What we will learn about this week:

- Declaring and referencing arrays
- Arrays in memory
- Initializing arrays
- indexed variables
- arrays as function arguments

Arrays

a way of expressing many of the same variable type:

```
//declare five variables of type int
int num1, num2, num3, num4, num5;

//declare five variables of type int as an array
int num[5];
```

declares these five variables:

num[0], num[1], num[2], num[3], num[4]

Review of BlackJack Lab

- we needed to keep track of up to 5 cards: card1, card2, card3, card4, card5
- we could have used an array:
- char cards[5];

Declaring an array

int score[5];

This makes five integer variables called score[0], score[1], score[2], score[3], score[4]

These are **indexed variables**, **subscripted variables**, or **elements**

score is the **name** of the array

5 is the declared **size** of the array

int is the **base type**

Array Declaration

Syntax:

Type_Name Array_Name[*Declared_Size*];

Examples:

```
int big_array[100];
double a[3];
double b[5];
char grade[10], one_grade;
```

An array declaration, of the form shown above, will define *Declared_Size* index variables, namely the indexed variables *Array_Name*[0] through *Array_Name*[*Declared_Size*–1]. Each index variable is a variable of type *Type_Name*.

The array a consists of the indexed variables a[0], a[1], and a[2], all of type *double*. The array b consists of the indexed variables b[0], b[1], b[2], b[3], and b[4], also all of type *double*. You can combine array declarations with the declaration of simple variables such as the variable one_grade shown above.

Common Error

- First element starts at zero.
- The last element is one less than the declared size.
- This is a constant source of errors!

Using indexed variables

- Array references may be used anywhere an ordinary variable is used.

```
cin >> score[4] >> score[2];
cout << score[2] << " " << score[4];
score[0] = 32;
score[3] = next;
score[next] =3;
```

- The index can be a constant or an integer variable

Display 10.1 page 495

```
//Reads in 5 scores and shows how much each score differs from the highest.
#include <iostream>
int main( )
{
    using namespace std;
    int i, score[5], max;
    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        //max is the largest of the values score[0],..., score[i].
    }
```

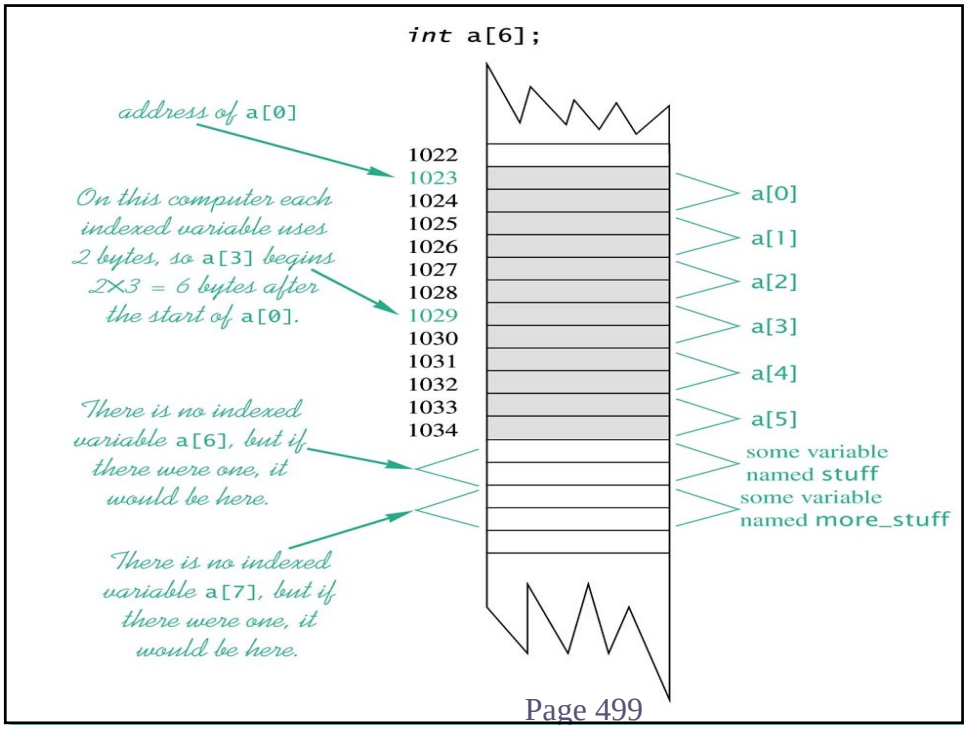
```
cout << "The highest score is " << max << endl
        << "The scores and their\n"
        << "differences from the highest are:\n";
   for (i = 0; i < 5; i++)
      cout << score[i] << " off by "
           << (max - score[i]) << endl;
   return 0;
}
```

Arrays in memory

- A computer's memory is like a long list of numbered locations called bytes.
- The numbers are called *addresses*, and the information written there is either some program's instructions or data.
- Every variable, whether the type is built-in or constructed by the programmer has a location where the variable starts and a number of bytes necessary to hold the variable that is determined by the type of the variable.
- Hence, every variable has an address and a type.
- An array variable is represented in this way, but here there is more. Consider:

  ```
  int a[6];
  ```

- Here the compiler decides where in memory to put the array, and the size of the array is 6 * the size of an *int*.
- The size of an array is the Declared_Size * sizeof(Array_Type)



Page 499

Array Index Out of Range

- The most common programming error when using arrays is the attempt to reference a non-existent array index.
- The array definition:

  int a[6]; //declares ONLY the indexed variables a[0] through a[5].
- An index value outside 0 to 5 is an **out of range** error, i.e., **illegal**.
- Consider: a[7] = 248;
- C++ treats 7 as if it were legal subscript, and attempts to write to memory where a[7] would be.
- Inspection of Display 9.3 suggests a[7] would be written two chunks of memory the size of an *int* beyond the real end of the array.
- Unfortunately, compilers do not detect this error. Your program may *seem to work correctly.* The effects range from no detectable effects, to abnormal end of program, to operating system crashing, to the next application crashing obscurely when it is started.
- C++ arrays were designed to be a low level construct that is not normally array bounds checked.

Initializing Arrays

- A variable of simple type can hold only one value in the variable
- A simple variable can be initialized in one statement:

```
int sum = 0;
```

- An array also can be initialized in one statement:

```
int children[3] = { 2, 12, 1};
```

- The compiler will count for you:

```
int b[ ] = { 5, 12, 11};
```

- This is equivalent to

```
int b[3] = { 5, 12, 11};
```

- If there is no initializer, no initialization is done.

13

Use defined constant for the Size of an Array

- Array indices always start at 0 and end with a value one less than the size of the array.
- Consider changing a program where the size of an array (say 50) appears several dozen times over thousands of lines.
- You might be looking for 49, 50 or perhaps 51. You must understand every instance where a number in this range was used. You might have to find where 25 (= 50/2) was used. You can't be certain you have completed the job.
- In order to write code that is easily and correctly modifiable, you should use a defined constant for the array size.
- Define a constant SIZE, and use that. Then you have only one point where you need to change the size:
- const int SIZE = 50;

Arrays in functions

- **An indexed variable is just a variable whose type is the base type of the array, and may be used anywhere any other variable whose type is the base type of the array might be used.**

```
int i, n, a[10];
my_function(n);
my_function(a[3]);
```

- **Display 10.3 illustrates this notion.**

Display 10.3page 503
Illustrates the use of an indexed variable as an argument.

```
//Adds 5 to each employee's allowed number of vacation days.
#include <iostream>
const int NUMBER_OF_EMPLOYEES = 3;
int adjust_days(int old_days);  //Returns old_days plus 5.
int main( )
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;
    cout << "Enter allowed vacation days for employees 1"
         << " through " << NUMBER_OF_EMPLOYEES << ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
       cin >> vacation[number-1];
    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
       vacation[number] = adjust_days(vacation[number]);
    cout << "The revised number of vacation days are:\n";
```

```
for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);
    cout << "The revised number of vacation days are:\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
             << " vacation days = " << vacation[number-1] << endl;
    return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

Arrays as function arguments

- It is possible to use an entire array as a formal parameter for a function.
- Remember a **formal parameter** is a kind of place holder that is filled in by the argument at the time the function is called. **Arguments** are placed in parentheses after the function name to signal that the function is being called and that this is the list of arguments to be “plugged in” for the parameter.
- A function can have a formal parameter for an entire array that is neither a call-by-value nor a call-by-reference parameter.
- This is a new parameter type called an **array parameter.**
- This is not call-by-value because we can change the array when passed this way.
- This is not call-by-reference because we cannot change the complete array by a single assignment when passed this way.
- While this is not call-by-reference, the behavior is like call-by-reference since we can change individual array elements by assignining them in the called function.

- When an array is used as an argument is a function call, any action that is performed on the array parameter is performed on the array argument, so the values of the indexed variables of the array argument can be chanted by the function.
- An array is stored in a contiguous chunk of memory.
- On declaration, C++ reserves enough memory to hold the indexed variables that make up the array.
- C++ does not remember the addresses of the indexed variables, but it knows how to find them given the array address and the index.
- If score[3] is needed, C++ knows that score[3] is 3 int variables past score[0]. To get score[3], C++ adds 3 * bytes_in_an_int.
- Viewed this way, an array is three things:
  1) an address of the start of the array in memory
  2) a type, which tells how big each indexed variable is
  3) the array size, which tells number of indexed variables.
- **The array argument tells the caller only the address and type, but not the size of the array.**

Display 10.4 Function with an Array Parameter

```
void fill_up(int a[ ], int size);
//Precondition: size is the declared size of the array a.The user will type in size integers.
//Postcondition: The array a is filled with size integers from the keyboard.

void fill_up(int a[ ], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

Using the const parameter modifier

- Array parameters allow the function to change any value stored in the array.
- Frequently, this, as in Display 10.4, is the intent of the function.
- Sometimes this is definitely not the case. Sometimes we want to avoid changing an array parameter.
- Example: const prevents inadvertent changing of array a
-

```
void show_the_world( const int a[ ], int size_of_a)
{
    cout << "Array values are:\n";
    for ( int i = 0; i < size_of_a; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

Array Formal Parameters and Arguments

An argument to a function may be an entire array, but an argument for an entire array is neither a call-by-value argument nor a call-by-reference argument. It is a new kind of argument known as an **array argument.** When an array argument is plugged in for an **array parameter,** all that is given to the function is the address in memory of the first indexed variable of the array argument (the one indexed by 0). The array argument does not tell the function the size of the array. Therefore, when you have an array parameter to a function, *you normally must also have another formal parameter of type* `int` *that gives the size of the array* (as in the example below).

An array argument is like a call-by-reference argument in the following way: if the function body changes the array parameter, then when the function is called, that change is actually made to the array argument. Thus, a function can change the values of an array argument (that is, can change the values of its indexed variables).

The syntax for a function prototype with an array parameter is:

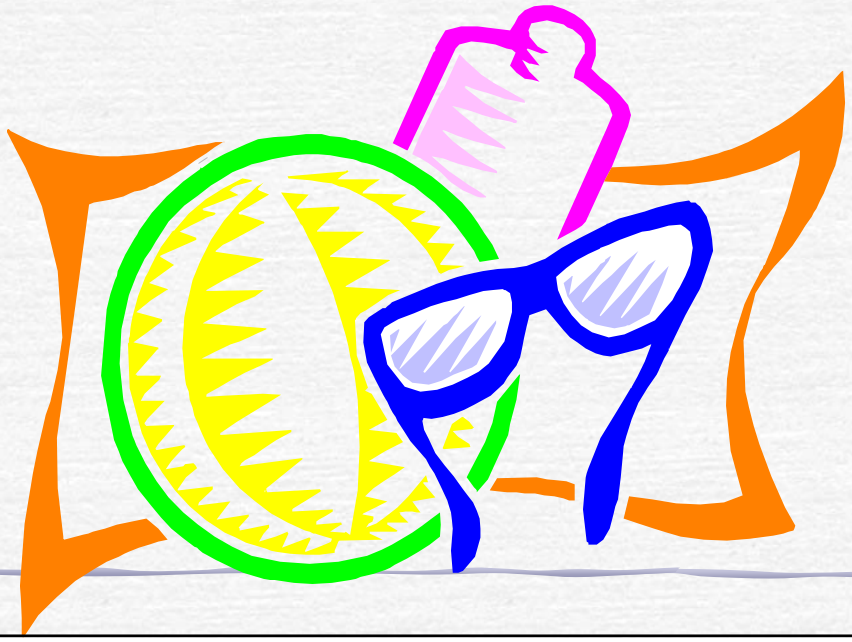
Syntax:

Type_Returned Function_Name(. . . , *Base_Type Array_Name*[], . . .);

Example:

```
void sum_array(double& sum, double a[], int size);
```

Take a break!



Reading Assignment

Strings:

10.5 (pg 545- 553)

11.1 (pg 569 - 586)