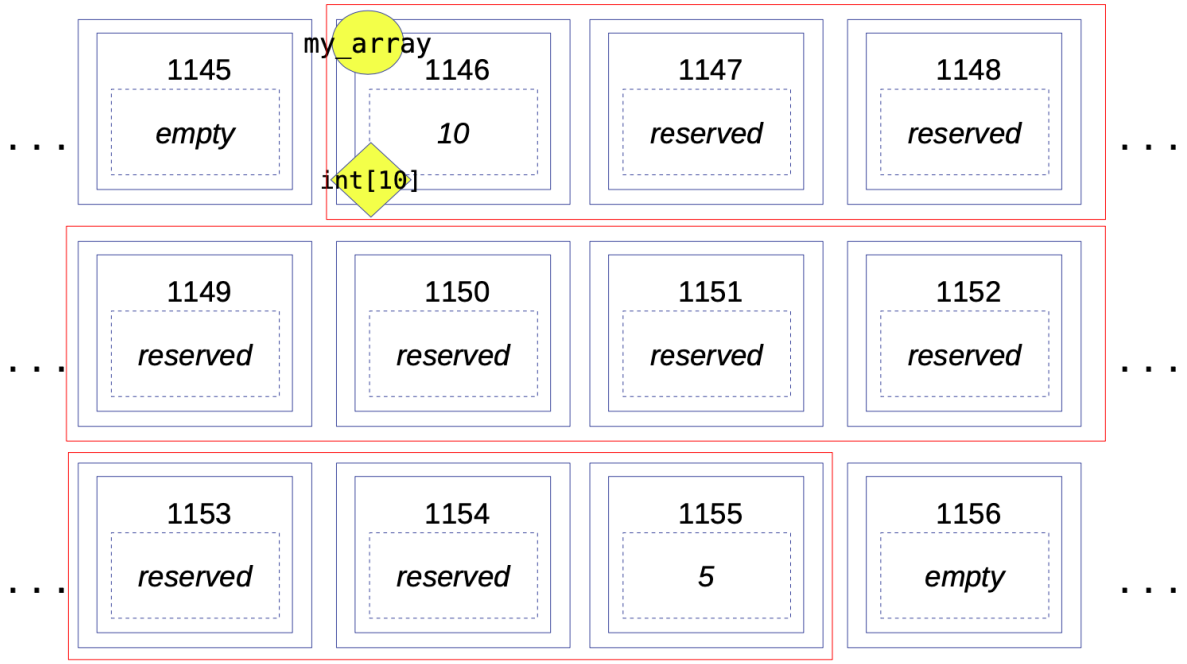
In C, we can reserve space for more than one data item of a given type. The resulting group of items is called an *array*, and it is the simplest data structure that will allow you to work with groups of data.

Here's how we declare an array in C:

```
1 #include<stdio.h>
2 int main()
3 {
4     int my_array[10];    // This is an array of 10 integer values
5     my_array[0]=10;      // This is the first entry in the array
6     my_array[9]=5;  // This is the last entry in the array
7 }
```

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, so the only operations they support are reading and storing values in the different array locations.

Here's what happens in memory when we compile and run the program above:



- First, 10 *consecutive* boxes suitable for holding integer values are *reserved*. The tag **my_array** becomes associated *with the first locker reserved for the array.*
- The reserved entries won't be assigned to any other variables, parameters, or return values in your program.
- ***However,*** as we know from before, the reserved entries contain *junk* until you have initialized them to some value.

Important fact 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.
- Therefore, you should think carefully what the size of the array should be for the task your program needs to do.
- Indexing is the same as in Python: **my_array[0]** is the first entry, **my_array[9]** is the last entry, thus, for an array with size k, valid array indices are from 0 to $k-1$.
- All entries in the array *must have the same type*. There is no mixing of data types within arrays.

- 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.***

This means that if your code does something like:

```
array[10]=5;
```

It is in effect trying to store an integer value of 5 in a location that is *one box beyond* what was reserved for your array. At that point several things could happen:

- The code runs and stores that 5 where you requested – *it's a bug, you're overwriting something else!* That box could be another local variable or array, it could be data from another function in your code. It will be hard to detect and fix this bug, and will produce unpredictable behaviour. *Such bugs are hard to find and fix because your program runs!*
- The code may crash – *obviously a bug*, in this case the box we're trying to write to belongs to another program, so the operating system blocks the write and our program crashes.
- The core may run, store the 5, and crash at a later point.

Either way, the error is present in the use of an invalid index for the array.

IMPORTANT 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!

You can declare arrays for any data type, including (as we later will see), any user-defined data types or data structures we will use to solve interesting problems. Remember that all entries in the array must be of the same type.

EXERCISE

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... Have your program print out the entries in the array in separate lines.

6 Strings

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.

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

The code above declares we want space to store a string with up to 1024 characters in it. *Remember – declaring the string gives you only the space, the initial contents of the string are junk.* 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*** (this is in contrast to Python, where strings were immutable):

```
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**. The last entry is especially important, it is the *end of string delimiter*, it is a backslash followed by a zero. This is needed because our declared string can contain up to 1024 characters, but it is possible we may want to store strings much shorter than that (like the 'Hello' above). Without the end-of-string character, C functions that use strings won't know how many of the characters in the array are actually part of the string. *Always remember to ensure your strings terminate with the* **\0** *end-of-string character.*

Now, printing the string we initialize above:

printf("%s\n",a_string);

Will produce:

Hello

(note the end-of-string character is not printed)

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() function
    strcpy(a_string, "This is a message for those learning C\n");

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

    // Let's print out what we have stored
    printf("%s\n",a_string);
}
```

Compiling and running the code will produce:

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

Question: Why do we get two lines of text from a single string?

Things to note:

- We use the *strcpy()* function (string copy) to initialize our string, in our case the source string is part of our program code, but it could be a different string variable in our code.
- We use the *strcat()* function (string conCATenation) to append one string to another. Like above, in the example the source is part of our program, but we could append the contents of two string variables.
- When we use constant character strings such as "Hello" to initialize strings, the compiler appends the `\0` character to ensure the resulting string variables are properly terminated.
- When you are using string variables (instead of constant strings) *you* must ensure they are properly terminated.

Other helpful functions in the string library:

strlen() - Gives you the length of a string
strcmp() - Compares two strings and returns 0 if they are equal.

If you want to see more string functions, you can consult the manual page for strings. To do so type:

```
man string
```

in a mac or linux terminal, or visit:

https://man7.org/linux/man-pages/man3/string.3.html

There are many other references and examples for using string library functions, so as ever, if you have a problem, Google is your friend!

IMPORTANT NOTE

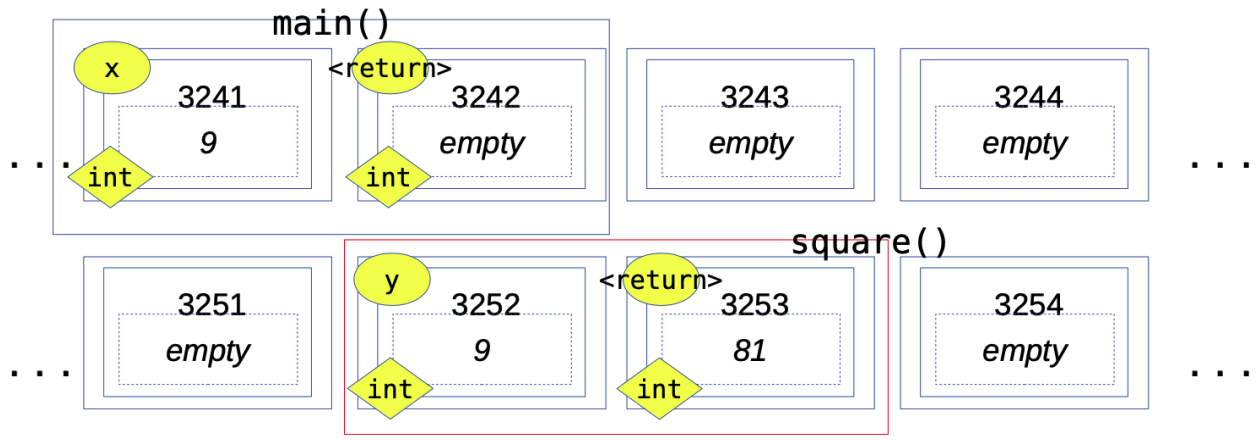
When managing strings, it is easy to run into trouble by assigning a message that is too long to a string variable that can't hold all of it. Since C does not enforce indexing bounds on arrays, the program will try to overwrite memory areas outside your declared string – a bug as described for arrays. Similarly, if a string is not properly terminated with `\0` string functions will continue to read characters well beyond the space reserved for the string – also a bug. Be careful to avoid both of these problems!

7 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:

```
1  #include <stdio.h>
2
3  int square(int y)
4  {
5      return y*y;
6  }
7
8  int main()
9  {
10     int x;
11     x=9;
12     x=square(x);
13 }
```

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.



The above shows what memory would look like just before the return statement sends '81' back to main() and **x** gets updated.

The point you need to note here is that *the only way for square() to change what is stored in* **x** *is via its return value.* This is because square() doesn't know **x**, it doesn't have access to it, can't reference it directly, or use it within its code.

In A08 you saw the difference between returning a *new value* given some input, and modifying the input itself. So far, all the code we have looked at in C returns a new value, and we don't know how to write code that can modify the input directly. There will be many situations that will require a function to access and/or change the data that was declared outside of it, let's see how we can do that using ***pointers***.

Let's start with an analogy to our locker-room model for memory:

You reserve a locker here at UTSC to keep your notebooks and textbooks between classes. You have a locker number so you know where your locker is, and you have a key to it. Only you know that lockers is yours and normally only you can access what's inside it. However, if you need something from it you could also tell your friend the locker number and the key, they can then find your locker and bring you that notebook you need for class.

In C, a ***pointer*** is nothing other than *a variable that contains a locker number* where information is stored. We can pass that to a function so that it has access to data that was declared outside of it! Let's see how we declare and use a pointer, and then we will see how to modify the square() function so that it uses pointers to change the variables we want to square directly.

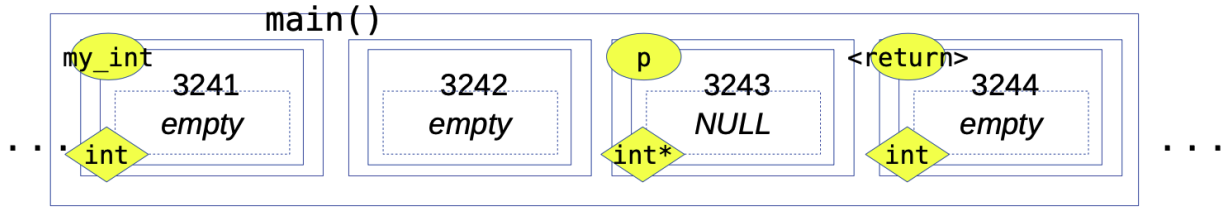
```c
#include <stdio.h>
int main()
{
    int my_int;
    int *p=NULL;    // A pointer to an int!

    my_int=10;
    printf("My int is: %d\n",my_int);

    my_int=15;
    printf("My int is: %d\n",my_int);

    p=&my_int;
    *(p)=21;
    printf("My int is: %d\n",my_int);
}
```

If you compile and run the above, you get:

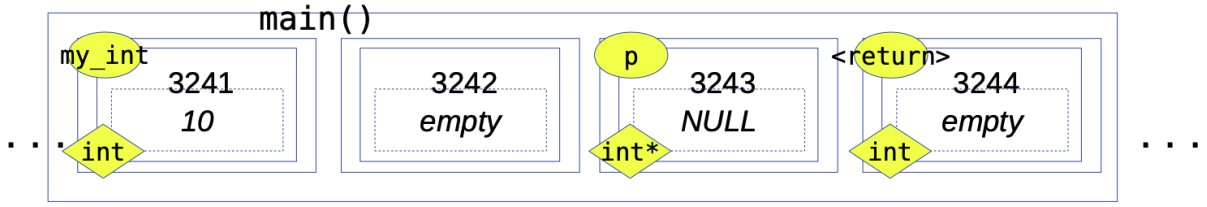
```
\a.out
My int is:  10
My int is:  15
My int is:  21
```

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

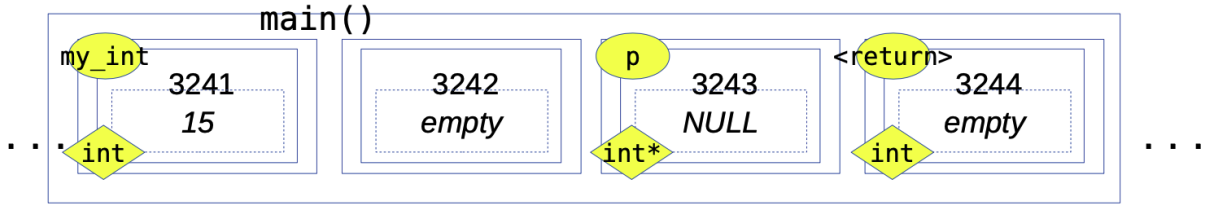
The syntax: **int *p=NULL;** specifies that what we are declaring is a *pointer* variable, and that the pointer will keep track of the locker where we stored an *int*. The pointer is initialized to ***'NULL'*** to indicate it's unassigned. In memory, this would look like:



Now main() updates 'my_int' to 10 and prints.



Next main() updates 'my_int' to 15 and prints.

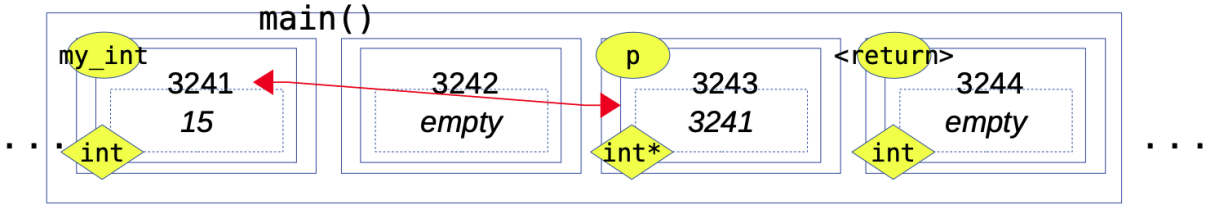


the next line is something we haven't seen before:

```
p=&my_int;
```

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, read the '&' as **the address of** so you know what the code is doing.

But, what is this *address* we are talking about? *The locker number, the box number reserved for my_int!*



So now, our pointer knows what locker number is 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).

IMPORTANT NOTE

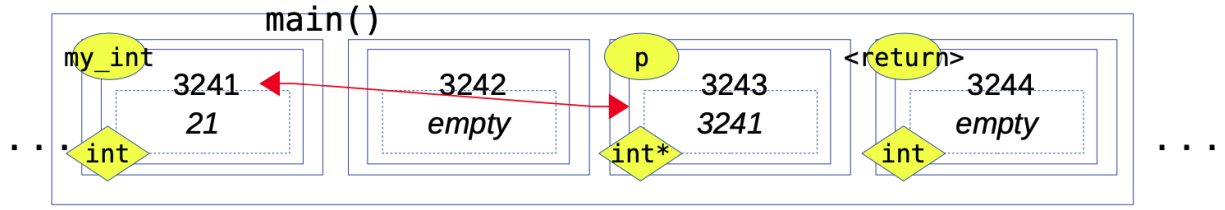
Note the meaning of '*' changes depending on what you're doing!

```
float *fp;  // Declares a pointer to a floating point value

*(fp)=3.14159265;   // Goes to the locker where a float value is
                    // stored and puts pi inside it.
```

So, what the program does with the line

`*(p)=21;`

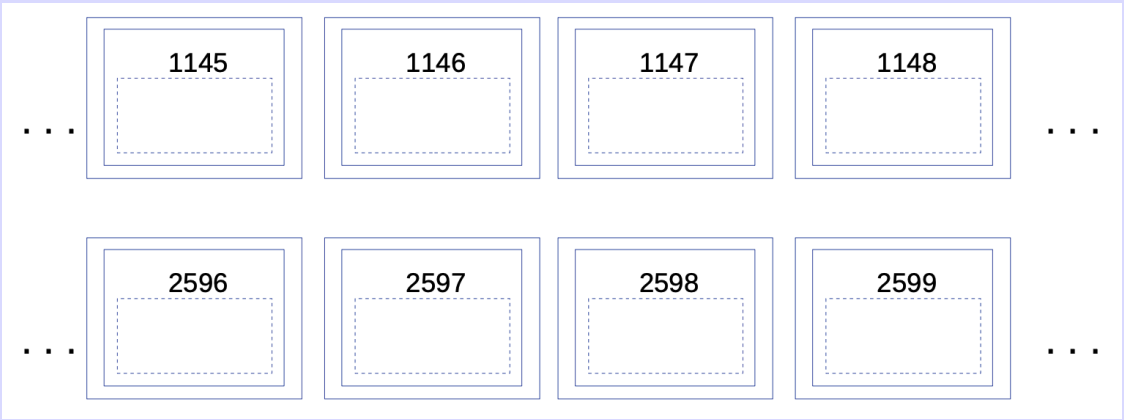
is: go to the box where 'p' is kept, get the number inside. We know it's a locker number (3241), so the program goes to locker 3241 and stores 21 inside, as we requested.



EXERCISE

In the memory diagram below, 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);
}
```



7.1 So why is this important?

As we mentioned before, without pointers the only way a function can change the value of a local variable outside of it is via its return value. We want to allow functions to access and modify data stored outside directly.

Let's update our example with the square() function so that it directly changes the variables being squared.

```
1 #include <stdio.h>
2 void square(int *p)
3 {
4     int x;
5     x=*(p);
6     x=x*x;
7     *(p)=x;
8 }
```

```
9  int main()
10 {
11     int my_int;
12     my_int=7;
13     square(&my_int);
14     printf("The final value for my_int is: %d\n",my_int);
15 }
```

Compiling and running the code gives us:

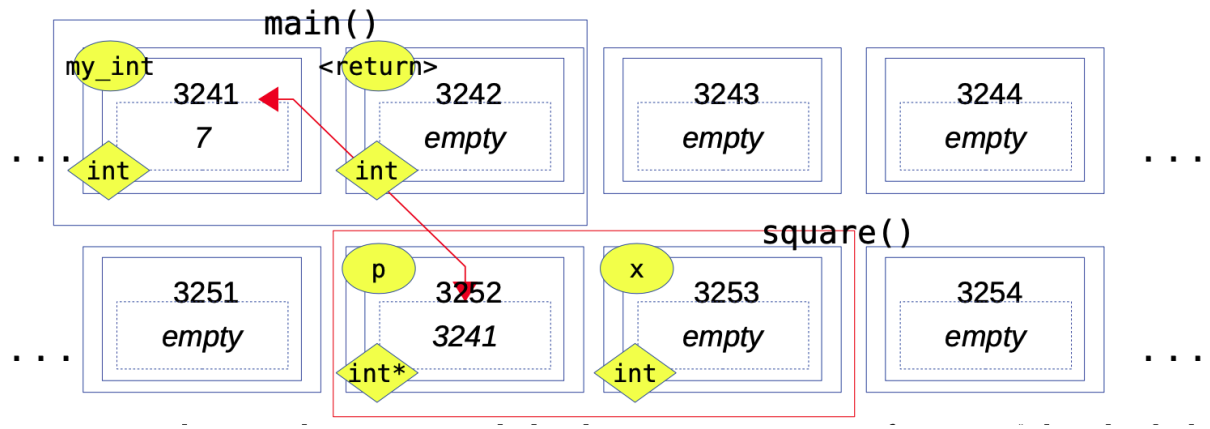
```
...\a.exe
The final value for my_int is:  49
```

Let's see what's happening in memory with the code above. First, main() declares an integer - type variable called 'my_int'. It sets the value of 'my_int' to 7, and then calls square:

```
square(&my_int);
```

Notice the '&', so what main() is passing to square() is 'the address of my_int'. In square(), we have declared an input parameter that is a *pointer to an integer*, and also a local *int* variable called **x**. *Notice that this time square() has no return value – in fact, it was declared 'void' and thus the compiler knows it never returns anything.*

Just after the call to square(), memory would look like this:

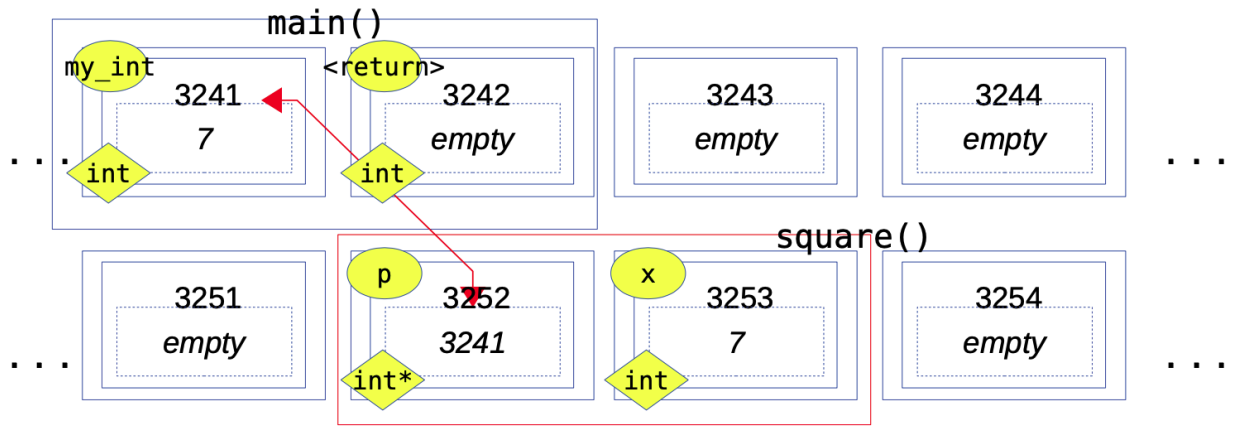


Notice that '*p', the pointer we declared as an input argument for square() has the *locker number* for 'my_int'. This means square() now knows where to find the integer value it's supposed to square.

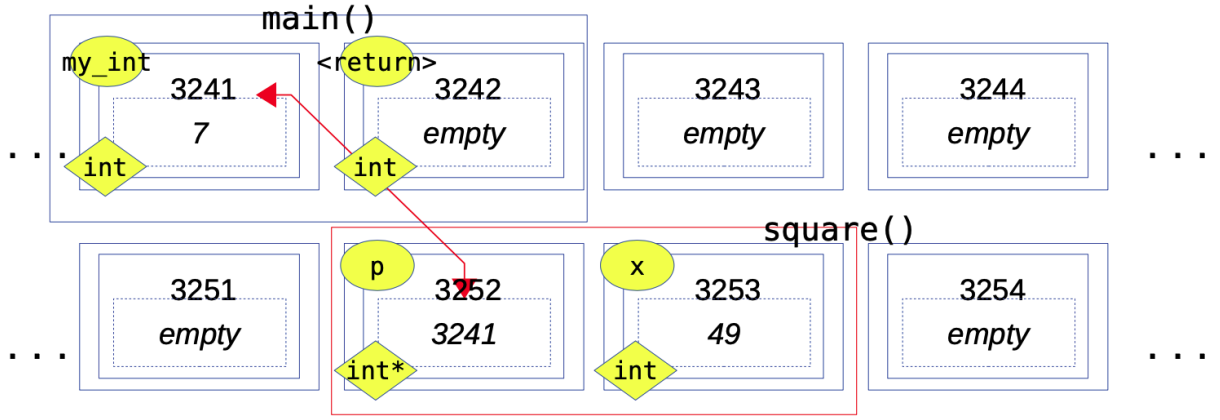
The next couple lines in square() are:

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

The first one says *'take the contents of the locker whose number is stored in (p) and copy them to x'*. This effectively copies the value of **my_int** onto **x**,



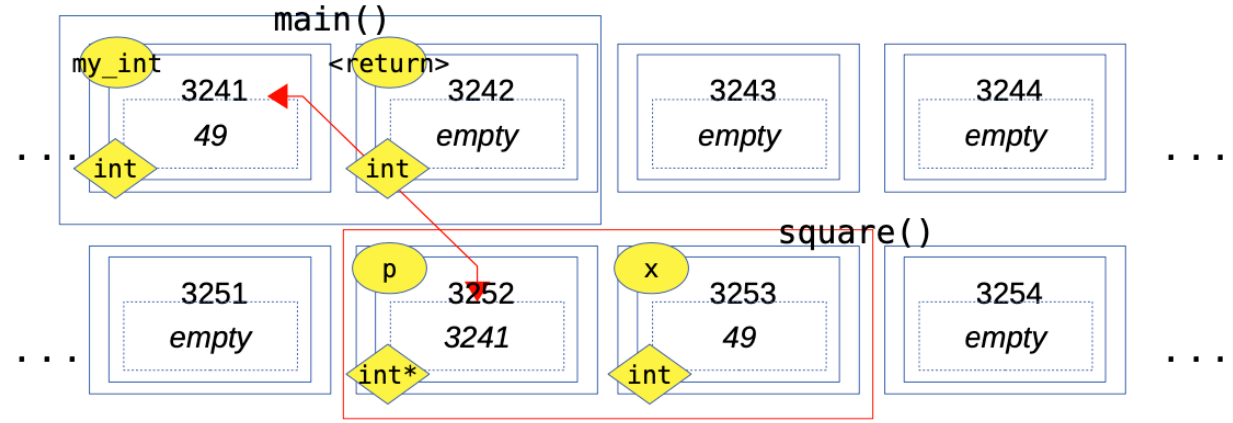
and the next line squares it.



The final line in square() uses the pointer to update the value of 'my_int' with its square.

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

This line reads *'make the contents of the locker whose number is in (p) equal to x'*. This effectively updates **my_int** with whatever is stored in **x**, which is 49, the square of the original value, 7, that was in **my_int** to start with.



7.2 Pointers and arrays

Pointers are particularly important when we are working with arrays. The reason is simple:

- Whenever we pass a variable from one function to another, a copy of the value of that variable is created.
- This is not necessarily appropriate for arrays. E.g. if you want to write a function to sort a very large array (say, 1,000,000 entries) it would be a waste of space and computer time to make a copy.
- Even if we were willing to spend time and use computer memory to pass a copy of the array to the sorting function, the sorting process would sort *the copy* of the input array, not the original array we wanted to sort!

So, in C, arrays are passed from one function to another as *pointers*.

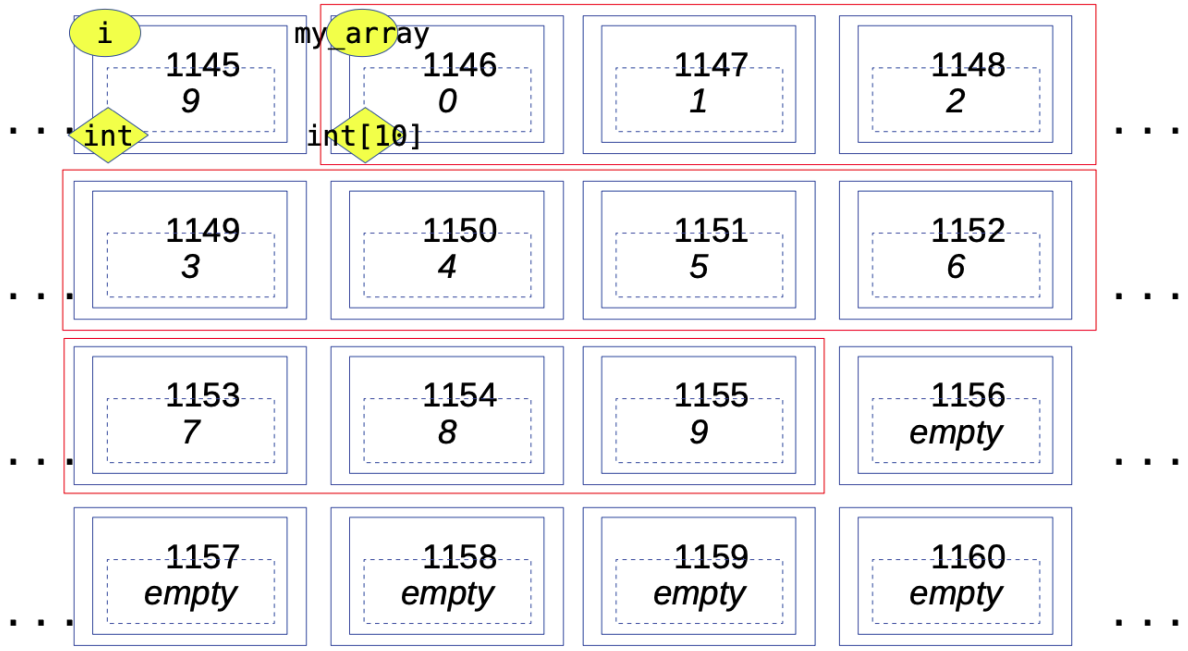
Let's see how this works:

```
#include <stdio.h>

void square_array(int array[10])
{
    int j;

    for (j=0; j<10; j=j+1)
    {
        array[j]=array[j]*array[j];
    }
}

int main()
{
    int my_array[10];
    int i;

    for (i=0; i<10; i=i+1)
    {
        my_array[i]=i;
    }

    square_array(my_array);

    for (i=0; i<10; i=i+1)
    {
        printf("%d squared equals %d\n",i,my_array[i]);
    }
}
```

Compiling and running the code above produces:

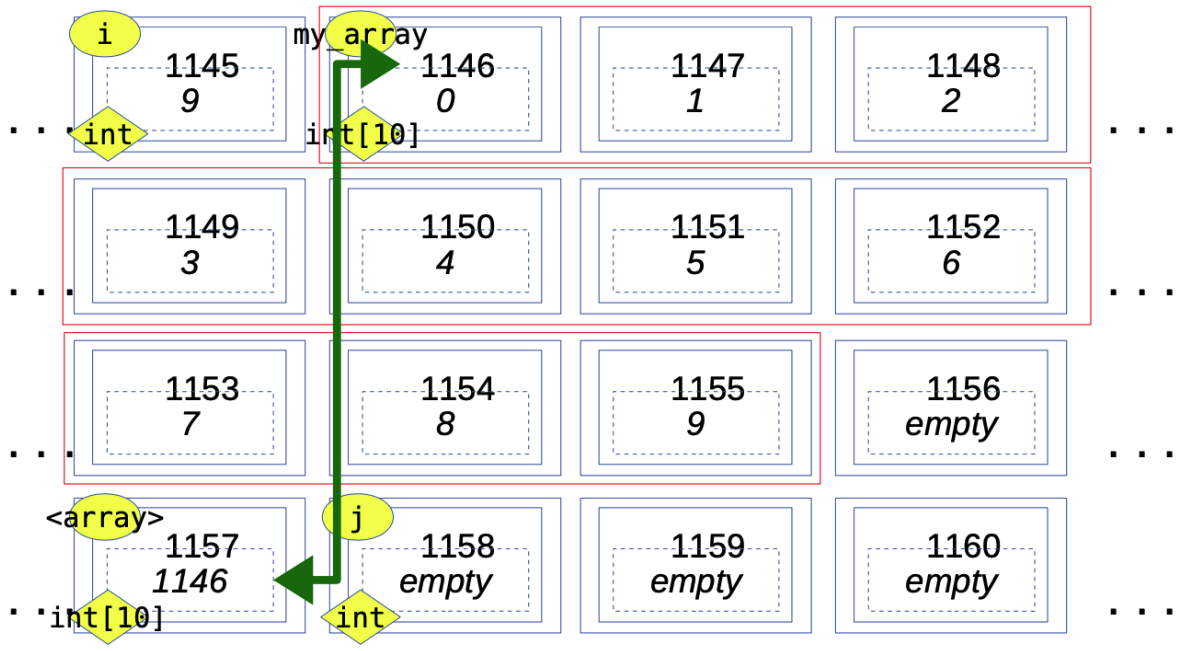
```
...\a.exe
0 squared equals 0
1 squared equals 1
2 squared equals 4
3 squared equals 9
4 squared equals 16
5 squared equals 25
6 squared equals 36
7 squared equals 49
8 squared equals 64
9 squared equals 81
```

Let's see what's happening in memory when we execute this code. First, main() reserves space for a 10-entry integer array, and fills it up with values from 0 to 9 (remember, array indexes for an array of length k go from 0 to $k-1$):



The memory reserved for the array is shown marked by red boxes. When the function square_array() is called, it reserves space for its input parameter – this is declared as an int array with 10 elements, ***but it does not create a separate array!*** Instead, because we are passing an array, C will reserve space for a *pointer to 'my_array', and copy the address of 'my_array' onto it* so that square_array() can access and modify the array directly.

The square_array() function also reserves space for an integer counter called j. At the moment of the call to square_array() the memory would look like this:



Notice that the **array** in square_array() is just a *pointer* to the original **my_array** from main().

Indeed, it contains just the locker number for the *first entry in 'my_array'*.

Thereafter, square_array() can use 'array' as it would any other array declared locally. So, the for loop in square_array() is *acting directly on the values stored in 'my_array'*. The end result of the for loop is that **my_array** gets updated with the square of each original entry.

Question: What memory location is being accessed if square_array() asks for array[3]?

Question: What do you think will happen if square_array() decides to do this:

```
array[11]=5;
```

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 (because strings are just character arrays). Indeed, if you look at the definitions for functions in the string library, they all take as input parameters *pointers to char arrays*.

7.3 Using pointers with offsets (pointer arithmetic) to access data

There is one more way in which pointers allow us to access information inside an array. The process is as follows:

- We declare an array of a given size
- We get a pointer to the *first entry in the array*
- Because array entries are stored consecutively in memory, we can use this pointer plus an *offset* to get any other entry in the array.

Example:

```c
#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 10

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

Compiling and running the code above results in:

```
...\a.exe array[0] is 5, array[4] is 10
```

Accessing arrays using pointers and offsets is very common in C. So you should be aware of what's happening when you find a program that uses expressions like the one above.

7.4 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;

e.g.

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

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

pointer = &variable;

Which is read as "assign to **pointer** the address of **variable**".

e.g.

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

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

The above initializes a float called 'pi', and sets the pointer 'fp' to the address of 'pi'.

How do we access or update values using pointers?

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

which read as "make the contents of the address stored in *(pointer)* equal to the value of **`expression`**", or "assign to *'variable'* the contents of the address stored in *(pointer)*".

e.g.

```
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];
```

Equivalently:

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

Because for arrays, the *name of the array is 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;

```

```
4  p=&array[0];
5
6  // Set the first entry in the array to 7
7  *(p)=7;
8
9  // Set the second entry in the array to 10
10 *(p+1)=10;
11
12 // Set the third entry in the array to 15
13 *(p+2)=15
14
15 // Set the last entry in the array to -15
16 *(p+9)=-15;
```

IMPORTANT NOTE

Be careful when using offsets and pointers to access data, like array indexes, C won't warn you if you are using an out-of-bounds offset or trying to access an invalid memory location, your program will behave unpredictably or crash.

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

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 test you wish, for example:
  "Now I know how to program in C!"
- Write a function that takes as input the character array for the string, 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 mess with the end-of-string delimiter*.
- Have your code print the reversed string
- Write a function that takes as input the character array for the string, a *query* character, and a *target* character, then goes over the string using *pointers and offsets* to replace any occurrence of the *query* character with the *target* character.
  e.g. suppose I call the function like so:

  ```
  search_and_replace(my_string,'a','o');
  ```

  Then the function should go over the input string, and replace any **a** with **o**.
- Have your code print the string after replacement

That's it!

And finally:

You've worked very hard in this section 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 5: Climbing a mountain is hard work, but the view from the top is worth it! Photo: WolfmanSF, Wikipedia Commons, CC-SA 3.0

Additional Exercises

Learning to program in a new language requires practice. Here are a few additional exercises for you to practice. Spend time working on them, and get help whenever you need to! Remember, there's lots of office hours this term, and drop-in lab sessions for you to use. Come and talk to us, and make sure you develop a solid ability to think about what's going on inside a program, and to implement algorithms in C.

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

Ex0 **-** Draw a memory diagram that illustrates what happens in memory when you run the following snippet of code:

```
1  void main()
2  {
3      int x;
4      int y;
5      float pi;
6
7      x=5;
8      y=x+3;
9      pi=y/2;
10
11     printf("%f\n",pi);
12 }
```

What will be the final value printed by the program?

EXERCISE

Ex1 **-** Consider the following snippet of code

```
1  float  convert_to_degrees(float angle)
2  {
3      // This function converts an angle given in radians to degrees
4      return angle*360.0/(2.0*3.1415926535);
5  }
6
7  void main()
8  {
9      int x;
10     float ang;
11
12     x=2;
13     ang=convert_to_degrees(x);
14     printf("%f\n",ang);
15 }
```

Draw a diagram that shows what happens in memory *at the moment of the call to convert_to_degrees()*. It should show the memory reserved for *main()* and for the *convert_to_degrees()* function.
What is the final value of 'ang'?

EXERCISE

Ex2 **-** Consider the next little program

```
1  void hard_working_function(int x)
2  {
3      x=x*x;
4      x=x/x;
5      x=3*x;
6      x=x+71;
7      x=x/2;
8  }
9
10 void main()
11 {
12     int x;
13     x=1;
14     hard_working_function(x);
15     printf("%d\n",x);
16 }
```

What is the final value printed by the program?

EXERCISE

Ex3 **-** 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

Ex4 **-** Remember that when we pass an array to a function, the function gets a pointer to the array *(not a copy)*, and recall that strings are arrays of chars. Now consider this program:

```
1  void start_a_story(char input_string[1014])
2  {
3      char little_story[2048];
4      strcpy(little_story,"Once upon a time...");
5      strcat(little_story,input_string);
6      printf("%s\n",little_story);
7  }
8
9  void main()
10 {
11     start_a_story("there was a blue rhinoceros named Randolph.");
12 }
```

What is the output of this little program?

EXERCISE

Ex5 **-** Now consider the modified program below:

```
1  char *start_a_story(char input_string[1014])
2  {
3      char little_story[2048];
4      strcpy(little_story,"Once upon a time...");
5      strcat(little_story,input_string);
6      return little_story;
7  }
8
9  void main()
10 {
11     char *story;
12     story=start_a_story("there was a blue rhinoceros named Randolph.");
13     printf("%s\n",story);
14 }
```

There is a ***major problem*** *with the code above. What is it?*
Re-write the code above so that main() can call 'start_a_story()' and somehow end up with a string that contains the little story.

EXERCISE

Ex6 **-** Let's consider the following little program

```
1  void main()
2  {
3      char little_string[1024];
4      char *p=NULL;
5      strcpy(little_string, "This is just a little harmless string");
6
7      p=&little_string[0];
8
9      printf("%c\n", *(p));
10
11     printf("%c\n", *(p+3));
12
13     p=p+5;
14     *(p)='u';
15     printf("%s\n", little_string);
16 }
```

What is the output of the little program?