Pointers

Chapter 10

Variable Declaration

- When a variable is defined, three fundamental attributes are associated with it:
  - Name
  - Type
  - Address
- The variable definition associates the name, the type, and the address of the location where the variable is to be stored
Names and Addresses

The number of storage locations allocated for the variable is determined by the type, e.g., one byte for a char

Accessing Values and Addresses

- The value of a variable is accessed by means of the variable name
- A variable's address is accessed by means of the address operator: &

```cpp
int x = 42;
cout << " The value of x is " << x;
cout << " x is stored in location " << &x;
cout << " The size of X is " << sizeof(x) << " bytes\n";
```
Pointer Variables

- The & operator returns the address of the variable to which the operator is applied.

What might we want to do with that address, other than print it?

How about store it?

- The type of variable that stores an address is a pointer variable, usually just called a pointer.

- Just as int variables are designed to store and work with integers, pointers are designed to store and work with addresses.

- A pointer can be thought of as a "locator"; it tells us where to locate another variable. That variable is of a specific type, so we declare a pointer as a pointer to a type:
  ```
  int * ptr1;  //ptr1 is a pointer to an int
  // ptr1 can hold the address of an int
  float * ptr2;// ptr2 is a pointer to a float
  // ptr2 can hold the address of a float
  ```

- The * (which is part of the data type) in front of the variable name indicates the variable is a pointer.
**Pointer Variables**

Example:

```c
int X = 25;
int * ptr = &X;
// place the address of X in pointer ptr
```

---

**Pointers are useful for the following:**

- Indirectly manipulating data stored in other variables
- Working 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
- Creating arbitrarily-sized lists of values in memory
# Pointer Variables – Program

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

int main ( void )
{
    int X = 25;
    int *Ptr;

    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;
} // main
```

The value in X is 25
The address of X is 0x7e00

---

## Dereferencing a Pointer

- The real benefit of pointers is that they allow us to indirectly access and modify the variables they point to

- Using a pointer to obtain the value to which it points is called **dereferencing** the pointer
The Indirection Operator

- Dereferencing is accomplished by using the *indirection (or dereferencing) operator*, 
  * (this is different than the * used in declarations)

  Example:
  - *ptr is the value pointed to by ptr
  - *ptr is the value that is found at the address stored in ptr

The Indirection Operator

The address operator (&) and the indirection operator (*) are inverses

Let’s look at what that means:

\[
\begin{align*}
\text{int } x &= 15; \\
\text{int } *ptr &= \&x;
\end{align*}
\]
Indirection Operator – Program

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

int main ( void )
{
    int X = 25;
    int *Ptr;
    Ptr = &X;  // Store the address of X in Ptr
    cout << "Here is the value in X, printed twice:\n";
    cout << X << "  " << *Ptr << endl;
    *Ptr = 100;
    cout << "Once again, here is the value in X:\n";
    cout << X << "  " << *Ptr << endl;
    return 0;
} // main
```

Using Pointers – Program

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

int main ( void )
{
    int X = 25, Y = 50, Z = 75;
    int *Ptr;
    cout << "Here are the values of X, Y, and Z:\n";
    cout << X << "  " << Y << "  " << Z << endl;
    Ptr = &X;
    *Ptr *= 2;
    Ptr = &Y;
    *Ptr *= 2;
    Ptr = &Z;
    *Ptr *= 2;
    cout << "Once again, here are the values of X, Y, and Z:\n";
    cout << X << "  " << Y << "  " << Z << endl;
    return 0;
} // main
```
Relationship Between Arrays and Pointers

- Recall: an array name, without brackets or subscript, represents the starting address of the array.
- That means that array names can be used as pointers, and vice-versa.
- As we've already seen, the address a pointer points to can be changed.
- However, the address an array name points to cannot be changed, because an array name is a constant pointer.

Basic Pointer Arithmetic

For declaration

```c
short Numbers [5];
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Numbers
Basics of Pointer Arithmetic

short Numbers[5];

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numbers</td>
<td>(Numbers+1)</td>
<td>(Numbers+2)</td>
<td>(Numbers+3)</td>
<td>(Numbers+4)</td>
</tr>
</tbody>
</table>
```

• Notice that adding 1 to the pointer Numbers, actually adds 1 multiplied by the size of the variable type
• *(Numbers+1) is really *(Numbers + 1 * sizeof(short))
• If Numbers was declared as double, *(Numbers+2) is really *(Numbers + 2 * sizeof(double))

Basics of Pointer Arithmetic

*arrayname (same as *(arrayname + 0) refers to arrayname[0]
*(arrayname + 1) refers to arrayname[1]
*(arrayname + 2) refers to arrayname[2]
  etc.

In general
*(arrayname + index) is equivalent to arrayname[index]

Be careful not to give a pointer an address outside the bounds of the array.
Arrays and Pointers – Program

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

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

    cout << "The first element of the array is ";
    cout << *Numbers << endl;
    cout << "The third element of the array is ";
    cout << *(Numbers + 2) << endl;
    return 0;
}
``` // main

The first element of the array is 10
The third element of the array is 30

Pointers and Arrays – Program

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

int 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:
```
```cpp
Enter five numbers: 5 10 15 20 25 [Enter]
Here are the numbers you entered:
5 10 15 20 25
```cpp
```
Difference Between Array Names and Pointers

Given

```c
float x[20], y[20];
float * fptr;
```

which of the following are legal?

- `fptr = x;` **legal**
- `fptr = y;` **legal**
- `x = y;` **illegal**
- `y = fptr;` **illegal**

More Pointers and Arrays – Program

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

int main ( void )
{
    float Coins[5] = { 0.05, 0.1, 0.25, 0.5, 1.0 };  
    float *FloatPtr; // Pointer to a float 
    int Count;       // Array index 

    FloatPtr = Coins; // FloatPtr now points to Coins array 
    cout << setprecision(2) << fixed;  
    cout << "Here are the values in the Coins array:\n";
```
More P and A – Program

```cpp
for ( Count = 0; Count < 5; Count++ )
    cout << FloatPtr[Count] << " ";

cout << "\nAnd here they are again:\n";

for ( Count = 0; Count < 5; Count++ )
    cout << *(Coins + Count) << " ";

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

Here are the values in the Coins array:
0.05 0.10 0.25 0.50 1.00
And here they are again:
0.05 0.10 0.25 0.50 1.00

Elements of the Array – Program

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
int main ( void )
{
    float Coins[5] = { 0.05, 0.1, 0.25, 0.5, 1.0 };
    float *FloatPtr; // Pointer to a float
    int Count;       // Array index
    cout << setprecision(2);
    cout << "Here are the values in the Coins array:\n";
```
Elements of the Array – Program (cont.)

for ( Count = 0; Count < 5; Count++ )
{ FloatPtr = &Coins[Count];
  cout << *FloatPtr << " ";
} // for
cout << endl;
} // main

Here are the values in the Coins array:
0.05 0.10 0.25 0.50 1

Chapter 10

Pointer Arithmetic

- Some mathematical operations may be performed on pointers
  - The ++ and -- operators may be used to increment or decrement a pointer variable
  - An integer may be added to or subtracted from a pointer variable.
    - This may be performed with the +, -, +=, or -= operators
  - A pointer may be subtracted from another pointer
# Pointer Arithmetic – Program

```cpp
#include <iostream>
using namespace std;
int main ( void )
{
    int Set[8] = { 5, 10, 15, 20, 25, 30, 35, 40 };
    int *Nums, Index;
    Nums = Set;
    cout << "The numbers in Set are:\n";
    for ( Index = 0; Index < 8; Index++, Nums++ )
        cout << *Nums << " ";
    cout << "\nThe numbers in Set backwards are:\n";
    for ( Index = 0; Index < 8; Index++ )
        { Nums--;
            cout << *Nums << " ";
        } // for
    return 0;
} // main
```

The numbers in Set are:
```
5 10 15 20 25 30 35 40
```
The numbers in Set backwards are:
```
40 35 30 25 20 15 10 5
```

---

## Initializing Pointers

- Pointers may be initialized with the **address** of an **existing** object
Which are valid?

| int one;         | float two, *ptr = &two; |
| int *ptr = one; |                      |
| int ages[20];   | int *ptr = &three;    |
| int *pt_arr = ages; | int three;           |
| float two;      | double *ptr;          |
| int *ptr = &two; | *ptr = 3.14159;       |

Comparing Pointers

- If one address comes before another address in memory, the first address is considered “less than” the second
- C++’s relational operators maybe used to compare pointer values
Addresses in an Array

An array of five integers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x5A00</td>
<td>0x5A04</td>
<td>0x5A08</td>
<td>0x5A0C</td>
<td>0x5A0F</td>
</tr>
</tbody>
</table>

Comparing Pointers – Program

#include <iostream>
using namespace std;
int main ( void )
{
    int Set[8] = { 5, 10, 15, 20, 25, 30, 35, 40 };
    int *Nums = Set;
    cout << "The numbers in Set are:\n";
    cout << *Nums << " ";
    for ( ; Nums < &Set[7]; )
    {
        Nums++;
        cout << *Nums << " ";
    } // for
Comparing Pointers – Program

cout << "\nThe numbers in Set backwards are:\n";
cout << *Nums << " ";
for ( ; Nums > Set; )
{
    Nums--;
    cout << *Nums << " ";
}
} // for
return 0;
} // main

The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backwards are:
40 35 30 25 20 15 10 5

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
#include <iostream>
using namespace std;

void GetNumber ( int * );
void DoubleValue ( int * );

int main ( void )
{
    int Number;
    GetNumber( &Number );
    DoubleValue( &Number );
    cout << "That value doubled is " << Number << endl;
    return 0;
} // main

Enter an integer number: 10 [Enter]
That value doubled is 20
#include <iostream>
#include <iomanip>
using namespace std;

void GetSales ( float *);
float TotalSales ( float *);

int main ( void )
{
    float Sales[4];

    GetSales( Sales );
    cout << setprecision(2);

    cout << fixed << showpoint;
    cout << "The total sales for the year are "$;
    cout << TotalSales(Sales) << endl;
    return 0;
} // main

void GetSales ( float *Array )
{
    for ( int Count = 0; Count < 4; Count++ )
    {
        cout << "Enter the sales figure for quarter ";
        cout << (Count + 1) << ", ";
        cin >> Array[Count];
    } // for
} // GetSales
More Pointers and Parameters – Program

float TotalSales ( float *Array )
{
  float Sum = 0.0;
  
  for ( int Count = 0; Count < 4; Count++ )
  {
    Sum += *Array;
    Array++;
  } // for

  return Sum;
} // TotalSales

More Pointers and Parameters – Output

Enter the sales figure for quarter 1: 10263.98 [Enter]
Enter the sales figure for quarter 2: 12369.69 [Enter]
Enter the sales figure for quarter 3: 11542.13 [Enter]
Enter the sales figure for quarter 4: 14792.06 [Enter]
The total sales for the year are $48967.86
Dynamic Memory Allocation

- Dynamic Memory Allocation: allowing the program to create its own variables "on the fly"

- With dynamic memory allocation, variables may be created and destroyed while a program is running

- 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

Dynamic Memory Allocation: the **new** Operator

- Use the new operator to dynamically allocate memory

```cpp
int * iptr;
iptr = new int;
//iptr contains address of
//newly allocated memory
*iptr = 25;
// assigns 25 to the new variable
```
Dynamic Memory Allocation of an Array

- Use the new operator to dynamically allocate memory for an array
  ```
  int * iptr;
  iptr = new int[100];
  // iptr contains address of the first element of newly allocated array of 100 elements
  for (int count=0; count<100; count++)
    iptr[count]=0;
  // using array notation to initialize values of the new array to 0
  ```

Dynamic Memory Allocation: the Null Pointer

- When memory cannot be dynamically allocated, the new operator returns the address 0 or NULL (NULL is a named constant in iostream)
- A pointer that contains the address 0 is called a null pointer
- A program that uses dynamically allocated memory should always check to see if the new operator returns NULL
Checking for the Null Pointer

```c
int * iptr;
iptr = new int[100];
if ( iptr == NULL )
{  cout << "Memory Allocation Error\n";
   return;
} // if
```

Dynamic Memory Allocation
the delete Operator

- Use the delete operator to dynamically deallocate memory that was previously allocated using new.
  - For a single variable
delete iptr;
  - for an array
delete [ ] iptr;
Dynamic Arrays – Program

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

int main ( void )
{
    float *Sales, Total = 0, Average;
    int NumDays;

    cout << "How many days of sales figures do you wish ";
    cout << "to process? ";
    cin >> NumDays;
    Sales = new float [NumDays];

    if ( Sales == NULL )
    {
        cout << "Error allocating memory!
";
        return;
    } // if

    // Get sales
    cout << "Enter the sales figures below.
";
    for ( int Count = 0; Count < NumDays; Count++ )
    {
        cout << "Day " << (Count + 1) << ": ";
        cin >> Sales[Count];
    } // for

    // Calculate the total sales
    for ( Count = 0; Count < NumDays; Count++ )
        Total += Sales[Count];
}
Dynamic Arrays – Program

// Calculate the average sales per day
Average = Total / NumDays;

// Display the results
cout << fixed << showpoint << setprecision(2);
cout << "\n\nTotal Sales: $" << Total << endl;
cout << "Average Sales: $" << Average << endl;

delete [] Sales;
return 0;
} // main

Dynamic Arrays – Output

How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]
Total Sales: $4067.13
Average Sales: $813.43
Returning Pointers from Functions

- Functions can return pointers, but you must be sure the object the pointer references still exists.
- You should only return a pointer from a function if it is:
  - A pointer to an object that was passed into the function as an argument.
  - A pointer to a dynamically allocated object.

Given the function:

```cpp
char *getName()
{
    char name[81];
    cout << "Enter your name: ",
    cin.getline( name, 81 );
    return name;
} // getName
```

Will this work?

Returns a pointer to a local variable. What happens to local variables when the function returns?
Returning Pointers from Functions

How about this one?

```cpp
// main function
char *title, *name;
title = getName( name );
...
char *getName(char *name)
{
    cout << "Enter your name: ";
    cin.getline( name, 81 );
    return name;
} // getName
```

Returning Pointers from Functions

And this one?

```cpp
char *getName()
{
    char *name;
    name = new char[81];
    cout << "Enter your name: ";
    cin.getline( name, 81 );
    return name;
} // getName
```