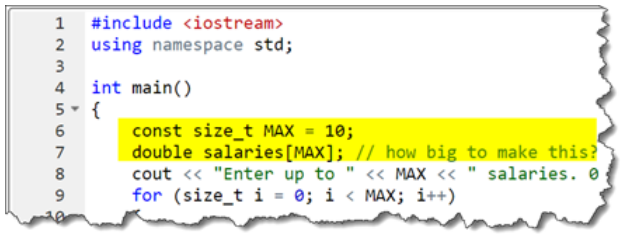
An Array Example

Take a look at this example. How many salaries do I need? Here I've planned on `10`, but if I **need** more, there is no `push_back()`, as there is with `vector`, which would allow expansion. This program is **limited** to a maximum of `10` salaries.



```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      const size_t MAX = 10;
7      double salaries[MAX]; // how big to make this?
8      cout << "Enter up to " << MAX << " salaries. 0
9      for (size_t i = 0; i < MAX; i++)
10
```

The loop itself can end in one of three ways:

- The user can enter `10` salaries and, because the array is full, the loop will end. **All** of the elements in the array **will be used**.
- The user can enter a `0` as a salary (**the sentinel**), and the loop will terminate. Only **some** of the elements **will be used** in the array.
- The user can enter a non-numeric value such as the word `"quit"` and the `cin` object will enter the **fail state** when trying to read the next salary. The *`if`* statement inside the loop checks for this and exits when this occurs.

Once you exit the loop, you **don't know which** of these occurred. Even worse, you can't tell how many elements are actually valid, and which are unused. Let's fix that.

Appending Elements

When you don't know how large an array should be when you write the code, then you should **plan for the "worst case"**, and declare an array that you know is larger than you could **ever possibly need**. Then, only use part of it.

Go ahead and modify the example on the previous page. We want to **allow the user to input any number of values**, (or at least a very large number), instead of automatically filling the array with `10` values.

1. Define a constant that indicates **the maximum number of elements** used, (like `100`), and use that constant in the declaration of the array.
2. Create a **separate variable** to track the **effective size** of the array. The names `size` and `capacity` are typically used for these variables.

3. Write an input loop using while that checks the **necessary bounds** condition, `while(size < capacity)...`. This ensures that the loop **never** overfills the array.
4. Use the **loop-and-a-half idiom** to leave the loop whenever the **sentinel** value (`0`) is entered, or, when `cin` **enters the failed state** from invalid input.
5. Store the value into the array, and update the `size` variable so that the next number entered will be placed **in the next element** of the array.

Traversing the Array

When you traverse a partially-filled array, your algorithms must use the effective size as your loop bounds. Here's an example which **computes the highest salary** in the array.

```
double highest{0.0};
if (size > 0)
{
    highest = salaries[0];
    for (size_t i = 1; i < size; ++i) {
        if (salaries[i] > highest) {
            highest = salaries[i];
        }
    }
}
```

Note that you can **only** inspect the elements with an index less than `size`, because the remaining elements have never been set, so their contents are undefined.

Inserting Elements

Instead of adding items to the end of the "unoccupied" section of numbers in an array, suppose you placed each number **into its correct (ordered) position** instead, like this:

The array shown here is partially filled, and the next number, `3.25`, has been input by the user. To put the number in its correct location you need to:

1. Locate the **first number** that is **larger** than the number you're going to insert into the array. Here, that's the number `7.1`.
2. Before you can store `3.25`, the `7.1` needs to be moved to the right. But first, the number occupying that spot must be moved, and so on.
3. After all of the existing numbers have been moved to the right, come back and **store the new number** in the spot that's been opened up.

This only makes sense **when used with partially filled arrays**; if you try to insert an element into a completely filled array, then the last element in the array will be lost when the previous items are moved to make room for the inserted item.

Applying the Algorithm

Let's look at a practical example of this algorithm. **The `insert()` function at** the top of the program has not been completed, so we're going to go ahead and do that.

1. **Find the position** where the new element **should be** inserted:

   ```
   size_t pos = 0;
   while (pos < size && a[pos] < value)
   {
       ++pos;
   }
   ```

   The variable (**`pos`**) is initialized to **`0`**. After the loop, it will contain the location where the new element should be inserted.

   If there are no elements larger than the number you are inserting, **`pos`** will contain the same value as **`size`**, and the number will then be added at the end of the array, which is what you want.

2. **Move the existing elements**. Before you can store value in the array, you must move the existing elements out of the way (to the right), to "open up a hole" for the new value.

   ```
   for (size_t i = size; i > pos; --i)
   {
       a[i] = a[i - 1];
   }
   ```

   You must start **at the end of the array**, traversing to the left, until you reach the location where you intend to insert the new element, so you don't overwrite data.

3. **Insert the new element**. After moving, copy the new number into position, and **update the `size`**.

   ```
   a[pos] = value;
   ++size;
   ```

Removing Elements

Removing elements from a partially-filled array uses a similar algorithm. Instead of moving a portion of the array to the right, to "open up a hole", you need to move all of the elements which are **to the right** of the deleted element leftwards to "close up the gap".

Here's a small fragment of code that does that. The variable `pos` is the location of the element you want to delete:

```
--size;
for (size_t j = pos; j < size; ++j)
{
    a[j] = a[j + 1];
}
```

Before the loop, you should decrement `size` so that when you grab `a[j + 1]`, it is a valid element when the array is already full.

Removing Multiple Elements

Since, when you remove an element, the following elements are moved into the position of the deleted value, you have to be especially careful when removing multiple values from the same array. Here's an example:

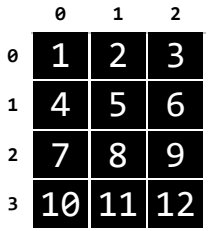
```
for (size_t i = 0; i < size; ++i)
{
    if (a[i] == 7)
    {
        --size;
        for (size_t j = i; j < size; ++j)
        {
            a[j] = a[j + 1];
        }
    }
}
```

If you run it, you'll see that it doesn't quite work. It **should** remove all of the sevens from the partially-filled array, but removes only some of them.

```
a->[1, 7, 7, 3, 7, 9, 7, 7]
a->[1, 7, 3, 9, 7]
```

As you can see, when you remove the first seven, you skip over the seven immediately following it. To fix this, make sure that you adjust the loop index, so that it **revisits the current index** whenever you remove an element. Here's the fixed version of this problem .

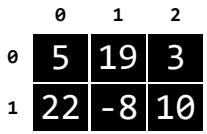
2D Arrays



A two-dimensional array (2D) is a block of memory, visualized as a table with rows and columns. In C++ this is implemented (under the hood) by creating a **1D** array whose elements are, themselves, also **1D** arrays.

Examine this code:

```
1  int a2d[2][3]; // a2d is a 2D array
2
3  a2d[0][0] = 5;
4  a2d[0][1] = 19;
5  a2d[0][2] = 3;
6  a2d[1][0] = 22;
7  a2d[1][1] = -8;
8  a2d[1][2] = 10;
```

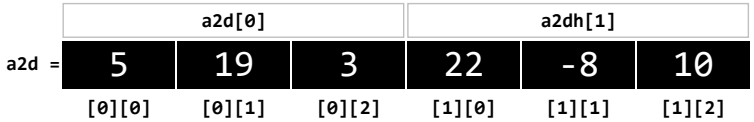


It is easiest to think of `a2d` as containing **two rows** and **three columns**: a `2 × 3` array. To access an element of a **2D** array, use **two subscripts**. The first is the **row**, the second is the **column**. Both are zero-based. On line 6, for instance, the first index (here `1`) signifies the second row and the second index (here `0`) denotes the column within the array.

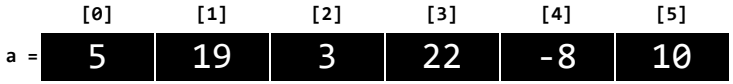
Row Major Order

While conceptually the array `a2d` contains rows and columns, physically the elements are stored linearly, with the elements of each row following the elements of the preceding row in memory.

The array **`a2d`** **actually** contains **two elements** (not 6!). Each is a one-dimensional **`int`** array of size **`3`**. This is how the compiler sees the declaration:



That means, instead of using a **2D** array, we **could** store the same elements in a **1D** array, here named **`a`**, like this:



To treat this **1D** array as a **2D** array, (as you've done with all of your image projects in this class), you need to recall the formula for **array access expressions**:

```
a[offset] = *(a + offset)
```

You can convert this to where a **2D array offset expression** like this:

```
a[row, col] = *(a + (row * row - width + col))
```

Notice how **similar** this is to **1D** pointer-address arithmetic; the only new addition is the expression **`row * row-width`** to the calculation.

2D Array Initialization

The 2D array `a2d` could be declared and initialized like this:

```
int a2d[2][3] = {
    {5, 19, 3},
    {22, -8, 10}
};
```

Each row appears is in its own set of curly braces. Because, the array is actually laid out in a linear fashion, you may **omit the inner braces** all together, but that is not as clear:

```
int a2d[2][3] = {5, 19, 3, 22, -8, 10};
```

The rules for partial initialization are similar to **1D** arrays: any uninitialized elements are **value initialized to `0`**. With embedded braces, partial initialization is on a **row-by-row** basis; if you omit them, the rows are ignored. These examples use the **same initial values**, but produce quite different results.

```
int a2d[2][3] = {
    {5},
    {22, -8}
};
```

	0	1	2
0	5	0	0
1	22	-8	0

```
int a2d[2][3] = {
    5, 22, -8
};
```

	0	1	2
0	5	22	-8
1	0	0	0

When initializing, you **may omit the first explicit dimension**, but the second dimension **`[3]` is required**. The compiler must **know how big** each element of `a2d` is.

```
int a2d[ ][3] = {
    {5, 19, 3},
    {22, -8, 10}
};
```

2D Arrays & Functions

Pass 2D arrays to functions by address, just like 1D arrays. The following function prints the contents of a `ROWS × COLS` **2D** array of `double`:

```
void print(const double m[ROWS][COLS])
{
    for (size_t row = 0; row < ROWS; ++row)
    {
        for (size_t col = 0; col < COLS; col++)
            cout << setw(5) << m[row][col];
        cout << endl;   // end of each row
    }
}
```

Of course, **this function is really quite limited** since **it can only be used** to process an array that is **exactly** `ROWS x COLS` elements in size.

You can make it a little more flexible by **omitting the first dimension**, and then passing the number of rows as a parameter. **You cannot**, however, leave off the number of columns. That must be a constant.

```
void print(const double m[][COLS], size_t nRows)
{
    for (size_t row = 0; row < nRows; ++row)
    {
      ...
    }
}
```

This inflexibility is one of the reasons that the built-in **2D** arrays are so limiting in C/C++.

An expression that uses just one subscript with a **2D** array **represents a single row** within the **2D** array. This row is itself a **1D** array. Thus, if `a2d` is a **2D** array and `i` is an integer, then the expression `a2d[i]` is a **1D** array representing row `i`.

C-Style Strings

C++ has two different "string" types; the `string` class from the standard library makes string manipulation easy, but is complex, since it uses dynamic memory. The original "string" type, inherited from the C language, is much simpler.

Though simpler, older C-strings are more **difficult to work with**. Sometimes more efficient, they are also **more error prone**, even **somewhat dangerous**. However, as C++ programmers, you can't ignore them.

Why should you dedicate **any** time to studying C-strings? There are several reasons:

- **Efficiency**. Library string objects use dynamic memory and the heap. C-strings are **built into the language**, so you don't need to link library code.
- **Legacy Code**. To **interoperate** with pure C code or older C++ code that predates the C++ `string` type.
- **Library Implementation**. You may want to **implement** your own `string` type. Knowing how to manipulate C-strings can greatly simplify this task.
- **Embedded Programming**. Programs written for embedded devices like those in your automobile or toaster, frequently use C-strings.
- **Platform O/S Programming**. For native Linux or Windows programming, you will need to use C-strings.

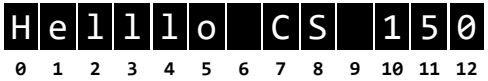
We will encounter many of these cases in the remainder of this course.

C-String Basics

The library `string` type works as if it were built into the C++ language. It uses C++ features to allow a `string` to act as a built-in type. C-strings are more primitive:

- C-strings are `char` **arrays** with a **sentinel terminator**, the `NUL` character `'\0'`.
- C-strings can be passed to functions without overhead.
- "String literals" automatically include the terminating `NUL`.

The literal ***"Hello, CS 150"*** contains `13` characters—`12` for the meaningful characters plus one extra for the terminating `NUL`. The compiler generates:



C-string functions all **assume** that this `NUL` exists; some insert it for you. Without a `NUL`, functions don't know when the string stops, either returning garbage or crashing. The length of a C-style string is **not stored explicitly**; the `NUL` serves as a sentinel, and your program loops through the characters, counting them when it needs to find the size.

> *Don't confuse `'0'` with `'\0'`. One has the ASCII value `48` and the other `0`.*

Array-based C-Strings

How you create a C-string determines where the characters are stored in memory. To copy characters **into user memory** where they can be modified, write this:

```
char s1[] = "String #1";
```

The C-string **`s1`** **contains** **exactly** **`10`** characters; the **`9`** that appear in "**`String #1`**" and the terminating **`NUL`** character. Space for these characters is **allocated on the stack or static storage area**. The **actual characters are copied** into this "user space". This declaration is shorthand for:

```
 char s1[] = {'S','t','r','i','n','g',' ','#','1','\0'};
```

Because the characters have been copied into memory that you control, you can **change them if you like** using the normal array subscripting operations.

```
s1[0] = 'C';  // OK; all characters are read-write
```

```
const size_t kLen = 1024;    // small strings
char s2[kLen] = "String #2";
```

The declaration for **`s2`** is slightly different. While the effective size of the string is also **`9`** characters, its **allocated size** is set by **`kLen`** or **`1024`** in this case. Use **`s2`** if you want to add information to the end of the string, similar to partially-filled arrays.

Pointer-based C-Strings

Pointers to `NUL`-terminated character array literals can be used as C-strings, provided you don't attempt to modify them:

```
const char *s3 = "String #3";
```

The character array itself is not copied into your user space. The characters are stored in the **static storage area** when the **program loads**. Attempting to change a character in **`s3`** is a **compiler error**, because of the **`const`**.

In C and in some older C++ code, you may see this declaration:

```
char *s4 = "String #4"
```

The declaration for **`s4`** is obsolete in modern C++, but may be found in older code (and is legal in C). The compiler will probably compile your code (**with warnings**), but **your program will probably crash** if you attempt to modify the string in any way. The portion of the static storage where string literals are stored is **effectively read-only**.