# Pointers

## Objectives

```
1. What is a pointer
2. The relationship between pointers and arrays
3. Homework exercises
4. Next steps
```

## 1.1 What is a Pointer

- Pointers are one of the most extraordinarily powerfull tools in the C language and also, the most confusing one if it is not used accordingly!
- When we first talked about memory, we said that whenever you, the developer, declare a variable, the computer allocates an area of memory. We refer to this area of memory using the variable name but once the program is compiled and running, the computer references it by the address of the memory location.
- Let's take a look at the following snippet:

  ```
  int number = 5;
  ```

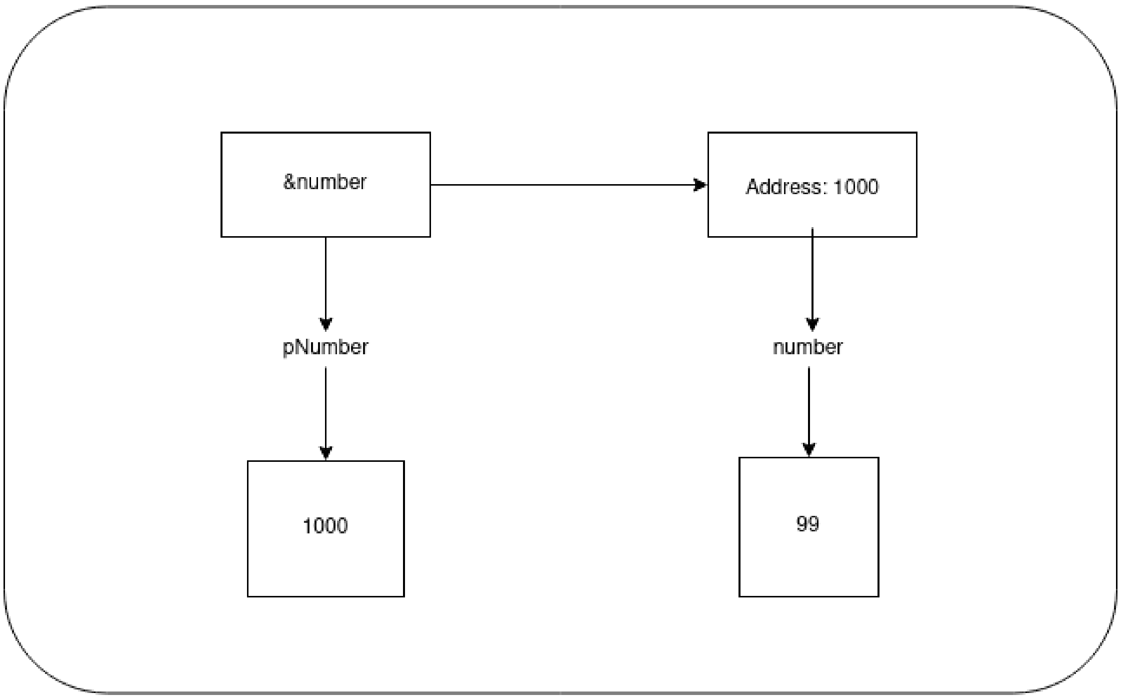
  - Here, memory is allocated to store an integer and we can simply reference it using the name `number`. The value 5 is stored in this area.
  - The computer references the area using an address. The specific address where these data will be stored dependes on the operating system and the compiler used for compiling the program and most likely will be different on different systems (and even on different runs on the same system)
- In order to get the address of a variable, we use the symbol `&`. Let's take a look at the following snippet where we see how to find the address of a variable:

  ```
  #include <iostream>

  using namespace std;
  int main() {
      int number = 10;
      cout <<"The address of number variable is: "<< &number;
      return 0;
  }
  ```

  - The most important part here is the fact that if you run this program for multiple times, most likely you will get a different address displayed. (Something of the form: `0x7ffc7cf9d8bc`)

- Now if we get back to our `number` variable, and exactly to what kind of variable is, it is a variable which has, as a value, the number 5. Now, what if we want to have a variable which has, as a value, an address? Well, this is a `pointer`:
    - `Pointers are variables that store addresses, and the address that's stored in a pointer, is usually that of another variable`
- Let's see another example:

```
#include <iostream>

using namespace std;
int main() {
    int number = 99;
    int *pNumber = &number;
    return 0;
}
```

  - Above we have a pointer `pNumber` that contains the address of another variable, called `number`, which is an integer variable containing the value 99. The value that's stored in `pNumber` is the address of the first byte of number.
  - A schematic representation can
      - 

  - As you see, we declare a pointer similar to an ordinary variable only that there is an asterisk prior to variable's name: `int *pointerToSomething`
- The first thing to note is that it is not enough to know that a particular variable, such as `pNumber` is a pointer. The developer, and more importantly, the compiler must know the type of data stored at the address to which the pointer points to.

  - Without this information, it's impossible to know how much memory is occupied or how to handle the contents of the memory to which it points
  - This has the following consequence: `Pointers of a specific type can only point to variables of that type` e.g `pointers to float` can only point to variables of type `float`.
  - Even if we repeat this kind of information, keep in mind that a pointer of a given type is written as `type*` (e.g `int* myPointer`)

- One special datatype of pointer is the pointer where the type is void (e.g `void* pointerToVoid`)

  - The type named `void` means abscence of the type, thus it can contain the address of a data item of any type.
  - Type `void*` is often used as a parameter type or return value type with functions that deal with data in a type-independent way.
  - Any kind of pointer can be passed around as a value of type void* and then cast to the appropriate type when you come to use it
  - We will see more about this later when we will talk abbout the `malloc` function which is used to allocate memory for use in your program and returns a pointer to the memory of type `void*`

## 1.2 Declaring pointers deep dive

- You can declare the pointer in one of the following ways:
  1. `int* pNumber`
  2. `int *Pnumber`
- Even if you can use both of them, please stick to one and use it consistently!
- If we have the following snippet:

  ```
  int *pNumber;
  ```

  this just creates the `pNumber` variable but doesn't initialize it. Uninitialized pointers are particularly hazardous, much more dangerous than on ordinary variable that is uninitialied, so you should always initialized a pointer when you declare it.
- You can initialize `pNumber` so that it doesn't point to anything by rewriting the declaration like this:

  ```
      int *pNumber = NULL;
  ```

  - `NULL` is a constant that's defined in the standard library and is the equivalent of zero for a pointer.
    - It is a value that's guaranteed not to point to any location in memory. This means that it implicitly prevents the accidental overwriting of memory by using a pointer that doesn't point to anything specific.
    - `NULL` s defined in several standard library header files, including stddef.h, stdlib.h, stdio.h, string.h, time.h, wchar.h, and locale.h. Anytime that it’s not recognized by the compiler, just add an #include directive for stddef.h to your source file.
- If you want to initialize the `pNumber` variable with the address of a certain variable, remember that we have to use the `address of` operator, namely `&` (see snippet at the beginning of the lesson)

- It is very important that the declaration of the variable to which addres the pointer points to, is declared before the pointer. If this is not the case, the compiler will complain!
- There is nothing special about the declaration of a pointer. You can declare regular variables and pointers in the same statement, for example:

  ```
  double value, *pVal, num;
  ```

  - The above snippet declared two double precision floating-point variables (`value` and `num`), and a variable `pVal` of type `pointer to double`.
  - It is very clear that only the second variable (`pVal`), is a pointer but consider the following statement:

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

    - This declares a pointer, p of type `int*`, and a variable, q, that is of type `int`. It is a common mistake to think that both p and q are pointers.

## 1.3 Accessing a Value through a pointer

- When we want to access the value of the variable pointed to by a pointer, we use the `*` operator which is also called the `indirection operator`

  - You can also find this operator called as the `dereference operator`, because you use it to `dereference` a pointer

- Let's see this operator in practice in the snippet below:

  ```
  int number = 15;
  int *pointer = &number;
  int result = 0;
  result = *pointer + 5; // result will be equal to 20
  ```

  - The pointer variable contains the address of the variable number so you can use this in an expression to calculate a new value for result, like in the last statement above.
  - The expresion `*pointer` will evaluate to the value stored at the address containted in the pointer. This is the value stored in number 15, so result will be set to 15+5, which is 20.

- Now let's see a couple of example with pointers where we will practice everything we learned so far:

  ```
  #include <iostream>
  using namespace std;

  int main() {
      int number = 15; // A variable of type int initialized to 0
  ```

```
        int *pointerToNumber = NULL; //A pointer that can point to type int

        number = 10;
        cout << "number's address: " << &number << endl; // outputs the
address of the number variable
        cout << "number's value: " << number << endl; // outputs the value
stored inside the number variable;

        pointerToNumber = &number; // Store the address of number variable
inside the pointerToNumber pointer

        cout << "pointerToNumber's address: " << &pointerToNumber << endl;
// outputs the address of the pointerToNumber;
        cout << "pointerToNumber's size: "<< sizeof(pointerToNumber) <<
endl; // outputs the size of the pointerToNumber;
        cout << "pointerToNumber's value: " << pointerToNumber;  // outputs
the value => a.k.a the address to which it points
        cout << "pointerToNumber's pointed to, value: "<< *pointerToNumber<<
endl; // outputs the value which can be found at the address stored inside
the pointer;

        return 0;
    }
```

## 1.4 Using pointers

- Because you can access the contents of a pointer which points to a number, you can use the dereference operator to compute arithmetic operations as in the snippet below:

```
    int number = 10;
    int *pNumber  = &number;
    *pNumber += 25;
    cout << number; // will print 35;
```

  - The statement above increments the value of whatever variable `pNumber` points to by 25. The `*` indicates you're accessing the content to which the variable called `pNumber` is pointing, in this case it's the content of the variable called number.

- The variable `pNumber` can store the address of any variable of type int. This means you can change the variable that `pNumber` points to like this:

```
    int value = 999;
    pNumber = &value;
```

  - Now if we would repeat the same statement that we have used previously:

```
    *pNumber += 25;
```

The statement will operate with the new variable, value, so the new contents of value will be 1024.

- What you should remember if the fact that a pointer can contain the address of any variable of the appropriate type, so you can use one pointer variable to change the values of many different variables, as long as they re of a type compatible with the pointer type.

- Let's see an example where we modify values stored in other variables, using pointers:

```
#include <iostream>
using namespace std;

int main() {
    long num1 = 0L;
    long num2 = 0L;
    long *pNum = NULL;
    pNum = &num1; // Get address of num1
    *pNum = 2L; // Set num1 to 2
    ++num2; // Increment num2
    num2 += *pNum; // Add num1 to num2
    pNum = &num2; // Get address of num2
    ++*pNum; // Increment num2 indirectly
    cout << "num1 = " <<num1 << endl;
    cout << "num2 = "<< num2 << endl;
    cout <<"*pnum = "<<*pNum << endl;
    cout << "*pnum + num2 = "<< *pNum + num2;
    return 0;
}
```

- Now let's rewind the old times when we were using `scanf` to read from the keyboard. If you recall, if we had to read an integer variable from the keyboard, we would have had to do something like:

```
int value;
printf("Enter an integer: ");
scanf("%d", &value);

print("You have entered: %d", value);
```

Here we have used to `&` operator to obtain the address of the variable that is to receive the input and used that as the argument to the function. When we have a pointer that already contains an address, we an use the pointer name as the argument, thus the example above will look like:

```
int value;
int *pValue = pValue;
```

```
    printf("Enter an integer: ");
    scanf("%d", &value);
    printf("You have entered: %d", value);
```

## 1.5 Testing for a NULL pointer

- Let's suppose that we create a pointer like this:

  ```
  int *pValue = NULL;
  ```

  As we know, `NULL` is a special symbol in C/C++ that represents for the pointers what 0 is for ordinary numbers. The Symbol is often defined as `((void*)0)`. When you assign 0 to a pointer, it's the equivalent of setting it to `NULL`, so you coud rewrite the above statement as:

  ```
  int *pValue = 0
  ```

  and because NULL is the equivalent of zero, if we want to test whether `pValue` is `NULL`, we can write this:

  ```
  if(!pValue) {
      //Do something when the pointer is NULL
  }
  ```

## 2. The relationship between pointers and arrays

- Before getting our hands dirty, let's recap what an array is and what a pointer is:

  1. An `array` is a collection of objects of the same type that you can refer to using a single name
     - for example, an array called `scores[50]` could contain all the basketball scores for a 50-game season. Then, you use a different index value to refer to each element in the array. The `scores[0]` is the first score and `scores[49]` is the last score.
  2. A `pointer` is a variable that has as its value a memory address that can reference another variable or constant of a given type. You can use a pointer to hold the address of different variables at different times, as long as they're all of the same type

- Arrays and pointers seem quite different, and indeed they are, but they are really very closely related and can sometimes be used interchangeably.

- Let's see some examples where arrays and pointers work together. These examples link together as a progression. With more and more practical examples of how arrays and pointers can work together, you should find it fairly easy to grasp the main ideas behind pointers and their relationship to arrays.

- In the snippet below we can see that an array name, by itself, refers to an address:

```
    #include <iostream>
    using namespace std;

    int main() {
        char multiple[] = "My string";

        char *p = &multiple[0];
        cout << "The address of the first array element is: " << (void *) p
<< endl;
        p = multiple;
        cout << "The address obtained from the array name: "<< (void
*)multiple << endl;
        return 0;
    }
```

- Here I didn't pasted the output as it will be different at each run (most of the time) but what I wanted you to see is that everytime, the same output will be displayed twice. That's because, always, the name of an array represents the address of the first element in the array.
    - Note that in order to display an address with `cout` you have to perform that cast `(void*)`
- Now let's see an example which will illustrate the effect of adding an integer value to a pointer that points to an array

```
    #include <iostream>
    #include <string.h>

    using namespace std;

    int main() {
        char multiple[] = "My string";

        char *p = &multiple[0];
        for(int i = 0; i < strnlen(multiple, sizeof(multiple)); i++){
            cout << "Using the pointer:    multiple["<<i<<"] = "<< *(p+i) <<
endl;
            cout << "Using the array name: multiple["<<i<<"] = " << *
(multiple+i) << endl;
            cout << "-------------------------------------------" << endl;
        }
        return 0;
    }
```

    - This example is a bit more complicated so let's take it step by step:
        1. First we declare an array of char containing some text
        2. Then we initialize the `p` pointer to the address of the first element
        3. We iterate through each of the characters in the array by using a `for` statement and the `strnlen` function which takes as input the array and its size

4. We display the value which can be found at each position in the array by dereferencing the pointer plus the position. You can see this similar to:
   - *p points to the first element
   - _(p+n) is like go the n'th element from the beginning(e.g _(p+3) is the fourth element because is the one which is at a distance of three positions from the first)
5. We display the same thing, only that this time we use the name of the array instead of the pointer name. This is just to prove that the name of the array represents the address of the first element.

- Very important: An array name refers to a fixed address and is not a pointer. You can use an array name and thus the address it references in an expression but you cannot modify it.

## 3. Homework exercises

```
1. Write a program in C++ to show the basic declaration of a pointer. The program
should output the address of the pointer, the address of the pointed to variable,
and the value stored at the address stored in the pointer.

2. Write a program in C++ which adds two numbers using pointers
    - Sample Input: 3 5
    - Sample Output: 8
3. Write a program in C++ which will output the maximum of two numbers using
pointers
    - Sample Input: 3 6
    - Sample Output: second number is greater
4. Write a program in C++ which will store elements inside an array and then will
print all the elements, one per line, using a pointer.
```

## 4. Next steps

- In the upcoming lessons we will learn about how can we dynamically allocate memory. This will heavily make use of pointers.
- Also, we will learn how to handle Strings using pointers