Chapter 9 Pointers

9.1 Getting the Address of a Variable

Concept:

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

- Every variable is stored in a section of memory.
- Each byte of memory has a unique address.
- A variable's address refers to the address of the first byte allocated for it.

Suppose the following variables are defined in a program:

```
char letter;
short number;
float amount;
```

Figure 9-1 illustrates how they might be arranged in memory and shows their addresses.

In Figure 9-1, the variable `letter` is shown at address 1200, `number` is at address 1201, and `amount` is at address 1203.

Note:

The addresses of the variables shown in Figure 9-1 are arbitrary values used only for illustration purposes.

- The address operator (`&`), when placed before a variable name, returns that variable's address.
- For example, `&amount` returns the address of the `amount` variable.
- You can display a variable's address using `cout`, as shown below:

```
cout << &amount;
```

Note:

Do not confuse the address operator with the `&` symbol used when defining a reference variable.

Program 9-1 demonstrates using the address operator.

Program 9-1

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

int main()
{
	int x = 25;

	cout << "The address of x is " << &x << endl;
	cout << "The size of x is " << sizeof(x) << " bytes\n";
	cout << "The value in x is " << x << endl;
	return 0;
}
```

Program Output

Note:

The address of the variable `x` is displayed in hexadecimal. This is the way addresses are normally shown in C++.

9.2 Pointer Variables

Concept:

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

- A pointer is a special variable that holds a memory address.

- It “points” to a piece of data stored elsewhere in memory.
- Passing an array as an argument to a function is an example of using its address. The function parameter then acts as a pointer to the original array.
- Reference variables are another mechanism that works with addresses. They act as aliases for other variables.
- When an argument is passed by reference, the reference parameter “points” to the original variable.
- C++ handles the address mechanics automatically for reference variables.
- Pointers are a lower-level mechanism than references, requiring you to manage addresses manually.
- Pointers are essential for operations like dynamic memory allocation and are widely used in algorithms and object-oriented programming.

Creating and Using Pointer Variables

- To define a pointer, use an asterisk (`*`) before the variable name. For example, `int *ptr;` declares `ptr` as a pointer to an `int`.
- The data type (`int` in this case) specifies the type of data the pointer can point to, not the type of the pointer variable itself. Pointers only hold addresses.
- The asterisk can be placed next to the type name (`int* ptr;`) or the variable name (`int *ptr;`). Both are correct.
- It is good practice to initialize pointer variables. Uninitialized pointers can cause errors by affecting unknown memory locations.
- In C++ 11 and later, initialize pointers with `nullptr`, which represents address 0. A pointer set to address 0 is called a *null pointer*.

```
int *ptr = nullptr;
```

Note:

If you are using an older compiler that does not support the C++ 11 standard, you should initialize pointers with the integer 0, or the value `NULL`. The value `NULL` is defined in the `<iostream>` header file (as well as other header files) to represent the value 0.

Program 9-2 shows how to store and print an address using a pointer.

Program 9-2

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

int main()
{
        int x = 25;
        int *ptr = nullptr;

         ptr = &x;
         cout << "The value in x is " << x << endl;
         cout << "The address of x is " << ptr << endl;
         return 0;
}
```

Program Output

- In the program above, the statement `ptr = &x;` stores the memory address of `x` in the pointer `ptr`.
- The primary benefit of pointers is the ability to indirectly access and modify the data they point to.
- This is done using the *indirection operator* (`*`). Placing `*` in front of a pointer variable's name *dereferences* it, allowing you to work with the value it points to.

Program 9-3 demonstrates the indirection operator.

Program 9-3

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

```

int main()
{
        int x = 25;
        int *ptr = nullptr;

        ptr = &x;

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

        *ptr = 100;

    cout << "Once again, here is the value in x:\n";
    cout << x << endl;
    cout << *ptr << endl;
    return 0;
}
```

Program Output

- The statement `cout << *ptr << endl;` displays the value that `ptr` points to (the contents of `x`).
- The statement `*ptr = 100;` assigns 100 to the location pointed to by `ptr`, effectively changing the value of `x`.

Program 9-4 shows that a pointer can be changed to point to different variables.

Program 9-4

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

int main()
{
    int x = 25, y = 50, z = 75;
    int *ptr = nullptr;

    cout << "Here are the values of x, y, and z:\n";
    cout << x << " " << y << " " << z << endl;


    ptr = &x;
    *ptr += 100;

    ptr = &y;
    *ptr += 100;

    ptr = &z;
    *ptr += 100;

    cout << "Once again, here are the values of x, y, and z:\n";
    cout << x << " " << y << " " << z << endl;
    return 0;
}
```

Program Output

- In the program above, the `ptr` variable is first made to point to `x`, then `y`, and finally `z`, modifying each variable's value in turn.

Note:

So far, you've seen three different uses of the asterisk in C++:

- As the multiplication operator, in statements such as

  ```
  distance = speed * time;
  ```

- In the definition of a pointer variable, such as

  ```
  int *ptr = nullptr;
  ```

- As the indirection operator, in statements such as

```
*ptr = 100;
```

9.3 The Relationship between Arrays and Pointers

Concept:

Array names can be used as constant pointers, and pointers can be used as array names.

- An array name without brackets represents the starting address of the array. Essentially, an array name is a pointer.

🗐 **Program 9-5** demonstrates dereferencing an array name with the `*` operator.

🗐 **Program 9-5**

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

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

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

💻 **Program Output**

- Because `numbers` acts as a pointer to the start of the array, `*numbers` retrieves the first element.
- When you add an integer to a pointer, C++ automatically scales the addition by the size of the data type the pointer references.
- Adding `1` to an `int` pointer moves it forward by `sizeof(int)` bytes to the next element.

- Therefore, `*(numbers + 1)` accesses the second element, `*(numbers + 2)` the third, and so on.

Note:

The parentheses are critical when adding values to pointers. The `*` operator has precedence over the + operator, so the expression `*number + 1` is not equivalent to `*(number + 1)`. `*number + 1` adds one to the contents of the first element of the array, while `*(number + 1)` adds one to the address in `number`, then dereferences it.

Program 9-6 uses pointer notation to access all elements of an array.

Program 9-6

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

int main()
{
    const int SIZE = 5;
    int numbers[SIZE];
    int count;

    cout << "Enter " << SIZE << " numbers: ";
    for (count = 0; count < SIZE; count++)
        cin >> *(numbers + count);

    cout << "Here are the numbers you entered:\n";
    for (count = 0; count < SIZE; count++)
        cout << *(numbers + count)<< " ";
    cout << endl;
    return 0;
}
```

Program Output

- The key relationship to remember is: `array[index]` is equivalent to `*(array + index)`.

Warning!

Remember that C++ performs no bounds checking with arrays. When stepping through an array with a pointer, it's possible to give the pointer an address outside of the array.

🗐 **Program 9-7** shows that subscript notation can be used with pointers, and pointer notation can be used with array names.

🗐 **Program 9-7**

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

int main()
{
   const int NUM_COINS = 5;
   double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
   double *doublePtr;
   int count;

   doublePtr = coins;

   cout << "Here are the values in the coins array:\n";
   for (count = 0; count < NUM_COINS; count++)
      cout << doublePtr[count] << " ";

   cout << "\nAnd here they are again:\n";
   for (count = 0; count < NUM_COINS; count++)
      cout << *(coins + count) << " ";
   cout << endl;
   return 0;
}
```

💻 **Program Output**

- The address operator (`&`) is not needed when assigning an array's address to a pointer because the array name is already an address.
- However, you can use `&` to get the address of an individual array element, like `&numbers[1]`.

🗐 **Program 9-8** uses this technique.

🗐 **Program 9-8**

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

int main()
{
     const int NUM_COINS = 5;
     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
     double *doublePtr = nullptr;
     int count;

    cout << "Here are the values in the coins array:\n";
    for (count = 0; count < NUM_COINS; count++)
    {
       doublePtr = &coins[count];

       cout << *doublePtr << " ";
    }
    cout << endl;
    return 0;
}
```

Program Output

- A key difference between array names and pointer variables is that you cannot change the address an array name points to. Array names are *pointer constants*.

9.4 Pointer Arithmetic

Concept:

Some mathematical operations may be performed on pointers.

- Pointer variables can be modified using addition and subtraction.

Program 9-9 shows a pointer being incremented and decremented to step through an array.

Program 9-9

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

int main()
{
    const int SIZE = 8;
    int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *numPtr = nullptr;
    int count;

    numPtr = set;

    cout << "The numbers in set are:\n";
    for (count = 0; count < SIZE; count++)
    {
        cout << *numPtr << " ";
        numPtr++;
    }

    cout << "\nThe numbers in set backward are:\n";
    for (count = 0; count < SIZE; count++)
    {
        numPtr--;
        cout << *numPtr << " ";
    }
    return 0;
}
```

Program Output

Note:

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

- Not all arithmetic operations can be used with pointers. You cannot multiply or divide a pointer.
- Allowable operations include:
  - Incrementing (`++`) and decrementing (`--`).
  - Adding or subtracting an integer (`+`, `-`, `+=`, `-=`).
  - Subtracting one pointer from another.

9.5 Initializing Pointers

Concept:

Pointers may be initialized with the address of an existing object.

- When initializing a pointer, the address must be of a matching data type. For instance, an `int` pointer must be initialized with the address of an `int` variable or an array of `int`s.
- An `int` pointer cannot be initialized with the address of a `float`.
- Pointers can be defined and initialized in the same statement as other variables.

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

```
double readings[50], *marker = readings;
```

- A pointer can only be initialized with the address of an object that has already been defined. The following is illegal:

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

Checkpoint

9.1 Write a statement that displays the address of the variable `count`.

9.2 Write the definition statement for a variable `fltPtr`. The variable should be a pointer to a `float`.

9.3 List three uses of the `*` symbol in C++.

9.4 What is the output of the following code?

```
int x = 50, y = 60, z = 70;
int *ptr = nullptr;

cout << x << " " << y << " " << z << endl;
ptr = &x;
*ptr *= 10;
ptr = &y;
*ptr *= 5;
ptr = &z;
*ptr *= 2;
cout << x << " " << y << " " << z << endl;
```

9.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)
   cout << arr[x] << endl;
```

9.6 Assume `ptr` is a pointer to an `int` and holds the address 12000. On a system with 4-byte integers, what address will be in `ptr` after the following statement?

```
ptr += 10;
```

9.7 Assume `pint` is a pointer variable. Is each of the following statements valid or invalid? If any is invalid, why?

`pint++;`

`--pint;`

`pint /= 2;`

`pint *= 4;`

`pint += x; // Assume x is an int.`

9.8 Is each of the following definitions valid or invalid? If any is invalid, why?

`int ivar;`

`int *iptr = &ivar;`

`int ivar, *iptr = &ivar;`

`float fvar;`

`int *iptr = &fvar;`

`int nums[50], *iptr = nums;`

`int *iptr = &ivar;`

`int ivar;`

9.6 Comparing Pointers

Concept:

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

- Pointers can be compared using relational operators (`>`, `<`, `==`, `!=`, `>=`, `<=`).
- This works because array elements are stored in consecutive memory locations, so the address of a later element is greater than the address of an earlier one.

Note:

Comparing two pointers is not the same as comparing the values the two pointers point to. For example, the following `if` statement compares the addresses stored in the pointer variables `ptr1` and `ptr2`:

```
if (ptr1 < ptr2)
```

The following statement, however, compares the values that `ptr1` and `ptr2` point to:

```
if (*ptr1 < *ptr2)
```

- Comparing pointer addresses is a useful way to ensure a pointer stays within the boundaries of an array.

🗐 **Program 9-10** demonstrates this by using pointer comparison in a `while` loop to traverse an array.

🗐 Program 9-10

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

int main()
{
     const int SIZE = 8;
     int numbers[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
     int *ptr = numbers;

     cout << "The numbers are:\n";
     cout << *ptr << " ";
   while (ptr < &numbers[SIZE - 1])
   {
      ptr++;
      cout << *ptr << " ";
   }

   cout << "\nThe numbers backward are:\n";
   cout << *ptr << " ";
   while (ptr > numbers)
   {
      ptr--;
      cout << *ptr << " ";
   }
   return 0;
}
```

💻 Program Output

9.7 Pointers as Function Parameters

Concept:

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

- A pointer parameter, like a reference parameter, allows a function to access and modify the original argument variable. This is another method for passing an argument by reference.
- While reference variables are often easier to use, pointers are necessary for certain tasks (like manipulating C-style strings) and are used throughout the C++ library.

Here is a function that uses a pointer parameter:

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

- In the function, the pointer `val` is dereferenced. The statement `*val *= 2;` multiplies the original variable (pointed to by `val`) by 2.
- When calling the function, you must pass the *address* of the variable you want to modify, using the address-of operator (`&`).

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

Program 9-11 illustrates this concept.

Program 9-11

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

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

int main()
{
    int number;
    getNumber(&number);

    doubleValue(&number);

    cout << "That value doubled is " << number << endl;
    return 0;
}


void getNumber(int *input)
{
    cout << "Enter an integer number: ";
    cin >> *input;
}


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

Program Output

- Function prototypes must use an asterisk (`*`) to indicate a pointer parameter, such as `void getNumber(int *);`.
- In the `getNumber` function, the statement `cin >> *input;` uses the indirection operator to store the user's input in the variable that `input` points to.

Warning!

It's critical that the indirection operator be used in the statement above. Without it, `cin` would store the value entered by the user in `input`, as if the value were an address. If this happens, `input` will no longer point to the `number` variable in function `main`. Subsequent use of the pointer will result in erroneous, if not disastrous, results.

- Pointers can also be used as parameters to accept the address of an array.
- Inside the function, you can work with the array using either subscript notation or pointer notation.

Program 9-12 demonstrates passing an array to a pointer parameter.

Program 9-12

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

void getSales(double *, int);
double totalSales(double *, int);

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

    getSales(sales, QTRS);

    cout << fixed << showpoint << setprecision(2);

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

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

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

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

```
        return sum;
    }
```

Program Output

- In `getSales`, subscript notation `arr[count]` is used with the pointer parameter.
- In `totalSales`, pointer notation `*arr` and `arr++` is used.

Note:

The two previous statements could be combined into the following statement:

```
sum += *arr++;
```

The * operator will first dereference `arr`, then the ++ operator will increment the address in `arr`.

Pointers to Constants

- To pass the address of a `const` variable or array to a function, the function's parameter must be declared as a pointer to `const`.
- For example, if `payRates` is a `const` array, a function to display it would have this header:

```
void displayPayRates(const double *rates, int size)
{
    cout << setprecision(2) << fixed << showpoint;
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1)
             << " is $" << *(rates + count) << endl;
    }
}
```

- In this case, `const` applies to the data that `rates` points to, not to the `rates` pointer itself. This prevents the function from modifying the original `const` data.

Passing a Nonconstant Argument into a Pointer to a Constant

- A pointer-to-`const` parameter can accept the address of both `const` and non-`const` arguments.
- This allows a function to guarantee it won't modify an argument, even if the original argument was not `const`.

Program 9-13 demonstrates this.

Program 9-13

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

void displayValues(const int *, int);

int main()
{
    const int SIZE = 6;

    const int array1[SIZE] = { 1, 2, 3, 4, 5, 6 };

    int array2[SIZE] = { 2, 4, 6, 8, 10, 12 };

    displayValues(array1, SIZE);

    displayValues(array2, SIZE);
    return 0;
}


void displayValues(const int *numbers, int size)
{
    for (int count = 0; count < size; count++)
    {
        cout << *(numbers + count) << " ";
    }
    cout << endl;
}
```

Program Output

```
2 3 4 5 6
4 6 8 10 12
```

Note:

When you are writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to `const`. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and nonconstant arguments.

Constant Pointers

- **Pointer to `const`**: The data that the pointer points to cannot be changed, but the pointer itself *can* be changed to point to a different address.
- `const` **pointer**: The pointer itself is constant and cannot be changed to point to a different address after it is initialized. The data it points to *can* be changed (unless it's pointing to `const` data).

A `const` pointer is declared by placing `const` after the asterisk.

```
int value = 22;
int * const ptr = &value;
```

- This means `ptr` is a constant pointer. It must be initialized with an address and cannot be reassigned.
- While the pointer `ptr` cannot be changed, the value it points to (`value`) can be changed through `*ptr` because `value` is not `const`.
- When used as a function parameter, a `const` pointer is initialized with the argument's address and cannot be changed within the function. However, the data it points to can be modified.

```
void setToZero(int * const ptr)
{
    *ptr = 0;
}
```

Constant Pointers to Constants

- You can also have a constant pointer to constant data, which means neither the pointer nor the data it points to can be changed.
- This is declared with `const` both before the type and after the asterisk.

```
int value = 22;
const int * const ptr = &value;
```

- Here, `ptr` is a `const` pointer to a `const int`.
- In this case, you cannot change where `ptr` points, and you cannot use `ptr` to change the contents of `value`.

9.8 Dynamic Memory Allocation

Concept:

Variables may be created and destroyed while a program is running.

- When you don't know at compile time how many variables you will need, you can use dynamic memory allocation.
- This allows a program to request memory from the operating system while it is running. This is only possible through pointers.
- The `new` operator is used to request memory. It returns the starting address of the allocated block, which must be stored in a pointer.

```
int *iptr = nullptr;
iptr = new int;
```

- The pointer is then used to access this new memory location.

```
*iptr = 25;
```

- The `new` operator is most practically used to create arrays of a size determined at runtime.

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

- If `new` cannot allocate the requested memory, it throws an exception, and the program terminates.
- When you are finished with dynamically allocated memory, you must free it using the `delete` operator. Failure to do so results in a *memory leak*.
- Use `delete` for a single variable:

```
delete iptr;
```

- Use `delete []` for an array:

```
delete [] iptr;
```

Warning!

Only use pointers with `delete` that were previously used with `new`. If you use a pointer with `delete` that does not reference dynamically allocated memory, unexpected problems could result!

Program 9-14 demonstrates dynamic allocation of an array.

Program 9-14

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

int main()
{
    double *sales = nullptr,
           total = 0.0,
           average;
    int numDays,
        count;

    cout << "How many days of sales amounts do you wish ";
```

```cpp
    cout << "to process? ";
    cin >> numDays;

    sales = new double[numDays];

    cout << "Enter the sales amounts below.\n";
    for (count = 0; count < numDays; count++)
    {
        cout << "Day " << (count + 1) << ": ";
        cin >> sales[count];
    }

    for (count = 0; count < numDays; count++)
    {
        total += sales[count];
    }

    average = total / numDays;

    cout << fixed << showpoint << setprecision(2);
    cout << "\n\nTotal Sales: $" << total << endl;
    cout << "Average Sales: $" << average << endl;

    delete [] sales;
    sales = nullptr;

    return 0;
}
```

Program Output

- It is good practice to set a pointer to `nullptr` after deleting its memory to prevent accidental use. The `delete` operator has no effect on a null pointer.

9.9 Returning Pointers from Functions

Concept:

Functions can return pointers, but you must be sure the item the pointer references still exists.

- Functions can be written to return a pointer. The return type in the function header must indicate this, such as `char *`.
- **Warning**: A function should never return a pointer to a local variable. The local variable is destroyed when the function terminates, leaving the returned pointer pointing to an invalid memory location.
- You should only return a pointer from a function if it is:
  - A pointer to an item that was passed into the function as an argument.
  - A pointer to a dynamically allocated chunk of memory.

Here is an example of an acceptable function that dynamically allocates memory and returns a pointer to it:

```
string *getFullName()
{
   string *fullName = new string[3];

   cout << "Enter your first name: ";
   getline(cin, fullName[0]);
   cout << "Enter your middle name: ";
   getline(cin, fullName[1]);
   cout << "Enter your last name: ";
   getline(cin, fullName[2]);
   return fullName;
}
```

Program 9-15 shows a function that returns a pointer to a dynamically allocated array of random numbers.

Program 9-15

```
#include <iostream>
#include <cstdlib>
```

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

int *getRandomNumbers(int);

int main()
{
    int *numbers = nullptr;

    numbers = getRandomNumbers(5);

    for (int count = 0; count < 5; count++)
        cout << numbers[count] << endl;

    delete [] numbers;
    numbers = nullptr;
    return 0;
}


int *getRandomNumbers(int num)
{
    int *arr = nullptr;

    if (num <= 0)
        return nullptr;

    arr = new int[num];

    srand( time(0) );

    for (int count = 0; count < num; count++)
        arr[count] = rand();

    return arr;
}
```

Program Output

In the Spotlight:

🗐 **Program 9-16** demonstrates a function that accepts an array, dynamically allocates a new array of the same size, copies the elements, and returns a pointer to the new, duplicate array.

🗐 **Program 9-16**

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

int *duplicateArray(const int *, int);
void displayArray(const int[], int);

int main()
{
    const int SIZE1 = 5, SIZE2 = 7, SIZE3 = 10;

    int array1[SIZE1] = { 100, 200, 300, 400, 500 };
    int array2[SIZE2] = { 10, 20, 30, 40, 50, 60, 70 };
    int array3[SIZE3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    int *dup1 = nullptr, *dup2 = nullptr, *dup3 = nullptr;

    dup1 = duplicateArray(array1, SIZE1);
    dup2 = duplicateArray(array2, SIZE2);
    dup3 = duplicateArray(array3, SIZE3);

    cout << "Here are the original array contents:\n";
    displayArray(array1, SIZE1);
    displayArray(array2, SIZE2);
    displayArray(array3, SIZE3);

     cout << "\nHere are the duplicate arrays: \n";
     displayArray(dup1, SIZE1);
     displayArray(dup2, SIZE2);
     displayArray(dup3, SIZE3);

     delete [] dup1;
     delete [] dup2;
     delete [] dup3;
     dup1 = nullptr;
     dup2 = nullptr;
     dup3 = nullptr;
     return 0;
}

int *duplicateArray(const int *arr, int size)
{
   int *newArray = nullptr;

   if (size <= 0)
      return nullptr;

   newArray = new int[size];

   for (int index = 0; index < size; index++)
      newArray[index] = arr[index];

   return newArray;
}


void displayArray(const int arr[], int size)
{
   for (int index = 0; index < size; index++)
      cout << arr[index] << " ";
   cout << endl;
}
```

💻 **Program Output**

Checkpoint

9.9 Assuming `arr` is an array of `int`s, will each of the following program segments display “True” or “False”?

```
if (arr < &arr[1])
      cout << "True";
```

```
   else
      cout << "False";
```

```
if (&arr[4] < &arr[1])
      cout << "True";
   else
      cout << "False";
```

```
if (arr != &arr[2])
      cout << "True";
   else
      cout << "False";
```

```
if (arr != &arr[0])
      cout << "True";
   else
      cout << "False";
```

9.10 Give an example of the proper way to call the following function:

```
void makeNegative(int *val)
{
   if (*val > 0)
   *val = -(*val);
}
```

9.11 Complete the following program skeleton. When finished, the program will ask the user for a length (in inches), then convert that value to centimeters, and display the result. You are to write the function `convert`. (*Note:* 1 inch = 2.54 cm. Do not modify function `main`.)

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


int main()
{
   double measurement;

   cout << "Enter a length in inches, and I will convert\n";
   cout << "it to centimeters: ";
```

```
    cin >> measurement;
    convert(&measurement);
    cout << fixed << setprecision(4);
    cout << "Value in centimeters: " << measurement << endl;
    return 0;
}
```

9.12 Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function `processArray` in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the `processArray` function?

`void processArray(const int *arr, int size)`

`void processArray(int * const arr, int size)`

9.13 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an integer variable and store its address in `ip`. Write a statement that will free the memory allocated in the statement you wrote above.

9.14 Assume `ip` is a pointer to an `int`. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in `ip`. Write a statement that will free the memory allocated in the statement you just wrote.

9.15 What is a null pointer?

9.16 Give an example of a function that correctly returns a pointer.

9.17 Give an example of a function that incorrectly returns a pointer.

9.10 Using Smart Pointers to Avoid Memory Leaks

Concept:

C++ 11 introduces smart pointers, objects that work like pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used.

- C++ 11 introduced *smart pointers* to automatically manage dynamically allocated memory and help prevent memory leaks.
- A smart pointer automatically deletes the memory it points to when that memory is no longer being used.
- C++ 11 provides three types of smart pointers: `unique_ptr`, `shared_ptr`, and `weak_ptr`.

Table 9-1 Smart Pointer Types

SMART POINTER TYPE	DESCRIPTION
`unique_ptr`	A `unique_ptr` is the sole owner of a piece of dynamically allocated memory. No two `unique_ptr`s can point to the same piece of memory. When a `unique_ptr` goes out of scope, it automatically deallocates the piece of memory that it points to.
`shared_ptr`	A `shared_ptr` can share ownership of a piece of dynamically allocated memory. Multiple pointers of the `shared_ptr` type can point to the same piece of memory. The memory is deallocated when the last `shared_ptr` that is pointing to it is destroyed.
`weak_ptr`	A `weak_ptr` does not own the memory it points to, and cannot be used to access the memory's contents. It is used in special situations where the memory pointed to by a `shared_ptr` must be referenced without increasing the number of `shared_ptr`s that own it.

- To use smart pointers, you must include the `<memory>` header file.
- The syntax for defining a `unique_ptr` is:

```
unique_ptr<int> ptr( new int );
```

- `<int>` indicates the pointer type. `ptr` is the name. `new int` allocates the memory.
- Once defined, a `unique_ptr` can be dereferenced with `*` just like a regular pointer.

🗐 **Program 9-17** demonstrates a `unique_ptr`.

🗐 **Program 9-17**

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

int main()
{
    unique_ptr<int> ptr( new int );

    *ptr = 99;

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

💻 **Program Output**

- Notice there is no `delete` statement. The smart pointer automatically frees the memory when `ptr` goes out of scope.
- To use a `unique_ptr` for a dynamically allocated array, use `<int[]>` as the type.

```
const int SIZE = 100;
unique_ptr<int[]> ptr( new int[SIZE] );
```

- You can then use the `[]` operator to access array elements.

🗐 **Program 9-18** gives a more complete demonstration.

🗐 **Program 9-18**

```
#include <iostream>
#include <memory>
```

```cpp
using namespace std;

int main()
{
    int max;

    cout << "How many numbers do you want to enter? ";
    cin >> max;

    unique_ptr<int[]> ptr( new int[max]);

    for (int index = 0; index < max; index++)
    {
        cout << "Enter an integer number: ";
        cin >> ptr[index];
    }

    cout << "Here are the values you entered:\n";
    for (int index = 0; index < max; index++)
        cout << ptr[index] << endl;

    return 0;
}
```

Program Output

- Smart pointers support dereferencing (`*` and `->`) but do not support pointer arithmetic (like `ptr++`).

9.11 Focus on Problem Solving and Program Design: A Case Study

Concept:

This case study demonstrates how an array of pointers can be used to display the contents of a second array in sorted order, without sorting the `second` array.

This case study involves writing a program for the United Cause charity to display a list of donations in both their original order and in ascending sorted order.

Variables

Table 9-2 Variables

VARIABLE	DESCRIPTION
`NUM_DONATIONS`	A constant integer initialized with the number of donations received from CK Graphics, Inc. This value will be used in the definition of the program's arrays.
`donations`	An array of integers containing the donation amounts.
`arrPtr`	An array of pointers to integers. This array has the same number of elements as the `donations` array. Each element of `arrPtr` will be initialized to point to an element of the `donations` array.

Programming Strategy

- The program uses two arrays: `donations` to store the original data, and `arrPtr` as an array of pointers.
- Each element of `arrPtr` is initialized to point to a corresponding element in the `donations` array.
- The `arrPtr` array is then sorted based on the donation values its elements point to. The `donations` array itself is not modified.

- After sorting `arrPtr`, dereferencing its elements in order provides access to the donation amounts in ascending order.
- This technique allows viewing the data in a sorted manner while preserving the original order in the `donations` array.

Functions

Table 9-3 Functions

FUNCTION	DESCRIPTION
`main`	The program's `main` function. It calls the program's other functions.
`arrSelectSort`	Performs an ascending order selection sort on its parameter, `arr`, which is an array of pointers. Each element of `arr` points to an element of a second array. After the sort, `arr` will point to the elements of the second array in ascending order.
`showArray`	Displays the contents of its parameter, `arr`, which is an array of integers. This function is used to display the donations in their original order.
`showArrPtr`	Accepts an array of pointers to integers as an argument. Displays the contents of what each element of the array points to. This function is used to display the contents of the `donations` array in sorted order.

Function `main`

The pseudocode for `main` is as follows:

```
For count is set to the values 0 through the number of donations
    Set arrPtr[count] to the address of donations[count].
End For
Call arrSelectSort
Call showArrPtr
Call showArray
```

The `arrSelectSort` Function

This function is a modified selection sort that sorts an array of pointers based on the values they point to. The pseudocode is:

```
For startScan is set to the values 0 up to (but not including) the
          next-to-last subscript in arr
   Set index variable to startScan
   Set minIndex variable to startScan
   Set minElem pointer to arr[startScan]
   For index variable is set to the values from (startScan + 1) through
                 the last subscript in arr
       If *(arr[index]) is less than *minElem
          Set minElem to arr[index]
          Set minIndex to index
       End If
   End For
   Set arr[minIndex] to arr[startScan]
   Set arr[startScan] to minElem
End For
```

The `showArrPtr` Function

This function displays the values pointed to by the elements of the pointer array. Pseudocode:

```
For every element in the arr
    Dereference the element and display what it points to
End For.
```

The `showArray` Function

This function displays the original donation array. Pseudocode:

```
For every element in arr
    Display the element's contents
End For.
```

The Entire Program

Program 9-19 shows the full source code.

Program 9-19

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

void arrSelectSort(int *[], int);
void showArray(const int [], int);
void showArrPtr(int *[], int);

int main()
{
    const int NUM_DONATIONS = 15;

    int donations[NUM_DONATIONS] = { 5,  100, 5,  25, 10,
                                     5,  25,  5,  5,  100,
                                     10, 15,  10, 5,  10 };

    int *arrPtr[NUM_DONATIONS] = { nullptr, nullptr, nullptr, nullptr, null
                                   nullptr, nullptr, nullptr, nullptr, null
                                   nullptr, nullptr, nullptr, nullptr, null

    for (int count = 0; count < NUM_DONATIONS; count++)
        arrPtr[count] = &donations[count];

    arrSelectSort(arrPtr, NUM_DONATIONS);

    cout << "The donations, sorted in ascending order, are: \n";
    showArrPtr(arrPtr, NUM_DONATIONS);

    cout << "The donations, in their original order, are: \n";
    showArray(donations, NUM_DONATIONS);
    return 0;
}


void arrSelectSort(int *arr[], int size)
{
    int startScan, minIndex;
    int *minElem;

    for (startScan = 0; startScan < (size - 1); startScan++)
    {
        minIndex = startScan;
        minElem = arr[startScan];
        for(int index = startScan + 1; index < size; index++)
        {
            if (*(arr[index]) < *minElem)
            {
                minElem = arr[index];
                minIndex = index;
            }
        }
        arr[minIndex] = arr[startScan];
        arr[startScan] = minElem;
    }
}


void showArray(const int arr[], int size)
{
    for (int count = 0; count < size; count++)
        cout << arr[count] << " ";
    cout << endl;
}


void showArrPtr(int *arr[], int size)
{
    for (int count = 0; count < size; count++)
        cout << *(arr[count]) << " ";
    cout << endl;
}
```

Program Output

Review Questions and Exercises

Short Answer

1. What does the indirection operator do?
2. Look at the following code.

   ```
   int x = 7;
   int *iptr = &x;
   ```

   What will be displayed if you send the expression `*iptr` to `cout`? What happens if you send the expression `ptr` to `cout`?
3. So far you have learned three different uses for the `*` operator. What are they?
4. What math operations are allowed on pointers?
5. Assuming `ptr` is a pointer to an `int`, what happens when you add 4 to `ptr`?
6. Look at the following array definition.

   `int numbers[] = { 2, 4, 6, 8, 10 };`

   What will the following statement display?

   `cout << *(numbers + 3) << endl;`
7. What is the purpose of the `new` operator?
8. What happens when a program uses the `new` operator to allocate a block of memory, but the amount of requested memory isn't available? How do programs written with older compilers handle this?
9. What is the purpose of the `delete` operator?
10. Under what circumstances can you successfully return a pointer from a function?
11. What is the difference between a pointer to a constant and a constant pointer?
12. What are two advantages of declaring a pointer parameter as a constant pointer?

Fill-in-the-Blank

1. Each byte in memory is assigned a unique __________________.
2. The __________________ operator can be used to determine a variable's address.
3. __________________ variables are designed to hold addresses.

4. The _________________ operator can be used to work with the variable a pointer points to.
5. Array names can be used as _________________, and vice versa.
6. Creating variables while a program is running is called _________________.
7. The _________________ operator is used to dynamically allocate memory.
8. Under older compilers, if the `new` operator cannot allocate the amount of memory requested, it returns _________________.
9. A pointer that contains the address 0 is called a(n) _________________ pointer.
10. When a program is finished with a chunk of dynamically allocated memory, it should free it with the _________________ operator.
11. You should only use pointers with `delete` that were previously used with _________________.

Algorithm Workbench

1. Look at the following code:

   ```
   double value = 29.7;
   double *ptr = &value;
   ```

   Write a `cout` statement that uses the `ptr` variable to display the contents of the `value` variable.
2. Look at the following array definition:

   `int set[10];`

   Write a statement using pointer notation that stores the value 99 in `set[7];`
3. Write code that dynamically allocates an array of 20 integers, then uses a loop to allow the user to enter values for each element of the array.
4. Assume `tempNumbers` is a pointer that points to a dynamically allocated array. Write code that releases the memory used by the array.
5. Look at the following function definition:

```
void getNumber(int &n)
{
   cout << "Enter a number: ";
   cin >> n;
}
```

In this function, the parameter `n` is a reference variable. Rewrite the function so `n` is a pointer.

6. Write the definition of `ptr`, a pointer to a constant `int`.
7. Write the definition of `ptr`, a constant pointer to an `int`.

True or False

1. T F Each byte of memory is assigned a unique address.
2. T F The `*` operator is used to get the address of a variable.
3. T F Pointer variables are designed to hold addresses.
4. T F The `&` symbol is called the indirection operator.
5. T F The `&` operator dereferences a pointer.
6. T F When the indirection operator is used with a pointer variable, you are actually working with the value the pointer is pointing to.
7. T F Array names cannot be dereferenced with the indirection operator.
8. T F When you add a value to a pointer, you are actually adding that number times the size of the data type referenced by the pointer.
9. T F The address operator is not needed to assign an array's address to a pointer.
10. T F You can change the address that an array name points to.
11. T F Any mathematical operation, including multiplication and division, may be performed on a pointer.
12. T F Pointers may be compared using the relational operators.
13. T F When used as function parameters, reference variables are much easier to work with than pointers.

14. T F The `new` operator dynamically allocates memory.

15. T F A pointer variable that has not been initialized is called a null pointer.

16. T F The address 0 is generally considered unusable.

17. T F n using a pointer with the `delete` operator, it is not necessary for the pointer to have been previously used with the `new` operator.

Find the Error

Each of the following definitions and program segments has errors. Locate as many as you can.

1. `int ptr* = nullptr;`

2.
```
int x, *ptr = nullptr;
&x = ptr;
```

3.
```
int x, *ptr = nullptr;
*ptr = &x;
```

4.
```
int x, *ptr = nullptr;
ptr = &x;
ptr = 100;
cout << x << endl;
```

5.
```
int numbers[] = {10, 20, 30, 40, 50};
cout << "The third element in the array is ";
cout << *numbers + 3 << endl;
```

6.
```
int values[20], *iptr = nullptr;
iptr = values;
iptr *= 2;
```

7.
```
float level;
int fptr = &level;
```

8.
```
int *iptr = &ivalue;
int ivalue;
```

9.
```
void doubleVal(int val)
{
   *val *= 2;
}
```

10.
```
int *pint = nullptr;
new pint;
```

11.
```
int *pint = nullptr;
pint = new int;
if (pint == nullptr)
   *pint = 100;
else
   cout << "Memory allocation error\n";
```

12.
```
int *pint = nullptr;
pint = new int[100];

   .
   .

Code that processes the array.

   .
   .

delete pint;
```

13.
```
int *getNum()
{
     int wholeNum;
     cout << "Enter a number: ";
     cin >> wholeNum;
     return &wholeNum;
}
```

14.
```
const int arr[] = { 1, 2, 3 };
int *ptr = arr;
```

15.
```
void doSomething(int * const ptr)
{
     int localArray[] = { 1, 2, 3 };
     ptr = localArray;
}
```

Programming Challenges

1. Array Allocator

   Write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should return a pointer to the array.

2. Test Scores #1

   Write a program that dynamically allocates an array large enough to hold a user-defined number of test scores. Once all the scores are entered, the array should be passed to a function that sorts them in ascending order. Another function should be called that calculates the average score. The program should display the sorted list of scores and averages with appropriate headings. Use pointer notation rather than array notation whenever possible.

   Input Validation: Do not accept negative numbers for test scores.

3. Drop Lowest Score

   Modify Problem 2 above so the lowest test score is dropped. This score should not be included in the calculation of the average.

4. Test Scores #2

   Modify the program of Programming Challenge 2 (Test Scores #1) to allow the user to enter name–score pairs. For each student taking a test, the user types the student's name followed by the student's integer test score. Modify the sorting function so it takes an array holding the student names, and an array holding the student test scores. When the sorted list of scores is displayed, each student's name should be displayed along with his or her score. In stepping through the arrays, use pointers rather than array subscripts.

5. Pointer Rewrite

The following function uses reference variables as parameters. Rewrite the function so it uses pointers instead of reference variables, then demonstrate the function in a complete program.

VideoNote

Solving the Pointer Rewrite Problem

```
int doSomething(int &x, int &y)
{
     int temp = x;
     x = y * 10;
     y = temp * 10;
     return x + y;
}
```

6. Case Study Modification #1

   Modify Program 9-19 (the United Cause case study program) so it can be used with any set of donations. The program should dynamically allocate the `donations` array and ask the user to input its values.

7. Case Study Modification #2

   Modify Program 9-19 (the United Cause case study program) so the `arrptr` array is sorted in descending order instead of ascending order.

8. Mode Function

   In statistics, the *mode* of a set of values is the value that occurs most often or with the greatest frequency. Write a function that accepts as arguments the following:

   1. An array of integers
   2. An integer that indicates the number of elements in the array

   The function should determine the mode of the array. That is, it should determine which value in the array occurs most often. The mode is the value the function should return. If the array has no mode (none of the values occur more than once), the function should return −1. (Assume the array will always contain nonnegative values.)

Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

9. Median Function

   In statistics, when a set of values is sorted in ascending or descending order, its *median* is the middle value. If the set contains an even number of values, the median is the mean, or average, of the two middle values. Write a function that accepts as arguments the following:

   1. An array of integers

   2. An integer that indicates the number of elements in the array

   The function should determine the median of the array. This value should be returned as a `double`. (Assume the values in the array are already sorted.)

   Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

10. Reverse Array

    Write a function that accepts an `int` array and the array's size as arguments. The function should create a copy of the array, except that the element values should be reversed in the copy. The function should return a pointer to the new array. Demonstrate the function in a complete program.

11. Array Expander

    Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array and initialize the unused elements of the second array with 0. The function should return a pointer to the new array.

12. Element Shifter

    Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is one element larger than the argument array. The first element of the new array should be set to 0. Element 0 of the argument array should be copied to element 1 of the new array, element 1 of the argument array should be copied to element 2 of the new array, and so forth. The function should return a pointer to the new array.

13. Movie Statistics

    Write a program that can be used to gather statistical data about the number of movies college students see in a month. The program should perform the following steps:

    1. Ask the user how many students were surveyed. An array of integers with this many elements should then be dynamically allocated.
    2. Allow the user to enter the number of movies each student saw into the array.
    3. Calculate and display the average, median, and mode of the values entered. (Use the functions you wrote in Programming Challenges 8 and 9 to calculate the median and mode.)

    Input Validation: Do not accept negative numbers for input.