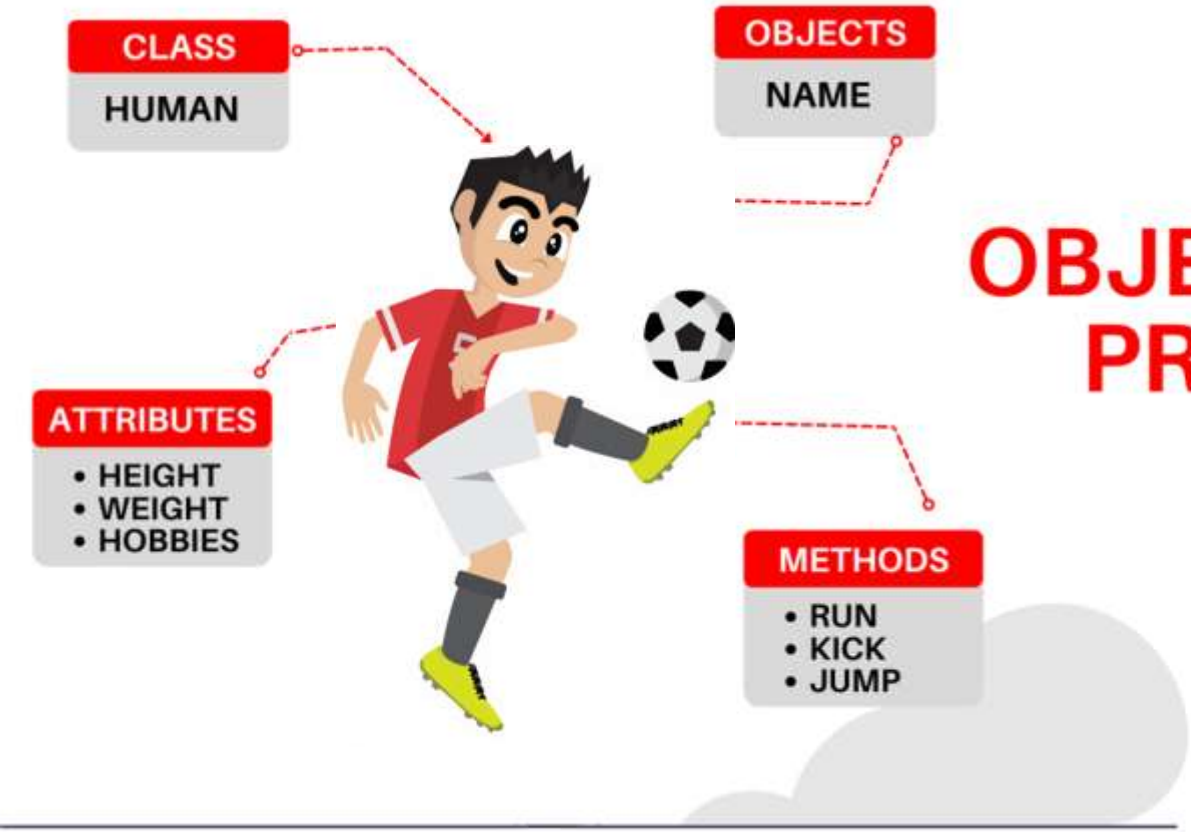
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

Summer 2023

Pir Sami Ullah Shah

Lecture # 3 Pointers and DMA

Casting pointers

- Pointers have types, so you cannot just do this

```
int *pi;
double *pd;
pd = pi;
```

- Even if they are both the same size, C++ will still give an error on implicit typecasting

Casting pointers

- C++ will let you change the type of a pointer with an explicit cast

```
int *pi; char *pd;
pd = (char*) pi;
```

Casting pointers

- C++ will let you change the type of a pointer with an explicit cast

```
int *pi = new int;
*pi = 65;
char *pd;
pd = (char*) pi;
cout<<*pd;
//prints A (ascii 65)
```

Casting pointers

```
int *pi = new int; //size 4 bytes
*pi = 131072;
```

```
 00000000  00000010  00000000  00000000
  byte 4    byte 3    byte 2    byte 1
```

```
short *pd;
pd = (short*) pi;
cout<<*pd;
//Prints 0 (short size 2 bytes)
```

Pointer Arithmatic

```
int x = 25;
int *intptr = &x;
```

```
intptr++;
```

What happens now?

Pointer address increases by 4 bytes because the pointer is of type int

Same as `intptr + 1`

Pointer Arithmatic

```
int x = 25;
int *intptr = &x;
```

```
intptr++;
```

e.g. address of x was 4000

`intptr` will now be 4004

Pointer Arithmatic

```
char x ='4';
char *charptr = &x;
```

```
charptr++;
```

What happens now?

Pointer address increases by 1 bytes because the pointer is of type char

Same as `charptr + 1`

Pointer Arithmetic

Expression	Assuming p is a pointer to a...	... and the size of *p is...	Value added to the pointer
p+1	char	1	1
p+1	short	2	2
p+1	int	4	4
p+1	double	8	8
p+2	char	1	2
p+2	short	2	4
p+2	int	4	8
p+2	double	8	16

Pointer Arithmatic

```
int x = 25;
int *intptr = &x;
```

Can we increment the value of x using the pointer?

```
(*intptr)++;
```

- * Gets the value the pointer points at.
- Value of **x** becomes **26**
- Brackets needed because * has lower precedence

Pointer Arithmetic

Only two types of arithmetic operations allowed:

1) **Addition** : only integers can be added
2) **Subtraction**: only integers be subtracted

I. pointer + integer (ptr+1) ✓
II. integer + pointer (1+ptr) ✓
III. pointer + pointer (ptr + ptr) ✗
IV. pointer – integer (ptr – 1) ✓
V. integer – pointer (1 – ptr) ✗
VI. pointer – pointer (ptr – ptr) ★
VII. compare pointer to pointer (ptr == ptr) ✓
VIII. compare pointer to integer (1 == ptr) ✗
IX. compare pointer to 0 (ptr == 0) ✓
X. compare pointer to NULL (ptr == NULL) ✓

Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
- Array name is a constant pointer that stores the starting address of an array
- All arrays elements are placed in the consecutive locations

The Relationship Between Arrays and Pointers

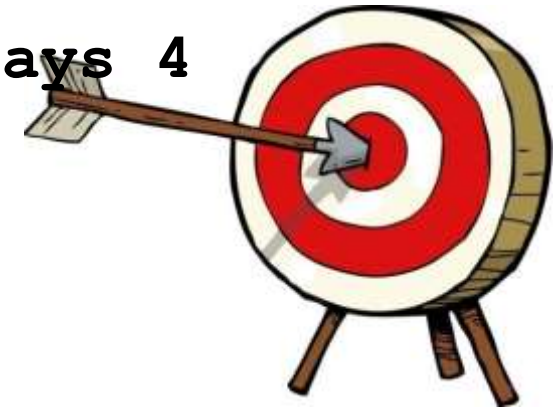
- Array name is the starting address of array

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

starting address of vals: 0x4a00

4	7	11

```
cout << vals;              // displays
0x4a00
cout << vals[0];       // displays 4
```



The Relationship Between Arrays and Pointers

- Array name can be used as a pointer:

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

- Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

Array Access

`vals[i]` is equivalent to `*(vals + i)`

- No bounds checking performed on array access, whether using array name or a pointer

Array Access

- We can access array elements using array index

```
int num = vals[2]; //value assignment
```

- We can access array elements using pointers

```
int* p = vals;          //address assignment
```

- List is an address, no need for &
- The pointer **p** will contain the address of the first element of array List.
- Element vals[2] can be accessed by *(p + 2)

Array Access

- Array elements can be accessed in many ways:

Array access method	Example
array name and `[]`	`vals[2] = 17;`
pointer to array and `[]`	`valptr[2] = 17;`
array name and subscript arithmetic	`*(vals + 2) = 17;`
pointer to array and subscript arithmetic	`*(valptr + 2) = 17;`

Pointers in Expressions

Given:

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

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

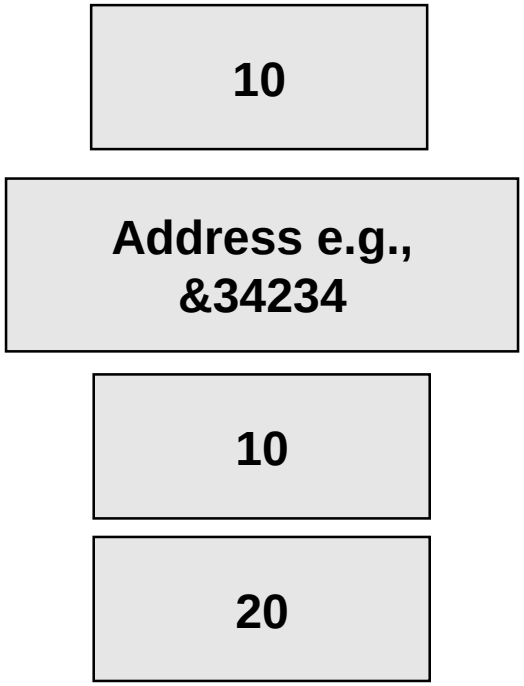
```
cout << *(valptr+1); //displays 7
cout << *(valptr+2); //displays 11
```

Must use （ ） as shown in the expressions

- This does not change the value of the pointer

Relationship between Arrays and Pointers

```
void main()
{
   int numbers[] = {10,20,30,40,50};
   cout<< numbers[0] <<endl;
   cout<< numbers <<endl;
   cout<< *numbers <<endl;
   cout<< *(numbers+1);
}
```



Pointer Arithmetic

- Operations on pointer variables:

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

`++, --`	`valptr++; // points at 7`
	`valptr--; // now points at 4`

`+, -` **(pointer and `int`)**	`cout << *(valptr + 2); // 11`

`+=, -=` **(pointer and `int`)**	`valptr = vals; // points at 4`
	`valptr += 2;  // points at 11`

`-` **(pointer from pointer)**	`cout << valptr-vals; // difference`
	`//(number of ints) between valptr`
	`// and val`

```
7     const int SIZE = 8;
8     int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9     int *numPtr;   // Pointer
10    int count;     // Counter variable for loops
11
12    // Make numPtr point to the set array.
13    numPtr = set;
14
15    // Use the pointer to display the array contents.
16    cout << "The numbers in set are:\n";
17    for (count = 0; count < SIZE; count++)
18    {
19       cout << *numPtr << " ";
20       numPtr++;
21    }
22
23    // Display the array contents in reverse order.
24    cout << "\nThe numbers in set backward are:\n";
25    for (count = 0; count < SIZE; count++)
26    {
27       numPtr--;
28       cout << *numPtr << " ";
29    }
```

```
7      const int SIZE = 8;
8      int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9      int *numPtr;   // Pointer
10     int count;     // Counter variable for loops
11
12     // Make numPtr point to the set array.
13     numPtr = set;
14
15     // Use the pointer to display the array contents.
16     cout << "The numbers in set are:\n";
17     for (count = 0; count < SIZE; count++)
18     {
19        cout << *numPtr << " ";
20        numPtr++;
21     }
22
23     // Display the array contents in reverse order.
24     cout << "\nThe numbers in set backward are:\n";
25     for (count = 0; count < SIZE; count++)
26     {
27        numPtr--;
28        cout << *numPtr << " ";
29     }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

Arrays and Pointers

Array name is the starting address of the array

```
int A[25];
int *p; int i=0, j=0;
```

`p = A;`

`p` points to `A[0]`

`p + i` points to `A[i]`
`&A[j]` is also the same as `p+j`
`A[j]` is also the same as `*(p+j)`

Arrays Decay to Pointers

- We learned to pass arrays as arguments to functions

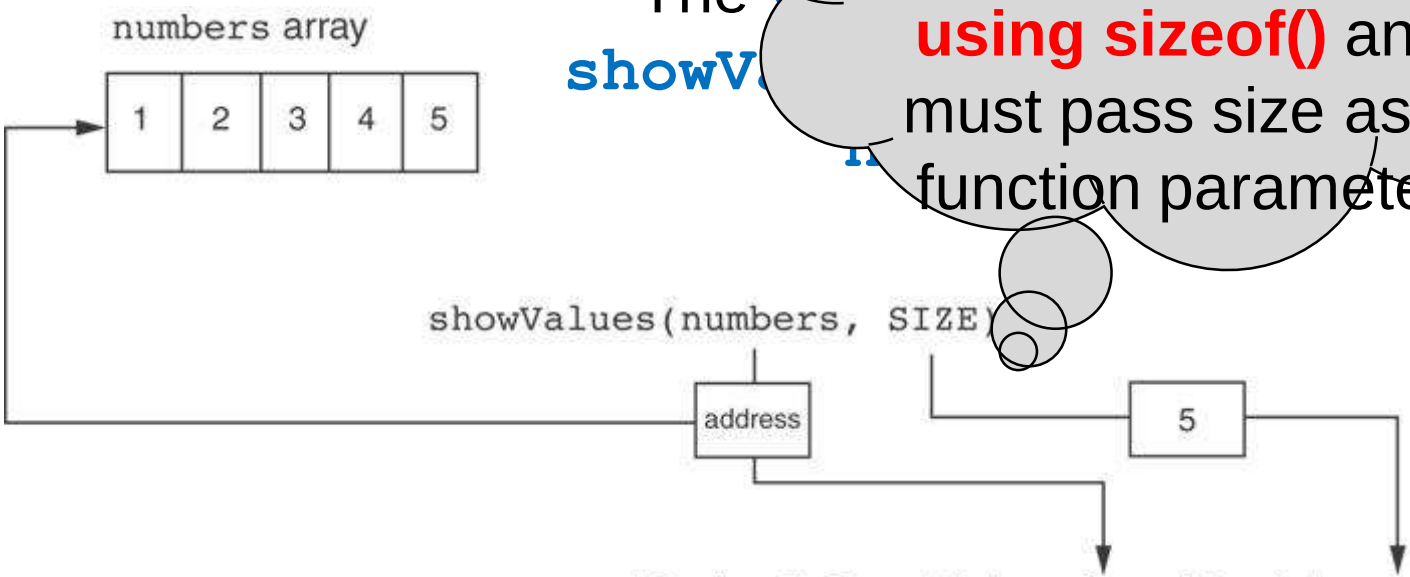
- For example, suppose we use this statement to pass the array **`numbers`** to the **`showValues`** function:

```
showValues(numbers, SIZE);
```

Arrays Decay to Poin

Arrays passed by value **decay into a pointer.**
Cannot find size using sizeof() and must pass size as a function parameter

The **showV**



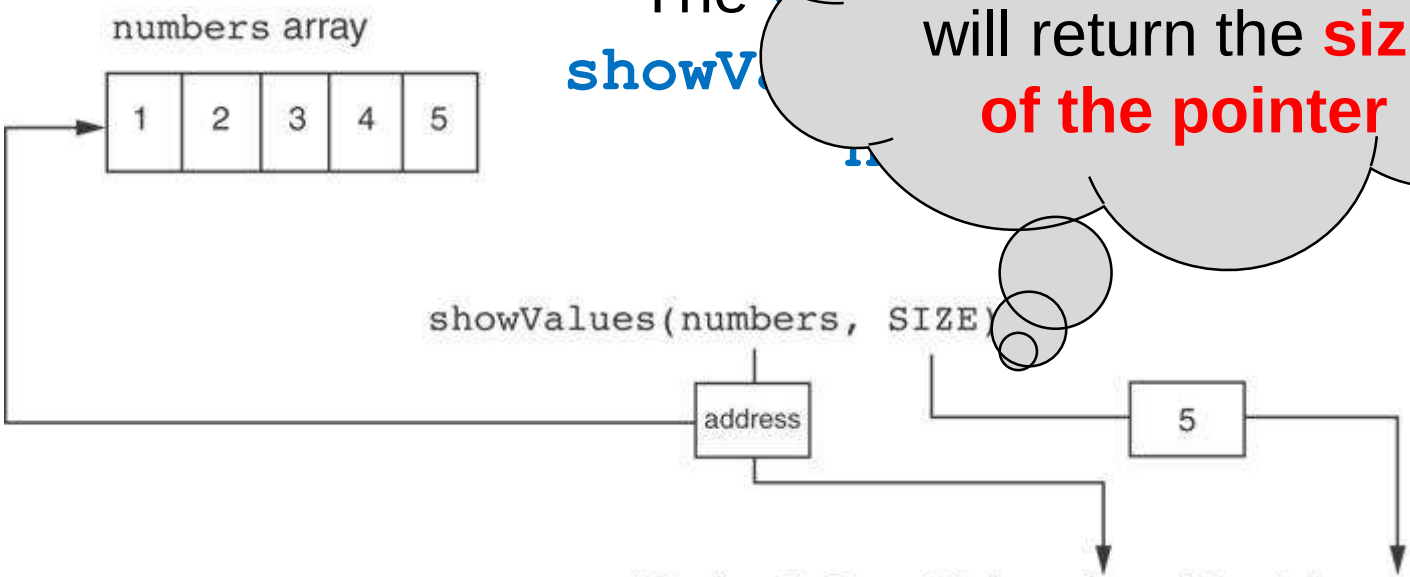
C++ automatically stores the address of **numbers** in the **values** parameter.

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

Arrays Decay to Poin

Using sizeof() on **an array passed by value to a function** will return the **size of the pointer**

The **showV**



C++ automatically stores the address of **numbers** in the **values** parameter.

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

Comparing Pointers

- Relational operators (<, >=, etc.) can be used to compare addresses in pointers

- Comparing addresses in pointers is not the same as comparing contents pointed at by pointers:

```
if (ptr1 == ptr2) // compares addresses
if (*ptr1 == *ptr2) // compares contents
```

Comparing Pointers

- If one address comes before another address in memory, the first address is considered less than the second address.

- In an array, elements are stored in consecutive memory locations, E.g., address of Arr[2] will be smaller than the address of Arr[3] etc.

Accessing 1-Dimensional Array Using Pointers

- We know, **Array name** denotes the **memory address**
  - Example:

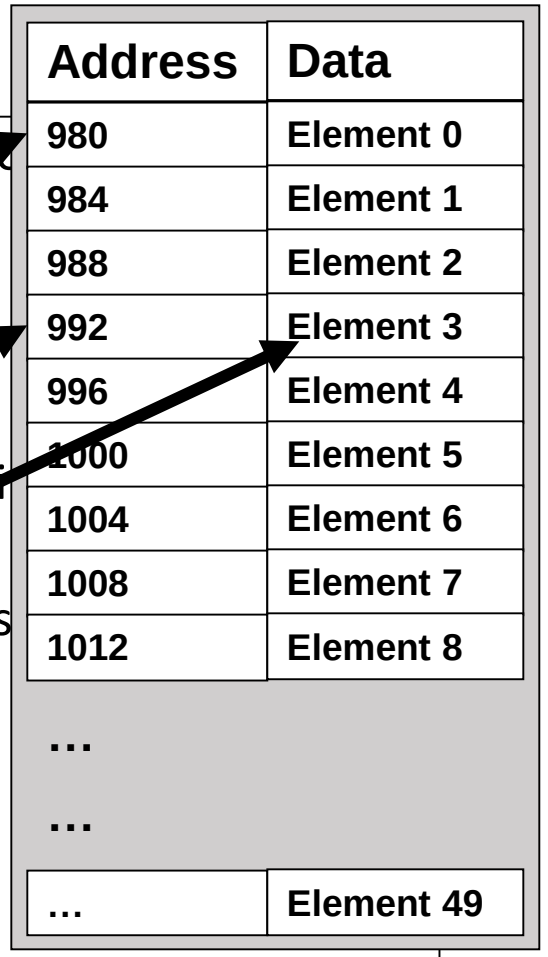
```
int List [ 50 ];
int *Pointer;
Pointer = List;
```

- Other slots of the **Array (List [50])** can be accessed usi **Arithmetic operations** on **Pointer**.
- For example the address of **(element 4th)** can be acces

```
int *Value = Pointer + 3;
```

- The value of **(element 4th)** can be accessed using:-

```
int Value = *(Pointer + 3);
```

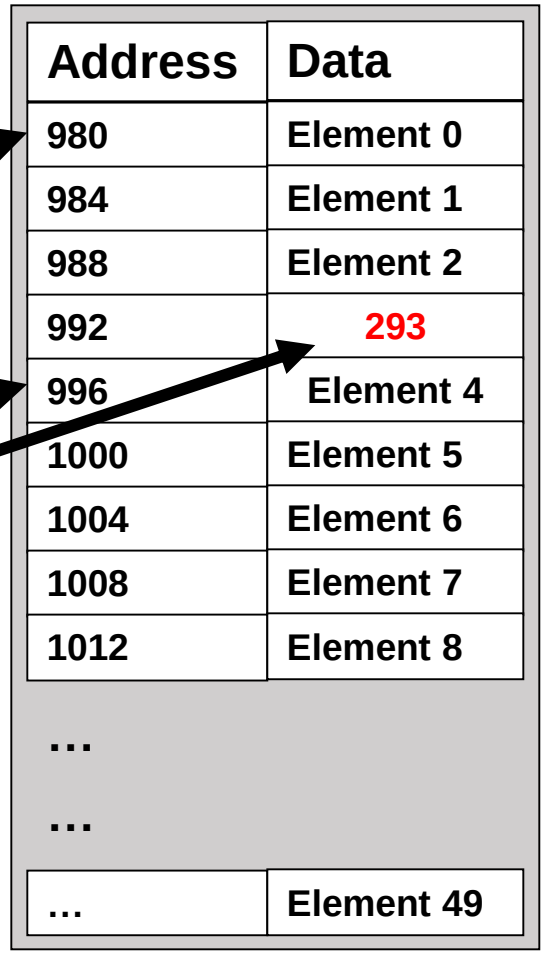


Accessing 1-Dimensional Array

```
....
....
    int List [ 50 ];
    int *Pointer;
    Pointer = List; // Address of first Element

    int *ptr;
    ptr = Pointer + 3; // Address of 4th Element
    *ptr = 293; // 293 value store at 4th element
    address

}
```



Accessing 1-Dimensional Array

We can access all element of List [50] using Pointers and a for loop

```
int List [ 50 ];
int *Pointer;
Pointer = List;
for ( int i = 0; i < 50; i++ )
{
      cout << *Pointer;
      Pointer++; // Address of next element
}
```

Address	Data
980	Element 0
984	Element 1
988	Element 2
992	Element 3
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
…	
…	
…	Element 49

This is Equivalent to

```
for ( int index = 0; index < 50; index++ )
      cout <<  Array [ index ] ;
```

Accessing 2-Dimensional Array

- **Note that the statements**

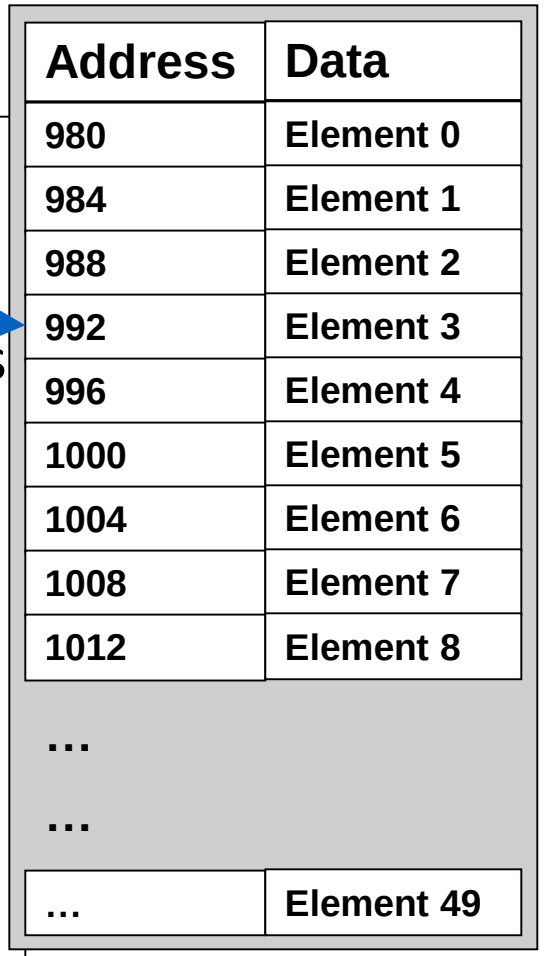
```
int *Pointer;
Pointer = &List [3];
```

- represents that we are accessing the address of **4th slot**

- In 2-Dimensional array the statements

```
int List[ 5 ][ 6 ];
int *Pointer;
Pointer = &List [3];
```

Represents that we are accessing the address of **4th row** and **1st column**

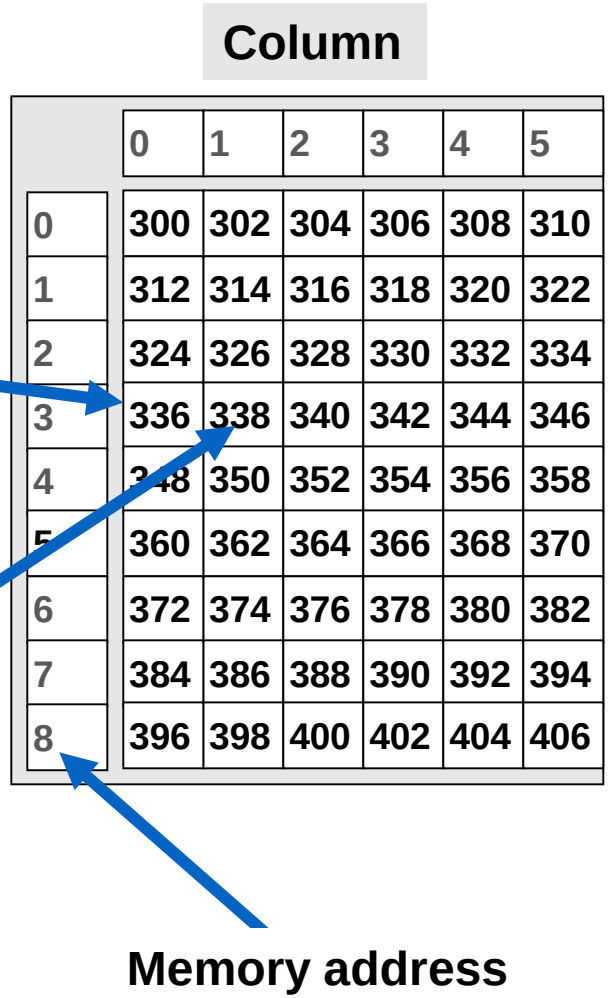


Address	Data
980	Element 0
984	Element 1
988	Element 2
992	Element 3
996	Element 4
1000	Element 5
1004	Element 6
1008	Element 7
1012	Element 8
…	
…	
…	Element 49

Accessing 2-Dimensional Array

- **int List [9] [6];**
- **int *ptr;**
- **ptr = &List [3];**

- To access the address of 4th row 2nd column:
  - **ptr++; // address of 4th row 2nd column**
  - Equivalent to **List [3][1] ;**



Dynamic Memory Allocation

- Can also use **new** to allocate array:

- ```
  int SIZE = 25;
  double *arrayPtr = new double[SIZE];
  ```

- Can then use `[]` or pointer arithmetic to access array:

  ```
  for(int i = 0; i < SIZE; i++)
       arrayptr[i] = i;
  ```

  or

  ```
  for(int i = 0; i < SIZE; i++)
       *(arrayptr + i) = i;
  ```

- Program will terminate if not enough memory available to allocate
  ```

```
// Dynamically allocate memory for 1d array
int N, i;
N = 15;

// Dynamically allocate memory of size N
int *array = new int[N];

// Assign values of allocated memory
for(i = 0; i < N; i++){
      cin >> *(array+i); }

for(i = 0; i < N; i++){
      cout<<array[i]<<" "; // is equal to cout<<*(array+i);

delete [] array; //deallocate memory
```

```
// Dynamically allocate memory for 1d array
int N, i;
cout<<"Enter size of array: ";
cin >> N;  //get size from user

// Dynamically allocate memory of size N
int *array = new int[N];

// Assign values of allocated memory
for(i = 0; i < N; i++){
      cin >> *(array+i); }

for(i = 0; i < N; i++){
      cout<<array[i]<<" "; // is equal to cout<<*(array+i);
delete [] array; //deallocate memory
```

Creating Dynamic 2D Arrays

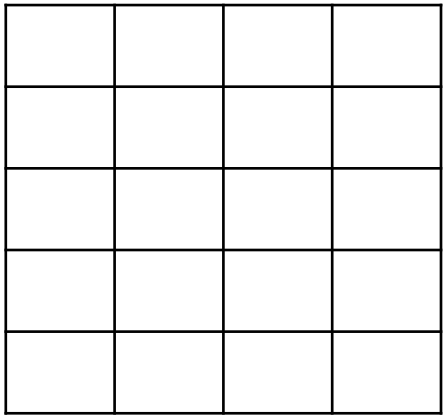
- Two basic methods:
  1. Using Only one Pointer/One array
  2. Using an Array of Pointers

Dynamic two dimensional arrays

1. **Using One Pointer/One array**
   - Total elements in a 2D Array:

     M * N (i.e., rows * cols)

5 rows * 4 columns = 20 elements



Approach:

- Allocate 20 elements using dynamic allocation
- Use a pointer to point and access those items

Accessing 2-Dimensional Array

- Access static array elements using a pointer

```
int arr[9]={ 1,2,3,
                4,5,6,
             7,8,9};
int *p = &arr[0];

*(p +(TotCols * rowIndx)+ colIndx);

//equivalent to arr[1][1]
cout<<*(p +(3 * 1)+ 1); //prints 5

//equivalent to arr[2][2]
cout<<*(p +(3 * 2)+ 2); //prints 9
```

Accessing 2-Dimensional Array

- Access dynamic array elements using a pointer

```
int *p = new int[9];
for(int i=0;i<9;i++){
     p[i] = i+1;
}
*(p +(TotCols * rowIndx)+ colIndx);

//equivalent to arr[1][1]
cout<<*(p +(3 * 1)+ 1); //prints 5

//equivalent to arr[2][2]
cout<<*(p +(3 * 2)+ 2); //prints 9
```

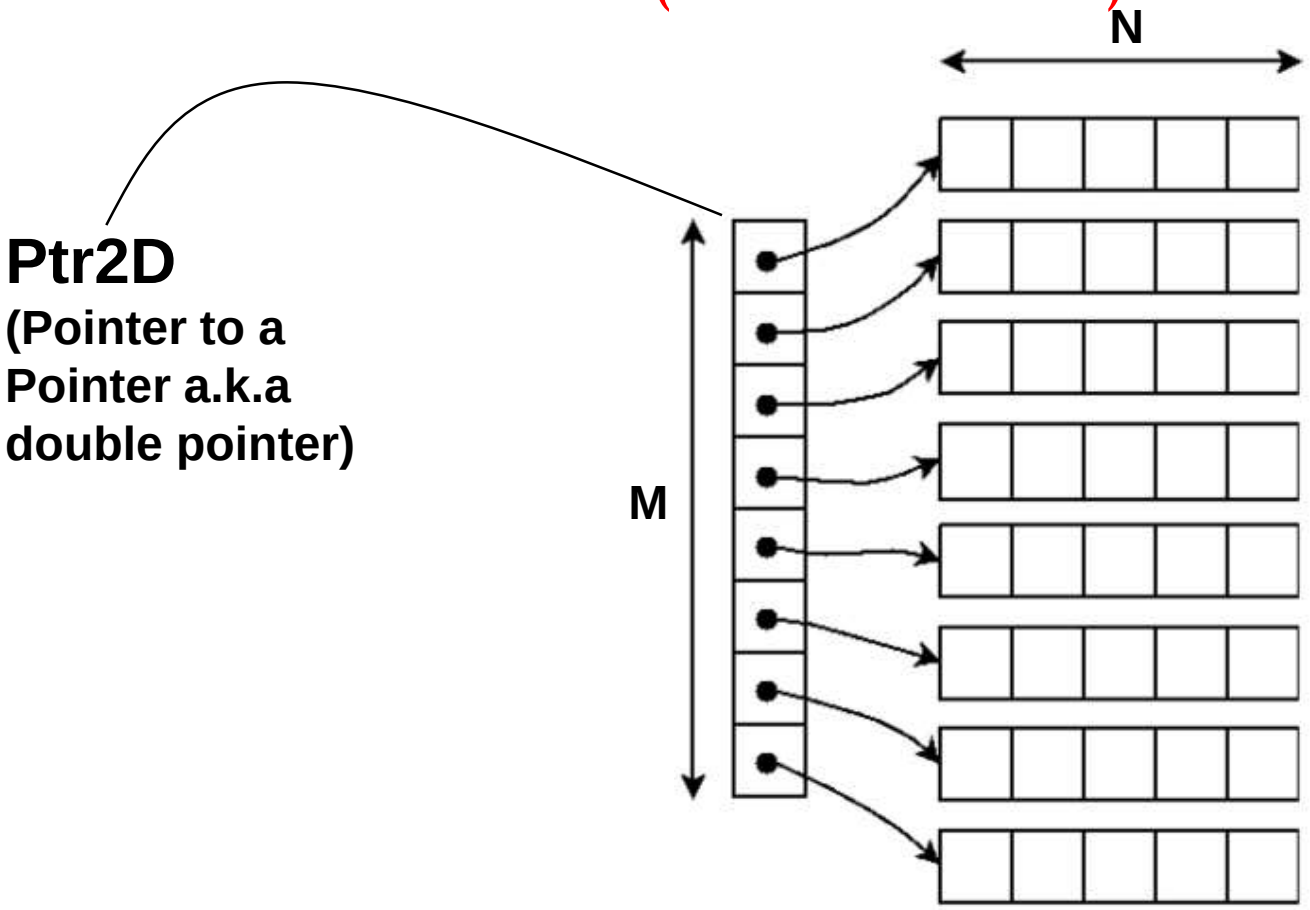
Dynamic 2D Arrays

```cpp
#include <iostream>

// M x N matrix
#define M 4
#define N 5

// Dynamically Allocate Memory for 2D Array in C++
int main()
{
    // dynamically allocate memory of size M*N
    int* A = new int[M * N];

    // assign values to allocated memory
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            *(A + i*N + j) = rand() % 100;

    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
            std::cout << *(A + i*N + j) << " ";

        std::cout << std::endl;
    }

    // deallocate memory
    delete[] A;

    return 0;
}
```

Dynamic 2D Array – Double Pointer

2. **Using a Pointer that points to an Array of Pointer**
   - Total elements in a 2D Array:

M * N i.e (rows * columns)



Dynamic 2D Array – Double Pointer

```
int M=3, N=4;


// Dynamically create
int **arr = new int*


// Dynamic allocat                each row
for(int i = 0; i < M; i
       arr[i] = new int[N];
       }
// deallocate memory
for(int i = 0; i < M; i++){
       delete[] arr[i]; }
delete[] arr;
```

arr Start of array of pointers

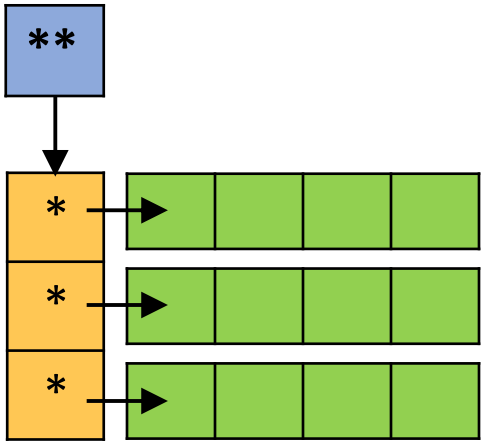
*arr First Address pointed by first row (sub array)

*(*arr) First value of first array

(*arr)++ Move to next address in the first

ow (second array

Can we vary the size of each column in Dynamic 2D Array (using double pointer)



Dynamic 2D Array
(Varying Row Size)

```cpp
// Dynamically Allocate Memory for 2D Array in C++
int main()
{
    // dynamically create array of pointers of size M
    int** A = new int*[M];

    // dynamically allocate memory of size N for each row
    for (int i = 0; i < M; i++)
        A[i] = new int[N+i];

    // assign values to allocated memory
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N+i; j++)
            A[i][j] = rand() % 100;

    // print the 2D array
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N+i; j++)
            std::cout << A[i][j] << " ";

        std::cout << std::endl;
    }

    // deallocate memory using delete[] operator
    for (int i = 0; i < M; i++)
        delete[] A[i];

    delete[] A;

    return 0;
}
```

Output

```
83 86 77 15 93
35 86 92 49 21 62
27 90 59 63 26 40 26
72 36 11 68 67 29 82 30
```

C - Strings

#include <string> VS #include <string.h>

#include <cstring>

String class header

C language file

strings as char arrays

Stores

C - Strings

- In C++, a C-string is a sequence of characters stored in consecutive memory locations (array), terminated by a null character.

```
string name= “Bailey”;
```

B	a	i	l	e	y	\0
0	1	2	3	4	5	6

\0 is the NULL character

Character Arrays

- Be careful, every character array is not a string, however the behavior is similar

```
char name[7]= {‘B’,’a’,’i’,’l’,’e’,’y’,’y’};
```

B	a	i	l	e	y	y
0	1	2	3	4	5	6

Character Arrays

- Character array initialized with string literal

```
char name[7]="Bail"; //always stores /0 at the end
```

- If initializer values of a char array are less than array size, remaining filled with null character – can treat array like a string

B	a	i	l	\0	\0	\0
0	1	2	3	4	5	6

Character Arrays

- Character array initialized with string literal

```
char name[]="Bailey"; //always stores /0 at the end
```

- Array size not specified, char array size = number of characters in string literal + 1 (for \0)

B	a	i	l	e	y	\0
0	1	2	3	4	5	6

Character Arrays

- Be careful, every character array is not a string

```
char name[4]="Bail"; //always stores /0 at the end
```

- Array size is 4, characters to store are also 4, no space for NULL character – ERROR!!

B	a	i	l	\0
0	1	2	3	4

Character Arrays

```
char name[]="Bail"; //always stores /0 at the end
cout<< name; //what happens now?
//Prints Bail
```

- The char array is treated as a string and cout prints characters from the starting address of the array till the null character

B	a	i	l	\0
0	1	2	3	4

Character Arrays

```
char name[4] = {‘B’,’a’,’i’,’l’};
cout<< name; //what happens now?
//Prints Bail
```

- The char array is treated as a string and cout prints characters from the starting address of the array till the end

B	a	i	l
0	1	2	3

Character Arrays

```
char name[4] = {‘B’,’a’,’i’,’l’};
cout<< (name+1); //what happens now?
//Prints ail
```

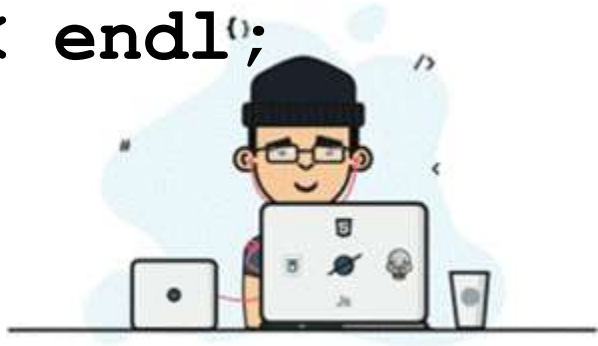
- The char array is treated as a string and cout prints characters from the starting address till the end

B	a	i	l
0	1	2	3

Character Arrays and Char Pointer

```
char arr[] = "****";
int i = 1;
for (; i < 3; i++) {
    cout << (arr+i) << endl;

}
for (; i >= 1; i--) {
    cout << (arr + i) << endl;

}
```

```
***
**
*
**
***
```



Character Arrays and Char Pointer

```
char name[]="Bail"; //always stores /0 at the end
char *ptr = name;
cout<< ptr; //what happens now?
//Prints Bail
```

- cout prints characters from the pointer address till the null character

B	a	i	l	\0
0	1	2	3	4

Character Arrays and Char Pointer

```
char name[]="Bail"; //always stores /0 at the end
char *ptr = name;
cout<< ptr;
ptr++;
```

- cout prints characters from the starting address of the array till the null character, print ail

B	a	i	l	\0
0	1	2	3	4

Character Arrays and Char Pointer

```
void match(char* str, char ch, int size){
    for(int i=0;i<size;i++){
        if(*str == ch){
            cout << str;
            break;
     }
        else
            str++;    }}
void main() {
    char arr[] = "some string";
    match(arr,'e',sizeof(arr));
}
```

e string

Character Arrays and Char Pointer

```
// Copying string using Pointers
char* str1 = "Self-conquest is the greatest
 victory.";
char str2[80]; //empty string
char* src = str1;
char* dest = str2;

while( *src ) //until null character,
      *dest++ = *src++; //copy from src to dest

*dest = '\0'; //terminate dest
cout << str2 << endl; //display str2
```

Cin vs Getline

Input can be performed by the cin object.

```
const int SIZE = 21;
char name[SIZE];
cin >> name;
```

This code allows the user to enter a string (with no whitespace characters) into the name array

Cin vs Getline

C-string input can be performed by the getline function of the cin object.

```
const int SIZE = 21;
char name[SIZE];

cin.getline(name, SIZE);
```

This code allows the user to enter a string
whitespace characters) into the name arra

Cin vs Getline

```
const int SIZE = 21;
char name[SIZE];
cin.getline(name, SIZE);
```

- first argument tells getline where to store the string input.

- second argument indicates maximum length of the string, including the null terminator

Cin vs Getline

```
const int SIZE = 21;
char name[SIZE];
cin.getline(name, SIZE);
```

In this example, the SIZE constant is equal to 21,
cin will read 20 characters, or until the user presses the Enter key, whichever comes first.

cin will automatically append the null terminator to the end of the string.

Cin vs Getline

```
const int SIZE = 21;
char name[SIZE];
cin.getline(name, SIZE);
```

If you are using cin and cin.getline together, you would have to clear the input buffer inbetween, to remove the null character inserted by cin

Use `cin.ignore();` for this.

Cin vs Getline

If you are using cin and cin.getline together, you would have to clear the input buffer inbetween

Use `cin.ignore();` for this.

```
int x, arr[3];
cin>>x;
cin.ignore(); //clears the input stream buffer
cin.getline(arr,3);
```

Pointers to Constants

```
const int a = 5;
int *foo = &a;
```

ERROR!!

If we want to store the address of a constant in a pointer, then we need to store it in a pointer-to-const

```
const int *foo= &a;
```

Pointers to Constants

This is a READ-ONLY pointer.

```
const int *foo= &a;

*foo = 26; //ERROR
```

Pointers to Constants

```
const int a = 5;
```

```
const int *foo= &a;
```

Be careful:

const is applied to the thing that foo points to, not foo itself

The pointer itself is **not constant**

Pointers to Constants

```
const int a = 5;
const int *foo= &a;
```

The data type

foo points to

Because foo is a pointer to a const, the compiler will not allow us to write code that changes the thing that foo points to.

Pointers to Constants

```
const int a = 5;
const int *foo= &a;

*foo = NULL;
```

- Like standard pointers you can also set a pointer to constant NULL

Pointers to Constants

```
const int a = 5;
const int *foo= &a;

*foo = new int;
```

- Like standard pointers you can also use a pointer to constant for dynamic memory allocation

Pointers to Constants

- Example: Suppose we have the following definitions:

```
const int SIZE = 6;
const double payRates[SIZE] =
      { 18.55, 17.45, 12.85,
        14.97, 10.35, 18.89 };
```

- In this code, **`payRates`** is an array of constant doubles.

Pointers to Constant

- Suppose we wish to pass t
  function? Here's an example o

A pointer to const **parameter** can also receive the **address of a non-constant** item.

```
void displayPayRates(const double *rates, int size)
{
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

The parameter, rates, is a pointer to `const double`.

Declaration of a Pointer to Constant

The asterisk indicates that `rates` is a pointer.

```
const double *rates
```

This is what `rates` points to.

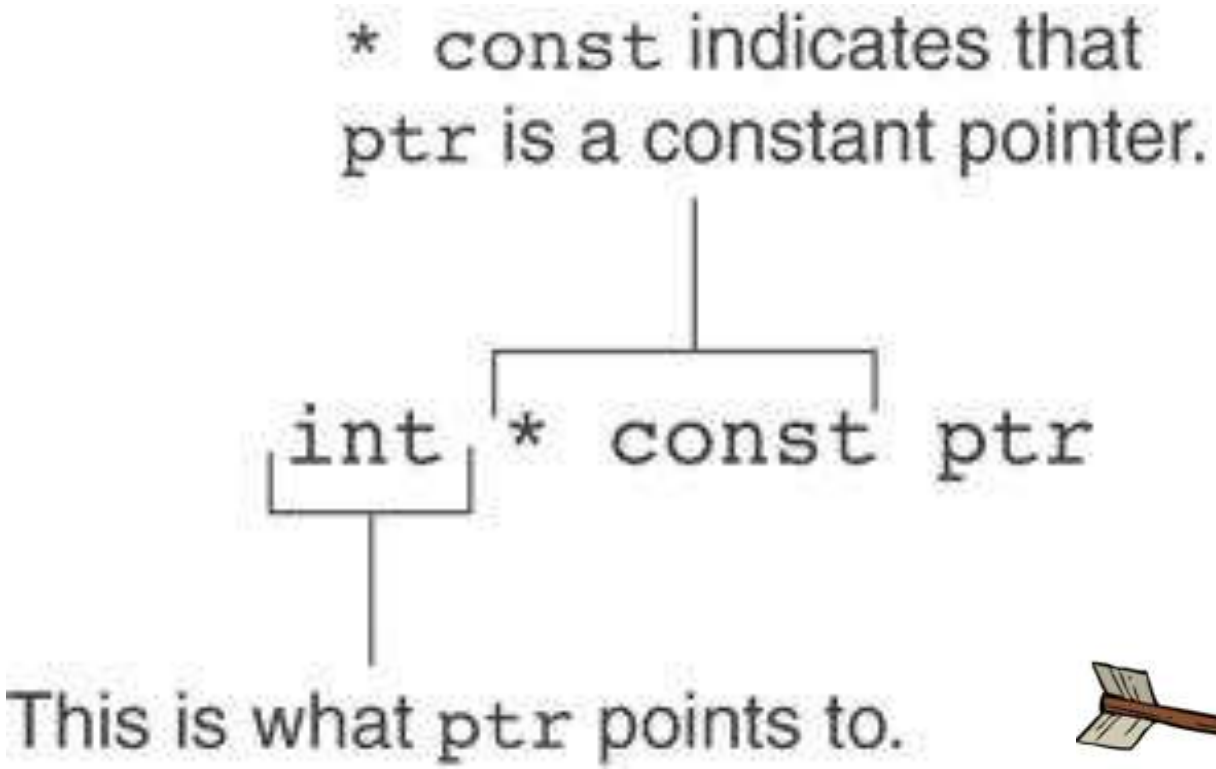
Constant Pointers

- A constant pointer is a pointer that is initialized with an address, and cannot point to anything else

- Example

```
int value = 22;
int * const ptr = &value;
```

Constant Pointers

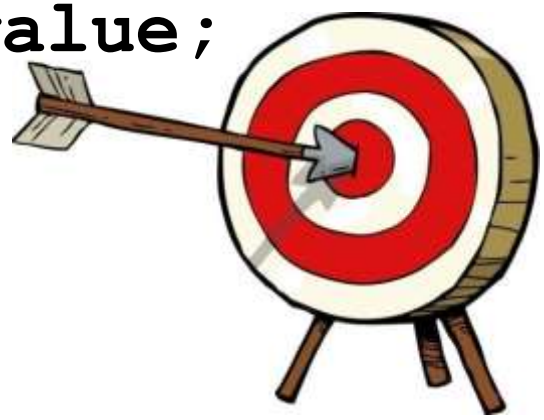


Constant Pointers to Constants

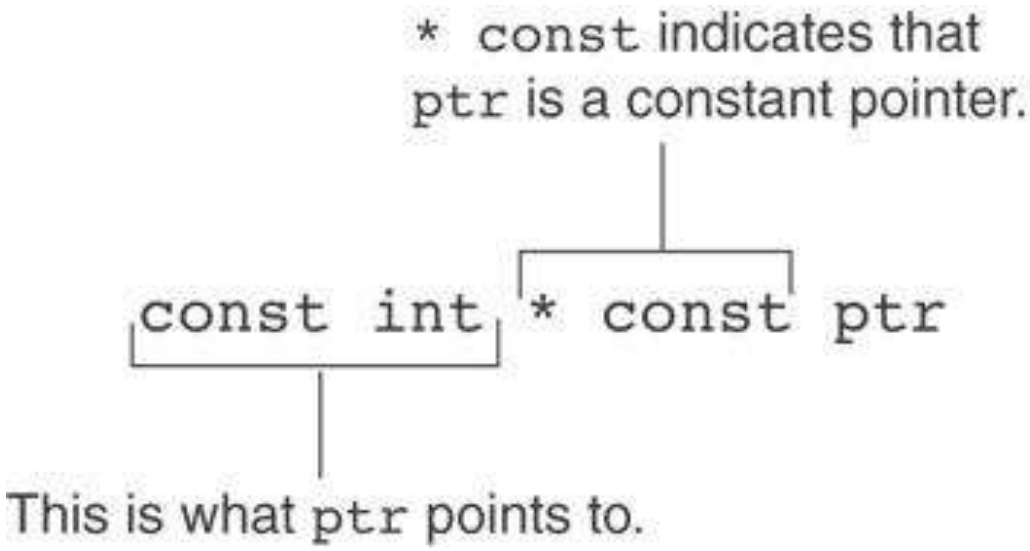
- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

```
const int value = 22;
const int * const ptr = &value;
```



Constant Pointers to Constants



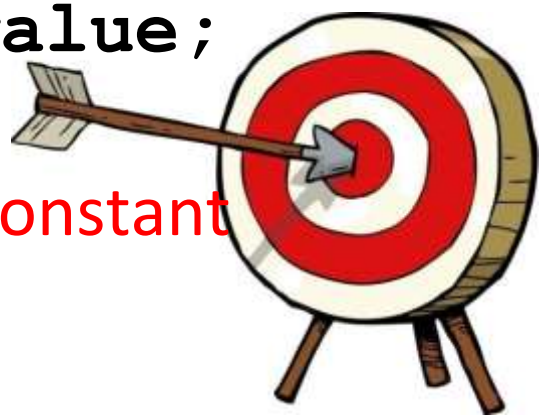
Constant Pointers to Constants

- A constant pointer to a constant is:
  - a pointer that points to a constant
  - a pointer that cannot point to anything except what it is pointing to

- Example:

```
int value = 22;
const int * const ptr = &value;
```

Pointer to const can also point to non-constant



Pointers as Function Parameters

- A pointer can be a function parameter

```
     void getNum(int
*ptr){//definition
              ……
     }
```

- Use pointers as function parameters and addresses as arguments
- Pass address using **&** operator

```
int num = 7;
getNum(&num); //function call
```

Pointers as Function Parameters

- A pointer can be a function parameter

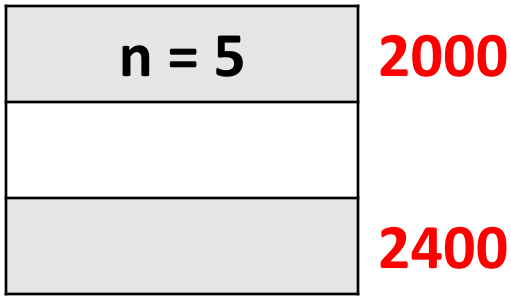
```
void getNum(int *ptr){//definition
        ……
}
```

- Can also use pointer as an argument

```
int num = 7; int *p = &num;
getNum(p); //function call
```

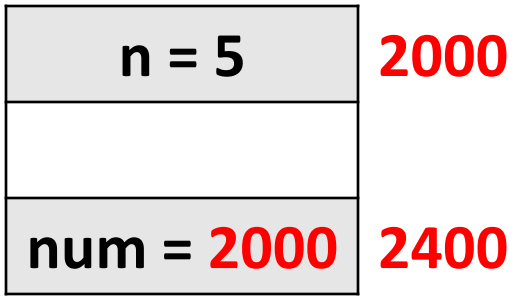
Pointers as Function Parameters

```
void func(int  *num)
{
     *num = 10;
}
void main()
{    int n = 5;
     cout<<"Before: n = "<<n<<endl;
     func(&n);
     cout<<"After: n = "<<n<<endl;
}
```



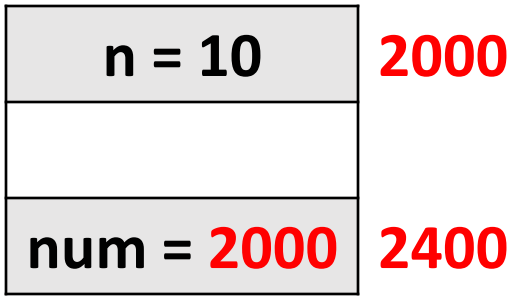
Pointers as Function Parameters

```
void func(int  *num)
{
     *num = 10;
}
void main()
{    int n = 5;
     cout<<"Before: n = "<<n<<endl;
     func(&n);
     cout<<"After: n = "<<n<<endl;
}
```



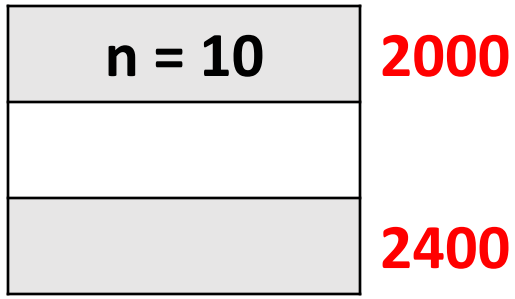
Pointers as Function Parameters

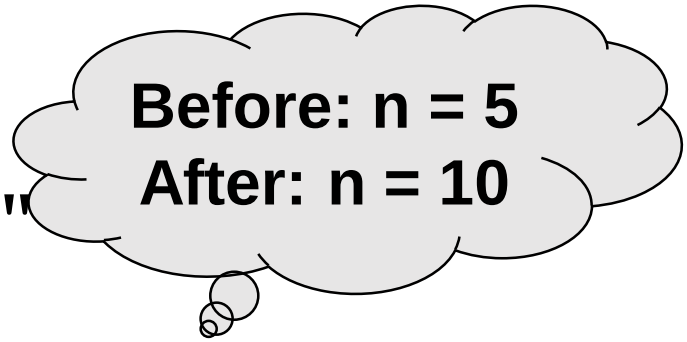
```
void func(int  *num)
{
     *num = 10;
}
void main()
{    int n = 5;
     cout<<"Before: n = "<<n<<endl;
     func(&n);
     cout<<"After: n = "<<n<<endl;
}
```



Pointers as Function Parameters

```
void func(int  *num)
{
     *num = 10;
}
void main()
{    int n = 5;
     cout<<"Before: n = "
     func(&n);
     cout<<"After: n = "<<n<<endl;
}
```





Pointers as Function Parameters

- Do not use **&** while passing an array
- Array name is already an address

```
void compDouble(int* Ar)
{
     ..............
}
void main()
{          int
Arr[10]={0,1,2,3,4,5,6,7,8,9};
           compDouble(&Arr); //ERROR
}
```

Pointers as Function Parameters

- Do not use **&** while passing an array
- Array name is already an address

```
void compDouble(int* Ar)
{
     ……….
}
void main()
{          int
Arr[10]={0,1,2,3,4,5,6,7,8,9};
          compDouble(Arr); //Correct way
}
```

Pointers as Function Parameters

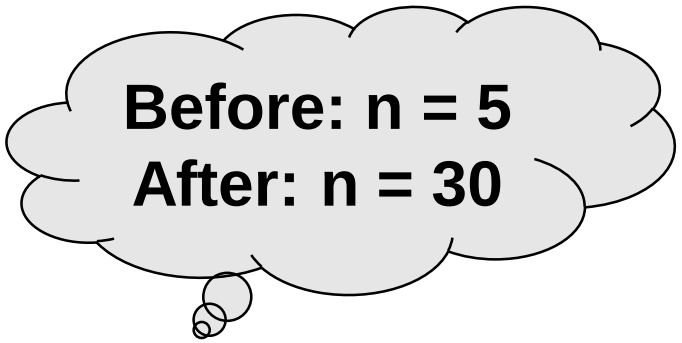
```
void compDouble(int* Ar)
{
    for(int i=0;i<10;i++)
          {     *Ar=(*Ar)*2;
                 Ar++;
          }  }
void main()
{         int
Arr[10]={0,1,2,3,4,5,6,7,8,9};
          compDouble(Arr);
}
```

Pointers as Function Parameters

- A function with a pointer parameter, called with pointer argument
  - **Case 1:** If you change the value at the address the pointer points to, the change will remain even after the function has returned
  - **Case 2:** If you change the value of the pointer (address), the change will not be retained after the function ends
    - This is because the function has a copy of the pointer

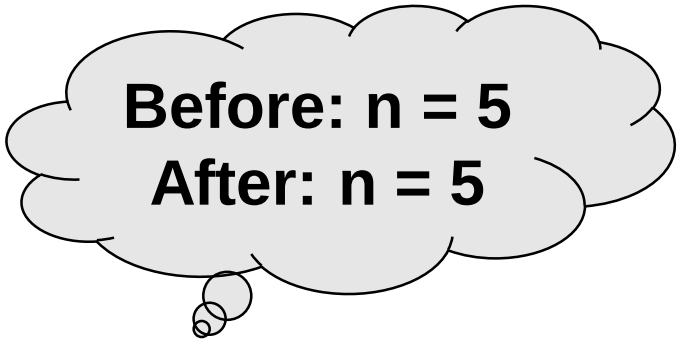
Case 1 Example

```
void func(int  *num)
{
     *num = 30;
}
void main()
{    int n = 5; int *p = &n;
     cout<<"Before: n = "<<n<<endl;
     func(p);
     cout<<"After: n = "<<n<<endl;
}
```



Case 2 Example

```
int gl = 93;
void func(int  *num)
{
     num = &gl;
}
void main()
{    int n = 5; int *p = &n;
     cout<<"Before: n = "<<n<<endl;
     func(p);
     cout<<"After: n = "<<n<<endl;
}
```



Pointers as Function Parameters

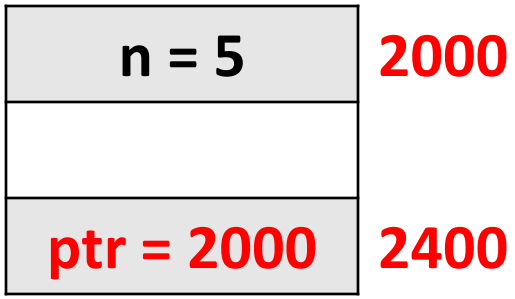
- Two ways to change the value of a pointer (address) in a function with a pointer parameter

1. **Pointer to pointer** (double pointer)

1. **Reference to pointer** (aliasing)

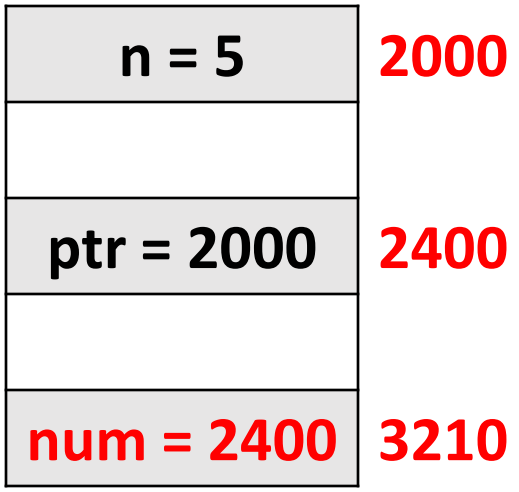
Pointer to Pointer Function Parameter

```
int gl = 95;
void func(int **num)
{
     *num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(&ptr);
     cout<<"After = "<< *ptr <<endl;
}
```



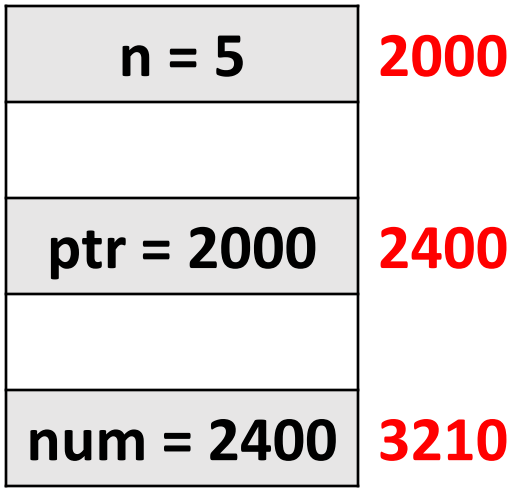
Pointer to Pointer Function Parameter

```
int gl = 95;
void func(int **num)
{
     *num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```



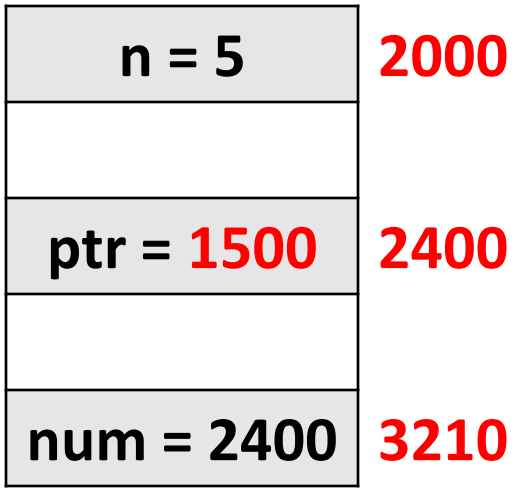
Pointer to Pointer Function Parameter

```
int gl = 95;//address 1500
void func(int **num)
{
     *num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```



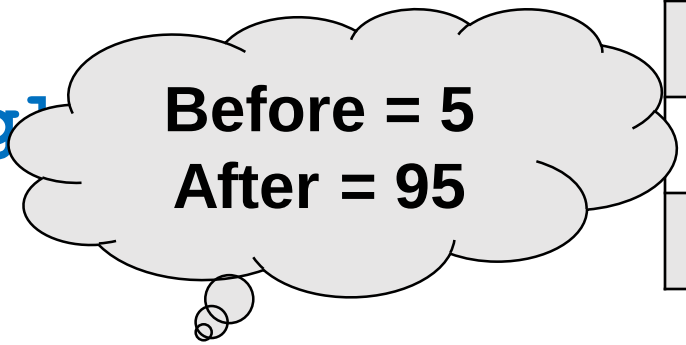
Pointer to Pointer Function Parameter

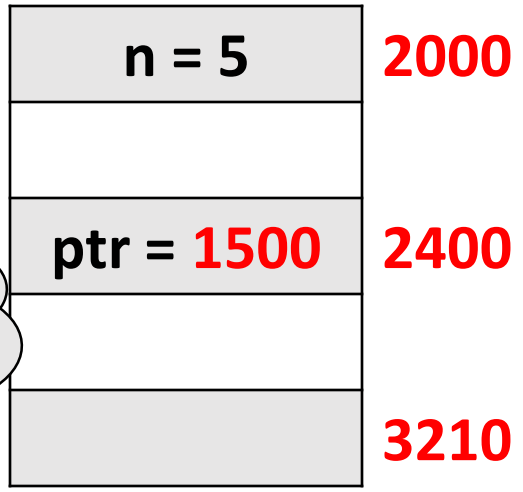
```
int gl = 95;//address 1500
void func(int **num)
{
     *num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```



Pointer to Pointer Function Parameter

```
int gl = 95;//address 1500
void func(int **num)
{
      *num = &gl
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```

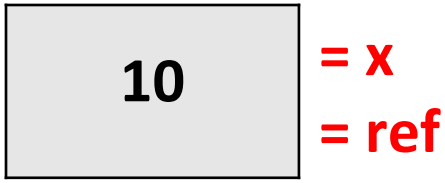




Reference to Pointer Function Parameter

- Recall reference parameters

```
void fun(int &ref) {
     ref = 20;
}
void main() {
     int x = 10;
     fun(x);
     cout<<"New value of x is "<< x;

}
```



The reference parameter **ref** is just another name for **x**. Like an alias/nickname

Reference to Pointer Function Parameter

- Recall reference parameters

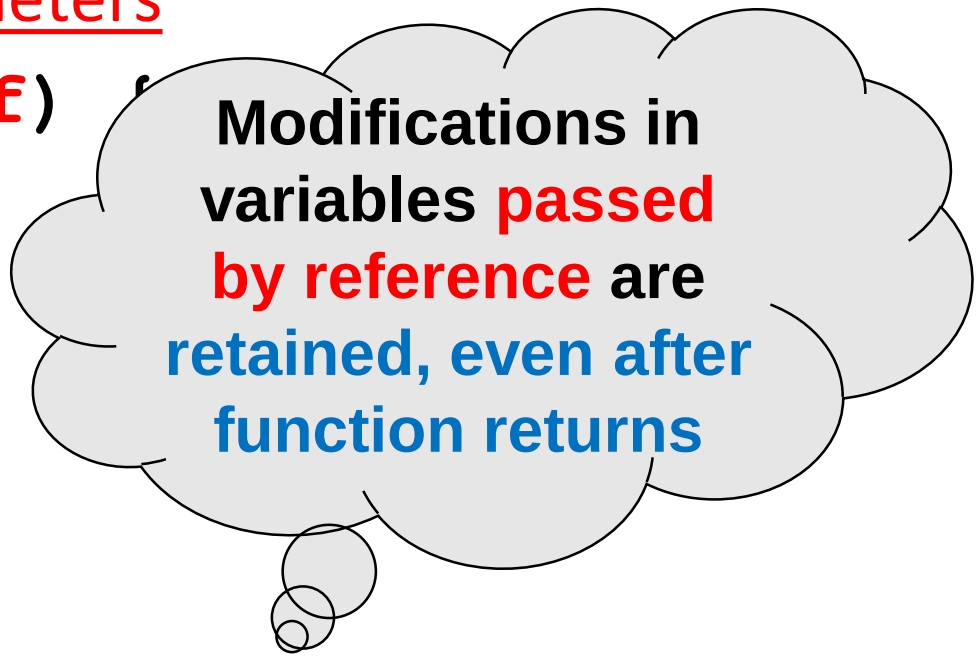
```
void fun(int &ref) {
     ref = 20;
}
void main() {
     int x = 10;
     fun(x);
     cout<<"New value of x is "<< x;
     //prints 20
}
```

The reference parameter **ref** is just another name for **x**. Like an alias/nickname

Reference to Pointer Function Parameter

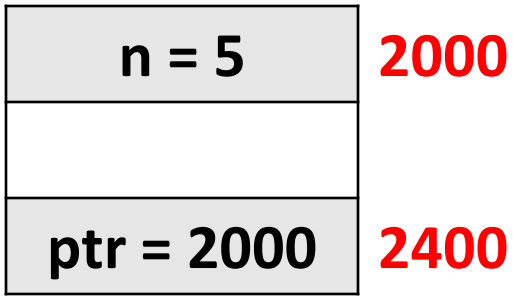
- Recall reference parameters

```
void fun(int &ref) {
     ref = 20;
}
void main() {
     int x = 10;
     fun(x);
     cout<<"New value of x is "<< x;
     //prints 20
}
```



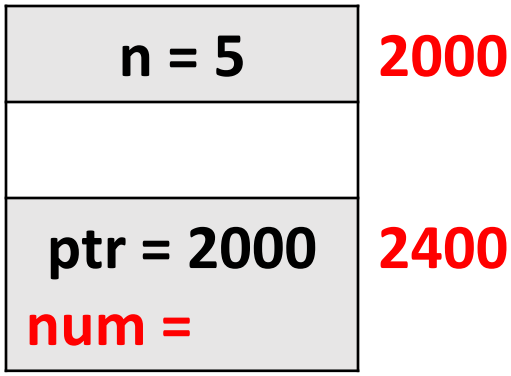
Reference to Pointer Function Parameter

```
int gl = 95;//address 1500
void func(int *&num)
{
     num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```



Reference to Pointer Function Parameter

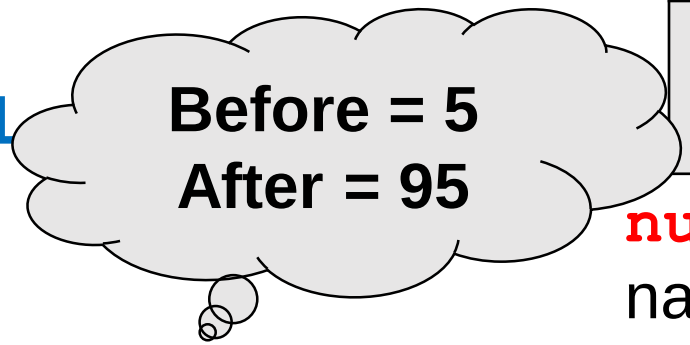
```
int gl = 95;//address 1500
void func(int *&num)
{
      num = &gl;
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```

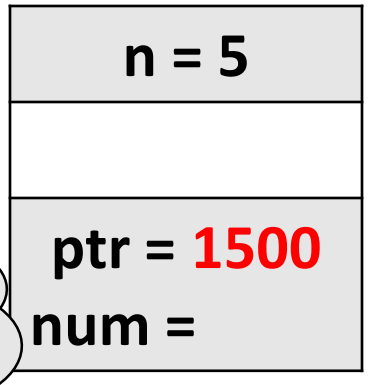


num is just another name for **ptr**.

Reference to Pointer Function Parameter

```
int gl = 95;//address 1500
void func(int *&num)
{
      num = &gl
}
void main()
{    int n = 5; int *ptr = &n;
     cout<<"Before = "<< *ptr <<endl;
     func(ptr);
     cout<<"After = "<< *ptr <<endl;
}
```





num is just another name for **ptr**.

Pointer to Pointer(**) vs Reference to Pointer(*&)

- Reference is just another name(alias), pointer is a variable that stores an address

- References are simpler to use, does most of the work for you
  - However, references cannot be NULL, pointers can

- Cannot make an array of references

Quiz Questions

```
int main() {
// use the following dimensions of an array
int N = 4;
int M = 4;
// Write code for dynamic allocation of a 2D array named A,
using a double pointer[1 marks]
int** A = new int* [N];
for (int i = 0; i < N; i++) {
        A[i] = new int[M];
}
// Assume that you have populated the array as following.
for (int i = 0; i < N; ++i) {
        for (int j = 0; j < M; ++j) {
                A[i][j] = i + 5;
        }
}
```

```
//Use pointer arithmetic and print the output of the following statements [5 marks]
//Assume starting address of this dynamically allocated array of pointers is 100
cout << **A << endl;                    // write your output here: 5
cout << *(*A + 2) + 1 << endl;          // write your output here: 6
cout << **A + 100 << endl;              // write your output here: 105
cout << *(*(A + 2) + 3) + 5 << endl; // write your output here: 12
cout << *A + 3;// write your output here: starting address of first integer array + 12 bytes


// write your code to De-allocate the dynamic memory of the 2D array A [1 marks]
for (int i = 0; i < N; i++) {
        delete [] A[i];
}
delete[] A;
A = NULL;
return 0;
}
```

Q2.Write the output of the following code[3 marks]

```
void funct(int* x, int* y, int*& z)
{
z = x;
x = y;
*x = 200;
}
int main() {
int i = 10;
int j = 20;
int* p = &j;
funct(&i, &j, p);
cout << "i is = " << i << endl;// write your output here: 10
cout << "j is = " << j << endl;// write your output here: 200
cout << "p is = " << --(*p) << endl;// write your output here: 9
}
```

(a) Given the following code snippet, create a pointer to the given pointer (ptr) named ptr_to_ptr, then print the value stored in variable 'x' using this newly created ptr_to_ptr: [1 Mark]

```
int x = 10;

int* ptr = &x;

int **ptr_to_ptr=&ptr;

cout<<**ptr_to_ptr;
```

(b) Given the following code snippet, what will be the output? [1 Marks]

```
int x = 100;

int y = 200;

int *p = &x, *q = &y;

p = q;

cout<<*p;      200
```

(c) Given the following code snippet, what will be the output? [1 Marks]

```
char arr[20];

int i;

for ( i = 0; i < 10; i++ )

        *(arr + i) = 65 + i; // ‘65’ is ASCII code of ‘A’

*(arr + i) = ‘\0’;

cout << arr;
```

ABCDEFGHIJ

(d) Given the following code snippet, what will be the output? [2 Marks]

```
char *ptr;

char arr[ ] = “abcdefgh”;

ptr = arr;

ptr + = 5;

cout << ptr;
```

fgh

(e) Given the following code snippet, what will be the output? (Assuming memory address of variable ‘x’ is 01434CC3.) [2 Marks]

```
int x = 50;

int *ptr = &x;

ptr = ptr + 1;

cout << ptr;
```

01434CC7