Basic Arrays

Chris Wild & Steven Zeil

May 28, 2013

Contents

1 Description **2**

2 Example **2**

3 Tips **3**

4 String Literals **3**

4.1 Description . . . 3

4.2 Example . . . 5

4.3 Tips . . . 6

5 Common Errors Using Arrays **6**

1 Description

- Arrays are convenient for naming and using a collection of objects of the same type.

  For example, if you wanted to find the biggest of three integers you could name each integer individually, but if you wanted to store and find the biggest of a thousand integers, naming each integer is inconvenient to say the least.

- All the objects in an array must be of the same type (e.g. integers, floats, user defined objects, or even arrays themselves)
- An array has a fixed *size* which must be known at the time the array is defined.
- Each array has a unique name and each *element* in the array has a unique *position*.
- The position is designated by an integer expression called the *array index*.
- The positions in an array start at the integer 0 (the first element is at position 0) and goes to the size of the array – 1.

  Thus an array with 3 elements has positions 0,1 and 2

- To name an element in an array, use the array name followed by the position enclosed in square brackets ("[" and "]")

2 Example

```
int someArray[3]; // this is an array of three integers.
someArray[0] = 567; // set the first element in the array
                    //   to the integer constant "567";
int someInteger = 4; // this is NOT an array - just a plain
                     //   integer with initial value "4"
someArray[1] = someInteger; // sets the second element in the
                            //   array to the value "4"
// At this point the value of the third element of the
//  array (someArray[2]) is undefined

float aVector[100]; // a vector of 100 floating point
```


```
                    //   numbers, indexed from 0 to 99
aVector[99] = 3.4; // sets the value of the last element to 3.4

char aString[20]; // an array of characters is also known as a string
aString[9] = 'z'; // sets the 10th character to 'z'
```

3 Tips

- Counting from 0 instead of 1 is strange at first and is the cause of errors.
- Why count from 0? think about this - how many single digits numbers are there in the decimal system?

  Answer: 10, the digits 0 through 9.

  If you start counting at 1 the last number would be "10" which requires two digits (thus more memory) then if you started counting at 0 and went to 9 for the 10th digit

  0 is the first digit, 9 is the 10th digit
- The size of an array must be a constant (so that the compiler can known how much memory to allocate), however it can be any (practical) value. It is recommended that you define an integer constant and use that to define your arrays. - This allows the size of the array to change easily by changing the value of the constant. You will see more examples of this later on which better motivate the recommendation.

  ```
  const int SIZE_OF_SOME_ARRAY = 3;
  int   someArray[SIZE_OF_SOME_ARRAY]; // defines an array of three elements
  ```
- Arrays are prone to several limitations and can be the source of errors in logic if misused. For more details check here

4 String Literals

4.1 Description

- When we write a "string" inside quotes, it is called a *string literal*, e.g.,

```
"This is a string literal"
```

Oddly enough, this is not a `std::string`. String literals are actually done as *null-terminated character arrays*.

- "character arrays", because they are simply arrays of `char`
- "null-terminated", meaning that the final character in the string is an ASCII NUL character (the `char` of value zero). This character is "invisible" when printed, so it does not affect the visible appearance of the string. But when our code finds that zero value in the array, it knows that it has reached the end of the string.

- Strings that have been stored into a null-terminated aray are variously referred to as *char arrays* and *C-strings*. (The latter refers to the fact that this style of storage for strings is inherited from C++'s "parent" language, C.)

  Sadly, lots of people (including programmers and textbook authors who should know better) get pretty lax on this point, and frequently use the term "string" ambiguously to refer to both std::string and C-strings.

- A *string literal* is of type "`const char *`".

  (We have not yet introduced the * types in C++. For now it's enough to know that `char*` is pretty much the same as `char[]` and the "const" means that you can look at the individual characters inside a string literal, but you aren't allowed to change them.)

- You can input and output C-strings using the standard insertion "«" and extraction "»" operators on `cin` and `cout`.
- When outputting a C-string, the insertion operator stops at the *null termination* character
- When inputting a C-string, the extraction operator stops at the first whitespace character (e.g. blank, tab, new line)
- To include a double quote in a *string literal* put a backslash character '\' before the double quote. This is called *escaping* the character.
- Other escape sequences include:

character desired	escape sequence	description
"	\"	double quote
\	\\	backslash
null character	\0	Used as **null termination** character
tab	\t	tab character
new line	\n	new line character

4.2 Example

```
char name[4] = "Pam"; // need 4 characters – don't forget the
                      //  null termination character implicit at
                      //  the end of a string literal
name[0] = 'S'; // changes name to "Sam"
cout << name; // prints out the string "Sam"

char anotherName[6]; // unused array
anotherName[0] = name[0]; // copies the character 'S' to anotherName
anotherName[1] = name[1]; // copies the character 'a' to anotherName
anotherName[2] = name[2]; // copies the character 'm' to anotherName
anotherName[3] = name[3]; // copies the character '\0' (the null
                          //   termination character)  to anotherName
cout << anotherName; // prints out the string "Sam"
cin >> name; // reads character up to the first
             //    white space, puts in name and adds null termination character
//// NOTE: assumes user types less than 4 characters – otherwise
//        will overflow the array and lead to possible programming errors.
//// This is one of the reasons to avoid character arrays and to use
//     the  "string" class.
```


4.3 Tips

- As a general rule, use C-strings/character arrays when you want to type a specific string within your code and to use it without modification. When you want to change strings, modify them, or pull pieces out of them, use a `std::string`.
- Remember to allocate an extra space for the null character when defining `char` arrays that hold null terminated strings
- There is an older C string library ("`<cstring>`") for operating on character arrays to perform common string manipulation operations
- Input of C-strings is tricky because the compiler and run time system does not check to see if there is enough room for the input.

5 Common Errors Using Arrays

- There are several cases where array variables behave differently than regular C++ variables. For instance:
  - You cannot use an array in an assignment statement (you can use an array element though)
  - You cannot return an array as the return value of a function (you can return a pointer to an array)
- The size of an array is not an inherent property of it. When you pass an array to a function, the function in general has no idea how long it is.
  - This is why C-strings (which are `char` arrays) are typically null terminated. The null termination character is one way to know where the string ends.
- Null termination does not work for other kinds of arrays. "0" is a "real" value typically used for purposes other than signaling the end of the array.
- Because the compiler and C++ run-time system do not know the size of an array, it is possible to attempt to access an element which is not in the array (by using an incorrect index value.).

  Accessing elements outside the array can lead to unpredictable results and is a *serious error in logic*.

- When reading in an array, it is possible to overwrite memory that does not belong to the array.

Example: Common Errors

```
// the following statements are not allowed and will cause compiler errors

int a[10], b[10];

a = b; // cannot assign an entire array - visual c++ gives left operand not a l-value
a[3] = b[3]; // this is OK

int [ ] myFunction( ); // cannot return an entire array
int* myFunction( ); // this is OK
```


Example: Errors in Reading Arrays

```
// Examples in reading input
   char someString[11]; // holds up to 10 characters plus the null character

   cin >> someString; // reads next bunch of non-whitespace characters into somestring
                      // inserts null character are end
                      // assumes that this will be 10 or less characters

   cin.getline(someString,11); // reads at most 10 characters or until
                               //   end of line
        // inserts null character at end
        // if the line contains 9 or less characters, the new line
        //    character is removed
        // if the line contains more than 9 characters, the extra
        //    characters (if any) and the newline are kept.

   cin.get(someString,11);  // reads at most 10 characters or until end of line
        // inserts null character at end
        // Unlike getline above, the new line character is never removed.

   cin.get(someString,11,'#'); // reads at most 10 characters or until
                               //   the character '#' is found


// to handle getting one line of input, even if it is too long
// ignoring the extra characters (if any)
   cin.get(someString, 11);
   cin.ignore(200,'\n'); // ignore up to 200 characters but stop at
                         //    the first newline is less than 200
              // assumes that there will be less than 200 characters
              //    extra on this line.
```