Programming in C/C++

Lecture: Chapter 8 Pointers

Introduction to Pointers

(Chapter 9 in old book)

Kristina Shroyer

Addresses and Pointers

- High-level languages use memory addresses throughout executable programs
  - Keeps track of where data and instructions are physically located inside of computer

- **Advantage of C++:** The programmer is provided access to addresses of program variables
  - This access enables a programmer to enter directly into the computer's inner workings and manipulate the computer's basic storage structure
  - This capability is typically not provided in other high-level languages

- This lecture will present the basics of declaring variables to store addresses. Such variables are referred to as pointer variables or simply pointers
  - ***Pointer (Pointer Variable):*** a variable that stores the address of another variable

Addresses and Pointers

- Every variable has three major parts associated with it:

1. **Its Data type:** declared in a declaration statement

2. **Its Actual Value:** Stored in a variable by:
   - Initialization when variable is declared
   - Assignment
   - Input

3. **Its Address (where the variable is stored in memory):** For most applications, variable name is sufficient to locate variable's contents
   - **Translation of variable's name to a storage location (memory address) is done by the computer each time variable is referenced**

Addresses and Pointers

- **Concept:** The address operator `&` returns the memory address of a variable

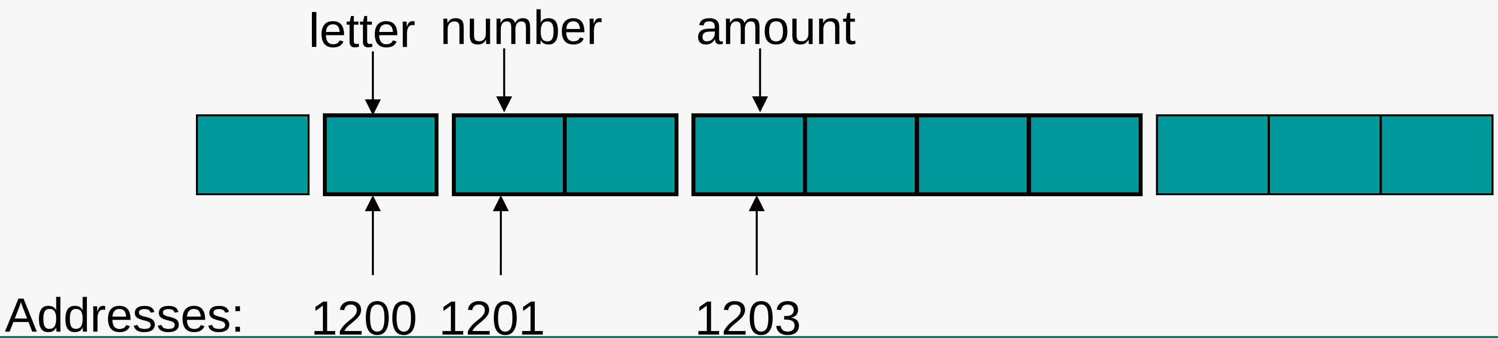
- When a variable is declared it is allocated a section of memory large enough to hold a value of that variable's data type.
  - On a PC it's common for 1 byte to be allocated for `char`, 4 bytes for `int` and `float`, and 8 bytes for `double`
  - **Each byte of memory has a unique address**
    - A ***variable's address*** is the ***first byte allocated to that variable***

Addresses and Pointers

- **Illustration of concept of a variable’s memory address:** Suppose the following variables are declared in a program:

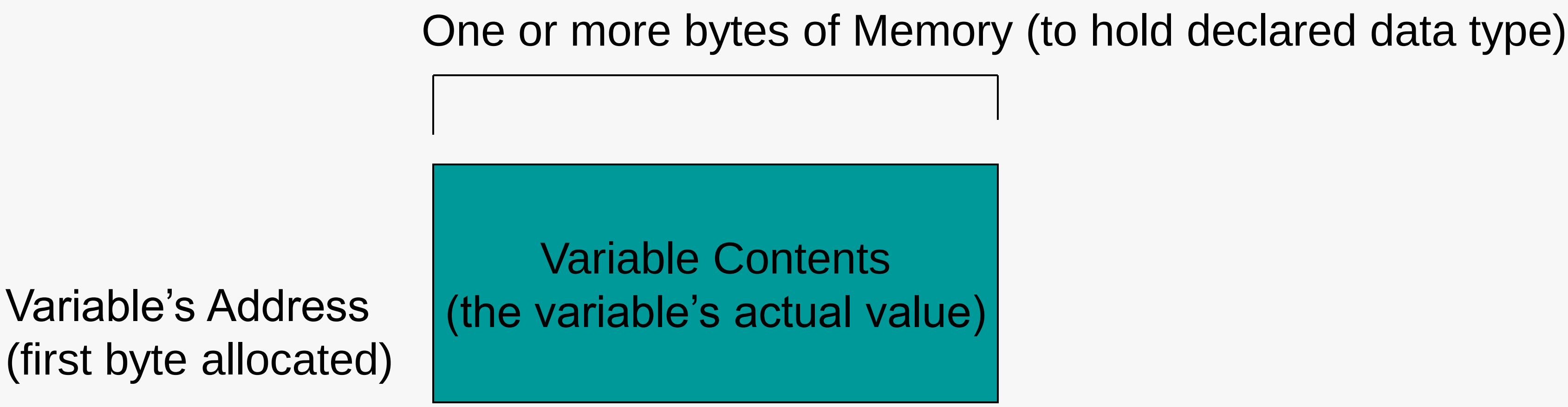
```
char letter;    //1 byte
short number;   //2 bytes
float amount;   //4 bytes
```

- The illustration below shows how these variables might be arranged in memory and shows their addresses
  - **Note that the addresses shown in the figure below are just arbitrary values and are used only for illustration purposes**
    - **In reality, Addresses are NOT stored in decimal notation but instead in hexidecimal (base 16)**
  - In the illustration each box represents a byte in memory
  - In the illustration the address of letter would be 1200, the address of number would be 1201 and the address of amount would be 1203
    - Remember: A **variable’s address** is the **first byte allocated to that variable**



Addresses and Pointers

- This figure illustrates the relationship between the three parts of a variable: type, contents, and location



Addresses and Pointers

- ***Example #1:*** Two of the three parts of a variable. Programmers are usually only concerned with the value assigned to a variable and pay little attention to where it is stored (its address)

```
/*Example of two of the three items associated with a
variable*/
#include <iostream>
using namespace std;

int main()
{
   int num;
   num = 22;
   cout << "The value stored in num is " << num << endl;
   cout << sizeof(num) << " bytes are used to store this variable"
   << endl;

   system("PAUSE");
   return 0;
}
```

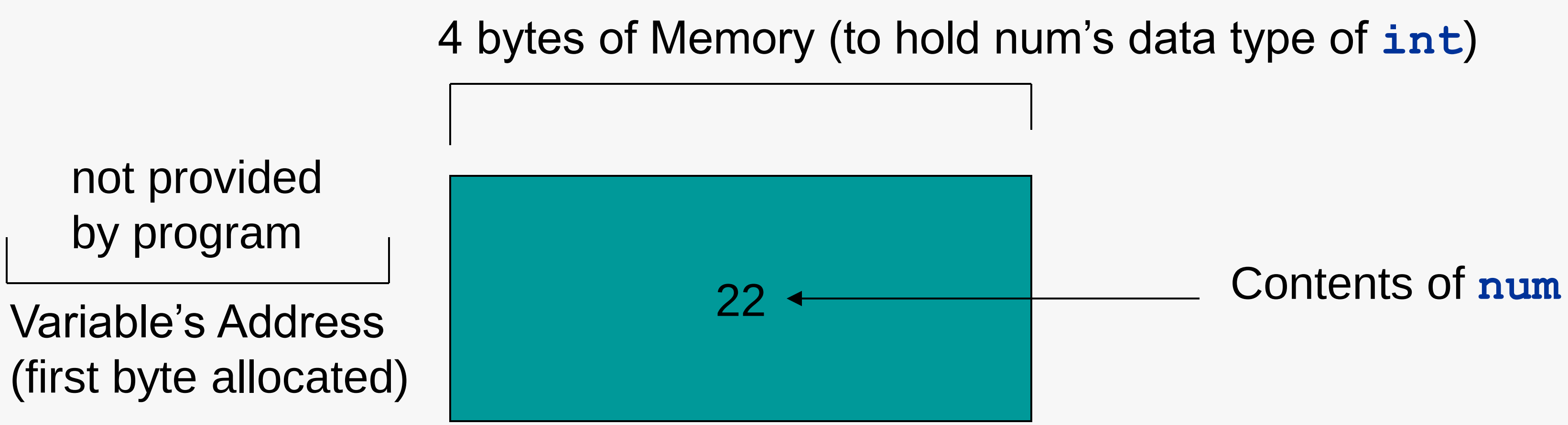
Addresses and Pointers

- An illustration of the information provided by the previous program (Example #1)
- Program output (Example #1):

    The value stored in num is 22

    4 bytes are used to store this variable

    Press any key to continue . . .



Addresses and Pointers

- What if we wanted the program to also obtain the address corresponding the variable num?
- **Address operator `&`:** determines the address of a variable
  - `&` means "address of"
  - When the address operator `&` is placed in front of a variable name it returns the address of that variable
  - When placed in front of variable `&num`, `&` is translated as **"the address of `num`"**
- Program 9.2 (shown on the next slide) uses the address operator to display the address of variable `num`

Addresses and Pointers

- ***Example #2: (AddressOperator.cpp)*** Using the address operator to display the address of a variable (remember an address is the first byte the variable is stored at in memory)

```
/*Example of the three items associated with a variable*/
#include <iostream>
using namespace std;

int main()
{
   int num;
   num = 22;
   cout << "The value stored in num is " << num << endl;
   cout << "The address of num is " << &num << endl;
   cout << sizeof(num) << " bytes are used to store this
   variable" << endl;

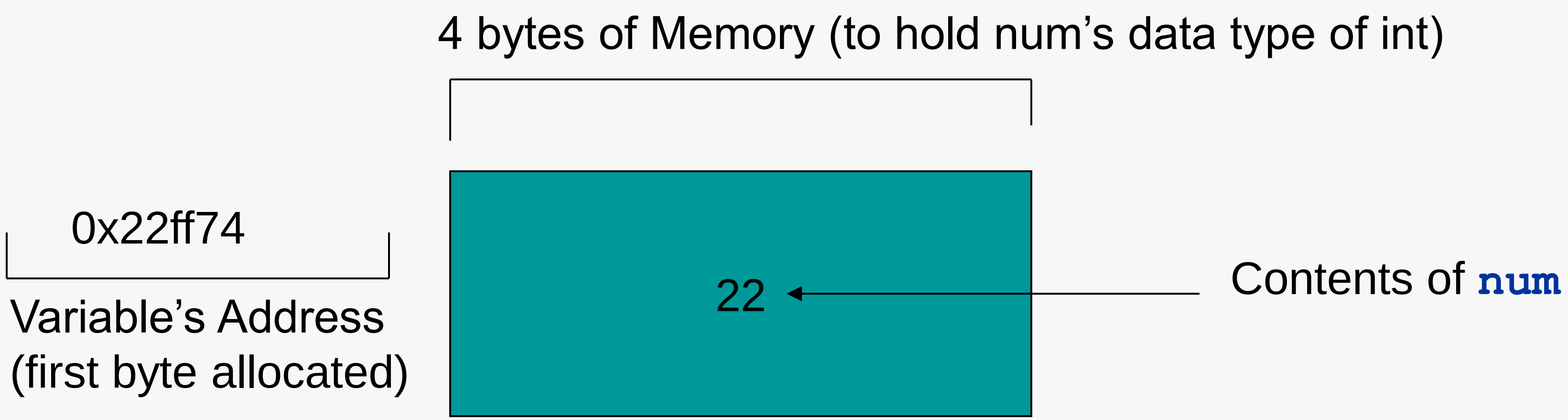
   system("PAUSE");
   return 0;
}
```

Addresses and Pointers

- An illustration of the information provided by the previous program (Example #2)
- Program output (Example #2):

```
The value stored in num is 22
The address of num is 0x22ff74
4 bytes are used to store this variable
Press any key to continue . . .
```

- ***The address output by this program depends on the computer used to run this program***
  - However every time this program is executed it displays the address of the first memory byte used to store the variable num
  - The address of the variable is displayed in hexadecimal (this is the way addresses are normally shown in C++)



Pointer Variables

- **Concept:** ***Pointer variables***, which are often just called ***pointers*** are variables designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables

- Many operations are best performed with pointers and some tasks aren't possible without them

- Some examples of tasks pointers are useful for:
  - Working directly with memory locations that regular variables don't give you access to
  - Working with strings and arrays
  - Creating new variables in memory while the program is running (this is called dynamically allocating memory for variables)
  - Creating arbitrarily sized lists of values in memory
  - Passing objects to functions without wasting memory copying them – memory usage often becomes very important in C++ programs
    - Remember what we discussed in the structs lecture with pass by reference, pass by value and pass by const reference

Pointer Variables

- Pointers are special variables in C++ designed for working with memory addresses.
  - Just like `int` variables are designed to work with integers, **pointer variables area designed to hold and work with addresses**
  - The definition of a pointer variable (much like any other variable definition):
  - ```
    //these two declaration styles are both correct
                 int *ptr;
                 int* ptr;
    ```
    - The asterisk (star) in front of the variable name indicates that `ptr` is a pointer variable
    - The `int` data type indicates that `ptr` can be used to hold the ***address of an integer type variable***
      - So in the pointer variable declaration you MUST declare what type of variable the pointer will point to…what type of address will it hold?
      - ***The pointer can ONLY hold an address of the data type specified***
    - This declaration statement would read **"ptr is a pointer to an int"**
  - NOTE: The word `int` does NOT mean that pointer is an integer variable. It means that `ptr` can hold the address of an integer variable. **Pointers can only hold one thing: addresses.**
    ```

Pointer Variables

- ***Example #3: (Pointer1.cpp)*** This program stores the address of a variable in a pointer

```
/*This program stores the address of a variable in a pointer and
   prints the address*/
#include <iostream>
using namespace std;

int main()
{
   int x = 25;
   int *ptr;   //ptr is a pointer to an int

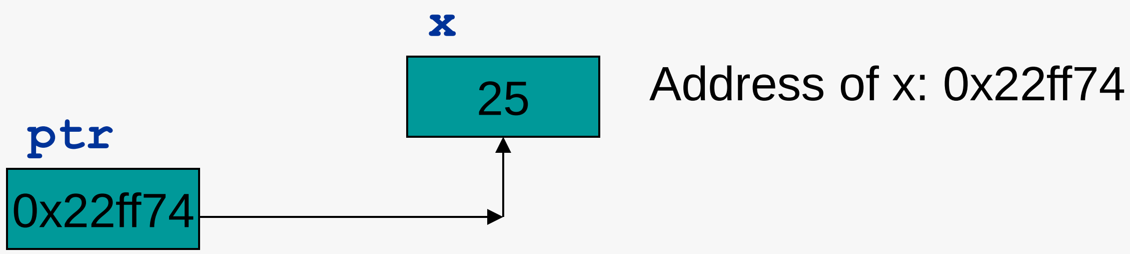
   ptr = &x;  //store the address of x in ptr

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

   system("PAUSE");
   return 0;
}
```

Pointer Variables

- **Example #3:** This program stores the address of a variable in a pointer
  - **Program Output:**
    - The value in x is 25
    - The value in ptr is 0x22ff74
    - Press any key to continue . . .
  - Two variables are defined in the program: **`x`** and **`ptr`**.
    - The variable **`x`** is initialized with 25
    - The variable **`ptr`** is assigned the address of **`x`** with the following statement
      - **`ptr = &x;`**
    - The figure below illustrates the relationship between **`ptr`** and **`x`**
    - The variable **`x`** is located at memory address 0x22ff74 while the pointer variable **`ptr`** contains the address 0x22ff74.
      - In essence **`ptr`** "points" to the variable **`x`**



Pointer Variables

- **Pointers can allow you to *indirectly access and modify the variable being pointed to***
  - In Example 3 the `ptr` variable could be used to indirectly modify the contents of the variable x
    - This is done with the ***indirection operator***, which is an asterisk (`*`)
    - The `*` symbol when followed by a pointer variable means **“the variable whose address is stored in”**
      - Thus the `*ptr` would mean “the variable whose address is stored in” `ptr`
    - **When you use the *indirection operator* it is called *dereferencing a pointer***
      - When you dereference a pointer you are actually working with the value the pointer is pointing to
      - Best seen with an example

Pointer Variables

- ***Example #4: (Pointer2_IndirOp.cpp)*** Dereferencing a pointer

```
/*This program demonstrates the use of the indirection operator on a
   pointer(dereferencing a pointer)*/
#include <iostream>
using namespace std;

int main()
{
   int x = 25;
   int *ptr;   //ptr is a pointer to an int

   ptr = &x;  //store the address of x in ptr

   cout << "Here is the value of x printed twice: "
        << "x = " << x << " *ptr = " << *ptr << endl;

   *ptr = 100;  //we have actually changed the value pointed to by ptr
                   //(which is x) - this is because when you work with a
               //dereferenced pointer you are actually working
                //with the value the pointer is pointing to

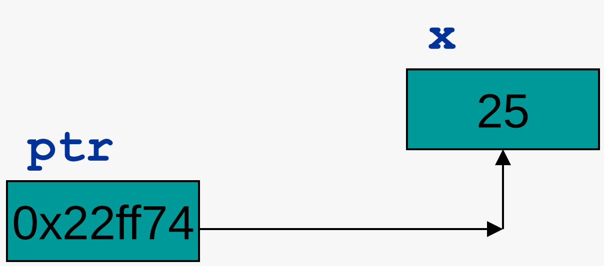
   cout << "Here is the new value of x printed twice: "
        << "x = " << x << " *ptr = " << *ptr << endl;

   system("PAUSE");
   return 0;
}
```

Pointer Variables

- Example #4: This program stores the address of a variable in a pointer
  - **Program Output:**
    - Here is the value of x printed twice: x = 25 *ptr = 25
    - Here is the new value of x printed twice: x = 100 *ptr = 100
    - Press any key to continue . . .. .

  - Every time the expression **`*ptr`** appears in the program, the program indirectly uses the variable **`x`**
    - With the indirection operator, **`ptr`** can be used to indirectly access the variable it is pointing to
      - With this indirect access **`*ptr`** is even able to modify **`x`** in this example



Address of x: 0x22ff74

The value of **`*ptr`** is 25

Pointer Variables

- ***Example #5: (Pointer3.cpp)*** Pointers can point to different variables

```
/*This program demonstrates the ability of a pointer to point to different
variables*/
#include <iostream>
using namespace std;

int main()
{
   int x = 2, y = 2, z = 2;
   int *ptr;   //ptr is a pointer to an int

   cout << "Original Values: x = " << x << " y = " << y << " z = " << z <<
   endl;

   ptr = &x;  //store the address of x in ptr
   *ptr = *ptr * 2;  //we are indirectly changing the value in x
   ptr = &y;  //store the address of y in ptr
   *ptr = *ptr * 3;  //we are indirectly changing the value in y
   ptr = &z;   //store the address of z in ptr
   *ptr = *ptr * 4;  //we are indirectly changing the value in z

   cout << "Modified Values: x = " << x << " y = " << y << " z = " << z <<
   endl;

   system("PAUSE");
   return 0;
}
```

Pointer Variables

- Example #5: Pointers can point to different variables
  - **Program Output:**
    - Original Values: x = 2 y = 2 z = 2
    - Modified Values: x = 4 y = 6 z = 8
    - Press any key to continue . . .

  - Every time the expression **`*ptr`** appears in the program, the program indirectly accesses and modifies the value of the variable pointer is pointing to (**`ptr`** is pointing to the variable stored located in the address stored in it)

- You've seen three different uses of the asterisk **`*`**(star) in C++

1. As the multiplication operator (distance = speed * time;)
2. In the definition of a pointer variable (int* ptr;)
3. As the indirection operator (*ptr = 100;)

Relationship Between Arrays and Pointers

- An array name without the brackets and subscript actually represents the starting address of the array in memory
  - This means that ***an array name is really a pointer to the starting memory address of the array which holds the first element*** ***(Example #6) (Array1.cpp)***

```
/*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}; //declares and initializes an array
                                          //of 5 shorts

   //the array name numbers is a pointer to the beginning of the array, the //first
    element is stored in the first memory location the array name is //pointing to

   //Notice that numbers is dereferenced so we print the value in the first
    //element of the array rather than the address of the first element in the
    //array
   cout << "The first element of the array is " << *numbers << endl;

   system("PAUSE");
   return 0;
}
```

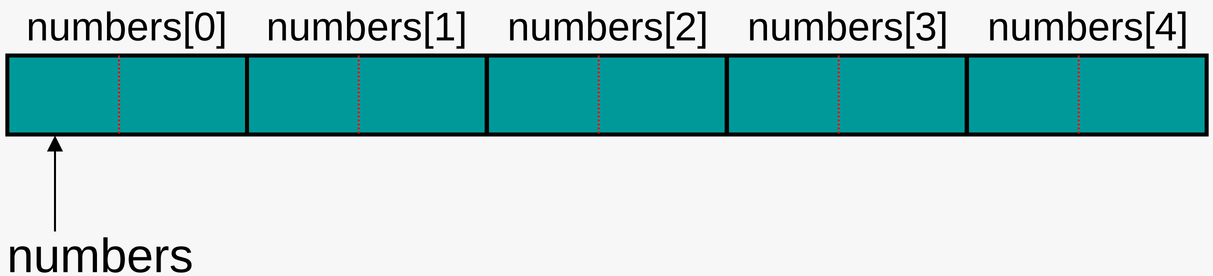
  - ***Remember an array of ints is a list of ints stored sequentially in memory***

Relationship Between Arrays and Pointers

- Example #6 An array name is really a pointer (Example #6)
  - Output:
    - The first element of the array is 10
    - Press any key to continue . . .

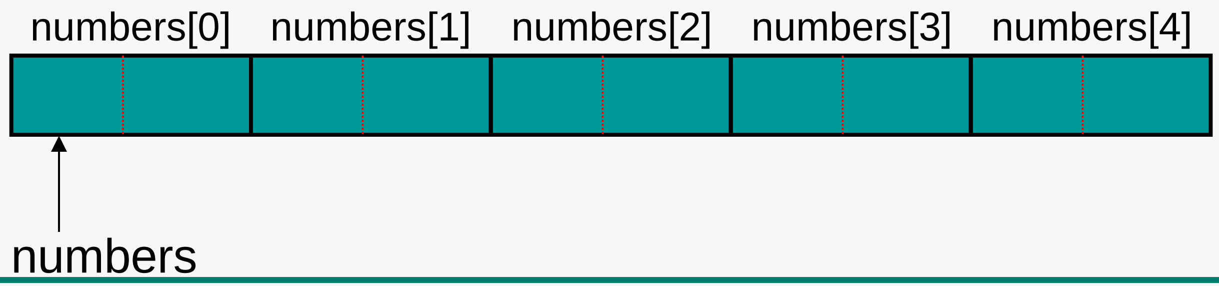
- Because **`numbers`** in our numbers array works like a pointer to the starting address of the array, the first element was retrieved when **`numbers`** was de-referenced using the indirection operator **`*`**

- Remember Array Elements are stored together in memory as shown in the figure below
  - A short is normally two bytes so the figure shows two bytes being allocated for each array element



Relationship Between Arrays and Pointers

- The entire array could be retrieved using pointers and the indirection operator (`*`) – de-referenced pointers
  - If `numbers` is the address of `numbers[0]`, certain values can be added to `numbers` to get the **addresses of other elements** in the array since the values are stored sequentially in memory
  - However, **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 multiplied by the size of the data type** being referenced by the pointer
    - So if you add 1 to `numbers` you are actually adding `1 * sizeof(short)` **to** `numbers`
    - If you add 2 to `numbers` you are adding: `2* sizeof(short)` to `numbers`



Relationship Between Arrays and Pointers

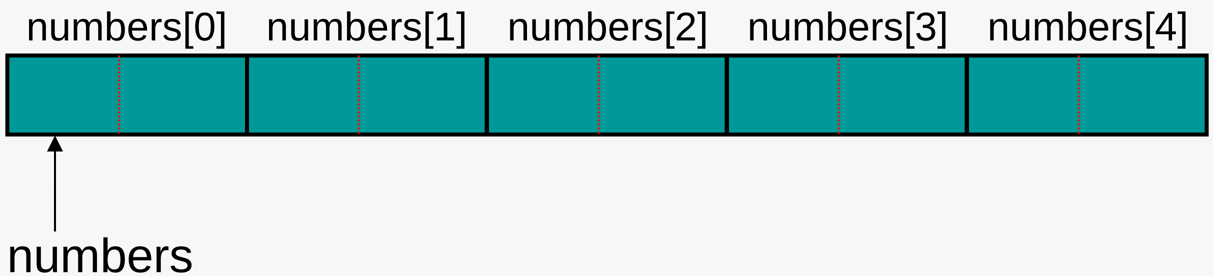
- On the PC this means the following is true because short integers usually use 2 bytes:

```
*(numbers + 1) is actually *(numbers + (1 *2))
*(numbers + 2) is actually *(numbers + (2 *2))
*(numbers + 3) is actually *(numbers + (3 *2))
*(numbers + 4) is actually *(numbers + (4 *2))
```

- This automatic conversion means that an element in an array can be retrieved using its subscript **or by adding its subscript to a pointer to the array**

  - If the expression `*numbers` (which is the same as `*(numbers +0)`) retrieves the first element in the array then `*(numbers + 1)` retrieves the second element and so on

  - ***The parenthesis are CRUCIAL when adding values to pointers***
    - The `*` operator has precedence over the `+` operator
    - `*numbers + 1` is NOT equivalent to `*(numbers + 1)`
    - `*numbers + 1` dereferences the first element in the array and then adds one to the value of the first element in the array while `*(numbers + 1)` adds one to the address in numbers and then dereferences it



Relationship Between Arrays and Pointers

- ***Example #7: (Array2.cpp)*** Program to Process an array using pointer notation

```
/*This program shows an array being processed with pointers*/
#include <iostream>
using namespace std;

int main()
{
   short numbers [] = {10,20,30,40,50}; //declares and initializes an array
                                        //of 5 ints

   //adds 1 * sizeof(short) to the pointer numbers to access the second
   //element of the array an so on through all of the array elements
   cout << "The first element of the array is " << *numbers << endl;
   cout << "The second element of the array is " << *(numbers + 1) << endl;
   cout << "The third element of the array is " << *(numbers + 2) << endl;
   cout << "The fourth element of the array is " << *(numbers + 3) << endl;
   cout << "The fifth (last) element of the array is " << *(numbers + 4) <<
   endl;

   system("PAUSE");
   return 0;
}
```

- ***Look at example with () around numbers + subscript value taken out – it won't work correctly***

Relationship Between Arrays and Pointers

- ***Example #8: (Array3.cpp)*** Program to Process an array using pointer notation – a loop is added to Example #7

```
/*This program shows an array being processed with pointers*/
#include <iostream>
using namespace std;

int main()
{
   short numbers [] = {10,20,30,40,50}; //declares and initializes an array
                                       //of 5 ints

   //adds 1 * sizeof(short) to the pointer numbers to access the second
   //element of the array an so on through all of the array elements
   for(int index = 0; index < 5; index++)
   {
      cout << "numbers[" << index << "] = "<< *(numbers + index) << endl;
   }

   system("PAUSE");
   return 0;
}
```

Relationship Between Arrays and Pointers

- **Example #8:** Program to Process an array using pointer notation – a loop is added to Example #7
  - **Output:**

    numbers[0] = 10
    numbers[1] = 20
    numbers[2] = 30
    numbers[3] = 40
    numbers[4] = 50
    Press any key to continue . . .

  - When working with arrays and pointers remember the following is true
    - **array[index] is equivalent to *(array + index)**

Initializing Pointers

- **Concept:** Pointers may be initialized with the address of an existing object
- Remember a pointer is designed to point to a the address of a specific type of data (int, double etc)
  - When a pointer is initialized it must be initialized to the type of data it was designed to point to
  - LEGAL INITIALIZATIONS:

```
int myValue;
int *ptr = &myValue;  //myValue is an int and ptr is designed to point
                          //to an int

int ages[20];
int *ptr = ages;   //legal because ages is the special pointer of the
                       //first int in an array of integers

//pointers may be defined in the same statement as other variables of
//the same type, ptr is a pointer to an int initialized to the address
//of myValue
int myValue, *ptr = &myValue;

//the following defines an array of doubles readings and a pointer to
//a double, marker, that is initialized with the first element in the
//array
   double readings[50], *marker = readings;
```

  - ILLEGAL:

```
float myFloat;
int *ptr = &myFloat;  //ILLEGAL myFloat is NOT an int

//pointer can only be initialized to with the address of an object that
//has been defined
int *ptr = &myValue;  //ILLEGAL, myValue is not defined yet
int myValue;
```

Initializing Pointers

- Initializing pointers to **NULL**
  - In most computers the memory address at 0 is inaccessible to user programs because it is occupied by operating system data structures
    - This fact allows programmers to signify that a pointer does not point to a memory location accessible to the program by initializing the pointer to 0
    - By initializing a pointer to 0 (NULL) the programmer is initializing the pointer by not having it point to anything
      - NULL is a constant defined in many header files and has a value of zero
      - ***If not initialized the pointer may try to point to something unexpected***
  - **LEGAL INITIALIZATIONS:**

```
//Both of these pointers point to 0, indicating that neither
//is pointing to a legitimate address
int *ptrToInt = 0;
float *ptrToFloat = 0;

//Many header files that are used often such as iostream and
//fstream define a constant named NULL whose value is zero
int *ptrToInt = NULL;
float *ptrToFloat = NULL;
```

    - A pointer whose value is the address zero is often called the **null pointer**

Comparing Pointers

- Pointers may be compared using any of C++'s relational operators: >,<,==,!=,<=, >=
  - If one address comes before another in memory, the first address is considered less than the second
    - **In an array all of the elements are stored in consecutive memory locations**
      - This means the address of element 1 in an array is greater than the address of element 0

Comparing Pointers

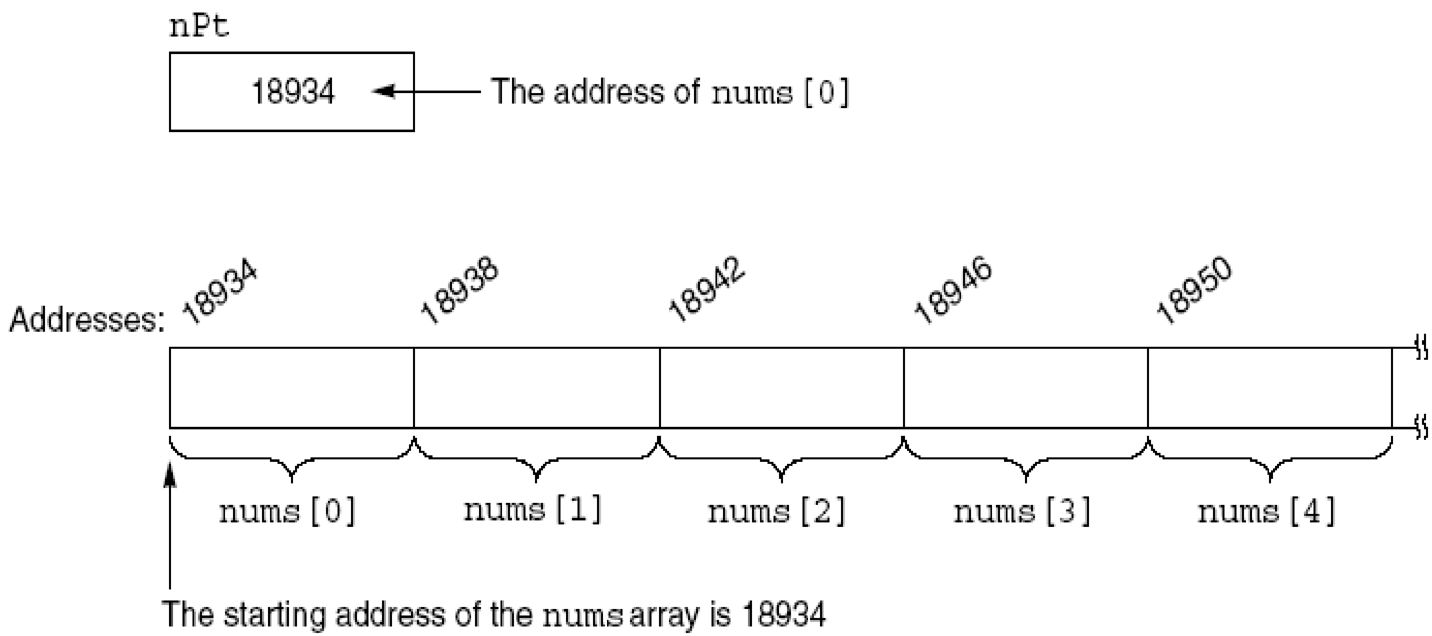


FIGURE 9.16 *The* nums *Array in Memory*

- Because addresses grow larger for each subsequent element in an array the Following are all true for the nums array shown above:

```
&nums[1] > &nums[0];
nums < &nums[4];
nums = = &nums[0];
&nums[2] != &nums[3];
```

Comparing Pointers

- Comparing two pointers is not the same as comparing the values two pointers point to
  - The following statement compares the addresses stored in ptr1 and ptr2
    - `if(ptr1 > ptr2)`
  - The next statement however, compares the values ptr1 and ptr2 point to
    - `if(*ptr1 < *ptr2)`

Comparing Pointers

- ***Example #9 (ComparePtr.cpp)*** – This program shows how you can use addresses to make sure a pointer does not go beyond its boundaries

```
/*This program uses a pointer to display the contents of an integer array.  It
   illustrates the comparison of pointers*/
#include <iostream>
using namespace std;

int main()
{
   const int SIZE = 8;  //to use as the array size
   int set[] = {5,10,15,20,25,30,35,40};  //declare and initialize an array of 8
                                          //ints

   int *nums = set;  //nums is pointing to set (the first element in the array

   //use the pointer to nums print out the set array
  //while the address of nums is less than the address of set[8]
  while(nums < &set[SIZE])   //comparing the pointer num to the address of set[8]
  {
     cout << *nums << " " << endl;
     nums++;
   }

   system("PAUSE");
   return 0;
}
```

Comparing Pointers

- Most comparisons involving pointers compare a pointer to NULL (0) to determine whether or not the pointer points to a legitimate address
  - This is why it's a good idea to initialize pointers either to an address right away or to null right away
    - These kind of comparisons can catch errors in your code where you might be trying to use a pointer that is not initialized properly or pointing to what you want it to be
  - Assuming **`ptrToInt`** has been defined as a pointer to an integer, the following code prints the integer pointed to by **`ptrToInt`** only after checking that the pointer is not **`NULL`**

```
if(ptrToInt != NULL)
    cout << *ptrToInt;
else
   cout << "null pointer";
```

Pointers as Function Parameters

- ***Concept:*** A pointer can be used as a function parameter. It gives the function access to the original argument passed to the function much like a reference parameter does

- In Chapter 6, you learned how to use reference variables as function parameters
  - A reference variable acts as an alias to the original variable used as an argument
  - When a variable is passed into a function parameter that contains a reference parameter the argument is said to be passed by reference

Pointers as Function Parameters – Example Pass by Reference

```
/*This program passes variables by reference into two functions*/(Example 10-PassByRef.cpp)
#include <iostream>
using namespace std;

void getNumber(int&);
void doubleValue(int&);

int main()
{
   int number;

   getNumber(number);  //the variables are passed by reference to the function
   cout << "\nThe number input was " << number << endl;
   doubleValue(number); //the details of how it works are hidden
   cout << "The value of number doubled is " << number << endl;

   system("PAUSE");
   return 0;
}

//since the parameter is passed by reference the variable passed to this function is
//altered by the function
void getNumber(int& input)
{
   cout << "Please enter a number: ";
   cin >> input;
}

//since the parameter is passed by reference the variable passed to this function is
//altered by the function
void doubleValue(int& value)
{
   value = value *2;
}
```

Pointers as Function Parameters

- An alternative to passing a argument to a function by reference is to use a pointer value as the parameter in the function
  - This is essentially the same as passing by reference
  - Passing by reference is simpler
    - **However, reference variables hide the mechanics behind the dereferencing and indirection**
      - so if you pass by reference you can't use the indirection operator inside your function
    - **There may be tasks (especially when dealing with C-Strings) where that are best done passing pointers rather than passing by reference**
  - Here is the `doubleVal` function in the previous program with a pointer passed to it instead of a reference variable

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

    - When value is dereferenced with the `*`, the multiplication operator works on the value pointed to by value
      - This statement `*value = (*value) * 2;` multiplies the original variable passed into the function by 2
    - Here is the function call for this function that takes a pointer parameter

      `doubleValue(&number);`

      - This statement uses the & operator to pass the address of number into the value parameter
    - The next example shows the previous pass by reference example passing pointers instead
      - The results are the same

Pointers as Function Parameters – Example Passing Pointers

```
/*This program uses two functions that accept pointers as arguments*/   (Example 11)_
#include <iostream>
using namespace std;

void getNumber(int*);  //The functions now take pointer parameters
void doubleValue(int*);

int main()
{
   int number;

   getNumber(&number);  //the address of the variable is passed to the function
   cout << "\nThe number input was " << number << endl;
   doubleValue(&number); //the address of the variable is passed to the function
   cout << "The value of number doubled is " << number << endl;

   system("PAUSE");
   return 0;
}

//since the parameter is a pointer so an address should be passed to it
void getNumber(int* input)
{
   cout << "Please enter a number: ";
   cin >> *input;  //The * is necessary to store the value entered by the user
}

//since the parameter is a pointer so an address should be passed to it
void doubleValue(int* value)
{
   *value = (*value) *2;  //The parenthesis ensure the order of operations
}
```

Dynamic Memory Allocation

- Pointers can be used to create and destroy variables while a program is running

- **<u>Why would we want to do this?</u>**
- As long as we know how many variables we need in a program we can define those variables before we use them in our program.
  - **Examples:**
    - program to calculate the area of a circle – we need the radius and PI
    - program to calculate the payroll data for 30 employees
      - We would probably create an array of 30 elements for this (an array of structs or objects)
  - ***Remember we needed to define variables before we can use them in our programs***
    - ***You get a compile time error in your program if you try to use a variable that hasn't been defined***
  - ***In addition remember with arrays the size is fixed.***

- **But what if we don't know how may variables we need in a program?**
  - What if we want a program that calculates payroll data for ANY NUMBER of employees?
  - What if we want to write a program that will calculate the average of ANY NUMBER of student tests?
  - These types of programs would be nice because they would be very versatile **but how can we store the individual data items in memory if we don't know how many variables to define?**
    - What we want to do is have a program that us able to do is create variables "on the fly"

Dynamic Memory Allocation

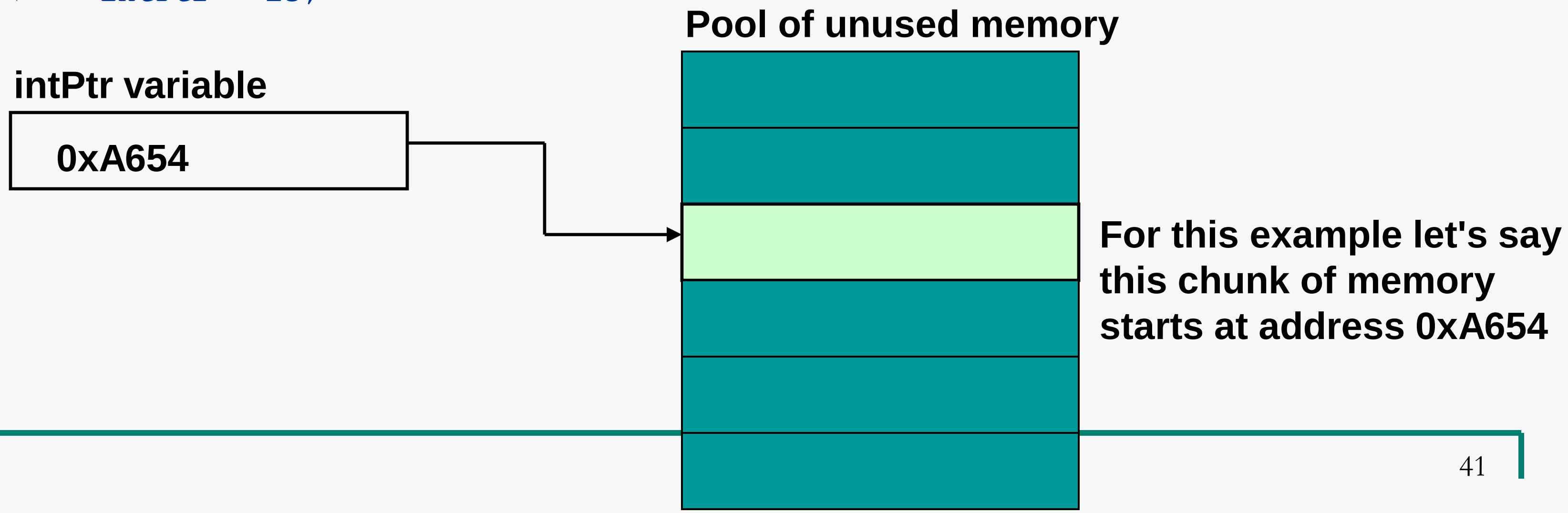
- ***Dynamic Memory Allocation:*** To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type
  - ***Remember so far in our programs we have had to have all variables defined before they could be used otherwise we got compilation errors***
    - ***This was so the compiler could have the required memory set aside to run the program***
  - We still need to have variables defined before we can use them but ***with dynamic memory allocation we're going to be able to define/create variables on the fly at RUN TIME (not at compile time)***

- How will a program dynamically allocate memory for a variable of a specified data type (say an `int`)?
  - While running, the program will make a request to the computer that it allocate enough bytes to store an `int`
  - The computer fulfills this request by finding and setting aside a chunk of unused memory large enough to hold a variable of the requested type (an `int` in this case).
    - This memory is taken from a special section of memory called ***the heap***.
  - ***The computer then gives the program the starting address of the chunk of memory it set aside***
  - **Since the program only has the starting address of the chunk of memory the computer set aside for it the program will only be able to access this newly allocated memory using its address – This means a pointer is required to access these memory bytes**

The `new` operator

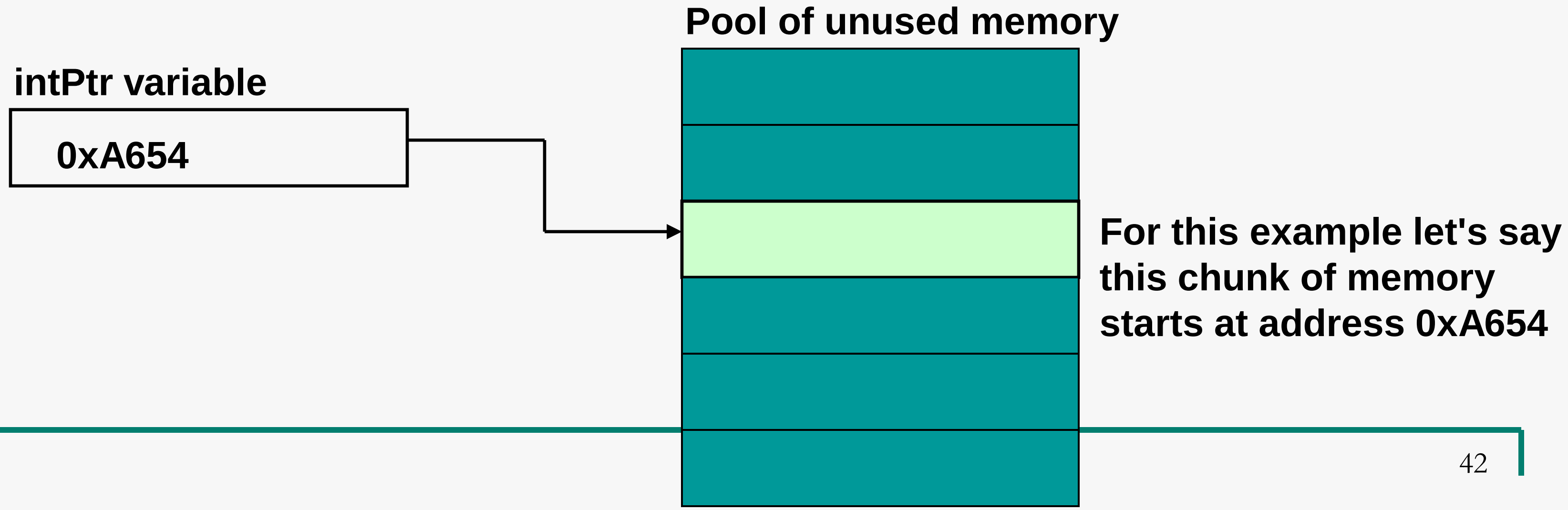
- **The way a C++ program requests dynamically allocated memory is by using the `new` operator**

- Let's say our program wants to dynamically create an `int` type variable while a program is running
  - First the program needs to define a pointer to an `int`:
    - `int *intPtr;`
  - Here is an example of how `intPtr` can be used with the `new` operator:
    - `intPtr = new int;`
    - This statement is requesting that the computer allocate enough memory from the heap for a new `int` variable
    - The operand of the `new` operator is a data type (in this case `int`) of the variable being created
    - The `new` operator returns an address to memory allocated for the requested data type
    - Once the above statement executes, `intPtr` will contain the address of the newly allocated memory
  - A value can be stored in this dynamically created variable by ***dereferencing the pointer***
    - `*intPtr = 25;`



The `new` operator

- The way a C++ program requests dynamically allocated memory is by using the `new` operator
  - `int *intPtr;`
  - `intPtr = new int;`
  - `*intPtr = 25;`
- Operations can be performed on the new variable by simply using the dereferenced pointer – Examples:
  - `cout << *intPtr << endl;`
  - `cin >> *intPtr;`
  - `total = total + *intPtr;`



The `new` operator – Dynamic Array Allocation

- There's not a lot of point in dynamically allocating a **single** variable
- A more practical use of dynamic memory allocation is to use the `new` operator to dynamically allocate an array
  - This will allow us to create arrays of **any** size as the program is running
  - Remember before we had to have the array size determined at compile time so we could NOT let the user enter the size of the array
- The following code dynamically allocates a 100-element array of integers

```
intPtr = new int[100];
```

- Once the array is created, the pointer may be used with subscript notation to access it. Here's a for loop that could be used to store the value 50 in each element of the array:

```
for(int i = 0; i < 100; i++)
{
   intPtr[i] = 50;
}
```

The `new` operator – Dynamic Array Allocation

- What if we tried to dynamically allocate an array and there wasn't enough memory to accommodate our request?
  - What if the program asks for a chunk of memory large enough to hold an 100,000 element array of **`ints`** but that much memory isn't available?
    - The program cannot continue to operate normally and in most cases will terminate

- In newer versions of C++ when memory cannot be allocated, the **`new`** operator will by default cause the termination of the program with an appropriate error message in a process called throwing an exception

- In older versions of C++, the **`new`** operator returns the address of **`0`** (**`NULL`**) if it cannot allocate the requested memory
  - In these older version of C++, a program calling/using **`new`** should first check the address returned by new and make sure it is not **`0`** before using it.
  - So in an older version of C++ you might do something like this:

```
//for older versions of C++
if(intPtr == NULL)
{
   cout << "Error Allocating Memory" << endl;
   exit(1);
}
```

The `delete` operator

- When a program is finished using a dynamically allocated chunk of memory, it should release that memory for future use
  - You don't want chunks of memory no longer being used reserved so they can't be reused
    - If you fail to release the memory it will stay reserved and won't be able to be used by other programs either

- The **`delete`** operator is used to free memory that was allocated with **`new`**

- Here's how the **`delete`** operator can be used to free a **single** variable

```
delete intPtr;
```

The `delete` operator

- If `intPtr` points to a dynamically allocated array the `[]` symbol must be placed between delete and `intPtr` to free the array

```
delete [] intPtr;
```

- Any memory allocated in the constructor of a class should be deallocated in the destructor of a class

- **WARNING: Only use the `delete` operator with pointers that were previously used with `new`. If you use a pointer with `delete` that does NOT reference dynamically allocated memory (a regular pointer not used with `new`) unexpected problems could (and usually will) result.**

Dynamic Memory Allocation – *Example 11a* (Example11a.cpp)

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

int main()
{
   double *sales;  //a pointer to a double - we'll use it to point to our dynamically allocated memory
   double total = 0;
   double average;
   int numDays;

   cout << "How many days to you wish to process sales figures for?" << endl;
   cin >> numDays;  //this is how big we want our array to be - unknown until runtime
   sales = new double[numDays];   //dynamically allocate memory for the array of the user input size

   //now get the sales figures from the user
   cout << "You will enter the sales figures for each day below:" << endl;
   for(int i = 0; i < numDays; i++)
   {
       cout << "Day " << (i + 1) << ": ";
        cin >> sales[i];
   }

   //calculate total sales
   for(int i = 0; i < numDays; i++)
   {
        total = total + sales[i];
   }

   average = total/numDays;

   cout <<fixed << setprecision(2) << "\nTotal Sales = " << total << "\nAverage Sales =  "<< average << endl;
   delete [] sales;  //don't forget to free the dynamically allocated memory
   system("PAUSE");
   return 0;
}
```

Returning Pointers from Functions

- Functions can return pointers, but you must make sure the object the function is referencing still exists when the pointer is returned

- **Example:** This function locates the null terminator `'\0'` (the last character in a C-Style string) and returns a pointer to it (since the null terminator is a char the function returns a pointer to a char)
  - Remember a C-string is just an array of characters so we're passing a pointer to the first element of the string array into the function
  - This example is OK

```
char *findNull(char *str)
{
  char *ptr = str;
   while(ptr != '\0')
   {
     ptr++;
   }

   return ptr;
 }
```

- The `char *` return type in the function header indicates the function returns a pointer to a `char`

Returning Pointers from Functions

- Be careful when writing functions that return pointers – they can contain elusive bugs
- Example: What's wrong with the following function?

```
char *getName()
{
  char name[81];             //a C-string with 80 chars plus '\0'
  cout << "Enter your name;
  cin.getline(name,81);      //a special getline for C-Style Strings
  return name;  //name would be a pointer to the first element of  the
                //char array
}
```

- The problem?
  - The function returns a pointer to an object that no longer exists
  - Because **`name`** is a local array the function **`getName`**, it is destroyed when the function returns
  - Since the returned pointer points to an object that no longer exists attempting to use the pointer will result in unpredictable results and errors

Returning Pointers from Functions

- Be careful when writing functions that return pointers – they can contain elusive bugs

- **RULES:** Only return a pointer from a function if it is:
  1. A pointer to an object passed into the function as an argument
  2. A pointer to a dynamically allocated object

- Corrected Function

```
char *getName(char *name)
{
  cout << "Enter your name;
   cin.getline(name,81);   //assigns user input to name array
  return name;  //name would be a pointer to the first element of
                           //the char array
}
```

- Now name points to a memory location that was valid before the function was called so it sill exists after the function exits

Pointers to Structures

- **Concept:** You can create pointers that point to structures

- Declaring a variable that is a pointer to a `struct` is the same as declaring any other pointer variable: the data type followed by an asterisk and the name of the pointer variable:

```
struct Circle  //the Circle structure type
{
   double radius;
};

Circle piePlate;  //declare a variable of type circle

Circle *circPtr; //declare a pointer to a Circle Structure

circPtr = &piePlate //store the address of piePlate in the
                    //circPointer variable

//to dereference the circle pointer parenthesis must be used
//the dot operator has higher precedence than the indirection operator
(*circPtr).radius = 10;  //sets piePlate’s member radius to 10

//A special operator called the structure/object pointer operator can be
//used to dereference both structs and objects using pointers
circPointer ->radius = 10;  //does the same thing as the previous statement
```

Pointers to Structures – Example 12 (StructPointers.cpp)

```
/*This program illustrates the use of pointers to structs*/
#include <iostream>
using namespace std;

struct Circle  //declare the type Circle
{
  double radius;
};

int main()
{

   Circle piePlate;  //declare a variable of type Circle

   Circle *circPtr; //declare a pointer to a Circle Structure

   circPtr = &piePlate; //store the address of piePlate in the circPointer variable

   //set piePlate's member radius to 10 using the circPointer
   circPtr->radius = 10;

   cout << "piePlate's radius is " << circPtr->radius << endl;

   system("PAUSE");
   return 0;
}
```

Pointers to Class Objects - Example 13

- Concept: You can create pointers that point to class objects (instances of classes)
  - If you go into game programming you will see this a lot/ Works very similarly to stucts

- The Circle class we went over last time;

```
class Circle  //class declaration
{
   private:
     double radius;

   public:                  //function prototypes
     void setRadius(double);
     double getRadius();
     double getArea();
};

 void Circle::setRadius(double r)  //start of the implementation section
 {
    if(r > 0)
       radius = r;
    else
       radius = 0;
 }

double Circle::getRadius()
{
   return radius;
}

double Circle::getArea()
{
   return 3.141592 * pow(radius,2);
}
```

Pointers to Class Objects

- **Concept:** You can create pointers that point to class objects (instances of classes)
  - If you go into game programming you will see this a lot/ Works very similarly to stucts

- Using pointers with the circle class

```
Circle circ1;  //instantiate a variable of type Circle

Circle *circPtr; //define a pointer to a circle called
      //circPtr

circPtr = &circ1; //circPtr points to the circ1 object
       //(contains circ1's address)

//the cirPtr can then be used to call circ1's member
//functions using the -> operator
circPtr->setRadius(15);  //sets circ1's radius to 15

double area;
area = circPtr->getArea()  //calls circ1's getArea
         //member function and stores
        //it in the area variable
```

- **See Example**

Pointers to Class Objects

- **Example: Pointers to Class Objects (*Example 13*)**

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

int main()
{
    Circle circ1;  //instantiate a variable of type Circle
    double area;  //to hold the area of the circle after its calculated

    Circle *circPtr; //define a pointer to a circle called circPtr

    circPtr = &circ1; //circPtr points to the circ1 object (contains circ1's
              //address)

    //the cirPtr can then be used to call circ1's member
    //functions using the -> operator
    circPtr->setRadius(10);  //sets circ1's radius to 10


   area = circPtr->getArea();  //calls circ1's getArea member function and stores
                                  //the result in the area variable

   cout << "circ1's radius is " << circ1.getRadius() << " and the area is " << area << endl;

   system("PAUSE");
   return 0;
}
```

Dynamically Allocating Class Objects and Structs

- Class objects and struct type variables may be dynamically allocated in memory just like other variable types

- For example assume a class named Rectangle exists and the following pointer to a Rectangle is declared in a program
  - `Rectangle *boxPtr;`

- You may use the pointer above to dynamically allocate an object of the Rectangle class like this:
  - `boxPtr = new Rectangle;`

- **When the `new` operator creates the Rectangle class object in memory, its constructor is executed**.
  - Since the constructor is executed you may pass arguments to the dynamically allocated object's constructor like this:
  - `boxPtr = new Rectangle(10,20);`

- The delete operator destroys dynamically allocated objects:
  - `delete boxPtr;`

Dynamically Allocating Class Objects

- Often in a class definition the constructor dynamically allocates memory
- What if you wanted to have an array member of your class but wanted the array to be any length?
  - You need a pointer in your class and you want to dynamically allocate the array when the object is created
- If you dynamically allocate a member of an array when an object is created in a constructor you also need to free up that memory when the object is destroyed
  - Remember a **destructor** is run when an object is destroyed
- The next example shows how to have a dynamically allocated array as a class member
  - NOTE: You DON'T need to do this in your homework (unless you really wanted to try it)

Example 14

```
class StatsArray
{
     private:
        double *list;
        int listSize;
     public:
          StatsArray(int);
           void displayArray();
         ~StatsArray();
};

StatsArray::StatsArray(int size)
{
     list = new double[size];
     listSize = size;

     for(int i = 0; i < size; i++) {       //initializes all array values to 5.1
             list[i] = 5.1;
     }
}

StatsArray::~StatsArray()
{
     delete [] list;
}

void StatsArray::displayArray()
{
   for(int i = 0; i < listSize; i++) {
        cout << list[i] << endl;
   }
}

int main()
{
     int arrSize;
     cout << "How big would you like the array to be?" << endl;
     cin >> arrSize;
     StatsArray arr(arrSize);
     arr.displayArray();

     system("PAUSE");
    return 0;
}
```