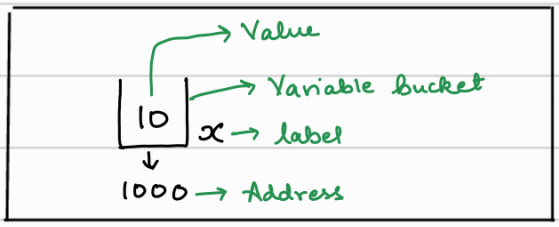
* Allocation of memory & storage of values :-

```
int x = 10;
```



→ The C++ compiler allocates the addresses on its own when the variables are created during compilation.

* Address-of operator (&) :-

→ Used to access the address of the memory bucket.

```
int x = 10;
cout << &x;  ⟶ 0x7ffd43d201b4   (Prints the memory address at which the variable x is stored)
```

Hex value of memory

→ It is used to retrieve the address of any variable.

→ The address is always in hexadecimal value.

* Pointers :-

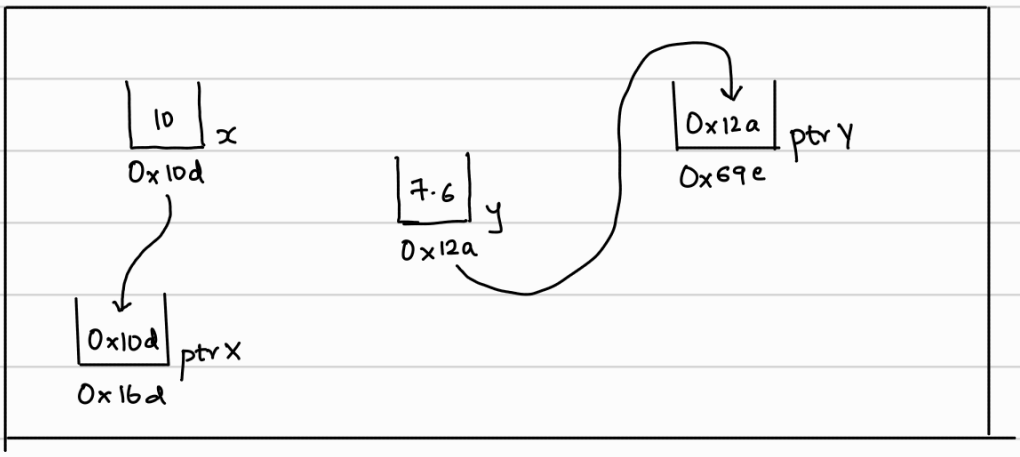
→ Pointers are variables that store addresses.

```
datatype *pointerName;
```

→ If we want to store address of an int variable, we need to create an int pointer & so on for other data types.

```
int x = 10;
int *ptrX = &x;   → Pointer that stores the address of int variables
float y = 7.6;
float *ptrY = &y;   → Pointer that stores the address of float variable.

cout << &x;
cout << ptrX;
```

} Both will print the same value ie address of x.



* Valid pointer declaration :-

```
int* ptr = &x;
int * ptr = &x;
int *ptr = &x;
```

All of them are valid. You can use any one of them. But make sure the use consistent throughout the codebase.

`int * ptr;` → Points to some random memory location.

`ptr = &x;` → We can re-assign it as well.

* Dereference operator :-

→ Extracting the value stored at particular address. is called dereferencing.

```
int x = 10;
int* ptr = &x;
cout << ptr;
```
→ Prints address of x

```
cout << *ptr;
```
→ 10, as we use * which is the de-reference operator that print the value stored at a particular address.

```
x = 69;
cout << *ptr;
```
→ 69, this is because ptr is pointing to the same bucket where x is stored.

→ It can be more clear with the following analogy :-

→ Consider two people A & B.

→ A has put an apple at a particular place & gave B the location / address of that place.

→ Later, B goes to that address & finds out that an apple has been placed over there.

→ Now, A replaced the apple with banana. & asked B to revisit the same address.

→ When B reached the address, banana was found.

→ We can use the de-referencing mechanism to update the original value as well.

```
int x = 10;
int* ptr = &x;
*ptr = 69;
cout << x;  → 69
```

→ If you have already created a pointer ptr, doing *ptr will be de-referencing it.

```
int temp = *ptr;
cout << temp;  → 69.
```

* Add two numbers using pointers :-

```
int x = 10;
int y = 59;

int* ptrx = &x;
int* ptry = &y;

int result;
int* ptr_result = &result;
*ptr_result = *ptrx + *ptry;
```

```
cout << result ;   ⟶ 69
```

* Note :-

```
int x = 10;
int* ptr = &x;
ptr = 5;
```
→ We can't do this as ptr is only supposed to store addresses.

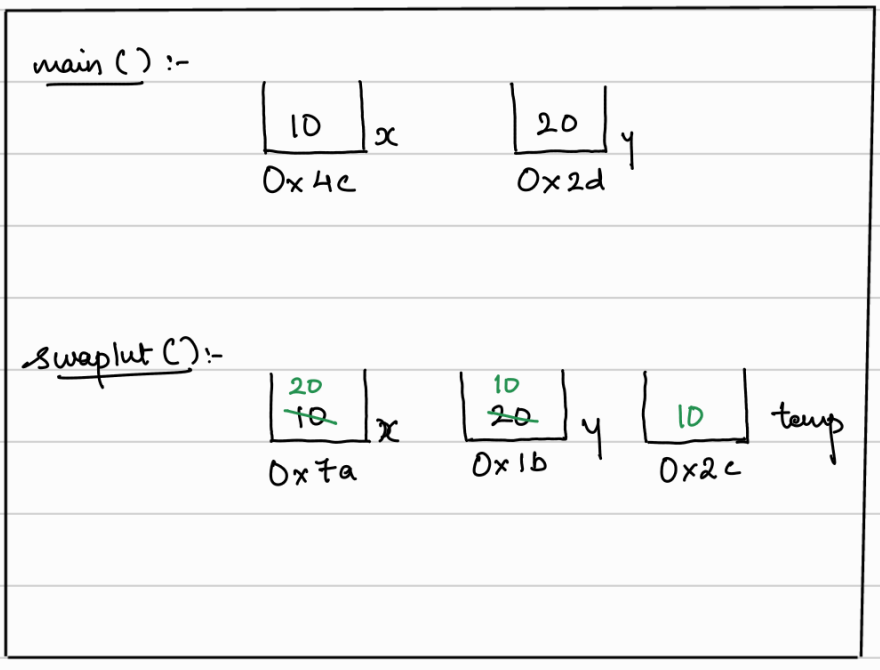
```
int y = 29;
*ptr = &y;
```
→ We can't do this as well because *ptr means de-referencing & it will be pointing to an int value. We can't store an address at that point.

```
cout << &ptr;
```
→ Prints the address of the pointer & not the address that the pointer is storing.

* Call - by - reference using pointers :-

→ Consider the below code for swapping two integers.

```
void swapInt(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
int main() {
    int x = 10;
    int y = 20;
    swapInt(x, y);
    cout << x;   ⟶ 10
    cout << y;   ⟶ 20
}
```

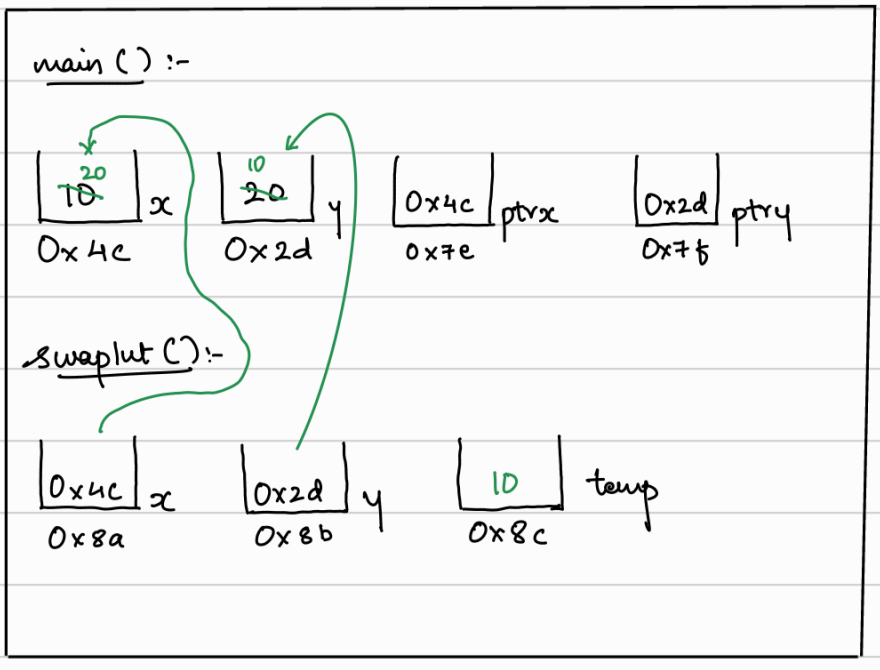


→ We see that the original x & y are unchanged.

→ It is because in swaplut(), a new local copy of x & y are created & we are swapping the local copies rather than the original ones.

→ To deal with this, we need to make use of pointer variables.

```
void swaplut (int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main () {
    int x = 10;
    int y = 20;
    int *ptrx = &x;
    int *ptry = &y;
    swaplut (ptrx, ptry);
    cout << x;   → 20
    cout << y;   → 10
}
```



→ By using pointers we were able to swap both the integers.

* Example 2 :- For a given string, find the index of first & last occurrence of a char in the string.

Input :- str = "aaabac"

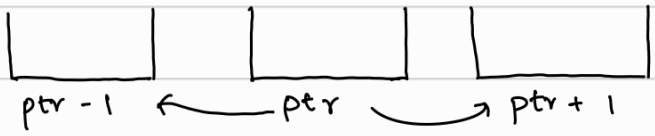
ch = 'a'

Output :- 0 4

```
void firstAndLastIndex (string , char , int* , int* );

int main() {
  string str = "aaabac";
  char ch = 'a';
  int first = -1;
  int last = -1;

  int *pf = &first;
  int *pl = &last;

  firstAndLastIndex (str, ch , pf , pl);
  cout << first << "  " << last << endl;   → 0 4
}

void firstAndLastIndex (string str , char ch , int *pf , int *pl ) {
   for (int i=0 ;  i < str.size() ;  i++) {
      if (str[i] == ch) {
         *pf = i;
         break;
      }
   }
   for (int i = str.size()-1 ;  str >=0  ;  i--) {
      if (str[i] == ch) {
         *pf= i;
         break;
      }
   }
}
```

* Pointer arithmetic :-

→ We can use increment & decrement operations on a pointer.



→ This way we can point to the next address or previous address.

→ The next & previous addresses will depend on the size of the data type the pointer is pointing to.

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

int main()
{
    int x = 10;
    int *ptrx = &x;

    cout << "Size of x: " << sizeof(x) << endl;
    cout << "ptrx address: " << ptrx << endl;
    cout << "ptrx + 1 address: " << ptrx + 1 << endl;
    cout << "ptrx + 2 address: " << ptrx + 2 << endl;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

→ cpp_workspace ./a.out
Size of x: 4
ptrx address: 0x7ffda2e30ecc
ptrx + 1 address: 0x7ffda2e30ed0
ptrx + 2 address: 0x7ffda2e30ed4
```

Here size of x is 4 bytes as it is an int.
The pointer to x stores the address ecc.
When we do ptrx + 1 →. it implies

$$ptrx + (1 \times sizeof(x)) = ptrx + 4$$
$$= ed0$$

c → d → e → f → 0 (↳ 1 carry so d)

$$ptrx + 2 = ptrx + (2 \times sizeof(x))$$
$$= ptrx + (2 \times 4)$$
$$= ptrx + 8$$
$$= ecc + 8$$
$$= ed4$$

c → d → e → f → 0 → 1 → 2 → 3 → 4

Hex values :- 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → A → B → C → D → E → F (Base 16)

$$ptr + p = ptr + (p + sizeof(datatype\ of\ ptr))$$

→ Same works for decrement as well.

* Arrays & Pointers :-

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = &arr[0]; → Storing the address of the first element
cout << *ptr; → 1
cout << *ptr++; → First increment, then de-referencing → 1
cout << *ptr; → 2
```

```
*ptr = &arr[0];
cout << *(ptr+1); → 2
(*ptr)++;
cout << arr[0]; → 3
```

```
int arr2[2] = {7, 5};
cout << *++ptr; → 5, first increment the pointer then de-reference.
                  Read right to left.
ptr = &arr[0];
cout << ++*ptr; → 8
```

* Cases to remember :-

```
*ptr++
(*ptr)++
*++ptr
++*ptr
```

* Arrays as pointers :-

→ The name of the array acts like a pointer that stores the address of the first element.

```
4   int main()
5   {
6       int arr[2] = {2, 5};
7       int *ptr = &arr[0];
8
9       cout << arr << endl;
10      cout << ptr << endl;
11  }
12
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE

➜ cpp_workspace ./a.out
0x7ffc02e88eb0
0x7ffc02e88eb0
```

`cout << *arr;` → 2

`cout << *(arr+1);` → 5

∴ `*(arr+0)` → 0^{th} index value

`*(arr+1)` → 1^{st} index value

main.cpp > ...

```
1   #include <iostream>
2   using namespace std;
3
4   void process(int *arr, int n)
5   {
6       for (int i = 0; i < n; i++)
7       {
8           cout << *(arr + i) << " ";
9       }
10      cout << endl;
11  }
12
13  int main()
14  {
15      int arr[5] = {1, 2, 3, 4, 5};
16      process(arr, 5);
17  }
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

➜ cpp_workspace g++ main.cpp
➜ cpp_workspace ./a.out
1 2 3 4 5
```

→ So, if we pass arrays like this, it is passed by reference.

```
int arr[3] = {1, 2, 3};
int (*p)[3] = &arr;
```
→ Now p is pointing to the entire array instead of only the first element.

* Types of pointers :-

1) **Wild pointer** :- A pointer that is only declared but not initialized is called as a wild pointer.

```
int *ptr;
```
→ Pointer pointing at some random address.

→ In some cases, it might also throw segmentation fault.

2) **Null pointer** :- If we want to have a pointer variable, which is just declared but will get address later to store, we can use null pointer.

```
int *ptr = NULL;   OR   int *ptr = nullptr;   OR   int *ptr = 0   OR   int *ptr = '\0';
cout << ptr;
```
→ 0 or 0x0

```
cout << *ptr;
```
→ Segmentation fault

→ We cannot de-reference a null pointer.

→ 0 is a special reserved memory address.

3) **Dangling pointer** :-

```
int x = 22;
int *ptr = &x;
```
→ Assume that x was removed from the memory.

→ The pointer still points to x. which is an invalid memory location.

→ So, a dangling pointer is a pointer that points to an invalid memory location.

4) Void pointer :- It is also called as generic pointer as the data type is not specified. It can point to any data type value.

```
int x = 69;
void *ptr = &x;
float f = 6.9;
ptr = &f;
```

→ Such pointers cannot be de-referenced directly.
→ But we can use type casting to access its value.

```
int *intPtr = (int *) ptr;
```