Pointer Variables
Lecture 2

OOP/Computer Programming

Instructor: Dr. Danish Shehzad

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

int main()
{
      int var1 = 3;
      int var2 = 24;
      int var3 = 17;
      cout << &var1 << endl;
      cout << &var2 << endl;
      cout << &var3 << endl;
}
```

Output

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

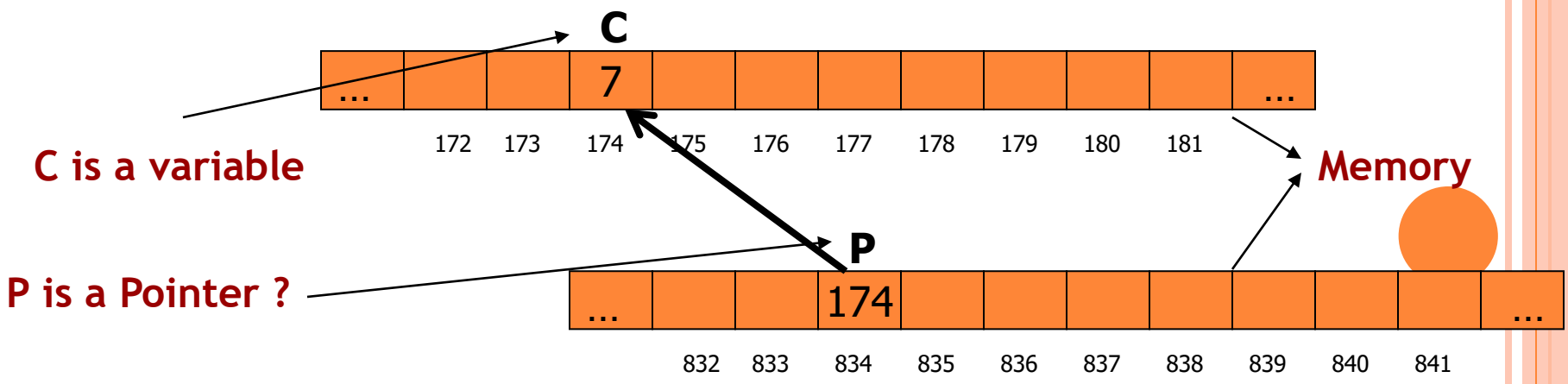
Pointer Variable Declarations and Initialization

- **Why we need Pointers:**
  - Accessing array elements
  - Passing arrays and strings as arguments
  - Creating data structures such as linked lists
  - Dynamic memory allocation, which can grow at runtime

Pointer Variable Declarations and Initialization

The main difference between normal variables and pointers is that, variables contain a value, while pointers contain a memory address.

- Memory address:
  - We know each memory address contain some value.
  - Thus if pointers contain memory addresses, we can get its value (indirectly).



Pointers

- A pointer variable must be declared before it can be used.

- Examples of pointer declarations:

```
int  *a;
float *b;
char *c;
```

- The **asterisk**, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data **(the value that we get indirectly)** that the pointer points to.

Referencing

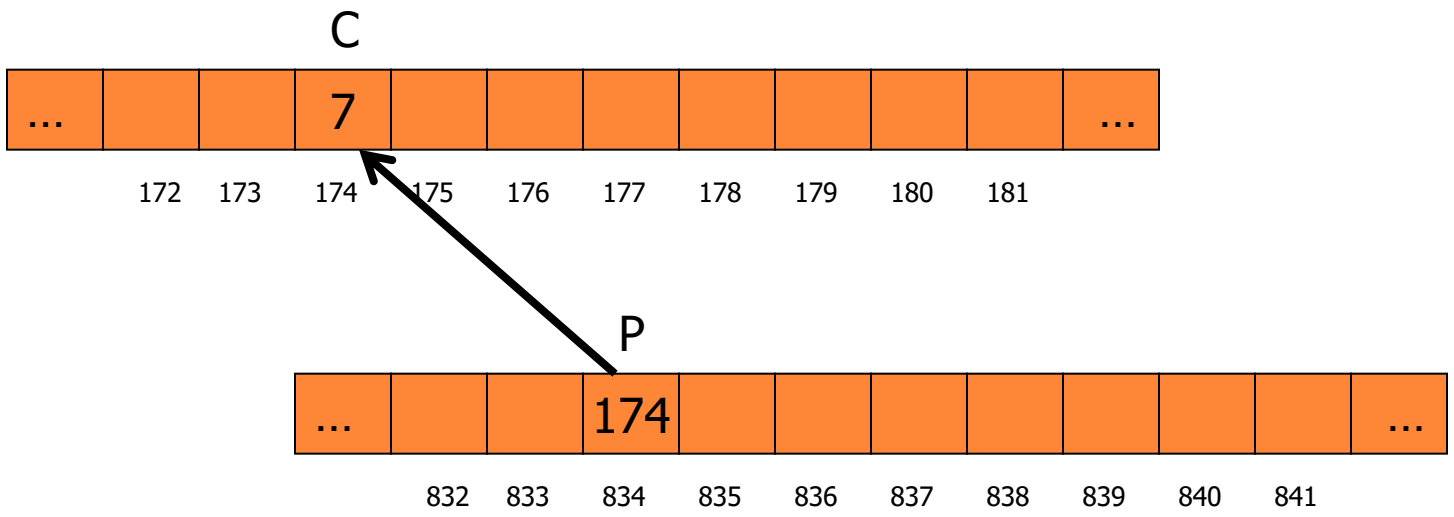
- The unary operator **&** gives the address of a variable
- The statement

```
P = &C;
```

- assigns the address of **C** to the variable **P**, and now **P points to C**

Referencing

```
int C;
int *P; /* Declare P as a pointer to int */
C = 7;
P = &C;
```



Dereferencing

- The unary operator ***** is the dereferencing operator

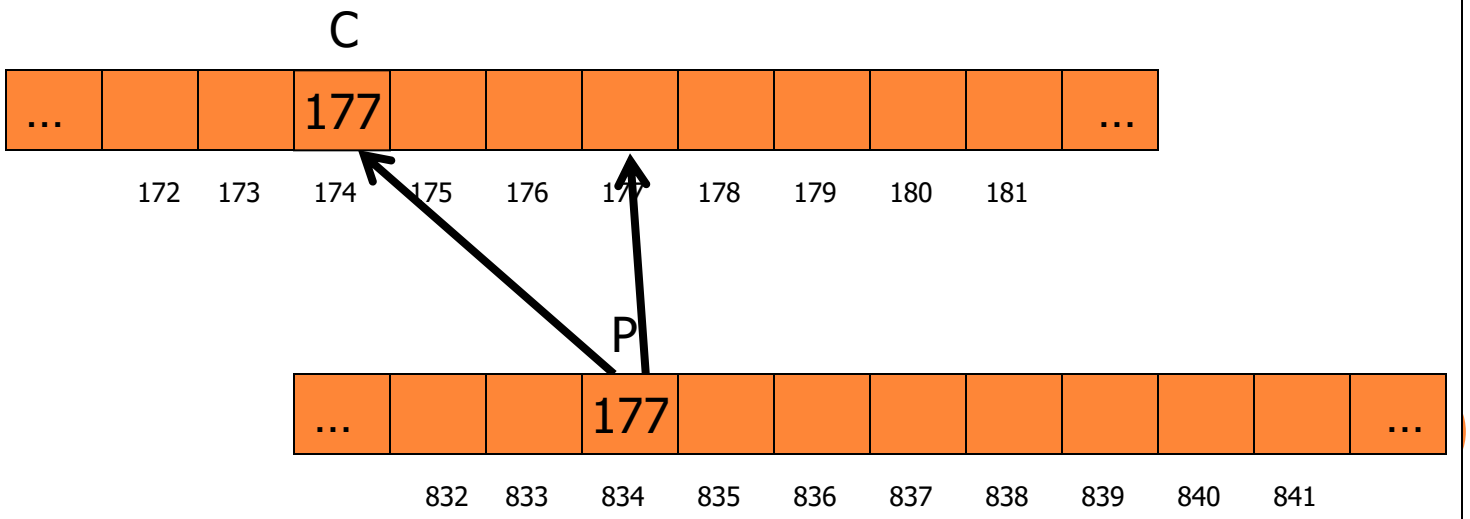
- Applied on pointers
- Access the **value of object** the pointer points to
- The statement

  ***P = 5;**

  puts in **C** (the variable pointed by **P**) the value 5

Dereferencing

```
cout << *P; /* Prints out '7' */
*P = 177;
P = 177; /* This is unadvisable!! */
```

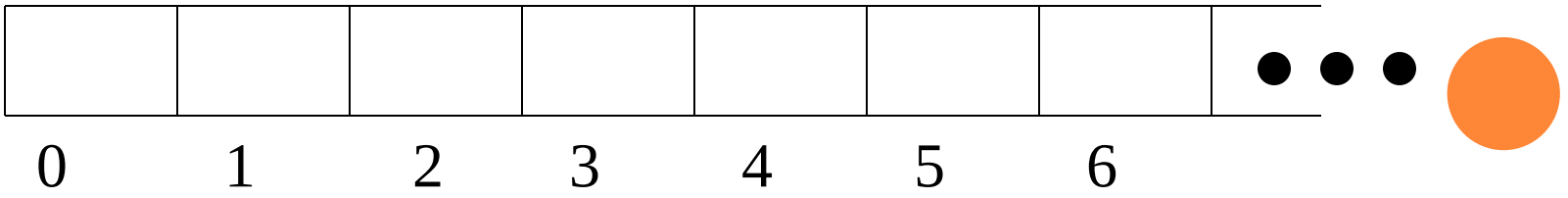


Review

- All data (software instructions, variables, arrays) is in memory.
- Each memory location has an address.
- A pointer is a C version of the address.
  - * denotes we are accessing or storing the value in the address of the pointer.
  - & denotes we are accessing or storing the memory address in the pointer.
- If any pointer knows the address of any memory address. It can easily change its value (virus concept).

Bits and Bytes

- Memory is divided into bytes, each of which are further divided into bits
  - Each bit can have a value of 0 or 1
  - A byte is eight bits
    - Can hold any value from 0 to 255
  - Memory can be thought of as a sequence of bytes, numbered starting at 0



Storing Variables

- Each variables is stored in some sequence of bytes
  - Number of bytes depends on what?
  - Number of bytes depends on the data type
  - Can two variables share a byte?
  - Two variables will never share a byte – a variable uses all of a byte, or none of it
- Example:
  - An int is usually stored as a sequence of four bytes

Using the Address

- This holds true for every language – each variable has to be stored somewhere
  - In C/C++, you can get and use the address
    - For any variable **x**, **&x returns the address of x**

Pointers & Allocation (1/2)

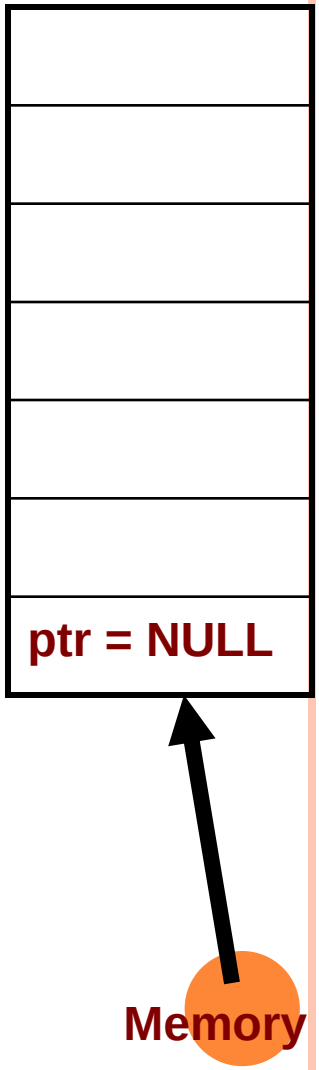
- After declaring a pointer:

```
int *ptr1;
```

`ptr1` doesn't actually point to anything yet. So its address is NULL.

. We can either:

- make it point to something that already exists,
  - `ptr1 = &c`
- or
- allocate room in memory for something new that it will point to… (dynamic memory will discuss later)



Pointers & Allocation (2/2)

- Pointing to something that already exists:

```
int *ptr, var1, var2;
var1 = 5;
ptr  = &var1;
var2 = *ptr;
```

- `var1` and `var2` have room implicitly allocated for them.

More C/C++ Pointer Dangers

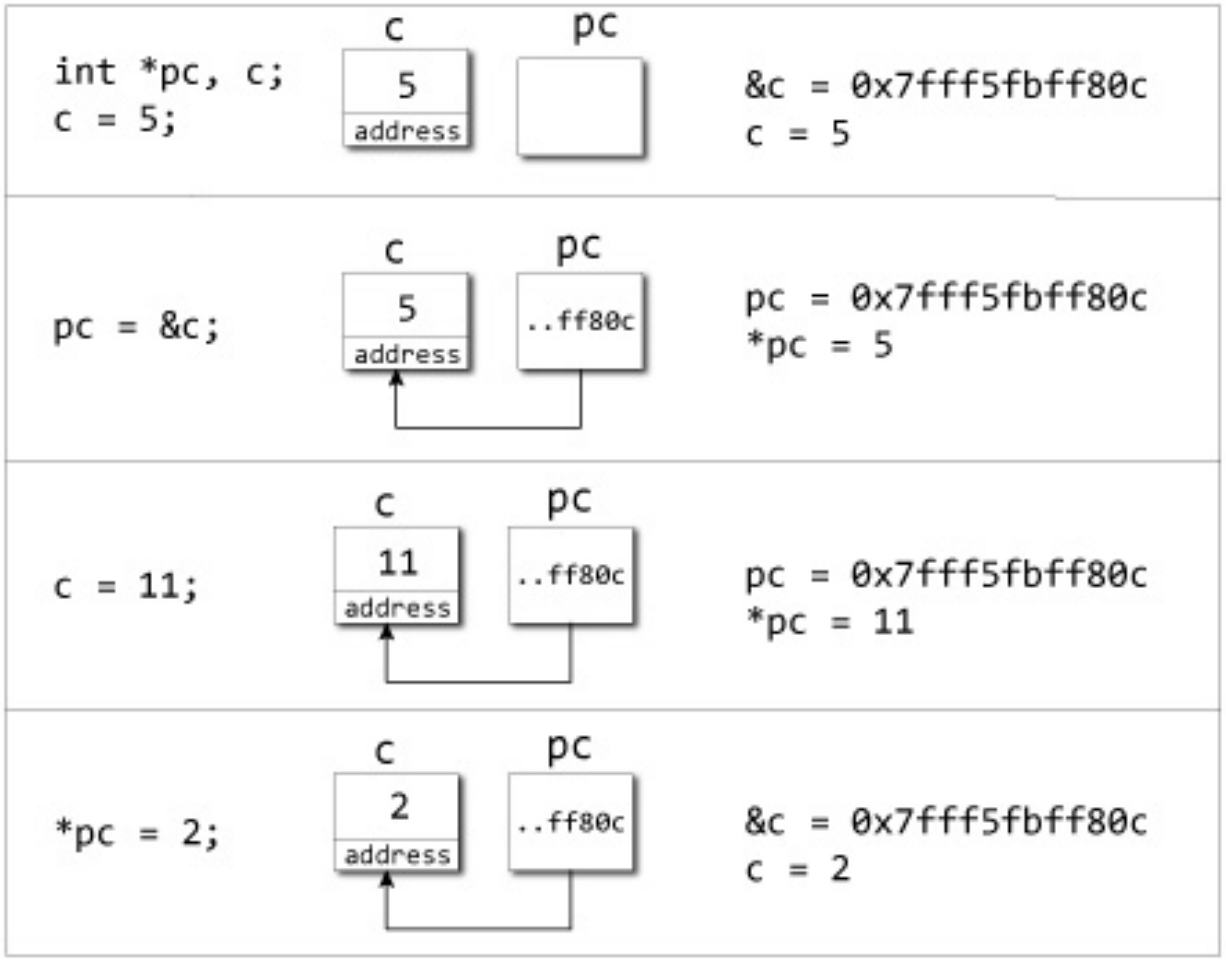
- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to! Thus the address is NULL.

- **What does the following code do?**
- We can't store anything in the pointer **(ptr)** unless **ptr** contains some address.

```
void f()
{
    int *ptr;
    *ptr = 5;
}
```

```cpp
int main() {
    int *pc, c;

    c = 5;
    cout << "Address of c (&c): " << &c <<
    cout << "Value of c (c): " << c << end

    pc = &c;     // Pointer pc holds the me
    cout << "Address that pointer pc holds (pc): "<< pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    c = 11;     // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    return 0;
}
```

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

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

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

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


int *pc, c;
c = 5;
c
5
address
pc
&c = 0x7fff5fbff80c
c = 5
pc = &c;
c
5
address
pc
..ff80c
pc = 0x7fff5fbff80c
*pc = 5
c = 11;
c
11
address
pc
..ff80c
pc = 0x7fff5fbff80c
*pc = 11
*pc = 2;
c
2
address
pc
..ff80c
&c = 0x7fff5fbff80c
c = 2

Arithmetic on Pointers

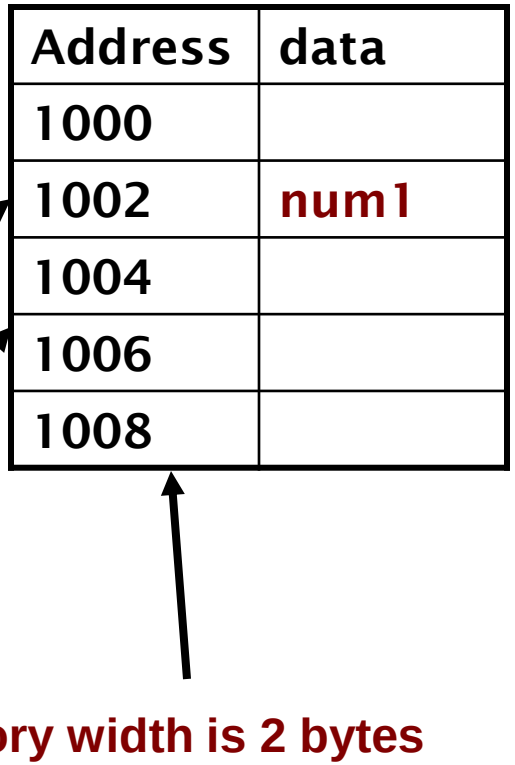
- A pointer may be incremented or decremented.
- This means only address in the pointer is incremented or decremented.
- An integer (can be constant) may be added to or subtracted from a pointer.
- Pointer variables may be subtracted from one another.
- Pointer variables can be used in comparisons, but usually only in a comparison to pointer variables or NULL.

Arithmetic on Pointers

- When an integer **(n)** is added to or subtracted from a pointer **(ptr)**
  - The new pointer value **(ptr)** is changed by the **ptr** address **plus (+) n multiple (*)** the (bytes of **ptr** data type).
  - **ptr** + **n** * (bytes of **ptr** data type)
- Example
  - int *ptr;
  - ptr = 1000;
  - ptr = ptr + 2;
  - // ptr address is now changed to 1000 + 2*4 (because integer consumes 4 bytes) New address is now 1008

Arithmetic on Pointers

- Example (2)
  - double *ptr;
  - ptr = 1000;
  - ptr++; **1000 + 1*8 = 1008**
- Example (3)
  - float *ptr;
  - ptr = 1000;
  - ptr+=3; **1000 + 3*4 = 1012**
- Example (4)
  - int *ptr;
  - int num1 = 0;
  - ptr = &num1;
  - ptr++; **1002 + 4 = 1006**



Arithmetic on Pointers

- Example (5)

```
void main (void)
{
    int *pointer1, *pointer2;
    int num1 = 93;
    pointer1 = &num1; //address of num1
    pointer2 = pointer1; // pointer1 address
      is assign to pointer2

}
```

Address	data
1000	
1002	num1 = 93
1004	
1006	pointer1 = 1002
1008	pointer2 = 1002

Logical operators on Pointers

- We can apply logical operators (<, >, <=, >=, ==, !=) on pointers.
  - But remember pointers can be compared to pointers or
  - NULL

- Example (6)
  - int *pointer1, *pointer2; // both pointer2 contains NULL addresses
  - int num1 = 93;
  - If (pointer1 == NULL) // pointer compared to NULL
  - pointer1 = &num1;
  - pointer2 = &num1;
  - If (pointer1 == pointer2) // pointer compared to pointer
    - printf("Both pointers are equal");

```cpp
#include <iostream>

int main()
{
    int array [5] = { 9, 7, 5, 3, 1 };

    std::cout << &array[1] << '\n';
    std::cout << array+1 << '\n';

    std::cout << array[1] << '\n';
    std::cout << *(array+1) << '\n';

    return 0;
}
```

```
0017FB80
0017FB80
7
7
```

Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name is like constant pointer
  - All arrays elements are placed in the consecutive locations. (This is only valid in static memory allocation)
    - **Example:-** **int List [10];**
  - Pointers can do array subscripting operations (We can access array elements using pointers).
    - **Example:-** **int value = List [2];**

Relationship Between Pointers and Arrays (Cont.)

- Accessing array elements with pointers
  - Assume declarations:

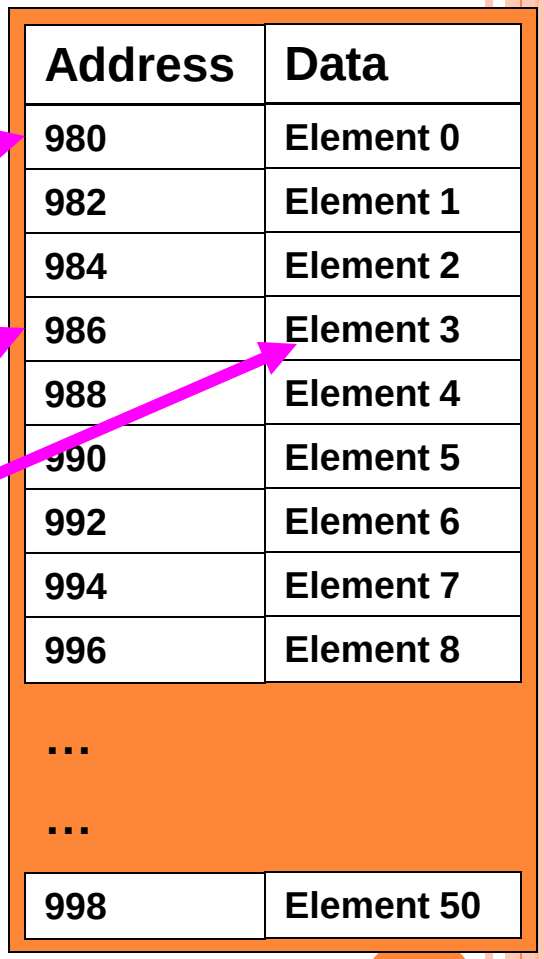
```
int List[ 5 ];
int *bPtr;
bPtr = List;
```

Effect:-

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

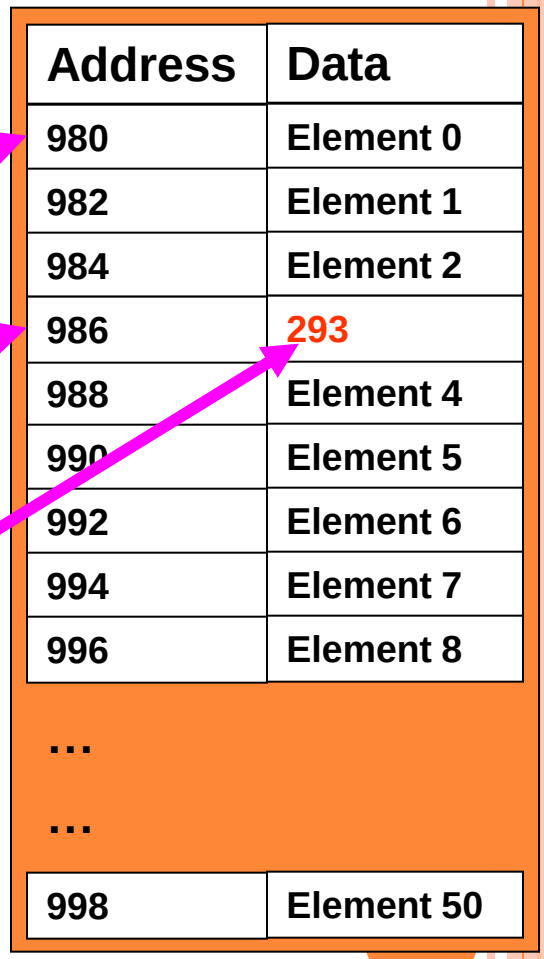
Accessing 1-Demensional Array Using Pointers

- We know, Array name denotes the memory address of its first slot.
  - Example:
    - **int List [50];**
    - **int *Pointer;**
    - **Pointer = List;**
- Other slots of the Array (List [50]) can be accessed using by performing Arithmetic operations on Pointer.
- For example the **address** of (element 4^{th}) can be accessed using:-
  - int *Value = Pointer + 3;
- The **value** of (element 4^{th}) can be accessed using:-
  - int Value = *(Pointer + 3);



Accessing 1-Demensional 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-Demensional Array

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

```
...
...
    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
982	Element 1
984	Element 2
986	Element 3
988	Element 4
990	Element 5
992	Element 6
994	Element 7
996	Element 8
...	
...	
998	Element 50

This is Equivalent to

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

- Some Examples (CodeProject)

- *http://www.codeproject.com/Articles/11560/Pointers-Usage-in-C-Beginners-to-Advanced*

- Questions?