FAST

الذی علم بالقلم۔ علم الانسان ما لم یعلم۔

National University of Computer and Emerging Sciences

DEPARMENT OF COMPUTER SCIENCE

Object Oriented Programming-Lab

Lab-08

Engr. Khuram Shahzad, FAST NUCES Peshawar

OOP (LAB-08)
POINTERS IN C++

Engr. Khuram Shahzad
(Instructor)

LAB-08 CONTENTS

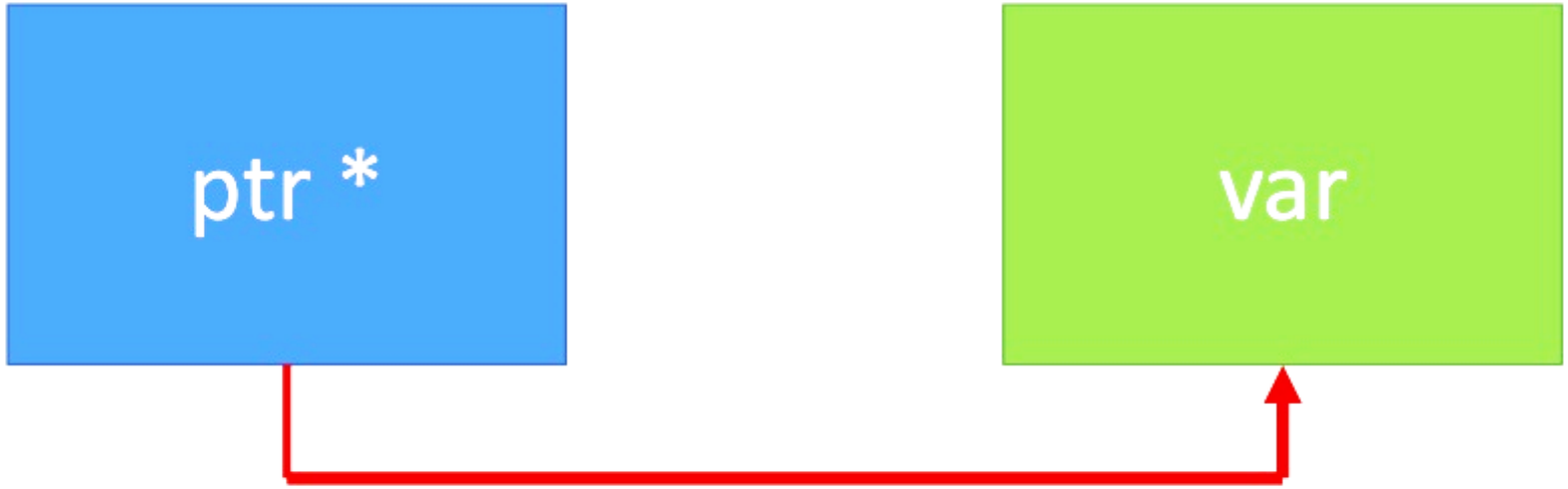
- Pointers in C++
- Type of Pointer in C++
- NULL Pointer in C++
- Void Pointer in C++
- Pointer Variable in C++
- Reference in C++
- Pointer and Function in C++

LAB-08 CONTENTS

- Pointer and Array in C++
- Pointer and Strings in C++

C++ Pointers

- Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python.
- Pointers are used in C++ program to access the memory and manipulate the address.



Pointers

Pointers

- Pointers are the most powerful feature of C and C++. These are used to create and manipulate data structures such as linked lists, queues, stacks, trees etc.
- The virtual functions also require the use of pointers. These are used in advanced programming techniques. (e.g. OOP).
- To understand the use of pointers, the knowledge of memory locations, memory addresses and storage of variables in memory is required.

Memory addresses & Variables

- Computer memory is divided into various locations.
- Each location consists of 1 byte. Each byte has a unique address.
- When a program is executed, it is loaded into the memory from the disk.
- It occupies a certain range of these memory locations. Similarly, each variable defined in the program occupies certain memory locations.
- For example, an int type variable occupies two bytes and float type variable occupies four bytes.

Computer Memory

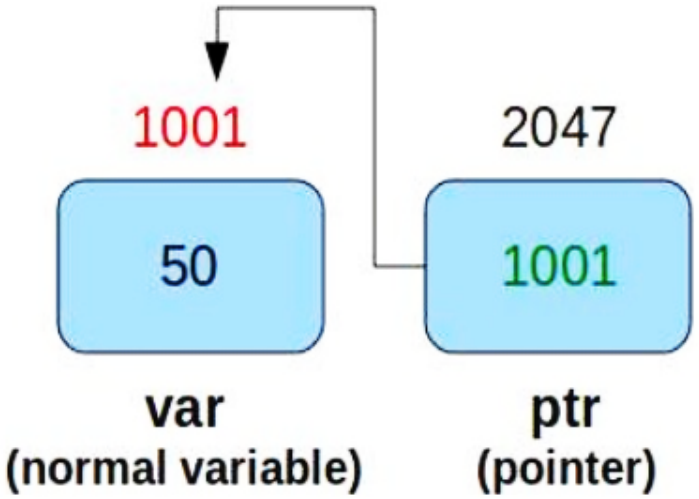
- To understand pointers, you should have the knowledge of address in computer memory.
- A *computer memory location* has an *address* and holds a *content*. The *address* is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly.
- Typically, each address location holds 8-bit (i.e., 1-byte) of data.
- It is entirely up to the programmer to interpret the meaning of the data, such as integer, real number, characters or strings.

Computer Memory

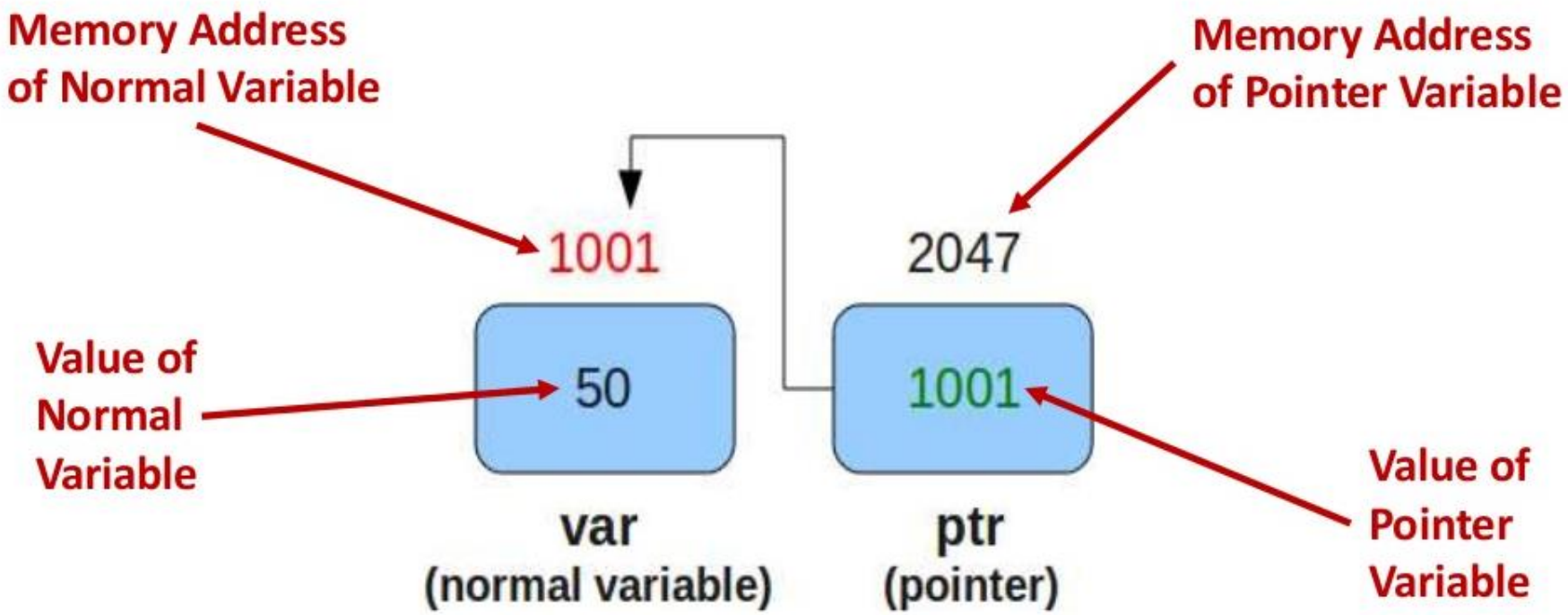
- Each address location typically hold 8-bit (i.e., 1-byte) of data. A 4-byte int value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses.
- Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.
- To know where the data is stored, C++ has an & operator. The & (reference) operator gives you the address occupied by a variable.
- If var is a variable then, &var gives the address of that variable.

Pointers in C++

- A normal variable is used to store value.
- A pointer variable is used to store address / reference of another variable.
- Pointers are symbolic representation of addresses
- We can have a pointer to any variable type.

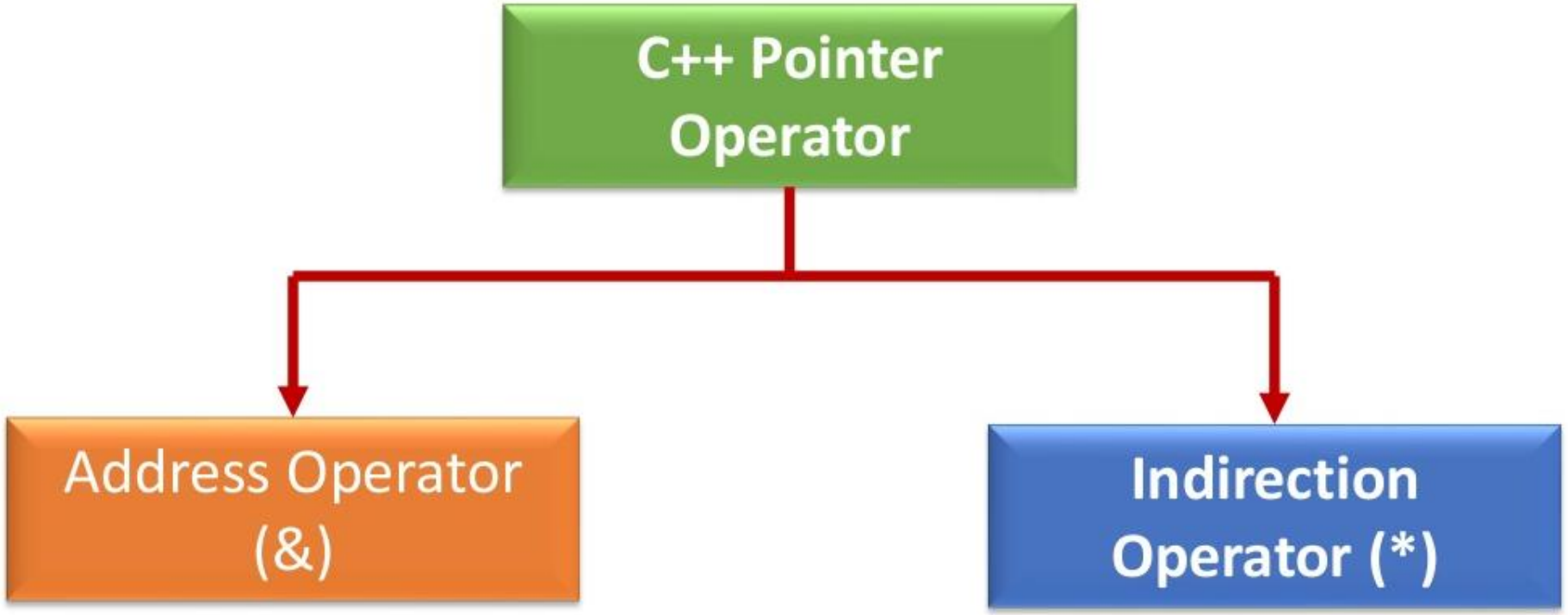


Pointers in C++



C++ Pointer Operator

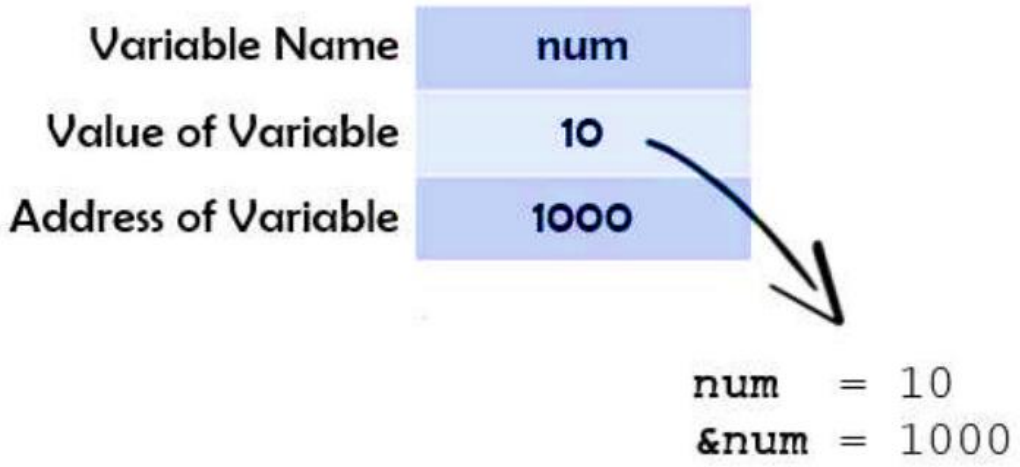
- C++ provides two pointer operators one is value at operator or indirection operator and address operator.



C++ Address of Operator (&)

- The & is a unary operator means it requires only one operand.
- The Address of Operator returns the memory address of its operand.
- Address of operator has the same precedence and right-to-left associativity as that of other unary operators.
- Symbol for the bitwise AND and the address of operator are the same but they don't have any connection between them

C++ Address of Operator (&)



Address of Operator Example

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

int main()
{
  int num=10;

  cout << "Value of number: " << num << endl;
  cout << "Address of number : " << &num << endl;
}
```

```
Value of number: 10
Address of number : 0x6ffe3c
```

One More Example

```
#include <iostream>
using namespace std;
int main()
{
   int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout << &var1 << endl;
   cout << &var2 << endl;
   cout << &var3 << endl;
}
```

```
0x7fff5fbff4c
0x7fff5fbff48
0x7fff5fbff44
```

Explanation of the Previous Program

- **Note:** We may not get the same result on your system.
- The 0x in the beginning represents the address is in hexadecimal form.
- Notice that first address differs from second by 4-bytes and second address differs from third by 4-bytes.
- This is because the size of integer (variable of type int) is 4 bytes in 64-bit system.

Summary-Address of Operator

Point	Explanation
Operator	Address of
Type	Unary Operator
Operates on	Single Operand
Returns	Address of the operand
Associativity	right-to-left
Explanation	Address of operator gives the computer's internal location of the variable.

Pointers Variables

- C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.
- A pointer is variable in which the memory address of another variable is stored.
- **Syntax to Declare Pointer**

```
data_type *variable_name;

data_type*  variable_name;
```

Pointers Variables

- **Examples**

```
int *p;

    OR,

int* p;
```

- The statement above defines a pointer variable p. It holds the memory address
- The asterisk is a dereference operator which means pointer to.
- Here, pointer p is a pointer to int, i.e., it is pointing to an integer value in the memory address.

Declaring Pointers

```
int *myptr;
```

myptr is a pointer to integer

Pointers Variables

```
// a pointer to an integer value
• int *iPtr;
// a pointer to a double value
• double *dPtr;
// also valid syntax (acceptable, but not favored)
• int* iPtr2;
// also valid syntax (but don't do this)
• int * iPtr3;
// declare two pointers to integer variables
• int *iPtr4, *iPtr5;
```

C++ Indirection/ Deference Operator *

- Getting the address of a variable isn't very useful by itself.
- The dereference operator (*) allows us to get the value at a particular address:
- The Indirection operator * is a unary operator means it requires only one operand.
- Indirection Operator (*) is the complement of Address of Operator (&).
- Indirection Operator returns the value of the variable located at the address specified by its operand.

Indirection Operator Example

```
#include<iostream>
using namespace std;
int main()
{
  int num=10;
  int *ptr;

  ptr=&num;

  cout << " num = " << num << endl;
  cout << " &num = " << &num << endl;
  cout << " ptr = " << ptr << endl;
  cout<< " *ptr = " << *ptr << endl;
}
```

Indirection Operator Example

```
#include<iostream>
using namespace std;
int main()
{
  int num=10;
  int *ptr;

  ptr=&num;

  cout << " num = " << num << endl;
  cout << " &num = " << &num << endl;
  cout << " ptr = " << ptr << endl;
  cout<< " *ptr = " << *ptr << endl;
}
```

```
num = 10
&num = 0xffcc2bd8
ptr = 0xffcc2bd8
*ptr = 10
```

Calculations

// Sample Code for Dereferencing of Pointer

*(ptr) = value at (ptr)

= value at (0xffcc2bd8)

= 10

//Address in ptr is nothing but address of num

Summary Indirection Operator *

Point	Explanation
Operator	Indirection Operator
Type	Unary Operator
Operates on	Single Operand
Returns	Value at address of the operand)
Associativity	right-to-left

- Though multiplication symbol and the "Indirection Operator" symbol are the same still they don't have any relationship between each other.
- Both & and * have a higher precedence than all other arithmetic operators except the unary minus/plus, with which they share equal precedence.

Reference Operator (&) and Deference Operator (*)

- Reference operator (&) as already discussed, gives the address of a variable.
- To get the value stored in the memory address, we use the dereference operator (*).
- **For Example:** If a number variable is stored in the memory address 0x123, and it contains a value 5.
- The reference (&) operator gives the value 0x123, while the dereference (*) operator gives the value 5.
- **Note:** The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

Reference Operator (&) and Deference Operator (*)

```
int main() {
   int *pc, c;
   c = 5;
   cout << "Address of c (&c): " << &c << endl;
   cout << "Value of c (c): " << c << endl << endl;
   pc = &c;    // Pointer pc holds the memory address of variable c
   cout << "Address that pointer pc holds (pc): "<< pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
   c = 11;    // The content inside memory address &c is changed from 5 to 11.
   cout << "Address pointer pc holds (pc): " << pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
   *pc = 2;
   cout << "Address of c (&c): " << &c << endl;
   cout << "Value of c (c): " << c << endl << endl;
   return 0;
}
```

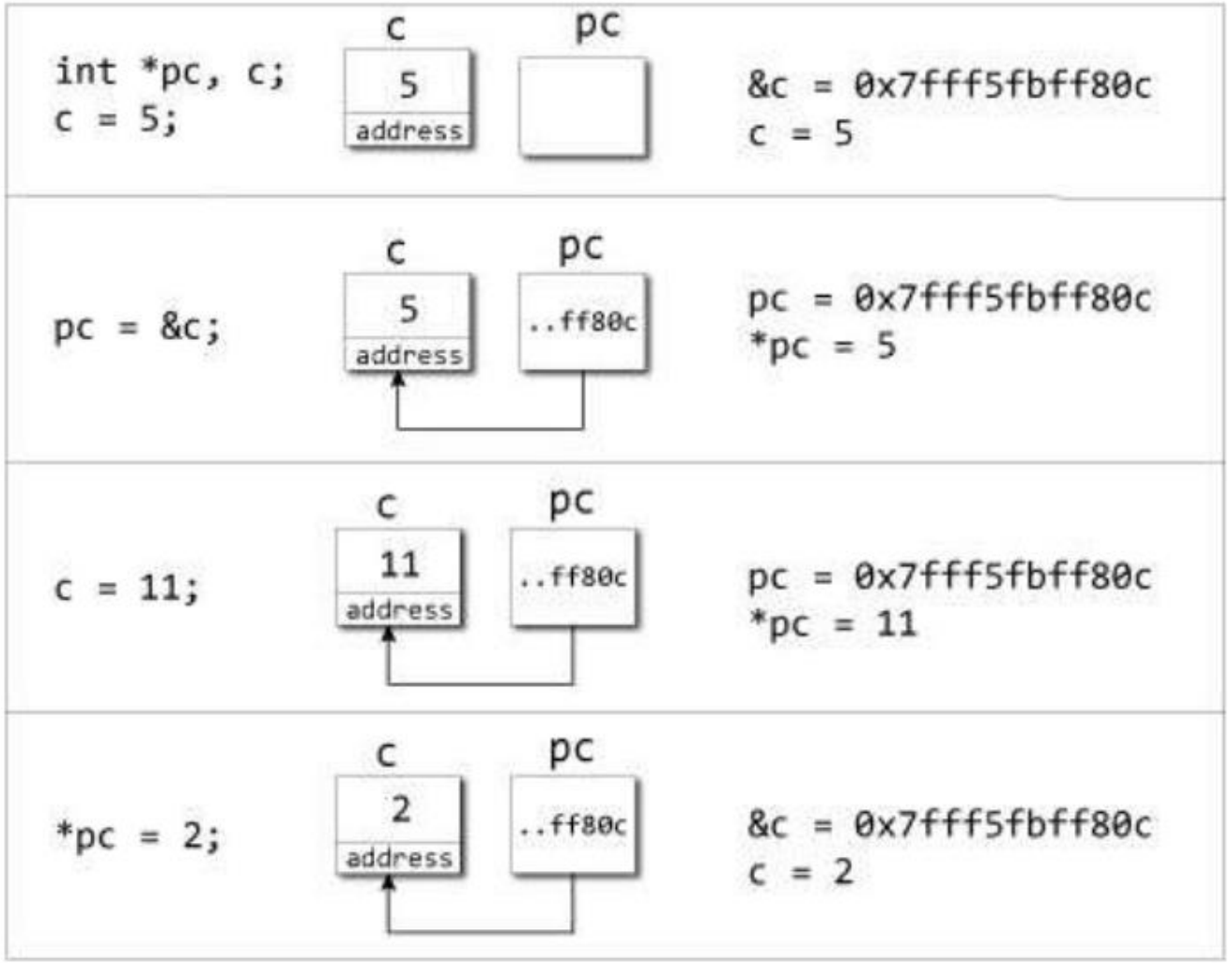
Output of the Previous Program is :

```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c
Value of c (c): 2
```

Explanation of the Previous Program

- When c = 5; the value 5 is stored in the address of variable c - 0x6ffe34.
- When pc = &c; the pointer pc holds the address of c - 0x6ffe34, and the expression (dereference operator) *pc outputs the value stored in that address, 5.
- When c = 11; since the address pointer pc holds is the same as c - 0x6ffe34, change in the value of c is also reflected when the expression *pc is executed, which now outputs 11.
- When *pc = 2; it changes the content of the address stored by pc - 0x6ffe34. This is changed from 11 to 2. So, when we print the value of c, the value is 2 as well.

Explanation of the Previous Program



Why C++ Use Pointer?

- By using pointer variable we can implement dynamic memory allocation.
- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It produce a compact and efficient code.
- It provide a very powerful tool
- Pointer provided alternative way to accessing array element

Rules for Pointer

- An asterisk before a variable in declaration tells the compiler that it is a pointer
- An ampersand(&) before a variable means that you are referring to the address of the variable not its contents is called referencing.
- An asterisk before a pointer variable in the code means that you are dereferencing or trying to access the value in the variable that holds the address of another variable.

Pointer

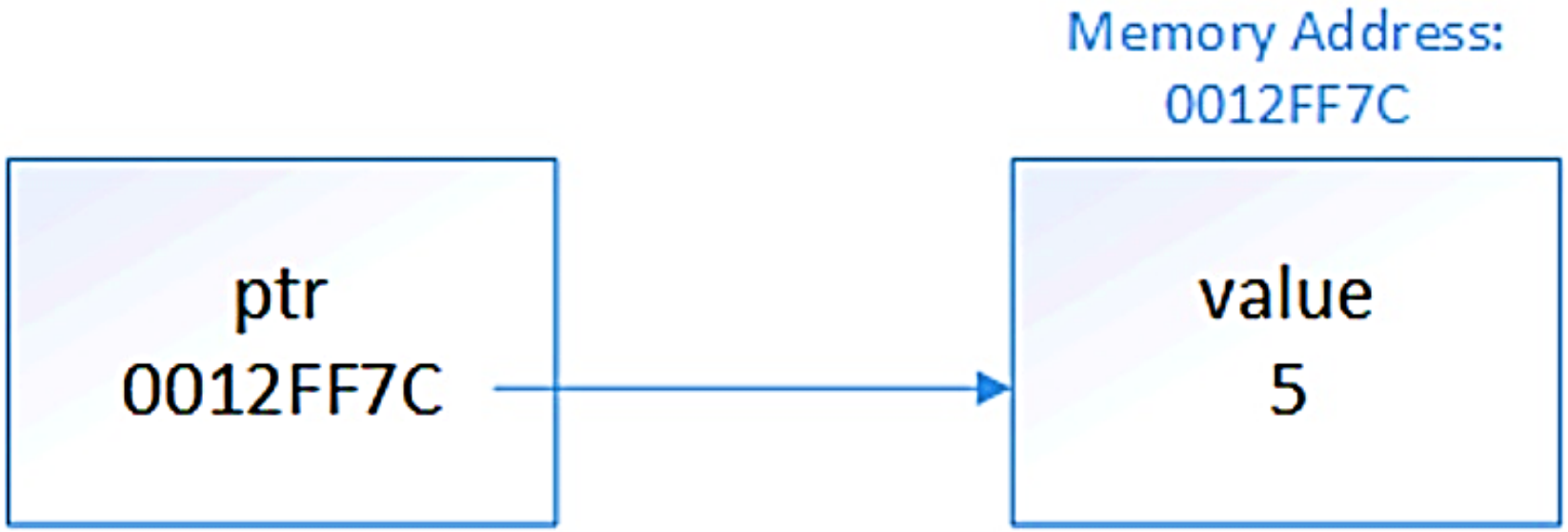
- **Advantages**
- Pointers provide direct access to memory.
- Pointers allows us to return more than one value from the functions.
- Reduces the execution time of the program.
- Pointers provide a way to perform dynamic memory allocation and deallocation.
- **Disadvantages**
- Uninitialized pointers might cause segmentation fault.
- Dynamically allocated block needs to be freed explicitly.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Assigning a Value to a Pointer

- Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address. One of the most common things to do with pointers is have them hold the address of a different variable.
- To get the address of a variable, we use the address-of operator:

```
int value = 5;
int *ptr;
// assign address of variable value to ptr
int ptr = &value;
```

Assigning a Value to a Pointer



Example of Assign Value to Pointer

```
#include <iostream>
using namespace std;
int main()
{
        int *ptr , x;
        x = 10;
        ptr = &x;
        cout << "The value of x: \t" << x << endl;
        cout << "The value of *ptr: \t" <<*ptr << endl;;
        cout << "The value of ptr: \t" <<ptr << endl;;
        cout << "The value of &x: \t" <<&x << endl;;
}
```

```
The value of x:         10
The value of *ptr:      10
The value of ptr:       0x6ffe34
The value of &x:        0x6ffe34

--------------------------------
Process exited after 0.006709 secc
Press any key to continue . . .
```

Types of Pointer

- Null Pointer
- Wild Pointer
- Near Pointer
- Far Pointer
- Huge Pointer
- Generic or Void Pointer

NULL Pointers

- NULL pointer is a type of pointer of any data type and generally takes a value as zero.
- This denotes that NULL pointer does not point to any valid memory address.

```
#include <iostream>
using namespace std;
int main ()
{
  int  *ptr = NULL;
  cout << "The value of ptr is " << ptr ;
  return 0;
}
```

```
The value of ptr is 0
```

void Pointer

- The type of pointers which can **store the address of all types of data types** is called Void pointer.
- It is declared by the keyword void before the name of a pointer.
- **Example:**

```
void *ptr;
```

- The above statement will declare the void pointer ptr which can store the address of all the variable of different data types.

Example of void Pointer-1

```
int main()
{
        void *ptr1,*ptr2,*ptr3,*ptr4;
        int i; char c; float f; double d;
        cout<<"Enter the integer value : "<<endl;
        cin>>i;
        cout<<"Enter the character: "<<endl;
        cin>>c;
        cout<<"Enter the float value: "<<endl;
        cin>>f;
        cout<<"Enter the double value: "<<endl;
        cin>>d;

ptr1=&i;
ptr2=&c;
ptr3=&f;
ptr4=&d;

cout<<"Your entered integer address is: "<<ptr1<<endl;
cout<<"Your entered character address is: "<<ptr2<<endl;
cout<<"Your entered float value address is: "<<ptr3<<endl;
cout<<"Your entered double value address  is: "<<ptr4<<endl;
return 0;
}
```

```
Enter the integer value : 2
Enter the charater: a
Enter the float value: 1.1
Enter the double value: 235

Your entered integer address is: 0x22fe2c
Your entered character address is: 0x22fe2b
Your entered float value address is: 0x22fe24
Your entered double value address  is: 0x22fe18
```

Differences Between A Pointer Variable And A Reference Variable

- **POINTER**
  - Its not necessary to initialize the pointer at the time of declaration.
    - `int a = 10;`
    - `int *P;`
    - `P = &a;` **//It is not necessary**
- **REFERENCE**
  - Its necessary to initialize the Reference at the time of declaration.
    - `int &a = 10;`
    - `int &a;` **//Error here but not in case of Pointer.**

Differences Between A Pointer Variable And A Reference Variable

- **POINTER**
- We can assign NULL to the pointer that indicate that they are not pointing to any valid thing.
  - `int *P = NULL; //Valid`
- **REFERENCE**
- We can not assign NULL to the reference
  - `int &a = NULL; //Error`

Differences Between A Pointer Variable And A Reference Variable

- **POINTER**
- We can take the address of a pointer
- We can create the array of Pointer.
- We can use pointer to pointer.
- **REFERENCE**
- We can't take the address of a reference like you can with pointers.
- We can not create the Array of reference.
- We can not use reference to reference.

Pointer Arithmetic

- The C++ language allows you to perform integer addition or subtraction operations on pointers.
- If ptr points to an integer , ptr + 1 is the address of the next integer in memory after ptr.
- ptr - 1 is the address of the previous integer before ptr.
- **Note** that ptr + 1 does not return the *memory address* after ptr, but the memory address of the *next object of the type* that ptr points to. If ptr points to an integer (assuming 4 bytes), ptr + 3 means 3 integers after ptr, which is 12 memory addresses after ptr. If ptr points to a char, which is always 1 byte, ptr + 3 means 3 chars after ptr, which is 3 memory addresses after ptr.
- When calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to. This is called **scaling**.

Pointer Arithmetic

- We can do some arithmetic operations on pointers

```
z = *ptr *2;

z = *ptr + 10;

z = *ptr / 3;

ptr++ ;

ptr-- ;
```

Division is meaningless so it not possible in pointer

Pointer Arithmetic

- If the pointer contains address of array elements then it is possible to perform **subtraction** on pointer.
- **Example**

```
p2-p1; //valid
```

- In other case subtraction, multiplication, division of pointer is not allowed.

```
p1+p2;    //invalid
p2*p1;   //invalid
p2/p1;  //invalid
```

Pointer Arithmetic

- We can compare two pointer to know which one is bigger or smaller.

```
if(pr1 < ptr2 )
if(pr1 == ptr2 )
if(pr1 <= ptr2 )
if(pr1 >= ptr2 )
```

- We can change the value of the variable through the value at operator.

```
a=10;
ptr=&a;
*ptr=40;
```

Pointer Arithmetic

```
int main()
{
   int value = 7;
   int *ptr = &value;

   cout << ptr << '\n';
   cout << ptr+1 << '\n';
   cout << ptr+2 << '\n';
   cout << ptr+3 << '\n';
 return 0;
}
```

As you can see, each of these addresses differs by 4 (7C + 4 = 80 in hexadecimal). This is because an integer is 4 bytes on the author's machine

```
0012FF7C
0012FF80
0012FF84
0012FF88
```

Pointer Arithmetic

- A pointer can be assigned to the other pointer if both are of same type

```
int main()
{
    int a = 10, *b, *c;
    b = &a;
    c = b;
cout << *b << "\t" << *c;
}
```

```
10 10
```

Pointer Arithmetic

- Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.
- If “p” is a character pointer then “p++” will increment “p” by 1 byte.
- If “p” were an integer pointer its value on “p++” would be incremented by 4 bytes.

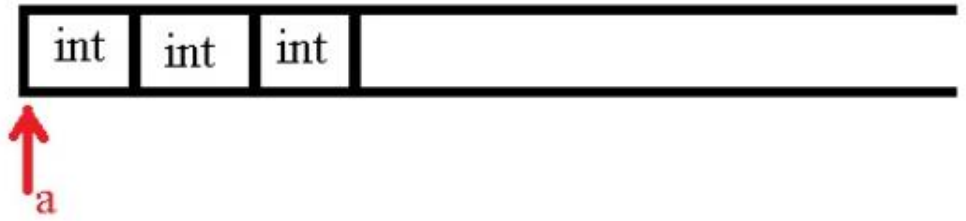
Pointer Arithmetic

- The increment operator (++) will increment the value of pointer according to the pointer's type.
- The address will be incremented by the size of pointer's type. So, the pointer will change its value and will store the address of the next element of the pointer's type. For example, we have a pointer to an integer array:

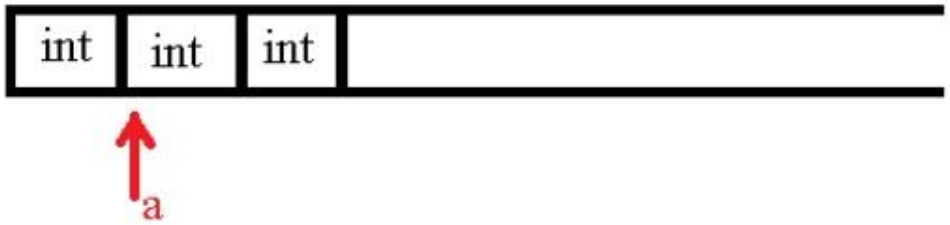
```
int b[3];
int* a=b;
```

Pointer Arithmetic

- Initially this pointer stores the value of the start address of the memory:



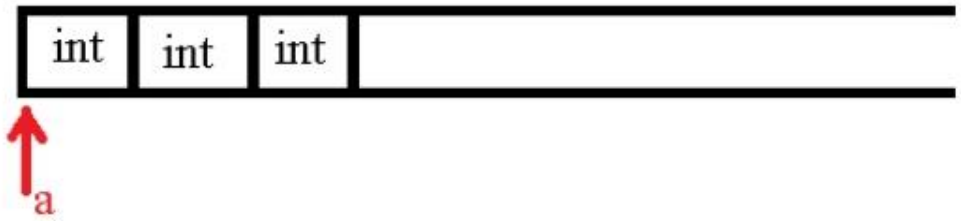
- If we will increment a: (a++)
- It will change its value to the next integer in the array:



Pointer Arithmetic

- The decrement operator (--) decrements the value of the pointer and it stores the address of the previous element of the pointer's type. If we decrement the "a" (a--)pointer again:

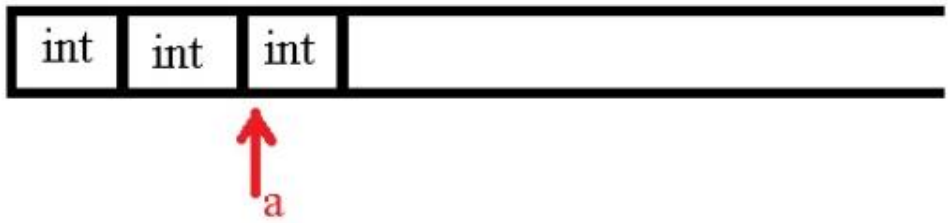
- It will store the address of the previous element again:



Pointer Arithmetic

- We can add or subtract an integer value from a pointer. The addition of a value N to a pointer will forward the pointer to the next N elements of the pointer's type. For example, we can increment the value of pointer a by 2:

- a = a + 2;
- This will change pointer "a" to store address of the 3rd element of the array:

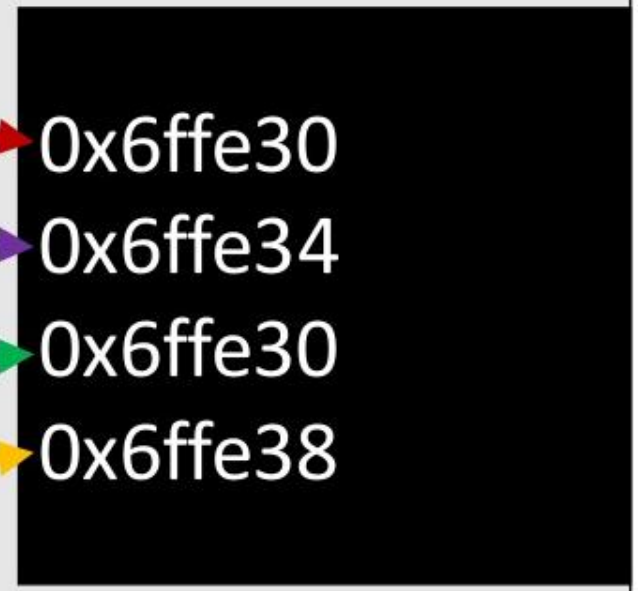


Pointer Arithmetic

```
#include<iostream>
using namespace std;
int main() {

        int b[3];
        int* a=b;
        cout<<a<<endl;
        a++;
        cout<<a<<endl;
        a--;
        cout<<a<<endl;
        a = a + 2;
        cout<<a<<endl;

}
```

Pointer Arithmetic

```
#include<iostream>
using namespace std;
int main() {

        int b[3];
        int* a=b;
        cout<<a<<endl;
        a++;
        cout<<a<<endl;
        a--;
        cout<<a<<endl;
        a = a + 2;
        cout<<a<<endl;

}
```



Pointer in C++

- Common mistakes when working with pointers

```
int c, *pc;
// Wrong! pc is address whereas, c is not an address
pc=c;
// Wrong! *pc is the value pointed by address whereas, %amp; c is an address
*pc=&c;
//Correct! pc is an address and, %amp; pc is also an address
pc=&c;
// Correct! *pc is the value pointed by address and, c is also a value
*pc=c;
```

- In both cases, pointer pc is not pointing to the address of c.

Const Pointer

- Const pointer is a pointer which you don't want to be pointed to a different value. That is, the location stored in the pointer can not change. We can not change where the pointer points.
- **Example :**

```
char * const p
```

- Since the location to which a const pointer points to can not be changed, the following code :

```
char ch1 = '';
char ch2 = 'B' A;
char * const p = &ch1;
p = &ch2;
//will throw an error since address stored in p can not be changed.
```

Using Const With Pass-by-Address

- The keyword **const** can be used on pointer parameters, like we do with references. It is used for a similar situation -- it allows parameter passing without copying anything but an address, but protects against changing the data (for functions that should not change the original)
- **Syntax**

```
const typeName * v
```

- This establishes v as a pointer to an object that cannot be changed through the pointer v.
- Note: This does **not** make v a constant! The pointer v **can** be changed. But, the **target** of v cannot be changed (through the pointer v).
- **Example:**

```
int Function1(const int * list);   // the target of list can't
                                   // be changed in the function
```

Const Pointer

- **Pointer to Constant**

const char * myPtr

- declares a **pointer to a constant character**. You cannot use this pointer to change the value being pointed to:

```
char char_A = 'A';
const char * myPtr = &char_A;
*myPtr = 'J'; // error - can't change value of *myPtr
```

Const Pointer

- **Constant Pointers**

```
char * const myPtr
```

- declares a **constant pointer to a character**. The location stored in the pointer cannot change. You cannot change where this pointer points:

```
char char_A = 'A';
char char_B = 'B';
 char * const myPtr = &char_A;
myPtr = &char_B;
// error - can't change address of myPtr
```

Const Pointer

- **Constant Pointer to a Constant**

```
const int * const ptr4
```

- A constant pointer to a constant is a pointer that can neither change the address it's pointing to and nor can it change the value kept at that address.

```
int var3 = 0;
int var4 = 0;
const int * const ptr4 = &var3;
*ptr4 = 1;          // Error
ptr4 = &var4;  // Error
```

Summary

- Non-constant pointer to non-constant data

```
int * ptr;
```

- Non-constant pointer to constant data

```
const int * ptr;
```

- Constant pointer to non-constant data

```
int x = 5;
int * const ptr = &x;   // must be initialized here
```

- An array name is this type of pointer - a constant pointer (to non-constant data).
- Constant pointer to constant data

```
int x = 5;
const int * const ptr = & x;
```

Which are Able to Change or Not

- Able to change p && *p:
  - `int* p;`
- Able to change p, but not *p
  - `int* const p;`
- Able to change *p, but not p
  - `const int* p;`
- Unable to change p || *p
  - `const int* const p`

Pointers and Functions

- We've seen that regular function parameters are pass-by-value
  - A *formal parameter* of a function is a local variable that will contain a *copy* of the argument value passed in
  - Changes made to the local parameter variable do not affect the original argument passed in
- If a pointer type is used as a function parameter type, then an actual *address* is being sent into the function instead
  - When addresses (pointers) are passed into functions, the function could affect actual variables existing in the scope of the caller
  - In this case, we are not sending the function a *data value* -- instead, you are telling the function where to find a specific piece of data

Pointers and Functions

- Like normal variable, pointer variable can be passed as function argument and function can return pointers as well.

- There are two approaches to passing argument to a function:
- Call by Value
- Call by Reference / Address

Call by Value

```
void fun(int , int );
int main(){
        int A=10,B=20;
        cout << "\nValues before calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
        fun(A,B);          //Statement 1
        cout << "\nValues after calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
}
void fun(int X, int Y) {      //Statement 2
   X=11;
   Y=22;
}
```

Call by Value

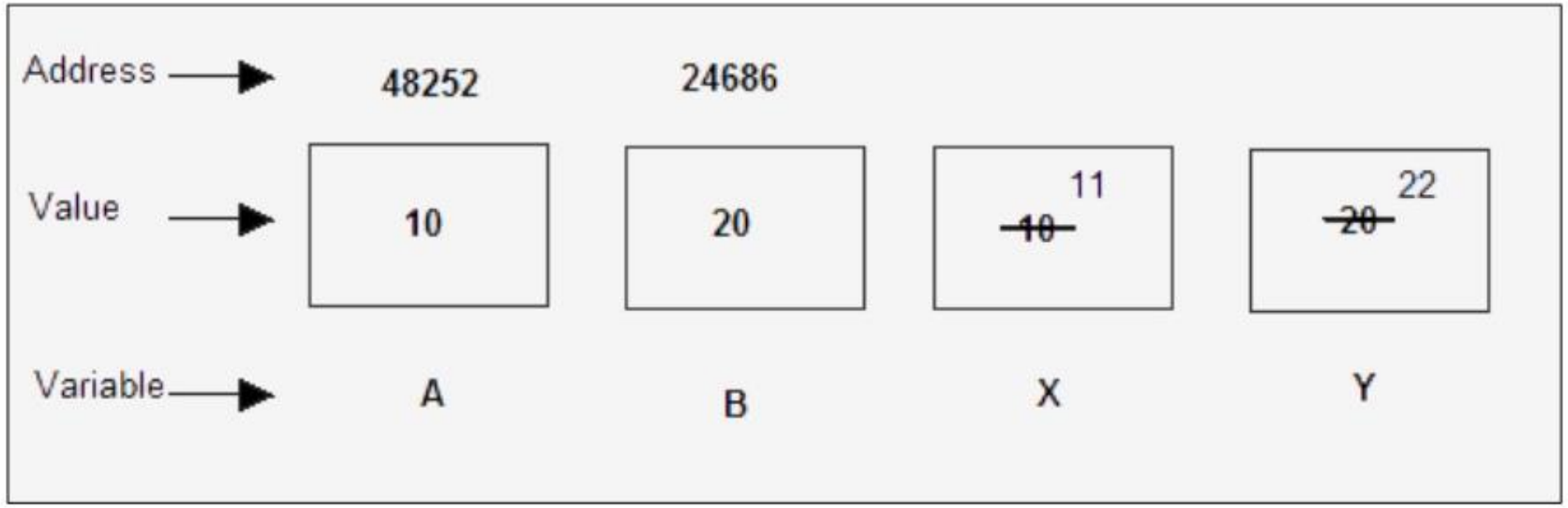
```
void fun(int , int );
int main(){
        int A=10,B=20;
        cout << "\nValues before calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
        fun(A,B);          //Statement 1
        cout << "\nValues after calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
}
void fun(int X, int Y) {      //Statement 2
   X=11;
   Y=22;
}
```

In this approach, the values are passed as function argument to the definition of function

```
Values before calling
A : 10
 B : 20
Values after  calling
A : 10
B : 20
```

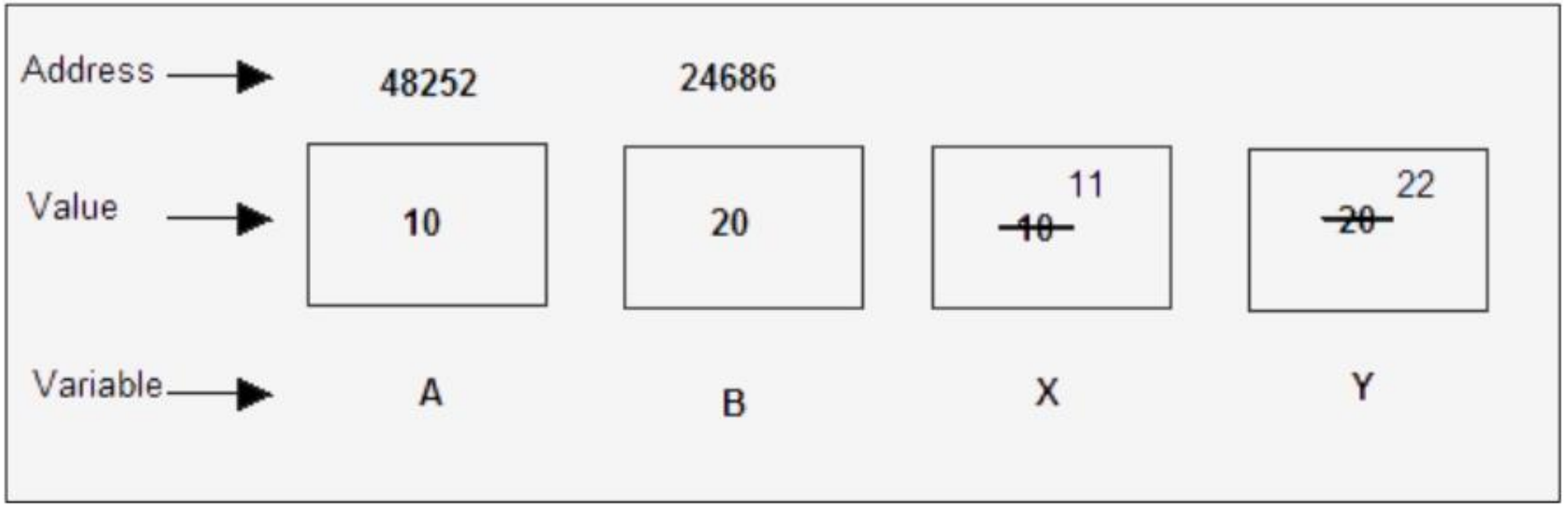
Explanation of the Previous Program

- In the previous example, statement 1 is passing the values of A and B to the calling function fun(). fun() will receive the value of A and B and put it into X and Y respectively. X and Y are value type variables and are local to fun(). Any changes made by value type variables X and Y will not effect the values of A and B.



Explanation of the Previous Program

- In the previous example, statement 1 is passing the values of A and B to the calling function fun(). fun() will receive the value of A and B and put it into X and Y respectively. X and Y are value type variables and are local to fun(). Any changes made by value type variables X and Y will not effect the values of A and B.



Call by Reference

```
void fun(int*, int*);
int main(){
        int A=10,B=20;
        cout << "\nValues before calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
        fun(&A,&B);          //Statement 1
        cout << "\nValues after calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
}
void fun(int *X, int *Y) {    //Statement 2
   *X=11;
   *Y=22;
}
```

Call by Reference

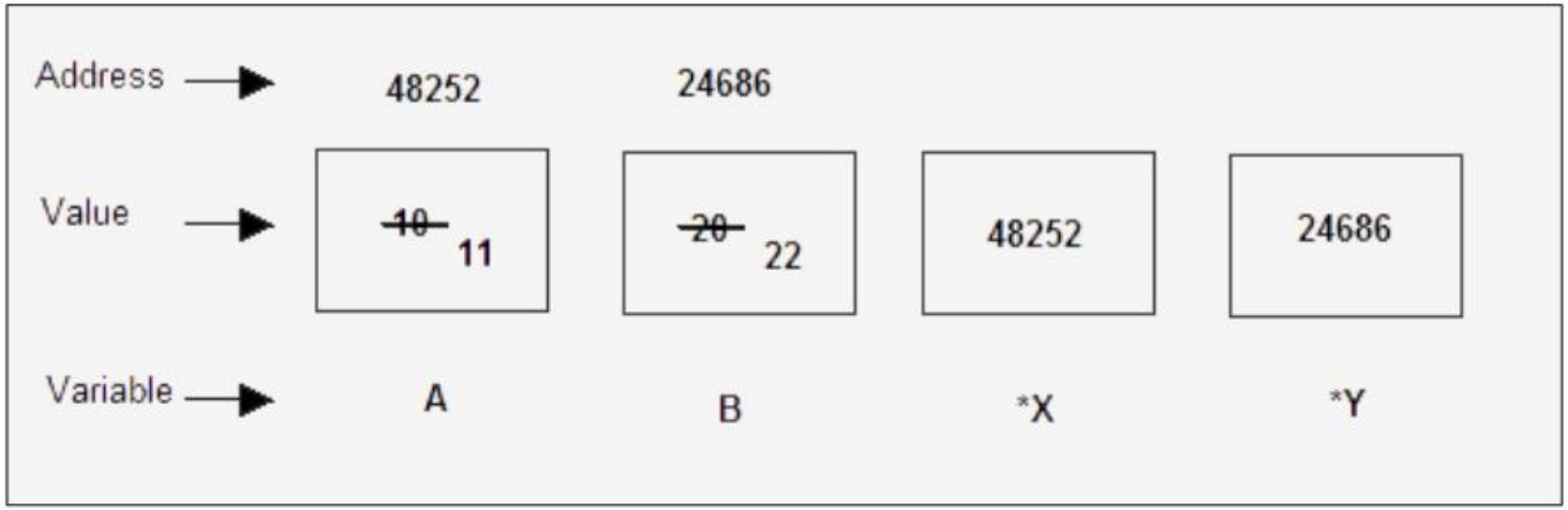
```
void fun(int*, int*);
int main(){
        int A=10,B=20;
        cout << "\nValues before calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
        fun(&A,&B);          //Statement 1
        cout << "\nValues after calling";
        cout << "\nA : " << A;
        cout << "\nB : " << B;
}
void fun(int *X, int *Y) {    //Statement 2
   *X=11;
   *Y=22;
}
```

In this approach, the references / addresses are passed as function argument to the definition of function.

```
Values before calling
A : 10
 B : 20
Values after  calling
A : 11
B : 22
```

Explanation of the Previous Program

- In the previous example, statement 1 is passing the reference of A and B to the calling function fun(). fun() must have pointer formal arguments to receive the reference of A and B. In statement 2 *X and *Y is receiving the reference A and B. *X and *Y are reference type variables and are local to fun(). Any changes made by reference type variables *X and *Y will change the values of A and B respectively.



Call by Value Vs. Call by Reference

Call by Value	Call by Reference
The actual arguments can be variable or constant.	The actual arguments can only be variable.
The values of actual argument are sent to formal argument which are normal variables.	The reference of actual argument are sent to formal argument which are pointer variables.
Any changes made by formal arguments will not reflect to actual arguments.	Any changes made by formal arguments will reflect to actual arguments.

Another Example

```
void GetNumber (int *num)
{
        cout << "Enter Number:\t";
        cin >> *num;
}
```

This Function Get Input From User

```
void Square (int *num)
{
        int result = (*num) * (*num);
        cout << "Square is:\t" << result << endl;
}
```

This Function Calculate The Square Of Number

Another Example

```
//main function
int main()
{
        int number;
        // Calling of GetNumber function
        GetNumber(&number);
        // Calling of Square function
        Square(&number);
}
```

```
Enter Number:  25
Square is:    625
```

Function Returning Pointer

- Like normal variable, a function can also return reference or address. When function returns reference or address, the return type of function must be pointer type.
- **Syntax for Function Returning Pointer:**

```
return-type *function-name(argument list)
{
        - - - - - - - - - - - - - -
        body of function
        - - - - - - - - - - - - - -
}
```

Example for Function Returning Pointer

```
#include<iostream>
using namespace std;
int *reference(int);
int main() {
   int A=10;
   int *ptr;
          cout << "\nAddress of " << A ;
          cout<< " in main() is " << &A;
   ptr = reference(A);
          cout << "\nAddress of " << A ;
          cout<< " in reference() was " << ptr;
}
int *reference(int n) {
   return &n;
}
```

Example for Function Returning Pointer

```
#include<iostream>
using namespace std;
int *reference(int);
int main() {
   int A=10;
   int *ptr;
          cout << "\nAddress of " << A ;
          cout<< " in main() is " << &A;
   ptr = reference(A);
          cout << "\nAddress of " << A ;
          cout<< " in reference() was " << ptr;
}
int *reference(int n) {
   return &n;
}
```

Function Returning a Address or Reference

```
Address of 10 in main() is 0x6ffe34
Address of 10 in reference() was 0x6ffe10
```

Pointer to Function

- Like normal variable, Every function has reference or address, and if we know the reference or address of function, we can access the function using its reference or address. This is the another way of accessing function using pointer.
- **Syntax for Declaring a Pointer to Function**

```
return-type (*ptr-function)(argument list);
```

- **return-type :** type of value function will return.
- **argument list :** represents the type and number of value function will take, values are sent by the calling statement.
- **(*ptr-function) :** The parentheses around *ptr-function tells the compiler that it is pointer to function.

Pointer to Function

- **Syntax for Declaring a Pointer to Function**

return-type (*ptr-function)(argument list);

- If we write *ptr-function without parentheses then it tells the compiler that ptr-function is a function that will return a pointer.

- The pointer to function will be accessed as:

(*ptr-function)(val1,val2...n);

Example of using Pointer to Function

```
#include<iostream>
using namespace std;
int Sum(int , int);
int (*ptr)(int , int);
int main() {
        int a, b, rs;
        cout << "\nEnter 1st number : ";
        cin >> a;
        cout << "\nEnter 2nd number : ";
        cin >> b;
        ptr = Sum;
        rs = (*ptr)(a,b);
        cout << "\nThe sum is : " << rs;
}
int Sum(int x , int y) {
   return x + y;
}
```

Example of using Pointer to Function

```
#include<iostream>
using namespace std;
int Sum(int , int);
int (*ptr)(int , int);
int main() {
        int a, b, rs;
        cout << "\nEnter 1st number : ";
        cin >> a;
        cout << "\nEnter 2nd number : ";
        cin >> b;
        ptr = Sum;
        rs = (*ptr)(a,b);
        cout << "\nThe sum is : " << rs;
}
int Sum(int x , int y) {
   return x + y;
}
```

Assigning The Address of Sum() to Pointer ptr.

Example of using Pointer to Function

```
#include<iostream>
using namespace std;
int Sum(int , int);
int (*ptr)(int , int);
int main() {
        int a, b, rs;
        cout << "\nEnter 1st number : ";
        cin >> a;
        cout << "\nEnter 2nd number : ";
        cin >> b;
        ptr = Sum;
        rs = (*ptr)(a,b);
        cout << "\nThe sum is : " << rs;
}
int Sum(int x , int y) {
   return x + y;
}
```

Calling and Passing two Values to Sum()

Example of using Pointer to Function

```
#include<iostream>
using namespace std;
int Sum(int , int);
int (*ptr)(int , int);
int main() {
        int a, b, rs;
        cout << "\nEnter 1st number : ";
        cin >> a;
        cout << "\nEnter 2nd number : ";
        cin >> b;
        ptr = Sum;
        rs = (*ptr)(a,b);
        cout << "\nThe sum is : " << rs;
}
int Sum(int x , int y) {
   return x + y;
}
```

```
Enter 1st number : 25
Enter 2nd number : 50
The sum is : 75
```

Arrays and Pointers

- A special relationship exists between pointers and arrays.
- When we declare an array, compiler allocates continuous blocks of memory so that all the elements of an array can be stored in that memory.
- The name of the array is like a pointer which contains the address of the first element.
- So, if a program declared an array:

```
int a[10];
```

- a (array's name) is the address of the first array element and is equivalent to the expression,

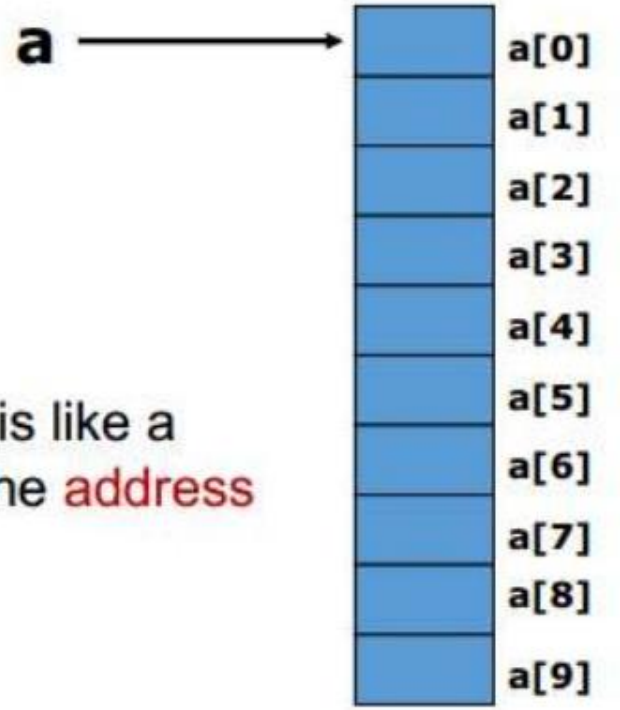
```
&a[0]
```

Arrays and Pointers

- Thus array name works as pointer variable.
- The address of first element is also known as base address.

int a[10];

The name of the array is like a pointer which contain the address of the first element



Arrays and Pointers

- Array name is base address of array

```
int vals[] = {4, 7, 11};
cout << vals;      // displays 0x4a00
cout << vals[0];   // displays 4
```

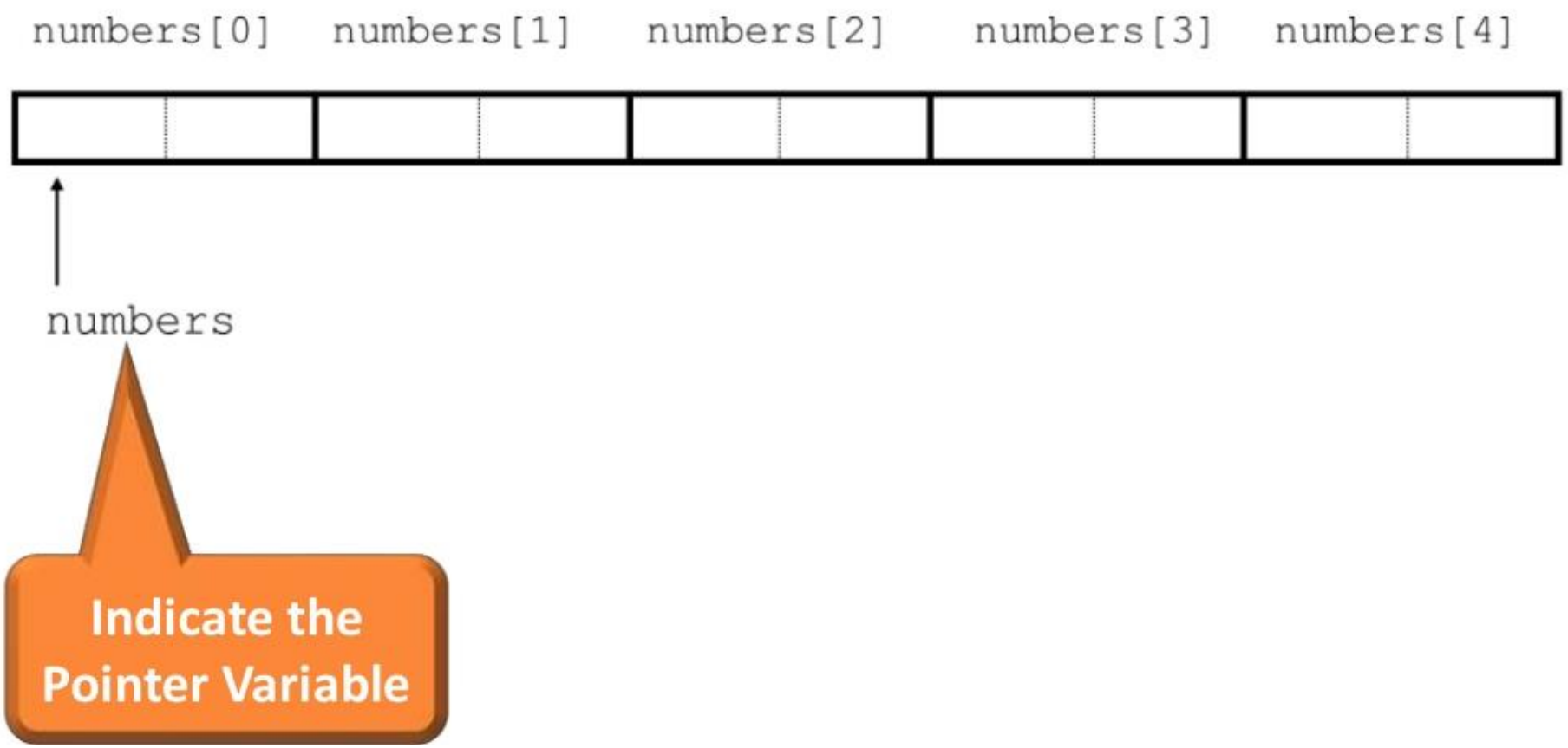
Lets Takes Another Example

```
int arr[]={4,7,11};
int *ptr = arr;
```

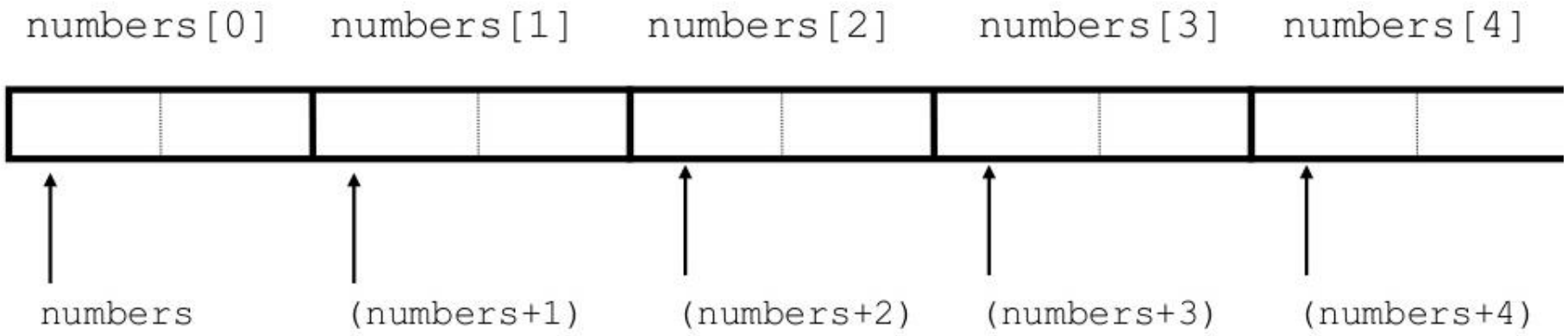
- What is ptr + 1?
- It means (address in ptr) + (1 * size of an int)

```
// displays 7
cout << *(ptr+1);

// displays 11
cout << *(ptr+2);
```

Array and Pointers



Array and Pointers



Program Example

```
#include <iostream>
using namespace std;
int main()
{
        short numbers[] = {10, 20, 30, 40, 50};

        cout << "The first element of the array is ";
        cout << *numbers << endl;
        return 0;
}
```

```
The first element in the array is 10
```

Program Example - 2

```
int main()
{
        int array[5] = {2, 4,  6, 8, 10};
        int *aptr;
        aptr = array;
        cout << "Address of Array:\t" << &array[0] << endl;
        cout << "Address of aptr:\t" << aptr << endl;
        cout << "\nValue of Array:\t" << array[0] << endl;
        cout << "Value of aptr:\t\t" << *aptr << endl;
}
```

```
Address of Array:  0x22ff20
Address of aptr:    0x22ff20

Value of  Array:  2
Value of aptr:    2
```

Array Access

- Array notation arr[i] is equivalent to the pointer notation *(arr + i)
- Assume the variable definitions

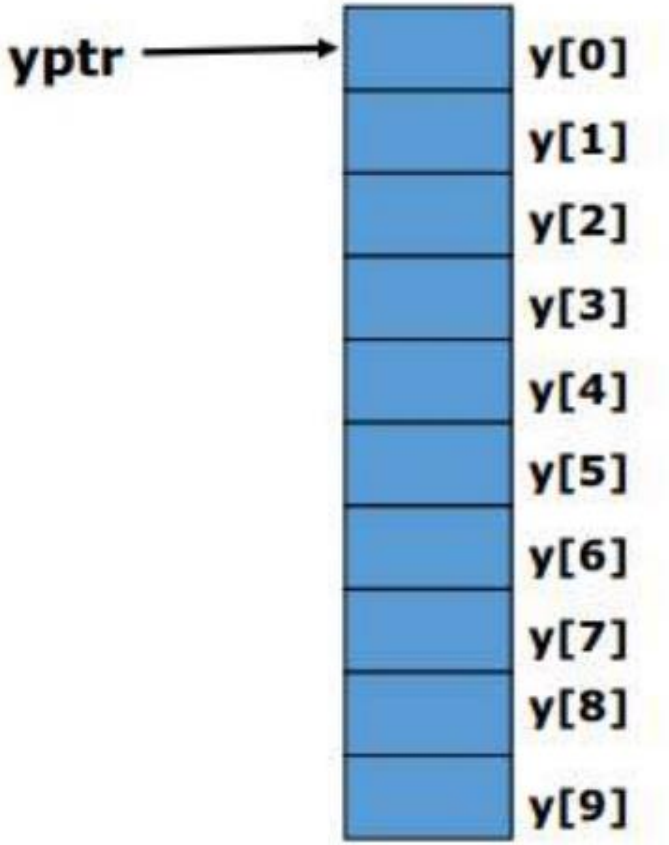
```
int arr[]={4,7,11};
int *ptr = arr;
```

- Examples of use of ++ and --

```
ptr++;   // points at 7
ptr--;   // now points at 4
```

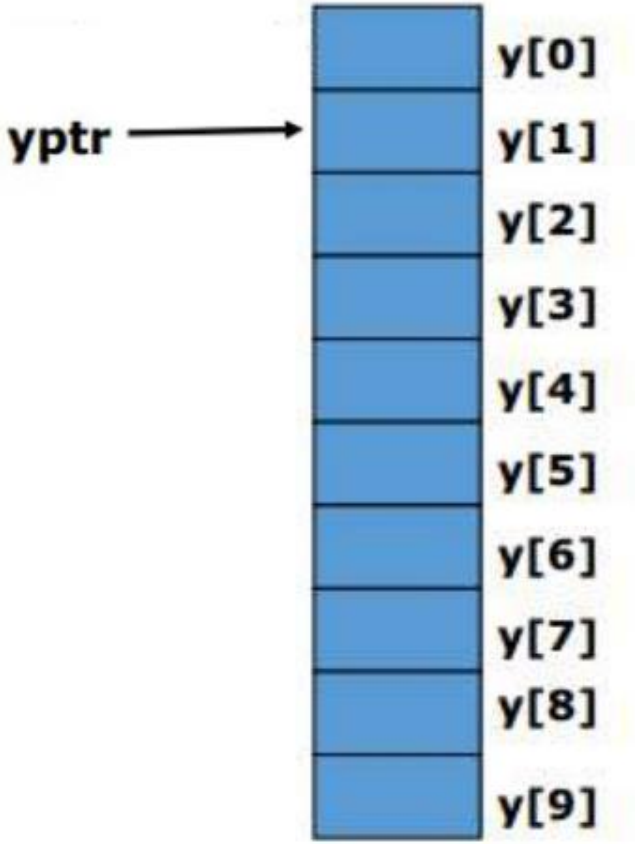
Array Access

```
int y [ 10 ] ;
int *yptr ;
yptr = y ;
```



Array Access

```
int y [ 10 ] ;
int *yptr ;
yptr = y ;
yptr ++ ;
```



Program Example

```
#include<iostream>
using namespace std;
int main()
{
        int y[10];
        int *yptr;
        yptr=y;
        cout <<"Pointer Address is: " <<yptr<<endl;
        yptr++;
        cout <<"Pointer Address is: " <<yptr;
}
```

```
Pointer Address is: 0x6ffe20
Pointer Address is: 0x6ffe24
```

Program Example-2

```
int main()
{
int array[5] = {2, 4,  6, 8, 10};
int *aptr;
aptr = array;
        cout << "Address of Array:\t" << &array[0] << endl;
        cout << "Address of aptr:\t" << aptr << endl;
        cout << "\nValue of Array:\t" << array[0] << endl;
        cout << "Value of aptr:\t\t" << *aptr << endl;
return 0;
}
```

```
Address of Array:  0x22ff20
Address of aptr:    0x22ff20

Value of  Array:  2
Value of aptr:    2
```

Program Example - 3

```
int main() {
   float a[5];
   float *ptr;
   cout << "Displaying address using arrays: "<<endl;
   for (int i = 0; i < 5; ++i) {
      cout << "&a[" << i << "] = " << &a[i] <<endl;
   }
         ptr = a;   // ptr = &a[0]
         cout<<"\nDisplaying address using pointers: "<< endl;
    for (int i = 0; i < 5; ++i) {
      cout << "ptr + " << i << " = "<<ptr+i <<endl;
   }
}
```

Output of the Previous Program is :

```
Displaying address using arrays:
&a[0] = 0x7fff5fbff880
&a[1] = 0x7fff5fbff884
&a[2] = 0x7fff5fbff888
&a[3] = 0x7fff5fbff88c
&a[4] = 0x7fff5fbff890

Displaying address using pointers:
ptr + 0 = 0x7fff5fbff880
ptr + 1 = 0x7fff5fbff884
ptr + 2 = 0x7fff5fbff888
ptr + 3 = 0x7fff5fbff88c
ptr + 4 = 0x7fff5fbff890
```

Program Example - 4

```
int main() {
   int array []={78,45,12,89,56,23,79,46,13,82};
   int *ptr; // Pointer variable
    // Assigning reference of array in pointer variable
          ptr = array;
   cout << "\nValues : ";
   for(int a=1;a<=10;a++)
   {
      cout <<*ptr<<" "; // Displaying values of array
       // using pointer Incrementing pointer variable
      ptr++;
   }
return 0;
}
```

Output of the Previous Program is :

```
Values : 78, 45, 12, 89, 56, 23, 79, 46, 13, 82,
```

In the previous example statement 1 creates an array of 10 elements. Statement 2 creates a pointer variable ptr. As said above array name works as pointer variable therefore statement 3 is assigning the address of array in pointer variable ptr. Now ptr have the address of first element in an array. Statement 4 will display the value at address of ptr. After display the first value, statement 5 increase the pointer variable ptr to point to next element in an array and statement 4 will display the next value in an array until the loop ends.

Pointer Array

- Array is a collection of values of similar type. It can also be a collection of references of similar type.
- **Syntax**

Data-type * array [size];

- **Example**

```
int *array[3]
```

Pointer Array Example

```
int main() {
    int x=10,y=20,z=30;
   // Declaring array of three pointer
   int *array[3];
    // Assigning reference of x to array 0th position
   array[0] = &x;
   // Assigning reference of y to array 1th position
   array[1] = &y;
   // Assigning reference of z to array 2nd position
   array[2] = &z;
   cout << "\nValues : ";
   for(int a=0;a<3;a++)
      cout << *array[a]<<" ";

}
```

Pointer Array Example

```
int main() {
    int x=10,y=20,z=30;
   // Declaring array of three pointer
   int *array[3];
    // Assigning reference of x to array 0th position
   array[0] = &x;
   // Assigning reference of y to array 1th position
   array[1] = &y;
   // Assigning reference of z to array 2nd position
   array[2] = &z;
   cout << "\nValues : ";
   for(int a=0;a<3;a++)
      cout << *array[a]<<", ";

}
```

```
Values : 10, 20, 30,
```

Character Pointers and Strings

- Initialize to a character string.

```
char* a = "Hello";
```

- a is pointer to the memory location where ‘H’ is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

Character Pointers and Strings

```
#include<iostream>
using namespace std;
int main() {
        char* a = "Hello";
        cout<<a<<endl;
        cout<<&a<<endl;
        cout<<*a<<endl;
        cout<<a[0]<<endl;
        cout<<a++<<endl;
        cout<<*a++<<endl;

}
```

Character Pointers and Strings

```
#include<iostream>
using namespace std;
int main() {
        char* a = "Hello";
        cout<<a<<endl;
        cout<<&a<<endl;
        cout<<*a<<endl;
        cout<<a[0]<<endl;
        cout<<a++<<endl;
        cout<<*a++<<endl;

}
```

```
Hello
0x6ffe48
H
H
Hello
e
```

Const Pointers and C-style Strings

- We've seen how to declare a character array and initialize with a string:

```
char name[20] = "Hina Adil";
```

- Note that this declaration creates an array called name (of size 20), which can be modified.
- Another way to create a variable name for a string is to use just a pointer:

```
char ch = "Hina Adil";
```

- However, this does NOT create an array in memory that can be modified. Instead, this attaches a pointer to a fixed string, which is typically stored in a "read only" segment of memory (cannot be changed). So it's best to use const on this form of declaration:

```
const char* ch= "Hina Adil";     // better
```

Const Pointers and C-style Strings

- Note: It would be legal to modify the contents of name above, but it would NOT be legal to modify the contents of ch:

```
name[1] = 'I';      // name is now "HIna Adil"
```

```
greeting[1] = 'I';     // ILLEGAL!
```

Near, Far And Huge Pointers

- The near, far and huge pointers are pointers that have to be used to deal with segmented memory architecture. They are relevant to the 16 bit Intel Architecture, so they are not used with the most part of the modern computers. 16 bits compilers (Turbo C++, Borland C++) support different types of pointers: the type of pointers is different by the memory location it can point up. The memory location in this case is calculated by using segment register and offset register. Both registers have size of 16 bits. The address is calculating by the following formula:

Address = SegmentRegister << 4 + OffsetRegister

Near, Far And Huge Pointers

- Near pointer is only 16 bits wide. It holds only the offset address. That's why it can access only memory with the maximum 0xFFFF offset in the current segment.
- Far pointer is 32 bits wide. It holds the segment address (16 bits) and the offset address (16 bits). Far pointer can point to any segment and any offset in this segment. You can create a far pointer by using MK_FP(segment, offset) macro:

```
int far * farIntptr = (int far * )MK_FP(0xA000, 0x1234);
```

- Using a far pointer for segment 0xA000 you can access address in range from 0xA0000 up to 0xAFFFF. If you will try to increase the pointer's offset value 0xFFFF it will be changed to zero.

Near, Far And Huge Pointers

- Huge pointer is 32 bits wide too. The main difference from the far pointer is that you can change the pointer's segment address. For example, if you will try to increment the pointer with address 0xAFFFF - we will change its segment address to 0xB000.
- Huge pointer can be created by using MK_FP(segment, offset) macro too:

```
int huge * hugeIntPtr = (int huge * )MK_FP(0xA000, 0xFFF0);
```

Double and Triple pointer. Multiple indirection

- Double and triple pointer is a kind of multiple indirection. A double pointer is a pointer to pointer. It means that this pointer stores the address of another pointer that points to a variable. Take a look on example:

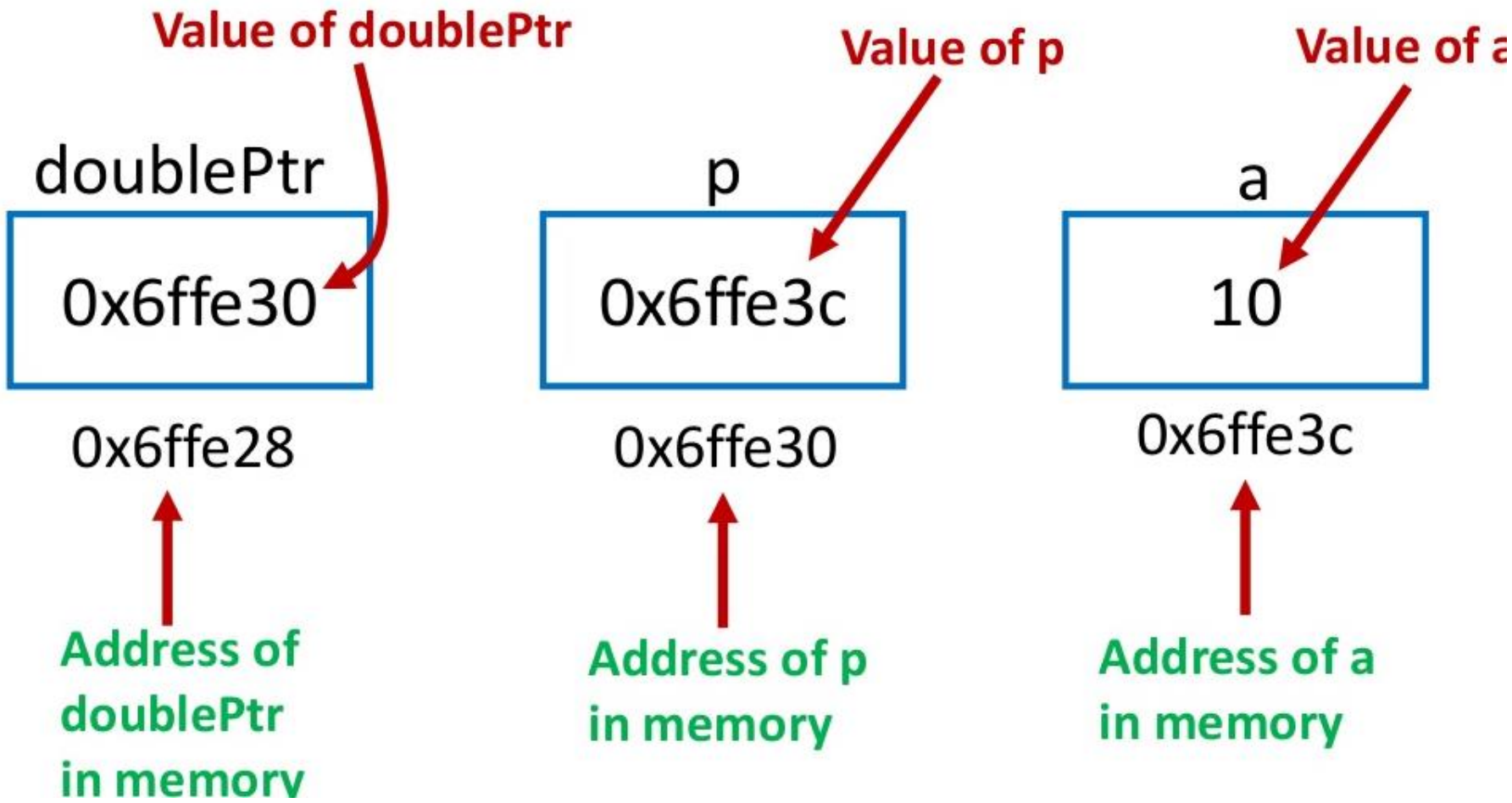
```
int a = 10;
cout << "The address of \"a\" variable is " << &a << endl;

//pointer to int
int* p;

p = &a;  // now p is equal to the address of 'a' variable
cout << "The value of \"p\" variable is " << p << endl;
//pointer to pointer
int** doublePtr = &p;
cout << "The value of \"doublePtr\" variable is " << doublePtr << endl;
```

Output of the Previous Program is :

```
The address of \"a\" variable is 0x6ffe3c
The value of \"p\" variable is is 0x6ffe3c
The value of \"doublePtr\" variable is is 0x6ffe30
```

Double and Triple pointer. Multiple indirection

- It can be represented by the following image:



Double and Triple pointer. Multiple indirection

- If you want to get the value referenced by a double pointer you will have to use the dereference operator twice:

```
int a = 10;
cout << "The address of \"a\" variable is " << &a << endl;

//pointer to int
int* p;

p = &a;  // now p is equal to the address of 'a' variable
cout << "The value of \"p\" variable is " << p << endl;
//pointer to pointer
int** doublePtr = &p;
cout << "The value of \"doublePtr\" variable is " << doublePtr << endl;
cout<< "The value is: " <<endl;
```

Output of the Previous Program is :

```
The address of \"a\" variable is 0x6ffe3c
The value of \"p\" variable is is 0x6ffe3c
The value of \"doublePtr\" variable is is 0x6ffe30
The Value is: 10
```

Double and Triple pointer. Multiple indirection

- Double and triple pointers are often used for 2 reasons - passing a pointer to a function and dynamic allocation of memory for 2D and 3D arrays. The use of pointers for dynamic memory allocation is described in C++ Dynamic Memory.
- To understand how double pointers are used with functions, we can examine the following example: you have a function with pointer inside it:

```
void function1()
{
    int *p = new int;   // a pointer to int
    (*p) = 10;
    //pass pointer p by reference to function2
    function2(&p);
}
```

Double and Triple pointer. Multiple indirection

- And in the function1 you want to pass pointer "p" to another function - function2 by reference:

```
function2(&p);
```

- In this case the function2's parameter must be a pointer to pointer:

```
void function2(int** p2)
{
    (**p2)+= 10;
}
```

THANKS ☺