

# System Design and Programming II

CSCI – 1943

David L. Sylvester, Sr., Professor



# Chapter 9

Pointers

# Pointers – Getting the Address of a Value

The address operator (&) returns the memory address of a variable.

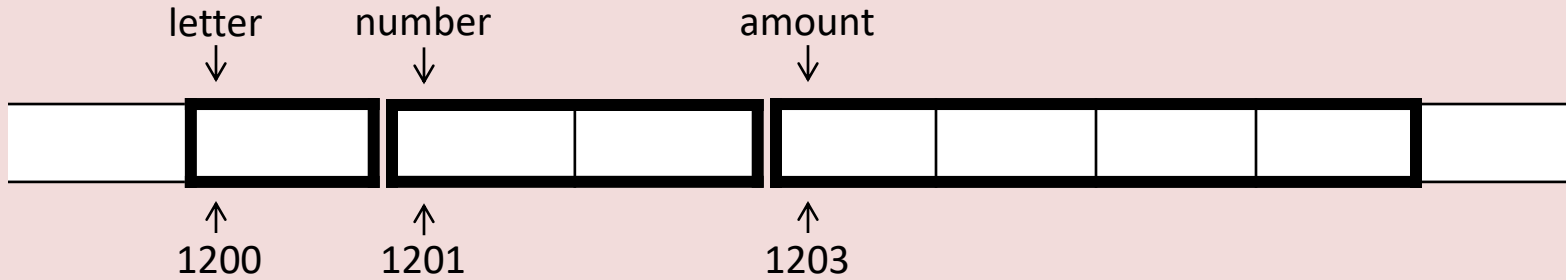
Every variable is allocated a section of memory large enough to hold a value of the variable data type.

## Commonly

|                  |             |
|------------------|-------------|
| char             | one bytes   |
| shorts           | two bytes   |
| int, long, float | four bytes  |
| double           | eight bytes |

Each byte of memory has a memory address. A variable's address is the address of the first byte allocated to that variable.

Ex: char letter;  
short number;  
float amount;



The addresses of the variables above are used as an example. Getting the address of a variable is accomplished by using the (&) operator in front of the variable name. It allows the system to return the address of that variable in hexadecimal.

```
Ex:    &amount           //returns the variable's address  
      cout << &amount    // displays the variable's address
```

**Note: Do not confuse the address operator with the & symbol used when defining a referenced variable.**

# Pointers – Sample Program

```
// This program uses the & operator to determine a variable's  
// address and the sizeof operator to determine its size.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x = 25;
```

```
    cout << "The address of x is " << &x << endl;
```

```
    cout << "The size of x is " << sizeof(x) << " bytes\n";
```

```
    cout << "The value of x is " << x << endl;
```

```
}
```

## Sample output

```
The address of x is 001EFE68  
The size of x is 4 bytes  
The value of x is 25
```

# Pointers Variables

Pointer variables or pointers, are special variables that hold a memory address. Just as int variables are designed to hold integers, pointers variables are designed to hold memory addresses. Pointers variables allow you to indirectly manipulate data stored in other variables.

Memory addresses identify specific locations in the computer's memory. Because a pointer variable holds a memory address, it can be used to hold the location of some other piece of data. (i.e. points to some piece of data that is stored in the computer's memory). Pointer variable allows you to work with data that they point to.

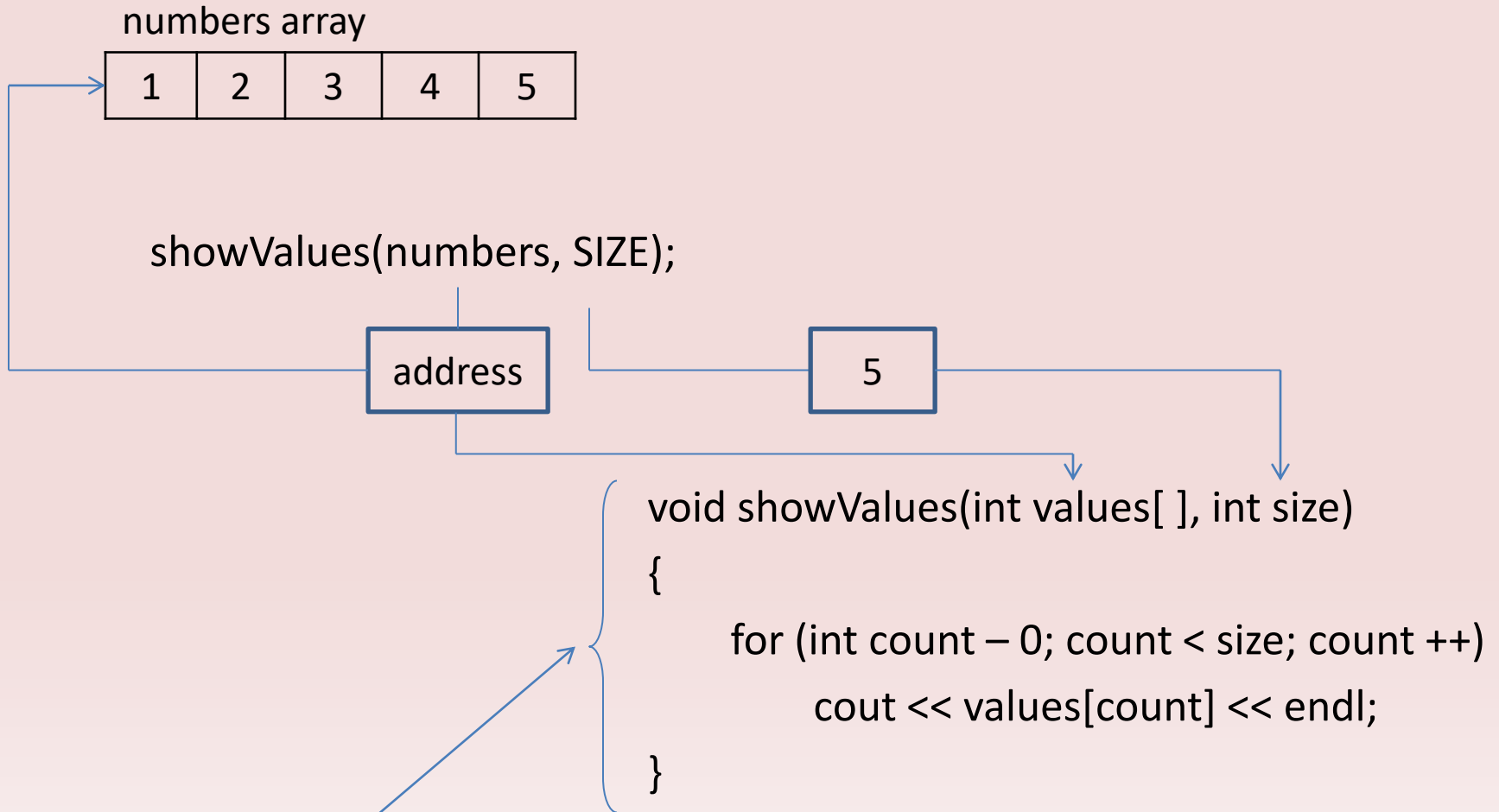
```
Ex:   const int SIZE = 5;
      int numbers[SIZE] = {1, 2, 3, 4, 5};
      showValues(numbers, SIZE);
```

Here we are passing the name of the array, numbers, and its size as arguments to the showValues function.

### showValues Defined

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

The values parameter receives the address of the numbers array. It works like a pointer because it “points” to the number array.



Inside the `showValues` function, anything that is done to the `values` parameter is actually done to the `numbers` array. (*values parameter referenced the numbers array*)

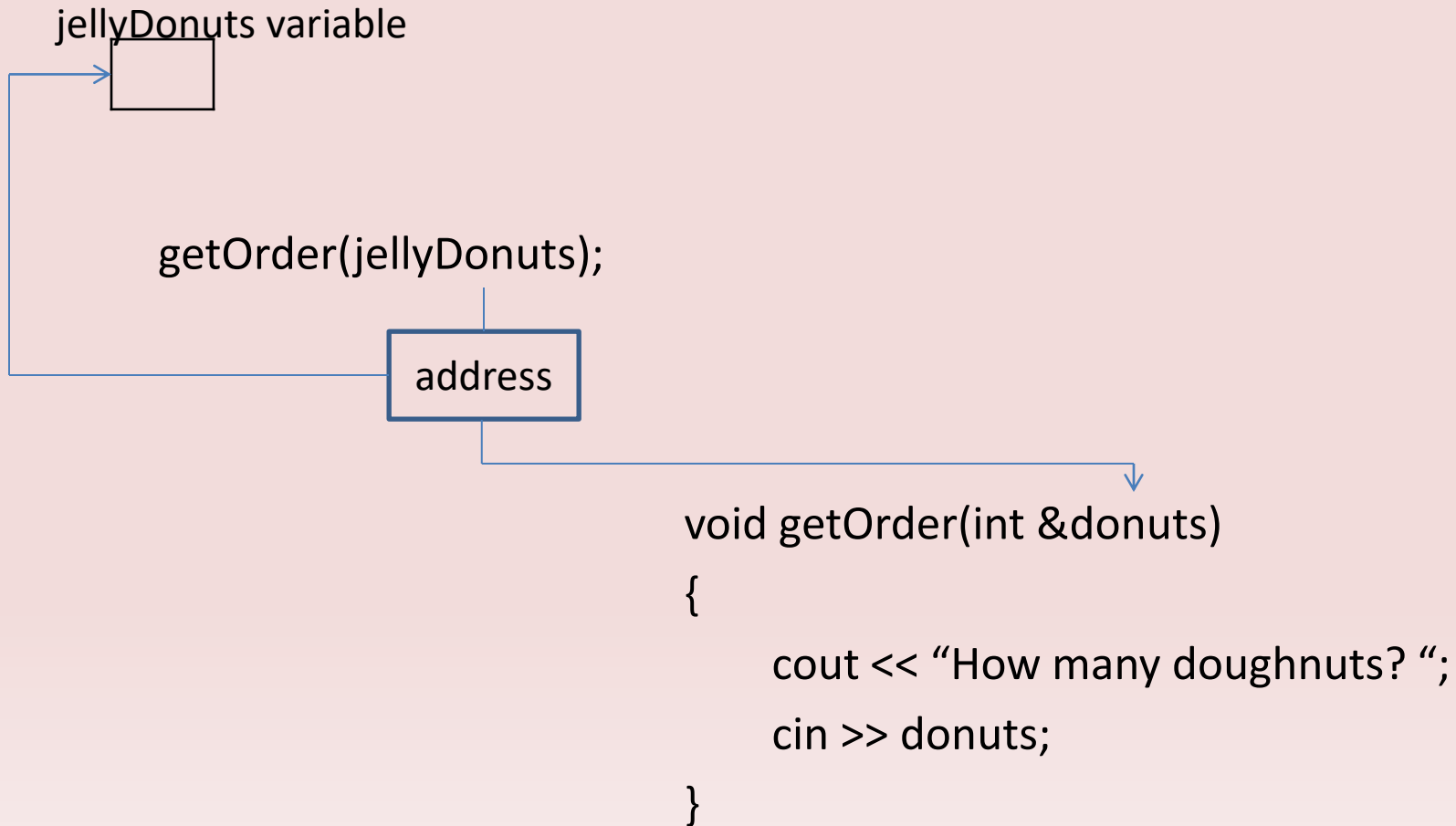


```
Ex:    int jellyDonuts;  
       getOrder(jellyDonuts);
```

## getOrder Defined

```
void getOrder(int &donuts)  
{  
    cout << "How many doughnuts do you want? ";  
    cin >> donuts;  
}
```

In this function, the donuts parameter is a reference variable, and receives the address of the jellyDonuts variable. Anything that is done to the donuts parameter is actually done to the jellyDonuts variable.



When the user enters a value, the cin statement uses the donuts reference variable to indirectly store the value in the jellyDonuts variable. The connection between the two variables are automatically established, so there is no need to worry about finding the memory address of the jellyDonuts variable.

In C++, pointer variables are yet another mechanism for using memory addresses to work with pieces of data. Pointer variables are similar to reference variables, but pointer variables operate at a lower level. This means that C++ does not automatically do as much work for you with pointer variables as it does with reference variables. In order to make a pointer variable reference another item in memory, you have to write code that fetches the memory address of that item and assign the address to the pointer variable. Also, when you use a pointer variable to store a value in the memory location that the pointer references, your code has to specify that the value should be stored in the location referenced by the pointer variable, and not in the pointer variable itself.

As you can see, reference variables are easier to work with, but pointers are useful, and even necessary for operations such as

- Dynamic memory allocation
- Algorithms that manipulate arrays and C-strings

# Creating and Using Pointer Variables

Sample pointer variable: `int *ptr; // reads ptr is a pointer of int`

- Asterisk in front of variable indicates that ptr is a pointer variable
- int data type indicates that ptr can hold the addresses of an integer value

Note: int does not mean that ptr is an integer variable, but rather ptr can hold the address of an integer variable. (**pointer can hold only addresses**)

Two different programming styles when defining pointers.

```
int *ptr; }  
int* ptr; } Both are correct.
```

# Storing Address of Pointers

// This program stores the address of a variable in a pointer.

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x = 25;                // int variable
```

```
    int *ptr;                 // Pointer variable, can point to an int
```

```
    ptr = &x;                 // Store the address of x in ptr
```

```
    cout << "The value in x is " << x << endl;
```

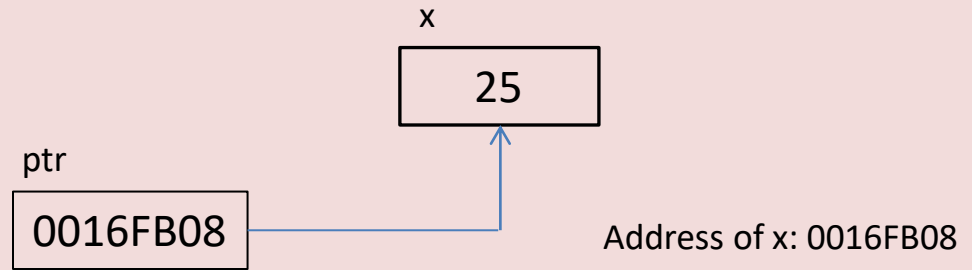
```
    cout << "The address of x is " << ptr << endl;
```

```
    return 0;
```

```
}
```

Sample output

```
The value in x is 25
The address of x is 0016FB08
```



x which is located at address 0016FB08, contains the number 25.

One real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. (i.e. ptr could be used to change the contents of the variable x using the indirection operator “\*”, the asterisk)

# Storing Address of Pointers

```
// This program demonstrates the use of the indirection operator.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int x = 25;           // int variable
```

```
    int *ptr;           // Pointer variable, can point to an int
```

Assigns the address of the x variable to the ptr variable.

```
    ptr = &x;           // Stores address of x in ptr
```

```
    // Use both x and ptr to display the value in x.
```

```
    cout << "Here is the value in x, printed twice:\n";
```

```
    cout << x << endl;           // Displays the contents of x
```

```
    cout << *ptr << endl;       // Displays the content of x
```

```
    // Assign 100 to the location pointed to by ptr. This
```

```
    // will actually assign 100 to x.
```

```
    *ptr = 100;
```

When applying the indirection operator (\*) to a pointer variable, you are working, not with the pointer variable itself, but with the item it points to.

```
    // Use both x and ptr to display the value of x.
```

```
    cout << "Once again, here is the value in x:\n";
```

```
    cout << x << endl;           // Displays the contents of x
```

```
    cout << *ptr << endl;       // Displays the contents of x
```

```
    return 0;
```

```
}
```

## Sample output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```

Ex: `cout << *ptr << endl;`

This statement references the value ptr is pointing to, which is that address of the value.

Ex: `cout << ptr << endl;`

Notice that this statement does not have the indirection operator (\*) in front of ptr, which now will output the memory location that is stored at ptr.

Ex: `*ptr = 100;`

Given that the indirection operator (\*) is used with ptr, 100 is assigned to the item ptr points to, which is the variable x.



# Pointer Referencing Different Variables

```
// This program demonstrates a pointer
// variable referencing different variables.
#include <iostream>
using namespace std;

int main()
{
    int x = 25, y = 50, z = 75;    // Three int variables
    int *ptr; // Pointer variable

    // Display the contents of x, y, and z.
    cout << "\n\n\tHere are the values of x, y,
            and z:\n";
    cout << "\t" << x << " " << y << " " << z << endl;

    // Use the pointer to manipulate x, y, and z.
    ptr = &x; // Store the address of x in ptr.
    *ptr += 100; // Add 100 to the value in x.
```

```
ptr = &y; // Store the address of y in ptr.
*ptr += 100; // Add 100 to the value in y.
```

```
ptr = &z; // Store the address of z in ptr.
*ptr += 100; // Add 100 to the value in z.
```

```
// Display the contents of x, y, and z.
cout << "\tOnce again, here are the values of
        x, y and z:\n";
cout << "\t" << x << " " << y << " " << z
        << endl << endl;
return 0;
}
```

## Sample output

```
Here are the values of x, y, and z:
25 50 75
Once again, here are the values of x, y and z:
125 150 175
```

Ex: `ptr = &x;`                      `ptr = &y;`                      `ptr = &z;`

These statements, when executed, assigns the address of x, y and z to the ptr variable.

Ex: `*ptr += 100;`                      `*ptr += 100;`                      `*ptr += 100;`

Notice in these statements the indirection operator (\*) is used. This means that you are not working with ptr, but rather the item that ptr points to.

Ex:        `cout << ptr;`        ← Will out the address of the item ptr is pointing to.

`cout << *ptr;`        ← Will output the value of the item ptr is pointing to.

| <b>Three different uses of the asterisk (*) so far.</b> |                                       |
|---|---------------------------------------|
| Multiplication Operator                                 | <code>distance = speed * time;</code> |
| Definition of Pointer Variable                          | <code>int *ptr;</code>                |
| Indirection Operator                                    | <code>*ptr = 100;</code>              |

# Arrays and Pointers - Relationship

```
// This program shows an array name being
// dereferenced with the * operator.
#include <iostream>
using namespace std;

int main()
{
    short numbers[] = {10, 20, 30, 40, 50};

    cout << "The first element of the array is ";
    cout << *numbers << endl;
    return 0;
}
```

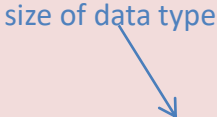
As we have learned in Chapter 7, an array name without brackets and a subscript, actually represents the starting address of the array.

Therefore, **an array name is actually a pointer**. This is done by using the array name preceded by the indirection operator.

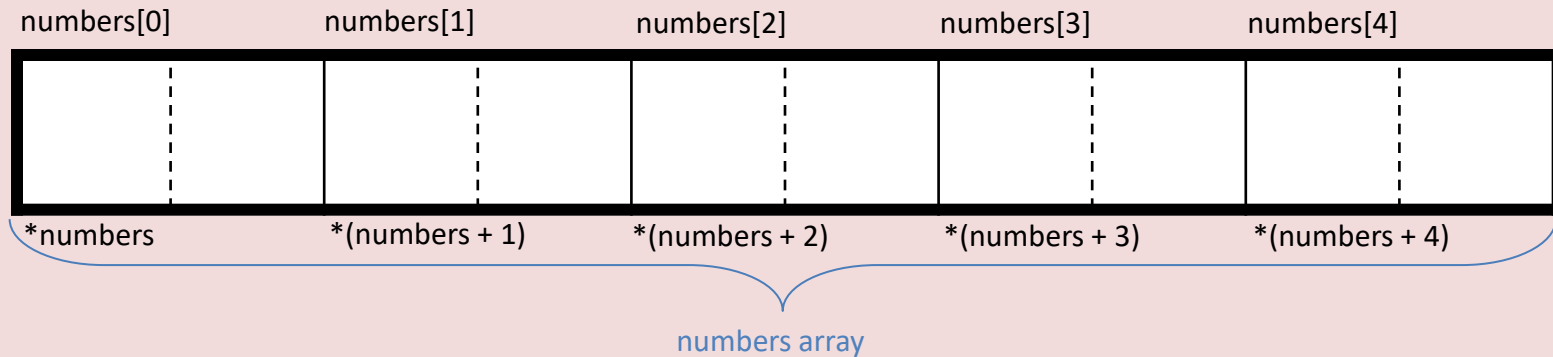
Because numbers in the above code works like a pointer to the starting address of the array, the first element is retrieved when numbers is dereferenced. Remember, array element are stored together in memory.

It is important to know that pointers do not work like regular variables when used in mathematical statements. In C++ when you add a value to a pointer, you are actually adding that value *times the size of the data type being referenced by the pointer*. Therefore, if you add one to numbers, you are actually adding  $1 * \text{sizeof}(\text{shorts})$  to numbers. If you add two to numbers, the result is  $\text{numbers} + 2 * \text{sizeof}(\text{short})$ , and so forth.

Ex:       $*(\text{numbers} + 1)$       is actually  $*(\text{numbers} + 1 * 2)$   
          $*(\text{numbers} + 2)$       is actually  $*(\text{numbers} + 2 * 2)$   
          $*(\text{numbers} + 3)$       is actually  $*(\text{numbers} + 3 * 2)$



This automatic conversion means that an element in an array can be retrieved by using its subscript or by adding its subscript to a pointer to the array. If the expression  $*\text{numbers}$ , which means  $*(\text{numbers} + 0)$ , retrieves the first element in the array, the  $*(\text{numbers} + 1)$  retrieves the second element.



The parentheses are critical when adding values to pointers. The `*` operator has precedence over the `+` operator, so the expression `*numbers + 1` is not equivalent to `*(numbers + 1)`.

When working with arrays, remember the following:

- **`array[index]`** is equivalent to **`*(array + index)`**
- C++ does not do bound checking, so when stepping through an array with pointers, it's possible to give the pointer an address outside of the array.

```
// This program shows the difference between
```

```
// *(array + n) and *array + n.
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
short numbers[] = {10, 20, 30, 40, 50};
```

```
cout << "x" << " *(numbers + x)" << "*numbers + x" << endl;
```

```
cout << "\t-----" << endl;
```

```
for (int x = 0; x < 5; x++)
```

```
cout << x << " *(numbers + x)" << "*numbers + x" << endl;
```

```
cout << endl << endl;
```

```
return 0;
```

```
}
```

The use of the & operator would be incorrect, because the name of the array is already an address.

| x | *(numbers + x) | *numbers + x |
|---|----------------|--------------|
| 0 | 10             | 10           |
| 1 | 20             | 11           |
| 2 | 30             | 12           |
| 3 | 40             | 13           |
| 4 | 50             | 14           |

```
// This program use the address operator to get
// the address of an individual element in an array.
```

```
#include <iostream>
using namespace std;
short *prt_num;
```

```
int main()
{
    short numbers[] = {10, 20, 30, 40, 50};
    cout << "\n\n\tx\t" << "*prt_num + x" << endl;
    cout << "\t-----" << endl;
    for (int x = 0; x < 5; x++)
    {
        prt_num = &numbers[x];
        cout << "\t" << x << "\t\t" << *prt_num << endl;
    }
    cout << endl << endl;
    return 0;
}
```

| x | *prt_num |
|---|----------|
| 0 | 10       |
| 1 | 20       |
| 2 | 30       |
| 3 | 40       |
| 4 | 50       |

The only difference between array names and pointer variables is that you cannot change the address an array name points to.

```
Ex: Given      double readings[20] , totals[20];  
              double *dprt;
```

Legal statements

```
dprt = readings;    // Make dprt point to readings starting address.  
dprt = totals;     // Make dprt point to totals starting address.
```

Illegal statements

```
readings = totals; // Illegal – cannot change readings or totals starting  
                  address.  
totals = dprt;     // Illegal – cannot change starting address of an array.
```

**NOTE: Array names are pointer constants. (They can only point to the array they represent.)**



# Pointer Arithmetic

The content of pointer variables can be changed with mathematical statements that perform addition and subtraction.

```
// This program you to manipulate and array by
// using pointers with addition and subtraction
#include <iostream>
using namespace std;
short *prt_num;

int main()
{
    short numbers[] = {10, 20, 30, 40, 50};
    prt_num = numbers;
    cout << "\n\n\tArray Content" << endl;
    cout << "\t-----" << endl;
    cout << "\tAscending" << " ";
    for (int x = 0; x < 5; x++)
    {
        cout << *prt_num << " ";
        prt_num++;
    }
}
```

```
        cout << "\n\tDescending" << " ";

    prt_num--;
    for (int x = 0; x < 5; x++)
    {
        cout << *prt_num << " ";
        prt_num--;
    }

    cout << endl << endl;
    return 0;
}
```

Sample output

```
Array Content
-----
Ascending 10 20 30 40 50
Descending 50 40 30 20 10
```

Ex:                    `short *prt_num;`

Because `prt_num` is a pointer of short data type, the increment operator adds the size of one integer to `prt_num`, so it points to the next element in the array. Likewise, the decrement operator subtracts the size of one integer from the pointer.

Not all arithmetic operations may be performed on pointers. You cannot multiply or divide a pointer.

| <b>Operations Allowable by Pointers</b>  |
|--|
| 1. The ++ and – operators may be used to increment or decrement a pointer variable.  |
| 2. An integer may be added to or subtracted from a pointer variable. This may be performed with the + and – operators, or the += and -= operators. |
| 3. A pointer may be subtracted from another pointer.   |

# Initializing Pointers

Pointers are designed to point to objects of a specific data type. When a pointer is initialized with an address, it must be the address of an object the pointer can point to.

Ex:     int     to     int  
          float  to     float

Ex:     int myValue; Is legal because pint and myValue is an integer  
          int \*pint = &myValue;

also,

          int ages[20]; Is legal because ages is an array of integer  
          int \*pint = ages;

but

          float myFloat;  
          int \*pint = &myFloat; illegal because myFloat is not an integer

Pointers can be defined in the same statement as other variables of the same data type.

```
Ex:    int myValue, *pint = &myValue;
```

The following statement defines an array, called readings, and a pointer, called marker, which is initialized with the address of the first element in the array.

```
Ex:    int readings[50], *marker = readings;
```

```
However,    int *pint = &myValue;  
            int myValue;
```

Is invalid because myValue has not been declared, but is used to initialize pint.

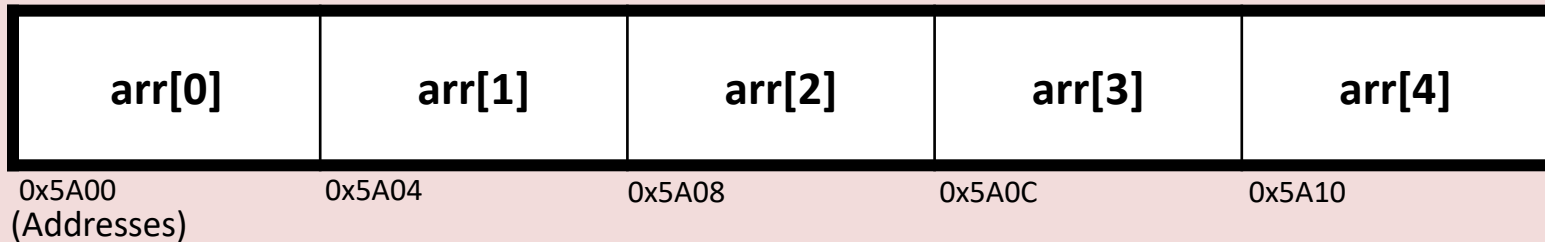
# Comparing Pointers

If one address comes before another address in memory, the first address is considered “less than” the second. C++ relational operators may be used to compare pointer values.

| <b>Relational Operators used to compare Pointers</b> |                          |
|--|--------------------------|
| <b>&gt;</b>  | Greater than             |
| <b>&lt;</b>  | Less than                |
| <b>==</b>  | Equal to                 |
| <b>!=</b>  | Not equal to             |
| <b>&gt;=</b>   | Greater than or equal to |
| <b>&lt;=</b>   | Less than or equal to    |

If one address comes before another address in memory, the first address is considered “less than” the second. C++ relational operators may be used to compare pointer values.

Array of five integers



Because the address grows larger for each subsequent element in the array, the following if statements are all true.

```
if (&arr[1] > &arr[0])
```

```
if (arr < &arr[4])
```

```
if (arr == &arr[0])
```

```
if (&arr[2] != &arr[3])
```

Compares the `ptr1`  
and `ptr2` addresses

Compares the values `ptr1`  
and `ptr2` is pointing to.

Comparing two pointers is not the same as comparing the values the two pointers point to. Ex: `if(ptr1 < ptr2)`      `if (*ptr1 < *ptr2)`

```

// This program uses a pointer to display
// the contents of an integer array
#include <iostream>
using namespace std;

int main()
{
    int set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *nums = set;    // Make nums point to set

    // Display the numbers in the array.
    cout << "The numbers in set are:\n";
    cout << *nums << " ";    // Display first element
    while (nums < &set[7])
    {
        // Advance nums to point to the next element.
        nums++;
    }
}

```

Address of last element in array

First time through the loop nums equals the starting address of set

Nums has been set to next address in memory list

```

// Display the value pointed to by nums.
cout << *nums << " ";
}

// Display the numbers in reverse order.
cout << "\nThe numbers in set backward are:\n";
cout << *nums << " "; // Display first element
while (nums > set)
{
    // move backward to the previous element.
    nums--;
    // Display the value pointed to by nums.
    cout << *nums << " ";
}
return 0;
}

```

Starting address of the set array

Nums has been set back one, to next address in memory list

Nums in the previous loop has been incremented to the address of the last memory location

# Pointers as Function Parameters

A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.

You can pass an argument by reference by using pointer variables as the parameter. Reference variables are much easier to work with than pointer, because reference variables hide all the “mechanism” of dereferencing and indirection. This is helpful when dealing with strings and the C++ library has many functions that use pointers as parameters.



# Pointers as Function Parameters

## Function prototype with a pointer parameter

```
void getNumber(int *);  
void doubleValue(int *);
```

Shows that the function will pass a pointer argument of integer

## Function call with a pointer parameter

```
getNumber(&number);  
doubleValue(&number);
```

Calls the function getNumber passing address of number as argument

Calls the function doubleValue passing address of number as argument

## Function definition with a pointer parameter

```
void doubleValue(int *val)  
{  
    *val *= 2;  
}
```

This function doubles the variable pointed to by val.

```
// This program uses two function that
// accept addresses of variables as
// as arguments.
#include <iostream>
using namespace std;

// Function prototypes
void getNumber(int *);
void doubleValue(int *);

int main()
{
    int number;

    // Call getNumber and pass the address of
    // number.
    getNumber(&number); ← The address of number is
                          passed as the argument.

    // Call doubleValue and pass the address of
    // number.
    doubleValue(&number); ← The address of number is
                           passed as the argument.
```

```
// Display the value in number.
cout << "That value doubled is " << number <<
    endl;
return 0;
}

// Definition of getNumber.
void getNumber(int *input)
{
    cout << "Enter an integer number: ";
    cin >> *input; ← The value entered is
                    stored in number.
}

// Definition of doubleValue.
void doubleValue(int *val)
{
    *val *=2; ← Doubles the value
              stored in number.
}
```

Pointer variables can also be used to accept array address as arguments. Either subscript or pointer notation may then be used to work with the contents of the array.

```
// This program demonstrates that a pointer
// may be used as a parameter to accept the
// address of an array.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void getSales(double *, int);
double totalSales(double *, int);

int main()
{
    const int QTRS = 4;
    double sales[QTRS];

    // Get the sales data for all quarters.
    getSales(sales, QTRS);

    // Set the numeric output formatting.
    cout << fixed << showpoint << setprecision(2);

    // Display the total sales for the year.
    cout << "The total sales for the year are $";
    cout << totalSales(sales, QTRS) << endl;
    return 0;
}
```

```
// Definition of getSales
void getSales(double *arr, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << "Enter the sales figure for quarter ";
        cout << (count + 1) << ": ";
        cin >> arr[count];
    }
}

// Definition of totalSales
double totalSales(double *arr, int size)
{
    double sum = 0.0;

    for (int count = 0; count < size; count++)
    {
        sum += *arr;
        arr++;
    }
    return sum;
}
```

arr defined as a pointer parameter, but subscript notation is used in cin statement.

arr is used with the indirection operator

arr incremented to point to the next element