

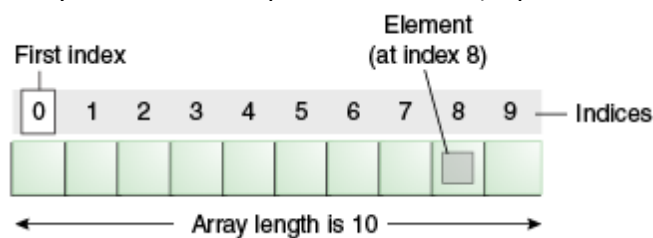
Arrays in C++	C++ Standard Template Library array container	Character Sequences in C++	Pointer in C++	Reference in C++	Difference between Pointers and Reference
---------------	---	----------------------------	----------------	------------------	---

## Compound data types

### Arrays

An array is a series of elements of the same type placed in **contiguous memory locations** that can be individually referenced by adding an index to a unique identifier.

In C++ elements are numbered from **0**. Typical declaration : **type name [elements];** where type is a valid type (like int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies how many of these elements the array has to contain.



### Initializing arrays

- Elements of a **regular** array will **not be initialized** to **any value by default**, so their content will be **undetermined**.
- The elements of **global and static arrays**, on the other hand, are **automatically initialized with their default values**, which for all fundamental types this means they are **filled with zeros**.

In both cases, we can assign initial values as shown below (enclosing the values in braces { } ) :

```
int students [5] = { 16, 2, 77, 40, 12071 };
```

```
int students [] = { 16, 2, 77, 40, 12071 };
```

For empty [ ], the compiler will assume a size for from the number of values included between braces { }:

```
float v[] = {1.0, 2.0, 3.0}, w[] = {7.0, 8.0, 9.0};
```

The list initialization in **C++11** cannot be narrowed any further. This will rarely make a difference in practice. For instance, the following:

```
int v[] = {1.0, 2.0, 3.0}; // Error in C++11: narrowing
```

It was legal in **C++03** but not in C++11 since the conversion from a floating-point to int potentially loses precision.

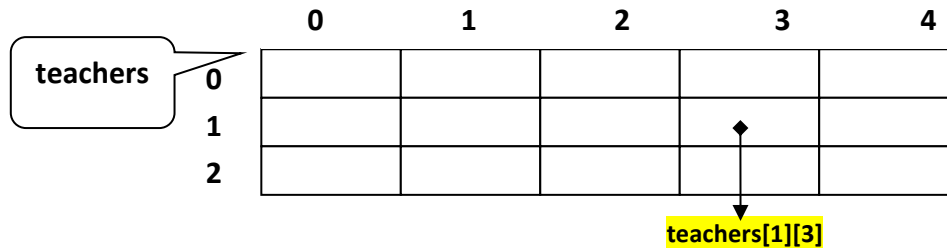
### Accessing the values of an array

The format for accessing the element is as simple as: **name[index]**, example : to store the value 75 in the third element of **students**, we could write `students[2] = 75;` Accessing out-of-range elements do

not cause compilation errors but can cause runtime errors.

## Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays".



`teachers` represents a bidimensional array of 3 per 5 elements of type `int`. The way to declare this array in C++ would be: `int teachers [3][5];`

for example, to reference the second element vertically and fourth horizontally in an expression would be: `teachers[1][3]`

## Arrays as parameters

- A complete block of memory like array cannot be passed **by value** as a parameter to a function
- Only allowed to **pass its address**. For accepting arrays as parameters in a function we need to use following syntax `void procedure (int arg[])`

accepts a parameter of type "array of int" called `arg`. In order to pass to this function an array declared as: `int myarray [40];` To call using above array as parameter : `procedure (myarray);`

```
// arrays as parameters
#include <iostream>
using namespace std;
void printarray (int arg[], int length) {
    for (int n=0; n<length; n++){
        cout << arg[n] << " ";
        cout << "\n";
    }
}
int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    return 0;
}
5 10 15
```

As you can see, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

2 4 6 8 10

For a multidimensional array parameter we can use : `base_type[][depth][depth]`  
for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets [] are left blank while the following ones are not. This is so because the compiler must be able to determine within the function which is the depth of each additional dimension.

## C++ Standard Template Library

**array** container in STL provides us the implementation of static array. We can use various method of this array container to manipulate fixed size array.

SYNTAX of array container : `array <object_type, array_size> array_name;`

The above code creates an empty array of **object\_type** with maximum size of **array\_size**. However, if you want to create an array with elements in it, you can do so by simply using the = operator as shown below. Let's examine some important member functions of array template.

**at()** : This method returns value in the array at the given range. If the given range is greater than the array size, **out\_of\_range** exception is thrown. Here is a code snippet explaining the use of it.

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

int main ()
{
    array<int,10> array1 = {1,2,3,4,5,6,7,8,9};

    cout << array1.at(2) // prints 3
    cout << array1.at(4) // prints 5
}
```

**fill()** : This method assigns the given value to every element of the array, example :

```
#include <array>
int main()
{
    array<int,8> myarray;
    myarray.fill(1);
}
This will fill the array myarray with value as 1,
at all of its 8 available positions.
```

### [ ] Operator

The use of operator [ ] is same as it was for normal arrays. It returns the value at the given position in the array. Example : In the above code, statement `cout << array1[5];` would print 6 on console as 6 has index 5 in array1.

**front()** : This method returns the **first** element in the array.

**back()** : This method returns the **last** element in the array. If the array is not completely filled, `back()` will return the rightmost element in the array.

**swap()** : This method swaps the content of two arrays of same type and same size. It swaps index wise, thus element of index i of first array will be swapped with the element of index i of the second array.  
#include <array>

```
int main()
{
    array<int,8> a = {1,2,3,4,5,6,7,8};
    array<int,8> b = {8,7,6,5,4,3,2,1};
```

a.**swap**(b) *// swaps array a and b*

```
cout << "a is : ";
for(int i=0; i < 8; i++) { cout << a[i] << " "; }
cout << endl;
cout << "b is : ";
for(int i=0; i < 8; i++) { cout << a[i] << " "; }
/* output will be
a is : 8 7 6 5 4 3 2 1
b is : 1 2 3 4 5 6 7 8 */
}
```

**size()** : this member function can be used to retrieve the length of the array:

```
array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };
std::cout << "length: " << myarray.size();
```

This prints: length: 5

## Sorting array

You can sort std::array using std::sort, which lives in the **algorithm** header.

```
#include <iostream>
#include <array>
#include <algorithm> // for std::sort
int main()
{
    std::array<int, 5> myarray { 7, 3, 1, 9, 5 };
    std::sort(myarray.begin(), myarray.end());
}
```

The sort() function in the algorithm header can be a very useful tool to both new and experienced programmers. It's use is to sort containers like arrays and vectors.

## Sort Syntax in **C++11** compilers

Example 1 : std::sort(myvector.begin(), myvector.end())

Example 2 : std::sort(myvector.begin(), myvector.end(), **myCompFunction**)

An **optional** overloaded compare function.

**Parameter 1 myvector.begin()** : You will be putting a iterator(Pointer) to the first element in the range that you want to sort. The sort will include the element that the iterator points to.

**Parameter 2 myvector.end()** : A iterator to the last element. One very important difference is that the

search won't include the element that this iterator points to. It is [First,Last) meaning it includes the first parameter in the sort but it doesn't include the second parameter in the sort.

**Parameter 3 myCompFunction() Optional** : Used to define how you do the search. For example if you have a struct that has 3 different variables in it, how does the function know which one to sort? Or how does it know how it should sort it?

### Array sorting in descending order using greater

sort() takes a third parameter that is used to specify the order in which elements are to be sorted. We can pass "**greater<>()**" function to sort in increasing order. This function does comparison in a way that puts greater element before.

using namespace std;

```
int main()
{
    int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    int n = sizeof(arr)/sizeof(arr[0]);

    sort(arr, arr+n, greater<int>());

    cout << "Array after sorting : \n";
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    return 0;
}
```

**greater<>()** – it is provided by API

You can also develop your function for comparison as per requirement.

**sizeof(array)** returns the **number of bytes** the array occupies. Since each element can take more than 1 byte of space, you have to divide the result with the size of one element (**sizeof(array[0])**). This gives you number of elements in the array.

## Character Sequences in C++

The C++ Standard Library implements a powerful **string** class. As the strings are in fact sequences of characters, we can represent them also as plain arrays of char elements like in C.

For example, the following array: **char jenny [20];** is an array that can store up to 20 elements of type char. But we can also store shorter sequences. A **special character is used to signal the end of the valid sequence**: the **null character**, whose literal constant can be written as **'\0'**

Our array of 20 elements of type char, called jenny, can be represented storing the characters sequences "Hello" and "Merry Christmas" as:

H	e	l	l	o	\0														
M	e	r	r	y			C	h	r	i	s	t	m	s	\0				

The panels in gray color represent char elements with undetermined values.

## Initialization of null-terminated character sequences

Initialize an array of characters with some predetermined sequence of characters :

```
char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Double quoted strings (") are literal constants whose type is in fact a null-terminated array of characters. So string literals enclosed between double quotes always have a null character ('\0') automatically appended at the end. So following commands have the same effect :

```
char myword [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

```
char myword [] = "Hello";
```

In 2nd case when using double quotes (") null character is appended automatically.

Please **notice** that we are initializing an array of characters when it is being declared, and not about assigning values to them once they have already been declared. Assuming **mystext** is a char[] variable, expressions within following source codes would not be valid :

```
mystext = "Hello";    mystext[] = "Hello";  
mystext = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

## Using null-terminated sequences of characters

Null-terminated sequences of characters are the natural way of treating strings in C++, so they can be used as such in many procedures. For example, cin and cout support null-terminated sequences as valid containers for sequences of characters, so they can be used directly to extract strings of characters from cin or to insert them into cout. For example:

```
// null-terminated sequences of characters  
#include <iostream>  
using namespace std;  
int main ()  
{  
    char question[] = "Please, enter your first name: ";  
    char greeting[] = "Hello, ";    char yourname [80];  
    cout << question;    cin >> yourname;  
    cout << greeting << yourname << "!";  
    return 0;  
}
```

```
Please, enter your first name: John  
Hello, John!
```

The first two arrays were initialized with string literal constants, while the third one was left uninitialized.

Sequences of characters stored in char arrays can easily be converted into string objects just by using the assignment operator:

```
string mystring;  
char myntcs[]="some text";    mystring = myntcs;
```

## Strings in C++

Here we demonstrate some useful member functions of C++ string class.

```
#include <iostream>  
#include <string>  
using namespace std;  
int main() {  
    string f(" Leading and trailing blanks ");  
    cout << "String f : " << f << endl;  
    cout << "String length : " << f.length() << endl;  
    cout << "String f : " << f.append("ZZZ") << endl;  
    cout << "String length : " << f.length() << endl;  
    cout << "substr(1,9) : " << f.substr(1,9) << endl;  
    cout << "assign(\"xyz\",0,3) : " << f.assign("xyz",0,3) << endl;  
    cout << "insert(1,\"abc\") : " << f.insert(1,"abc ") << endl;  
  
    string g("abc abc abd abc");  
    cout << "String g : " << g << endl;  
    cout << "replace(12,1,\"xyz\") : " << g.replace(12,1,"xyz") << endl;  
    cout << "replace(4,3,\"xyz\",2) : " << g.replace(4,3,"xyz",2) << endl;  
    cout << "replace(4,3,\"ijk\",1) : " << g.replace(4,3,"ijk",1) << endl;  
    cout << "find(\"abd\",1) : " << g.find("abd",1) << endl;  
    cout << "find(\"xyzb\") : " << g.find("xyzb") << endl;  
    return 0;  
}
```

The produces the following output:

```
String f : Leading and trailing blanks  
String length : 27  
String f : Leading and trailing blanks  
ZZZ  
String length : 30  
substr(1,9) : eading an  
assign("xyz",0,3) : xyz  
insert(1,"abc") : xabc yz  
String g : abc abc abd abc  
replace(12,1,"xyz") : abc abc abd  
xyzbc
```

```
replace(4,3,"xyz",2): abc xy  
abd xyzbc  
replace(4,3,"ijk",1): abc iabd  
xyzbc  
find("abd",1) : 5  
find("xyzb") : 9
```

### Note:

- **assign** : Assigns a new value to the string, replacing its current contents.  
1<sup>st</sup> Parameter - Position of the first character in str (here "xyz") that is copied to the object (here f) as a substring. 2nd Parameter - Length of the substring to be copied.
- In find starting position of search is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
- Return of Find : Position of the first character of the found substring or **npos** if no such substring is found.
- **length** or **size** function returns the number of actual bytes that conform the contents of the string, which is not necessarily equal to its storage capacity.

Note that `string` objects handle bytes without knowledge of the encoding that may eventually be used to encode the characters it contains. Therefore, the value returned may not correspond to the actual number of encoded characters in sequences of multi-byte or variable-length characters (such as UTF-8).

Both `string::size` and `string::length` are synonyms and return the same value.

A further important member function is `string::c_str()` which returns a pointer to a character array with the same characters as the string that generates the call. This method is needed if you want to use some "older" library routines such as `scanf`.

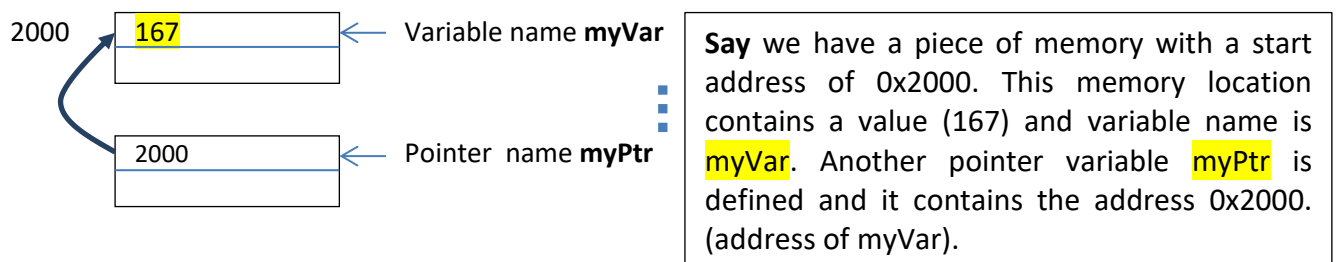
## Pointers in C++

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address) automatically by the operating system during runtime.

However, in some cases we may be interested in knowing the address where our variable is being stored during runtime.

A pointer is a variable that points to another variable. A pointer holds the memory address of that variable.



## Initialize a pointer

To declare a pointer you have to put an `*` in front of its name. A pointer can be **typed** or **un-typed**. (A typed pointer points to a particular variable type such as an integer. An un-typed pointer points to any data type).

```
void main() {  
    int * ptr_A;      /* A typed pointer */  
    void * ptr_B;     /* A untyped pointer */  
}
```



Before you can use a pointer in for instance a `cout` statement, you have to initialize the pointer. The following example will **not** initialize the pointer:

```
#include<iostream>
using namespace std;
void main()
{
    int *ptr_p;
    cout << *ptr_p;
}
```

- Most compilers will give a warning when you try to compile the example, others will not.
- Here we print the value that `ptr_p` points to. However, we did not initialize the pointer.

So the pointer contains a random address or 0. The result of this program is a segmentation fault (meaning you have used a pointer that points to an invalid address).

```
int * number;    char *
```

- All of the pointers declared above will occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is going to run).
- Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type.
- Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

## Reference and dereference operators in C++

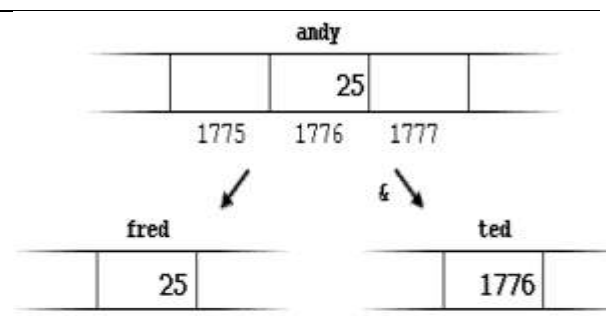
The address that locates a variable within memory is called a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (`&`), known as reference operator. If the reference operator is used you will get the "address of" a variable.

For example: `ted = &andy;` This would assign to `ted` ( a pointer) the address of variable `andy`.

Assume that `andy` is placed during runtime in the memory address 1776.

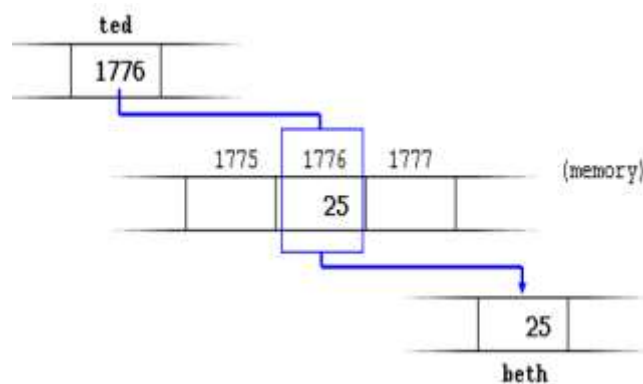
Consider the following code fragment:  
`andy = 25; fred = andy; ted = &andy;`

The variable that stores the reference to another variable (like `ted` in the previous example) is what we call a **pointer**. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming.



Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an **asterisk (\*)**, which acts as **dereference** operator

and that can be literally translated to "value pointed by". Therefore, following with the values of the previous example,  
if we write: `beth = *ted;`



Notice the difference of including or not including the dereference operator.

```
beth = ted; // beth equal to ted ( 1776 )
beth = *ted; // beth equal to value pointed
              by ted ( 25 )
```

Notice the difference between the reference and dereference operators:

- `&` is the reference operator and can be read as "address of"
- `*` is the dereference operator and can be read as "value pointed by"

**Note:** The asterisk (\*) sign in the declaration of the pointer does not mean "value pointed by", it only means that it is a pointer (it is part of its type compound specifier). It should not be confused with the dereference operator. They are simply two different things represented with the same sign.

## Using pointers to pass values

```
#include<iostream>
using namespace std;
int main ()
{
    int x;          int * ptr_p;
    ptr_p = &x;     *ptr_p = 5;
    cout << x;     return 0; }
}
```

- Although no value is store in x directly it will print the content of x as 5.
- Here we store the address of x into the pointer `ptr_p`. Then we say we want the value 5 stored at the address where the pointer is pointing to (in this case x).

In this way we can use pointers to pass values to other variables.

## Pointers and arrays

The identifier of an array is equivalent to the address of its first element. So pointer and array identifier are similar concept. For example, two declare : `int numbers [20]; int * p;`

The following assignment operation would be valid: `p = numbers;`

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a **constant pointer**.

Therefore, the following allocation would not be valid: `numbers = p;` because we cannot assign value to a **constant pointer**.

The C++ language allows **pointer addition and subtraction**. Let's take a look at this example:

```
char num[10];  
char *ptr_toarray = &num[0];
```

The pointer `*ptr_toarray` must point at the first element of the array (`num[0]`).

Now we could do the following (note the round brackets):

```
char num[10];  
char *ptr_toarray = &num[0];  
*(ptr_toarray + 1);
```

This is the same as `num[2]`. Or we can do this:

```
char num[10];  
char *ptr_toarray = &num[0];  
ptr_toarray++;
```

So now the pointer is pointing at the second element: `num[1]`.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

```
#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; }
```

The bracket sign operators `[]` of an array are also a **dereference** 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. For example:

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

Output : 10, 20, 30, 40, 50,

## Pointer arithmetic

- Only **addition** and **subtraction** operations are allowed.
- But both addition and subtraction have a different behavior with pointers **according to the size of the data type** to which they point.
- Different fundamental data types occupy more or less space than others in the memory. For example, for a given compiler for a specific machine, **char takes 1 byte, short takes 2 bytes and long takes 4.**

Suppose that we define three pointers in this compiler:

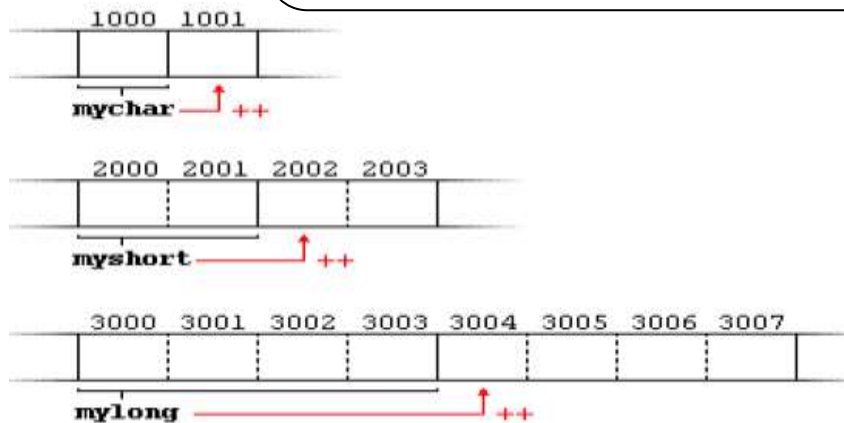
```
char *mychar;
```

```
short *myshort;  
long *mylong;
```

Say they point to memory locations 1000, 2000 and 3000 respectively.

So if we write:  
mychar++;  
myshort++;  
mylong++;

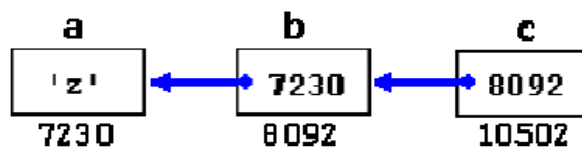
**mychar** would contain the value 1001, **myshort** would contain the value 2002, and **mylong** would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type, and therefore the **size in bytes of the type** pointed is added to the pointer.



C++ allows the use of **pointers that point to pointers**. In order to do that, we only need to add an asterisk (\*) for each level of reference in their declarations:

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

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



See the variable **c**, which can be used in three different levels of indirection, each one of them would correspond to a different value:

- **c** has type **char\*\*** and a value of 8092
- **\*c** has type **char\*** and a value of 7230
- **\*\*c** has type **char** and a value of 'z'

## Operator precedence

Both the increase (++) and decrease (--) operators have **greater operator precedence** than the **dereference** operator (\*), but both have a special behavior when used as suffix (the expression is

evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion: `*p++`

Let's take a look at an example:

<code>*ptr_p++;</code>	As ++ has greater precedence than *, this expression is equivalent to <code>*(ptr_p++)</code> . So it increases the value of ptr_p (so it now points to the <b>next element</b> ). You might think this but because ++ is used as post-fix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).
<code>(*ptr_p)++</code>	Here the value pointed by ptr_p is increased by one. The value of the pointer itself (p_ptr) is not modified. So the only thing that is modified is what it is being pointed to by the pointer.
<code>*ptr_p++ = *ptr_a++;</code>	The increase operator (++) has a higher precedence than *. Because we use the increase operators as post-fix (instead of prefix) first the value of *ptr_a is assigned to *ptr_p. After this is done both are increased by one.  It would be roughly equivalent to: <code>*p = *q; ++p; ++q;</code>

So always remember the operator precedence. Also use parentheses () in order to avoid unexpected results and confusion when reading the code.

## void pointers

The void type of pointer is a special type of pointer. In C++, void represents the **absence of type**, so void pointers are pointers that **point to a value that has no type**.

This allows void pointers to point to **any data type**, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the **data pointed by them cannot be directly dereferenced** (which is logical, since we have no type to dereference to), and for that reason we will always **have to cast the address** in the void pointer to some other pointer type that points to a concrete data type before dereferencing it. One of its uses may be to pass generic parameters to a function:

```
#include <iostream>
using namespace std;
void increase(void* data, int psize)
{
    if ( psize == sizeof(char) )
        { char* pchar; pchar=(char*)data; ++(*pchar); }
    else if (psize == sizeof(int) )
        { int* pint; pint=(int*)data; ++(*pint); }
}
int main ()
{
```

Output : y, 1603

**sizeof** is an operator integrated in the C++ language that returns the size in bytes of its parameter. For nondynamic data types this value is a constant. Therefore, for example, sizeof(char) is 1, because char type is one byte long.

```

char a = 'x';           int b = 1602;
increase (&a,sizeof(a)); increase (&b,sizeof(b));
cout << a << ", " << b << endl;
return 0;
}

```

## Null pointer

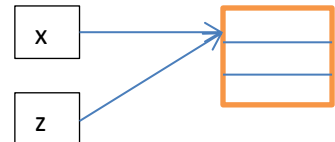
A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to **any valid reference** or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;           p = 0; // p has a null pointer value
```

**Do not confuse null pointers with void pointers.** A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

## Reference in C++

- Reference variable = alias for another variable or object
  - Contains the address of a variable (like a pointer)
  - No need to perform any **dereferencing** (unlike a pointer)
  - Must be initialized when it is declared
- Once a reference is initialized with a variable, either the **variable name** or the **reference name** may be used to refer to the variable.



### Creating References in C++:

```

int x = 5;
int &z = x;    // z is another name for x
int &y;        //Error: reference must be initialized
cout << x << endl;    -> prints 5
cout << z << endl;    -> prints 5
z = 9;        // same as x = 9;
cout << x << endl;    -> prints 9
cout << z << endl;    -> prints 9

```

The syntax is as follow:

```

type &newName = existingName; // or
type& newName = existingName; // or
type & newName = existingName;

```

Read the **&** as reference. Thus, above declaration as **"z is an integer reference initialized to x"**

```

void swap(int& i, int& j)
{
    int tmp = i;
    i = j;  j = tmp;
}

```

- Here i and j are aliases for main's x and y respectively. In other words, i is x — **not a pointer to x, nor a copy of x, but x itself**. Anything you **do to i gets done to x**, and vice versa.
- That's how you should think of references as a programmer.

<pre>int main() {     int x, y;     ...     swap(x,y);     ... }</pre>	<ul style="list-style-type: none"> <li>Underneath it all, a reference i to object x is typically the machine address of the object x. But when the programmer says <code>i++</code>, the compiler generates code that increments x.</li> </ul>
--	--

## C++ References vs Pointers:

References are often confused with pointers but there are some differences:

- You cannot have **NULL references**. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, **it cannot be changed to refer to another object**. Pointers can be pointed to another object at any time.
- A reference must **be initialized when it is created**. Pointers can **be initialized at any time**.
- Reference variables are primarily used as **function parameters**
- Advantages of using references:
  - you don't have to pass the **address** of a variable
  - you don't have to **dereference** the variable inside the **called function**
- There's no **pointer-arithmetic** for the reference.
- You can have a pointer to a pointer **but not a reference to a reference**.
- References are safer and easier to use:
  - Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist.
  - Easier to use: References don't need **dereferencing** operator to access the value. They can be used like normal variables. **'&' operator is needed only at the time of declaration**. Also, members of an object reference can be accessed with dot operator ('.'), unlike **pointers where arrow operator (->)** is needed to access members.

### Example :

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

int main ()
{
    // declare simple variables
    int i; double d;

    // declare reference variables
    int& r = i; double& s = d;    i = 5;
```

When the code is compiled together and executed, it produces the following result:

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

```

cout << "Value of i : " << i << endl;
cout << "Value of i reference : " << r << endl;

d = 11.7;
cout << "Value of d : " << d << endl;    cout << "Value of d reference : " << s << endl;
return 0;
}

```

**Reference** is **not** a **pointer**, they're different although they serve similar purpose.

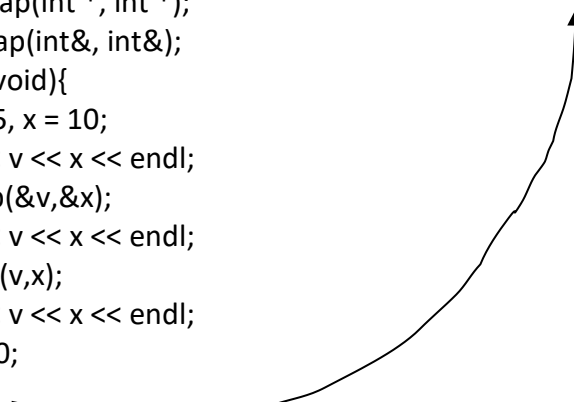
The reference argument (&) gets dereferenced automatically. You can think of a reference as an alias to another variable, i.e. the second variable having the same address. It doesn't contain address itself, it just *references* the same portion of memory as the variable it's initialized from. A reference **creates another name, an alias, for something that exists elsewhere**. That's it. There are no hidden pointers or addresses involved.

### Reference Variables Example

```

#include <iostream.h>
// Function prototypes as we define swap after main
// function where we are using swap (required in C++)
void p_swap(int *, int *);
void r_swap(int&, int&);
int main (void){
    int v = 5, x = 10;
    cout << v << x << endl;
    p_swap(&v,&x);
    cout << v << x << endl;
    r_swap(v,x);
    cout << v << x << endl;
    return 0;
}

```



```

void p_swap(int *a, int *b)
{
    int temp;
    temp = *a;           (2)
    *a = *b;             (3)
    *b = temp;
}

```

```

void r_swap(int &a, int &b)
{
    int temp;
    temp = a;           (2)
    a = b;              (3)
    b = temp;
}

```

### Passing parameters by references in C++

We have discussed how we implement **call by reference** concept using pointers. Here is another example of call by reference which makes use of C++ reference:

```

#include <iostream>

using namespace std;

// function declaration
void swap(int& x, int& y);

int main ()

```

So when reference is sent as parameter, variable does not need to be copied; so if it were of a large type such as `std::list<int>`, then passing it by reference would be much more efficient.



```

{
    // local variable declaration:
    int a = 100;    int b = 200;
    cout << "Before swap, value of a : " << a << endl;
    cout << "Before swap, value of b : " << b << endl;

    /* calling a function to swap the values.*/
    swap(a, b);

    cout << "After swap, value of a : " << a << endl;
    cout << "After swap, value of b : " << b << endl;
    return 0;
}
// function definition to swap the values.
void swap(int& x, int& y)
{
    int temp;
    temp = x; /* save the value at address x */
    x = y;    /* put y into x */
    y = temp; /* put x into y */
    return;
}

```

When the above code is compiled and executed, it produces the following result:

```

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100

```

## Returning values by reference in C++

A C++ program can be made easier to read and maintain by using references rather than pointers. A C++ function can return a reference in a similar way as it returns a pointer.

When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement. For example, consider this simple program:

```

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

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};
double& setValues( int i )
{
    return vals[i]; // return a reference to the ith element
}

// main function to call above defined function.
int main ()
{
    cout << "Value before change" << endl;

```

When the code is compiled and executed, it produces the following result:

```

Value before change
vals[0] = 10.1
vals[1] = 12.6
vals[2] = 33.1
vals[3] = 24.1
vals[4] = 50
Value after change
vals[0] = 10.1
vals[1] = 20.23
vals[2] = 33.1
vals[3] = 70.8
vals[4] = 50

```

```

for ( int i = 0; i < 5; i++ )
{
    cout << "vals[" << i << "] = ";  cout << vals[i] << endl;
}
setValues(1) = 20.23; // change 2nd element
setValues(3) = 70.8; // change 4th element
cout << "Value after change" << endl;
for ( int i = 0; i < 5; i++ )
{
    cout << "vals[" << i << "] = ";  cout << vals[i] << endl;
}
return 0;
}

```

When returning a reference, be careful that the **object being referred to does not go out of scope**. So it is not legal to return a **reference to local var**. But you can always return a reference on a **static variable**.

```

int& func() {
    int q;
    //! return q; // Compile time error
    static int x;
    return x; // Safe, x lives outside this scope }

```

## What happens if you return a reference?

The function call can appear **on the left hand side of an assignment operator**.

This ability may seem strange at first. For example, no one thinks the expression `f() = 7` makes sense. Yet, if `a` is an object of class `Array`, most people think that `a[i] = 7` makes sense even though `a[i]` is really just a function call behind the scene (it calls `Array::operator[](int)`, which is the subscript operator for class `Array`).

```

class Array {
public:
    int size() const;
    float& operator[] (int index);
    ...
};

```

```

int main()
{
    Array a;
    for (int i = 0; i < a.size(); ++i)
        a[i] = 7;  // This line invokes Array::operator[](int)
    ...
}

```

## Reference Types in Java

- All primitive types have well-defined standard sizes, so all primitive values can be stored in a fixed amount of memory. Java handles values of the primitive types directly, or by value.
- But classes and array types are composite types; **objects** and arrays contain other values, so they do not have a standard size. For this reason, Java does not manipulate objects and arrays directly.

- Java manipulates objects and arrays with their **references** (**memory address** at which the object or array is stored). Because Java handles objects and arrays by reference, classes and array types are known as reference types. In contrast,
- A reference to an object or an array is simply some fixed-size value (generally 4 byte integer) that refers to the object or array in some way. When you assign an object or array to a variable, you are actually setting the variable to hold a reference to that object or array. Similarly, when you pass an object or array to a method, what really happens is that the method is given a reference to the object or array through which it can manipulate the object or array.
- Like C++ , Java does not support the **& address-of operator or the \* and -> dereference operators**. In Java, primitive types are always handled exclusively by value, and objects and arrays are always handled exclusively by reference. Furthermore, unlike pointers in C and C++, references in Java are entirely opaque: they cannot be **converted to or from integers**, and they cannot be **incremented or decremented**.
- Java programs cannot manipulate references in any way. Despite this, there are significant differences between the behavior of primitive types and reference types in two important areas: the way values are copied and the way they are compared for equality.

## Copying Objects

Consider the following code that manipulates a primitive int value:

```
int x = 42;
int y = x;
```

Inside the Java VM, there are two independent copies of the 32-bit integer 42.

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

Here the variable q holds a copy of the reference held in the variable p. There is still only one copy of the Point object in the VM, but there are now two copies of the reference to that object.

Now if we execute following lines :

```
System.out.println(p.x); // Print out the X coordinate of p: 1.0
q.x = 13.0;              // Now change the X coordinate of q
System.out.println(p.x); // Print out p.x again; this time it is 13.0
```

Since the variables p and q hold references to the same object, either variable can be used to make changes to the object, and changes are visible through the other variable as well.

This behavior is not specific to objects; the same thing happens with arrays.

On the other hand, consider what happens if we modify the method so that the parameter is a reference type:

```
void changeReference(Point p) {
    while(p.x > 0)
        System.out.println(p.x--);
}
```

When this method is invoked, it is passed a private copy of a reference to a Point object and can use this reference to change the Point object.

Consider the following:

```
Point q = new Point(3.0, 4.5); // A point with an X coordinate of 3
changeReference(q);           // Prints 3,2,1 and modifies the Point
System.out.println(q.x);      // The X coordinate of q is now 0!
```

- Now both the variable q and the method parameter p hold references to the same object.
- The method can use its reference to change the contents of the object.
- However, it cannot **change the contents of the variable q**. In other words, it cannot change the fact that the variable q refers to that object.

## Comparing Objects

When working with **reference** types, there are **two kinds of equality**:

- equality of reference and
- equality of object.

The method for testing whether one object is equivalent to another is named **equals()**. To test two objects for equivalence, pass one of them to the equals() method of the other:

```
String letter = "o";
String s = "hello";           // These two String objects contain exactly the same text.
String t = "hell" + letter;
if (s.equals(t))              // And the equals() method tells us so.
    System.out.println("equivalent");
```

- All objects inherit an **equals()** method (from **Object**, but the default implementation simply uses **==** to test for equality of references, not equivalence of content.
- A class that wants to allow objects to be compared for equivalence can define its own version of the **equals()** method. Our Point class does not do this, but the String class does, as indicated by the code above.

**The null Reference** : The **default** value for all reference types is **null**. The null value is unique in that it can be assigned to a variable of any reference type whatsoever.

## Reference Type Conversions

Values of certain primitive types can be converted to values of other types. Widening conversions are performed automatically by the Java interpreter, as necessary. Narrowing conversions, however, can result in lost data, so the interpreter does not perform them **unless explicitly directed to do so with a cast**.

Java does not allow any kind of conversion **from primitive types to reference types or vice versa**. Java does allow widening and narrowing conversions among **certain reference types as discussed below**.

Every Java class *extends* some other class, known as its *superclass*. There is a special class named **Object** that serves as the **root of the class hierarchy in Java**.

With this simple understanding of the class hierarchy, we can return to the rules of reference type conversion:

- An object cannot be converted to an unrelated type. The Java compiler does not allow you to convert a **String to a Point**, for example, even if you use a cast operator.
- An object can be converted to the type of a **superclass**. This is a widening conversion, so **no cast** is required. For example, a **String** value can be assigned to a variable of type **Object** or passed to a method where an Object parameter is expected.
- An object can be converted to the type of a **subclass**, but this is a narrowing conversion and requires a **cast**. The Java compiler provisionally allows this kind of conversion, but the Java interpreter checks at runtime to make sure it is valid. If it is not, the interpreter throws a **ClassCastException**. For example, if we assign a String object to a variable of type Object, we can later cast the value of that variable back to type String:

```
Object o = "string"; // Widening conversion from String to Object
// Later in the program...
String s = (String) o; // Narrowing conversion from Object to String
```

- All array types are distinct, so an array of one type cannot be converted to an array of another type, even if the individual elements could be converted. For example, although a byte can be widened to an int, a byte[] cannot be converted to an int[], even with an explicit cast.
- Arrays do not have a type hierarchy, but all arrays are considered instances of Object, so any array can be converted to an Object value through a widening conversion. A narrowing conversion with a cast can convert such an object value back to an array. For example:

```
Object o = new int[] {1,2,3}; // Widening conversion from array to Object
// Later in the program...
int[] a = (int[]) o; // Narrowing conversion back to array type
```