# Pointers

**T1** Chapters: **24-26**

When we create variables in our programs, they occupy some amount of space in the computer's memory (RAM). How much, depends on the type definition of the variable. In other words, a char variable occupies less space compared to a double. But, in general, every variable, by virtue of occupying space in memory, has a location. This location is called an **address**, and a variable that stores such an address is called a **pointer variable** or simply **pointer**. This is because, by holding the address, the pointer variable is pointing to the location contained inside it.

This feature is quite powerful and most real-world problems to some extent will make use of them either for efficiency or to implement a particular functionality that necessitates memory access. Effectively what this means is that pointers allow for indirect access to the contents of a variable or an object in general.

**NOTE**

An object is always something, usually a variable of a particular type, that has been initialized and is occupying space in memory.

In the C/C++ programming languages, there are two operations and thus two terms you must familiarize yourself with - **referencing** and **dereferencing**.

**Referencing** uses the & operator to obtain the address of an object, while **dereferencing** uses the \* operator to access the contents at the memory location pointed to by the address.

Declaring a pointer variable has the structure TYPE\* NAME = ADDRESS;, where TYPE refers to the type of the variable or object at the given ADDRESS. Reading the left hand side from the right, it goes as NAME is a pointer to an object of type TYPE.

For instance,

int x = 5;  
int\* px = &x; // reference x  
x = 2 \* (\*px); // dereference px, and modify the value of x

If we were to run the above snippet of code, you will notice that the new value of x would be 10. Here, the address of x is stored in px, which is dereferenced to access the contents of x, and in the last line, we multiply the value at the address x with 2 and overwrite the original value at the address.

## Relationship with Arrays

An array is a container object that facilitates the storing of objects of a particular type together, as seen in the previous lecture. From the perspective of computer memory, an array is just a contiguous block in memory. When a program declares an array variable, the operating system allocates the requested amount of memory for this purpose and returns the starting address of this block to be stored in the variable.

For instance,

int x[4] = {1,2,3,4};  
int\* px = x;  
int\* py = &x[0];

Here, px and py are both pointers holding the address to the start of the block of memory allocated by the operating system for x. The assignment of px refers to the phenomenon termed pointer decay, wherein the name of an array decays into a pointer to the start of the array.

**NOTE**

const TYPE\* vs TYPE\* const vs const TYPE\* const

The distinction between the three, and their meaning, is quite important and useful to know.

1. Pointer to a constant value. In other words, modifications to the value through the pointer are forbidden.
2. Constant pointer to a value. In other words, the pointer cannot be modified but the value its pointing to can.
3. Constant pointer to a constant value. In other words, both the pointer and the value cannot be modified.

Elements of an array can be accessed through pointers because of the pointer decay phenomenon, and this is the basis on which C++ iterators are founded. There is much more to be said about this, which will be deferred to another time. For example,

int x[4] = {1,2,3,4};  
for (int\* b = x, e = x + 4; b != e; b++)  
{  
 \*b += 2;  
 printf("%d\n", \*b);  
}

In the snippet above, b and e are declared to hold pointers to the start and end of the array x, which would, technically, be called a **span**. The for loop iterates over the array x through the pointer b until it reaches the end e, while incrementing the values it is pointing to by 2 and displaying the results. In the header of the loop, the expression b++ is performing **pointer arithmetic**, where in, the address pointed to by b is incremented by 1 unit.

**NOTE**

It is generally not advised to use pointer arithmetic to iterate over an array or access elements of arrays. This is because it is quite easy to make mistakes and also hurt readability. Most of the time, use the normal **subscript operator** [] to access elements of arrays, and let the compiler take care of the conversion to pointer arithmetic.

## References in C++

Consider the scenario where the address of an object is stored in a pointer variable, and is used to access this object indirectly. Also consider that this object is suddenly gone or simply deleted. This results in a situation where the pointer holds the address of an object that does not exist anymore. The name for this type of a pointer is **dangling pointer**. Accessing memory through such a pointer can cause your program to crash or in the worst case something seriously bad to happen. An example of how such a situation can arise is as follows,

int x = 4;  
int\* px = nullptr;  
{  
 int y = 5;  
 px = &y;  
}  
printf("%d\n", \*px);

To address such situations and better safety around the use of pointers, **references** were introduced by C++. A reference preserves the indirect access to an object property of a pointer but discards the rest. More specifically, you lose -

1. Pointer Arithmetic
2. Null Pointers
3. Constant Pointers

Importantly, note that references always refer to complete objects.

For instance,

int x[4] = {1,2,3,4};  
int& rx = x; // error! will not compile  
int (&rx)[4] = x; // correct!

The last assignment says that rx is a reference to the array x. However, int& x[4] means that x is an array of references to integers.

**NOTE**

1. There is only const TYPE& but no TYPE& const. The latter will, in fact, throw an error.
2. Most cases of pointer use can be replaced with references, and hence it is generally advised to use the latter whenever possible.

## Exercises

1. What will be the contents of the a array after the following statements are executed? (E)

* #include <cstdio>  
  #include <array>  
    
  int main()  
  {  
   constexpr int N = 4;  
     
   int a[N] = {1,2,3,4};  
   int\* p = &a[0];  
   int\* q = &a[N-1];  
   int tmp = -1;  
     
   while (p < q)  
   {  
   temp = \*p;  
   \*p++ = \*q;  
   \*q-- = temp;  
   }  
  }

1. Rewrite the following program to use pointer arithmetic instead of array subscripting. (Hint: Eliminate the variable i and all uses of the [] operator.) Make as few changes as possible. (M)

* #include <cstdio>  
  #include <array>  
    
  int main()  
  {  
   const int x[4] = {1,2,3,4};  
   int sum = 0;  
   for (int i = 0; i < std::size(x); i += 1)  
   {  
   sum += x[i];  
   }  
  }

1. Write the following program such that the pointers hr, min, and sec store the time in time\_sec representing the number of seconds since midnight, in hours (0-23), minutes (0-59), and seconds (0-59), respectively. Then the time should be displayed in the format HH:MM:SS. (M)
2. Given an array of length N, write a program that will search for the largest and second largest numbers, storing them in the pointers p\_fl and p\_sl. Display the position of these values in the array and their values through the pointers. (M)
3. Assume that the two dimensional array temperatures[7][24] contains a week's worth of hourly temperature readings, with each row containing the readings for one day. (H)
   1. Write a program that searches the entire array for the value 32.
   2. Print all temperature readings in every row, using a pointer to visit each element of the row.
4. Write a program that reads a message, then checks whether it's a palindrome. Ignore all characters that aren't letters. (H)

* Enter a message: He lived as a devil, eh?  
  Palindrome  
    
  Enter a message: Madam, I am Adam.  
  Not a palindrome
  1. Revise the program to use pointers instead of integers to keep track of positions in the array.
  2. Simplify the program by taking advantage of the fact that an array name can be used as a pointer.

## Homework

1. Read **T1** Chapters:
   1. **30-32**
2. The following function, which computes the area of a triangle, contains two errors. Locate the errors and show how to fix them. (Hint: There are no errors in the formula.) (E)

* double triangle\_area(double base, height)  
  double product;  
  {  
   product = base \* height;  
   return product/2;  
  }

1. Write a function check(x,y,n) that returns 1 if both x and y fall between 0 and n-1, inclusive. The function should return 0 otherwise. Assume that x, y, and n are all of type int. (M)
2. Write a function num\_digits(n) that returns the number of digits in n (a positive integer). Hint: To determine the number of digits in a number n, divide it by 10 repeatedly. When n reaches 0, the number of divisions indicates how many digits n originally had. (H)