

# Fundamentals of Computer Programming



## Chapter 5 Pointers

Chere L. (M.Tech)  
Lecturer, SWEG, AASTU<sup>1</sup>

- Variables in a Memory
- Basics of Pointers
  - *What is pointer?*
  - *Why pointers*
  - *Pointer Declaration*
  - *Pointers Initialization*
- Pointer Operators (& and \*)
- Types of Pointers
  - *NULL Pointer*
  - *Void pointers*
  - *Pointers of Pointer*
  - *Dangling Pointers*
  - *Wild Pointers*
- Pointers Expression
  - *Pointers Arithmetic*
  - *Pointers Comparison*
- Pointers and Constants
- Pointers and Arrays/Strings
- Pointers with Function
  - *Parameter pass-by-address*
  - *Pointer as return type/value*
- Dynamic Memory Management
  - *Memory Allocation (new)*
  - *Memory Allocation (delete)*
- Smart Pointers (new)

# 1) Variables in a Memory

- **Variable** is a **named memory** that characterized by its *identifier (place holder), Data type (type + size), Value its store, Scope, Memory Address*.
- Each variable *occupies some bytes in memory* (according to its size; *one byte char, 4 bytes int*)
  - ✓ i.e. depend on the size of the data type of the **Variables**, it allocated **memory slot(S)**
- Basically memory consists of **consecutive, numbered storage cells (slots)**, see next slide
- And the *assigned memory slot(s)* identified by its **memory address** which is a **hexadecimal notation**. e.g. 0x01AC4D
- The memory address allocation depends on computer/operating system and vary (changed) from execution to execution.

# 1) Variables in a Memory (Cont'd)

## Memory Allocation

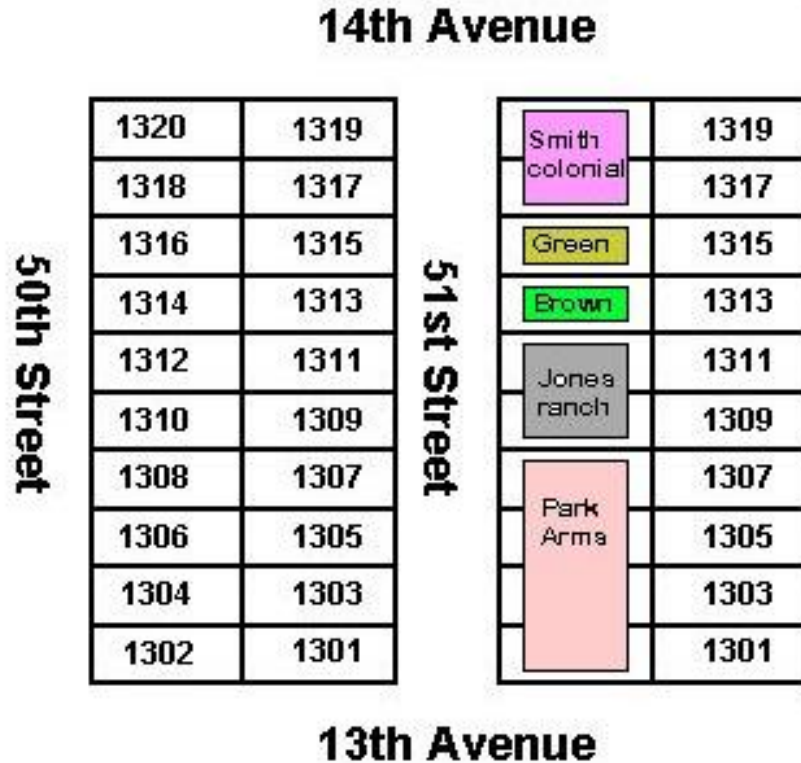
Address	Content	Name	Type	Value
90000000	00	iii	int	000000FF (255 <sub>10</sub> )
90000001	00			
90000002	00			
90000003	FF	sss	short	FFFF (-1 <sub>10</sub> )
90000004	FF			
90000005	FF	ddd	double	1FFFFFFFFFFFFFFF (4.4501477170144023E-308 <sub>10</sub> )
90000006	1F			
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF	ptr	int*	90000000
9000000D	FF			
9000000E	90			
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

- When we create these variables **sum**, **age**, **average** it tells our program to reserve *4bytes, 2bytes, 8bytes* of RAM in memory respectively
- And then *associate the address of the first byte (base address)* of each allocated memory (*90000000, 90000004, 90000006*) to the variables name and restrict the type of data to be stored to the specified data types.

# 1) Variables in a Memory (Cont'd)

## Analogy for Memory Allocation



- Each lot on the map has a sequential number along some street with the streets themselves numbered sequentially. Each lot is the same size, say 20' by 100'.
- Now suppose people buy some of the lots and build houses.
- Suppose a developer takes the first 4 lots (1302-1308 51st Street) and builds an apartment building, calling it the "Park Arms".
- Also Mr. Jones and Ms. Smith each decide to build big houses, a ranch and a colonial, respectively, which span two lots each (1310-1312 and 1318-1320), and that Mr. Green and Mrs. Brown each decide to build a similar modest one-lot houses (1314 and 1316).
- The lots still have their numbers, so that the post-office can deliver mail, but people know the buildings by the appropriate names, "The Park Arms", the "Jones" ranch, the "Brown" house, the "Green" house and the "Smith" colonial

# 1) Variables in a Memory (Cont'd)

## In general, Variable Identifier

- ✓ Is just a “**symbolic name**” and a reference to that memory address.
- ✓ The compiler is responsible to maintain the details of the variable in the **symbol table** to manage the assigned memory location.



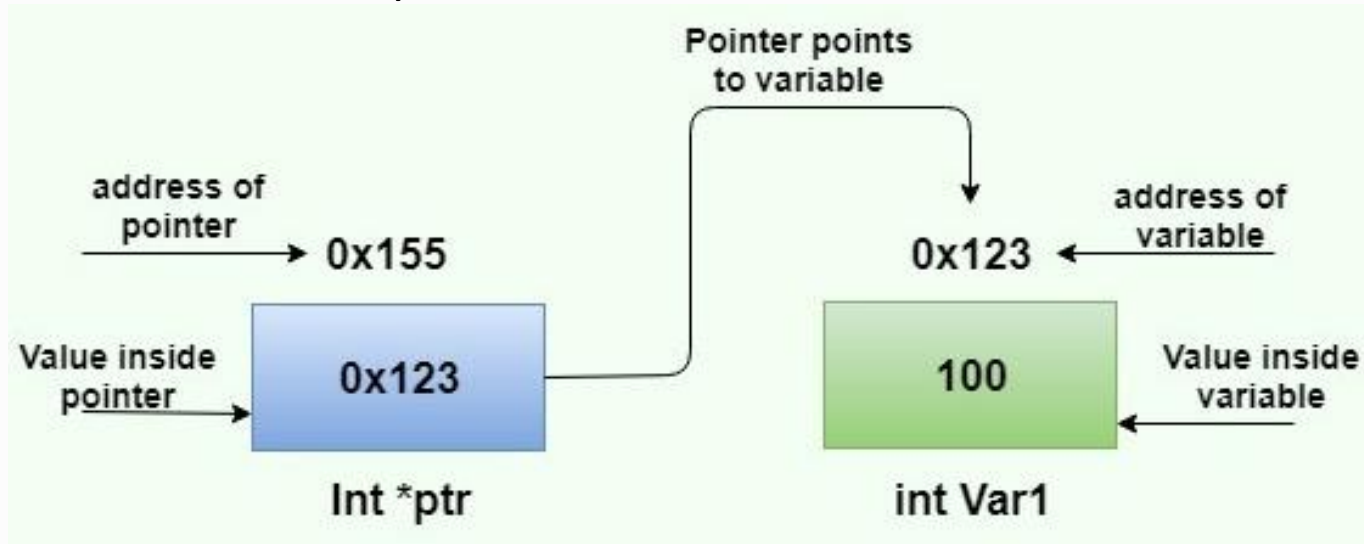
```
int num = 100;
```

Variable **num's** value, i.e., 100, is stored at memory location 1024

## 2) Basics of Pointers

### 2.1 What is Pointer?

- ✓ A **pointer** is a **variable** that stores (holds) the **address** of a **memory cell (slot)** of another variable.
- ✓ In other words, pointers are used to reference the **memory cell allocated** to other variables (see diagram below).
- ✓ The pointer data type should be the same with the variable whose it address is referenced/stored



## 2) Basics of Pointers (cont'd)

### 2.2 Why Pointers?

- For dynamic (on-the-fly) memory allocation and de-allocation
- It provides a way of *indirect accessing a variable without referring* to its name.
  - i.e. Obtaining memory from the system directly
- Used to refer to a particular member of a group (such as an array)
  - ✓ Provide easy and efficient way of array manipulation
- Efficient way to work with functions
  - ✓ *Provide a function to return more than one value*
  - ✓ *Provide functions to access large data blocks (pass arrays and strings more conveniently to function - avoid copying large objects)*
- ✓ Used in building complex data structures (*linked lists, stacks, queues, trees, etc.*)



## 2) Basics of Pointers (cont'd)

### 2.3 Pointer Declaration

- ✓ A Pointer variable is declared as follows:

**Syntax:**     **datatype\* pointer\_name;**  
                  **//or**  
                  **datatype \*pointer\_name;**

where **datatype** is the type of data pointed to.

- ✓ **Examples:**

```
int *P;      /* P is var that can point to an int variable */
float *Q;    /* Q is a float pointer that holds address of float variable */
char *R;     /* R is a char pointer points to char variable */
```

- ✓ **Complex example:**

```
int *AP[5];  /* AP is an array of 5 pointers to int variables */
Date *myBoD /* a structure pointer, assume Date is struct tag */
```

## 2) Basics of Pointers (cont'd)

### Pointers Data types

- ✓ Pointers can be declared to any types
  - *Pointer can hold memory address of any variable of the same type*
- ✓ In other word, each type of object has a pointer type point to it
- ✓ In general, pointer types can be
  - Primitive Data types
    - pointers to int, char, float, bool etc.
  - Derived and User defined data types
    - pointers to derived types (array, strings, functions)
    - pointers to user defined objects (struct, union)
  - Even pointers to pointers

## 2) Basics of Pointers (cont'd)

### 2.4 Pointer Initialization

- The memory associated with *what pointer is pointing* to is NOT allocated/created just by the defining of pointer,

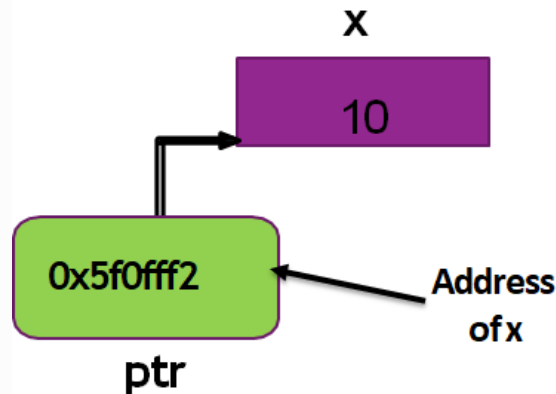
i.e., `int *ptr;` - doesn't allocate any memory for **ptr** to point to

- A pointer can be initialized to any *valid variable address* or to **NULL**.

- Example 1:**

```
int x=10;
```

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



➤ Now **ptr** will contain address where the **variable x** is stored in memory.

**Note:** Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.

## 2) Basics of Pointers (cont'd)

- **Example 2:** initializing pointer to null

```
int *p = 0;  
int *q = NULL;  
int *ptr;  
ptr = 0 or ptr = NULL //is also possible
```

- The value specified for a null pointer is either **0** or **NULL**.
- And all the above create a pointer initialized to **0** in which all pointers are currently a NULL pointer

```
cout << p << endl;    //prints 0
```

- **NULL** is defined in several standard library headers to represent the value 0.
- Prior to C++11, 0 was used by convention
- The value 0 is the *only* integer value that can be assigned directly to a pointer variable without first *casting* the integer to a pointer type.

## 2) Basics of Pointers (cont'd)

- Another Examples:

```
int num, *numPtr = &num;
```

```
int val[3], *valPtr = &val;
```

**Note:**

Be careful *numptr* is declared as a pointer to an integer while *num* is just a regular int

- Cannot mix data types:

```
double cost;
```

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

# 3. Pointer Operators

## 3.1 Address of Operator (&)

- ✓ The "*address of* " operator (&) is a **unary operator** that returns (gives) the memory address of its operand (variable).
- ✓ It can be used in front of any variable object and the result of the operation is the location in memory of the variable

### ✓ Syntax:

**&variable\_name**      *// Here the operand is a valid identifier.*

### ✓ Example 1:

```
int a = 100;
// get the value
cout << a;    //prints 100

//get the memory address
cout << &a;   //prints 1024
```

Memory

address:    1020                      1024



a

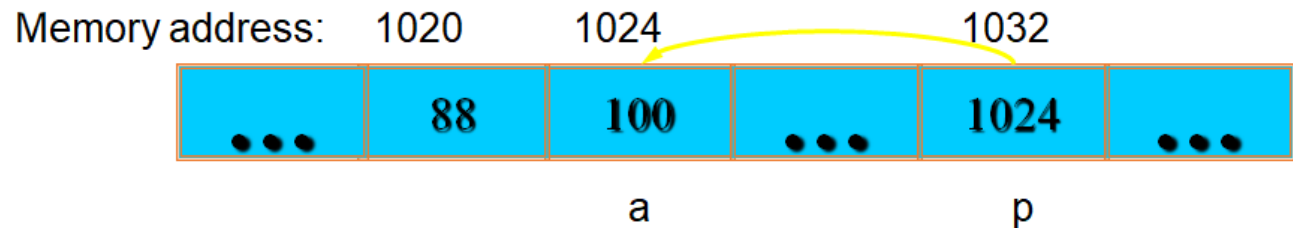
# 3. Pointer Operators (cont'd)

## Example 2: Program to demonstrate of Address of Operator (&)

```
#include <iostream>
using namespace std;
void main(){
    int num = 88, item = 100;

    cout << "The address of a is: " << &num << endl;
    cout << "The address of b is: " << &item << endl;

    int *ptr = &item;
    cout << b << " " << &item << endl;
    cout << p << " " << &ptr << endl;
}
```



- Result is:  
The address of a is: 1020  
The address of b is: 1024

- Result is:  
100 1024  
1024 1032

**Note:** the value of pointer **ptr** is the address of variable **item** and the pointer is also a variable, so it has its own memory address

## 3. Pointer Operators (cont'd)

### Remark

- Variable in the **address of** operator must be *of the right type* for the pointer (i.e. an integer pointer points only at integer variables)

- Exercise: find the output**

```
int V;  
int *Ptr1, ptr2 = &V;  
int Arr1[5];  
Ptr1 = &(Arr1[2]);  
cout<<V<<" "<<&V<<"\n"<<ptr2<<" "<<&ptr2<<endl;  
cout<<Arr1<<" "<<&Arr1<<"\n"<<ptr1<<" "<<&ptr1<<endl;  
  
char *Ptr3 = &V;  
double Arr2[5]; Ptr1 = &(Arr2);  
cout<<V<<" "<<&V<<"\n"<<ptr2<<" "<<&ptr2<<endl;  
cout<<Arr1<<" "<<&Arr1<<"\n"<<ptr1<<" "<<&ptr1<<endl;
```



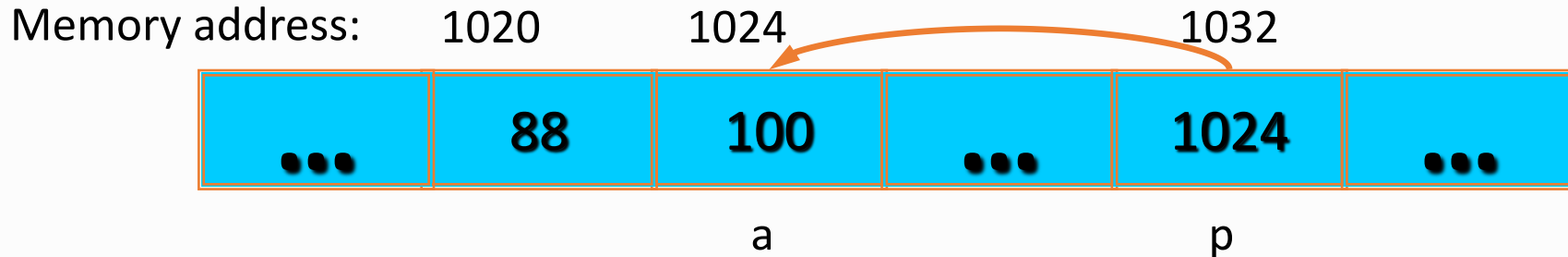
## 3. Pointer Operators (cont'd)

### 3.2 Dereferencing Operator (\*)

- ✓ It is also a unary operator that **returns the value** stored at the address pointed to by the pointer.
- ✓ It also used in pointer declaration.
- ✓ In other words, it is used to access (reference) a value through pointer which is called **indirection**.
- ✓ In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value**.
- ✓ **Syntax:**  
**\**pointer\_name*** // the operand should be not NULL pointer

# 3. Pointer Operators (cont'd)

## Example 1: Dereferencing



```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

● Result is:

```
100
1024
1024 100
1032
```

```
int *ptr = 0;
cout << *ptr << endl;
```

//error, Cannot dereference a pointer whose value is null

# 3. Pointer Operators (cont'd)



- Don't be confused between pointer definition and dereferencing which use the same operator (\*)....
- *They are simply two different tasks represented with the same sign.*

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
```

```
p2 = &b;           //p2 points to b
p2 = p1;           //p2 points to a
b = *p3;           //assign c to b
*p2 = *p3;         //assign c to a
```

```
cout << a << b << c;
```

Result is:

888

## Pointer variable, *p1, p2, p3*

- ✓ **Does not** hold the value of any of the variable **a, b, c**
- ✓ **Only holds an address**
- ✓ **Only points** to the memory location of **a, b, c**
- ✓ They are a **variable themselves** and has their own **memory location**

## 3. Pointer Operators (cont'd)

### Example 2: Dereferencing - a complete program

```
#include <iostream>
using namespace std;
int main (){
    int value1 = 5, value2 = 15;
    int *p1, *p2;
    p1 = &value1;           // p1 = address of value1
    p2 = &value2;           // p2 = address of value2
    *p1 = 10;               // value pointed to by p1=10
    *p2 = *p1;              // value pointed to by p2= value pointed to by p1
    p1 = p2;                // p1 = p2 (pointer value copied)
    *p1 = 20;               // value pointed to by p1 = 20
    cout << "value1==" << value1 << "/ value2==" << value2;
    cout << sizeof(p1) << sizeof(*p1)<< sizeof(&p1) << endl;
    return 0;
}
```

● Let's figure out:  
value1==? / value2==?  
Also, p1=? p2=?

## 3. Pointer Operators (cont'd)

### 4.3 Summary on two operators (\* and &)

#### \* Has two usages:

- pointer – in declaration it **indicates that the object is a pointer.**

`char* s; // s is of type pointer to char`

- pointer dereferencing - indicates the object the pointer pointed to

#### & Has two usages:

- getting address of memory
- to declare a reference variable

e.g. 'call by ref', in function (see later)

## 4. Types of Pointers

- There different types of pointers
- Here below some of those pointers
  - *Null pointer* – a pointer that points to nothing and store zero.
  - *Wild pointer* – uninitialized pointers, holds a garbage address
  - *Dangling pointer* – a pointer that points to a memory location that has been deleted
  - *Void pointer* - a pointer that is not allied with any data types
  - *Pointers of pointer* – a pointer that points to another pointer

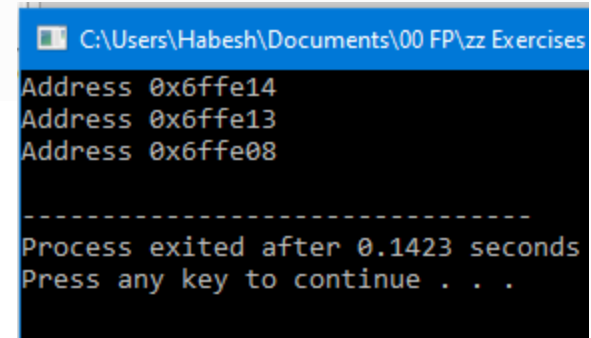
# 4. Types of Pointers (cont'd)

## 4.1 Void pointer

- The **void pointer**, also known as the **generic pointer**, is a special type of pointer that can be pointed at objects of **any data type**.
- A void pointer is declared like a normal pointer, using the **void** keyword as the pointer's type.

### Example

```
#include <iostream>
using namespace std;
int main(){
    int age = 10;
    char sex = 'm';
    double salary = 1452;
    void *ptr = &age; //holds address of int 'age'
    cout<<"Address "<<ptr<<endl;
    ptr = &sex; //void pointer holds address of char 'sex'
    cout<<"Address "<<ptr<<endl;
    ptr = &salary; //holds address of double 'salary'
    cout<<"Address "<<ptr<<endl;
    return 0;
}
```



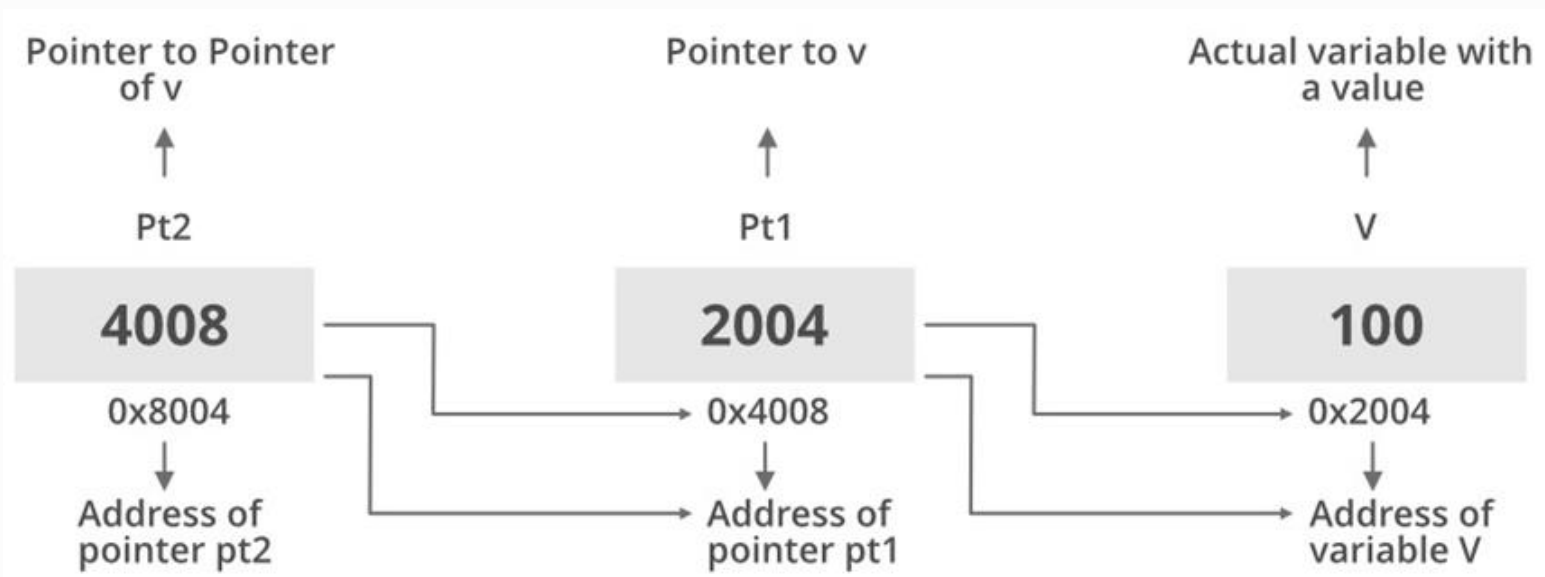
```
C:\Users\Habesh\Documents\00 FP\zz Exercises
Address 0x6ffe14
Address 0x6ffe13
Address 0x6ffe08

-----
Process exited after 0.1423 seconds
Press any key to continue . . .
```

# 4. Types of Pointers (cont'd)

## 4.2 Pointers of pointer

- ✓ A pointer can also be made to point to a pointer variable.  
i.e. pointer of pointer (also called double pointer)
- ✓ But the pointer must be of a type that allows it to point to a pointer
- ✓ A pointer that points to another pointer doesn't allowed to hold address of normal variable even of the same type





# 4. Types of Pointers (cont'd)

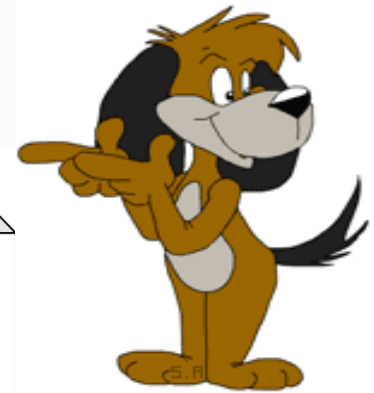
## ✓ Example 1: predict output

// Local Declarations

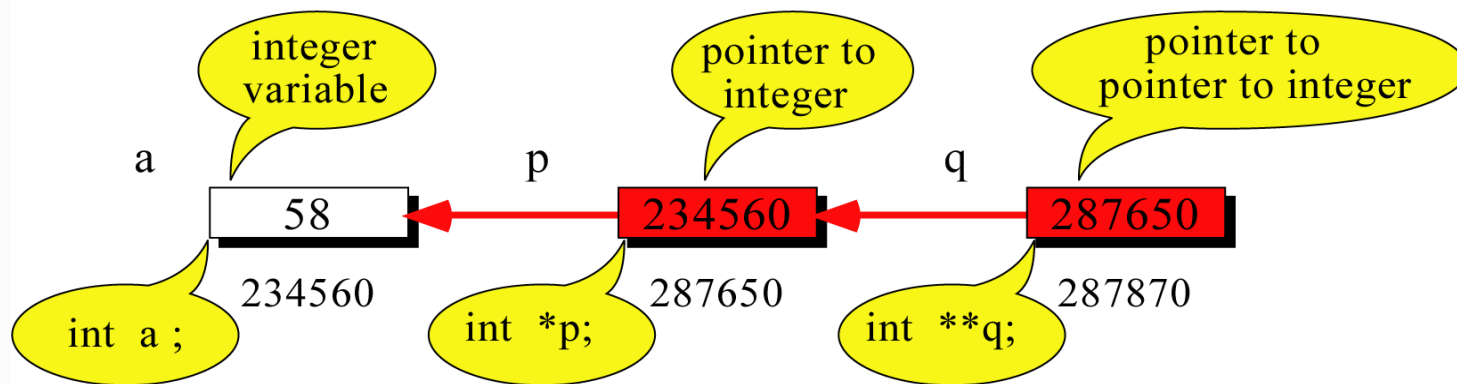
```
int    a ;
int    *p ;
int    **q ;
```

// Statements

```
a = 58 ;
p = &a ;
q = &p ;
cout <<    a << " " ;
cout <<    *p << " " ;
cout <<    **q << " " ;
```



**What is the output?**



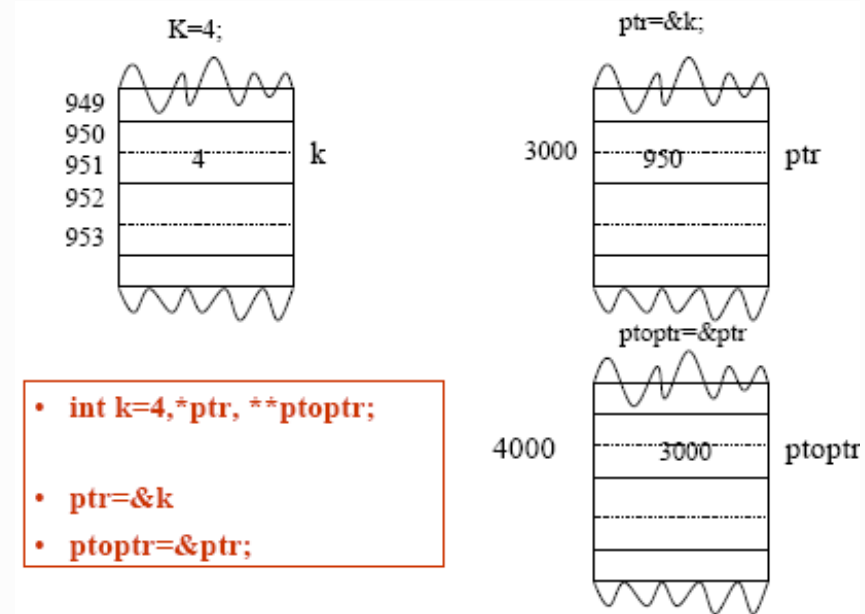
# 4. Types of Pointers (cont'd)

## Example 2: Program to demonstrate pointers of pointer

```
#include <iostream>
using namespace std;
int main (){
    int k = 4, *ptr, **ptoptr;
    ptr = &k, **ptoptr = &ptr;
    cout << "K = " << K << endl;

    * ptr = k+10;
    cout << "x = " << k << endl;

    **proptr = *ptr + k;
    cout << "k = " << k << endl;
}
```



What is the output?

58 53 58

## 4. Types of Pointers (cont'd)

### Exercise 5.1

1. `int i = 5, j = 10;`
2. `int *ptr;`
3. `int **pptr;`
4. `ptr = &i;`
5. `pptr = &ptr;`
6. `*ptr = 3;`
7. `**pptr = 7;`
8. `ptr = &j;`
9. `**pptr = 9;`
10. `*pptr = &i;`
11. `*ptr = -2;`

### Output?

Line	Value			
	i	j	ptr	pptr
4				
5				
6				
7				
8				
9				
10				
11				

# 5. Pointers Expression

## 7.1 Pointers Arithmetic

- Only two arithmetic operations, **addition and subtraction**, may be performed on pointers.
- When we add 1 to a pointer, we are actually adding the size of data type in bytes, the pointer is pointing at.

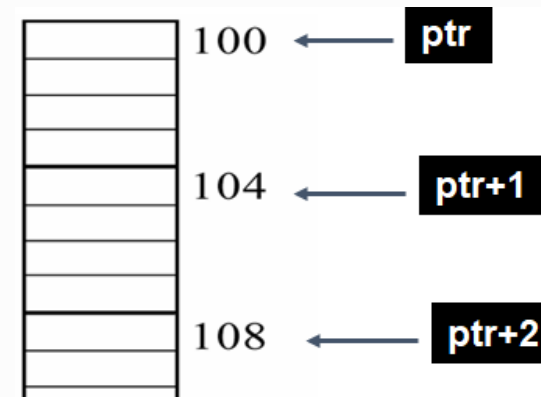
e.g. `int *ptr; ptr++;`

**address = pointer + size of element**

- Assume if current address of x is 1000, then `x++` statement will increase x by 4 (size of int data type) and makes it 1004, not 1001.

- *What's `ptr + 1`? - The next memory location!*
- *What's `ptr - 1`? - The previous memory location!*
- *What's `ptr * 2` and `ptr / 2`?*

*Invalid operations!!!*



# 5. Pointers Expression (cont'd)

## Pointer ++ and -- operations

<code>*ptr++</code>	<code>*(ptr++)</code>	Increments pointer, and dereferences unincremented address i.e. This command first gives the value stored at the address contained in ptr and then it increments the address stored in ptr.
<code>*++ptr</code>	<code>*(++ptr)</code>	Increment pointer, and dereference incremented address i.e. This command increments the address stored in ptr and then displays the value stored at the incremented address.
<code>++*ptr</code>	<code>++(*ptr)</code>	Dereference pointer, and increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it increments the value by 1.
<code>(*ptr)++</code>		Dereference pointer, and post-increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it post increments the value by 1.

## 5. Pointers Expression (cont'd)

### Example: An Illustration of Pointer Arithmetic

```
int i = 5, j = 10, int *ptr, **pptr;  
ptr = &i; pptr = &ptr;  
  
cout<<*ptr++<<" "<<ptr<<endl;  
cout<<*(ptr++)<<" "<<ptr<<endl;  
cout<<*++ptr<<" "<<ptr<<endl;  
cout<<*(&ptr)<<" "<<ptr<<endl;  
cout<<++*ptr<<" "<<ptr<<endl;  
cout<<++(*ptr)<<" "<<ptr<<endl;  
cout<<(*ptr)++<<" "<<ptr<<endl;
```

Output ?

```
int i = 5, j = 10;  
int *ptr, **pptr;  
ptr = &i;          pptr = &ptr;  
  
cout<<ptr+1<<endl;  
cout<<ptr+2<<endl;  
cout<<ptr-1<<endl;  
cout<<ptr-3<<endl;
```

Output ?

**Note:** Run each pairs of cout statements for the program on the left hand and each of the cout statements for the program on the right hand separately to get similar outputs .

# 5. Pointers Expression (cont'd)

## 7.2 Pointers Comparison

- **Relational operators** can be used to compare addresses in pointers.
- Comparing addresses in pointers is not the same as comparing the values pointed at by pointers.
- Only pointer of the same types are comparable.

- **Example:** pointer comparison, consider the declaration `int ptr1, ptr2;`

`if (ptr1 == 0)`      `//check if the pointer is null or not.`

`if (ptr1 == ptr2)`      `//check if addresses stored in two pointers are equal`

`if (ptr1 > ptr2)`      `//check if ptr1 has greater memory address index`

`if (*ptr1 == *ptr2)`      `// compares values pointed by the pointers`

`if (*ptr1 == ptr2)`      `// invalid: address cannot compared with data`

- **Note:** comparing pointers of different data types.

## 6. Pointers and Constants (1/5)

- In programming, a **constant** is a value that should not be altered by the program during normal execution, i.e., the value is fixed (constant).
- It can be either "**named/symbolic constant**" simply referred as **constant** or **literal value**.
- A pointer can be either constant itself (**Constant Pointer**) or pointed to an other constant objects (**Pointer to Constant**) .

### (a) Constant Pointer:

- A pointer is constant it self (i.e. the address it hold cannot be changed).
- It will always point to the same address.
- The value stored at the address it pointing can be changed.
- How to declare it?

**Syntax:** **data-type \*const pointerName = &object;**

- The constant pointer can hold address of:
  - *non-constant object or*
  - *constant object*



# 6. Pointers and Constants (2/5)

## Example 1: constant pointer to non-constant object

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

```
int main(){
```

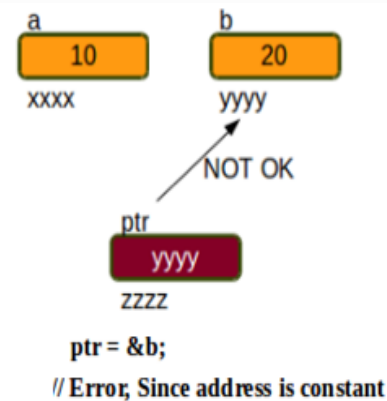
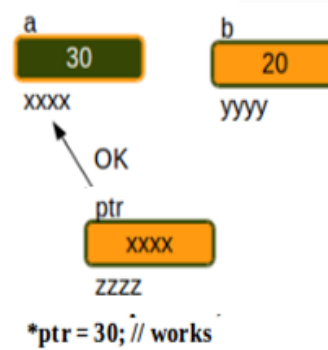
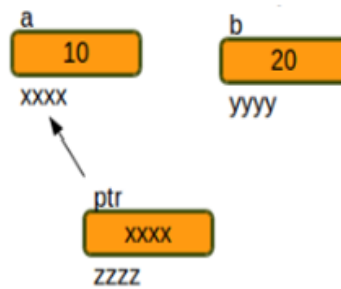
```
    int a{ 10 };
    int b{ 20 };
```

```
    int* const ptr = {&a};
    cout << *ptr << "\t";
    cout << ptr << "\n";
```

```
    *ptr = 30; // Acceptable to change the value of a
    cout << *ptr << "\t";
    cout << ptr << "\n";
```

```
    // ptr = &b; //Error: trying to change value of
    // read-only variable 'ptr'
    return 0;
```

```
}
```



## 6. Pointers and Constants (3/5)

### (b) Pointer to Constant:

- A pointer through which the value of the variable that the pointer points cannot be changed.
- The address of these pointers can be changed.
- How to declare it?

**Syntax:** **const** <type of pointer>\* <name of pointer>

- **Example:**

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

```
int main(){
    int a{ 10 }, b{ 20 };
    const int* ptr;
    ptr = &a;
```

```
cout << *ptr << "\t";
cout << ptr << "\n";
ptr = &b;
cout << *ptr << "\t";
cout << ptr << "\n";
```

```
*ptr = 30; //error: not acceptable to change value
return 0;
}
```

## 6. Pointers and Constants (4/5)

### (c) Pointer constant to Constant:

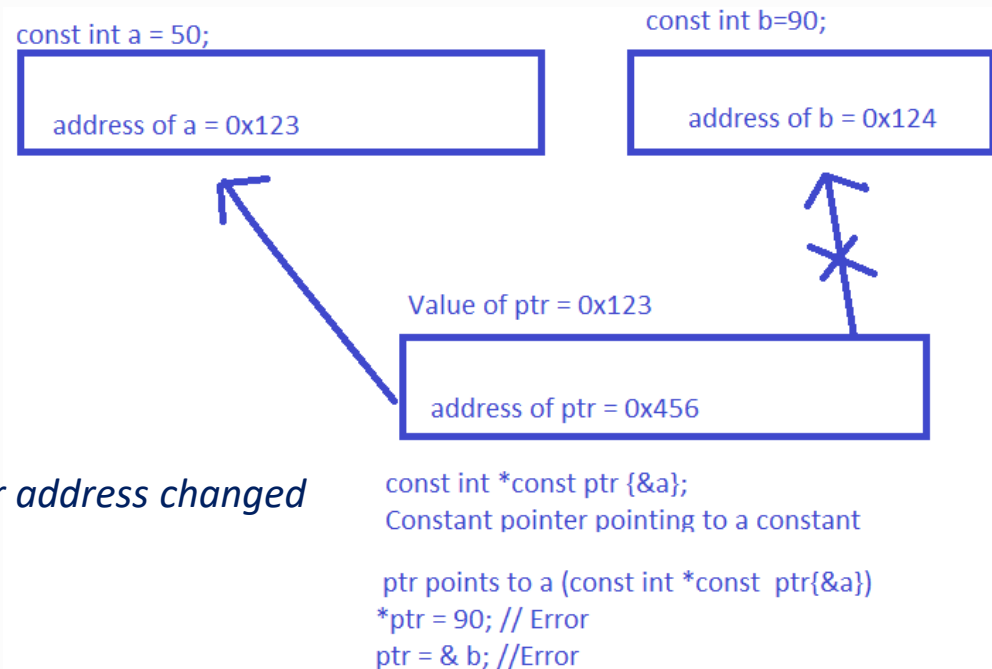
- The combination of the two pointers.
- It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

**Syntax:** **const** <type of pointer>\* **const** <name of pointer>

- Example:

```
#include <iostream>
using namespace std;
int main(){
    int a{ 50 }, b{ 90 };
    const int* const ptr = &a;
    cout << *ptr << "\t" << ptr << "\n";
```

```
//the below are invalid: neither the value nor address changed
    ptr = &b;    *ptr = 30;
    return 0;
}
```



## 6. Pointers and Constants (5/5)

### Exercise 5.2:

Analyze the segment below and identify

- Type of pointers
- Invalid statements

```
int x=10, y=20;  
const int z=10;
```

```
int * p1=&x;  
int *p1=20;  
p1=&y;
```

```
const int * p2=zx;  
*p2=50;
```

```
p2=&y;  
*p2=100;
```

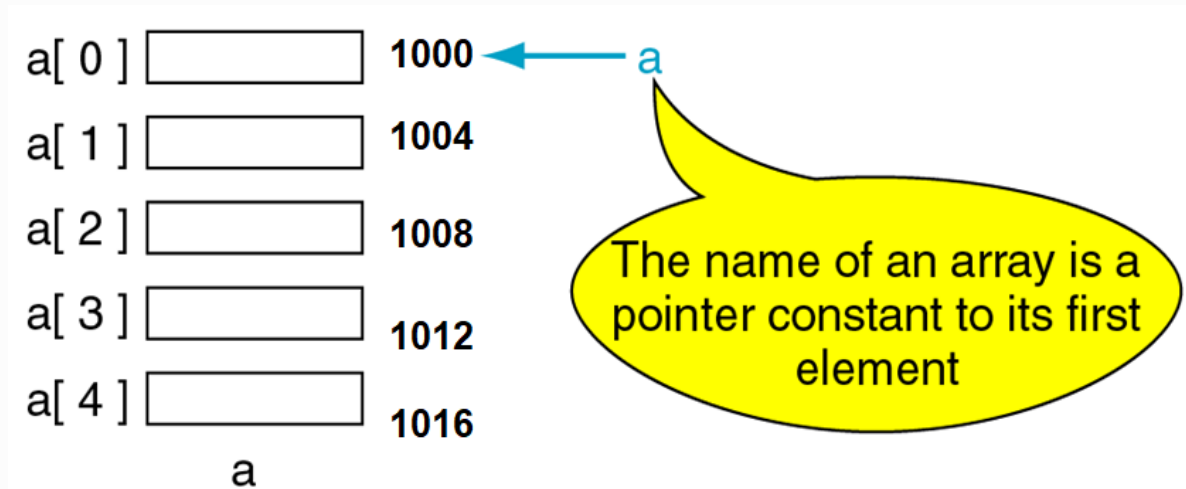
```
int * const p3= &x;  
*p3=60;  
p3=&y;
```

```
const int * const p4=&x;
```

```
p4=&y;  
*p4=90;
```

# 7. Pointers and Arrays

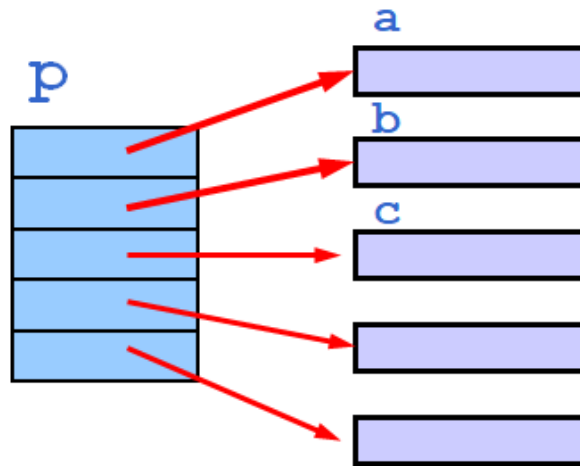
- Basically the concept of array is very similar to the concept of pointer.
- The **identifier of an array** actually a pointer that holds the **address of the first element (base address)** of the array.
- Therefore if you have two declarations : `int a[10];`      `int* p;`  
then the assignment **`p = a;`** is perfectly valid



- The only difference between the two is that *the value of “p” can be changed to any integer variable address* whereas *“a” will always point to the integer array of length 10 defined.*

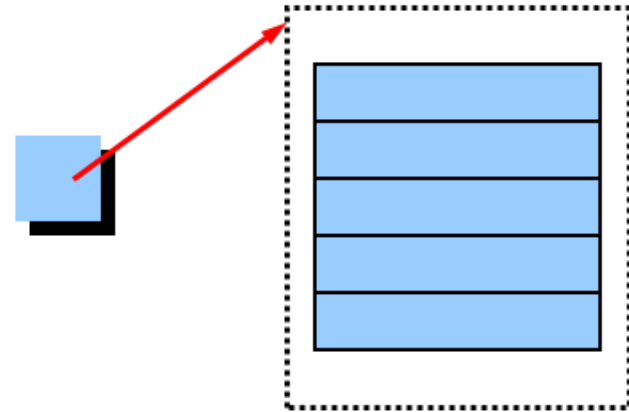
# 7. Pointers and Arrays (cont'd)

Differentiate between Array of Pointers & Pointers to Array



**An array of Pointers**

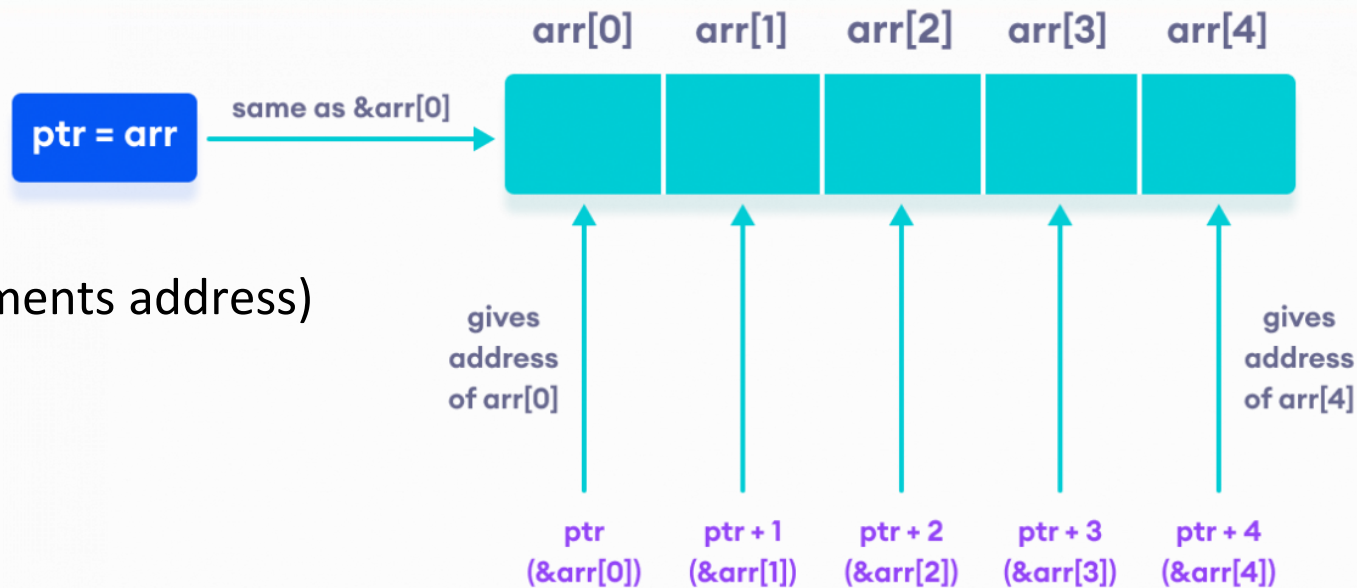
```
int a = 1, b = 2, c = 3;
int *p[5];
p[0] = &a;
p[1] = &b;
p[2] = &c;
```



**A pointer to an array**

```
int list[5] = {9, 8, 7, 6, 5};
int *p;
P = list; //points to 1st entry
P = &list[0]; //points to 1st entry
P = &list[1]; //points to 2nd entry
P = list + 1; //points to 2nd entry
```

# 7. Pointers and Arrays (cont'd)



## Example 1:

(Printing 1D array elements address)

```
#include <iostream>
using namespace std;
int main (){
    int arr[5];
```

```
    cout<<"\nAddress of a[0]: "<<&arr[0];
    cout<<"\nArray Name as pointer: "<<arr;
```

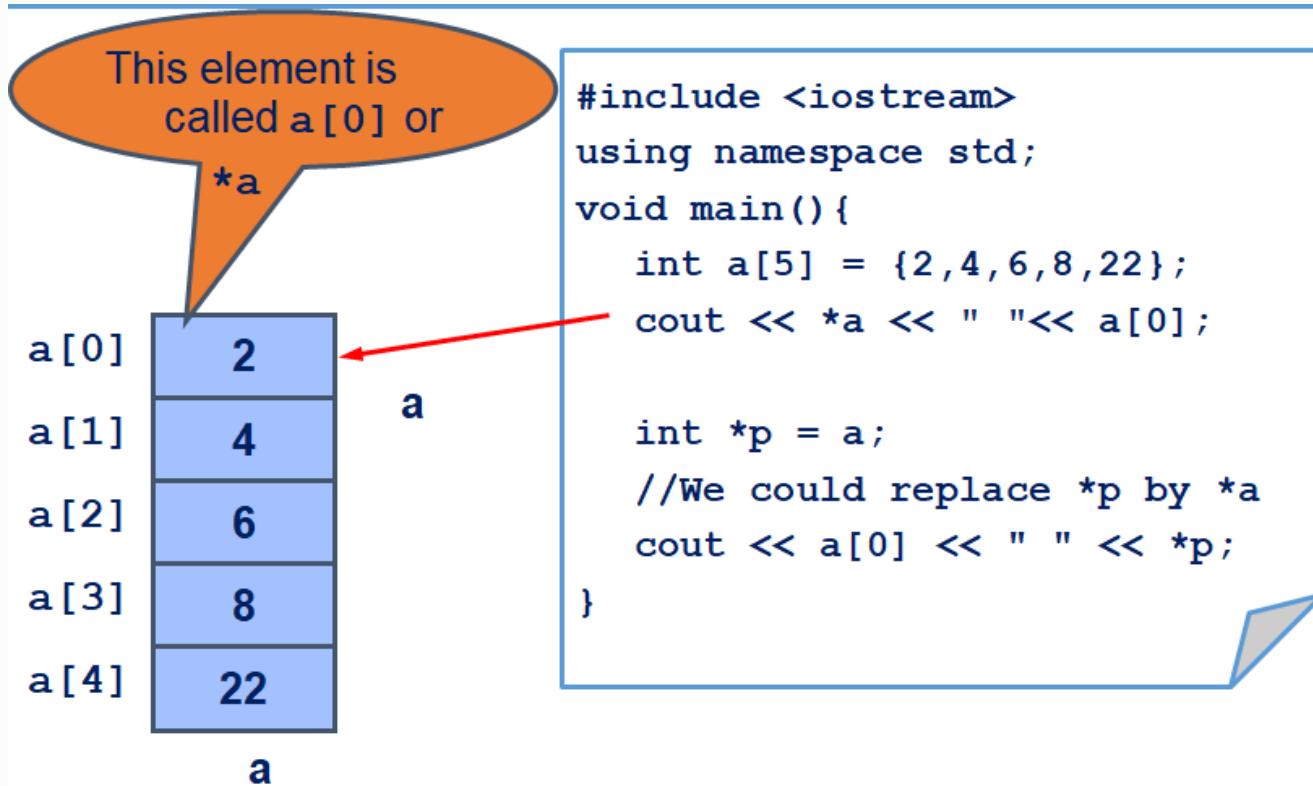
```
    cout<<"\n\nAddress of a[1]: "<<&arr[1] <<"\t"<<(arr+1);
    cout<<"\nAddress of a[4]: "<<&arr[4]<<"\t"<<(arr+4);
```

```
    int *ptr = arr;
    cout<<"\n\nAddress of array: "<<ptr; }
```

# 7. Pointers and Arrays (cont'd)

## Dereferencing an Array Name (using array name as pointer)

- To access an array, any pointer to the first element can be used instead of the name of the array



### Note:

- Array names is used as Pointers
- `*(a+n)` & `a[n]` is identical to



## 7. Pointers and Arrays (cont'd)

**Example 2:** Using a pointer notation to print array elements

```
// C++ Program to insert and display  
// data entered by using pointer notation.
```

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

```
int main() {  
    float arr[5];
```

```
// Insert data  
cout << "Enter 5 numbers: ";  
for (int i = 0; i < 5; ++i) {
```

```
    // insert value to arr[i]  
    cin >> *(arr + i);
```

```
}
```

```
// Display data  
cout << "Displaying data:\n ";  
for (int i = 0; i < 5; ++i) {
```

```
    // display value of arr[i]  
    cout << *(arr + i) << endl ;
```

```
}
```

```
return 0;
```

```
}
```

## 7. Pointers and Arrays (cont'd)

**Example 3:** Manipulate array elements using an other pointer

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

int main () {
    double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
    double *p;
    p = balance;

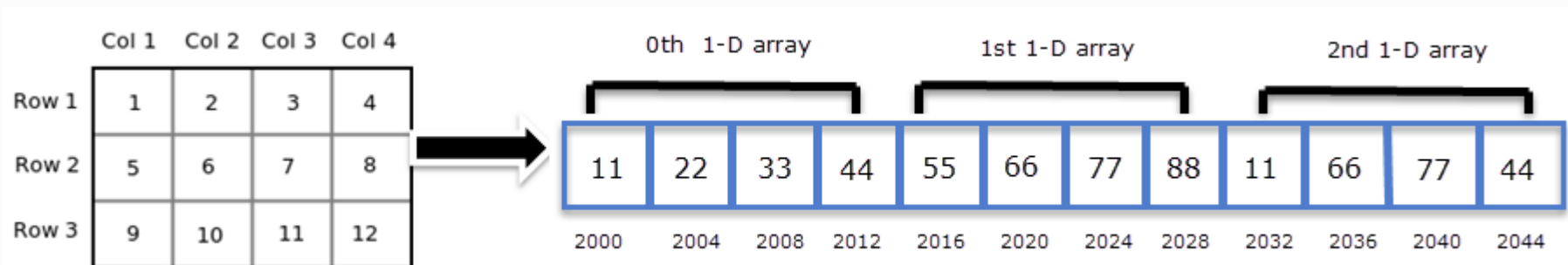
    // output each array element's value
    cout << "Array values using pointer \n";

    for ( int i = 0; i < 5; i++ ) {
        cout << "*(p + " << i << " ) : " << *(p + i) << endl;
    }
    return 0;
}
```

# 7. Pointers and Arrays (cont'd)

## Pointers and 2-D Arrays

- The elements of 2-D array can be accessed with the help of pointer notation also.
- Before we discuss how to manipulate 2D array with pointers, let's understand how memory is allocated for 2D-array elements.
  - Since memory in a computer is organized linearly ***it is not possible*** to store the ***2-D array in rows and columns***.
  - A 2-D array can be considered as a collection of one-dimensional arrays that are placed one after another
  - The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e. rows are placed next to each other.



# 7. Pointers and Arrays (cont'd)

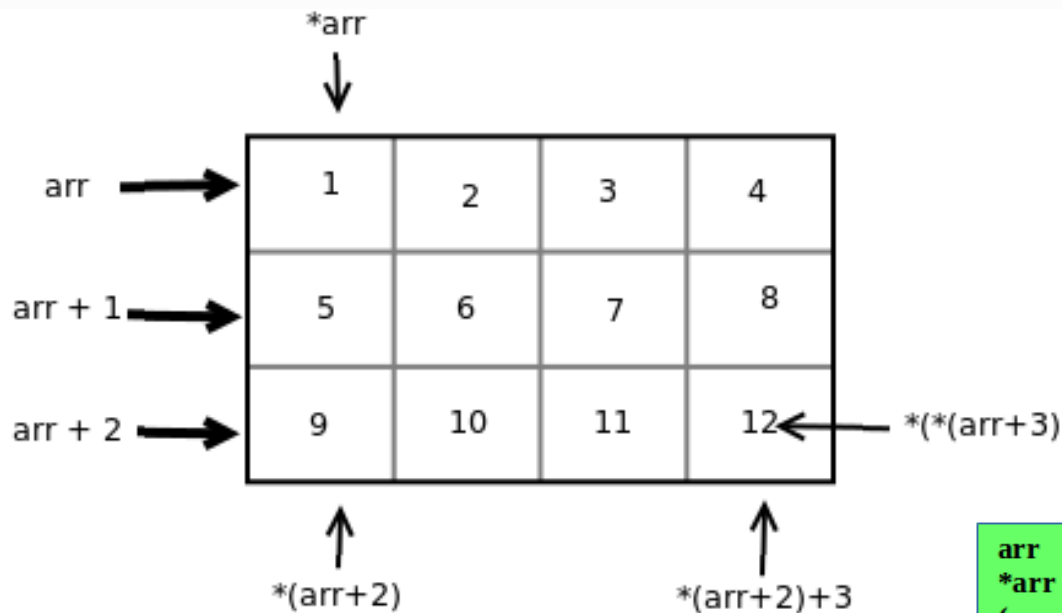
## Note

- Consider the 2-D array declaration **int arr[3][4]**,
- Recall that
  - *The name of an array is a **constant pointer** that points to the 1st elements of 1-D array.*
  - *A pointer (array) can remember members of particular data group. i.e. **arr** is an array of **3 elements** where each element is a 1-D array of 4 integers.*
- Accordingly, in 2-D array the **array identifier** is pointing to the 1<sup>st</sup> row (group of data elements)
  - *i.e. the address of the 1<sup>st</sup> element of each row is used as a base address of the respective row.*
  - Hence, **arr** is a '**pointer to an array of 4 integers**' and according to pointer arithmetic the expression **arr + 1** and **arr + 2** will represent the address of the 1<sup>st</sup> element of the 2<sup>nd</sup> and 3<sup>rd</sup> row respectively.

# 7. Pointers and Arrays (cont'd)

In general,

- ✓  $\text{arr} = *arr = \text{arr}[0] = \&\text{arr}[0][0]$
- ✓  $\text{arr}+1 = *arr + 1 = (\text{arr}[0] + 1)$   
 $= (\&\text{arr}[0][0] + 3 * 4\text{bytes}) = (\text{arr}[0] + 12 \text{ bytes})$



## How 2-D array operates?

- $\&\text{arr}[i][j] = \text{arr}[i]+j = *(\text{arr}+i)+j$
- $\text{arr}[i][j] = *(\text{arr}[i] + j)$   
 $= *(*(\text{arr} + i) + j)$

$\text{arr}$	Points to 0 <sup>th</sup> 1-D array
$*arr$	Points to 0 <sup>th</sup> element of 0 <sup>th</sup> 1-D array
$(\text{arr} + i)$	Points to i <sup>th</sup> 1-D array
$*(\text{arr} + i)$	Points to 0 <sup>th</sup> element of i <sup>th</sup> 1-D array
$*(\text{arr} + i) + j$	Points to j <sup>th</sup> element of i <sup>th</sup> 1-D array
$*(*(\text{arr} + i) + j)$	Represents the value of j <sup>th</sup> element of i <sup>th</sup> 1-D array

# 7. Pointers and Arrays (cont'd)

## Example 1: using 2d Array name as a pointer notation

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

int main(){
    int arr[3][4] = { { 10, 11, 12, 13 },
                     { 20, 21, 22, 23 },
                     { 30, 31, 32, 33 } };

    for (int i = 0; i < 3; i++) {
        cout<<"\nThe 1st Address of "<<i<<"th array = ";
        cout<<*(arr + i); //arr[i]

        cout<<"\n"<<i+1<<" Address of "<<i<<" row elements ";
        for (int j = 0; j < 4; j++)
            cout<<*(arr + i) + j<<" "; //arr[i]+j

        cout<<"\n"<<i+1<<" row elements: ";
        for (int j = 0; j < 4; j++)
            cout<<*(*(arr + i) + j)<<" ";
        cout<<endl;
    }
    return 0;
}
```

# 7. Pointers and Arrays (cont'd)

**Example 2:** Manipulation 2-D Array elements using a pointer without nested loop

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

```
int main(){
    int arr[3][4] = { { 10, 11, 12, 13 },
                      { 20, 21, 22, 23 },
                      { 30, 31, 32, 33 } };

    int *ptr = arr[0]; /*(arr+1)
    cout<<"\nAddress and value of the "<<endl;
    for (int i = 0; i < 12; i++) {
        cout<<i+1<<" element: ";
        cout<<"\t"<<ptr+i<<"\t"<<*(ptr+i)<<endl;
    }
    return 0; }
```

**The output looks**

```
C:\Users\Habesh\Documents\00 FP\zz Exercises F
Address and value of the
1 element:      0x6ffdd0      10
2 element:      0x6ffdd4      11
3 element:      0x6ffdd8      12
4 element:      0x6ffddc      13
5 element:      0x6ffde0      20
6 element:      0x6ffde4      21
7 element:      0x6ffde8      22
8 element:      0x6ffdec      23
9 element:      0x6ffdf0      30
10 element:     0x6ffdf4      31
11 element:     0x6ffdf8      32
12 element:     0x6ffdfc      33

-----
Process exited after 0.3071 seconds
Press any key to continue . . .
```

## 7. Pointers and Arrays (cont'd)

### Exercise 5.3:

What is the value of the array elements after execution of the below code segment?

1. `float a[5];`
2. `float *ptr;`
3. `ptr = &(a[3]);`
4. `*ptr = 9.8;`
5. `ptr -= 2;`
6. `*ptr = 5.0; ptr--;`
7. `*ptr = 6.0;`
8. `ptr += 3;`
9. `*ptr = 7.0;`

**Output?**

a[0]:

a[1]:

a[2]:

a[3]:

a[4]:



## 7. Pointers and Arrays (cont'd)

### Exercise 5.4:

Consider the below 2D array declaration and identify the elements that will be updated?

```
float mark[3][5] = {{10,11,12,13,14},{20,21,22,23,24},  
                    {30,31,32,33,34}};
```

1. `float *ptr = mark[0];`
2. `ptr +=2;`
3. `*ptr *= 2;`
4. `Ptr = *(mark+1)+3;`
5. `*ptr += 7.0;`
6. `ptr += 4;`
7. `*ptr += 8.0;`

# 7. Pointer and Strings

- Like an array, strings (character array) can be handled using pointers

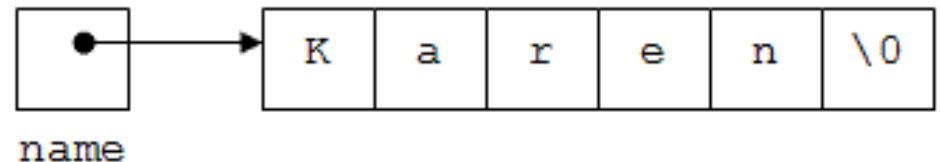
- Example:**

```
char str[] = "computer";
char *cp = str;
cout<<str;           // using variable name
cout << cp;          // using pointer variable
```

- By using the **char\*** pointer, the whole string can be referenced using single pointer

- Examples:**

```
char* name = "Karen";
```



```
char *names[] = {"Mengistu", "Abera",
                 "Chala", "Barega", "Hawi"};
```

## 7. Pointer and Strings (Cont'd)

### Example : String pointer

```
#include<iostream>
#include<string.h>
using namespace std;

int main(){
    char *names[] = {"Mengistu", "Abera", "Chala", "Barega", "Hawi"} ;

    int len=strlen(names[1]);    // length of 2nd string
    cout<<"Originally:\n\tstring 2 is ";    cout.write(names[1],len).put('\n');
    cout<<"\tand string 4 is ";    cout.write(names[3],len).put('\n');

    // now exchange the position of string 2 and 4
    char *tptr;
    tptr = *(names+1);
    names[1] = *(names+3);
    names[3] = tptr;

    // now print the exchanged string
    cout<<"\nExchanged:\n\tstring 2 is ";    cout.write(names[1],len).put('\n');
    cout<<"\tand string 4 is ";    cout.write(names[3],len).put('\n');
}
```

# 7. Pointer and Strings (Cont'd)

## Advantages of String Pointer

- *Pointer to string can be used to declare a unnamed string.*
- *No need to declare the size of string beforehand.*
- *Makes more efficient use of available memory by consuming lesser number of bytes to store the strings*
- *Makes the manipulation of the strings much easier*

# 8. Pointers with Function

- In a function pointer can be used
  - *Function calling and parameter passing*
  - *Return multiple value from functions*
  - *Passing array to function conveniently*
  - *Function pointer (hold address of functions)*
  
- Function parameter passing - 3 ways
  - *Parameter pass by value*
  - *Parameter pass by reference*
  - *Parameter pass by pointers (also called pass by address)*
  - *Parameter pass by reference pointer (rare)\*\* //reading assignment*

# 8. Pointers with Function (cont'd)

## Reference Variable Vs. Pointers

- A reference variable is an additional name (alias – alternative name) to an existing memory location.
- Unlike a reference variable, pointers hold address of an other variable

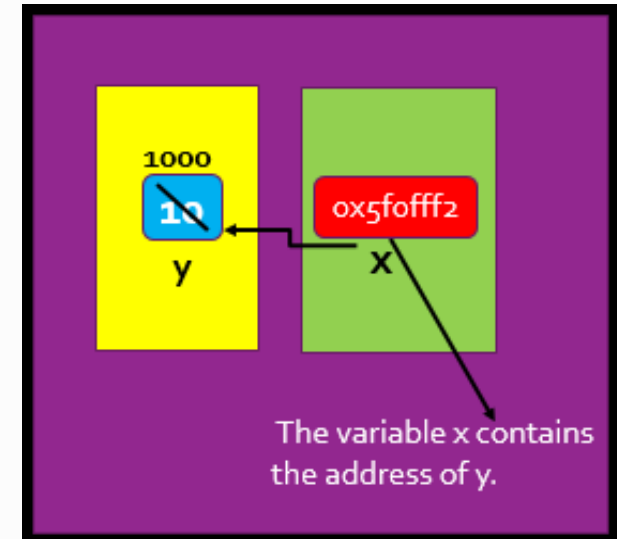
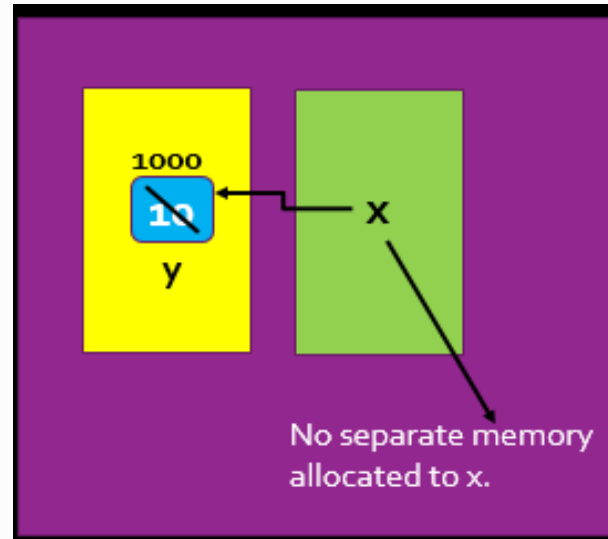
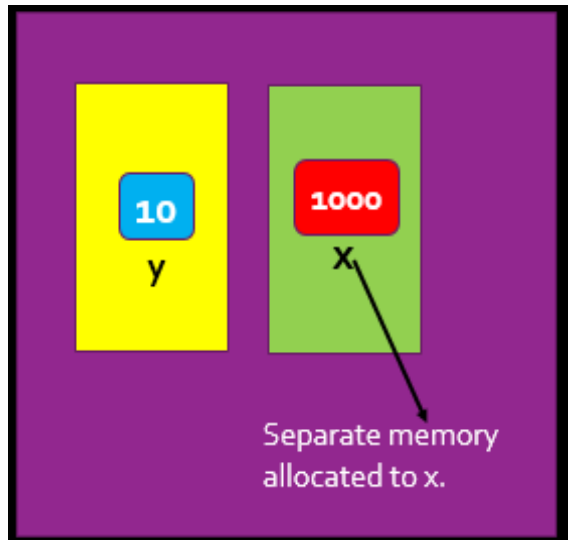


### Note:

- Unlike a pointer, a reference variable always refers to the same object.
- Assigning a reference variable with a new value actually changes the value of the referred object.
- Reference variables are commonly used for parameter passing to a function

# 8. Pointers with Function (cont'd)

## Comparison of Summary of parameter passing



# 8. Pointers with Function (cont'd)

## Comparison of Summary of parameter passing

PASS BYVALUE	PASS BYREFERENCE	PASS BYPOINTER
Separate memory is allocated to formal parameters.	Formal and actual parameters share the same memoryspace.	Formal parameters contain the address of actual parameters.
Changes done in formal parameters are not reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.
<b>For eg.</b> <pre>void cube(int x) {x= x*x*x;} void main(){     int y=10;     cout&lt;&lt;y&lt;&lt;endl;     cube(y);     cout&lt;&lt;y&lt;&lt;endl;} output: 1010</pre>	<b>For eg.</b> <pre>void cube(int x) {x= x*x*x;} void main(){     int y=10;     cout&lt;&lt;y&lt;&lt;endl;     cube(y);     cout&lt;&lt;y&lt;&lt;endl;} output: 101000</pre>	<b>For eg.</b> <pre>void cube(int x) {x= x*x*x;} void main(){     int y=10;     cout&lt;&lt;y&lt;&lt;endl;     cube(y);     cout&lt;&lt;y&lt;&lt;endl;} output: 101000</pre>



## 8. Pointers with Function (cont'd)

### Example 1: parameter passing by address

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

void calc_IBM(float w, float h, float *ibm){
    *ibm = w/(h*h);
}

int main(){
    float weight, height, myIBM;

    cout<<"Enter Weight & height: ";
    cin>>weight>>height;

    calc_IBM(weight, height, &myIBM);
    cout<<"Your IBM is: "<<myIBM;
}
```

## 8. Pointers with Function (cont'd)

### Example 2: Pointer Return Values

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

float *findMax(float A[], int N) {
    float *theMax = &(A[0]);

    for (int i = 1; i < N; i++){
        if (A[i] > *theMax)
            theMax = &(A[i]);
    }
    return theMax;
}
```

```
int main() {
    float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
    float *maxA;

    maxA = findMax(A,5);
    cout<<"The Max is: "<<*maxA<<endl;

    return 0;
}
```

## 8. Pointers with Function (cont'd)

### Example 3: Pointer Return address

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

float *findMax(float *A, int N) {
    float theMax = A[0];

    for (int i = 1; i < N; i++){
        if (A[i] > theMax)
            theMax = A[i];
    }
    return &theMax;
}
```

```
int main() {
    float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};
    float *maxA;

    maxA = findMax(A,5);
    //findMax(&A[0],5); is also possible

    cout<<"The Max is: "<<*maxA<<endl;

    return 0;
}
```

## 8. Pointers with Function (cont'd)

**Example 4:** Return multiple value using pointer

```
#include<iostream>
using namespace std;
void division(int a, int b, int* q, int* r) {
    *q = a/b;    *r = a%b;
}
int main(){
    int quotient, reminder, x, y;
    cout<<"Enter two numbers: ";    cin>>x>>y;

    // The last two arguments are passed
    // by giving addresses of memory locations
    division(x, y, &quotient, &reminder);
    cout<<"The quotient is "<<quotient<<endl;
    cout<<"The reminder is "<<reminder<<endl;
    return 0;
}
```

**Hot to return multiple value in function?**

- Array
- Structure
- Calling by reference
- Calling by address



## 8. Pointers with Function (cont'd)

### Example 5: Passing array & string to function using pointers

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

```
double getAverage(int arr[], int size);
int main () {
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    // pass pointer to the array as an argument.
    avg = getAvg(balance, 5 );
    //&balance[0] also possible

    // output the returned value
    cout<<"Average value is: "<<avg;
}
```

```
double getAvg(int *arr, int size)
{
    int sum = 0;
    double avg;

    for (int i = 0; i < size; ++i) {
        sum += *arr+i;
    }
    avg = double(sum) / size;

    return avg;
}
```

# 10. Pointer to Objects

## Example: Pointer to structure object

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

```
struct student{
    int rollno;
    char name[20];
};
```

```
int main(){
    student s1;
    cout<<"Roll No. ";
    cin>> s1.rollno;
    cin.ignore();
    cout<<"Name: ";
    gets(s1.name);
```

```
    student *stu;
    stu=&s1; //now stu points to s1
    cout<<"\n\n Roll No."<<stu->rollno;
    cout<<"\n Name: "<<stu->name;

    return 0;
}
```

# Summary of Pointer

- Pointers are variables and must be declared
- Pointer points to a variable by remembering its address:
- An address of (&) operator tells where the variable is located in memory.

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

- The dereference (\*) operator is used either to identify a pointer as a variable or indirection

```
double* d_ptr; // * identifies d_ptr as a pointer variable.
```

```
cout<<*d_ptr; //indirection
```

- An array identifier is a constant pointer that points to the 1<sup>st</sup> element.
- Pointer is very use full in dynamic memory allocation
- Pointer used to pass parameter by address and also return multiple value

# Activities (predict output and debugging error)

## 1. Find the output of the following program

**(a)**

```
int a = 3;
char s = 'z';
double d = 1.03;
int *pa = &a;
char *ps = &s;
double *pd = &d;
cout << sizeof(pa) << sizeof(*pa) << sizeof(&pa) << endl;
cout << sizeof(ps) << sizeof(*ps) << sizeof(&ps) << endl;
cout << sizeof(pd) << sizeof(*pd) << sizeof(&pd) << endl;
```

**(b)**

```
int a[5] = {2,4,6,8,22};
int *p = &a[1];
cout << a[0] << " " << p[-1] << "\n";
COUT << a[1] << " " << p[0] << endl;
cout << a[2] << *(p+2) << "\n" << (a[4] == *(p+3)) << endl;
```

**(c)** Given the following lines of codes,

```
int ival = 1024;
int ival2 = 2048;
int* pi1 = &ival;
int* pi2 = &ival2;
int** pi3 = 0;
```

Are the following statements legal or illegal?

- (1) *pi2 = \*pi1;*
- (2) *ival = pi1;*
- (3) *pi3 = &pi2;*

**(d)** what is wrong?

```
int a = 58, *p = &a;
int **q = &p;
int ***r = &q;
q = &a;
s = &q;
```



# Activities (Short answer)

1. What is the primary use of pointer?
2. Where array of pointer can be used?
3. What makes pointer to be convenient for array manipulation?
4. Write a program to print 2x3 2-D array initialized with {45, 67, 75, 64, 83, 59} using a pointer.
5. Which parameter passing is better advantageous? Why?
6. Is type casting applicable in pointer? If yes, give an example.
7. Demonstrate the concept of pointers to function (function pointer).

# Practical Exercise

- 1) Write a program that declares and initializes three one-dimensional arrays named **price**, **quantity**, and **amount** in the `main()`. Each array is able to hold 10 double-precision numbers. Have your program pass these three arrays to a function called **extend()**, which calculates the elements in the **amount array** as the product of the equivalent elements in the **price** and **quantity** arrays using pointers. After **extend ()** has put values in the **amount array**, display the values in the **amount array** from within `main()` using for loop.
  - The numbers to be stored in **price** are 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, and 3.98.
  - The numbers to be stored in **quantity** are 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, and 4.8.

# Practical Exercise

- 2) Write a C++ function `swap` that takes the name of a 2D array, `num rows`, `num columns`, and two values `i` and `j` and swap the two rows `i` and `j`. All error checking must be done.
- 3) Write a function to find the largest and smallest of three numbers using pointer.
- 4) Write a program that store the string “**Vacation is near**” on character array named **message**. Include a function call that accepts **message** in a pointer argument named **strng** and then displays the contents of message by using the pointer.
- 5) Write a program in to print all permutations of a given string using pointers.

# Reading Assignment

- *Pointers to functions*
- *Purpose of pointer to functions*

# Reading Resources/Materials

## *Chapter 9:*

- ✓ Walter Savitch; Problem Solving With C++ [10th edition, University of California, San Diego, 2018]

## *Chapter 8:*

- ✓ P. Deitel , H. Deitel; C++ how to program, 10th edition, Global Edition (2017)

## *Chapter 12:*

- ✓ Gary J. Bronson; C++ for Engineers and Scientists (3rdEdition), 2010 Course Technology, Cengage Learning

---

Thank You  
For Your Attention!!

Any Questions

