Objectives

• In this chapter you will learn about:
  – Addresses and pointers
  – Array names as pointers
  – Pointer arithmetic
  – Passing addresses
  – Common programming errors
Addresses and Pointers

• The address operator, \&, accesses a variable’s address in memory
• The address operator placed in front of a variable’s name refers to the address of the variable
  – \&num means the address of num
Addresses and Pointers (continued)

Program 12.1

```cpp
#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 = " << &num << endl;

    return 0;
}
```
Addresses and Pointers (continued)

Figure 12.1 A more complete picture of the `num` variable
Storing Addresses

• Addresses can be stored in a suitably declared variable

Figure 12.2 Storing num’s address in numAddr
Storing Addresses (continued)

• Example statements store addresses of the variable \texttt{m}, \texttt{list}, and \texttt{ch} in the variables \texttt{d}, \texttt{tabPoint}, and \texttt{chrPoint}

\[
\begin{align*}
\texttt{d} & = \&\texttt{m}; \\
\texttt{tabPoint} & = \&\texttt{list}; \\
\texttt{chrPoint} & = \&\texttt{ch};
\end{align*}
\]

• \texttt{d}, \texttt{tabPoint}, and \texttt{chrPoint} are called pointer variables or pointers
Storing Addresses (continued)

Figure 12.3 Storing more addresses
Using Addresses

- To use a stored address, C++ provides the **indirection operator**, *,
- The * symbol, when followed by a pointer, means “the variable whose address is stored in”
  - *numAddr means the variable whose address is stored in numAddr
Using Addresses (continued)

Figure 12.4 Using a pointer variable

A pointer variable \( y \)

The contents at address \( mmmm \) are \( bbbb \)

The contents of \( y \) are an address

\( mmmm \)
Using Addresses (continued)

• When using a pointer variable, the value that is finally obtained is always found by first going to the pointer for an address
• The address contained in the pointer is then used to get the variable’s contents
• Since this is an indirect way of getting to the final value, the term **indirect addressing** is used to describe it
Declaring Pointers

• Like all variables, pointers must be declared before they can be used to store an address

• When declaring a pointer variable, C++ requires specifying the type of the variable that is pointed to
  – Example: \texttt{int \,*numAddr;}

• To understand pointer declarations, reading them “backward” is helpful
  – Start with the indirection operator, \texttt{\,*}, and translate it as “the variable whose address is stored in” or “the variable pointed to by”
Declaring Pointers (continued)

Program 12.2

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

int main()
{
    int *numAddr;       // declare a pointer to an int
    int miles, dist;   // declare two integer variables

    dist = 158;        // store the number 158 in dist
    miles = 22;        // store the number 22 in miles
    numAddr = &miles;  // store the 'address of miles' in numAddr

    cout << "The address stored in numAddr is " << numAddr << endl;
    cout << "The value pointed to by numAddr is " << *numAddr << "\n\n";

    numAddr = &dist;   // now store the address of dist in numAddr
    cout << "The address now stored in numAddr is " << numAddr << endl;
    cout << "The value now pointed to by numAddr is " << *numAddr << endl;

    return 0;
}
```
Figure 12.5  Addressing different data types by using pointers
References and Pointers

• A reference is a named constant for an address
  – The address named as a constant cannot be changed

• A pointer variable’s value address can be changed

• For most applications, using references rather than pointers as arguments to functions is preferred
  – Simpler notation for locating a reference parameter
  – Eliminates address (\&) and indirection operator (*) required for pointers

• References are automatically dereferenced, also called implicitly dereferenced
Reference Variables

• References are used almost exclusively as formal parameters and return types
• Reference variables are available in C++
• After a variable has been declared, it can be given additional names by using a reference variable
• The form of a reference variable is:
  – `dataType& newName = existingName;`
• Example: `double& sum = total;`
• The * symbol is called the dereferencing operator
Reference Variables (continued)

Figure 12.6  \textit{sum} is an alternative name for \textit{total}
Reference Variables (continued)

Program 12.3

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

int main()
{
    double total = 20.5;       // declare and initialize total
    double& sum = total;       // declare another name for total

    cout << "sum = " << sum << endl;
    sum = 18.6;                 // this changes the value in total
    cout << "total = " << total << endl;

    return 0;
}
```
Array Names as Pointers

• There is a direct and simple relationship between array names and pointers

![Diagram showing the grade array in storage]

**Figure 12.9** The `grade` array in storage

• Using subscripts, the fourth element in grade is referred to as `grade[3]`, address calculated as:

```
&grade[3] = &grade[0] + (3 * sizeof(int))
```
Array Names as Pointers (continued)

Figure 12.10 Using a subscript to obtain an address
Array Names as Pointers (continued)

Program 12.4

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

int main()
{
    const int ARRAYSIZE = 5;

    int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

    for (i = 0; i < ARRAYSIZE; i++)
        cout << "\nElement " << i << " is " << grade[i];

    cout << endl;

    return 0;
}
```
Array Names as Pointers (continued)

Figure 12.11  The variable pointed to by *gPtr is grade[0]
Array Names as Pointers (continued)

Figure 12.12  An offset of 3 from the address in \textit{gPtr}
Array Names as Pointers (continued)

<table>
<thead>
<tr>
<th>Array Element</th>
<th>Subscript Notation</th>
<th>Pointer Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 0</td>
<td>grade[0]</td>
<td>*gPtr or (gPtr + 0)</td>
</tr>
<tr>
<td>Element 1</td>
<td>grade[1]</td>
<td>*(gPtr + 1)</td>
</tr>
<tr>
<td>Element 2</td>
<td>grade[2]</td>
<td>*(gPtr + 2)</td>
</tr>
<tr>
<td>Element 3</td>
<td>grade[3]</td>
<td>*(gPtr + 3)</td>
</tr>
<tr>
<td>Element 4</td>
<td>grade[4]</td>
<td>*(gPtr + 4)</td>
</tr>
</tbody>
</table>

**Table 12.1** Array Elements Can Be Referenced in Two Ways
Array Names as Pointers (continued)

Figure 12.13 The relationship between array elements and pointers
Array Names as Pointers (continued)

Program 12.5

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

int main()
{

    const int ARAYSIZE = 5;

    int *gPtr;              // declare a pointer to an int
    int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

    gPtr = &grade[0];       // store the starting array address
    for (i = 0; i < ARAYSIZE; i++)
        cout << "\nElement " << i << " is " << *(gPtr + i);

    cout << endl;

    return 0;
}
```
Array Names as Pointers (continued)

Figure 12.14  Creating an array also creates a pointer
#include <iostream>
using namespace std;

int main()
{
    const int ARRAYSIZE = 5;

    int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

    for (i = 0; i < ARRAYSIZE; i++)
        cout << "Element " << i << " is " << *(grade + i);
    cout << endl;

    return 0;
}
Dynamic Array Allocation

- As each variable is defined in a program, sufficient storage for it is assigned from a pool of computer memory locations made available to the compiler.
- After memory locations have been reserved for a variable, these locations are fixed for the life of that variable, whether or not they are used.
- An alternative to fixed or static allocation is **dynamic allocation** of memory.
- Using dynamic allocation, the amount of storage to be allocated is determined or adjusted at run time.
  - Useful for lists because allows expanding or contracting the memory used.
Dynamic Array Allocation (continued)

- **new** and **delete** operators provide the dynamic allocation mechanisms in C++

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or <strong>NULL</strong> if not enough memory is available.</td>
</tr>
<tr>
<td>delete</td>
<td>Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator.</td>
</tr>
</tbody>
</table>

**Table 12.2** The **new** and **delete** Operators (Require the **new** Header File)
Dynamic Array Allocation (continued)

- Dynamic storage requests for scalar variables or arrays are made as part of a declaration or an assignment statement
  - Example:
    ```
    int *num = new int;         // scalar
    ```
  - Example:
    ```
    int *grades = new int[200]; // array
    ```
    - Reserves memory area for 200 integers
    - Address of first integer in array is value of pointer variable `grades`
Pointer Arithmetic

- Pointer variables, like all variables, contain values
- The value stored in a pointer is a memory address
- By adding or subtracting numbers to pointers you can obtain different addresses
- Pointer values can be compared using relational operators (==, <, >, etc.)
Pointer Arithmetic (continued)

Figure 12.15 The `nums` array in memory
Figure 12.16 Increments are scaled when used with pointers
Pointer Arithmetic (continued)

• Increment and decrement operators can be applied as both prefix and postfix operators

*ptNum++ // use the pointer and then increment it
*++ptNum  // increment the pointer before using it
*ptNum--  // use the pointer and then decrement it
*--ptNum  // decrement the pointer before using it

• Of the four possible forms, *ptNum++ is most common
  – Allows accessing each array element as the address is “marched along” from starting address to address of last array element
#include <iostream>
using namespace std;

int main()
{
    const int NUMS = 5;

    int nums[NUMS] = {16, 54, 7, 43, -5};
    int i, total = 0, *nPt;

    nPt = nums;  // store address of nums[0] in nPt
    for (i = 0; i < NUMS; i++)
        total = total + *nPt++;

    cout << "The total of the array elements is " << total << endl;

    return 0;
}
Pointer Arithmetic (continued)

Program 12.9

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

int main()
{
    const int NUMS = 5;

    int nums[NUMS] = {16, 54, 7, 43, -5};
    int total = 0, *nPt;

    nPt = nums;  // store address of nums[0] in nPt
    while (nPt < nums + NUMS)
        total += *nPt++;

    cout << "The total of the array elements is " << total << endl;

    return 0;
}
```
Pointer Initialization

• Pointers can be initialized with they are declared
  ```
  int *ptNum = &miles;
  ```

• Pointers to arrays can also be initialized when they are declared
  ```
  double *zing = &volts[0];
  ```
Passing Addresses

• Reference pointers can be used to pass addresses through reference parameters
  – Implied use of an address

• Pointers can be used explicitly to pass addresses with references
  – Explicitly passing references with the address operator is called **pass by reference**
  – Called function can reference, or access, variables in the calling function by using the passed addresses
Passing Addresses (continued)

Variable name: `firstnum`
Variable address: an address

A value

Variable name: `secnum`
Variable address: an address

A value

`swap(&firstnum, &secnum)`

**Figure 12.17** Explicitly passing addresses to `swap()`
Program 12.10

#include <iostream>
using namespace std;

void swap(double *, double *); // function prototype

int main()
{
    double firstnum = 20.5, secnum = 6.25;

    swap(&firstnum, &secnum); // call swap

    return 0;
}

// this function illustrates passing pointer arguments
void swap(double *nm1Addr, double *nm2Addr)
{
    cout << "The number whose address is in nm1Addr is "
         << *nm1Addr << endl;
    cout << "The number whose address is in nm2Addr is "
         << *nm2Addr << endl;

    return;
}
Passing Addresses (continued)

Figure 12.18 Storing addresses in parameters
Figure 12.19 Indirectly storing `firstnum`’s value
Passing Addresses (continued)

Figure 12.20 Indirectly changing `firstnum`’s value
Passing Addresses (continued)

Figure 12.21 Indirectly changing \texttt{secnum}'s value
Passing Addresses (continued)

Program 12.11

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

void swap(double *, double *);       // function prototype

int main()
{
    double firstnum = 20.5, secnum = 6.25;

    cout << "The value stored in firstnum is: " << firstnum << endl;
    cout << "The value stored in secnum is: " << secnum << "\n\n";
}
```

C++ for Engineers and Scientists, Third Edition 47
swap(&firstnum, &secnum); // call swap

cout << "The value stored in firstnum is now: ";
    << firstnum << endl;
cout << "The value stored in secnum is now: ";
    << secnum << endl;

return 0;
}

// this function swaps the values in its two arguments
void swap(double *nm1Addr, double *nm2Addr)
{
    double temp;

    temp = *nm1Addr; // save firstnum's value
    *nm1Addr = *nm2Addr; // move secnum's value into firstnum
    *nm2Addr = temp; // change secnum's value

    return;
}
Passing Arrays

• When an array is passed to a function, its address is the only item actually passed
  – “Address” means the address of the first location use to store the array
  – First location is always element zero of the array
Passing Arrays (continued)

Figure 12.22 Array’s address is address of the first location reserved for the array.
Passing Arrays (continued)

Figure 12.23 Pointing to different elements
Advanced Pointer Notation

- You can access multidimensional arrays by using pointer notation
  - Notation becomes more cryptic as array dimensions increase
- Sample declaration:

```cpp
int nums[0][1] = {{16, 18, 20}, {25, 26, 27}};
```
Advanced Pointer Notation (continued)

Figure 12.24  Storage of the nums array and associated pointer constants
Advanced Pointer Notation (continued)

• Ways to view two-dimensional arrays
  − As an array of rows
    • `nums[0]` is address of first element in the first row
    • Variable pointed to by `nums[0]` is `nums[0][0]`

• The following notations are equivalent:

<table>
<thead>
<tr>
<th>Pointer Notation</th>
<th>Subscript Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*nums[0]</code></td>
<td><code>nums[0][0]</code></td>
<td>16</td>
</tr>
<tr>
<td><code>*(nums[0] + 1)</code></td>
<td><code>nums[0][1]</code></td>
<td>18</td>
</tr>
<tr>
<td><code>*(nums[0] + 2)</code></td>
<td><code>nums[0][2]</code></td>
<td>20</td>
</tr>
<tr>
<td><code>*nums[1]</code></td>
<td><code>nums[1][0]</code></td>
<td>25</td>
</tr>
<tr>
<td><code>*(nums[1] + 1)</code></td>
<td><code>nums[1][1]</code></td>
<td>26</td>
</tr>
<tr>
<td><code>*(nums[1] + 2)</code></td>
<td><code>nums[1][2]</code></td>
<td>27</td>
</tr>
</tbody>
</table>
Advanced Pointer Notation (continued)

- You can replace $\text{nums}[0]$ and $\text{nums}[1]$ with pointer notations
  - $*\text{nums}$ is $\text{nums}[0]$
  - $*(\text{nums} + 1)$ is $\text{nums}[1]$

<table>
<thead>
<tr>
<th>Pointer Notation</th>
<th>Subscript Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(*\text{nums})$</td>
<td>$\text{nums}[0][0]$</td>
<td>16</td>
</tr>
<tr>
<td>$(*\text{nums} + 1)$</td>
<td>$\text{nums}[0][1]$</td>
<td>18</td>
</tr>
<tr>
<td>$(*\text{nums} + 2)$</td>
<td>$\text{nums}[0][2]$</td>
<td>20</td>
</tr>
<tr>
<td>$(*\text{((nums} + 1))$</td>
<td>$\text{nums}[1][0]$</td>
<td>25</td>
</tr>
<tr>
<td>$(*\text{((nums} + 1) + 1)$</td>
<td>$\text{nums}[1][1]$</td>
<td>26</td>
</tr>
<tr>
<td>$(*\text{((nums} + 1) + 2)$</td>
<td>$\text{nums}[1][2]$</td>
<td>27</td>
</tr>
</tbody>
</table>
Advanced Pointer Notation (continued)

• Same notation applies when a two-dimensional array is passed to function

• The \texttt{calc} function requires two dimensional array \texttt{nums} as a parameter
  – Suitable declarations
    • \texttt{calc(int pt[2][3])}
    • \texttt{calc(int pt[][3])}
    • \texttt{calc(int (*pt)[3])}
      – Parenthesis are required
      – Without parenthesis would evaluate to \texttt{int *pt[3]}, array of three pointers to integers
Advanced Pointer Notation (continued)

- The following is a table of equivalent notations for accessing `pt` from within `calc`

<table>
<thead>
<tr>
<th>Pointer Notation</th>
<th>Subscript Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(*pt)</code></td>
<td><code>pt[0][0]</code></td>
<td>16</td>
</tr>
<tr>
<td><code>(*(pt+1))</code></td>
<td><code>pt[0][1]</code></td>
<td>18</td>
</tr>
<tr>
<td><code>(*(pt+2))</code></td>
<td><code>pt[0][2]</code></td>
<td>20</td>
</tr>
<tr>
<td><code>(*(pt+1)[0])</code></td>
<td><code>pt[1][0]</code></td>
<td>25</td>
</tr>
<tr>
<td><code>(*(pt+1)[1])</code></td>
<td><code>pt[1][1]</code></td>
<td>26</td>
</tr>
<tr>
<td><code>(*(pt+1)[2])</code></td>
<td><code>pt[1][2]</code></td>
<td>27</td>
</tr>
</tbody>
</table>
Common Programming Errors

• Attempting to store address in variable not declared as pointer
• Using pointer to access nonexistent array elements
• Forgetting to use bracket set, [], after delete operator
• Incorrectly applying address and indirection operators
• Taking addresses of pointer constants
Common Programming Errors (continued)

- Taking addresses of a reference argument, reference variable, or register variable
- Initialized pointer variables incorrectly
- Becoming confused about whether a variable *contains* an address or *is* an address
Common Programming Errors (continued)

• Although a pointer constant is synonymous with an address, it’s useful to treat pointer constants as pointer variables with two restrictions:
  – Address of a pointer constant can’t be taken
  – Address “contained in” the pointer can’t be altered

• Except for these restrictions pointer constants and pointer variables can be used almost interchangeably
Common Programming Errors
(continued)

• When an address is required, any of the following can be used:
  – A pointer variable name
  – A pointer argument name
  – A pointer constant name
  – A non-pointer variable name preceded by the address operator
  – A non-pointer variable argument name preceded by the address operator
Chapter Summary

• Every variable has a data type, an address, and a value
• In C++, obtain the address of variable by using the address operator, &
• A pointer is a variable used to store address of another variable
  – Must be declared
  – Use indirection operator, *, to declare the pointer variable and access the variable whose address is stored in pointer
Chapter Summary (continued)

• Array name is a pointer constant
  – Value of the pointer constant is address of first element in array

• Any access to an array element with subscript notation can always be replaced with pointer notation

• Arrays can be created dynamically as program is executing

• Arrays are passed to functions as addresses
  – Called function always receives direct access to the originally declared array elements
Chapter Summary (continued)

• When a one-dimensional array is passed to a function, the function’s parameter declaration can be an array declaration or a pointer declaration
  – The following are equivalent:
    • double a[];
    • double *a;

• Pointers can be incremented, decremented, compared, and assigned
  – Numbers added to or subtracted from a pointer are scaled automatically
  – Scale factor is number of bytes required to store the pointer data type