Drawbacks to Arrays

Arrays in C++ have a number of drawbacks that make them inconvenient to use:

- The size of the array must be a **fixed constant,** not a variable that you compute or read in from some input stream. If you want to read data from a file and store it in an array, you must make the array large enough to hold whatever amount of data you might ever conceivably need.
- Passing an array as a parameter to a function is inconvenient; you must also pass the size of the array as a separate parameter.
- There is no way to conveniently insert elements at the beginning or in the middle of an array. To do this, you must manually shift all of the elements over by one position to make a "hole," and then place the new element in that hole.
- Similarly, it is inconvenient to remove an element from an array; you again have to shift all of the elements after it over by one position to fill in the gap left by the missing element.
- You cannot return an array from a function.

Vectors

Many of the problems discussed on the previous slide can be worked around by using advanced techniques, but those are beyond the scope of this course.

Fortunately, many of these advanced techniques have been prepackaged into a class called a `vector`, which we can use without having to understand all of the underlying techniques.

Vectors, like strings, are a *class* in C++. Like we did with strings, we will ignore the object-oriented nature of vectors and simply note that they are another type that you can use to declare variables.

```
#include <vector>          // you must include this header

using namespace std;       // like everything else, vectors
                           // live in the std namespace
```

Advantages of Vectors

Like an array, a vector is a sequence of elements of the same type, but they provide many advantages over arrays:

- The size of a vector does not have to be a fixed constant, and it can also grow or shrink during execution. If you want to read data from a file and store it in an vector, you can just keep adding new elements as you receive them, without having to know what the largest amount would be.
- A vector always knows its own size, so passing one to a function does not require you to separately pass this information.
- Elements can be accessed by position in a vector just as they are in an array, but you can also insert and remove elements anywhere in a vector. If a “hole” needs to be made for an inserted element or closed for a removed element, the rest of the elements are automatically shifted to make or fill the hole.
- Vectors can be returned from a function easily.

Declaring a Vector

Consider the syntax for declaring an array of 20 `int`s:

```
const int SIZE = 20;
int Numbers[SIZE];
```

The corresponding declaration for a `vector` of 20 `int`s is:

```
const int SIZE = 20;
vector<int> Numbers(SIZE);
```

Unlike an array, the elements of a vector *are* initialized with appropriate default values. This means 0 for `int`s, 0.0 for `double`s, and "" for `string`s.

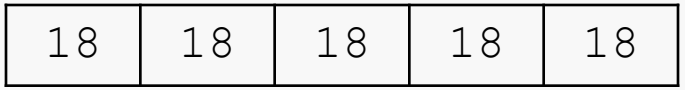
Notice that the type of elements in the vector is written in angle brackets after the name `vector`. This is a *template* in C++. Just as functions take values as parameters, some types like `vector` take other *types* as parameters. Using `vector` by itself would be incomplete; you must include the element type, and the combination of the template and its parameters form a unique type. For example, `vector<int>` and `vector<double>` are two distinct types.

Initializing a Vector

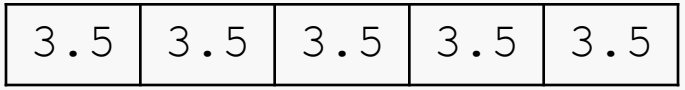
We can also pass an optional second parameter when we declare a vector, which denotes the value that should be placed in every element of the vector. The value must be of the same type as the vector's element type, or be implicitly convertible to it.

```
const int SIZE = 5;
vector<int> Numbers(SIZE);
```


```
const int SIZE = 5;
vector<int> Numbers(SIZE, 18);
```




```
const int SIZE = 5;
vector<double> Numbers(SIZE, 3.5);
```



This is considerably more convenient than writing an explicit `for` loop that initializes every element in the vector.

Accessing a Vector

Vectors provide a `size()` method that reports the number of elements currently in the vector. Accessing elements by their position in a vector uses exactly the same syntax as accessing elements of an array:

```
void PrintElements(const vector<int>& Vec) {
    for (size_t i = 0; i < Vec.size(); i++) {
        cout << "Vec[" << i << "] = " << Vec[i] << endl;
    }
}
```

We pass the vector by constant reference because the function does not need to modify it. Had we passed it by value, a copy would have been made, which is potentially costly if it has a large number of elements. If the function needed to modify the vector, we could have passed it by reference instead.

* Notice that the type of the variable is `size_t`, not `int`. `size_t` is a numeric type that represents the largest number of elements that a vector can hold. Using this instead of `int` isn't required but it prevents compiler warnings.

Range Bound Checking

- Indexing vector locations beyond the bounds of the vector can lead to the same memory errors as in arrays.
- The vector class provides a function, `at()`, to access a vector location that automatically checks that the memory is within the bounds of the vector:

```
void PrintElements(const vector<int>& Vec) {
    for (size_t i = 0; i < Vec.size(); i++) {
        cout << "Vec[" << i << "] = " << Vec.at(i) << endl;
    }
}
```

- If an index passed to the vector is outside the bounds the at function throws an exception error that can be used to debug the violation.

Vectors Without an Initial Size

Previously we saw that we can pass an initial size to the vector. When we write:

```
vector<int> V(20);
```

…this means that we can immediately access elements `V[0]` through `V[19]` but nothing beyond that, just like an array.

We can also declare a variable without an initial size:

```
vector<int> V;
```

When we do this, the vector is **empty.** It has no elements, so you can't even access `V[0]`. Calling the `size()` method will return 0. There is also a method named `empty()` that returns `true` if the vector is empty or `false` if it is not.

An empty vector doesn't seem to be very useful. Fortunately since it can grow and shrink arbitrarily, we have other ways of inserting data into it.

Resizing a Vector Explicitly

`V.resize(int N)`

Resizes the vector `V` so that it contains exactly `N` elements. If `V` previously had less than `N` elements (thus making it larger), all of the old elements will remain as they were. If `V` previously had more than `N` elements (thus shrinking it), any elements at index `N` or greater will be removed.

`V.clear()`

Removes all of the elements from the vector and sets its size to 0.

```
vector<int> V;          // V starts out empty
V.resize(10);           // can now access V[0]...V[9]
V[9] = 35;
V.resize(9);            // can now only access up to V[8]
                        // The 35 that we set previously no
                        // longer exists

V.clear();              // Now V is empty again
bool e = V.empty();     // e == true
```

Adding Items to a Vector

`V.push_back(`*`Type`* `element)`

Adds an element to the end of the vector, increasing its size by 1. The Type of the argument is the element type used when declaring the vector.

```
vector<int> V;          // V starts out empty
int size = V.size();    // size == 0

V.push_back(10);        // Insert 10 onto the end of V
size = V.size();        // size == 1

V.push_back(20);        // Insert 20 onto the end of V
size = V.size();        // size == 2

int Foo = V[0] + V[1];
```

Other methods exist to insert elements at the beginning or in the middle of a vector, and to remove items, but we need to discuss another concept first.

Vectors provide the concept of an *iterator,* which is another form of representing the position of an element in the sequence.

Recall that elements in a vector of size N are identified by their *index*, an integer between 0 and $N - 1$. Iterators implement a similar concept.

The reason for this distinction is that the standard library provides other *containers* as well. We will see some other containers later, but in short, a container is an object that can hold some set or sequence of other objects or values.

Some containers, like `vector`, allow us to access their elements by numerical position. Other containers (like `list`, `set`, and `map`) do not. Regardless of these differences, all containers provide iterators that give us the ability to examine every element in the container, and to identify some abstract notion of an element's location in the container. At the very least, iterators allow us to start at the beginning of a container and advance from one element to the next until we have covered them all and reached the end.

Obtaining an Iterator

`vector<`*`Type`*`>::iterator`

The type of the iterator object provided by a vector.

`vector<`*`Type`*`>::const_iterator`

If the vector you want to iterate over is const, or a const reference to a vector, then you must use the iterator of type `const_iterator`. Using `iterator` will result in mysterious compiler errors.

`V.begin()`

Returns an iterator that represents the first element in the vector (that is, it represents `V[0]`).

`V.end()`

Returns an iterator that represents **one element past the last element** in the collection (that is, it represents a position just after `V[N - 1]`).

Iterator Operations

`Iter++` or `++Iter`

Advances the iterator variable `Iter` to the next element in the vector. So, if `Iter` originally represented the element `V[i]`, it would now represent `V[i + 1]`.

`Iter--` or `--Iter`

Moves back the iterator variable `Iter` to the previous element in the vector. So, if `Iter` originally represented the element `V[i]`, it would now represent `V[I - 1]`.

`Iter + N, Iter - N`

Returns a new iterator that represents the element N positions after or before the element represented by Iter. So, if `Iter` originally represented the element `V[i]`, it would now represent `V[I ± N]`.

Iterator Operations

`IterA - IterB`

We can subtract two vector iterators to find out the distance between two elements that they represent. If `IterA` represents the element `V[A]` and `IterB` represents the element `V[B]`, then the result of `IterA - IterB` is equal to the integer `A - B`.

`IterA == IterB, IterA != IterB`
`IterA < IterB, IterA <= IterB`
`IterA > IterB, IterA >= IterB`

All of the standard relational operators can be used with iterators. If `IterA` represents the element `V[A]` and `IterB` represents the element `V[B]`, then these operations return the same value that the corresponding relational operation on `A` and `B` would return.

`*Iter`

Returns the element at the position designated by the iterator, or sets it if used as an *l-value*. This is called *dereferencing* the iterator.

Iterator Example

```
vector<int> V;          // V starts out empty
V.push_back(1);
V.push_back(2);
V.push_back(3);

vector<int>::iterator Iter = V.begin();
int Num = *Iter;        // Num = 1

Iter++;
Num = *Iter;            // Num = 2

Iter++;
Num = *Iter;            // Num = 3

Iter++;                 // Iter = V.end();
```

If we try to dereference the iterator that is equal to `V.end()`, undefined behavior will result. This could manifest itself as the program crashing, or the operation just silently failing and your program continuing as if nothing happened. Don't do it!

The `end()` iterator

Why does the `end()` iterator point to a position **after** the last element, rather than the last element? Consider how iterators can be used to walk through a sequence:

```
vector<int> V;          // V starts out empty
V.push_back(0);
V.push_back(1);
V.push_back(2);

vector<int>::iterator Iter;
for (Iter = V.begin(); Iter != V.end(); Iter++) {
    cout << *Iter;
}
```

Iterator points to position…	Action
`begin()` (index 0)	output "0" and increment
index 1	output "1" and increment
index 2	output "2" and increment
`end()`	terminate loop

Iterators to Other Elements

If we want an iterator that points to the 100th element in a vector, writing Iter++ 100 times would not be desirable. Instead, we can write `Iter = Iter + I` to advance the iterator `I` positions.

```
vector<int> V(100);              // Create a 100-element vector

vector<int>::iterator Iter;
Iter = V.begin() + 50;           // Iter points to V[50]

*Iter = 12345;                   // Sets V[50] to 12345

Iter = V.begin() + 100;
bool B = (Iter == V.end());      // B == true
```

Just as it is incorrect to access a vector element with an improper index (in the above example, `V[100]` or higher), it is also incorrect to add a number that's too large to an iterator and move it out of range.

Subtracting from iterators works in a similar fashion.

Inserting at Arbitrary Positions

`V.insert(iterator Iter, Type element)`

Inserts a new element before the element at the position denoted by `Iter`.

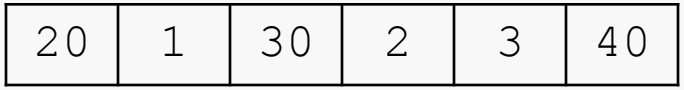
```
vector<int> V;                  // V starts out empty
V.push_back(1);
V.push_back(2);
V.push_back(3);                 // Line 1

V.insert(V.begin(), 20);        // Line 2
V.insert(V.begin() + 2, 30);    // Line 3
V.insert(V.end(), 40);          // Line 4
```

After Line 1:

After Line 2:

After Line 3:

After Line 4:



Erasing Elements in a Vector

`V.erase(iterator Iter)`
Erases the element at the position denoted by `Iter`, shifting the elements after it back to fill the space left by the deleted elements. The size of the vector will decrease by 1.

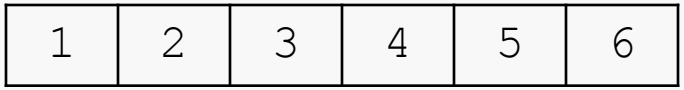
`V.erase(iterator First, iterator Last)`
Erases elements starting at `First` and stopping at the element just before `Last`, shifting elements after the range back to fill the space. The size of the vector will decrease by the distance between `First` and `Last`.

`V.pop_back()`
A shorthand method for erasing the last element in the vector. This is the opposite of `push_back`.

Erasing Elements in a Vector

```
vector<int> V;
for (int i = 1; i <= 6; i++) {
    V.push_back(i);
}                                          // Line 1

V.erase(V.begin() + 2);                    // Line 2
V.erase(V.begin(), V.begin() + 2);         // Line 3
V.pop_back();                              // Line 4
V.erase(V.begin(), V.end());               // Line 5
```

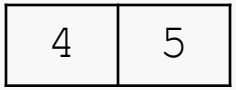
After Line 1:



After Line 2:

After Line 3:

After Line 4:



After Line 5: *empty*

Nested Vectors

Vectors can take almost any type as a parameter, including other vectors. This lets us mimic multidimensional arrays.

Array Declaration	Correspondng Vector Declaration
`int A[N]`	`vector<int> A(N)`
`int A[M][N]`	`vector< vector<int> > A(` `  M, vector<int>(N))`

`int A[K][M][N]`	`vector< vector< vector<int> > >`
	`  A(K, vector< vector<int> >(`
	`  M, vector<int>(N)))`

The syntax is a bit complicated compared to multidimensional arrays. This is explained on the next slide.

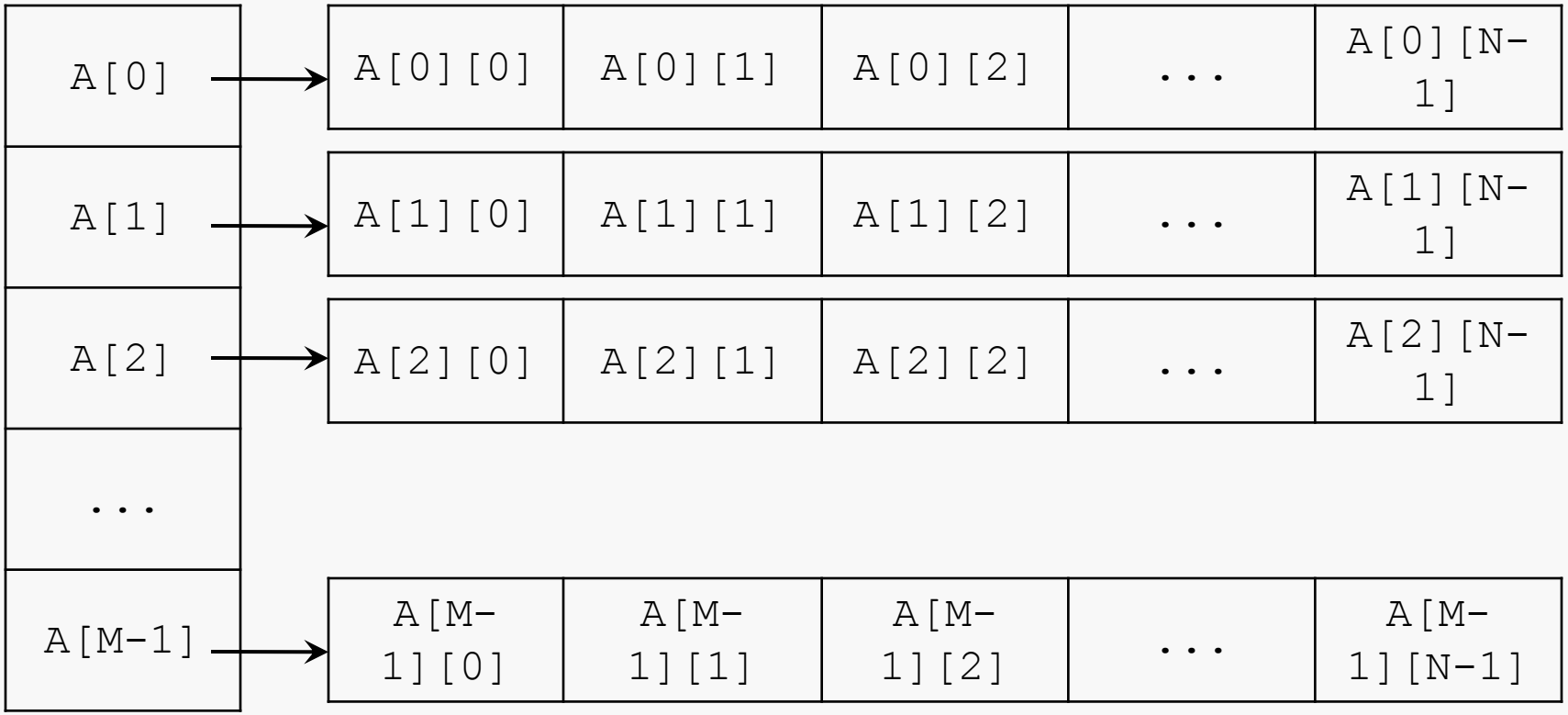
A key thing to note here is that there must be a space between the > symbols that close off the template parameter list. This deficiency in the C++ language will be fixed in the next version of the language, due to be finalized in 2010 or 2011.

Nested Vectors

Let's examine one of these nesting examples further:

```
vector< vector<int> > A(M, vector<int>(N))
```

This can be read as: **A** is a vector with **M** elements, whose elements are themselves vectors of **N** integers. We can visualize this in memory like so:

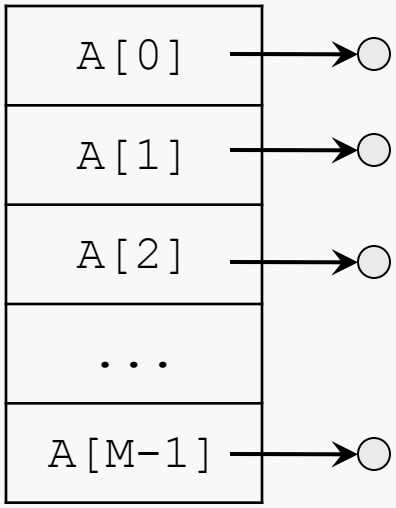


Nested Vectors

Notice that the layout of the vectors on the previous slide was not a grid, like the multidimensional array example. If we left out the initializer for the inner vectors, we would allocate a vector that contained M *empty* vectors:

```
vector< vector<int> >
```

A (**M**)



Since vectors can change in size, there is no requirement that nested vectors represent a perfect grid. You could easily change the size of a single row by writing something like `A[1].push_back(X)`. If we want to treat nested vectors like a grid, however, it is best to make sure that each row is initialized to the same size, as we did in the previous slide.