Pointers and Dynamic Arrays 9

Memory is necessary for all the operations of reason.

BLAISE PASCAL, *Pensées*

INTRODUCTION

A *pointer* is a construct that gives you more control of the computer's memory. This chapter shows how pointers are used with arrays and introduces a new form of array called a *dynamic array*. Dynamic arrays are arrays whose size is determined while the program is running, rather than being fixed when the program is written.

PREREQUISITES

Section 9.1, which covers the basics of pointers, uses material from Chapters 2 through 6. It does not require any of the material from Chapters 7 or 8. Section 9.2, which covers dynamic arrays, uses material from Section 9.1, and Chapters 2 through 7. It does not require any of the material from Chapter 8.

9.1 POINTERS

Do not mistake the pointing finger for the moon.

ZEN SAYING

A **pointer** is the memory address of a variable. Recall that the computer's memory is divided into numbered memory locations (called bytes) and that variables are implemented as a sequence of adjacent memory locations. Recall also that sometimes the C++ system uses these memory addresses as names for the variables. If a variable is implemented as, say, three memory locations, then the address of the first of these memory locations is sometimes used as a name for that variable. For example, when the variable is used as a call-by-reference argument, it is this address, not the identifier name of the variable, that is passed to the calling function.

An address that is used to name a variable in this way (by giving the address in memory where the variable starts) is called a *pointer* because the address can be thought of as "pointing" to the variable. The address "points" to the variable because it identifies the variable by telling *where* the variable is, rather than telling what the variable's name is. A variable that is, say, at location number 1007 can be pointed out by saying "it's the variable over there at location 1007."

You have already been using pointers in a number of situations. As we noted in the previous paragraph, when a variable is a call-by-reference argument in a function call, the function is given this argument variable in the form of a pointer to the variable. This is an important and powerful use for pointers, but it is done automatically for you by the C++ system. In this chapter, we show you how to write programs that manipulate pointers in any way you want, rather than relying on the system to manipulate the pointers for you.

Pointer Variables

Declaring pointer variables

A pointer can be stored in a variable. However, even though a pointer is a memory address and a memory address is a number, you cannot store a pointer in a variable of type *int* or *double* without type casting. A variable to hold a pointer must be declared to have a pointer type. For example, the following declares p to be a pointer variable that can hold one pointer that points to a variable of type *double*:

```
double *p;
```

The variable p can hold pointers to variables of type *double*, but it cannot normally contain a pointer to a variable of some other type, such as *int* or *char*. Each variable type requires a different pointer type.

In general, to declare a variable that can hold pointers to other variables of a specific type, you declare the pointer variable just as you would declare an ordinary variable of that type, but you place an asterisk in front of the variable name. For example, the following declares the variables p1 and p2 so that they can hold pointers to variables of type *int*; it also declares two ordinary variables, v1 and v2, of type *int*:

```
int *p1, *p2, v1, v2;
```

There must be an asterisk before *each* of the pointer variables. If you omit the second asterisk in the previous declaration, then p2 will not be a pointer variable; it will instead be an ordinary variable of type *int*. The asterisk is the same symbol you have been using for multiplication, but in this context it has a totally different meaning.

When discussing pointers and pointer variables, we usually speak of *pointing* rather than of *addresses*. When a pointer variable, such as p1, contains the address of a variable, such as v1, the pointer variable is said to *point to the variable* v1 or to be *a pointer to the variable* v1.

Pointer variables, like p1 and p2 declared earlier, can contain pointers to variables like v1 and v2. You can use the operator & to determine the address of a variable, and you can then assign that address to a pointer variable. For example, the following will set the variable p1 equal to a pointer that points to the variable v1:

```
p1 = &v1;
```

Pointer Variable Declarations

A variable that can hold pointers to other variables of type *Type_Name* is declared similarly to the way you declare a variable of type *Type_Name*, except that you place an asterisk at the beginning of the variable name.

SYNTAX

```
Type_Name *Variable_Name1, *Variable_Name2, . . .;
```

EXAMPLE

```
double *pointer1, *pointer2;
```

Addresses and Numbers

A pointer is an address, and an address is an integer, but a pointer is not an integer. That is not crazy. That is abstraction! C++ insists that you use a pointer as an address and that you not use it as a number. A pointer is not a value of type `int` or of any other numeric type. You normally cannot store a pointer in a variable of type `int`. If you try, most C++ compilers will give you an error message or a warning message. Also, you cannot perform the normal arithmetic operations on pointers. (You can perform a kind of addition and a kind of subtraction on pointers, but they are not the usual integer addition and subtraction.)

You now have two ways to refer to `v1`: You can call it `v1` or you can call it "the variable pointed to by `p1`." In C++, the way that you say "the variable pointed to by `p1`" is `*p1`. This is the same asterisk that we used when we declared `p1`, but now it has yet another meaning. When the asterisk is used in this way, it is often called the **dereferencing operator**, and the pointer variable is said to be **dereferenced.**

Putting these pieces together can produce some surprising results. Consider the following code:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

This code outputs the following to the screen:

```
42
42
```

As long as p1 contains a pointer that points to v1, then v1 and *p1 refer to the same variable. So when you set *p1 equal to 42, you are also setting v1 equal to 42.

The symbol & that is used to obtain the address of a variable is the same symbol that you use in function declarations to specify a call-by-reference parameter. This use is not a coincidence. Recall that a call-by-reference argument is implemented by giving the address of the argument to the calling function. So, these two uses of the symbol & are very much the same. However, the usages are slightly different and we will consider them to be two different (although very closely related) usages of the symbol &.

The * and & Operators

The *operator in front of a pointer variable produces the variable it points to. When used this way, the *operator is called the **dereferencing operator.**

The operator & in front of an ordinary variable produces the address of that variable; that is, produces a pointer that points to the variable. The & operator is called the address-of operator.

For example, consider the declarations

```
double *p, v;
```

The following sets the value of p so that p points to the variable v:

```
p = &v;
```

*p produces the variable pointed to by p, so after the assignment above, *p and v refer to the same variable. For example, the following sets the value of v to 9.99, even though the name v is never explicitly used:

```
*p = 9.99;
```

Pointers in assignment statements

You can assign the value of one pointer variable to another pointer variable. This copies an address from one pointer variable to another pointer variable. For example, if p1 is still pointing to v1, then the following will set p2 so that it also points to v1:

```
p2 = p1;
```

Provided we have not changed v1's value, the following also outputs a 42 to the screen:

```
cout << *p2;
```

Be sure that you do not confuse

```
    p1 = p2;
```

and

```
    *p1 = *p2;
```

When you add the asterisk, you are not dealing with the pointers p1 and p2, but with the variables that the pointers are pointing to. This is illustrated in Display 9.1.

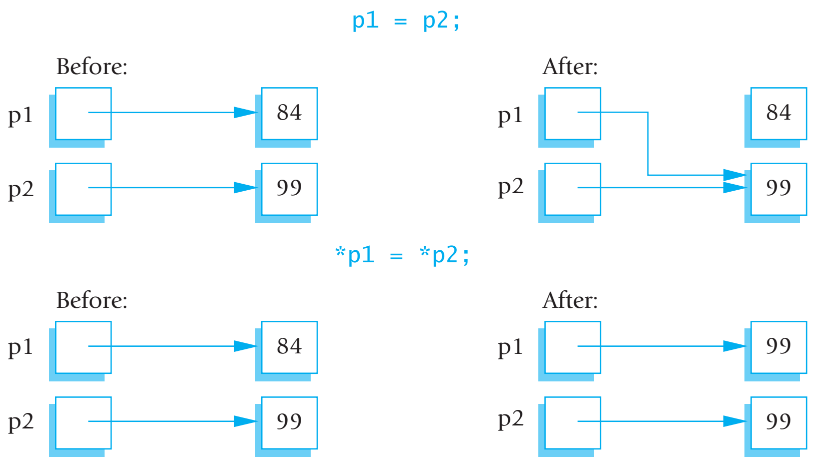
Since a pointer can be used to refer to a variable, your program can manipulate variables even if the variables have no identifiers to name them. The operator *new* can be used to create variables that have no identifiers to serve as their names. These nameless variables are referred to via pointers. For example, the following creates a new variable of type *int* and sets the pointer variable p1 equal to the address of this new variable (that is, p1 points to this new, nameless variable):

```
    p1 = new int;
```

This new, nameless variable can be referred to as *p1 (that is, as the variable pointed to by p1). You can do anything with this nameless variable that you can do with any other variable of type *int*. For example, the following reads a value of type *int* from the keyboard into this nameless variable, adds 7 to the value, then outputs this new value:

```
    cin >> *p1;
    *p1 = *p1 + 7;
    cout << *p1;
```

DISPLAY 9.1 Uses of the Assignment Operator



The *new* operator produces a new, nameless variable and returns a pointer that points to this new variable. You specify the type for this new variable by writing the type name after the *new* operator. Variables that are created using the *new* operator are called **dynamic variables** because they are created and destroyed while the program is running. The program in Display 9.2 demonstrates some simple operations on pointers and dynamic variables. Display 9.3 illustrates the working of the program in Display 9.2. In Display 9.3, variables are represented as boxes and the value of the variable is written inside the box. We have not shown the actual numeric addresses in the pointer variables. The actual numbers are not important. What is important is that the number is the address of some particular variable. So, rather than use the actual number of the address, we have merely indicated the address with an arrow that points to the variable with that address. For example, in illustration (b) in Display 9.3, p1 contains the address of a variable that has a question mark written in it.

DISPLAY 9.2 Basic Pointer Manipulations *(part 1 of 2)*

```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout<< "*p1 == " << *p1 << endl;
    cout<< "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout<< "*p1 == " << *p1 << endl;
    cout<< "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout<< "*p1 == " << *p1 << endl;
    cout<< "*p2 == " << *p2 << endl;
    cout<< "Hope you got the point of this example!\n";
    return 0;
}
```

(continued)

DISPLAY 9.2 Basic Pointer Manipulations *(part 2 of 2)*

Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Pointer Variables Used with =

If p1 and p2 are pointer variables, then the statement

```
p1 = p2;
```

changes p1 so that it points to the same thing that p2 is currently pointing to.

The *new* Operator

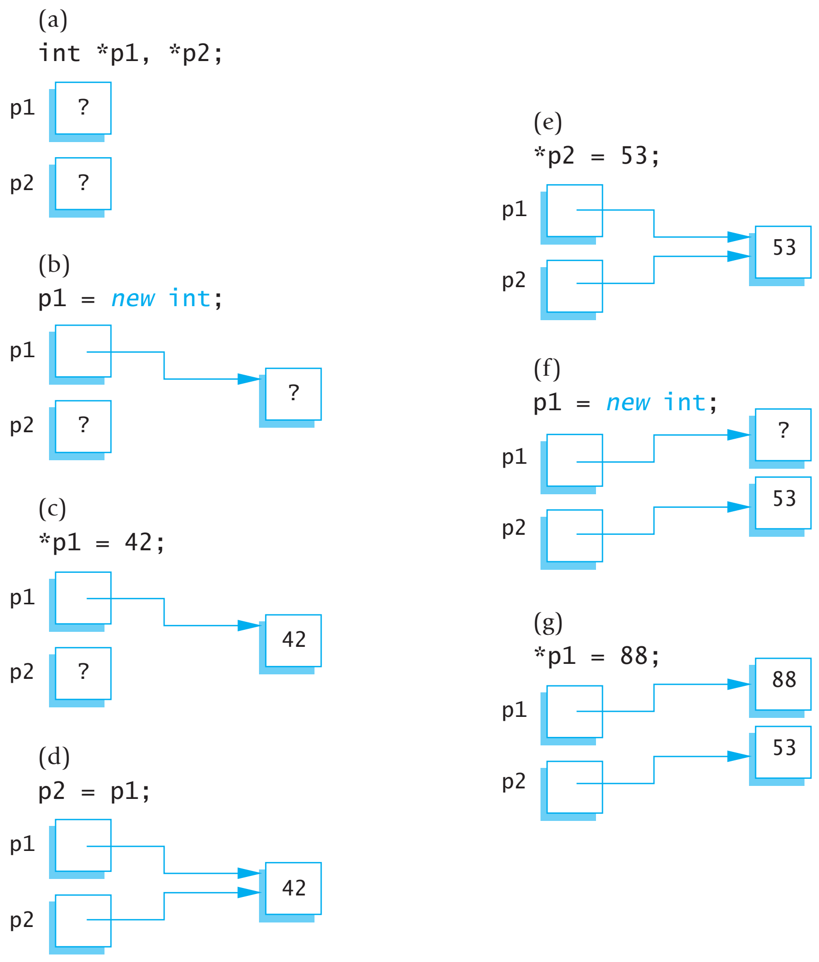
The *new* operator creates a new dynamic variable of a specified type and returns a pointer that points to this new variable. For example, the following creates a new dynamic variable of type MyType and leaves the pointer variable p pointing to this new variable:

```
MyType *p;
p = new MyType;
```

The C++ standard specifies that if there is not sufficient memory available to create the new variable, then the *new* operator, by default, terminates the program.[1]

[1] Technically, the *new* operator throws an exception, which, if not caught, terminates the program. It is possible to "catch" the exception or install a `new handler`, but these topics are not covered until Chapter 16.

DISPLAY 9.3 Explanation of Display 9.2



SELF-TEST EXERCISES

1. Explain the concept of a pointer in C++.

2. What unfortunate misinterpretation can occur with the following declaration?

```
int* int_ptr1, int_ptr2;
```

3. Give at least two uses of the * operator. State what the * is doing, and name the use of the * that you present.

4. What is the output produced by the following code?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

How would the output change if you were to replace

```
*p1 = 30;
```

with the following?

```
*p2 = 30;
```

5. What is the output produced by the following code?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; //This is different from Exercise 4
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Basic Memory Management

A special area of memory, called the **freestore,** is reserved for dynamic variables. Any new dynamic variable created by a program consumes some of the memory in the freestore.[2] If your program creates too many dynamic variables, it will consume all of the memory in the freestore. If this happens, any additional calls to *new* will fail.

The size of the freestore varies by computer and implementation of C++. It is typically large, and a modest program is not likely to use all the memory in the freestore. However, even on modest programs it is a good practice to recycle any freestore memory that is no longer needed. If your program no

[2] The freestore is also sometimes called the *heap*.

longer needs a dynamic variable, the memory used by that dynamic variable can be recycled. The *delete* operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore so that the memory can be reused. Suppose that p is a pointer variable that is pointing to a dynamic variable. The following will destroy the dynamic variable pointed to by p and return the memory used by the dynamic variable to the freestore:

```
delete p;
```

After this call to *delete*, the value of p is undefined and p should be treated like an uninitialized variable.

The *delete* Operator

The *delete* operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable p:

```
delete p;
```

After a call to *delete*, the value of the pointer variable, like p above, is undefined. (A slightly different version of *delete*, discussed later in this chapter, is used when the dynamic variable is an array.)

PITFALL Dangling Pointers

When you apply *delete* to a pointer variable, the dynamic variable it is pointing to is destroyed. At that point, the value of the pointer variable is undefined, which means that you do not know where it is pointing, nor what the value is where it is pointing. Moreover, if some other pointer variable was pointing to the dynamic variable that was destroyed, then this other pointer variable is also undefined. These undefined pointer variables are called **dangling pointers.** If p is a dangling pointer and your program applies the dereferencing operator * to p (to produce the expression *p), the result is unpredictable and usually disastrous. Before you apply the dereferencing operator *to a pointer variable, you should be certain that the pointer variable points to some variable. ■

Static Variables and Automatic Variables

Variables created with the `new` operator are called **dynamic variables,** because they are created and destroyed while the program is running. When compared with these dynamic variables, ordinary variables seem static, but the terminology used by C++ programmers is a bit more involved than that, and ordinary variables are not called *static variables.*

The ordinary variables we have been using in previous chapters are not really static. If a variable is local to a function, then the variable is created by the C++ system when the function is called and is destroyed when the function call is completed. Since the main part of a program is really just a function called `main`, this is even true of the variables declared in the main part of your program. (Since the call to `main` does not end until the program ends, the variables declared in `main` are not destroyed until the program ends, but the mechanism for handling local variables is the same for `main` as it is for any other function.) The ordinary variables that we have been using (that is, the variables declared within `main` or within some other function definition) are called **automatic variables** (not to be confused with variables defined of type `auto`), because their dynamic properties are controlled automatically for you; they are automatically created when the function in which they are declared is called and automatically destroyed when the function call ends. We will usually call these variables **ordinary variables,** but other books call them *automatic variables.*

There is one other category of variables, namely, **global variables.** Global variables are variables that are declared outside of any function definition (including being outside of `main`). We discussed global variables briefly in Chapter 4. As it turns out, we have no need for global variables and have not used them.

PROGRAMMING TIP Define Pointer Types

You can define a pointer type name so that pointer variables can be declared like other variables without the need to place an asterisk in front of each pointer variable. For example, the following defines a type called `IntPtr`, which is the type for pointer variables that contain pointers to `int` variables:

```
typedef int* IntPtr;
```

Thus, the following two pointer variable declarations are equivalent:

```
IntPtr p;
```

and

```
int *p;
```

You can use `typedef` to define an alias for any type name or definition. For example, the following defines the type name `Kilometers` to mean the same thing as the type name `double`:

```
typedef double Kilometers;
```

Once you have given this type definition, you can define a variable of type `double` as follows:

```
Kilometers distance;
```

Renaming existing types this way can occasionally be useful. However, our main use of `typedef` will be to define types for pointer variables.

There are two advantages to using defined pointer type names, such as `IntPtr` defined earlier. First, it avoids the mistake of omitting an asterisk. Remember, if you intend `p1` and `p2` to be pointers, then the following is a mistake:

```
int *p1, p2;
```

Since the * was omitted from the `p2`, the variable `p2` is just an ordinary `int` variable, not a pointer variable. If you get confused and place the * on the `int`, the problem is the same but is more difficult to notice. C++ allows you to place the * on the type name, such as `int`, so that the following is legal:

```
int* p1, p2;
```

Although this line is legal, it is misleading. It looks like both `p1` and `p2` are pointer variables, but in fact only `p1` is a pointer variable; `p2` is an ordinary `int` variable. As far as the C++ compiler is concerned, the *that is attached to the identifier `int` may as well be attached to the identifier `p1`. One correct way to declare both `p1` and `p2` to be pointer variables is

```
int *p1, *p2;
```

An easier and less error-prone way to declare both `p1` and `p2` to be pointer variables is to use the defined type name `IntPtr` as follows:

```
IntPtr p1, p2;
```

The second advantage of using a defined pointer type, such as `IntPtr`, is seen when you define a function with a call-by-reference parameter for a pointer variable. Without the defined pointer type name, you would need to include both an * and an & in the function declaration for the function, and the details can get confusing. If you use a type name for the pointer type, then a call-by-reference parameter for a pointer type involves no complications. You define a call-by-reference parameter for a defined pointer type just like you define any other call-by-reference parameter. Here's a sample:

```
void sample_function(IntPtr& pointer_variable);
```
■

Type Definitions

You can assign a name to a type definition and then use the type name to declare variables. This is done with the keyword *typedef*. These type definitions are normally placed outside of the body of the main part of your program (and outside the body of other functions) . We will use type definitions to define names for pointer types, as shown in the example below.

SYNTAX

```
typedef Known_Type_Definition New_Type_Name;
```

EXAMPLE

```
typedef int* IntPtr;
```

The type name `IntPtr` can then be used to declare pointers to dynamic variables of type *int*, as in the following:

```
IntPtr pointer1, pointer2;
```

SELF-TEST EXERCISES

6. Suppose a dynamic variable were created as follows:

   ```
   char *p;
   p = new char;
   ```

   Assuming that the value of the pointer variable `p` has not changed (so it still points to the same dynamic variable), how can you destroy this new dynamic variable and return the memory it uses to the freestore so that the memory can be reused to create new dynamic variables?

7. Write a definition for a type called `NumberPtr` that will be the type for pointer variables that hold pointers to dynamic variables of type *int*. Also, write a declaration for a pointer variable called `my_point` that is of type `NumberPtr`.

8. Describe the action of the *new* operator. What does the operator *new* return?

9.2 DYNAMIC ARRAYS

In this section you will see that array variables are actually pointer variables. You will also find out how to write programs with dynamic arrays. A **dynamic array** is an array whose size is not specified when you write the program, but is determined while the program is running.

Array Variables and Pointer Variables

In Chapter 7 we described how arrays are kept in memory. At that point we had not learned about pointers, so we discussed arrays in terms of memory addresses. But, a memory address is a pointer. So, in C++ an array variable is actually a pointer variable that points to the first indexed variable of the array. Given the following two variable declarations, p and a are the same kind of variable:

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

The fact that a and p are the same kind of variable is illustrated in Display 9.4. Since a is a pointer that points to a variable of type *int* (namely the variable a[0]), the value of a can be assigned to the pointer variable p as follows:

```
p = a;
```

After this assignment, p points to the same memory location that a points to. So, p[0], p[1], … p[9] refer to the indexed variables a[0], a[1], … a[9]. The square bracket notation you have been using for arrays applies to pointer variables as long as the pointer variable points to an array in memory. After this assignment, you can treat the identifier p as if it were an array identifier. You can also treat the identifier a as if it were a pointer variable, but there is one important reservation. *You cannot change the pointer value in an array variable, such as* a. You might be tempted to think the following is legal, but it is not:

```
IntPtr p2;
...//p2 is given some pointer value.
a = p2;//ILLEGAL. You cannot assign a different address to a.
```

Display 9.5 illustrates the working of the program in Display 9.4. As in Display 9.3, variables are represented as boxes and the value of the variable is written inside the box. An arrow indicates a pointer or reference to another memory location, in this case, the first element of the array.

DISPLAY 9.4 Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p are also changes to the array a.

Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

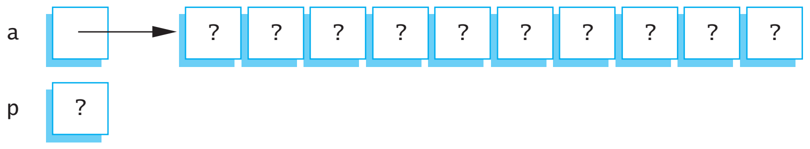
Creating and Using Dynamic Arrays

One problem with the kinds of arrays you have used thus far is that you must specify the size of the array when you write the program—but you may not know what size array you need until the program is run. For example, an array might hold a list of student identification numbers, but the size of the class may be different each time the program is run. With the kinds of arrays you have used thus far, you must estimate the largest possible size you may need

DISPLAY 9.5 Explanation of Display 9.4

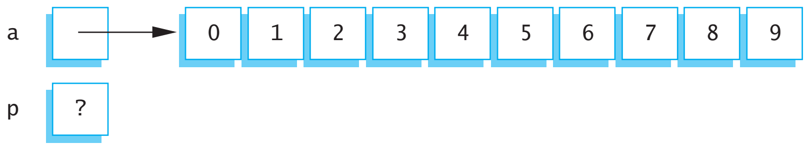
(a)

```
IntPtr p;
int a[10];
```



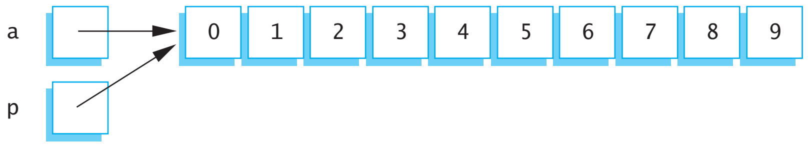
(b)

```
for (index = 0; index < 10; index++)
   a[index] = index;
```



(c)

```
p = a;
```



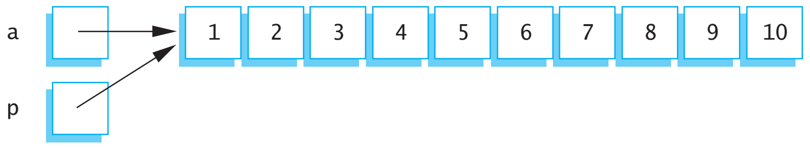
```
for (index=0; index < 10; index++)
   cout << p[index] << " ";
```

Iterating through p is the same as iterating through a

```
Output  0  1  2  3  4  5  6  7  8  9
```

(d)

```
for (index = 0; index < 10; index++)
   p[index] = p[index] + 1;
```



```
for (index=0; index < 10; index++)
   cout << a[index] << " ";
```

Iterating through a is the same as iterating through p

```
Output  1  2  3  4  5  6  7  8  9  10
```

for the array and hope that size is large enough. There are two problems with this. First, you may estimate too low, and then your program will not work in all situations. Second, since the array might have many unused positions, this can waste computer memory. Dynamic arrays avoid these problems. If your program uses a dynamic array for student identification numbers, then the size of the class can be entered as input to the program and the dynamic array can be created to be exactly that size.

Creating a dynamic array

Dynamic arrays are created using the *new* operator. The creation and use of dynamic arrays is surprisingly simple. Since array variables are pointer variables, you can use the *new* operator to create dynamic variables that are arrays and treat these dynamic array variables as if they were ordinary arrays. For example, the following creates a dynamic array variable with ten array elements of type *double*:

```
typedef double* DoublePtr;
DoublePtr p;
p = new double [10];
```

To obtain a dynamic array of elements of any other type, simply replace *double* with the desired type. To obtain a dynamic array variable of any other size, simply replace 10 with the desired size.

There are also a number of less obvious things to notice about this example. First, the pointer type that you use for a pointer to a dynamic array is the same as the pointer type you would use for a single element of the array. For instance, the pointer type for an array of elements of type *double* is the same as the pointer type you would use for a simple variable of type *double*. The pointer to the array is actually a pointer to the first indexed variable of the array. In the previous example, an entire array with ten indexed variables is created and the pointer p is left pointing to the first of these ten indexed variables.

Also notice that when you call *new*, the size of the dynamic array is given in square brackets after the type, which in this example is the type *double*. This tells the computer how much storage to reserve for the dynamic array. If you omit the square brackets and the 10, the computer will allocate enough storage for only one variable of type *double*, rather than for an array of ten indexed variables of type *double*. As illustrated in Display 9.6, you can use an *int* variable in place of the constant 10 so that the size of the dynamic array can be read into the program.

The program in Display 9.6 sorts a list of numbers. This program works for lists of any size because it uses a dynamic array to hold the numbers. The size of the array is determined when the program is run. The user is asked how many numbers there will be, and then the *new* operator creates a dynamic array of that size. The size of the dynamic array is given by the variable array_size.

Notice the *delete* statement, which destroys the dynamic array variable a in Display 9.6. Since the program is about to end anyway, we did not really need this *delete* statement; however, if the program went on to do other

DISPLAY 9.6 A Dynamic Array *(part 1 of 2)*

```
//Sorts a list of numbers entered at the keyboard.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);
//Precondition: size is the size of the array a.
//Postcondition: a[0] through a[size- 1] have been
//filled with values read from the keyboard.

void sort(int a[], int size);
//Precondition: size is the size of the array a.
//The array elements a[0] through a[size–1] have values.
//Postcondition: The values of a[0] through a[size–1] have been rearranged
//so that a[0] <= a[1] <= ... <= a[size–1].

int main( )
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];

    fill_array(a, array_size);
    sort(a, array_size);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [] a;

    return 0;
}

//Uses the library iostream:
void fill_array(int a[], int size)
{
```

Ordinary array parameters

The dynamic array a is used like an ordinary array.

(continued)

DISPLAY 9.6 A Dynamic Array *(part 2 of 2)*

```
    using namespace std;
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)
```

<Any implementation of sort may be used. This may or may not require some additional function definitions. The implementation need not even know that sort will be called with a dynamic array. For example, you can use the implementation in Display 7.12 (with suitable adjustments to parameter names).>

things with dynamic variables, you would want such a *delete* statement so that the memory used by this dynamic array is returned to the freestore. The *delete* statement for a dynamic array is similar to the *delete* statement you saw earlier, except that with a dynamic array you must include an empty pair of square brackets, like so:

```
delete [] a;
```

The square brackets tell C++ that a dynamic array variable is being eliminated, so the system checks the size of the array and removes that many indexed variables. If you omit the square brackets, you would be telling the computer to eliminate only one variable of type *int*. For example,

```
delete a;
```

is not legal, but the error is not detected by most compilers. The ANSI C++ standard says that what happens when you do this is "undefined." That means the author of the compiler can have this do anything that is convenient—convenient for the compiler writer, not for you. Even if it does something useful, you have no guarantee that either the next version of that compiler or any other compiler you compile this code with will do the same thing. The moral is simple: Always use the

```
delete [] array_ptr;
```

syntax when you are deleting memory that was allocated with something like

```
array_ptr = new MyType[37];
```

You create a dynamic array with a call to *new* using a pointer, such as the pointer a in Display 9.6. After the call to *new*, you should not assign any other pointer value to this pointer variable, because that can confuse the system when the memory for the dynamic array is returned to the freestore with a call to *delete*.

How to Use a Dynamic Array

- ***Define a pointer type:*** Define a type for pointers to variables of the same type as the elements of the array. For example, if the dynamic array is an array of *double*, you might use the following:

```
typedef double* DoubleArrayPtr;
```

- ***Declare a pointer variable:*** Declare a pointer variable of this defined type. The pointer variable will point to the dynamic array in memory and will serve as the name of the dynamic array.

```
DoubleArrayPtr a;
```

- ***Call new:*** Create a dynamic array using the *new* operator:

```
a = new double[array_size];
```

  The size of the dynamic array is given in square brackets as in the example above. The size can be given using an *int* variable or other *int* expression. In the example above, `array_size` can be a variable of type *int* whose value is determined while the program is running.

- ***Use like an ordinary array:*** The pointer variable, such as a, is used just like an ordinary array. For example, the indexed variables are written in the usual way: `a[0]`, `a[1]`, and so forth. The pointer variable should not have any other pointer value assigned to it, but should be used like an array variable.

- ***Call delete*[ ]*:* When your program is finished with the dynamic variable, use *delete* and empty square brackets along with the pointer variable to eliminate the dynamic array and return the storage that it occupies to the freestore for reuse. For example:

```
delete [] a;
```

Dynamic arrays are created using *new* and a pointer variable. When your program is finished using a dynamic array, you should return the array memory to the freestore with a call to *delete*. Other than that, a dynamic array can be used just like any other array.

SELF-TEST EXERCISES

9. Write a type definition for pointer variables that will be used to point to dynamic arrays. The array elements are to be of type *char*. Call the type `CharArray`.

10. Suppose your program contains code to create a dynamic array as follows:

```
int *entry;
entry = new int[10];
```

so that the pointer variable `entry` is pointing to this dynamic array. Write code to fill this array with ten numbers typed in at the keyboard.

11. Suppose your program contains code to create a dynamic array as in Self-Test Exercise 10, and suppose the pointer variable `entry` has not had its (pointer) value changed. Write code to destroy this new dynamic array and return the memory it uses to the freestore.

12. What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int a[10];
int *p = a;
int i;
for (i = 0; i < 10; i++)
    a[i] = i;

for (i = 0; i < 10; i++)
    cout << p[i] << " ";
cout << endl;
```

13. What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int [array_size];
int *p = a;
int i;
for (i = 0; i < array_size; i++)
    a[i] = i;
p[0] = 10;

for (i = 0; i < array_size; i++)
    cout << a[i] << " ";
cout << endl;
```

Pointer Arithmetic (*Optional*)

There is a kind of arithmetic you can perform on pointers, but it is an arithmetic of addresses, not an arithmetic of numbers. For example, suppose your program contains the following code:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

After these statements, d contains the address of the indexed variable d[0]. The expression d + 1 evaluates to the address of d[1], d + 2 is the address of d[2], and so forth. Notice that although the value of d is an address and an address is a number, d+1 does not simply add 1 to the number in d. If a variable of type *double* requires 8 bytes (eight memory locations) and d contains the address 2001, then d+1 evaluates to the memory address 2009. Of course, the type *double* can be replaced by any other type and then pointer addition moves in units of variables for that type.

This pointer arithmetic gives you an alternative way to manipulate arrays. For example, if array_size is the size of the dynamic array pointed to by d, then the following will output the contents of the dynamic array:

```
for (int i = 0; i < array_size; i++)
    cout << *(d + i)<< " ";
```

This code is equivalent to the following:

```
for(int i = 0; i < array_size; i++)
    cout << d[i] << " ";
```

You may not perform multiplication or division of pointers. All you can do is add an integer to a pointer, subtract an integer from a pointer, or subtract two pointers of the same type. When you subtract two pointers, the result is the number of indexed variables between the two addresses. Remember, for subtraction of two pointer values, these values must point into the same array! It makes little sense to subtract a pointer that points into one array from another pointer that points into a different array. You can use the increment and decrement operators ++ and --. For example, d++ will advance the value of d so that it contains the address of the next indexed variable, and d-- will change d so that it contains the address of the previous indexed variable.

VideoNote
Dynamic Arrays and Pointer Arithmetic

SELF-TEST EXERCISES

These exercises apply to the optional section on pointer arithmetic.

14. What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
    *(a + i) = i;
for (i = 0; i < array_size; i++)
    cout << a[i] << " ";
cout << endl;
```

15. What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
    a[i] = i;
while (*a < 9)
{
    a++;
    cout << *a << " ";
}
cout << endl;
```

Multidimensional Dynamic Arrays (*Optional*)

You can have multidimensional dynamic arrays. You just need to remember that multidimensional arrays are arrays of arrays, or arrays of arrays of arrays, or so forth. For example, to create a two-dimensional dynamic array, you must remember that it is an array of arrays. To create a two-dimensional array of integers, you first create a one-dimensional dynamic array of pointers of type *int**, which is the type for a one-dimensional array of *int*s. Then you create a dynamic array of *int*s for each indexed variable of the array of pointers.

A type definition may help to keep things straight. The following is the variable type for an ordinary one-dimensional dynamic array of *int*s:

```
typedef int* IntArrayPtr;
```

To obtain a 3-by-4 array of *int*s, you want an array whose base type is `IntArrayPtr`. For example:

```
IntArrayPtr *m = new IntArrayPtr[3];
```

This is an array of three pointers, each of which can name a dynamic array of *int*s, as follows:

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

The resulting array `m` is a 3-by-4 dynamic array. A simple program to illustrate this is given in Display 9.7.

Be sure to notice the use of *delete* in Display 9.7. Since the dynamic array `m` is an array of arrays, each of the arrays created with *new* in the *for* loop must be returned to the freestore manager with a call to *delete*[];

then, the array m itself must be returned to the freestore with another call to *delete*[]. There must be one call to *delete*[] for each call to *new* that created an array. (Since the program ends right after the calls to *delete*[], we could safely omit these calls, but we wanted to illustrate their usage.)

DISPLAY 9.7 A Two-Dimensional Dynamic Array *(part 1 of 2)*

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

typedef int* IntArrayPtr;

int main( )
{
    int d1, d2;
    cout << "Enter the row and column dimensions of the array:\n";
    cin >> d1 >> d2;

    IntArrayPtr *m = new IntArrayPtr[d1];
    int i, j;
    for (i = 0; i < d1; i++)
        m[i] = new int[d2];
    //m is now a d1 by d2 array.

    cout << "Enter " << d1 << " rows of "
         << d2 << " integers each:\n";
    for (i = 0; i < d1; i++)
        for (j = 0; j < d2; j++)
            cin >> m[i][j];

    cout << "Echoing the two-dimensional array:\n";
    for (i = 0; i < d1; i++)
    {
        for (j = 0; j < d2; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }
    for (i = 0; i < d1; i++)
        delete[] m[i];
    delete[] m;

    return 0;
}
```

Note that there must be one call to **delete[ ]** *for each call to* **new** *that created an array. (These calls to* **delete[ ]** *are not really needed, since the program is ending, but in another context it could be important to include them.)*

(continued)

DISPLAY 9.7 A Two-Dimensional Dynamic Array *(part2 of 2)*

Sample Dialogue

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

CHAPTER SUMMARY

- A **pointer** is a memory address, so a pointer provides a way to indirectly name a variable by naming the address of the variable in the computer's memory.
- **Dynamic variables** are variables that are created (and destroyed) while a program is running.
- Memory for dynamic variables is in a special portion of the computer's memory called the **freestore.** When a program is finished with a dynamic variable, the memory used by the dynamic variable can be returned to the freestore for reuse; this is done with a `delete` statement.
- A **dynamic array** is an array whose size is determined when the program is running. A dynamic array is implemented as a dynamic variable of an array type.

Answers to Self-Test Exercises

1. A pointer is the memory address of a variable.
2. To the unwary, or to the neophyte, this looks like two objects of type pointer to `int`, that is, `int*`. Unfortunately, the * binds to the *identifier*, not to the type (that is, not to the `int`). The result is that this declaration declares `int_ptr1` to be an `int` pointer, while `int_ptr2` is just an ordinary `int` variable.

3.
```
int *p;   //This declares a pointer variable that can
          //hold a pointer to an int variable.
*p = 17;  //Here, * is the dereference operator.
//This assigns 17 to the memory location pointed to by p.
```

4.
```
10 20
20 20
30 30
```

   If you replace `*p1 = 30;` with `*p2 = 30;`, the output would be the same.

5.
```
10 20
20 20
30 20
```

6. `delete p;`

7.
```
typedef int* NumberPtr;
NumberPtr my_point;
```

8. The *new* operator takes a type for its argument. *new* allocates space on the freestore of an appropriate size for a variable of the type of the argument. It returns a pointer to that memory (that is, a pointer to that new dynamic variable), provided there is enough available memory in the freestore. If there is not enough memory available in the freestore, your program ends.

9. `typedef char* CharArray;`

10.
```
cout << "Enter 10 integers:\n";
for (int i = 0; i < 10; i++)
    cin >> entry[i];
```

11. `delete [] entry;`

12. 0 1 2 3 4 5 6 7 8 9

13. 10 1 2 3 4 5 6 7 8 9

14. 0 1 2 3 4 5 6 7 8 9

15. 1 2 3 4 5 6 7 8 9

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. In the C programming language there is no pass-by-reference syntax to pass a variable by reference to a function. Instead a variable is passed by pointer (just to be confusing, sometimes passing by pointer is referred to as pass by reference). This Practice Program asks you to do the same thing as C, which in

practice would be simpler to implement using C++'s reference parameter syntax. Here is the header for a function that takes as input a pointer to an integer:

```
void addOne(int *ptrNum)
```

Complete the function so it adds one to the integer referenced by `ptrNum`. Write a `main` function where an integer variable is defined, give it an initial value, call `addOne`, and output the variable. It should be incremented by 1.

2. Write a program that asks the user to input an integer named `numDoubles`. Create a dynamic array that can store `numDoubles` doubles and make a loop that allows the user to enter a double into each array entry. Loop through the array, calculate the average, and output it. Delete the memory allocated to your dynamic array before exiting.

3. This Practice Program requires that you read the optional section about pointer arithmetic. Complete the function `isPalindrome` so that it returns `true` if the string `cstr` is a palindrome (the same backwards as forwards) and `false` if it is not. The function uses the `cstring` library.

```
bool isPalindrome(char* cstr)
{
   char* front = cstr;
   char* back = cstr + strlen(cstr)-1;

   while (front < back)
   {
      // Complete code here
   }
   return true;
}
```

Here is a sample `main` function for quick and dirty testing:

```
int main()
{
    char s1[50] = "neveroddoreven";
    char s2[50] = "not a palindrome";
    cout << isPalindrome(s1) << endl; // true
    cout << isPalindrome(s2) << endl; // false
    return 0;
}
```

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Do Programming Project 7 in Chapter 7 using a dynamic array. In this version of the problem, use dynamic arrays to store the digits in each large

integer. Allow an arbitrary number of digits instead of capping the number of digits at 20.

2. Do Programming Project 3 in Chapter 7. In this version of the problem, return a new dynamic array where all repeated letters are deleted instead of modifying the partially filled array. Don't forget to free the memory allocated for these returned dynamic arrays when the data is no longer needed.

3. Do Programming Project 11 in Chapter 7 using a dynamic array (or arrays). In this version, your program will ask the user how many rows the plane has and will handle that many rows (and so not always assume the plane has 7 rows as it did in Programming Project 11 of Chapter 7).

4. Write a function that takes a C string as an input parameter and reverses the string. The function should use two pointers, *front* and *rear*. The *front* pointer should initially reference the first character in the string, and the *rear* pointer should initially reference the last character in the string. Reverse the string by swapping the characters referenced by *front* and *rear*, then increment *front* to point to the next character and decrement *rear* to point to the preceding character, and so on, until the entire string is reversed. Write a main program to test your function on various strings of both even and odd length.

5. You run four computer labs. Each lab contains computer stations that are numbered as shown in the table below:

Lab Number	Computer Station Numbers
1	1–5
2	1–6
3	1–4
4	1–3

Each user has a unique five-digit ID number. Whenever a user logs on, the user's ID, lab number, and the computer station number are transmitted to your system. For example, if user 49193 logs onto station 2 in lab 3, then your system receives (49193, 2, 3) as input data. Similarly, when a user logs off a station, then your system receives the lab number and computer station number.

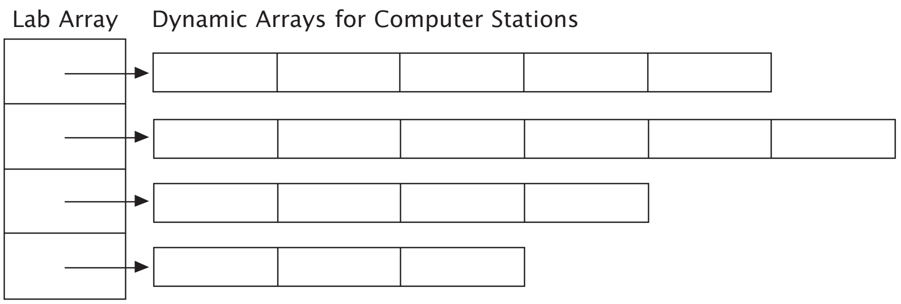
Write a computer program that could be used to track, by lab, which user is logged onto which computer. For example, if user 49193 is logged into station 2 in lab 3 and user 99577 is logged into station 1 of lab 4, then your system might display the following:

```
Lab Number Computer Stations
1          1: empty 2: empty 3: empty 4: empty 5: empty
2          1: empty 2: empty 3: empty 4: empty 5: empty 6: empty
3          1: empty 2: 49193 3: empty 4: empty
4          1: 99577 2: empty 3: empty
```

Create a menu that allows the administrator to simulate the transmission of information by manually typing in the login or logoff data. Whenever someone logs in or out, the display should be updated. Also write a search option so that the administrator can type in a user ID and the system will output what lab and station number that user is logged into, or "None" if the user ID is not logged into any computer station.

You should use a fixed array of length 4 for the labs. Each array entry points to a dynamic array that stores the user login information for each respective computer station.

The structure is shown in the figure below. This structure is sometimes called a ragged array since the columns are of unequal length.



VideoNote
Solution to Programming Project 9.6

6. One problem with dynamic arrays is that once the array is created using the new operator, the size cannot be changed. For example, you might want to add or delete entries from the array as you can with a vector. This project asks you to create functions that use dynamic arrays to emulate the behavior of a vector.

First, write a program that creates a dynamic array of five strings. Store five names of your choice into the dynamic array. Next, complete the following two functions:

```
string* addEntry(string *dynamicArray, int &size, string
                 newEntry);
```

This function should create a new dynamic array one element larger than `dynamicArray`, copy all elements from `dynamicArray` into the new array, add the new entry onto the end of the new array, increment size, delete `dynamicArray`, and return the new dynamic array.

```
string* deleteEntry(string *dynamicArray, int &size, string
                    entryToDelete);
```

This function should search `dynamicArray` for `entryToDelete`. If not found, the request should be ignored and the unmodified `dynamicArray` returned. If found, create a new dynamic array one element smaller than `dynamicArray`. Copy all elements except `entryToDelete` into the new array, delete `dynamicArray`, decrement size, and return the new dynamic array.

Test your functions by adding and deleting several names to the array while outputting the contents of the array. You will have to assign the array returned by `addEntry` or `deleteEntry` back to the dynamic array variable in your `main` function.

7. What if C++ had no built-in facility for two-dimensional arrays? It is possible to emulate them yourself with wrapper functions around a one-dimensional array. The basic idea is shown below. Consider the following two-dimensional array:

```
int matrix[2][3];
```

It can be visualized as a table:

matrix[0][0]	matrix[0][1]	matrix[0][2]
matrix[1][0]	matrix[1][1]	matrix[1][2]

The two-dimensional array can be mapped to storage in a one-dimensional array where each row is stored in consecutive memory locations (your compiler actually does something very similar to map two-dimensional arrays to memory).

```
int matrix1D[6];
```

matrix[0][0]	matrix1D[0][1]	matrix1D[0][2]	matrix1D[1][0]	matrix1D[1][1]	matrix1D[1][2]

Here, the mapping is as follows:

```
matrix[0][0] would be stored in matrix1D[0]
matrix[0][1] would be stored in matrix1D[1]
matrix[0][2] would be stored in matrix1D[2]
matrix[1][0] would be stored in matrix1D[3]
matrix[1][1] would be stored in matrix1D[4]
matrix[1][2] would be stored in matrix1D[5]
```

Based on this idea, complete the definitions for the following functions:

```
int* create2DArray(int rows, int columns);
```

This creates a one-dimensional dynamic array to emulate a two-dimensional array and returns a pointer to the one-dimensional dynamic array.

rows is the number of rows desired in the two-dimensional array.
columns is the number of columns desired in the two-dimensional array.
Return value: a pointer to a one-dimensional dynamic array large enough to hold a two-dimensional array of size rows * columns.

Note that `int ptr = create2DArray(2,3);` would create an array analogous to that created by `int ptr[2][3];`

```
void set(int *arr, int rows, int columns,
         int desired_row, int desired_column, int val);
```

This stores val into the emulated two-dimensional array at position desired_row, desired_column. The function should print an error message and exit if the desired indices are invalid.

arr is the one-dimensional array used to emulate a two-dimensional array.
rows is the total number of rows in the two-dimensional array.
columns is the total number of columns in the two-dimensional array.
desired_row is the zero-based index of the row the caller would like to access.
desired_column is the zero-based index of the column the caller would like to access.
val is the value to store at desired_row and desired_column.

```
int get(int *arr, int rows, int columns,
        int desired_row, int desired_column);
```

This returns the value in the emulated two-dimensional array at position desired_row, desired_column. The function should print an error message and exit if the desired indices are invalid.

arr is the one-dimensional array used to emulate a two-dimensional array.
rows is the total number of rows in the two-dimensional array.
columns is the total number of columns in the two-dimensional array.
desired_row is the zero-based index of the row the caller would like to access.
desired_column is the zero-based index of the column the caller would like to access.

Create a suitable test program that invokes all three functions.

8. Write a program that outputs a histogram of student grades for an assignment. The program should input each student's grade as an integer

and store the grade in a vector (covered in Chapter 8). Grades should be entered until the user enters -1 for a grade. The program should then scan through the vector and compute the histogram. In computing the histogram, the minimum value of a grade is 0 but your program should determine the maximum value entered by the user. Use a dynamic array to store the histogram. Output the histogram to the console. For example, if the input is:

```
20
30
4
20
30
30
-1
```

Then the output should be:

```
Number of 4's:   1
Number of 20's:  2
Number of 30's:  3
```