Pointers (cplusplus.com)
succession = birbirini izleme, art arda gelme
precede = -den önce olmak

Generally, C++ programs **do not actively decide the exact memory addresses** where its variables are stored.
Fortunately, **that task is left to the environment where the program is run** - generally, an operating system that decides the particular memory locations on runtime.
The actual address of a variable in memory cannot be known before runtime.

ampersand sign (&) Address-of-operator, and can be read simply as **"address of"**
asterisk operator(*) Dereference operator (*), and can be read as **"value pointed to by"**

Pointers are said to "point to" the variable whose address they store.
They can be used to access the variable they point to directly by preceding the pointer name with the **dereference operator (*)**. The operator itself can be read as **"value pointed to by".**

baz = * foo; (baz equal to value pointed to by foo)

baz = foo; (baz equal to foo)

complementary = birbirini tamamlayıcı
sort of = bir çeşit
infer = sonucuna varmak
intended = planlanan, istenilen
elaborate = özenle / ayrıntılı hazırlanmış

Dereferenced, the type needs to be known. The declaration of a pointer needs to include the data type the pointer is going to point to.

int * number;
char * character;
double* decimals;
This type is not the type of the pointer itself, but the type of the data the pointer points to.

The size in memory of a pointer depends on the platform where the program runs.

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Pointers And Arrays

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50
```

The main difference being that pointers can be assigned new addresses, while arrays cannot.

Arrays brackets “[ ]” are a dereferencing operator known as **offset operator.** They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced.

a [5] = 0 // a [offset of 5] = 0
*(a+5) = *; // pointed to by (a+5) = 0

Not only if **a** is a pointer, but also if **a** is an array. If an array, its name can be used just like a pointer to its first element.

int myvar;
int * myptr = &myvar;

int myvar;
int * myptr;
myptr = &myvar;

When pointers are initialized, what is initialized is the address they point to, never the value being pointed. Therefore, the code above shall not be confused with:
int myvar;

int * myptr;
***myptr** = &myvar;
Which anyway would not make such sense (and is not valid code).

int myvar;
int * foo = &myvar;
int * bar = foo;

Only **addition and subtraction** operations are allowed; the others make no sense in world of pointers.

```
1 char *mychar;
2 short *myshort;
3 long *mylong;
```

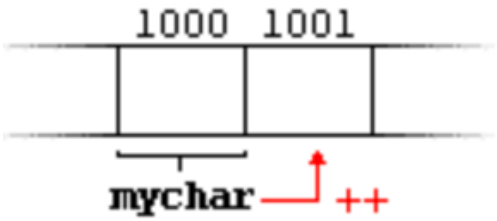
char, 1byte
short 2bytes,
long 4bytes,
Suppose that they point to the memory locations 1000, 2000, and 3000, respectively. Therefore, if we write:

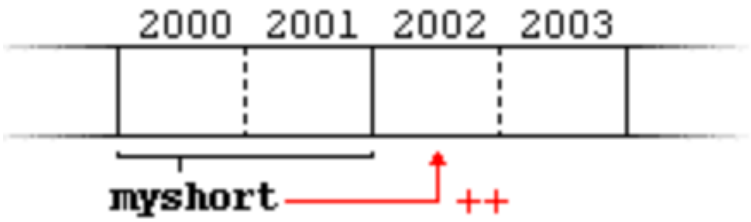
```
1 ++mychar;
2 ++myshort;
3 ++mylong;
```

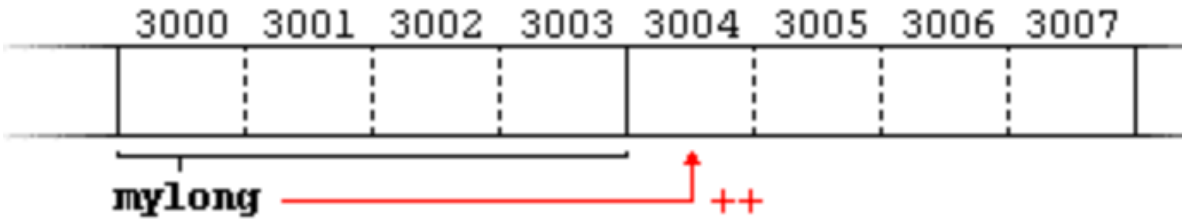
As a prefix (——x or ++x) the increment happens before the expression is evaluated, as a suffix (x—— or x++) the increment happens after the expression is evaluated.

When adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer.

This is applicable both when adding and subtracting any number to a pointer.







Recall that, postfix operators, such as increment and decrement, have higher precedence than prefix operators, such as the dereference operator (*).

```
*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

*p++ = *q++;
same as
*p = *q;
++p;
++q;
LIKE ALWAYS, PARANTHESES REDUCE CONFUSION BY ADDING LEGIBILITY TO EXPRESSIONS.

qualify = nitelemek

```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualified
```

p points to it in a const-qualified manner, it can read the value pointed, but it cannot modify it.

A pointer to **non-const** can be implicitly converted to a pointer to **const**. As a safety feature, pointers to **const** are not implicitly convertible to pointers to **non-const.**

One of the use cases of pointers to **const** elements is as function parameters: a function that takes a pointer to **non-const** as parameter can modify the value passed as argument, while a function that takes a pointer to **const** as parameter cannot.

```
// pointers as arguments:
#include <iostream>
using namespace std;

void increment_all (int* start, int* stop)
{
  int * current = start;
  while (current != stop) {
    ++(*current);  // increment value pointed
    ++current;     // increment pointer
  }
}

void print_all (const int* start, const int* stop)
{
  const int * current = start;
  while (current != stop) {
    cout << *current << '\n';
    ++current;     // increment pointer
  }
}

int main ()
{
  int numbers[] = {10,20,30};
  increment_all (numbers,numbers+3);
  print_all (numbers,numbers+3);
  return 0;
}
```

```
11
21
31
```

Note that, **print_all** uses pointers that point to constant elements. These pointers point to constant content they cannot modify, but they are not constant themselves. **The pointers can still be incremented or assigned different addresses, although they cannot modify the content they point to.**

Pointers can also be themselves **const.** This is specified by appending **const** to the pointed type(after the asterisk).

```
int x;
      int *       p1 = &x;  // non-const pointer to non-const int
const int *       p2 = &x;  // non-const pointer to const int
      int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

grasping = kavrama
merits = davanın esası, gerçek değer

The **const** qualifier can either precede or follow the pointed type, with the exact same meaning.

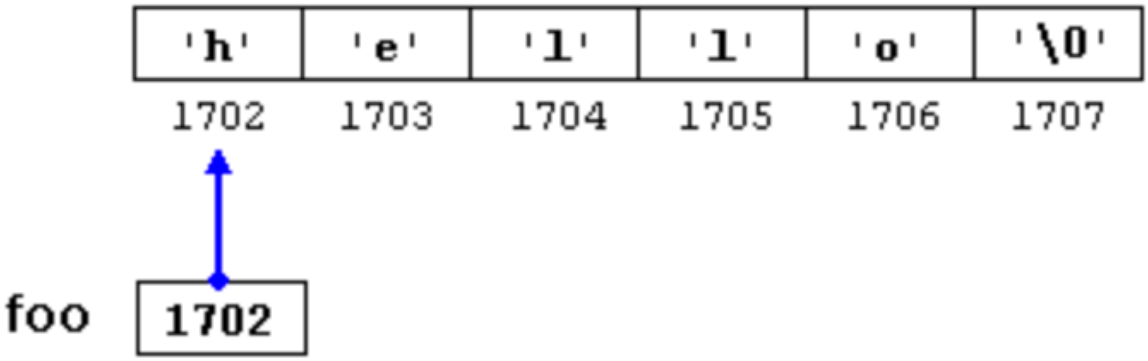
```
1 const int * p2a = &x;  //      non-const pointer to const int
2 int const * p2b = &x;  // also non-const pointer to const int
```

Pointers And String Literals

String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type **const** char (As literals, they can never be modified.)

const char * foo = “hello” ;

This declares an array with the literal representation for “hello”, and then a pointer to its first element is assigned to **foo.** If we imagine that “hello” is stored at the memory locations that start at address 1702, we can represent the previous declaration as:



***(foo+4)**

foo[4]

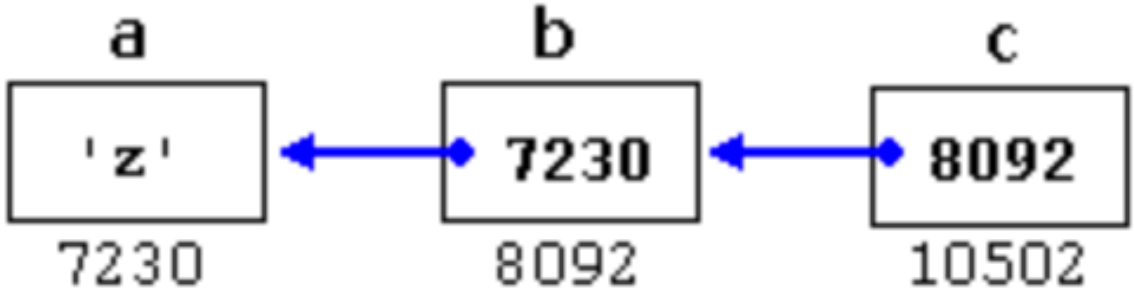
Both expressions have a value of ‘o’ (the fifth element of the array).

Pointers To Pointers

C++ allows the use of pointers that point to pointers.

```
1 char a;
2 char * b;
3 char ** c;
4 a = 'z';
5 b = &a;
6 c = &b;
```

Assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:



c is of type char** and a value of 8092
c is of type char and a value of 7230
**c is of type char and a value of ‘z’

Void Pointers (Special type of pointer.)

In C++, void represents the absence of type. Void pointers are pointers that point to a value that has no type
(and thus also an undetermined length and undetermined dereferencing properties**)**
(able to point to any data type, from an integer value or a float to a string of characters**).**

The data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

```
1  // increaser
2  #include <iostream>
3  using namespace std;
4
5  void increase (void* data, int psize)
6  {
7    if ( psize == sizeof(char) )
8    { char* pchar; pchar=(char*)data; ++(*pchar); }
9    else if (psize == sizeof(int) )
10   { int* pint; pint=(int*)data; ++(*pint); }
11 }
12
13 int main ()
14 {
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << '\n';
20   return 0;
21 }
```

```
y, 1603
```

sizeof is an operator integrated in the C++ language that returns the size in bytes of its argument. For non-dynamic data types, this value is a constant. Therefore, **sizeof (char)** is 1, because **char** has always a size of one byte.

Invalid Pointers And Null Pointers

out of bound = sınırların dışında, yasak, girilmez

```
int * p;                  // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Neither **p** nor **q** point to addresses known to contain a value, but none of the above statements causes an error. In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not.

Sometimes, a pointer really needs to **explicitly point to nowhere**, and not just an invalid address. There exists a special value that any pointer type can take: **the null pointer value.** This value can be expressed in C++ in two ways: either with an integer value of zero, or with the **nullptr** keyword:

int * p = 0;
int * q = **nullptr;**
int * r = **NULL;** (in several headers of stdlib)

DON'T CONFUSE **null pointers**(is a value that any pointer can take to represent that it is pointing to "nowhere") with **void pointers**(is a type of pointer that can point to somewhere without a specific type.)**!** One refers to the value stored in the pointer, and the other to the type of data it points to.

Pointers To Functions

The typical use of this is for passing a function as an argument to another function.

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  int (*minus)(int,int) = subtraction;

  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

8

minus, is a pointer to a function that has two parameters of type **int.** It is directly initialized to point to the function **subtraction.**

int (* minus) (int, int) = subtraction;