Chapter 5: STL Sequence Containers

> *In October of 1976 I observed that a certain algorithm – parallel reduction – was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures.*
>
> – Alex Stepanov, inventor of the STL. [Ste07]

The Standard Template Library (STL) is a programmer's dream. It offers efficient ways to store, access, manipulate, and view data and is designed for maximum extensibility. Once you've gotten over the initial syntax hurdles, you will quickly learn to appreciate the STL's sheer power and flexibility.

To give a sense of exactly where we're going, here are a few quick examples of code using the STL:

- We can create a list of random numbers, sort it, and print it to the console *in four lines of code!*

```
vector<int> myVector(NUM_INTS);
generate(myVector.begin(), myVector.end(), rand);
sort(myVector.begin(), myVector.end());
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"));
```

- We can open a file and print its contents *in two lines of code!*

```
ifstream input("my-file.txt");
copy(istreambuf_iterator<char>(input), istreambuf_iterator<char>(),
     ostreambuf_iterator<char>(cout));
```

- We can convert a string to upper case *in one line of code!*

```
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

If you aren't already impressed by the possibilities this library entails, keep reading. You will not be disappointed.

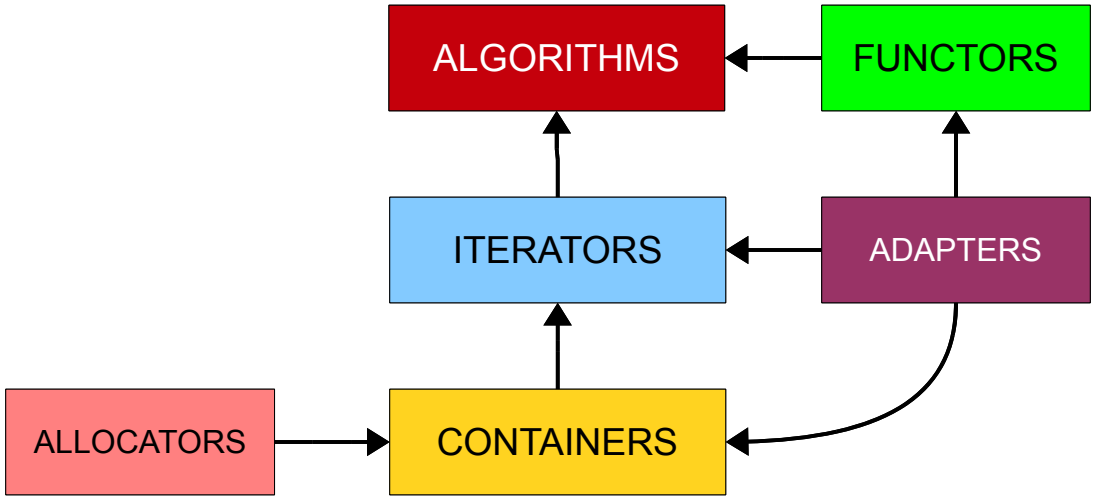
Overview of the STL

The STL is logically divided into six pieces, each consisting of generic components that interoperate with the rest of the library:

- **Containers**. At the heart of the STL are a collection of container classes, standard C++'s analog to the CS106B/X ADTs. For example, you can store an associative collection of key/value pairs in an STL `map`, or a growing list of elements in an STL `vector`.
- **Iterators**. Each STL container exports iterators, objects that view and modify ranges of stored data. Iterators have a common interface, allowing you to write code that operates on data stored in arbitrary containers.
- **Algorithms**. STL algorithms are functions that operate over ranges of data specified by iterators. The scope of the STL algorithms is staggering – there are algorithms for searching, sorting, reordering, permuting, creating, and destroying sets of data.

- **Adapters**. STL *adapters* are objects which transform an object from one form into another. For example, the `stack` adapter transforms a regular `vector` or `list` into a LIFO container, while the `istream_iterator` transforms a standard C++ stream into an STL iterator.
- **Functors**. Because so much of the STL relies on user-defined callback functions, the STL provides facilities for creating and modifying functions at runtime. We will defer our discussion of functors to much later in this text, as they require a fairly nuanced understanding of C++.
- **Allocators**. The STL allows clients of the container classes to customize how memory is allocated and deallocated, either for diagnostic or performance reasons. While allocators are fascinating and certainly worthy of discussion, they are beyond the scope of this text and we will not cover them here.

Diagrammatically, these pieces are related as follows:



Here, the containers rely on the allocators for memory and produce iterators. Iterators can then be used in conjunction with the algorithms. Functors provide special functions for the algorithms, and adapters can produce functors, iterators, and containers. If this seems a bit confusing now, don't worry, you'll understand this relationship well by the time you've finished the next few chapters.

Why the STL is Necessary

Up until this point, all of the programs you've encountered have declared a fixed number of variables that correspond to a fixed number of values. If you declare an `int`, you get a single variable back. If you need to create multiple different `int`s for some reason, you have to declare each of the variables independently. This has its drawbacks. For starters, it means that you need to know how exactly how much data your program will be manipulating before the program begins running. Here's a quick programming challenge for you:

> Write a program that reads in three integers from the user, then prints them in sorted order.

How could you go about writing a program to do this? There are two main technical hurdles to overcome. First, how would we store the numbers the user enters? Second, how do we sort them? Because we know that the user will be entering three distinct values, we could simply store each of them in their own `int` variable. Thus the first step of writing code for this program might look like this:

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

int GetInteger(); // From the streams chapter

int main() {
    int val1 = GetInteger();
    int val2 = GetInteger();
    int val3 = GetInteger();

    /* Sort, then print out. */
}
```

We now have our three values; how can we sort them? It turns out that sorting a list of integers is a classic algorithms question and there are many elegant solutions to the problem. Some algorithms like *quicksort* and *heapsort* run extremely quickly, but are rather difficult to implement. Instead, we'll use a variant algorithm called *selection sort*. The idea behind selection sort is as follows. If we want to sort a list of numbers, we can find the smallest element in the list, then move it to the front of the list. We can then find the second-smallest value, then put it in the second position, find the third-smallest and put it in the third position, etc. Amazingly, with just the tools we've seen so far this simple algorithm is extremely hard to implement. Here's some code for the operation:

```
int main() {
    int val1 = GetInteger();
    int val2 = GetInteger();
    int val3 = GetInteger();

    /* Three cases: Either val1 is the smallest, val2 is the smallest, or
     * val3 is the smallest.  Whichever ends up being the case, we'll put
     * the smallest value into val1.  This uses the swap() function, which is
     * defined in the <algorithm> header and simply swaps the values of
     * two variables.
     */
    if (val2 <= val1 && val2 <= val3) // val2 is smallest
        swap (val1, val2);
    else if (val3 <= val1 && val3 <= val2) // val3 is smallest
        swap (val1, val3);
    // Otherwise, val1 is smallest, and can remain at the front.

    /* Now, sort val2 and val3.  Since there's just two elements, we can do a
     * simple comparison to determine which is the smaller of the two.
     */
    if (val3 <= val2) // val3 is smaller
       swap (val2, val3);
    // Otherwise, val2 is smallest and we don't need to do anything.

    cout << val1 << ' ' << val2 << ' ' << val3 << endl;
}
```

This code is incredibly dense. Don't panic if you don't understand it – part of the purpose of this example is to illustrate how difficult it can be to write good code for this problem without the STL!

But of course, there's another major problem with this code. Let's modify the problem statement a bit by allowing the user to enter *four* numbers. If we make this change, using just the techniques we've covered so far we'll be forced to write code along the following lines:

```
    int main() {
        int val1 = GetInteger();
        int val2 = GetInteger();
        int val3 = GetInteger();
        int val4 = GetInteger();

        /* Find the smallest. */
        if (val2 <= val1 && val2 <= val3 && val2 <= val4) // val2 is smallest
            swap (val1, val2);
        else if (val3 <= val1 && val3 <= val2 && val3 <= val4) // val3 is smallest
            swap (val1, val3);
        else if (val4 <= val1 && val4 <= val2 && val4 <= val3) // val4 is smallest
            swap (val1, val4);
        // Otherwise, val1 is smallest, and can remain at the front.

        /* Find the second-smallest. */
        if (val3 <= val2 && val3 <= val4) // val3 is smallest
            swap (val2, val3);
        else if (val4 <= val2 && val4 <= val3) // val4 is smallest
            swap (val2, val4);
        // Otherwise, val2 is smallest, and can remain at the front.

        /* Find the third-smallest. */
        if (val4 <= val3) // val4 is smaller
           swap (val3, val4);
        // Otherwise, val3 is smallest and we don't need to do anything.

        cout << val1 << ' ' << val2 << ' ' << val3 << ' ' << val4 << endl;
}
```

This code is just downright awful! It's cryptic, difficult to read, and not the sort of code you'd like to bring home to mom and dad. Now imagine what the code would look like if we had *five* numbers instead of four. It will keep growing and growing, becoming progressively more impossible to read until eventually we'd give up in frustration. What's worse, though, is that each of these programs is a special case of a general problem – read in *n* integers from the user and print them in sorted order – but the code for each case bears little to no resemblance to the code for each other case.

Introducing the `vector`

What's missing from our programming repertoire right now is the ability to create and access a *variable number* of objects. Right now, if we want to store a sequence of five integers, we must create five independent integer variables and have no way of accessing them uniformly. Fortunately, the STL provides us a versatile tool called the *`vector`* that allows us to store sequences of elements using a single variable. As you'll see, the `vector` is nothing short of extraordinary and will arise time and again throughout your programming career.

At a high level, a `vector` is an object that represents a sequence of elements. This means that you can use a vector to store a grocery list, a list of Olympic figure skating scores, or a set of files to read. The elements in a `vector` are *indexed*, meaning that they have a well-defined position in the sequence. For example, given the sequence

Value	137	42	2718	3141	410
Index	0	1	2	3	4

The first element (137) is at position zero, the second (42) at position one, etc. Notice that elements in the sequence appear in the order 0, 1, 2, etc. rather than the more intuitive 1, 2, 3, ... You have seen this already in your exploration of the string class, so hopefully this notation isn't too startling.

The major difference between the vector and the string is that the `vector` can be configured to store elements of *any* type. That is, you can have a `vector` of `int`s, a `vector` of `string`s, or even a `vector` of `vector`s of `string`s. However, while the `vector` can store elements of any type, any single `vector` can only store elements of a single type. It is illegal to create a `vector` that stores a sequence of many different types of elements, meaning that you can't represent the list 0, Apple, 2.71828 because each of the list elements has a different type. This may seem like a rather arbitrary restriction – theoretically speaking, there's no reason that you shouldn't be able to have lists of all sorts of elements – but fortunately in most cases this restriction does not pose too much of a problem.

Because vectors can only store elements of a fixed type, when you create a `vector` in your programs you will need to explicitly indicate to the compiler what type of elements you aim to store in the `vector`. As an example, to create a `vector` of `int`s, you would write

```
vector<int> myVector;
```

Here, we declare a local variable called `myVector` that has type `vector<int>`. The type inside of the angle brackets is called a *template argument* and indicates to C++ what type of elements are stored in the `vector`. Here, the type is `int`, but it's legal to put pretty much whatever type you would like in the brackets. For example, all of the following declarations are legal:

```
vector<int>    intVector;  // Stores ints
vector<string> strVector;  // Stores strings
vector<double> realVector; // Stores real numbers
```

It is also perfectly legal to store your own custom structs in a `vector`, as seen here:

```
struct MyStruct {
    int myInt;
    double myDouble;
    string myString;
};

vector<MyStruct> myStructVector; // Stores MyStructs
```

In order to use the `vector` type, you will need to `#include <vector>` at the top of your program. As we explore some of the other STL containers, this pattern will also apply.

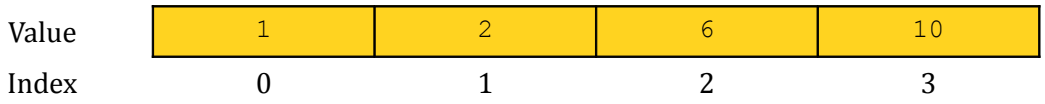
We now know how to *create* `vector`s, but how do we *use* them? To give you a feel for how the `vector` works, let's return to our previous example of reading in numbers and printing them out in sorted order. Using a `vector`, this can be accomplished very elegantly. We'll begin by defining a constant, `kNumValues`, which will represent the number of elements to read in. This is shown here:

```
#include <iostream>
#include <vector>   // Necessary to use vector
#include <string>
#include <sstream>
using namespace std;

string GetLine(); // As defined in the previous chapter
int GetInteger(); // As defined in the previous chapter

const int kNumValues = 10;

int main() {
    /* ... still more work to come ... */
}
```

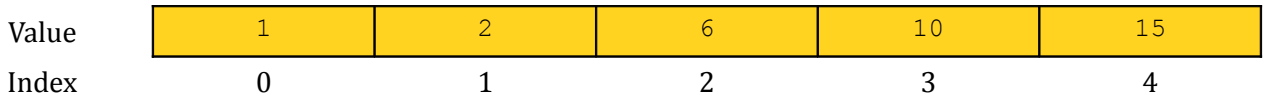
Now, we'll show to how to read in `kNumValues` values from the user and store them inside a `vector`. First, we'll have to create the `vector`, as shown here:

```
int main() {
    vector<int> values;

    /* ... */
}
```

A freshly-constructed `vector`, like a freshly-constructed `string`, is initially empty. Consequently, we'll need to get some values from the user, then store them inside the `vector`. Reading the values is fairly easy; we simply sit in a `for` loop reading data from the user. But how do we store them in the `vector`? There are several ways to do this, of which the simplest is the `push_back` function. The `push_back` function can be invoked on a `vector` to add a new element to the end of `vector`'s sequence. For example, given a `vector` managing the following sequence:



Then calling push_back on the vector to store the value fifteen would cause the vector to manage the new sequence



Syntactically, `push_back` can be used as follows:

```
int main() {
    vector<int> values;

    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter another value: ";
        int val = GetInteger();

        values.push_back(val);
    }
}
```

Notice that we write `values.push_back(val)` to append the number val to the sequence stored inside the `vector`.

Now that we have read in our sequence of values, let's work on the next part of the program, sorting the elements in the vector and printing them to the user. For this we'll still use the selection sort algorithm, but it will be miraculously easier to follow when implemented on top of the `vector`. Recall that the general description of the selection sort algorithm is as follows:

- Find the smallest element of the list.
- Put that element at the front of the list.
- Repeat until all elements are in place.

Let's see how to implement this function. We'll begin simply by writing out the function signature, which looks like this:

```
void SelectionSort(vector<int>& v) {
    /* ... */
}
```

This function is named `SelectionSort` and takes as a parameter a `vector<int>` by reference. There are two key points to note here. First, we still have to explicitly indicate what type of elements are stored in the `vector` parameter. That is, it's illegal to write code of this sort:

```
void SelectionSort(vector& v) { // Error:  What kind of vector?
    /* ... */
}
```

As a general rule, you will *never* see `vector` unless it's immediately followed with a type in angle brackets. The reason is simple – every `vector` can only store one type of element, and unless you explicitly indicate what that type is the compiler will have no way of knowing.

The other important detail about this function is that it takes its parameter by reference. We will be sorting the `vector` in-place, meaning that we will be reordering the elements of the `vector` rather than creating a new `vector` containing a sorted copy of the input.

We now have the function prototype set up, so let's get into the meat of the algorithm. To implement selection sort, we'll need to find the smallest element and put it in front, the the second-smallest element and put it in the second position, etc. The code for this is as follows:

```
void SelectionSort(vector<int>& v) {
    for (size_t i = 0; i < v.size(); ++i) {
        size_t smallestIndex = GetSmallestIndex(v, i); // We'll write this
                                                       // function momentarily
        swap (v[i], v[smallestIndex]);
    }
}
```

This code is fairly dense and introduces some syntax you have not yet seen before, so let's take a few moments to walk through exactly how it works. The first detail that might have caught your eye is this one:

```
for (size_t i = 0; i < v.size(); ++i)
```

This for loop looks strange for two reasons. First, instead of creating an `int` variable for the iteration, we create a variable of type `size_t`. A `size_t` is a special type of variable that can hold values that represent

sizes of things (`size_t` stands for "size type"). In many aspects `size_t` is like a regular `int` – it holds an integer value, can be incremented with `++`, compared using the relational operators, etc. However, unlike regular `int`s, `size_t`s cannot store negative values. The intuition behind this idea is that no collection of elements can have a negative size. You may have a list of no elements, or a list of billions of elements, but you'll *never* encounter a list with -1 elements in it. Consequently, when iterating over an STL container, it is customary to use the special type `size_t` to explicitly indicate that your iteration variable should always be nonnegative. The other detail of this for loop is that the loop iterates from 0 to `v.size()`. The `size()` member function on the STL `vector` returns the number of elements stored in the sequence, just like the `size()` and `length()` functions on the standard `string` class. In this particular case we could have iterated from 0 to `kNumValues`, since we're guaranteed that the main function will produce a `vector` of that many elements, but in general when iterating over a vector it's probably a wise idea to use the `vector`'s size as an upper bound so that the code works for vectors of arbitrarily many elements.

Let's continue onward through the code. The body of the loop is this code here:

```
size_t smallestIndex = GetSmallestIndex(v, i);
swap (v[i], v[smallestIndex]);
```

This calls some function called `GetSmallestIndex` (which we have not yet defined) which takes in the vector and the current index. We will implement this function shortly, and its job will be to return the index of the smallest element in the vector occurring no earlier than position `i`. We then use the `swap` function to exchange the values stored in at positions `i` and `smallestIndex` of the `vector`. Notice that, as with the standard string class, the syntax `v[i]` means "the element in `vector` `v` at position `i`." Let's take a minute to think about how this code works. The variable `i` counts up from `0` to `v.size() - 1` and visits every element of the `vector` exactly once. On each iteration, the code finds the smallest element in the `vector` occurring no earlier than position `i`, then exchanges it with the element at position `i`. This means that the code will

1. Find the smallest element of the `vector` and put it in position 0.
2. Find the smallest element of the remainder of the `vector` and put it in position 1.
3. Find the smallest element of the remainder of the `vector` and put it in position 2.

...

This is precisely what we set out to do, and so the code will work marvelously. Of course, this assumes that we have a function called `GetSmallestIndex` which returns the index of the smallest element in the `vector` occurring no earlier than position `i`. To finalize our implementation of `SelectionSort`, let's go implement this function. Again, we'll start with the prototype, which is shown here:

```
size_t GetSmallestIndex(vector<int>& v, size_t startIndex) {
    /* ... */
}
```

This function accepts as input a `vector<int>` by reference and a `size_t` containing the start index, then returns a `size_t` containing the result. There's an important but subtle detail to note here. At a high level, the `GetSmallestIndex` function has no business modifying the `vector` it takes as input. Its task is to find the smallest element and return it, no more. So why exactly does this function take the `vector<int>` parameter by reference? The answer is *efficiency*. In C++, if you pass a parameter to a function by value, then whenever that function is called C++ will make a full copy of the argument. When working with the STL containers, which can contain thousands (if not millions or tens of millions) of elements, the cost of copying the container can be staggering. Consequently, it is considered good program-

ming practice to pass STL containers into functions by reference rather than by value, since this avoids an expensive copy.*

The implementation of this function is rather straightforward:

```
size_t GetSmallestIndex(vector<int>& v, size_t startIndex) {
    size_t bestIndex = startIndex;

    for (size_t i = startIndex; i < v.size(); ++i)
        if (v[i] < v[bestIndex])
            bestIndex = i;

    return bestIndex;
}
```

This function iterates over the elements of the `vector` starting with position `i`, checking whether the element at the current position is less than the smallest element we've seen so far. If so, the function updates where in the `vector` that element appears. At the end of the function, once we've looked at every element in range, the `bestIndex` variable will hold the index of the smallest element in `vector v` occurring no earlier than `startIndex`, and so we return the value. We've implemented the `GetSmallestIndex` function, meaning that we have a working implementation of `SelectionSort`.

To finalize our program, let's update the main function to print out the sorted vector. This is shown here:

```
int main() {
    vector<int> values;

    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter another value: ";
        int val = GetInteger();
        values.push_back(val);
    }

    SelectionSort(values);

    for (size_t i = 0; i < kNumValues; ++i)
        cout << values[i] << endl;
}
```

Compare this implementation of the program to the previous version, which did not have the luxury of using `vector`. As you can see, the code in this program is significantly clearer than before. Moreover, it's much more *scalable*. If we want to read in a different number of values from the user, we can do so simply by adjusting the value of the `kNumValues` constant, and the rest of the code will update automatically.

An Alternative Implementation

In the previous section, we wrote a program which reads in some number of values from the user, sorts them, and then prints them out. Of course, the program we wrote was just one method for solving the problem. In particular, there is another implementation strategy we could have considered that lends itself to a substantially shorter implementation. Notice that in the above program, we read in a list of values from the user and blindly added them to the end of the list we wanted to sort. These values weren't necessarily in sorted order, and so we had to run a postprocessing step to sort the vector before displaying it to

* In practice you would almost certainly pass the parameter by *reference-to-`const`*, which indicates that the parameter cannot be modified. We will take this issue up in a later chapter, but for now pass-by-reference should be good enough for our purposes.

the user. But what if we opt for a different approach? In particular, suppose that whenever we read a value from the user, instead of putting that value at the end of the sequence, we find where in the vector the element should go, then insert the value at that point? For example, suppose that the user has entered the following four values:

Value	100	200	300	400
Index	0	1	2	3

Now suppose that she enters the number 137. If we append 137 to the `vector`, then the numbers will not all be in sorted order. Instead, we'll find the location where 137 should go in the sorted `vector`, then insert it directly into that location. This is shown here:

Value	100	137	200	300	400
Index	0	1	2	3	4

This strategy ends up being a bit simpler to implement than our previous program, which relied on selection sort. Of course, to implement the program using the above strategy, we need to answer two questions. First, how do we find out where in the `vector` the user's element should go? Second, how do we insert an element into a `vector` directly at that position? This first question is algorithmic; the second is simply a question of what operations are legal on the `vector`. Consequently, we'll begin our discussion with how to find the insertion point, and will then see how to add an element to a `vector` at a particular point.

Suppose that we are given a sorted list of integers and some value to insert into the list. We are curious to find at what index the new value should go. This is an interesting algorithmic challenge, since there are many valid solutions. However, there's one particular simple way to find where the element should go. In order for a list to be in sorted order, every element has to be smaller than the element that comes one position after it. Therefore, if we find the first element in the `vector` that is *bigger* than the element we want to insert, we know that the element we want to insert must come directly before that element. This suggests the following algorithm:

```
/* Watch out!  This code contains a bug! */
size_t InsertionIndex(vector<int>& v, int toInsert) {
    for(size_t i = 0; i < v.size(); ++i)
        if (toInsert < v[i])
            return i;
}
```

This code is *mostly* correct, but contains a pretty significant flaw. In particular, what happens if the element we want to insert is bigger than every element in the `vector`? In that case, the `if` statement inside of the `for` loop will never evaluate to true, and so the function will not return a value. If a function finishes without returning a value, the program has *undefined behavior*. This means that the function might return zero, it might return garbage, or your program might immediately crash outright. This certainly isn't what we want to have happen, so how can we go about fixing it? Notice that the only way that the above function never returns a value is if the element to be inserted is at least as big as every element in the `vector`. In that case, the correct behavior should be to put the element on the end of the `vector`. We'll signal this by having the function return `v.size()` if the element is bigger than every element in the sequence. This is shown here:

```
size_t InsertionIndex(vector<int>& v, int toInsert) {
    for(size_t i = 0; i < v.size(); ++i)
        if (toInsert < v[i])
            return i;
    return v.size();
}
```

All that's left to do now is use this function to build a sleek implementation of the program. But before we can do that, we have to see how to insert an element into a `vector` at an arbitrary position. The good news is that the `vector` supports this operation naturally, and in fact you can insert an element into a `vector` at any position. The bad news is that the syntax for doing so is nothing short of cryptic and without an understanding of STL iterators will look entirely alien. We'll talk about iterators more next chapter, but in the meantime you can just take it for granted that the following syntax is legal. Given a `vector` `v` and an element `e`, to insert `e` into `v` at position `n`, we use the syntax

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

For example, to insert the element 137 at position zero in the vector, you would write

```
v.insert(v.begin(), 137);
```

Similarly, to insert the element 42 at position five, we could write

```
v.insert(v.begin() + 5, 42);
```

One of the trickier parts of the `insert` function is determining exactly where the element will be inserted. Recall that `vector`s are zero-indexed, the above statement will insert the number 42 as the *sixth* element of the sequence, not the fifth. When an element is inserted at a position, all of the elements after it are shuffled down one spot to make room, so calling insert will never overwrite a value.

Given this syntax and the above implementation of `InsertionIndex`, we can write a program to read in a list of values and print them out in sorted order as follows:

```
int main() {
    vector<int> values;

    /* Read the values. */
    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter an integer: ";
        int val = GetInteger();

        /* Insert the element at the correct position. */
        values.insert(values.begin() + InsertionIndex(values, val), val);
    }

    /* Print out the sorted list. */
    for (size_t i = 0; i < values.size(); ++i)
        cout << values[i] << endl;
}
```

This code is much shorter than before, even when you factor in the code for `InsertionIndex`.

Additional `vector` Operations

The previous section on sorting numbers with the `vector` showcased many of operations that can be performed on a `vector`, but was by no means a complete survey of what the `vector` can do. While some `vector` operations require a nuanced understanding of *iterators*, which we will cover next chapter, there are a few common operations on the `vector` that we will address in this section before moving on to other container classes.

One of the key distinctions between the `vector` and other data types we've seen so far is that the `vector` has a variable size. It can contain no elements, dozens of elements, or even millions of elements. In the preceding examples, we explored two ways to change the number of elements in the `vector`: `push_back`, which appends new elements to the back of the `vector`, and `insert`, which adds an element to the `vector` at an arbitrary position. However, there are several more ways to add and remove elements from the `vector`, some of which are discussed here.

When creating a new vector to represent a list of values, by default C++ will make the `vector` store an empty list. That is, a newly-created `vector` is by default empty. However, at times you might want to initialize the `vector` to a certain size. C++ allows you to do this by specifying the starting size of the `vector` at the point where the vector is created. For example, to create a `vector` of integers that initially holds fifteen elements, you could write this as

```
vector<int> myVector(15);
```

That is, you declare the `vector` as normal, and put the default size in parentheses afterwards. Note that this only changes the starting size of the `vector`, and you are free to add additional elements to the `vector` later in your program.

When creating a `vector` that holds primitive types, such as `int` or `double`, the elements in the `vector` will default to zero (or `false` in the case of `bool`s). This means that the above line of code means "create a `vector` of integers called `myVector` that initially holds fifteen entries, all zero." Similarly, this line of code:

```
vector<string> myStringVector(10);
```

Will create a `vector` of `string`s that initially holds ten copies of the empty string.

In some cases, you may want to initialize the `vector` to a certain size where each element holds a value other than zero. You may wish, for example, to construct a `vector<string>` holding five copies of the string "`(none)`," or a `vector<double>` holding twenty copies of the value 137. In these cases, C++ lets you specify both the number and default value for the elements in the vector using the following syntax:

```
vector<double> myReals(20, 137.0);
vector<string> myStrings(5, "(none)");
```

Notice that we've enclosed in parentheses both the *number* of starting elements in the vector and the *value* of these starting elements.

An important detail is that this syntax is only legal when initially creating a `vector`. If you have an existing `vector` and try to use this syntax, you will get a compile-time error. That is, the following code is illegal:

```
vector<double> myReals;
myReals(20, 137.0); // Error: Only legal to do this when the object is created
```

If you want to change the number of elements in a `vector` after it has already been created, you can always use the `push_back` and `insert` member functions. However, if you'd like to make an abrupt change in the number of elements in the `vector` (perhaps by adding or deleting a large number of elements all at once), you can use the `vector`'s `resize` member function. The `resize` function is very similar to the syntax we've just encountered: you can specify either a number of elements or a number of elements and a value, and the `vector` will be resized to hold that many elements. However, `resize` behaves somewhat differently from the previous construct because when using `resize`, the `vector` might already contain elements. Consequently, `resize` works by adding or removing elements from the end of the `vector` until the desired size is reached. To get a better feel for how `resize` works, let's suppose that we have a function called `PrintVector` that looks like this:

```
void PrintVector(vector<int>& elems) {
    for (size_t i = 0; i < elems.size(); ++i)
        cout << elems[i] << ' ';
    cout << endl;
}
```

This function takes in a `vector<int>`, then prints out the elements in the `vector` one at a time, followed by a newline. Given this function, consider the following code snippet:

```
vector<int> myVector;     // Defaults to empty vector
PrintVector(myVector);    // Output: [nothing]

myVector.resize(10);      // Grow the vector, setting new elements to 0
PrintVector(myVector);    // Output: 0 0 0 0 0 0 0 0 0 0

myVector.resize(5);       // Shrink the vector
PrintVector(myVector);    // Output: 0 0 0 0 0

myVector.resize(7, 1);    // Grow the vector, setting new elements to 1
PrintVector(myVector);    // Output: 0 0 0 0 0 1 1

myVector.resize(1, 7);    // The second parameter is effectively ignored.
PrintVector(myVector);    // Output: 0
```

In the first line, we construct a new `vector`, which is by default empty. Consequently, the call to `PrintVector` will produce no output. We then invoke `resize` to add ten elements to the `vector`. These elements are added to the end of the `vector`, and because we did not specify a default value are all initialized to zero. On our next call to `resize`, we shrink the `vector` down to five elements. Next, we use `resize` to expand the vector to hold seven elements. Because we specified a default value, the newly-created elements default to 1, and so the sequence is now 0, 0, 0, 0, 0, 1, 1. Finally, we use `resize` to trim the sequence. Because the second argument to `resize` is only considered if new elements are added, it is effectively ignored.

We've seen several vector operations so far, but there is a wide class of operations we have not yet considered – operations which remove elements from the `vector`. As you saw, the `push_back` and `insert` functions can be used to splice new elements into the `vector`'s sequence. These two functions are balanced by the `pop_back` and `erase` functions. `pop_back` is the opposite of `push_back`, and removes the last element from the vector's sequence. `erase` is the deletion counterpart to `insert`, and removes an element at a particular position from the `vector`. As with `insert`, the syntax for `erase` is a bit tricky. To remove a single element from a random point in a `vector`, use the `erase` method as follows:

```
myVector.erase(myVector.begin() + n);
```

where `n` represents the index of the element to erase.

In some cases, you may feel compelled to completely erase the contents of the `vector`. In that case, you can use the `clear` function, which completely erases the `vector` contents. `clear` can be invoked as follows:

```
myVector.clear();
```

Summary of `vector`

The following table lists some of the more common operations that you can perform on a `vector`. We have not talked about `iterator`s or the `const` keyword yet, so don't worry if you're confused by those terms. This table is designed as a reference for any point in your programming career, so feel free to skip over entries that look too intimidating.

Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty vector.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T& default)`	`vector<string> myVector(5, "blank");` Constructs a vector of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); ++i) { ... }` Returns the number of elements in the vector.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the vector is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the vector and sets the size to zero.

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
 	
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The `at` member function will throw an exception if you try to access data beyond the end. We will cover exception handling in a later chapter.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	
	Resizes the vector so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the vector, so you can't use `resize` to add elements to or remove elements from the start of the vector.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	
	Returns a reference to the last element in the `vector`.

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	
	Returns a reference to the first element in the vector.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the vector.
`iterator begin();` `const_iterator begin() const;`	`vector<int>::iterator itr = myVector.begin();` Returns an iterator that points to the first element in the vector.
`iterator end();` `const_iterator end() const;`	`while(itr != myVector.end());` Returns an iterator to the element *after* the last. The iterator returned by end does not point to an element in the vector.

`iterator insert(iterator position,`
`const T& value);`
`void insert(iterator start,`
`size_type numCopies,`
`const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	
	The first version inserts the specified value into the vector, and the second inserts `numCopies` copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.

`iterator erase(iterator position);`
`iterator erase(iterator start,`
`iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	
	The first version erases the element at the position pointed to by position. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`. All iterators after the remove point are invalidated. If using this member function on a `deque` (see below), all iterators are invalidated.

deque: A New Kind of Sequence

For most applications where you need to represent a sequence of elements, the `vector` is an ideal tool. It is fast, lightweight, and intuitive. However, there are several aspects of the vector that can be troublesome in certain applications. In particular, the `vector` is only designed to grow in one direction; calling `push_back` inserts elements at the end of the vector, and `resize` always appends elements to the end.

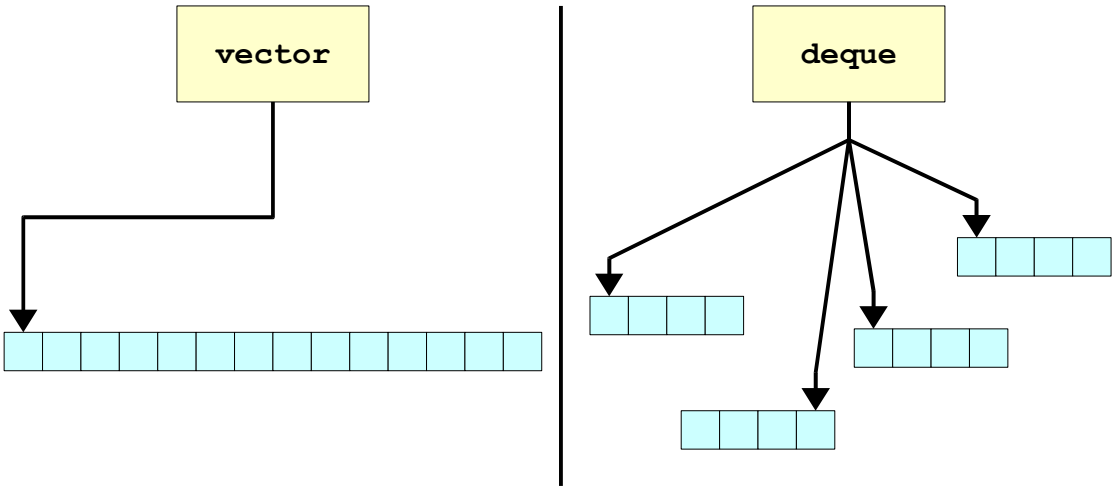
While it's possible to insert elements into other positions of the `vector` using the `insert` function, doing so is fairly inefficient and relying on this functionality can cause a marked slowdown in your program's performance. For most applications, this is not a problem, but in some situations you will need to manage a list of elements that will grow and shrink on both ends. Doing this with a `vector` would be prohibitively costly, and we will need to introduce a new container class: the `deque`.

`deque` is a strange entity. It is pronounced "deck," as in a deck of cards, and is named as a contraction of "**d**ouble-**e**nded **que**ue." It is similar to the `vector` in almost every way, but supports a few operations that the `vector` has trouble with. Because of its similarity to `vector`, many C++ programmers don't even know that the `deque` exists. In fact, of all of the standard STL containers, `deque` is probably the least-used. But this is not to say that it is not useful. The `deque` packs significant firepower, and in this next section we'll see some of the basic operations that you can perform on it.

What's interesting about the `deque` is that all operations supported by `vector` are also provided by `deque`. Thus we can `resize` a `deque`, use the bracket syntax to access individual elements, and erase elements at arbitrary positions. In fact, we can rewrite the number-sorting program to use a `deque` simply by replacing all instances of vector with `deque`. For example:

```
int main() {
    deque<int> values; // Use deque instead of vector

    /* Read the values. */
    for (int i = 0; i < kNumValues; ++i)
    {
        cout << "Enter an integer: ";
        int val = GetInteger();

        /* Insert the element at the correct position. */
        values.insert(values.begin() + InsertionIndex(values, val), val);
    }

    /* Print out the sorted list. */
    for (size_t i = 0; i < values.size(); ++i)
        cout << values[i] << endl;
}
```

However, `deque`s also support two more functions, `push_front` and `pop_front`, which work like the `vector`'s `push_back` and `pop_back` except that they insert and remove elements at the front of the `deque`. But this raises an interesting question: if `deque` has strictly more functionality than `vector`, why use `vector`? The main reason is speed. `deque`s and `vector`s are implemented in two different ways. Typically, a `vector` stores its elements in contiguous memory addresses. `deque`s, on the other hand, maintain a list of different "pages" that store information. This is shown here:



These different implementations impact the efficiency of the `vector` and `deque` operations. In a `vector`, because all elements are stored in consecutive locations, it is possible to locate elements through simple arithmetic: to look up the nth element of a `vector`, find the address of the first element in the vector, then jump forward n positions. In a `deque` this lookup is more complex: the `deque` has to figure out which page the element will be stored in, then has to search that page for the proper item. However, inserting elements at the front of a `vector` requires the `vector` to shuffle all existing elements down to make room for the new element (slow), while doing the same in the `deque` only requires the `deque` to rearrange elements in a single page (fast).

If you're debating about whether to use a `vector` or a `deque` in a particular application, you might appreciate this advice from the C++ ISO Standard (section 23.1.1.2):

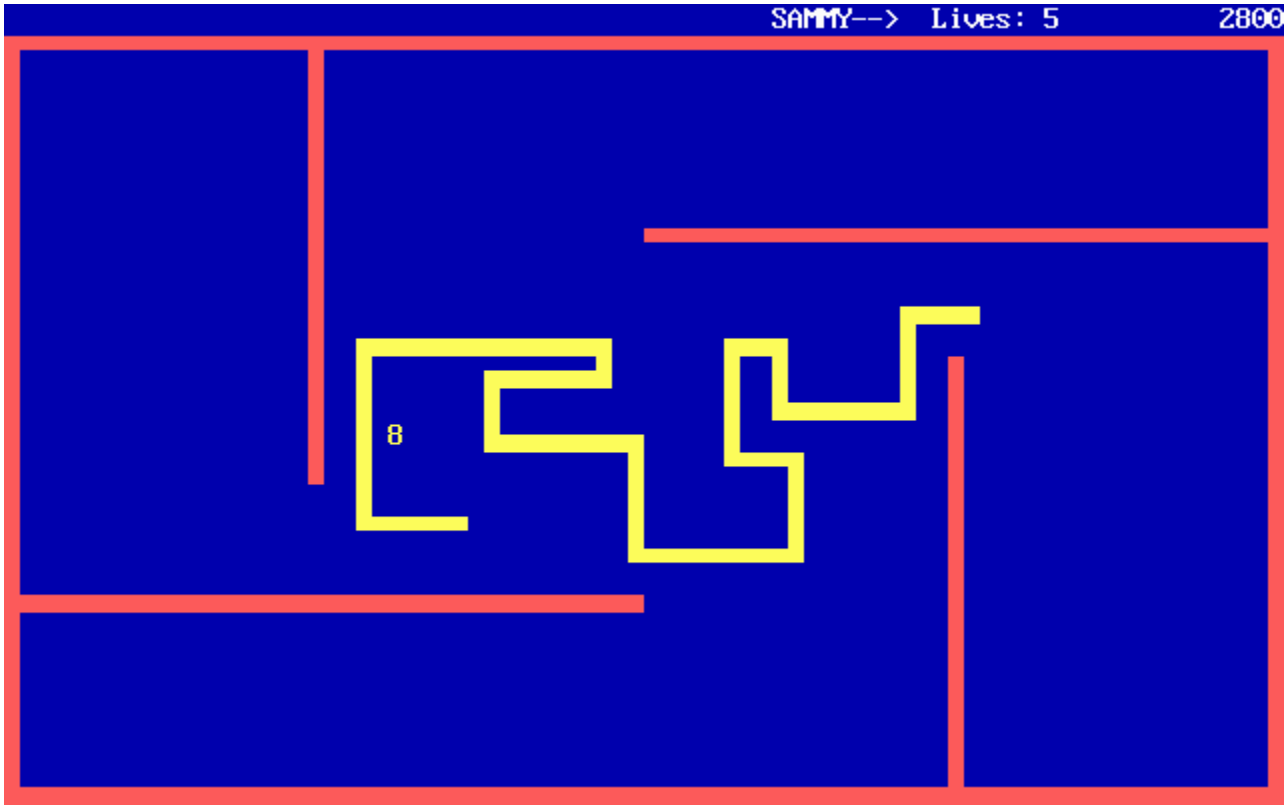
> *`vector` is the type of sequence that should be used by default... `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.*

If you ever find yourself about to use a vector, check to see what you're doing with it. If you need to optimize for fast access, keep using a `vector`. If you're going to be inserting or deleting elements at the beginning or end of the container frequently, consider using a `deque` instead.

Extended Example: Snake

Few computer games can boast the longevity or addictive power of *Snake*. Regardless of your background, chances are that you have played Snake or one of its many variants. The rules are simple – you control a snake on a two-dimensional grid and try to eat food pellets scattered around the grid. You lose if you crash into the walls or into your own body. True to Newton's laws, the snake continues moving in a single direction until you explicitly change its bearing by ninety degrees. Every time the snake eats food, a new piece of food is randomly placed on the grid and the snake's length increases. Over time, the snake's body grows so long that it becomes an obstacle, and if the snake collides with itself the player loses.

Here's a screenshot from QBasic *Nibbles*, a Microsoft implementation of Snake released with MS-DOS version 5.0. The snake is the long yellow string, and the number 8 is the food:



Because the rules of Snake are so simple, it's possible to implement the entire game in only a few hundred lines of C++ code. In this extended example, we'll write a Snake program in which the *computer* controls the snake according to a simple AI. In the process, we'll gain experience with the STL `vector` and `deque`, the streams library, and a sprinkling of C library functions. Once we've finished, we'll have a rather snazzy program that can serve as a launching point for further C++ exploration.

Our Version of Snake

There are many variants of Snake, so to avoid confusion we'll explicitly spell out the rules of the game we're implementing:

1. The snake moves by extending its head in the direction it's moving and pulling its tail in one space.
2. The snake wins if it eats twenty pieces of food.
3. The snake loses if it crashes into itself or into a wall.
4. If the snake eats a piece of food, its length grows by one and a new piece of food is randomly placed.
5. There is only one level, the starting level.

While traditionally Snake is played by a human, our Snake will be computer-controlled so that we can explore some important pieces of the C runtime library. We'll discuss the AI we'll use when we begin implementing it.

Representing the World

In order to represent the Snake world, we need to keep track of the following information:

1. The size and layout of the world.
2. The location of the snake.
3. How many pieces of food we've consumed.

Let's consider this information one piece at a time. First, how should we represent the world? The world is two-dimensional, but all of the STL containers we've seen so far only represent lists, which are inherently one-dimensional. Unfortunately, the STL doesn't have a container class that encapsulates a multidimensional array, but we can emulate this functionality with an STL `vector` of `vector`s. For example, if we represent each square with an object of type `WorldTile`, we could use a `vector<vector<WorldTile> >`. Note that there is a space between the two closing angle brackets – this is deliberate and is an unfortunate bug in the C++ specification. If we omit the space, C++ would interpret the closing braces on `vector<vector<WorldTile>>` as the stream extraction operator `>>`, as in `cin >> myValue`. Although most compilers will accept code that uses two adjacent closing braces, it's bad practice to write it this way.

While we could use a `vector<vector<WorldTile> >`, there's actually a simpler option. Since we need to be able to display the world to the user, we can instead store the world as a `vector<string>` where each `string` encodes one row of the board. This also simplifies displaying the world; given a `vector<string>` representing all the world information, we can draw the board by outputting each string on its own line. Moreover, since we can use the bracket operator [] on both `vector` and `string`, we can use the familiar syntax `world[row][col]` to select individual locations. The first brackets select the `string` out of the `vector` and the second the character out of the `string`.

We'll use the following characters to encode game information:

- A space character (`' '`) represents an empty tile.
- A pound sign (`'#'`) represents a wall.
- A dollar sign (`'$'`) represents food.
- An asterisk (`'*'`) represents a tile occupied by a snake.

For simplicity, we'll bundle all the game data into a single `struct` called `gameT`. This will allow us to pass all the game information to functions as a single parameter. Based on the above information, we can begin writing this `struct` as follows:

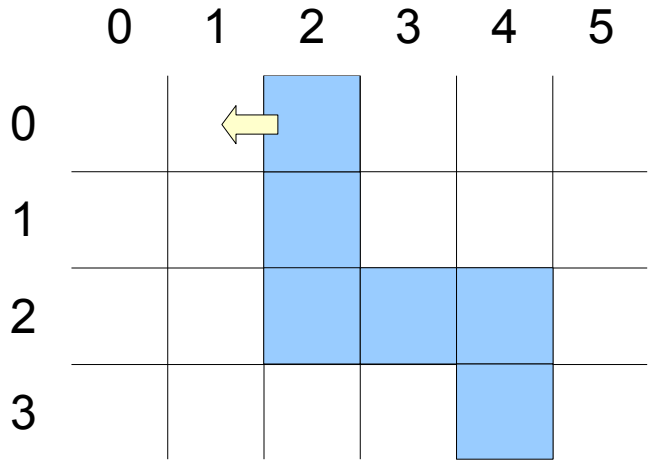
```
struct gameT {
    vector<string> world;
};
```

We also will need quick access to the dimensions of the playing field, since we will need to be able to check whether the snake is out of bounds. While we could access this information by checking the dimensions of the `vector` and the strings stored in it, for simplicity we'll store this information explicitly in the `gameT` struct, as shown here:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;
};
```

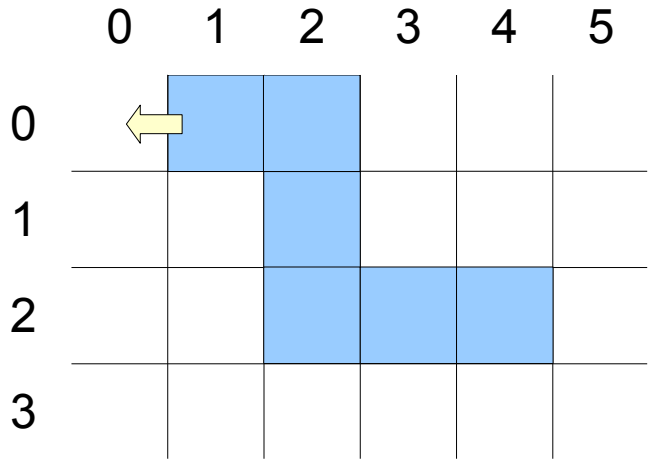
For consistency, we'll access elements in the `vector<string>` treating the first index as the row and the second as the column. Thus `world[3][5]` is row three, column five (where indices are zero-indexed).

Now, we need to settle on a representation for the snake. The snake lives on a two-dimensional grid and moves at a certain velocity. Because the grid is discrete, we can represent the snake as a collection of its points along with its velocity vector. For example, we can represent the following snake:



As the points (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3) and the velocity vector (-1, 0).

The points comprising the snake body are ordered to determine how the snake moves. When the snake moves, the first point (the *head*) moves one step in the direction of the velocity vector. The second piece then moves into the gap left by the first, the third moves into the gap left by the second piece, etc. This leaves a gap where the tail used to be. For example, after moving one step, the above snake looks like this:



To represent the snake in memory, we thus need to keep track of its velocity and an ordered list of the points comprising it. The former can be represented using two `int`s, one for the Δx component and one for the Δy component. But how should we represent the latter? We've just learned about the `vector` and `deque`, each of which could represent the snake. To see what the best option is, let's think about how we might implement snake motion. We can think of snake motion in one of two ways – first, as the head moving forward a step and the rest of the points shifting down one spot, and second as the snake getting a new point in front of its current head and losing its tail. The first approach requires us to update every element in the body and is not particularly efficient. The second approach can easily be implemented with a `deque` through an appropriate combination of `push_front` and `pop_back`. We will thus use a `deque` to encode the snake body.

If we want to have a `deque` of points, we'll first need some way of encoding a point. This can be done with this struct:

```
struct pointT {
    int row, col;
};
```

Taking these new considerations into account, our new `gameT` struct looks like this:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;
};
```

Finally, we need to keep track of how many pieces of food we've munched so far. That can easily be stored in an `int`, yielding this final version of `gameT`:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;

    int numEaten;
};
```

The Skeleton Implementation

Now that we've settled on a representation for our game, we can start thinking about how to organize the program. There are two logical steps – setup and gameplay – leading to the following skeleton implementation:

```
#include <iostream>
#include <string>
#include <deque>
#include <vector>
using namespace std;

/* Number of food pellets that must be eaten to win. */
const int kMaxFood = 20;

/* Constants for the different tile types. */
const char kEmptyTile = ' ';
const char kWallTile  = '#';
const char kFoodTile  = '$';
const char kSnakeTile = '*';

/* A struct encoding a point in a two-dimensional grid. */
struct pointT {
    int row, col;
};

/* A struct containing relevant game information. */
struct gameT {
    vector<string> world; // The playing field
    int numRows, numCols; // Size of the playing field

    deque<pointT> snake;  // The snake body
    int dx, dy;           // The snake direction

    int numEaten;         // How much food we've eaten.
};

/* The main program.  Initializes the world, then runs the simulation. */
int main() {
    gameT game;
    InitializeGame(game);
    RunSimulation(game);
    return 0;
}
```

Atop this program are the necessary `#include`s for the functions and objects we're using, followed by a list of constants for the game. The `pointT` and `gameT` structs are identical to those described above. `main` creates a `gameT` object, passes it into `InitializeGame` for initialization, and finally hands it to `RunSimulation` to play the game.

We'll begin by writing `InitializeGame` so that we can get a valid `gameT` for `RunSimulation`. But how should we initialize the game board? Should we use the same board every time, or let the user specify a level of their choosing? Both of these are resaonable, but for the this extended example we'll choose the latter. In particular, we'll specify a level file format, then let the user specify which file to load at runtime.

There are many possible file formats to choose from, but each must contain at least enough information to populate a `gameT` struct; that is, we need the world dimensions and layout, the starting position of the snake, and the direction of the snake. While I encourage you to experiment with different structures, we'll use a simple file format that encodes the world as a list of strings and the rest of the data as integers in a particular order. Here is one possible file:

File: *level.txt*

```
15 15
1 0
###############
#$             $#
#    #      #    #
#    #      #    #
#    #  $   #    #
#    #      #    #
#    #      #    #
#        *        #
#    #      #    #
#    #      #    #
#    #  $   #    #
#    #      #    #
#    #      #    #
#$             $#
###############
```

The first two numbers encode the number of rows and columns in the file, respectively. The next line contains the initial snake velocity as Δx, Δy. The remaining lines encode the game board, using the same characters we settled on for the world `vector`. We'll assume that the snake is initially of length one and its position is given by a `*` character.

There are two steps necessary to let the user choose the level layout. First, we need to prompt the user for the name of the file to open, reprompting until she chooses an actual file. Second, we need to parse the contents of the file into a `gameT` struct. In this example we won't check that the file is formatted correctly, though in professional code we would certainly need to check this. If you'd like some additional practice with the streams library, this would be an excellent exercise.

Let's start writing the function responsible for loading the file from disk, `InitializeGame`. Since we need to prompt the user for a filename until she enters a valid file, we'll begin writing:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        /* ... */
    }
    /* ... */
}
```

The `while(true)` loop will continuously prompt the user until she enters a valid file. Here, we assume that `GetLine()` is the version defined in the chapter on streams. Also, since we're now using the `ifstream` type, we'll need to `#include <fstream>` at the top of our program.

Now that the user has given us the a filename, we'll try opening it using the `.open()` member function. If the file opens successfully, we'll break out of the loop and start reading level data:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        /* ... */
    }
    /* ... */
}
```

If the file did *not* open, however, we need to report this to the user. Additionally, we have to make sure to reset the stream's error state, since opening a nonexistent file causes the stream to fail. Code for this is shown here:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    /* ... */
}
```

Now we need to parse the file data into a `gameT` struct. Since this is rather involved, we'll decompose it into a helper function called `LoadWorld`, then finish `InitializeGame` as follows:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    LoadWorld(game, input);
}
```

Notice that except for the call to `LoadWorld`, nothing in the code for `InitializeGame` actually pertains to our Snake game. In fact, the code we've written is a generic routine for opening a file specified by the user. We'll thus break this function down into two functions – `OpenUserFile`, which prompts the user for a filename, and `InitializeGame`, which opens the specified file, then hands it off to `LoadWorld`. This is shown here:

```
void OpenUserFile(ifstream& input) {
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for .c_str().
        if(input.is_open()) return;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
}

void InitializeGame(gameT& game) {
    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}
```

Let's begin working on `LoadWorld`. The first line of our file format encodes the number of rows and columns in the world, and we can read this data directly into the `gameT` struct, as seen here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    /* ... */
}
```

We've also resized the `vector` to hold `game.numRows` `string`s, guaranteeing that we have enough strings to store the entire world. This simplifies the implementation, as you'll see momentarily.

Next, we'll read the starting velocity for the snake, as shown here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    /* ... */
}
```

At this point, we've read in the parameters of the world, and need to start reading in the actual world data. Since each line of the file contains one row of the grid, we'll use `getline` for the remaining read operations. There's a catch, however. Recall that `getline` does not mix well with the stream extraction operator (`>>`), which we've used exclusively so far. In particular, the first call to `getline` after using the stream extraction operator will return the empty string because the newline character delimiting the data is still waiting to be read. To prevent this from gumming up the rest of our input operations, we'll call `getline` here on a dummy string to flush out the remaining newline:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    /* ... */
}
```

Now we're ready to start reading in world data. We'll read in `game.numRows` lines from the file directly into the `game.world vector`. Since earlier we `resize`d the `vector`, there already are enough `string`s to hold all the data we'll read. The reading code is shown below:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row) {
        getline(input, game.world[row]);
        /* ... */
    }

    /* ... */
}
```

Recall that somewhere in the level file is a single `*` character indicating where the snake begins. To make sure that we set up the snake correctly, after reading in a line of the world data we'll check to see if it contains a star and, if so, we'll populate the `game.snake deque` appropriately. Using the `.find()` member function on the `string` simplifies this task, as shown here:

```
    void LoadWorld(gameT& game, ifstream& input) {
        input >> game.numRows >> game.numCols;
        game.world.resize(game.numRows);

        input >> game.dx >> game.dy;

        string dummy;
        getline(input, dummy);

        for(int row = 0; row < game.numRows; ++row) {
            getline(input, game.world[row]);
            int col = game.world[row].find(kSnakeTile);
            if(col != string::npos) {
                pointT head;
                head.row = row;
                head.col = col;
                game.snake.push_back(head);
            }
        }

        /* ... */
    }
```

The syntax for creating and filling in the `pointT` data is a bit bulky here. When we cover classes in the second half of this course you'll see a much better way of creating this `pointT`. In the meantime, we can write a helper function to clean this code up, as shown here:

```
pointT MakePoint(int row, int col) {
  pointT result;
  result.row = row;
  result.col = col;
  return result;
}

    void LoadWorld(gameT& game, ifstream& input) {
        input >> game.numRows >> game.numCols;
        game.world.resize(game.numRows);

        input >> game.dx >> game.dy;

        string dummy;
        getline(input, dummy);

        for(int row = 0; row < game.numRows; ++row) {
            getline(input, game.world[row]);
            int col = game.world[row].find(kSnakeTile);
            if(col != string::npos)
                game.snake.push_back(MakePoint(row, col));
        }

        /* ... */
    }
```

There's one last step to take care of, and that's to ensure that we set the `numEaten` field to zero. This edit completes `LoadWorld` and the final version of the code is shown here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row) {
        getline(input, game.world[row]);
        int col = game.world[row].find(kSnakeTile);
        if(col != string::npos)
            game.snake.push_back(MakePoint(row, col));
    }

    game.numEaten = 0;
}
```

Great! We've just finished setup and it's now time to code up the actual game. We'll begin by coding a skeleton of `RunSimulation` which displays the current state of the game, runs the AI, and moves the snake:

```
void RunSimulation(gameT& game) {
    /* Keep looping while we haven't eaten too much. */
    while(game.numEaten < kMaxFood) {
        PrintWorld(game);     // Display the board
        PerformAI(game);      // Have the AI choose an action

        if(!MoveSnake(game)) // Move the snake and stop if we crashed.
            break;

        Pause();              // Pause so we can see what's going on.
    }
    DisplayResult(game);      // Tell the user what happened
}
```

We'll implement the functions referenced here out of order, starting with the simplest and moving to the most difficult. First, we'll begin by writing `Pause`, which stops for a short period of time to make the game seem more fluid. The particular implementation of `Pause` we'll use is a *busy loop*, a `while` loop that does nothing until enough time has elapsed. Busy loops are frowned upon in professional code because they waste CPU power, but for our purposes are perfectly acceptable.

The `<ctime>` header exports a function called `clock()` that returns the number of "clock ticks" that have elapsed since the program began. The duration of a clock tick varies from system to system, so C++ provides the constant `CLOCKS_PER_SEC` to convert clock ticks to seconds. We can use `clock` to implement a busy loop as follows:

1. Call `clock()` to get the current time in clock ticks and store the result.
2. Continuously call `clock()` and compare the result against the cached value. If enough time has passed, stop looping.

This can be coded as follows:

```
const double kWaitTime = 0.1; // Pause 0.1 seconds between frames
void Pause() {
    clock_t startTime = clock(); // clock_t is a type which holds clock ticks.

    /* This loop does nothing except loop and check how much time is left.
     * Note that we have to typecast startTime from clock_t to double so
     * that the division is correct.  The static_cast<double>(...) syntax
     * is the preferred C++ way of performing a typecast of this sort;
     * see the chapter on
     * inheritance for more information.
     */
    while(static_cast<double>(clock() - startTime) / CLOCKS_PER_SEC <
              kWaitTime);
}
```

Next, let's implement the `PrintWorld` function, which displays the current state of the world. We chose to represent the world as a `vector<string>` to simplify this code, and as you can see this design decision pays off well:

```
void PrintWorld(gameT& game) {
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

This implementation of `PrintWorld` is fine, but every time it executes it adds more text to the console instead of clearing what's already there. This makes it tricky to see what's happening. Unfortunately, standard C++ does not export a set of routines for manipulating the console. However, every major operating system exports its own console manipulation routines, primarily for developers working on a command line. For example, on a Linux system, typing `clear` into the console will clear its contents, while on Windows the command is `CLS`.

C++ absorbed C's standard library, including the `system` function (header file `<cstdlib>`). `system` executes an operating system-specific instruction as if you had typed it into your system's command line. This function can be very dangerous if used incorrectly,* but also greatly expands the power of C++. We will not cover how to use `system` in detail since it is platform-specific, but one particular application of system is to call the appropriate operating system function to clear the console. We can thus upgrade our implementation of `PrintWorld` as follows:

```
/* The string used to clear the display before printing the game board.
 * Windows systems should use "CLS"; Mac OS X or Linux users should use
 * "clear" instead.
 */
const string kClearCommand = "CLS";

void PrintWorld(gameT& game) {
    system(kClearCommand.c_str());
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

Because `system` is from the days of pure C, we have to use `.c_str()` to convert the string parameter into a C-style string before we can pass it into the function.

---

* In particular, calling `system` without checking that the parameters have been sanitized can let malicious users completely compromise your system. Take CS155 for more information on what sorts of attacks are possible.

The final quick function we'll write is `DisplayResult`, which is called after the game has ended to report whether the computer won or lost. This function is shown here:

```
void DisplayResult(gameT& game) {
    PrintWorld(game);
    if(game.numEaten == kMaxFood)
        cout << "The snake ate enough food and wins!" << endl;
    else
        cout << "Oh no!  The snake crashed!" << endl;
}
```

Now, on to the two tricky functions – `PerformAI`, which determines the snake's next move, and `MoveSnake`, which moves the snake and processes collisions. We'll begin with `PerformAI`.

Designing an AI that plays Snake intelligently is far beyond the scope of this class. However, it is feasible to build a rudimentary AI that plays reasonably well. Our particular AI works as follows: if the snake is about to collide with an object, the AI will turn the snake out of danger. Otherwise, the snake will continue on its current path, but has a percent chance to randomly change direction.

Let's begin by writing the code to check whether the snake will turn; that is, whether we're about to hit a wall or if the snake randomly decides to veer in a direction. We'll write a skeletal implementation of this code, then will implement the requisite functions. Our initial code is

```
const double kTurnRate = 0.2; // 20% chance to turn each step.
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that hits a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

Here we're calling three functions we haven't written yet – `GetNextPosition`, which computes the position of the head on the next iteration; `Crashed`, which returns whether the snake would crash if its head was in the given position; and `RandomChance`; which returns true with probability equal to the parameter. Before implementing the rest of `PerformAI`, let's knock these functions out so we can focus on the rest of the task at hand. We begin by implementing `GetNextPosition`. This function accepts as input the game state and returns the point that we will occupy on the next frame if we continue moving in our current direction. This function isn't particularly complex and is shown here:

```
pointT GetNextPosition(gameT& game) {
    /* Get the head position. */
    pointT result = game.snake.front();

    /* Increment the head position by the current direction. */
    result.row += game.dy;
    result.col += game.dx;
    return result;
}
```

The implementation of `Crashed` is similarly straightforward. The snake has crashed if it has gone out of bounds or if its head is on top of a wall or another part of the snake:

```
bool Crashed(pointT headPos, gameT& game) {
    return !InWorld(headPos, game) ||
           game.world[headPos.row][headPos.col] == kSnakeTile ||
           game.world[headPos.row][headPos.col] == kWallTile;
}
```

Here, `InWorld` returns whether the point is in bounds and is defined as

```
bool InWorld(pointT& pt, gameT& game) {
    return pt.col >= 0 &&
           pt.row >= 0 &&
           pt.col < game.numCols &&
           pt.row < game.numRows;
}
```

Next, we need to implement `RandomChance`. In CS106B/X we provide you a header file, `random.h`, that exports this function. However, `random.h` is not a standard C++ header file and thus we will not use it here. Instead, we will use C++'s `rand` and `srand` functions, also exported by `<cstdlib>`, to implement `RandomChance`. `rand()` returns a pseudorandom number in the range [0, `RAND_MAX`], where `RAND_MAX` is usually $2^{15} - 1$. srand seeds the random number generator with a value that determines which values are returned by `rand`. One common technique is to use the `time` function, which returns the current system time, as the seed for `srand` since different runs of the program will yield different random seeds. Traditionally, you will only call `srand` once per program, preferably during initialization. We'll thus modify `InitializeGame` so that it calls `srand` in addition to its other functionality:

```
void InitializeGame(gameT& game) {
    /* Seed the randomizer.  The static_cast converts the result of time(NULL)
     * from time_t to the unsigned int required by srand.  This line is
     * idiomatic C++.
     */
    srand(static_cast<unsigned int>(time(NULL)));

    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}
```

Now, let's implement `RandomChance`. To write this function, we'll call rand to obtain a value in the range [0, `RAND_MAX`], then divide it by `RAND_MAX + 1.0` to get a value in the range [0, 1). We can then return whether this value is less than the input probability. This yields true with the specified probability; try convincing yourself that this works if it doesn't immediately seem obvious. This is a common technique and in fact is how the CS106B/X `RandomChance` function is implemented.

`RandomChance` is shown here:

```
bool RandomChance(double probability) {
    return (rand() / (RAND_MAX + 1.0)) < probability;
}
```

Notice that we added 1.0 to `RAND_MAX`. This both adds the +1 necessary from the above discussion and implicitly converts the denominator into a `double`, which is necessary to avoid integer truncation.

Phew! Apologies for the lengthy detour – let's get back to writing the AI! Recall that we've written this code so far:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

We now need to implement the logic for turning the snake left or right. First, we'll figure out in what positions the snake's head would be if we turned left or right. Then, based on which of these positions are safe, we'll pick a direction to turn. To avoid code duplication, we'll modify our implementation of `GetNextPosition` so that the caller can specify the direction of motion, rather than relying on the `gameT`'s stored direction. The modified version of `GetNextPosition` is shown here:

```
pointT GetNextPosition(gameT& game, int dx, int dy) {
    /* Get the head position. */
    gameT result = game.snake.front();

    /* Increment the head position by the specified direction. */
    result.row += dy;
    result.col += dx;
    return result;
}
```

We'll need to modify `PerformAI` to pass in the proper parameters to `GetNextPosition`, as shown here:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

Now, let's write the rest of this code. Given that the snake's velocity is (`game.dx`, `game.dy`), what velocities would we move at if we were heading ninety degrees to the left or right? Using some basic linear algebra,* if our current heading is along dx and dy, then the headings after turning left and right from our current heading are be given by

$$dx_{left} = -dy$$
$$dy_{left} = dx$$

$$dx_{right} = dy$$
$$dy_{right} = -dx$$

Using these equalities, we can write the following code, which determines what bearings are available and whether it's safe to turn left or right:

* This is the result of multiplying the vector $(dx, dy)^T$ by a rotation matrix for either $+\pi/2$ or $-\pi/2$ radians.

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        /* ... */
    }
}
```

Now, we'll decide which direction to turn. If we can only turn one direction, we will choose that direction. If we can't turn at all, we will do nothing. Finally, if we can turn either direction, we'll pick a direction randomly. We will store which direction to turn in a boolean variable called `willTurnLeft` which is `true` if we will turn left and `false` if we will turn right. This is shown here:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        /* ... */
    }
}
```

Finally, we'll update our direction vector based on our choice:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        game.dx = willTurnLeft? leftDx : rightDx;
        game.dy = willTurnLeft? leftDy : rightDy;
    }
}
```

If you're not familiar with the `?:` operator, the syntax is as follows:

```
expression ? result-if-true : result-if-false
```

Here, this means that we'll set `game.dx` to `leftDx` if `willTurnLeft` is true and to `rightDx` otherwise.

We now have a working version of `PerformAI`. Our resulting implementation is not particularly dense, and most of the work is factored out into the helper functions.

There is one task left – implementing `MoveSnake`. Recall that `MoveSnake` moves the snake one step forward on its path. If the snake crashes, the function returns `false` to indicate that the game is over. Otherwise, the function returns `true`.

The first thing to do in `MoveSnake` is to figure out where the snake's head will be after taking a step. Thanks to `GetNextPosition`, this has already been taken care of for us:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    /* ... */
}
```

Now, if we crashed into something (either by falling off the map or by hitting an object), we'll return `false` so that the main loop can terminate:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    /* ... */
}
```

Next, we need to check to see if we ate some food. We'll store this in a `bool` variable for now, since the logic for processing food will come a bit later:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    /* ... */
}
```

Now, let's update the snake's head. We need to update the `world vector` so that the user can see that the snake's head is in a new square, and also need to update the `snake deque` so that the snake's head is now given by the new position. This is shown here:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    /* ... */
}
```

Finally, it's time to move the snake's tail forward one step. However, if we've eaten any food, we will leave the tail as-is so that the snake grows by one tile. We'll also put food someplace else on the map so the snake has a new objective. The code for this is shown here:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    if(!isFood) {
        game.world[game.snake.back().row][game.snake.back().col] = kEmptyTile;
        game.snake.pop_back();
    } else {
        ++game.numEaten;
        PlaceFood(game);
    }
    return true;
}
```

We're nearing the home stretch – all that's left to do is to implement `PlaceFood` and we're done! This function is simple – we'll just sit in a loop picking random locations on the board until we find an empty spot, then will put a piece of food there. To generate a random location on the board, we'll scale `rand()` down to the proper range using the modulus (`%`) operator. For example, on a world with four rows and ten columns, we'd pick as a row `rand() % 4` and as a column `col() % 10`. The code for this function is shown here:

```
void PlaceFood(gameT& game) {
    while(true) {
        int row = rand() % game.numRows;
        int col = rand() % game.numCols;

        /* If the specified position is empty, place the food there. */
        if(game.world[row][col] == kEmptyTile) {
            game.world[row][col] = kFoodTile;
            return;
        }
    }
}
```

More To Explore

There's so much to explore with the STL that we could easily fill the rest of the course reader with STL content. If you're interested in some more advanced topics relating to this material and the STL in general, consider reading on these topics:

1. **`stack` and `queue`**: The `vector` and `deque` pack a lot of firepower and can solve a wide array of problems. However, in some cases, you may want to use a container with a more restricted set of operations. For these purposes, the STL exports two *container adapters*, containers that export functionality similar to a `vector` or `deque` but with a slight reduction in power. The first of these is the `stack`, which only lets you view the final element of a sequence; the second is the `queue`, which is similar to line at a ticket counter. If you plan on pursuing C++ more seriously, you should take the time to look over what these container adapters have to offer.

2. **`valarray`**: The `valarray` class is similar to a `vector` in that it's a managed array that can hold elements of any type. However, unlike `vector`, `valarray` is designed for numerical computations. `valarray`s are fixed-size and have intrinsic support for mathematical operators. For example, you can use the syntax `myValArray *= 2` to multiply all of the entries in a `valarray` by two. If you're interested in numeric or computational programming, consider looking into the `valarray`.

3. There's an excellent article online comparing the performances of the `vector` and `deque` containers. If you're interested, you can see it at http://www.codeproject.com/vcpp/stl/vector_vs_deque.asp.

Practice Problems

1. List two differences between the `vector`'s `push_back` and `resize` member functions.

2. What header files do you need to `#include` to use `vector`? `deque`?

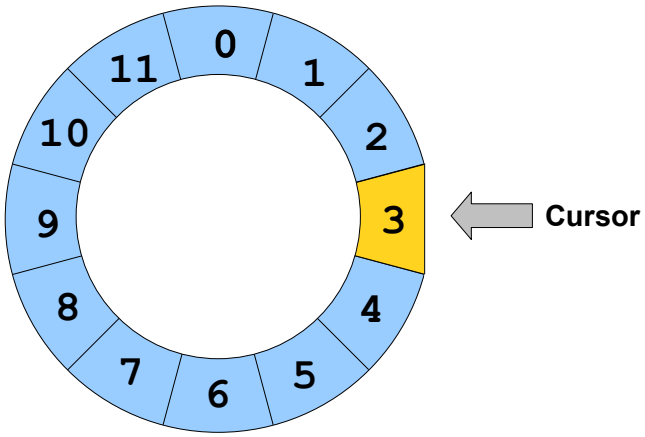
3. How do you tell how many elements are in a `vector`? In a `deque`?

4. How do you remove the first element from a `vector`? From a `deque`?

5. Write a function called `LinesFromFile` which takes in a `string` containing a filename and returns a `vector<string>` containing all of the lines of text in the file in the order in which they appear. If the file does not exist, you can return an empty `vector`. *(Hint: look at the code for reading the world file in the Snake example and see if you can modify it appropriately)*

6. Update the code for the sorting program so that it sorts elements in *descending* order instead of *ascending* order.

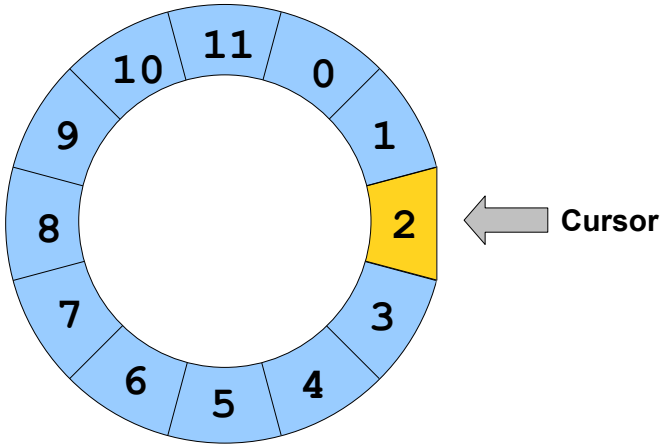
7. One use for the `deque` container is to create a *ring buffer*. Unlike the linear `vector` and `deque` containers you saw in this chapter, a ring buffer is circular. Here's an illustration of a ring buffer:



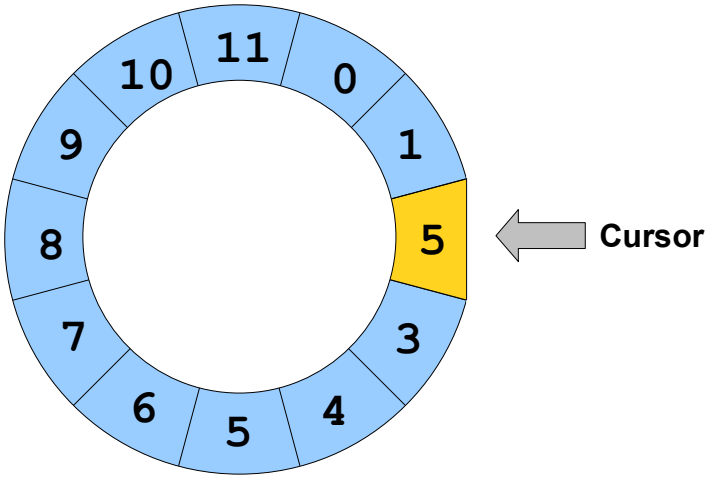
A ring buffer consists of a circular ring of elements with a *cursor* which selects some particular element out of the buffer. The four main operations on a ring buffer are as follows:

- Rotate the ring clockwise
- Rotate the ring counterclockwise
- Read the value at the cursor.
- Write the value at the cursor.

For example, given the above ring buffer, the result of rotating the ring clockwise would be

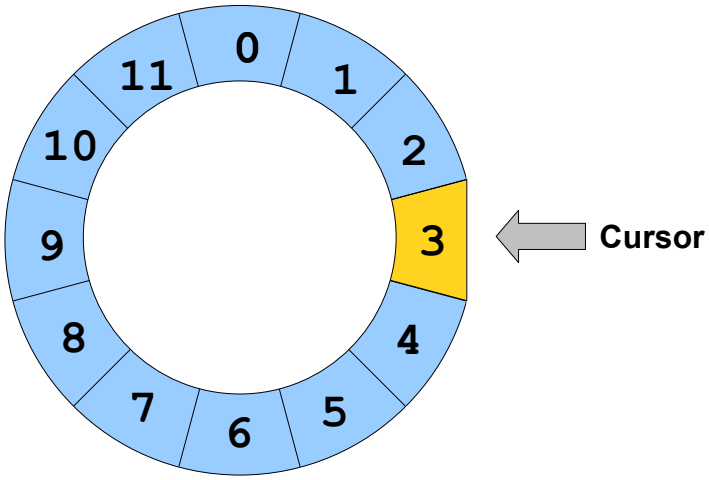


If we then wrote the value 5 to the location specified by the cursor, the buffer would look like this:

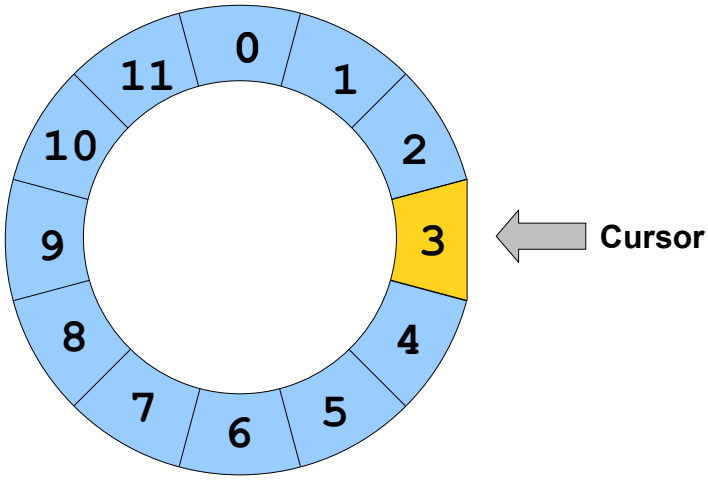


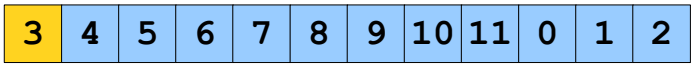
There is a particularly elegant construction which enables us to build a ring buffer out of a `deque`. The basic idea is to "unroll" the ring buffer into a linear sequence, then use a combination of `push_front`, `pop_front`, `push_back`, and `pop_back` to simulate moving the cursor to the left or to the right. This technique of simulating one data structure using another is ubiquitous in computer science, and many important results in computability theory use constructions of this form.

To see exactly how the construction works, suppose that we have the following ring buffer:

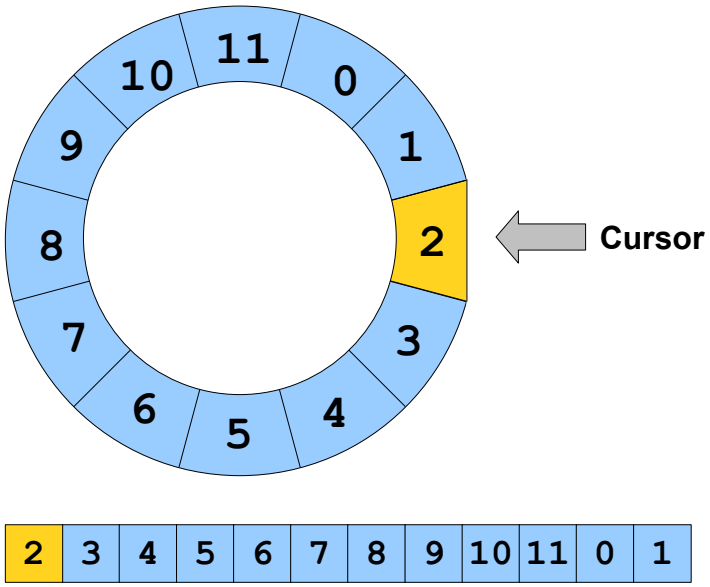


We can then represent this using a `deque` as follows:

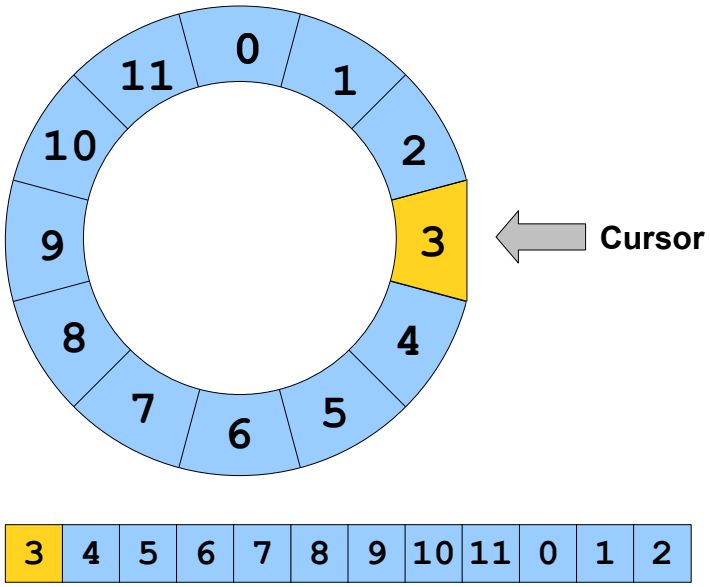




That is, the first element of the `deque` is the element under the cursor, and the rest of the elements in the `deque` are the elements in the ring buffer formed by walking clockwise around the ring buffer. Given this construction, we can simulate rotating the ring one position clockwise by moving the last element of the `deque` onto the front, as shown here:



Similarly, to move the cursor one step counterclockwise, we move the element at the end of the `deque` onto the front, as shown here:



Write a pair of functions `CursorClockwise` and `CursorCounterClockwise` which take in a `deque` representing a ring buffer and update the `deque` by simulating a cursor move in either direction. Then write functions `CursorRead` and `CursorWrite` which read and write the element stored at the cursor. You've just shown how to represent one data structure using another!

8. As mentioned earlier, the `deque` outperforms the `vector` when inserting and removing elements at the end of the container. However, the `vector` has a useful member function called `reserve` that can be used to increase its performance against the `deque` in certain circumstances. The `reserve` function accepts an integer as a parameter and acts as a sort of "size hint" to the `vector`. Behind the scenes, `reserve` works by allocating additional storage space for the `vector` elements, reducing the number of times that the `vector` has to ask for more storage space. Once you have called `reserve`, as long as the size of the `vector` is less than the number of elements you have reserved, calls to `push_back` and `insert` on the `vector` will execute more quickly than normal. Once the `vector` hits the size you reserved, these operations revert to their original speed.*

   Write a program that uses `push_back` to insert a large number of `string`s into two different `vector`s – one which has had `reserve` called on it and one which hasn't – as well as a `deque`. The exact number and content of strings is up to you, but large numbers of long strings will give the most impressive results. Use the `clock()` function exported by `<ctime>` to compute how long it takes to finish inserting the `string`s. Now repeat this trial, but insert the elements at the beginning of the container rather than the end. Did calling `reserve` help to make the `vector` more competitive against the `deque`?

9. In this next problem we'll explore a simple encryption algorithm called the *Vigenère cipher* and how to implement it using the STL containers.

   One of the oldest known ciphers is the *Caesar cipher*, named for Julius Caesar, who allegedly employed it. The idea is simple. We pick a secret number between 1 and 26, inclusive, then encrypt the input string by replacing each letter with the letter that comes that many spaces after it. If this pushes us off the end of the alphabet, we wrap around to the start of the alphabet. For example, if we were given the string "The cookies are in the fridge" and picked the number 1, we would end up with the resulting string "Uif dppljft bsf jo uif gsjehf." To decrypt the string, we simply need to push each letter backwards by one spot.

   The Caesar cipher is an extremely weak form of encryption; it was broken in the ninth century by the Arab polymath al-Kindi. The problem is that the cipher preserves the relative frequencies of each of the letters in the source text. Not all letters appear in English with equal frequency – e and t are far more common than q or w, for example – and by looking at the relative letter frequencies in the encrypted text it is possible to determine which letter in the encrypted text corresponds to a letter in the source text and to recover the key.

   The problem with the Caesar cipher is that it preserves letter frequencies because each letter is transformed using the same key. But what if we were to use *multiple* keys while encrypting the message? That is, we might encrypt the first letter with one key, the second with another, the third with yet another, etc. One way of doing this is to pick a sequence of numbers, then cycle through them while encrypting the text. For example, let's suppose that we want to encrypt the above message using the key string 1, 3, 7. Then we would do the following:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7
U	**K**	**L**	**D**	**R**	**V**	**L**	**L**	**L**	**T**	**D**	**Y**	**F**	**L**	**U**	**U**	**K**	**L**	**G**	**U**	**P**	**E**	**J**	**L**

   Notice that the letters KIE from COOKIES are all mapped to the letter L, making cryptanalysis much more difficult. This particular encryption system is the Vigenère cipher.

* Calling `push_back` n times always takes O(n) time, whether or not you call `reserve`. However, calling `reserve` reduces the constant term in the big-O to a smaller value, meaning that the overall execution time is lower.

Now, let's consider what would happen if we wanted to implement this algorithm in C++ to work on arbitrary strings. Strings in C++ are composed of individual chars, which can take on (typically) one of 256 different values. If we had a list of integer keys, we could encrypt a string using the Vigenère cipher by simply cycling through those keys and incrementing the appropriate letters of the string. In fact, the algorithm is quite simple. We iterate over the characters of the string, at each point incrementing the character by the current key and then rotating the keys one cycle.

a. Suppose that we want to represent a list of integer keys that can easily be cycled; that is, we want to efficiently support moving the first element of the list to the back. Of the containers covered in this chapter (`vector` and `deque`), which have the best support for this operation? ♦

b. Based on your decision, implement a function `VigenereEncrypt` that accepts a string and a list of `int` keys stored in the container of your choice, then encrypts the string using the Vigenère cipher. ♦