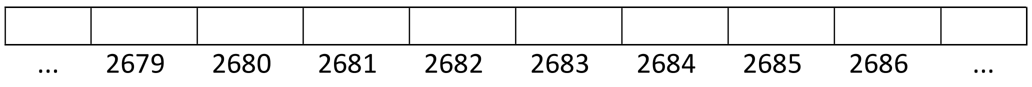
Pointers

So far we accessed variables in memory through their identifiers. We didn't need to worry about how and where they are stored physically.

The computer memory is organized as an array of cells. Each cell has a size of one byte. These cells are numbered consecutively. So, each cell has an address the same as the previous one plus one. The diagram below shows a memory block of cells and their addresses:



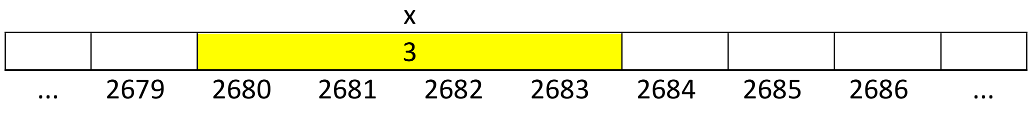
We need one memory location (byte) to store a `char`, two memory locations to store a `short int` (2 bytes), 4 locations to store an `int` (4 bytes) and so on.

Reference operator (&)

Once a variable is declared the required amount of memory is assigned for it at a specific location in memory. For example, assuming integer type requires 4 bytes, the declaration,

```
int x = 3;
```

takes 4 memory cells as shown bellow



and we say that the variable x is stored at memory address 2680 (the starting address of the x variable).
To obtain the memory address that locates a variable, we precede the variable with the reference operator (&) as:

```
cout << &x << endl;
```

will print the address of the variable x.

Pointer declaration

A pointer is a variable that stores a memory address. Therefore, it can be used to refer to (points to) other variables within the memory. A pointer variable can be declared similar to any other variable with only one difference that its identifier must be preceded by the asterisk (*) as follow:

```
dataType  *pointer;
```

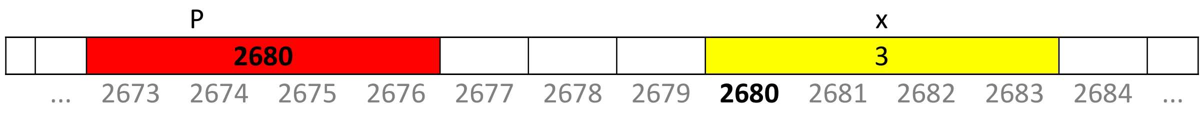
where *`dataType`* is the data type of the variable that the pointer is intended to store its address. This means that we can assign to the pointer variable, the address of another variable that has same data type (*`dataType`*). One exception is when using a void pointer. For example:

```
int x = 3;
int *p = &x;   //ok
```

but the assignment

```
float *q = &x; // error ..  x is an int while q is float *
```

Note: the * here is just to specify that the identifier is a pointer.



Pointer p points to the variable x (hold the x variable address)

Dereference operator (*)

To access the value at the memory address held by a pointer we use the dereference operator * preceding the pointer. For example:

```
int x = 3;
int y = x;       // assign the value of x to y
int *p = &x;     // assign the address of variable x to the pointer variable p
int y = *p;      // assign the value that p points at, to the variable y
*p = 30;         // update the value that p points at (which refers to x )
cout << x << endl;  // will print 30
```

Also array elements are variable

```
float A[] = {1,2,3,4};
float *p = &A[2];   // assign the address of A[2] to the pointer p
*p = 300;
cout << A[2] << endl; // prints 300
```

If a pointer p points to a variable x, then x can be replaced by (*p) in any context. For example,

```
x = x + 20
```

can be replaced by

```
*p = *p + 20;     // increment *p by 20
```

Pointer arithmetic:

Arithmetic operations on pointers are different from arithmetic operations on integers. To start with, only addition and subtraction are allowed in pointer arithmetic and they have different behavior depending on the data type of the object they point to. Different data type occupies different space in memory. For example, char occupy 1 byte; short occupy 2 bytes, while int takes 4 bytes.

If we declare a pointer p to point to a variable of data type "type" as,

```
type *p;
```

Then if we increment p by one

```
p = p + 1;
```

the address held by p will be incremented by the size of type.

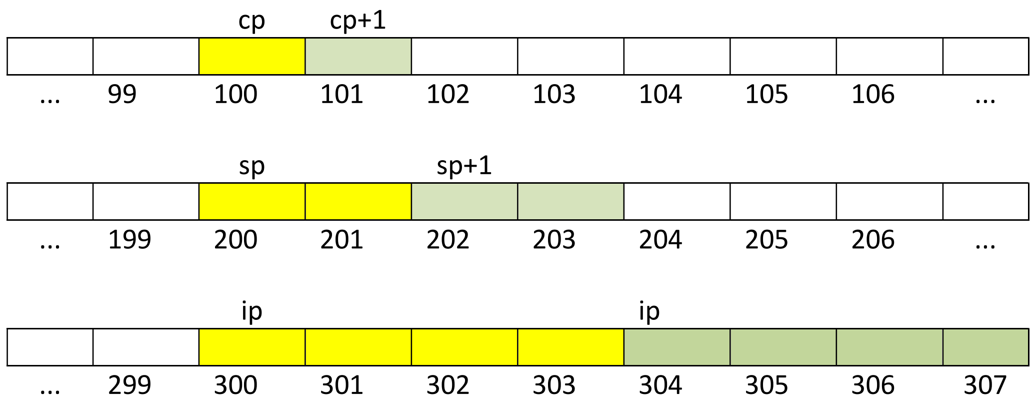
For example, suppose we declare the following 3 pointers,

```
char *cp;
short *sp;
int *ip;
```

and assume that they point to locations 100, 200, 300.
Now if we increment each pointer by one,

```
cp = cp + 1;
sp = sp + 1;
ip  = ip + 1;
```

The address in each pointer will be incremented by the size of the corresponding data type, cp will contain the address 101 (100 + sizeof(char)), ip will contain 202 (200 + sizeof(short)), and ip will contain the address 304 (300 + sizeof(int). See diagram below



Operator precedence:

The reference (&) and dereference (*) operators has higher precedence than arithmetic operators and lower precedence than the unary operators ++ and -- . For example,

```
int x = 4, y;
int *ip = &x;
int *iq;
```

y = *ip + 1; // adds 1 to 4 and assign the result (5) to y

*ip += 2; // adds 2 to the value that ip points to which is x.

cout << x; // prints 6

++*ip; // increments what the value ip points to

cout << x; // prints 7

(*ip)++; // increments the value that ip points to. Note the () are necessary here, otherwise it will increment the pointer first and then dereference.

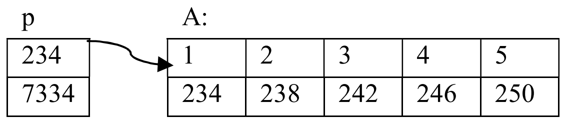
cout << x; // prints 8

iq = ip; // both ip and iq point to x now

Pointers and Arrays

The relationship between arrays and pointers is very close. In fact the name of the array sometimes thought as a constant pointer to its first element (in fact it is not, will come back to this).

Examples:



```
int A[5] = {1, 2, 3, 4, 5};
int *p;
p = A; // Assigns the address of the first element to the pointer p

cout  <<  sizeof(A)  << endl; // prints 20 (the size of array of 5 integers )
cout  <<  sizeof(p)   << endl; // prints 4 (the size of a pointer)

cout << A << endl; // prints the address of the first element of the array -- (234 in the above example)
cout << p << endl; // prints the address of the first element of the array  -- (234)

A = A+1;   // Error, can't increment the name of the array
p = p + 1;  // now p holds the address of the second element
cout << p << endl; // prints 238

p = &A[3]; // now the address of the 4th element is assigned to the pointer p
cout  <<   p << endl; // will print 246
cout  << *p << endl; // will print what is stored at the address 246 which is 4

cout  <<  *(A+2)  <<  endl;  // prints the number stored at address 234 + 8 = 242 which is 3
```

the same as

cout << A[2] << endl; // which is 3.

Be careful about the brackets

cout << *A + 2 << endl; // is equivalent to A[0] + 2 which is 1+2 = 3

We can obtain the address of the third element as

&A[2]

or

A+2

More examples:

The sum of the array element using array subscript notation

```
int sumArray(int X[], int size) {
        int sum = 0;
        for (int i=0; i<size; i++) {
                sum += X[i];
        }
        return sum;
}
```

using pointer offset notation

```
int sumArray(int *X, int size) {
        int sum = 0;
        for (int i = 0; i<size; i++) {
                sum += *(X+i);
        }
        return sum;
}
```

void pointers

The void data type in c++ represent the absence of data type. So a void pointer is a pointer that points to a value that has no data type and so has no determined length and can't be derefernced using the * operator. A void pointer can point to any data type (e.g. int, short, char, etc) and it has to be casted to a concrete object before it can be dereferences. It is usually used to pass generic parameters to a function.