Pointers

Introduction to Pointers

When we declare a variable some memory is allocated for it. The memory location can be referenced through the identifier “i”. Thus, we have two properties for any variable : its address and its data value. The address of the variable can be accessed through the referencing operator “&”. “&i” gives the memory location where the data value for “i” is stored.

Pointers

What is pointer?

- A variable that can hold an ***Address***
- A pointer variable is one that stores an address. We can declare pointers as follows int* p; .This means that p stores the address of a variable of type int.

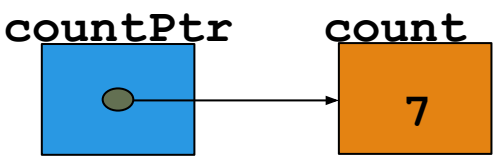
Related operators:

- ***** : can be used in two ways
  - In declaration : read as ***pointer***, e.g., int *p;
  - In accessing : read as ***content of***, e.g., x = *p;
- **&** : returns LValue (address) of a variable
  - Read as ***address of***

Declaring Pointer Variables

- Syntax:

```
dataType *identifier;
```



- Examples:

```
int *p;
char *ch;
```

- These statements are equivalent:

```
int  *p;
int*  p;
int * p;
```

Pointer Variable Declarations and Initialization

Pointer declarations

- **`*`** used with pointer variables

```
int *myPtr;
```

- Declares a pointer to an **`int`** (pointer of type **`int *`**)
- Multiple pointers, multiple **`*`**

```
int *myPtr1, *myPtr2;
```

- Can declare pointers to any data type
- Initialize pointers to **`0`**, **`NULL`**, or an address
  - **`0`** or **`NULL`** - points to nothing (**`NULL`** preferred)

Assignment of Pointer Variables

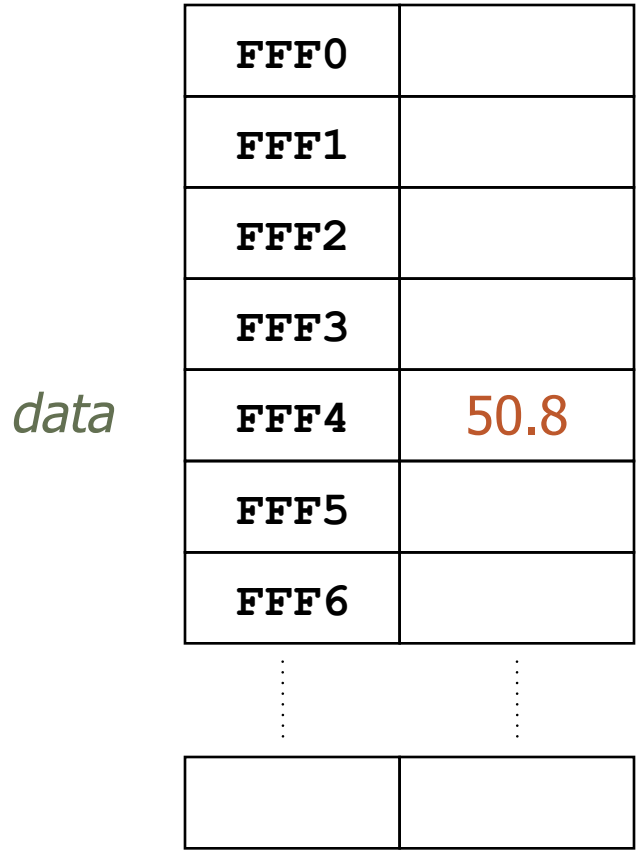
- A pointer variable has to be assigned a valid memory address before it can be used in the program
- Example:

```
float data = 50.8;
float *ptr;
ptr = &data;
```

  - This will assign the address of the memory location allocated for the floating point variable **`data`** to the pointer variable **`ptr`**. This is OK, since the variable **`data`** has already been allocated some memory space having a valid address

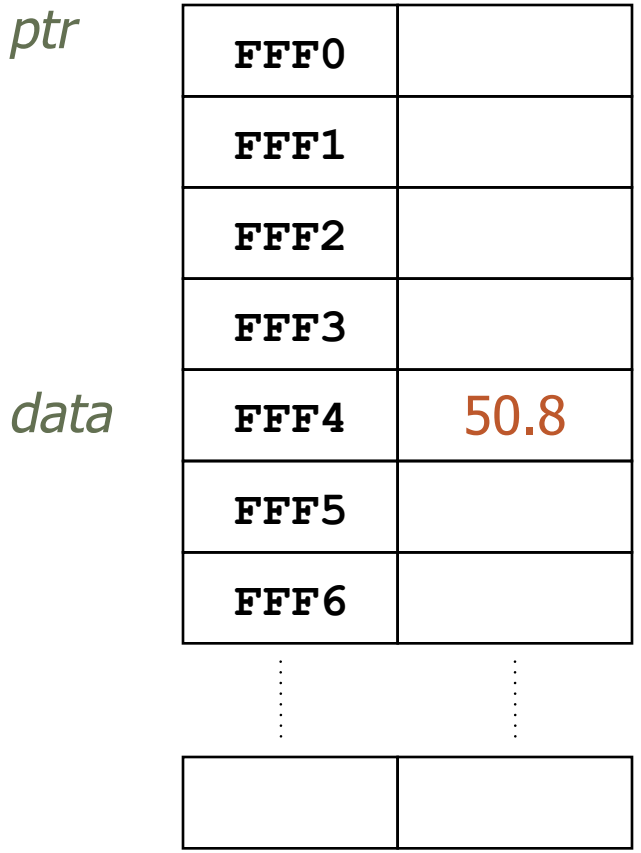
Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



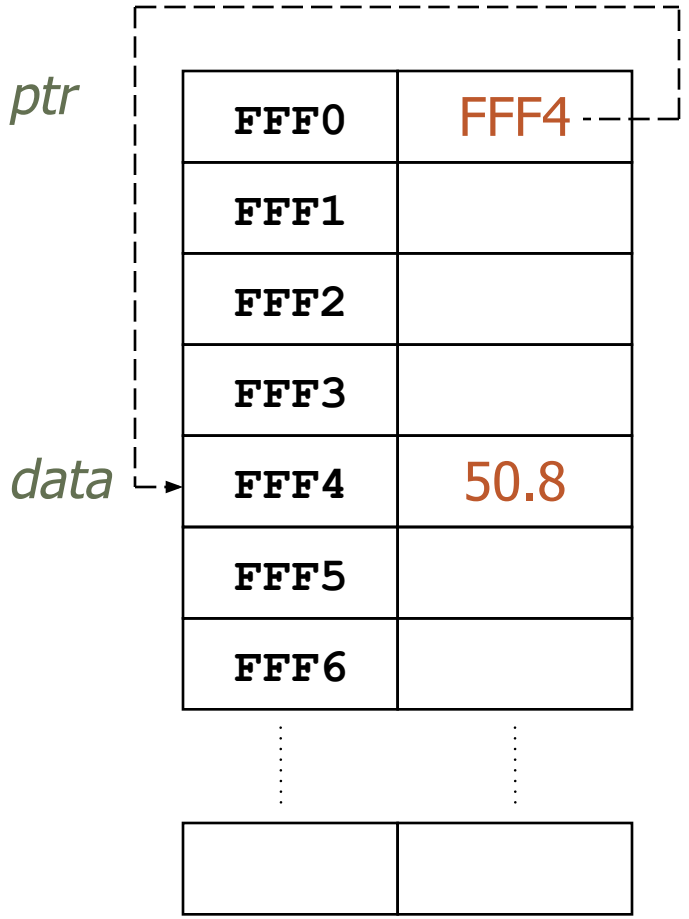
Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables (Cont ..)

Don’t try to assign a specific integer value to a pointer variable since it can be disastrous

```
float *ptr;
ptr = 120;
```

```
int data = 50;
float *ptr;
ptr = &data;
```

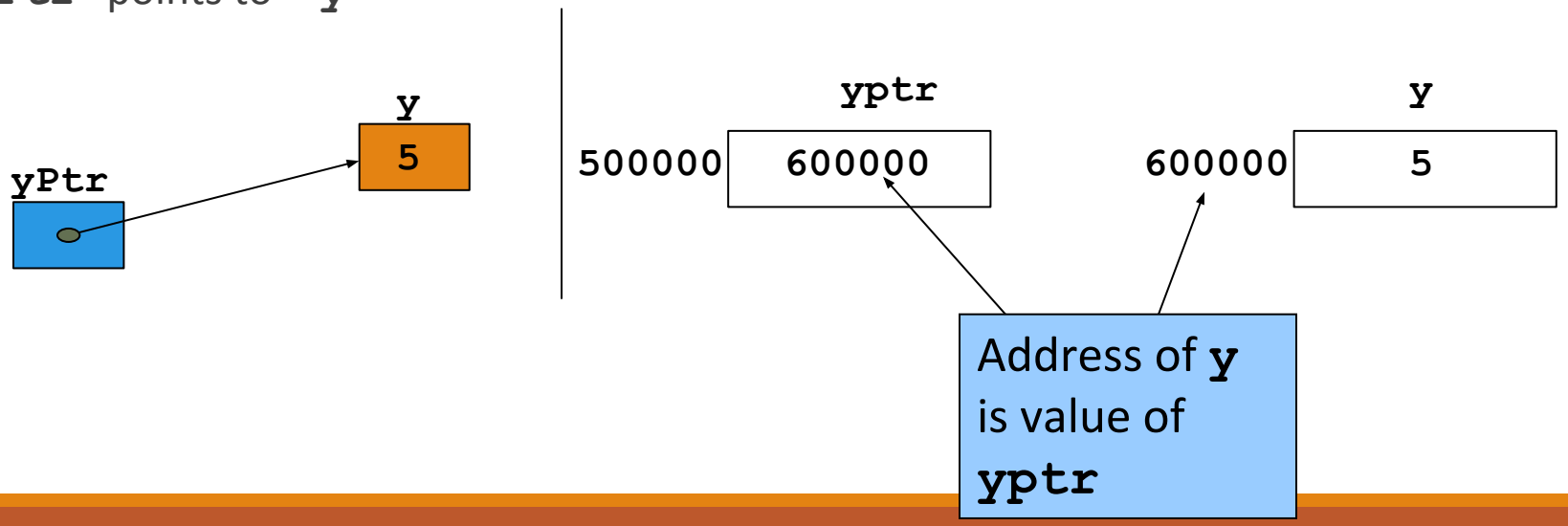
Pointer Operators

& (address operator)

- Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;  //yPtr gets address of y
```

- **yPtr** "points to" **y**



Dereferencing Operator (*)

When used as a unary operator, * is the dereferencing operator or indirection operator

- Refers to object to which its operand points

Example:

```
int x = 25;
int *p;
p = &x;    //store the address of x in p
```

- To print the value of x, using p:

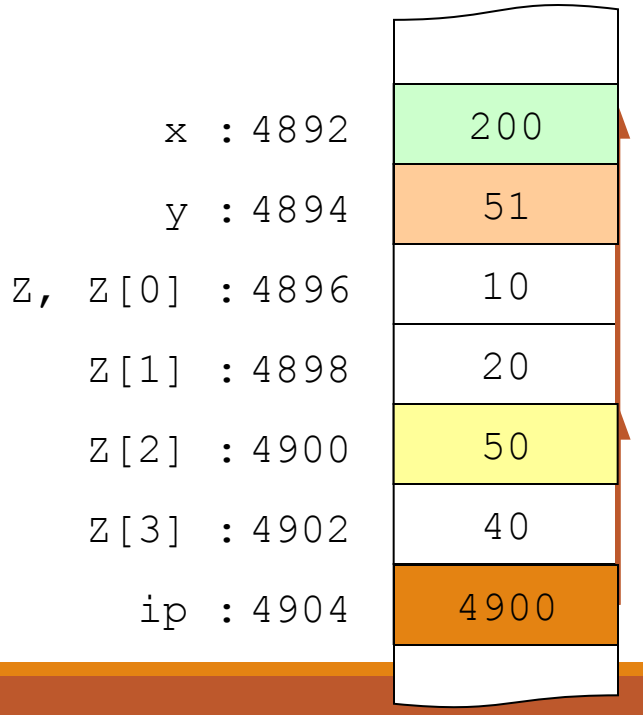
```
cout << *p << endl;
```

- To store a value in x, using p: `*p = 55;`

Pointer Examples

```
int x = 70, y = 80, z[4] = {10, 20, 30, 40 };
int *ip;    // int pointer ip

ip = &x;    // ip is assigned to address of x
*ip = 200; // content of ip is assigned to 200
y = *ip;    // y is assigned to content of ip
ip = &z[2];
*ip = *ip + 20;// same as *ip += 20;
y = *ip+1;
```



Pointer Examples

```
int x = 100;

int *p1 = &x;

cout << "&x == " << &x << "    x == " << x << endl << "p1 == " << p1 << "  *p1 == " <<
*p1 << endl;

*p1 += 20;

cout << "&x == " << &x << "    x == " << x << endl << "p1 == " << p1 << "  *p1 == " <<
*p1 << endl;
```

Output

```
&x == 0x7fff7e60f41c    x == 100

p1 == 0x7fff7e60f41c  *p1 == 100

&x == 0x7fff7e60f41c    x == 120

p1 == 0x7fff7e60f41c  *p1 == 120
```

const and Pointers

```
const int w=20;

int x, y;
```

const qualifier can be used with pointers in three ways:

Prohibit a pointer from changing its **content**

```
const int *p1 = &w;
 // p1 can point to (non-)const int
*p1 = 200;
// invalid : content can't change
p1 = &x;
// OK. p1 points to x (non-const)
```

const and Pointers

Prohibit a pointer from changing its **reference**:

```
int * const p3 = &x; // must be initialized at creation
p3 = &y;// invalid: reference can't change
*p3 = 123; // OK. changes value of x
```

Prohibit a pointer from changing **both** reference & content:

```
const int * const p4 = &w;
// must be initialized at creation & point to const int
```

Relationship Between Arrays and Pointers

array names can be used as pointers, and vice-versa.

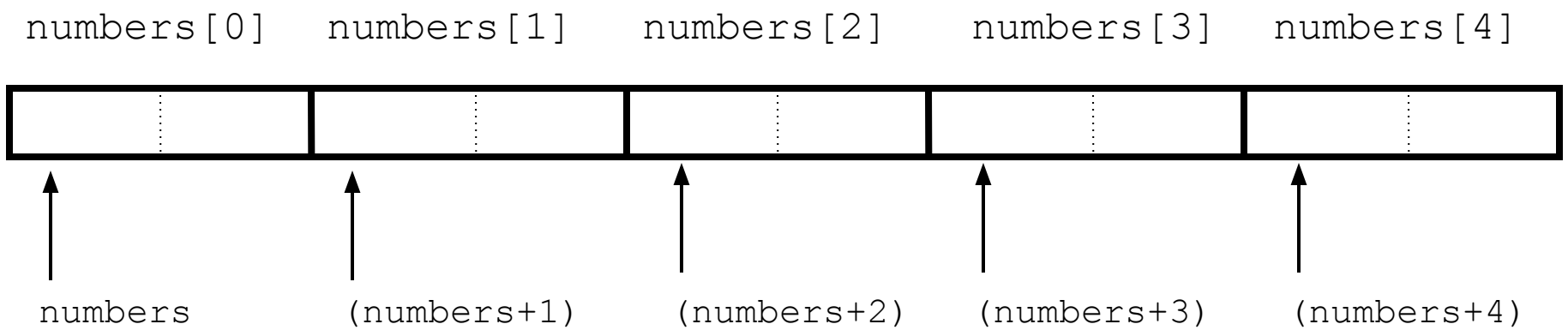
Program

```
// This program shows an array name being dereferenced
// with the * operator.
#include <iostream>
void main(void)
{
 short numbers[] = {10, 20, 30, 40, 50};
 cout << "The first element of the array is ";
 cout << *numbers << endl;
}
```

Output:

The first element in the array is 10

Figure


numbers[0]
numbers[1]
numbers[2]
numbers[3]
numbers[4]
numbers
(numbers+1)
(numbers+2)
(numbers+3)
(numbers+4)

Program

```
// This program processes the contents of an array. Pointer
// notation is used.
#include <iostream.h>
void main(void)
{
 int numbers[5];
 cout << "Enter five numbers: ";
 for (int count = 0; count < 5; count++)
    cin >> *(numbers + count);
 cout << "Here are the numbers you entered:\n";
 for (int count = 0; count < 5; count++)
    cout << *(numbers + count)<< " ";
 cout << endl;
}
```

Dynamic Variable

Variables that are created during program execution are called dynamic variables.

With the help of pointers, C++ creates dynamic variables.

C++ provides two operators, new and delete, to create and destroy dynamic variables, respectively.

When a program requires a new variable, the operator new is used.

When a program no longer needs a dynamic variable, the operator delete is used.

In C++, new and delete are reserved words

Operator new

The operator new has two forms: one to allocate a single variable and another to allocate an array of variables. The syntax to use the operator new is:

```
new dataType;              //to allocate a single variable
new dataType[intExp];      //to allocate an array of variables
```

in which **intExp** is any expression evaluating to a positive integer.
The operator new allocates memory (a variable) of the designated type and returns a pointer to it—that is, the address of this allocated memory. Moreover, the allocated memory is uninitialized.
p = new int;
This statement creates a variable during program execution somewhere in memory and stores the address of the allocated memory in p. The allocated memory is accessed via pointer dereferencing namely, *p.
Similarly, the statement:
q = new char[16];
creates an array of 16 components of type char and stores the base address of the array in q.

Because a dynamic variable is unnamed, it cannot be accessed directly. It is accessed indirectly by the pointer returned by new. The following statements illustrate this concept:

```
int *p;              //p is a pointer of type int
char *name;          //name is a pointer of type char
string *str;         //str is a pointer of type string

p = new int;         //allocates memory of type int
                     //and stores the address of the
                     //allocated memory in p
*p = 28;             //stores 28 in the allocated memory

name = new char[5];      //allocates memory for an array of
                         //five components of type char and
                         //stores the base address of the array
                         //in name
strcpy(name, "John");    //stores John in name

str = new string; //allocates memory of type string
                  //and stores the address of the
                  //allocated memory in str
*str = "Sunny Day";      //stores the string "Sunny Day" in
                         //the memory pointed to by str
```

Operator delete

Suppose you have the following declaration:

int *p;

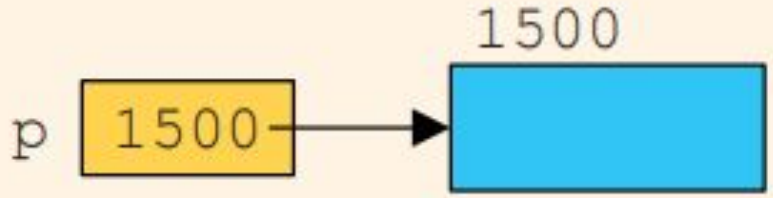
This statement declares p to be a pointer variable of type int. Next, consider the following statements:

p = new int; //Line 1

*p = 54; //Line 2

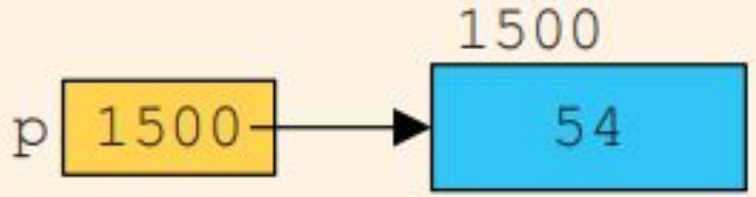
p = new int; //Line 3

*p = 73; //Line 4

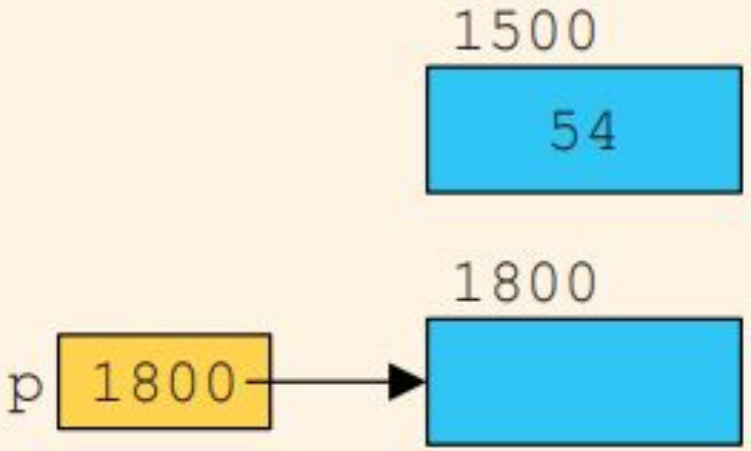


(*a*) `p` after the execution of

```
p = new int;
```



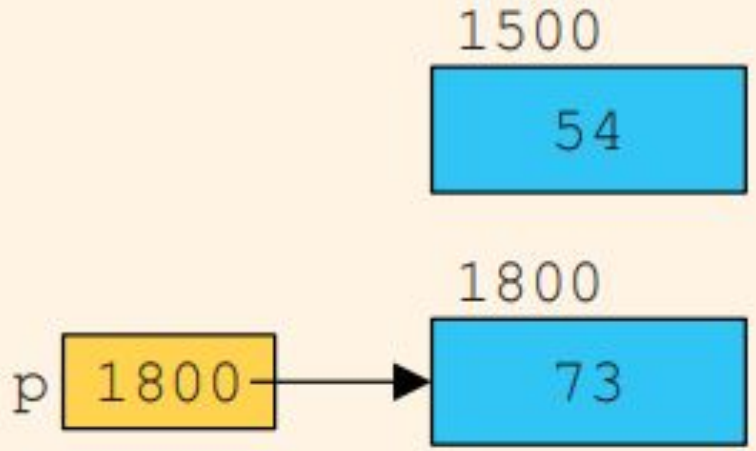
(*b*) `p` and `*p` after the execution of `*p = 54;`



(*c*) `p` after the execution of

```
p = new int;
```

main



(*d*) `p` and `*p` after the execution of `*p = 73;`

Memory leak

Now the obvious question is what happened to the memory space 1500 that p was pointing to after execution of the statement in Line 1. After execution of the statement in Line 3, p points to the new memory space at location 1800. The previous memory space at location 1500 is now inaccessible. In addition, the memory space 1500 remains as marked allocated. In other words, it cannot be reallocated. This is called memory leak. That is, there is an unused memory space that cannot be allocated

How to avoid memory leak

When a dynamic variable is no longer needed, it can be destroyed; that is, its memory can be deallocated.

operator delete is used to destroy dynamic variables. The syntax to use the operator delete has two forms

```
delete pointerVariable;     //to deallocate a single
                            //dynamic variable
delete [] pointerVariable; //to deallocate a dynamically
                            //created array
```

delete p;

delete [] name;

delete str

Suppose p and name are pointer variables, as declared previously. Notice that an expression such as:

```
delete p;
```

or:

```
delete [] name;
```

only marks the memory spaces that these pointer variables point to as deallocated. Depending on a particular system, after these statements execute, these pointer variables may still contain the addresses of the deallocated memory spaces. In this case, we say that these pointers are **dangling**. Therefore, if later you access the memory spaces via these pointers without properly initializing them, depending on a particular system, either the program will access a wrong memory space, which may result in corrupting data, or the program will terminate with an error message. One way to avoid this pitfall is to set these pointers to NULL after the delete operation. Also note that for the operator delete to work properly, the pointer must point to a valid memory space.

Operations on Pointer Variables

The operations that are allowed on pointer variables are the assignment and relational operations and some limited arithmetic operations.

The value of one pointer variable can be assigned to another pointer variable of the same type.

Two pointer variables of the same type can be compared for equality, and so on.

Integer values can be added and subtracted from a pointer variable.

The value of one pointer variable can be subtracted from another pointer variable

Operations on Pointer Variables

For example, suppose that we have the following statements:

int *p, *q;

The statement: **p = q;**

copies the value of q into p. After this statement executes, both p and q point to the same memory location. Any changes made to *p automatically change the value of *q, and vice versa

The expression:

p == q

evaluates to true if p and q have the same value—that is, if they point to the same memory location.

Similarly, the expression:

p != q evaluates to true if p and q point to different memory locations

Arithmetic operations on Pointers

int *p;

double *q;

char *chPtr;

the size of the memory allocated for an int variable is 4 bytes, a double variable is 8 bytes, and a char variable is 1 byte’

The statement: **p++; or p = p + 1;** increments the value of p by 4 bytes because p is a pointer of type int.

q++;

chPtr++;

increment the value of q by 8 bytes and the value of chPtr by 1 byte, respectively.

Arithmetic operations on Pointers

The increment operator increments the value of a pointer variable by the size of the

memory to which it is pointing. Similarly, the decrement operator decrements the value

of a pointer variable by the size of the memory to which it is pointing.

Moreover, the statement:

p = p + 2;

increments the value of p by 8 bytes

Thus, when an integer is added to a pointer variable, the value of the pointer variable is incremented by the integer times the size of the memory that the pointer is pointing to. Similarly, when an integer is subtracted from a pointer variable, the value of the pointer variable is decremented by the integer times the size of the memory to which the pointer is pointing

Dynamic Arrays

It would be helpful if, during program execution, you could prompt the user to enter the size of the array and then create an array of the appropriate size. This approach is especially helpful if you cannot even guess the array size. In this section, you will learn how to create arrays during program execution and process such arrays.

An array created during the execution of a program is called a dynamic array. To create a dynamic array, we use the second form of the new operator.

The statement:

int *p;

declares p to be a pointer variable of type int. The statement:

p = new int[10];

allocates 10 contiguous memory locations, each of type int, and stores the address of the

first memory location into p.

Dynamic Arrays (Continue)

In other words, the operator new creates an array of 10 components of type int, it returns the base address of the array, and the assignment operator stores the base address of the array into p

***p = 25;**

stores 25 into the first memory location, and the statements:

p++; //p points to the next array component

***p = 35** store 35 into the second memory location

C++ allows us to use array notation to access these memory locations.

For example, the statements:

p[0] = 25; p[1] = 35;

store 25 and 35 into the first and second array components, respectively.

In int list[5];

An array name is a constant pointer

If p is a pointer variable of type int, then the statement:

p = list;

copies the value of list, which is 1000, the base address of the array, into **p**. We are allowed to perform increment and decrement operations on **p** now

Array to Function as parameters

```
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;

    for(i = 0;i< size;i++)
    {
        sum+= A[i]; // A[i] is *(A+i)
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(A,size); // A can be used for &A[0]
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

Swapping Using pointers

CLASS ACTIVITY

Functions Returning Pointers

```
int* fun()
{
    int A = 10;
    return (&A);
}
int main()
{
    // Declare a pointer
    int* p;
    // Function call
    p = fun();
    cout<<p;
    cout<<*p;
    return 0;
```

```
#include <iostream>
using namespace std;
int* fun()
{
	static int A = 10;
	A++;
	return (&A);
}
int main()
{
	int* p;
	p = fun();
	cout << p<<endl;
	cout << *p<<endl;
	return 0;
}
```

Functions Returning Pointers

```
#include <iostream>
using namespace std;
int* biger(int&, int&);
void main()
{
   int num1, num2, * c;
   cout << "Enter two integers\n";
   cin >> num1 >> num2;
   c = bigger(num1, num2);
   cout << "The bigger value = " << *c;
}
```

```
int* biger(int& x, int& y)
{
   if (x > y)
   {
      return(&x);
   }
   else
   {
      return(&y);
   }
}
```