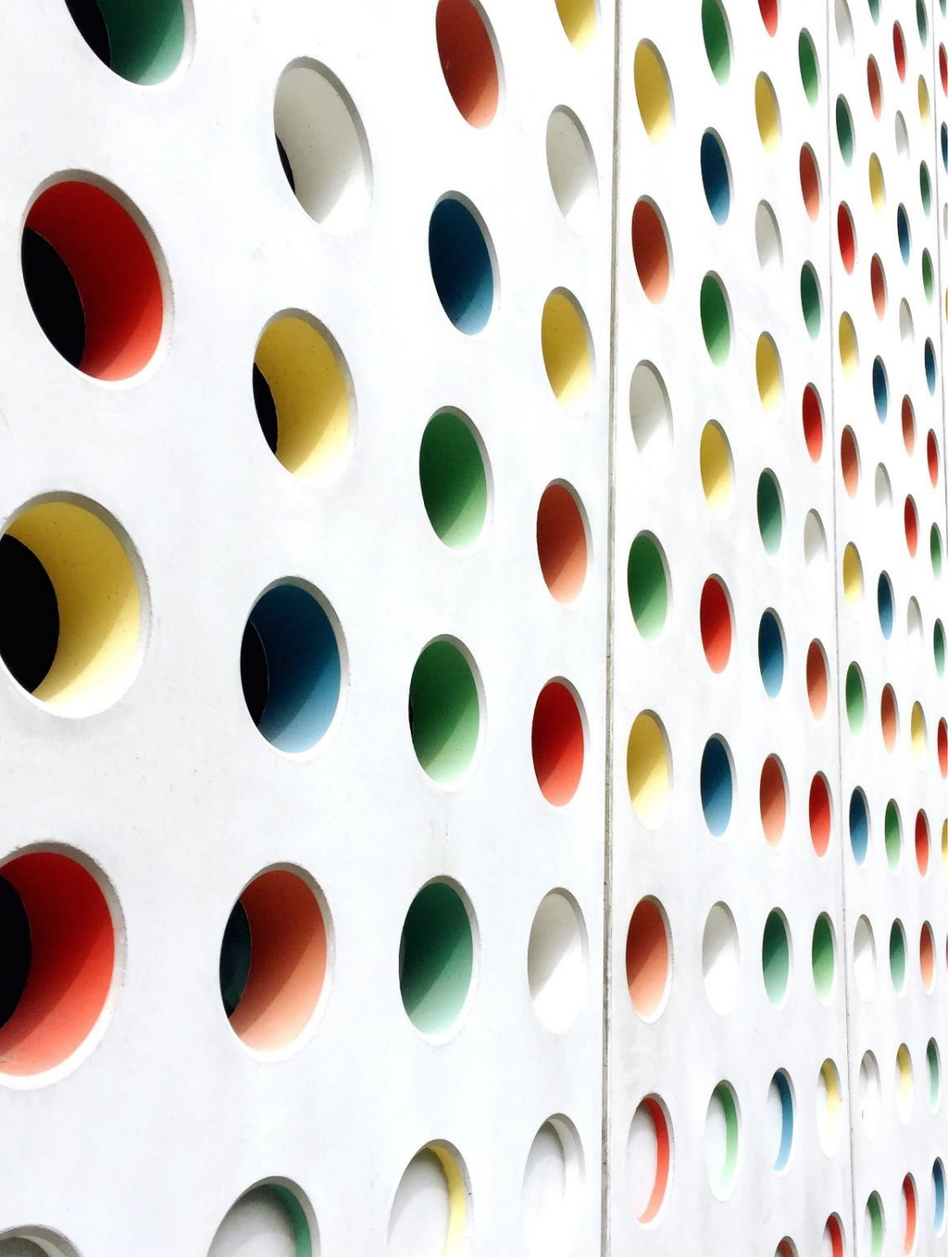


Chapter Two

Array, String & Pointers

Part: Two

Prepared by: | Sinodos G (MSc.)

Dire Dawa - 2017 E.C

Arrays

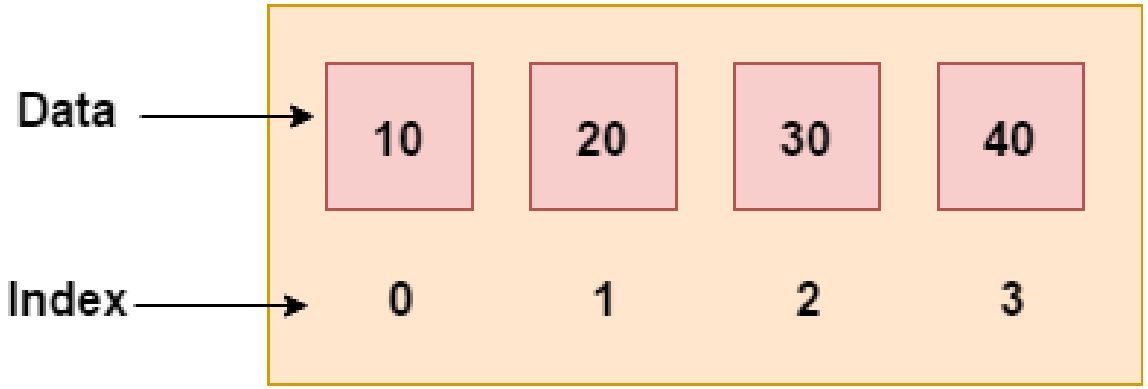
- a collection of elements, all of the same type, stored in contiguous memory locations.
  - It allows you to store multiple values in a single variable.
  - An array is a consecutive group of memory locations
  - total number of elements in the array is call the length of the array
- **Characteristics**
  - Fixed size
    - The size of the array is defined at the time of declaration and cannot be changed.
  - Access
    - Elements can be accessed using an index (e.g., array[i]).

- The memory locations in the array are known as elements of array
- arrays are objects, so they considered as a reference data types.
- size of an Array is “static”.
- an array can hold only one type of data!

Example:-

- int[] can hold only integers
- char[] can hold only characters

- Array indexing is always start from zero and highest address corresponds to last element



- Advantages of Array
  - Efficient Access
  - Simplicity
  - Memory Allocation
  - Code Optimization (less code)
  - Random Access
  - Easy Sorting and Searching
  - Easy understanding of program
- Disadvantages of Array
  - Fixed size
  - Lack of Flexibility

- **Array initialization**
  - the process of allocating memory for an array and optionally assigning values to its elements.
  - When you allocate an array, the elements are automatically initialized.

Example:

- Initializing an array with a fixed size

```
int array[5];
```

- Initializing an array with values

```
int array[] = {1, 2, 3, 4, 5};
```

- When you allocate an array, the elements are automatically initialized.
  - **Primitives**
    - Numeric primitives are zeroed,
    - Char primitives are made spaces,
    - Boolean primitives are made false
  - **References**
    - For an array of any other type the
    - “Rreferences” are made null.

Array Visualization

- Used to understand how arrays are structured and how their elements are accessed.
  - ***Index:*** 0 1 2 3 4
  - ***Values:*** 10, 20, 30, 40, 50

- Each element is stored at a specific index.
- Accessing an element is done using its index.
  - Example: array[2] gives you 30.

Array Declaration

- Involves defining an array and specifying its type, size, or initial values.
- use square brackets.
- To create an array object, you specify the type of the array elements and the number of elements as part of an array creation expression.
- ***Syntax:*** datatype[size] label;
  - Declaring an array with a fixed size

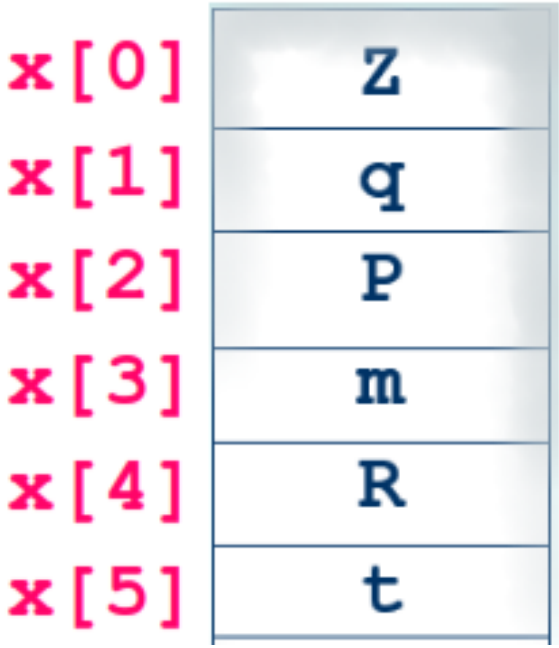
    int array[5];
  - Size 5, elements are uninitialized
  - Declaring and initializing an array

    int array[] = {1, 2, 3, 4, 5};
  - Size inferred

Array Indexes

- Array indexes are the positions of elements within an array.
- Crucial for accessing and manipulating array elements.
- Every compartment in an array is assigned an integer reference.
- This number is called the index of the compartment
- Array index starts from 0, and ends at n-1, where n is the size of the array

Example:



Modifying Array Elements

- Involves changing the value of specific elements in the array using their index.
- Straightforward and generally involves using the element's index
- The method for doing this can vary depending on the programming language.

```
int main() {
    int array[] = {10, 20, 30, 40, 50};
            // Modifying an element
    array[2] = 100;  // Changing the value at index 2
    for (int i : array) {
        cout << i << " ";  // Output: 10 20 100 40 50
    }
    return 0;
}
```

Accessing Array Elements

- To access an item in an array, type the name of the array followed by the item's index in square brackets.
- **Example:-** the expression:
  - names[0];
    - will return the first element in the names array
- Filling an Array:
  - Assign values to compartments:
    - prices[0]=6.75;
    - prices[1] = 80.43;
    - prices[2] = 10.02;

C++ Array Types

- arrays can be categorized into several types based on their characteristics and usage.

- There are Three types of arrays in C++ programming:
  - ***Single Dimensional Array***
    - It is a collection of elements of the **same data type** stored in a contiguous block of memory.
  - ***Multidimensional Array***
    - It is an array that contains one or more arrays as its elements.

- **Single-Dimensional Array**
  - simple structures used to store a list of values of the same data type.
  - a linear collection of elements of the same type, stored in contiguous memory locations.
  - provide a simple way to store and manipulate a collection of data

Example:

- The declaration and initialization of a single-dimensional array to store names of top rankers in a class.

- **Declaration**
  - To declare a single-dimensional array, you specify the type of elements and the size of the array.
  - Example:

```
int numbers[5]; // Declares an array of 5 integers
```

- **Initialization**
  - You can initialize an array at the time of declaration:

```
int numbers[5] = {1, 2, 3, 4, 5}; // Initializes the array with values
```

  - If you provide fewer values than the declared size, the remaining elements are initialized to zero

```
int numbers[5] = {1, 2}; // numbers[2], numbers[3], and numbers[4] will be 0
```

Example: C++ program that demonstrates the use of a single-dimensional array:

```
#include <iostream>
using namespace std;
int main() {
int numbers[5] = {1, 2, 3, 4, 5};

cout << "Array elements:" << endl;
for (int i = 0; i < 5; i++) {
cout << "numbers[" << i << "] = " << numbers[i] << endl;
}
cout << "Updated array elements:" << endl;
for (int i = 0; i < 5; i++) {
cout << "numbers[" << i << "] = " << numbers[i] << endl;
}    return 0;
}
```

```
Array elements:
numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
numbers[3] = 4
numbers[4] = 5
Updated array elements:

numbers[0] = 1
numbers[1] = 2
numbers[2] = 100
numbers[3] = 4
numbers[4] = 5
```

Multi-dimensional arrays

- allowing you to store data in a grid-like structure.
- The most common type is a two-dimensional array, which can be visualized as a matrix with rows and columns.
- particularly 2D arrays, are explained as structures to represent data in multiple dimensions.
- Each element of the 2-D array is accessed by providing two indexes:
  - a row index and a column index
  - called as a an array of arrays

- ***Example:***
  - the declaration, initialization, and printing of elements in a 2D array.

- A 2-d array is an array in which each element is itself an array
- It have a number of rows and columns

  int num[4][3];

- size of 2-D array
  - Total bytes= no of rows*no of columns*size of(base type)

	0	1
0	8	4
1	9	7
2	3	6

Declaration

- To declare a two-dimensional array, you specify the type of elements, followed by the number of rows and columns:

```
int matrix[3][4];
// Declares a 2D array with 3 rows and 4 columns
```

Initialization:

- You can initialize a two-dimensional array at the time of declaration.

```
int matrix[3][4] = {
                        {1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}
                    };
```

- If you provide fewer values than the specified size, the remaining elements are initialized to zero.

```
int matrix[3][4] = {
                        {1, 2},
                        {5, 6},
                        {9, 10}
                    };

// The remaining elements will be 0
```

- Accessing Elements
  - You can access elements using their row and column indices, which start at 0:

```
int value = matrix[1][2]; // Accesses the element in the 2nd row, 3rd column (value 7)
```

Modifying Elements

- You can modify the value of a specific element in the array:

  - Example:

    matrix[0][3] = 100; // Changes the element in the 1st row, 4th column to 100

- Example:

```
#include<iostream>
#include<conio.h>
using namespace std;
int main(){
 int aa[5][2] = { {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6} };
 int i, j;
 for(i=0; i<5; i++)
 {
  for(j=0; j<2; j++)
  {
   cout<<"aa["<<i<<"]["<<j<<"] = "<<aa[i][j]<<"\n";
  }
 }
 getch();
}
```

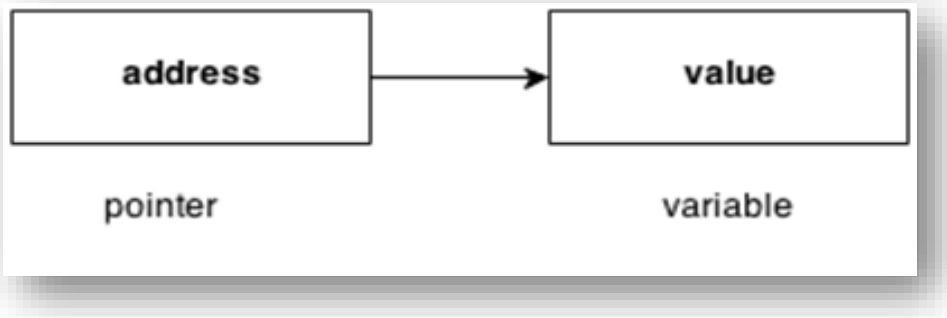
Example

```
#include <iostream>
using namespace std;
int main() {
// Declare and initialize a 2D array
int matrix[3][4] = {
{1, 2, 3, 4},
{5, 6, 7, 8},
 {9, 10, 11, 12}
 };
cout << "Matrix elements:" << endl;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 4; j++) {
cout << "matrix[" << i << "][" << j << "] = " <<
matrix[i][j] << " ";
}
cout << endl;
}
```

```
matrix[1][2] = 100;
cout << "\nUpdated matrix elements:" << endl;
for (int i = 0; i < 3; i++) {
for (int j = 0; j < 4; j++) {
cout << "matrix[" << i << "][" << j << "] = " <<
matrix[i][j] << " ";
 }
cout << endl;
 // New line after each row
 }
return 0;
}
```

Pointers

- variables that store the memory addresses of other variables.
- used to store addresses rather than values.
- It is also known as locator or indicator that points to an address of a value.
- Every ***variable*** has an associated location in the memory, which we call the memory address of the variable.
- If we have a variable ***aa*** in our program, **&aa** returns its memory address.



Cont'd ...

- In C++ pointers are used to perform a dynamic memory allocation

```
int main() {
    int a;
    char b[10];

    cout << "Address of a: ";
    cout << &a<< endl;
    cout << "Address of b: ";
    cout << &b<< endl;
   getch();
}
```

```
Address of a: 0x6ffe1c

Address of b: 0x6ffe10
```

Pointers . . .

- a variable whose value is the address of another variable.
- 1st declare a pointer before you can work with it.

Syntax:-

type *var-name;

- Following are the valid pointer declaration:-
  - int *ip; // pointer to an integer
  - double *dp; // pointer to a double
  - float *fp; // pointer to a float
  - char *ch // pointer to character

Using Pointers in C++

- There are few important operations, which we will do with the pointers very frequently.
  - define a pointer variable.
  - assign address of a variable to a pointer.
  - Finally access the value at the address available in the pointer variable.

- This is done by using unary **operator *** that returns the value of the variable located at the address specified by its operand.

Advantage of pointer

- Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc.
- used with arrays, structures and functions.
- we can return multiple values from function using pointer.
- It makes you able to access any memory location in the computer's memory.

Cont'd ...

a variable that **stores the memory address as its value**.

Example

```
int main () {
int a =20;
    int *b;
    b= &a;
    cout << "Value of a: ";
    cout <<a<< endl;
    // print the address
    cout <<"Address of b: ";
    cout <<b << endl;
    // access the value
    cout <<"Value of *b: ";
    cout <<*b<< endl;
getch();
}
```

```
Value of a: 20
Address of b: 0x6ffe14
Value of *b: 20
```

Pointer Operators

- **Dereference operator (*)**
  - Represented as variable and * operator
  - Used to dereference of a variable
  - There are three ways to declare pointer variables,

```
string* data; // Preferred
string *data;
string * data;
```

- **Reference Operator (&)**
  - Called as address operators
  - Used to display the address of a variable

Cont'd ...

- The address of variable to pointer we use **ampersand symbol** (&).
- They have data type just like variables
- Pointers are used to store addresses rather than values.

Changing Value Pointed by Pointers

- If ***pointVar*** points to the address of var, we can change the value of var by using ****pointVar.***

- **Example:-**

```
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Cont'd ..

Exercise:

- Create a pointer variable with the name ptr, that should point to a string variable named food:

```
String food = "Pizza";
string *ptr = &food;
```

Example

```
using namespace std;
    int main(){
        //Pointer declaration
        int *p, num=101;
        //Assignment
        p = &num;
        cout<<"\nAddress of num:"<<&num;
        cout<<"\nAddress of num:"<<p;
        cout<<"\nAddress of p: "<<&p;
        cout<<"\nValue of num: "<<*p;
        cout<<"\nValue of num: "<<num;
    getch();
}
```

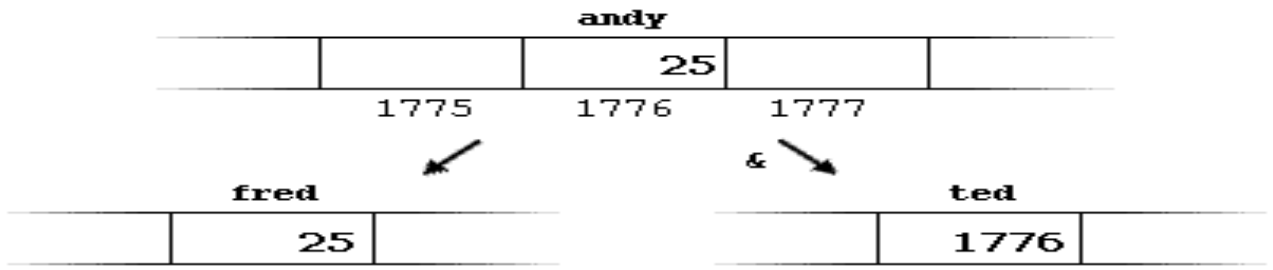
```
Address of num:0x6ffe04
Address of num:0x6ffe04
Address of p: 0x6ffe08
Value of num: 101
Value of num: 101
```

Reference Operator (&)

- Refer to the specific location in memory (its memory address).
- The address that locates a variable within memory is what we call a reference to that variable.
- Called an ampersand sign (&) symbol
- Also it is known as reference operator

int num=$data;

- The values contained in each variable after the execution of this, are shown in the following diagram:



- First, we have assigned the value 25 to andy (a variable whose address in memory we have assumed to be 1776).

- The second statement copied to fred the content of variable andy (which is 25).

Cont'd ...

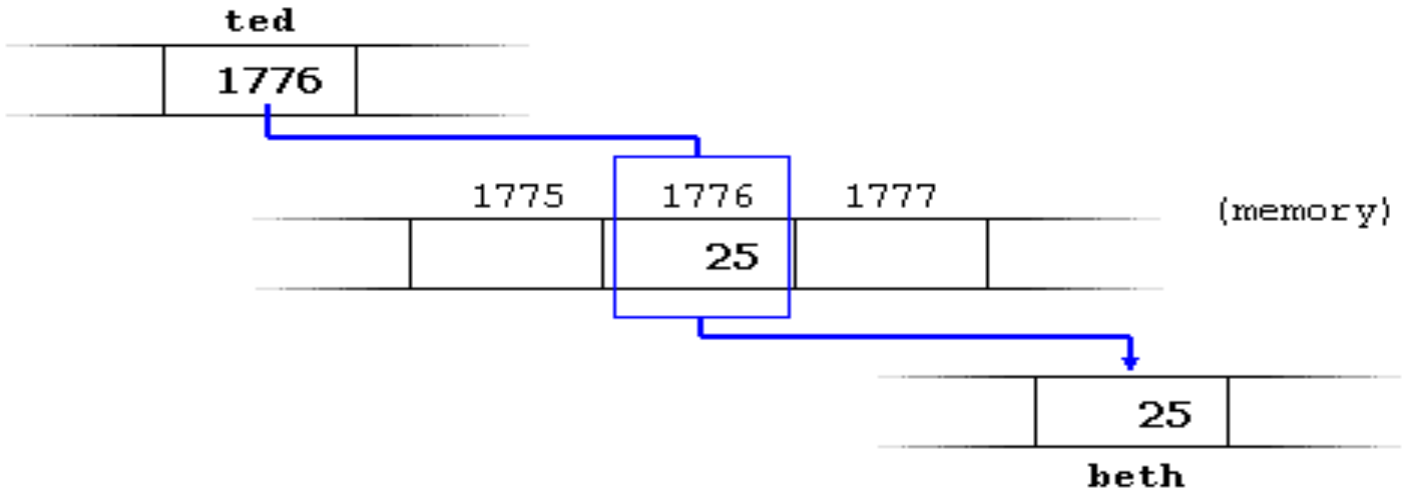
- **Finally . . .**
- ***The 3rd*** statement copies to ted not the value contained in andy but a reference to it (i.e., its address, which we have assumed to be 1776).
  - The reason is that in this third assignment operation we have preceded the identifier andy with the reference operator (&),
  - so we were no longer referring to the value of andy but to its reference (its address in memory).

Dereference Operator (*)

- translated to "value pointed by".

```
beth = *ted;
```

- (that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



while *ted (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25.

```
1 beth = ted;   // beth equal to ted ( 1776 )
2 beth = *ted;  // beth equal to value pointed by ted ( 25 )
```

- Difference between reference and dereference operators:
  - **&** is the reference operator and can be read as "address of"
  - * is the dereference operator and can be read as "value pointed by"
- they have opposite meanings.
- A variable referenced with & can be dereferenced with *.

- Right after these two statements, all of the following expressions would give true as result:

```
1 andy = 25;
2 ted = &andy;
```

```
1 andy == 25
2 &andy == 1776
3 ted == 1776
4 *ted == 25
```

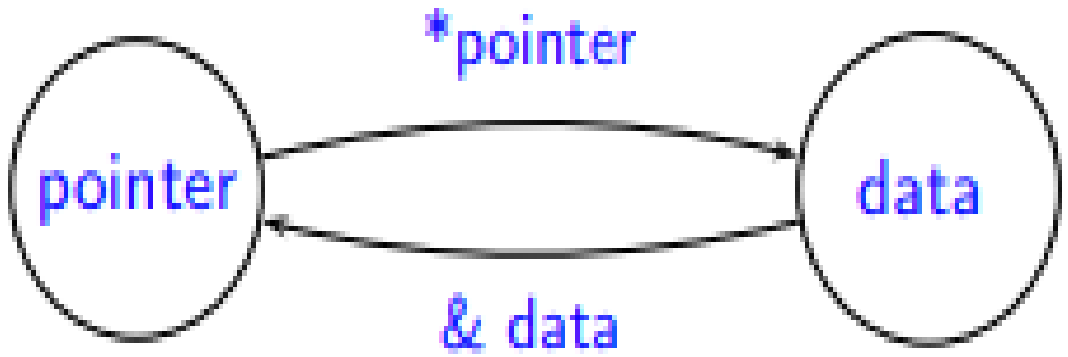
```
*ted == andy
```

Cont'd ...

- a variable which stores a reference to another variable is called a pointer.
- Pointers are said to "point to" the variable whose reference they store.
- Using a pointer we can directly access the value stored in the variable which it points to.

format:

```
Type * name;
```



Pointers and Arrays

- the address of the first element of the array in variable.
- Example: supposing these two declarations:

Consider this example:

```
int *ptr;
int arr[5];

// store the address of the first
// element of arr in ptr
ptr = arr;
```

- an array can be considered a constant pointer.
- Therefore, the following allocation would not be valid:

Cont'd ...

- Notice that we have used arr instead of &arr[0]. This is because both are the same. So, the code below is the same as the code above.
  - int *ptr;
  - int arr[5];
  - ptr = &arr[0];
- The addresses for the rest of the array elements are given by &arr[1], &arr[2], &arr[3], and &arr[4].

- while assigning the address of array to pointer don't use ampersand sign(&)

Cont'd . . .

```
using namespace std;
int main(){
    //Pointer declaration
    int *p;
    //Array declaration
    int arr[]={11, 22, 23, 34, 15, 26};
    //Assignment
    p = arr;
    for(int i=0; i<6;i++){
        cout<<"\n value: "<<*p;
        p++;
    }
getch();
}
```

Point to Every Array Elements

- Suppose we need to point to the fourth element of the array using the same pointer ptr.
- Here, if ptr points to the first element in the above example then ptr + 3 will point to the fourth element.

```
int *ptr;
int arr[5];
ptr = arr;

ptr + 1 is equivalent to &arr[1];
ptr + 2 is equivalent to &arr[2];
ptr + 3 is equivalent to &arr[3];
ptr + 4 is equivalent to &arr[4];
```

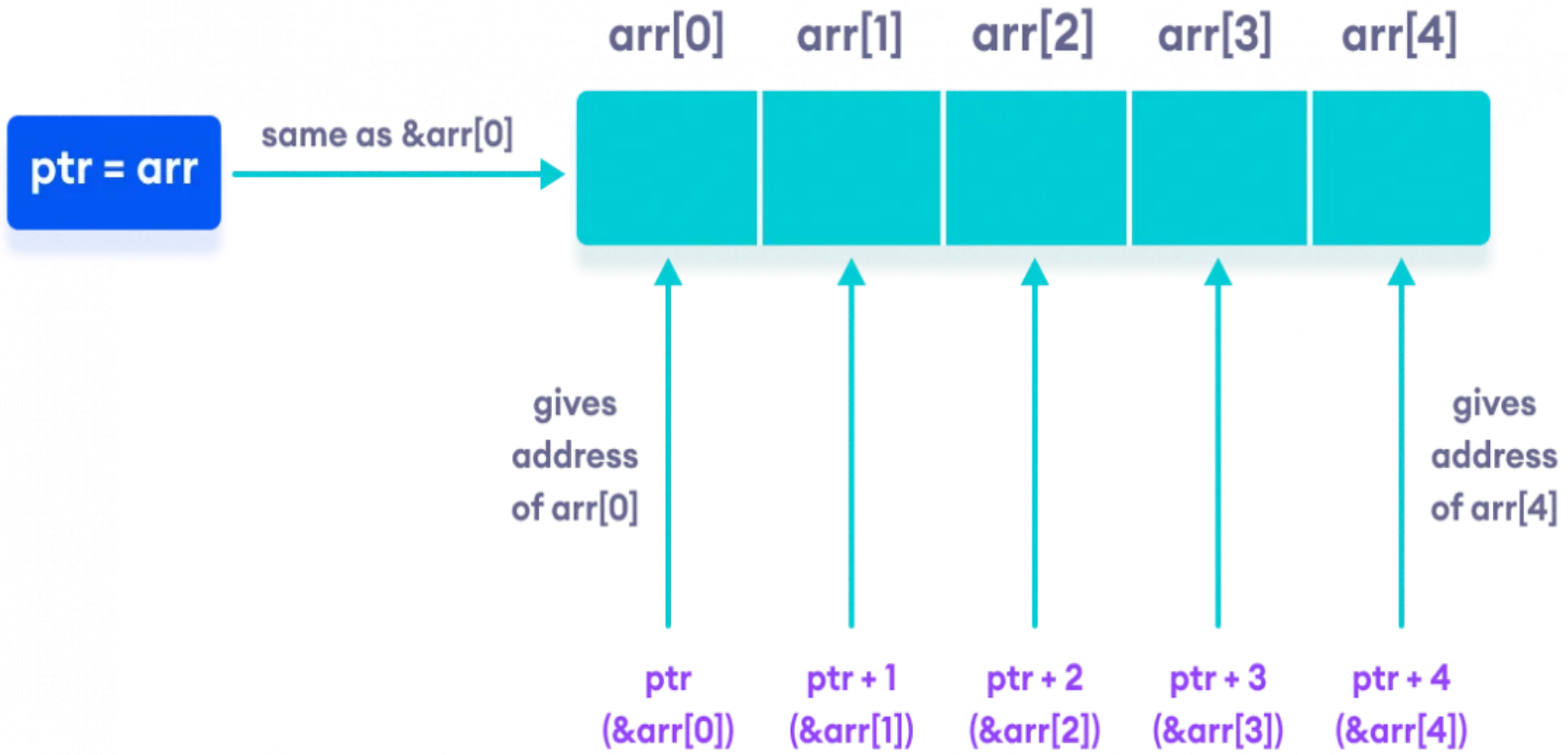
Similarly, we can access the elements using the single pointer. For example,

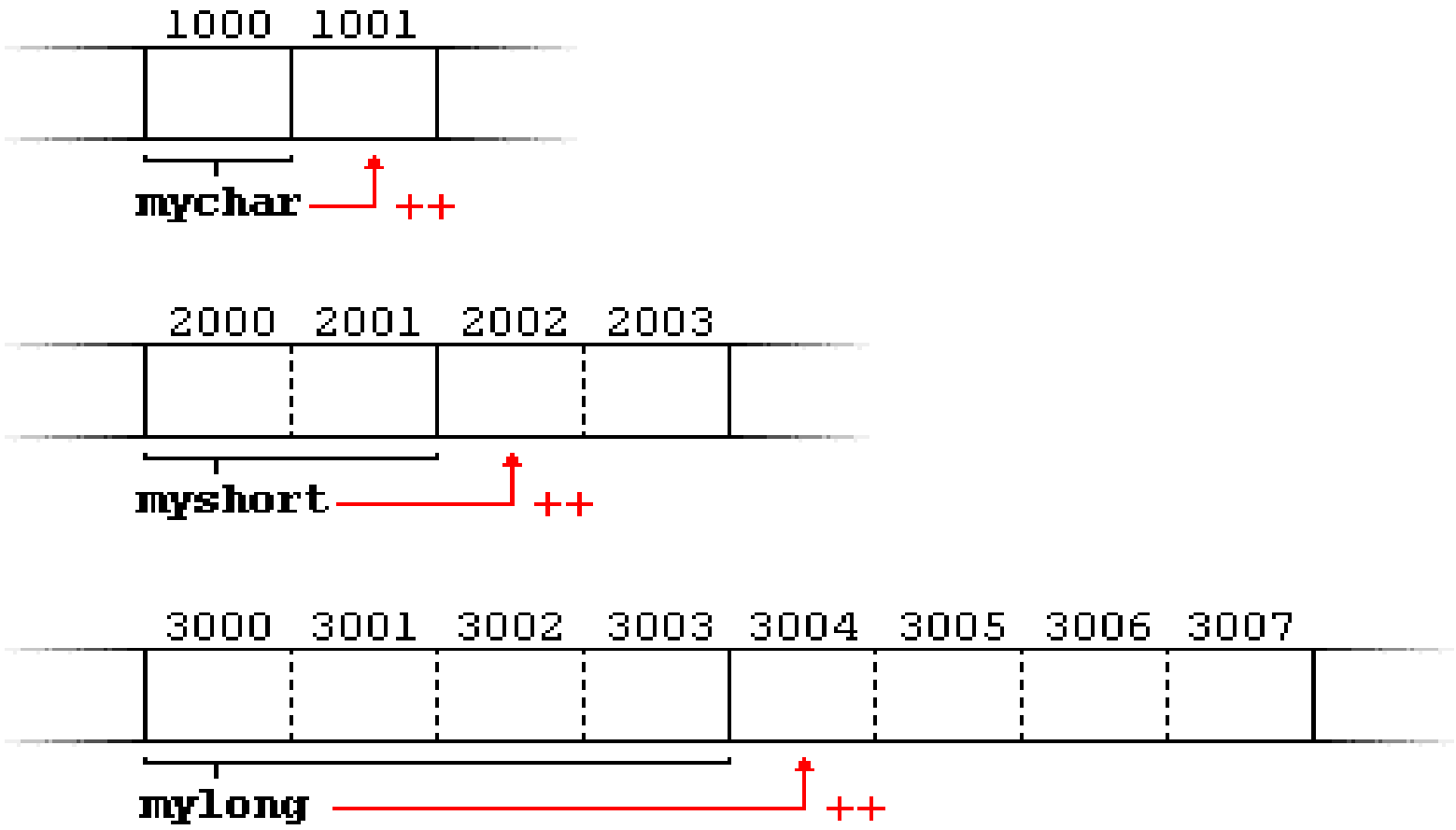
```
// use dereference operator
    *ptr == arr[0];
    *(ptr + 1) is equivalent to arr[1];
    *(ptr + 2) is equivalent to arr[2];
    *(ptr + 3) is equivalent to arr[3];
    *(ptr + 4) is equivalent to arr[4];
```

Suppose if we have initialized ptr = &arr[2]; then

```
ptr - 2 is equivalent to &arr[0];
ptr - 1 is equivalent to &arr[1];
ptr + 1 is equivalent to &arr[3];
ptr + 2 is equivalent to &arr[4];
```

Cont'd ...


arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
ptr = arr
same as &arr[0]
gives address of arr[0]
gives address of arr[4]
ptr (&arr[0])
ptr + 1 (&arr[1])
ptr + 2 (&arr[2])
ptr + 3 (&arr[3])
ptr + 4 (&arr[4])

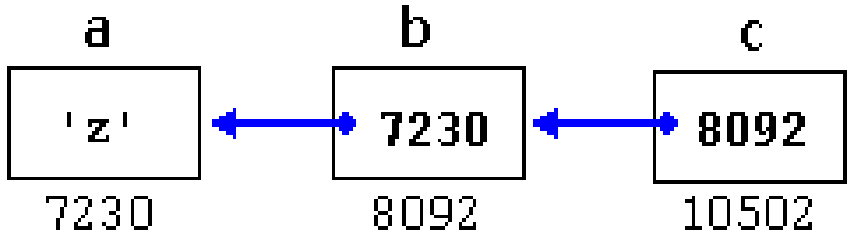

1000 1001
mychar
++
2000 2001 2002 2003
myshort
++
3000 3001 3002 3003 3004 3005 3006 3007
mylong
++

Pointers to Pointers

- C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers).
- In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
1 char a;
2 char * b;
3 char ** c;
4 a = 'z';
5 b = &a;
6 c = &b;
```

- This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



- The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

- The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

  - c has type char** and a value of 8092
  - *c has type char* and a value of 7230
  - **c has type char and a value of 'z'

Void Pointers

- A special type of pointer.
- void represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).
- This allows void pointers to point to any data type, from an integer value or a float to a string of characters.
- A void pointer is a special type of pointer that can point to somewhere without a specific type.

Null Pointer

- a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address.
- any pointer may take to represent that it is pointing to "nowhere"

Thank you!

Question