Chapter

20

Arrays & Functions

You cannot pass an array by value to a function as you can a `vector`. You also **cannot pass an array by reference to a function**. Does that mean that we can't write functions that work with arrays?

No, not at all! Instead, we need to learn about a new way of passing parameters: **pass by address**. But first, why even use arrays if `vectors` are so much more convenient?

Why use an array instead of a `vector`?

- `vector` **elements** are always **allocated on the heap**
- Arrays, may be **allocated on the stack**, **static area**, **or the heap**. This avoids performance issues that arise with dynamic memory.
- Arrays often have **higher performance** and **take less memory**.
- Arrays are generally used for **systems programming** (operating systems)
- Array performance is **deterministic**; for this reason, arrays are normally used for **embedded programs** that must run for long periods of time.

In short, arrays are usually faster and may take less memory than dynamic library types like `string` and `vector`. Using arrays in C++ is programming **as the CPU sees it**.

Pass by Address

Recall that an array name **is an address**, which you may store inside a pointer.

```
int array[5];
int *p = array;
```

This is the secret to writing functions that process arrays:

- Create a function **with a pointer as a parameter**. You **may** declare this pointer as `int a[]`, indicating that you **intend** to initialize it with the address of the first element in an array.
- **Call the function**, **supplying the name of an array** as the argument.

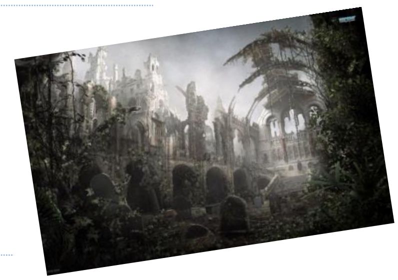
Here are two prototypes. The first uses the square brackets to declare the pointer variable **a**. The second uses the normal pointer parameter syntax. Both have identical meaning **as a parameter declaration**.

```
int aSum(const int a[], size_t size);
int aSum(const int *a, size_t size);
```

With "pointer notation", the star comes **before** the name, while with "array notation", the **brackets come after**. A common error, for Java programmers moving to C++, is to write the prototype like this, which is a syntax error:

```
int aSum(const int[] a, size_t size);
```

Decaying Arrays



When you pass an array to a function, we say that the array **"decays to a pointer"**. This is similar to what happens with primitive types in this case:

```
int n = 3.14;
```

The **int** variable **n** cannot store the fractional portion of **3.14**, so it **truncates the number** and stores **3**. When you pass an array name to a function, and it is converted into a pointer, it also **loses certain information**; specifically, it **loses the ability to determine the allocated size of the array**. That means we must **calculate an array size when the array is created**, and then **supply it when calling the function**.

Array Sharing and Const

Arrays passed to a function act **as if the array was passed by reference**. That **can be dangerous**, because the function may inadvertently modify the caller's argument.

```
for (size_t i = 0; i < len; ++i)
{
    sum += a[i];
    a[i] = sum;
}
```

This function is **intended to sum all the elements** in an array. If you were distracted and inadvertently added the highlighted line, perhaps copying the code from some other part of the program., the function would still produce the correct sum, but mistakenly **destroy the values in every array passed to it**.

Not a good thing. To fix, you use the same technique you previously applied:

- If a function **intends to modify** the array (initialization, shifting, sorting, etc.) then **do not use** `const` in front of the formal parameter. (Since you are passing by address, **you will never use &**.)
- If a function **does not intend to modify the array** (counting, summing, printing, etc.) then **always** use `const` in front of the parameter.

```
double average(const int a[], size_t len);
```

Array Loops in Functions

There are several ways to use loops to traverse an array.

1. Calculate the **number of elements in the array** and use that as a limit on a traditional counter-controlled `for` or `while` loop.
2. Use a **sentinel value** stored in the array to mark its end.
3. Use a pair of pointers: one to the first element in the array, and one to the address right past the end of the array. These are **iterator-based** loops.
4. Use the C++ 11 **range-based** `for` loop.

Inside a function, **only the first three are meaningful**. You **cannot** use the range-based `for` loop on an array name after it has decayed to a pointer.

For the first technique, **supply a second parameter** indicating the number of elements in the array. Often, this parameter will be of `size_t`, initialized with the `sizeof` "trick".

C-style strings use a special sentinel to mark the end of the array. For the algorithms from the standard library, you'll use the third technique, passing a pair of pointers.

Iterator Loops

Another iteration approach is to pass the address of the array's first element and the address of an imaginary element that is **just past the end of the array**. This is known as the **range-based** or **iterator** approach to passing array parameters. Click this link to run the function shown below.

```
double sum(const int * beg, const int * end)
{
    double result = 0.0;
    while (beg != end) { result += *beg++; }
    return result;
}
```

In the **iterator-based** approach the **caller** passes the array (the address of the element at index zero) and, the address of the imaginary element **just past the end**.

What is **beg++*?

Knowing pointer arithmetic helps you understand one of the **most common idiomatic constructions** in C++, the expression `*beg++` on Line 4.

- The `*` operator and the `++` operator **compete** for the operand `beg`. Because unary operators in C++ are **right-associative**, the `++` takes precedence over the `*`. The compiler interprets this as: `*(beg++)`
- The **postfix** `++` operator increments the value of `beg` but returns the value that `beg` **had prior to the increment operation**. Since `beg` is a pointer, the increment operation uses pointer arithmetic; adding `1` to the value of `beg` creates a pointer to the next element in the array.
- If `beg` originally pointed to `a[0]`, the increment causes it to point to `a[1]`. The value that is used for dereferencing `*`, is the address value it contained **before** the increment.

Thus, the expression `*beg++` has the following meaning in English:

> *Dereference the pointer **beg** and return as an lvalue the object to which it currently points. As a side effect, increment the value of **beg** so that, if the original lvalue was an element in an array, the new value of **beg** points to the next element in that array.*

The begin and end Functions

In C++ 11 (and later), use the library functions `begin()` and `end()`:

```
int a[] = {1, 3, 5, 19, 22, 12};
cout << "sum(a)->" << sum(begin(a), end(a)) << endl;
```

As you can see, the caller passes two pointers:

- `begin(a)` points to the first element (at index `0`) of the array, and
- `end(a)` points to the element just past the end of the array.

You can combine this with pointer arithmetic to process only a slice of an array. For instance, `sum(begin(a) + 1, end(a) - 1)` will process all but the first and last elements.

> *Of course, **you have to be careful**. If the **begin** pointer passed to the function is greater than the end pointer, you'll have an endless loop.*

Arrays & Algorithms

When working with arrays, there are a number of fundamental algorithms that you should memorize. You want these to become "second nature" so that you can pull them out of your toolbox when needed. These are:

- **Counting** for a match
- **Cumulative** algorithms
- **Extreme** values
- **Linear and binary search**
- **Adjacent elements** and the **fencepost** algorithm

Let's start with counting. To count all of the elements that **match a condition**:

```
Counter <- 0
For each element in the array
   If element matches condition
      Counter <- Counter + 1
```

Here's a **traditional implementation** of this that counts for exact matches:

```
int aCount(const int a[], size_t len, int value)
{
    int counter = 0;
    for (size_t i = 0; i < len; ++i)
        if (a[i] == value)
            counter++;
    return counter;
}
```

Using the Standard Algorithms

Using the standard library, we can just call `count(cbegin(a)`, `cend(a)`, `value)`. Could we do this from inside `aCount()`? NO. The parameter `a[]` is not really an array in `aCount()`. It has already **decayed to a pointer**.

However, with arrays, the functions `cbegin()` and `cend()` return **regular pointers**, not special kinds of iterators. Thus, we could replace the body of the `aCount()` function with:

```
return count(a, a + len, value);
```

Cumulative Algorithms

Cumulative algorithms, sum, average, standard deviation, and so on, visit each element and add, multiply or otherwise process it. Here is a function which adds all of the **even numbers** in an array named `a`:

- The array `a` is `const`, since the elements won't be changed.
- The accumulator `sum` is `double` so it doesn't overflow
- Only the **even** elements (those where the `n % 2 == 0`) are added.

```
double addEvens(const int a[], size_t len)
{
    double sum{0};
    for (size_t i = 0; i < len; ++i)
    {
        if (a[i] % 2 == 0) { sum += a[i]; }
    }
    return sum;
}
```

Using the Standard Library

In the header `<numeric>` is a function that implements the accumulation algorithm. Unsurprisingly, it is named `accumulate()`:

```
cout << accumulate(cbegin(a), cend(a), 0.0) << endl;
```

The third argument to `accumulate()` specifies both the type and starting value for the **accumulator** that is used in the algorithm. You can also pass a **fourth argument** that specifies the **binary operation** which should take place each time an element is visited.

```
int a[] = {1, 2, 3, 4, 5};
auto product = accumulate(cbegin(a), cend(a), // range
        1, // initial value
        [] (int e1, int e2) { return e1 * e2; });
```

This returns the product of `1 * a[0] * a[1] * a[2]`, etc. Notice that the initial value in this case must be `1`; if you used `0`, then the product would be `0` because `0 *` anything is `0`.

While there is no `accumulate_if()` function which takes a lambda, like `count_if()`, you can use the binary operation argument to **filter** the items you wish to process.

This call to **accumulate()** sums the even numbers in a range, similar to the **addEvens()** function shown earlier in this section.

```
int a[] = {1, 2, 3, 4, 5};
auto evens = accumulate(cbegin(a), cend(a), 0.0,
        [] (int e1, int e2) {
            return e2 % 2 == 0 ? e1 + e2 : e1;
        });
```

It is, arguably, not as clear or easy to read as the loop version.

Extreme Values

The largest (or smallest) value in a collection is called an **extreme value**. Here is the algorithm for finding the largest:

```
largest <- first
For each remaining element
   If element > largest Then
      largest <- element
Return largest.
```

The algorithm for finding the smallest is similar. What if there is no first element? Then there is no largest or smallest element; **it is an error condition**.

For many algorithms, you not only want to know the largest (or smallest) value, but **where it is**, either as an index or as a pointer. Click this link to look at both. We'll discuss the two functions in the next section.

Returning a Pointer

The **biggest()** function returns a **pointer to the largest item** in the array. We don't want to allow the element to change, and we don't want the pointer to be used to modify other elements, so the return type is **const double*** **const**.

When you **call biggest()**, **dereference** the returned pointer to get the value.

```
cout << *(biggest(da, 5)) << endl;
```

Let's **apply the steps** in the extreme values algorithm to this problem.

1. Save the first value as the largest. You need two variables to do this:

```
const double *p = a;
double largest = *p;
```

2. Now, loop through each **remaining element** like this:

```
for (size_t i = 1; i < n; ++i)
```

3. Each time through the loop, check to see if the current element is larger than the saved value, and, if so, update the saved values. Because you want to return a pointer, you'll need to update both `largest` and `p`. Note the use of the address operator.

```
if (a[i] > largest)
{
    p = &a[i];
    largest = a[i];
}
```

4. Finally, simply return the pointer `p`.

This is the same scheme used by the standard library algorithms `min_element()` and `max_element()`. When called using arrays, they return a pointer in exactly this manner.

The second problem, `smallest()`, returns an index instead of a pointer.

The Fencepost Algorithm

To print an array, you want to surround all of the elements with brackets (`[]`), and **separate the elements** from each other by a comma. This is called the **fencepost algorithm**. Here's the algorithm, assuming you have selected values for the opening and closing delimiters as well as the size.

```
Print opening delimiter (, {, [, etc.
If array size is > 0 Then
   Print first element
   For every remaining element
      Print separator
      Print element
Print closing delimiter
```

Click on this link to see this algorithm implemented as a template function.

Fencepost in Reverse

What if you want to use the same algorithm, but **print the elements in reverse** order? That's a little more difficult. Here is an "obvious" algorithm **which does not work:**

```
cout << a[len - 1];
for (size_t i = len - 2; i >= 0; --i)
{
    cout << separator << a[i];
}
```

The loop variable is `size_t`, so as soon as you print `a[0]` and decrement the control variable `i`, instead of becoming `-1`, it "wraps around" and becomes the **largest possible unsigned number**. Array subscripts are **not range checked**, so the loop prints at larger and larger indexes until the program crashes.

This **works correctly**. Notice the extra `if` statement and the highlighted changes:

```
if (len > 0)
{
    cout << a[len - 1];
    for (size_t i = len - 1; i > 0; --i)
    {
        cout << separator << a[i - 1];
    }
}
```

Searching Algorithms

One common operation is searching for a particular element in an array. Consider this function template to search an array:

```
template <typename T>
int aFind(const T& key, const T a[], size_t len);
```

If `key` is found is found, `aFind()` returns its index. If not, it returns `-1`.

> *The return type must be `int` so it can return the negative value on failure.*

Linear Search

If the array is unordered, then `aFind()` **must** check each element **sequentially**, until it finds a match or runs out of elements. This is called a linear-search. Here is an implementation of this function template which uses this algorithm.

```
template <typename T>
int aFind(const T& key, const T a[], size_t len)
{
    for (size_t i = 0; i < len; ++i)
        if (a[i] == key) return i;
    return -1;  // not found
}
```

In the standard library, the `find()` function does this for you. Like all of other **STL** algorithms, `find()` takes a **begin**-**end** range and a value to search for; it returns a pointer (or iterator) to the **first element found**. Here's an example:

```
int a[] = {1, 2, 3, 4, 5};
auto itr = find(begin(a), end(a), 4); // find value 4
if (itr == end(a))
    cout << "Not found" << endl;
else
    cout << "Index->" << distance(itr, begin(a)) << endl;
```

If `find()` **doesn't find** what it is looking for, then it points **to the end of the range.** To convert the iterator returned to an **index value**, use the `std::distance()` function instead of plain address arithmetic and pointer difference.

> *Note: in C++11 use the* ***`begin()`*** *and* ***`end()`*** *functions. In C++14 and later, you should use* ***`cbegin()`*** *and* ***`cend()`*** *when you want a constant iterator. Unfortunately, the CPP-Shell does not do this correctly.*

Like `count_if()`, there is a `find_if()` variant that takes a **lambda** as a third argument, so you can supply a **condition** instead of a value. You could use `find()` **inside** your `aFind()` function template like this example:

```
template <typename T>
int aFind(const T& key, const T a[], size_t len)
{
    auto itr = find(a, a + len, key);
    if (itr == a + len) return -1;  // not found
    return itr - a;
}
```

Finding the Last Match

To find the **last match** in a `vector` or `string`, you use the member function `rfind()`. There is no `rfind()` function among the standard algorithms, though. However, you can write your own easily. Here is one version which uses a **forward loop**:

```
template <typename T>
int findLast(const T& key, const T a[], size_t len)
{
    int pos = -1;
    for (size_t i = 0; i < len; ++i)
        if (a[i] == key) pos = i;
    return pos;
}
```

This is relatively inefficient, since you must search through the entire array to find the last match.

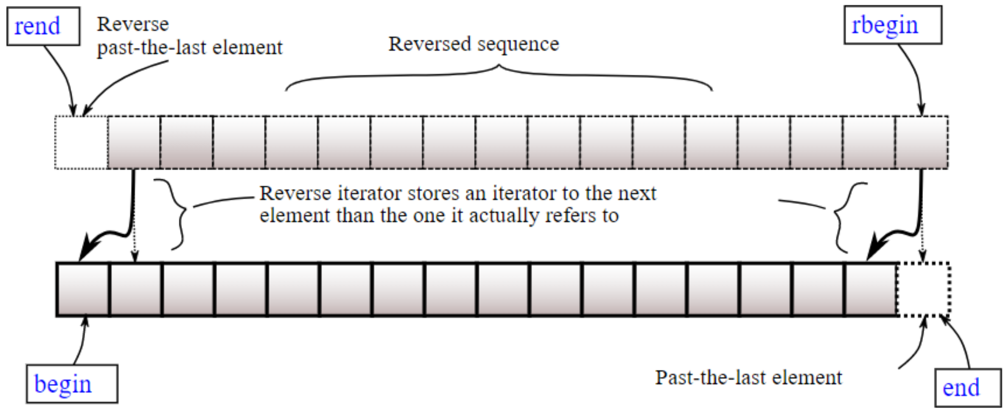
Make it more efficient by searching towards the front as this example does:

```
template <typename T>
int findLast(const T& key, const T a[], size_t len)
{
    int last = len - 1;  // convert size_t to int
    for (int i = last; i >= 0; --i)
        if (a[i] == key) return i;
    return -1;
}
```

Reverse Iterators

With the standard library, you can find the last match by using the `find()` function along with a **reverse iterator**, which starts at the last element and moves towards the front. Use `rbegin(a)` and `rend(a)` to get a reverse iterator for an array.

Here's a picture showing you the difference between these two types of iterators. In C++ 14, the library added constant reverse iterators `crbegin()` and `crend()`.



Here's a final example, showing the use of **`find()`** with reverse iterators:

```
const int a[] = {1, 2, 3, 4, 5, 4, 3, 2, 1};
auto itr = find(rbegin(a), rend(a), 3);
if (itr == rend(a))
    cout << "Not found" << endl;
else
    cout << "Last 3 found at position: "
         << distance(itr, rend(a)) -1 << endl;
```

Note:

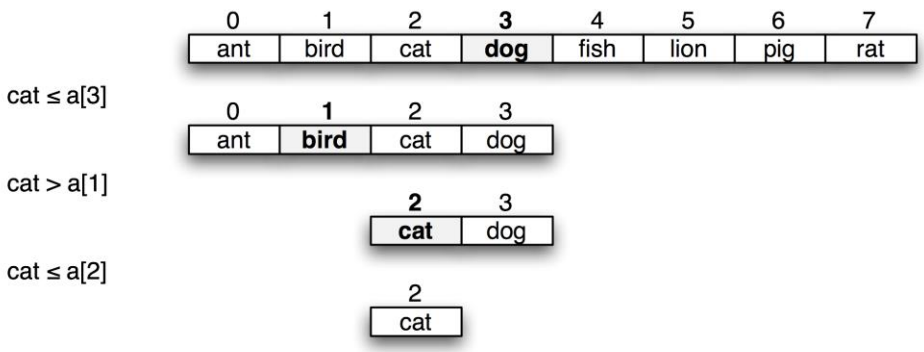
- To search backwards you pass a **pair of reverse iterators**. **`rbegin(a)`** points the last element, and **`rend(a)`** points just before the first.
- If you **don't find** the searched-for value, then the iterator will point to **`rend()`**
- If you do find it, then **the index is the distance – 1**, because the end is not the first element, but an imaginary element appearing before the first.

Binary Search

In Computer Science we say that **linear search** is an **O(n) algorithm** (**on the order of n**) because the time to find your element increases in a linear manner as the number of elements in the array (**n**) increases. (This is known as **Big O notation**).

On the other hand, if you **know the elements are in alphabetical order**, (**sorted**), you can adopt a more efficient approach: **divide the array in half** and compare the **key** you're trying to find (**cat** in the illustration) against the element closest to the middle, using the order defined by ASCII, which is called **lexicographic order**.

If the **key** you're looking for **precedes** the middle element, then the **key**—if it exists at all—**must be** in the **first half**. If the **key** follows the middle element in alphabetic order, you only need to look at the elements **in the second half**.



Because you can discard half the possible elements at each step in the process, it is much more efficient than linear search. Binary-search is a **divide-and-conquer** algorithm which is **naturally recursive**.

Here's an implementation, called **`bFind()`**, which uses **binary search**:

```
int bFind(const string& key, const string a[],
          int first, int last)
{
    if (last < first) return -1; // not found
    int midPos = (first + last) / 2;
    if (a[midPos] == key) return midPos; // found
    if (key < a[midPos]) // look in left side
    {
        return bFind(key, a, first, midPos - 1);
    }
    else // otherwise look in the right
    {
        return bFind(key, a, midPos + 1, last);
    }
}
```

Here's an example calling **`bFind()`** with an array:

```
string a[] = {"alpha", "beta", "gamma", "zeta"};
int pos = bFind("gamma", a, 0, 3);
```

Using the Standard Library

To sort an array, you can use the **`std::sort()`** function, and, then, to see if a value is found in the array, use **`std::binary_search()`** as in this example:

```
int a[] = {1, 2, 3, 4, 5, 4, 3, 2, 1};
sort(begin(a), end(a));
if (binary_search(cbegin(a), cend(a), 5))
    cout << "5 is found" << endl;
else
    cout << "5 is not found" << endl;
```

Note that **`sort()`** only works on non-**`const`** arrays, and **`binary_search()`** returns a Boolean, not the position where the element is found, like your **`bFind()`** function did.

To find the position where the value would be found, using binary search, call the **`lower_bound()`** library function. In this example, I've used it to reimplement **`bFind()`**:

```
template <typename T>
int bFind(const T& key, const T a[], size_t len)
{
    auto itr = lower_bound(a, a + len, key);
    if (itr == a + len || *itr != key) return -1;
    return distance(a, itr);
}
```

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **`make submit`**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.