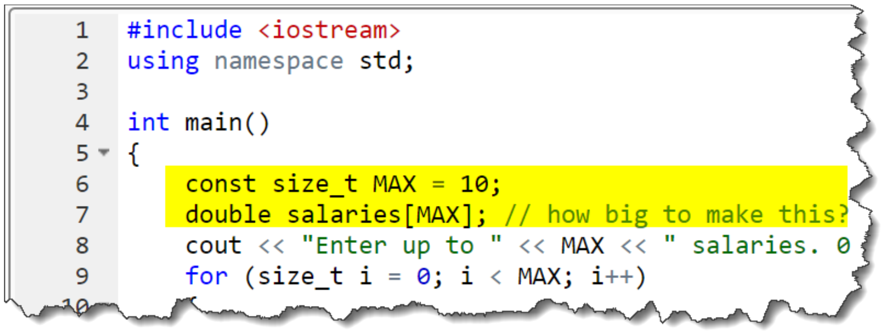
Chapter

21

Partially-Filled Arrays

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 with **`vector`**, which would allow expansion. My program is **limited** to a maximum of 10 salaries.



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

int main()
{
    const size_t MAX = 10;
    double salaries[MAX]; // how big to make this?
    cout << "Enter up to " << MAX << " salaries. 0
    for (size_t i = 0; i < MAX; i++)
```

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.

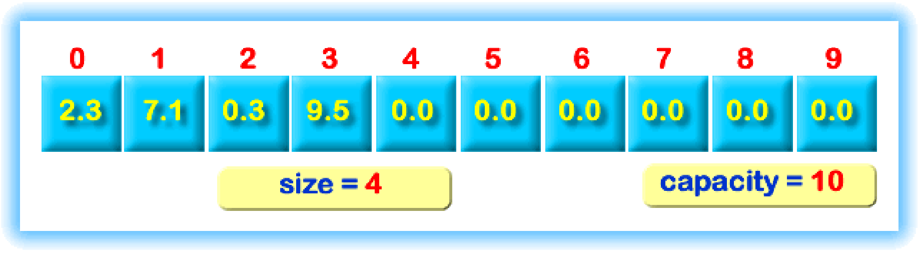
Once you exit the loop, you **don't know which** of these occurred. Even worse, you can't tell how many elements are actually valid, and which are unused. Let's fix that.

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.

To fix the deficiencies of the previous example, and to **allow the user to input any number of values**, 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**. Terms **capacity** and **size** for are commonly used for the allocated and effective sizes respectively.



3. Write an input loop using `while` that checks the **necessary bounds** condition, `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. If `cin.fail()`, then the previous read operation failed, so you can't rely on the value in your input variable.
5. Store the `salary` 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**. 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 & Deleting

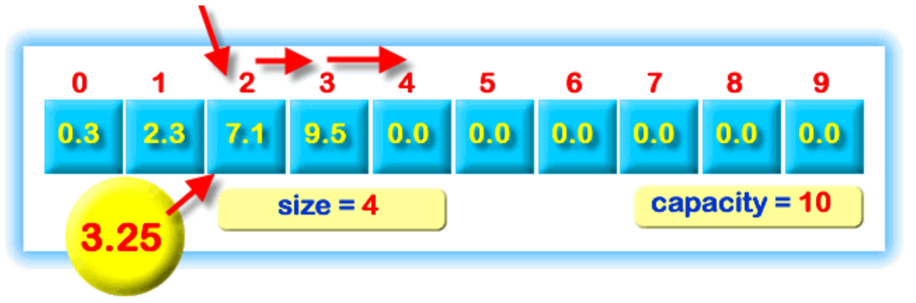
The `vector` supports **inserting an element** at an arbitrary position. Given a vector `v` and an element `e`, to insert `e` into `v` at position `n`, use the syntax:

```
v.insert(begin(v) + n, e);
```

All of the subsequent elements are shifted towards the end of the `vector` to make room for the new element. Let's do the same thing with arrays. With arrays, you need to understand the algorithm that is used.

Inserting in Order

Instead of adding items to the end of the "unoccupied" section of numbers in an array, suppose you placed each number **into its correct position** in the array.



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 then, 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 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.

Let's look at a practical example of this algorithm. The `insert()` function at the top of the program has not been completed. Go ahead and do that.

Find the Position

First. 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 set 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**, the number will then be added at the end of the array, which is what you want.

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

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

Insert the New Element

After moving, copy the new number into position, and update the size

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

Removing Elements

Removing elements 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 **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 (auto j = pos; j < size; ++j)
{
    a[j] = a[j + 1];
}
```

Before the loop, decrement **size** so that when you grab **a[j + 1],** it is a valid element.

Removing Multiple Elements

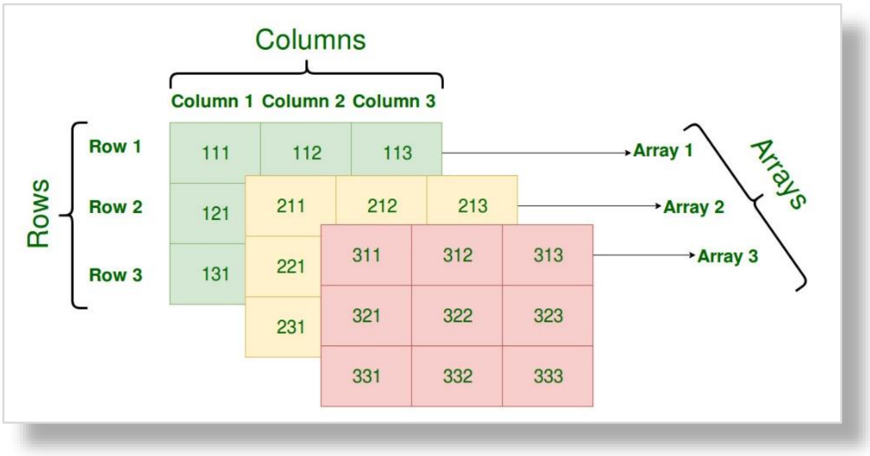
Since, when you remove an element, the following elements are moved into the position of the deleted value, you have to be especially careful when removing multiple values from the same array. Here's an example (which doesn't quite work). It **should** remove all of the sevens from the partially-filled 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 it removes an element.

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.

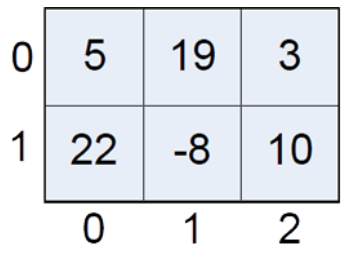


Creating 2D Arrays

Examine this code:

```
int a2d[2][3]; // a2d is a 2D array

a2d[0][0] = 5;
a2d[0][1] = 19;
a2d[0][2] = 3;
a2d[1][0] = 22;
a2d[1][1] = -8;
a2d[1][2] = 10;
```



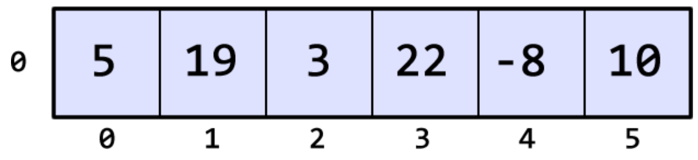
0	5	19	3
1	22	-8	10
	0	1	2

The array **a2d** contains **two elements** (not 6!). Each is a one-dimensional **int** array of size **3**. This is how the compiler sees the declaration, but it is easier 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 row and the second index (here `0`) denotes the column within the array.

Row Major Order

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. Physically we could draw the elements of `a2d` like this:



In fact, all array access expressions break down to:

```
*(address + offset)
```

where a 2D array offset expression means:

```
*(address + (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.

Initialization

The 2D array `a2d` from above **could be** declared and initialized as:

```
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 it is not as clear:

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

Partial Initialization

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	5	0	0
1	22	-8	0
	0	1	2

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

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

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

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

Processing One Row

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`.

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.