C7 Pointer

7.1 Getting Address

```
// use &

int x = 5;
cout << "The value of x is " << x << endl;
cout << "The address of x is " << &x << endl;   //using &


* = "go to the address stored in and look its value"
& = "Address of the operator"
```

7.2 Pointer Variable

1. The Core Difference

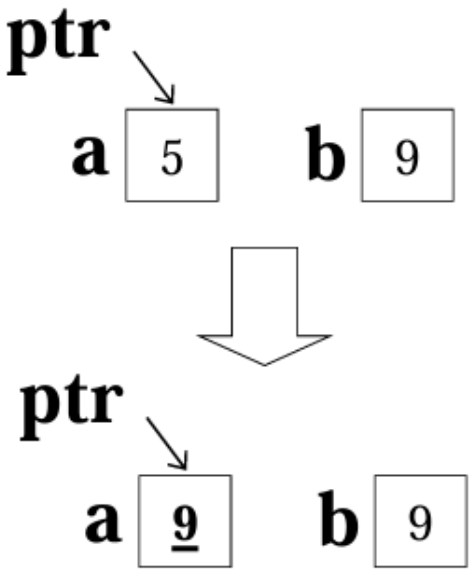
To understand this, imagine a house. The house has an **address** (e.g., 123 Maple St) and **people living inside** it.

Syntax	Name	What it represents	Example Value
`ptr`	**Pointer Variable**	The **memory address** of another variable.	`0x7ffee4b`
`*ptr`	**Dereferenced Pointer**	The **actual value** stored at that address.	`10`

```
int x =5,y=6;
int *ptr;   //now we deaclare the variable

ptr = &x;   //now the ptr point at x and store address
*ptr = y    //now thw ptr value change only means that x equal to y but the ptr still pointed at
```

Example (must know how to draw)

```
int a = 5, b = 9,
  *ptr = &a;



*ptr = b;
```



7.3 Relationship btw array and pointer

```
// 1. Array name as pointer
int val[4] = {1,2,3,4};

int *val;                    //auto set at val[0]
cout << *val << endl;        //Output = 1
-------------------------------------------------------
// 2. Pointer act as the array
int val[4] = {1,2,3,4};

int *valptr = val;
cout << *valptr << endl;     //Output = 1
cout << valptr[2] << endl;   //Output = 3
-------------------------------------------------------
// 3. Output for ptr and *ptr
int val[4] = {1,2,3,4};
int *valptr = val;

cout << valptr << endl;     //Here cout the address
cout << *valptr;            //Here cout 1
```

7.4 Pointers Arithmetic

```
int vals[] = {4,7,11};
int *valptr = vals;


// ++, --
valptr ++;
cout << *valptr << endl;         // Output = 7

// +,-
cout << *(valptr+2) << endl;     // Output = 11

// +=,-=
valptr +=2;
cout << *valptr << endl;         // Output = 11

//-
cout << valptr - val; << endl;
```

Operation	Action	What happen	Example
++/--	Step by step	Ptr change **permanently**	valptr++
+,-	Jump	Ptr does not change	*(valptr+2)
+=,-=	Jump	Ptr change **permanently**	valptr+=2

7.5 Initializing Pointers

- Can initialize at definition time:

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

- Cannot mix data types:

```
double cost;
int *ptr = &cost; // won't work
```

- Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```

7.6 Comparing Pointer

```
//Compare value
if (*ptr1 == *ptr2)

//Compare address
if (ptr1 == ptr2)
```

7.7 Pointers as Function Parameters

1) Pointer to Constant

If a data using **const** means that the value can't be change. Hence, if we gonna to **store it in a pointer** we have to use **Pointer-to-const**.

```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45, 12.85, 14.97, 10.35, 18.89}
// Here the values(18.55,17.45...) and the SIZE of array is locked
```

Declaring

```
double *ptr = payRates;             // Wrong
const double *ptr = payRates;       // True
```

- In this array, we can only **change the address** of pointer to see what value inside **instead of change value**.

Ex.

```
cout << *ptr;       // Output = 18.55
ptr ++;
cout << *ptr;       // Output = 17.45
```

2) Constant Pointer

A pointer that **cannot change address** after initialized but **change only the value**.

```
int *const ptr;
```

```
int num1 = 22;
int num2 = 50;
int *const ptr = &num1; //Initialized

*ptr = 30;          // True
ptr = &num2;       // False
```

3) Constant Pointer to Constant (combine)

- Rule 1 : **Cannot change the data** pointed to
- Rule 2 : **Cannot change the address** pointed to
- Rule 3 : Read from **right to left**

```
//Rule 3
const int *const ptr = &value; // ptr is a constant pointer that point to const int value


int value = 22;
int extra = 50;
const int * const ptr = &value;

*ptr = 30;              // Error
ptr = &extra;           // Error
```

4) Summary

Type	Can change address?	Can change data?
Regular	Yes	Yes
Pointer to Constant	Yes	No
Constant Pointer	No	Yes
Constant Pointer to Constant	No	No

7.8 Dynamic Memory Allocation

1) new

- make a valid address with garbage value

```cpp
// Three output address will be different but the output is the reference only
int main()
{
    int *a = new int;
    cout << new int << endl;    //Output = 0x100
    int *b = new int;
    cout << new int << endl;    //Output = 0x200
    int *c = new int;
    cout << new int << endl;    //Output = 0x300
}

int main()
{
    for (int i=0; i<numDays; i++)
    {
        cout << "Address is " << new int << endl;       //every value[i] the address will be di
        cout << "Enter a value: ";
        cin >> value[i];
    }
}


//normal variable

int* ptr = new int;  //create a valid address(0x100)
*ptr = 35;
cout << *ptr;       // Output = 35
cout << *new int;   // Output = 0   //create again a valid address(0x200)
```

```
//array (use hotel room and check out scenario)

cout << "Enter number of Day: ";
cin >> numDays;

int *values = new double[numDays];
// Here will create a Dynamic memory with numDays size and store in the value pointer

for (int i; i<numDays ; i++)
{
    cout << "Enter a value: ";
    cin >> value[i];
}
```

2) delete

```
//normal variable
int *ptr = new int;
delete ptr;

//array
int *value = new int[numDays];\
delete [] value;
```

Returning Pointers from Functions

```
// Rule 1: must not return a pointer to a **local variable** in function
int* returnLocal()
{
    int number = 100;
    return &number;      //number is destroy after out of function
}
```

```cpp
// Rule 2: must return to data that pass as **argument**
int* findLarger(int* a, int* b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
int main()
{
    int x = 10, y = 20;
    int* result = findLarger(&x, &y);
}


// Rule 3: must return to **dynamic memory**
int* createArray(int size)
{
    int* arr = new int[size];
    arr[0] = 50;
    return arr;
}
int main()
{
    int* myPtr = createArray(5);
    cout << myPtr[0];   // Output = 50
    cout << myPtr[3];   // Output = garbage value
    delete[] myPtr;
}
```