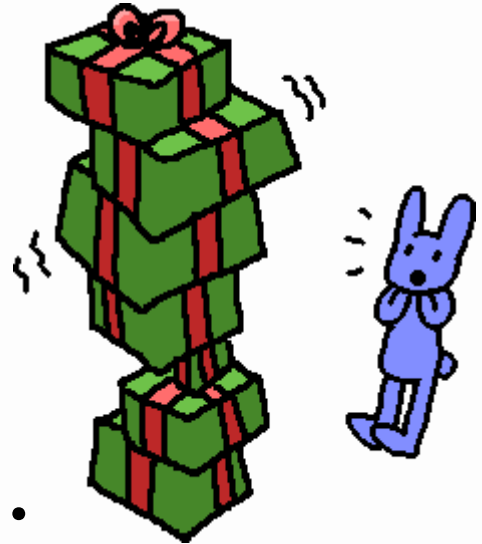


The Standard Template Library

About

C++ has some data structures already pre-built and available within the standard library that we can use to get familiar a bit with how these structures work.

Before we get into the details of implementing these data structures, it might be useful to see how they can actually be used first...



About

Wait, if data structures already exist, why are we going to spend all semester learning how to write 'em?!



About

- Well... learning about data structures & algorithm analysis is a big part of computer science.

About

- Well... learning about data structures & algorithm analysis is a big part of computer science.
- You also need to know the inner-workings of these structures, so that when you're implementing solutions, you can choose the best structure for your particular problem – it isn't one size fits all!

About

- Well... learning about data structures & algorithm analysis is a big part of computer science.
- You also need to know the inner-workings of these structures, so that when you're implementing solutions, you can choose the best structure for your particular problem – it isn't one size fits all!
- You might need to create your own data structures down the line, or customize an existing one!

About

- Well... learning about data structures & algorithm analysis is a big part of computer science.
- You also need to know the inner-workings of these structures, so that when you're implementing solutions, you can choose the best structure for your particular problem – it isn't one size fits all!
- You might need to create your own data structures down the line, or customize an existing one!
- You'll be asked about them during job interviews. Interviewers love asking about data structures.



About

So let's see how some of these structures work by utilizing the structures available in the Standard Template Library.

In particular...

- **STL Vector**
 - **STL List**
 - **STL Stack**
 - **STL Queue**
 - **STL Map**

Topics

1. Vectors

4. Stacks

2. Lists

5. Maps

3. Queues

I. STL Vector

In some ways, vector objects are similar to arrays, which you may have used in previous classes.

You can access specific items of the vector with the subscript operator, []

```
cout << "Price: " << itemPrices[5] << endl;
```

Notes

Vectors are implemented with arrays, so you can access specific elements with the subscript operator [].

I. STL Vector

A perk of the vector object is that it handles resizing on its own.

Recall that with a static array, we had to know what its size was at compile time, and it couldn't be resized!

```
int sadArray[100];
for ( int i = 0; i < 100; i++ )
{
    sadArray[i] = i * 2;
}
cout << "sadArray is full and cannot store any more...";
```

Notes

Vectors are implemented with arrays, so you can access specific elements with the subscript operator [].

Vectors handle resizing its internal array automatically.

I. STL Vector

With a STL vector, it handles resizing on its own (behind-the-scenes!), so we don't have to worry about it – we can just keep adding items on to it!

```
vector<float> itemPrices;  
  
itemPrices.push_back( 9.99 );  
itemPrices.push_back( 7.99 );  
itemPrices.push_back( 6.99 );
```

Someone else has implemented the vector for us, so that we don't have to worry about *how* it works, it just does!



The vector's **push_back** function is how we add items into the “array”.

Notes

Vectors are implemented with arrays, so you can access specific elements with the subscript operator [].

Vectors handle resizing its internal array automatically.

push_back is the function used to insert items into the vector.

I. STL Vector

A Vector can store any data-type...

```
vector<int> myNumbers;  
myNumbers.push_back( 20 );
```

```
vector<string> studentNames;  
studentNames.push_back( "Bob" );
```

```
vector<float> itemPrices;  
itemPrices.push_back( 9.99 );
```

Notes

Vectors are implemented with arrays, so you can access specific elements with the subscript operator [].

Vectors handle resizing its internal array automatically.

push_back is the function used to insert items into the vector.

I. STL Vector

A Vector can store any data-type...

```
vector<int> myNumbers;  
myNumbers.push_back( 20 );  
  
vector<string> studentNames;  
studentNames.push_back( "Bob" );  
  
vector<float> itemPrices;  
itemPrices.push_back( 9.99 );
```

The <int>, <string>, and <float> bits of code are because vector has been implemented as a **template**.

If you haven't covered templates before, or don't quite remember how they work, don't worry – we will go over them more later on.

Notes

Vectors can contain *any* data-type.

I. STL Vector

A Vector can store any data-type...

```
struct CoordPair  
{  
    float x, y;  
};
```

```
vector<CoordPair> coordinatePairs;
```

If we write a **struct** or a **class**, a vector can even store those!

Notes

Vectors can contain *any* data-type.

During declaration, the data-type the vector will store goes in < >.

I. STL Vector

Some handy functions of a vector are...

- **push_back**
Insert an item at the end of the vector
- **size**
Returns the amount of items in the vector
- **empty**
Returns whether the vector is empty or not (size == 0?)
- **operator[]**
Access an item in the vector at any index
- **clear**
Clears out all the elements of the vector.

Notes

Functions:
(*"T" refers to the data-type the vector stores*)

- `void push_back (T& item)`
- `size_type size()`
- `bool empty()`
- `T& operator[] (size_type n)`
- `void clear()`

I. STL *Vector*

Vectors are implemented with a **dynamic array**, which is why we can treat them *like* arrays.

By using a class to *wrap-around* a dynamic array, we can automatically have the structure handle resizing whenever space runs out, or when the user tries to use the subscript operator `[]`.

Notes

Functions:

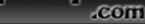
(“*T*” refers to the data-type the vector stores)

- `void push_back (T& item)`
- `size_type size()`
- `bool empty()`
- `T& operator[] (size_type n)`
- `void clear()`

I. STL *Vector*

cplusplus.com has useful reference pages on all the Standard Template Library structures that you can use to get more information on the vector and its functions.

<http://www.cplusplus.com/reference/vector/vector/>



Search: Go

Reference <vector> vector

C++

 Information
 Tutorials
 Reference
 Articles
 Forum

Reference

C library:

<cassert> (assert.h)

<cctype> (ctype.h)

<errno> (errno.h)

<fcntl> (fcntl.h)

<float> (float.h)

<inttypes> (inttypes.h)

<iso646> (iso646.h)

<limits> (limits.h)

<locale> (locale.h)

<math> (math.h)

<setjmp> (setjmp.h)

<signal> (signal.h)

<stdarg> (stdarg.h)

<stdbool> (stdbool.h)

class template

std::vector

<vector>

template < class T, class Alloc = allocator<T> > class vector; // generic template

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see [push_back](#)).

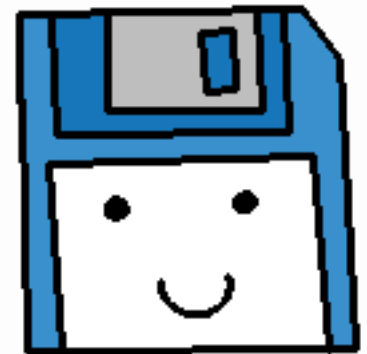
Notes

Functions:
(*"T" refers to the data-type the vector stores*)

- `void push_back (T& item)`
- `size_type size()`
- `bool empty()`
- `T& operator[] (size_type n)`
- `void clear()`

I. STL *Vector*

Example Time!



I. STL Vector

Using a Static Array...

```
int myNumbers[3];  
myNumbers[0] = 2;  
myNumbers[1] = 3;  
myNumbers[2] = 6;  
  
for ( int i = 0; i < 3; i++ )  
{  
    cout << i << "\t" << myNumbers[i] << endl;  
}
```

Declaring an int array

Assigning values to array

Displaying elements
of the array

I. STL Vector

Using a STL Vector...

```
vector<int> myNumbers2;  
myNumbers2.push_back( 2 );  
myNumbers2.push_back( 3 );  
myNumbers2.push_back( 6 );  
  
myNumbers2[0] = 1;  
  
for ( unsigned int i = 0; i < myNumbers2.size(); i++ )  
{  
    cout << i << "\t" << myNumbers2[i] << endl;  
}
```

Declaring an int vector

Assigning values to vector

Overwriting value at index 0

Displaying elements
of the vector

The `size()` function returns an *unsigned int* value, which is essentially a non-negative integer. This is why my for loop uses an unsigned int for *i*.

2. STL List

Lists also store a linear series of data, but they're a little different from vectors.

For one, you cannot randomly access data with the subscript operator [].

Generally, to step through a list, you have to start at the beginning and keep stepping through, one at a time.

The STL List does contain a `sort()` function and `reverse()` function, though!

Notes

Lists are also a *linear* data type, but you cannot *randomly access* items in the list because of how it is implemented.

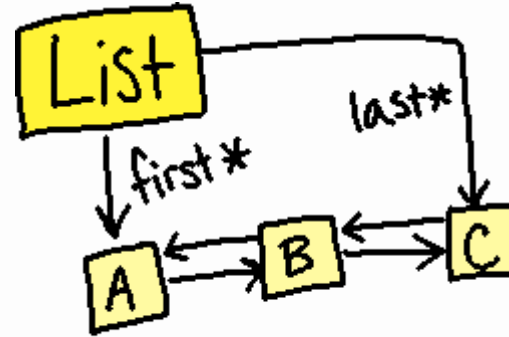
2. STL List

We cannot randomly access data in a List because it isn't implemented with an array, like vector is.

STL Lists use **pointers**. The list keeps track of what its starting element is, and each element points to the next element in the list.

Therefore, unlike an array, the elements are not in contiguous memory slots.

(This is why it's important to stay familiar with pointers for this class!)



Notes

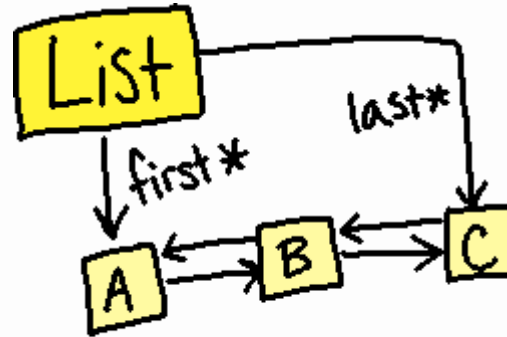
Lists are also a *linear* data type, but you cannot *randomly access* items in the list because of how it is implemented.

Lists are implemented with Nodes that contain pointers to the previous and next items in the list.

2. STL List

We will discuss the pointer and memory aspect of lists later on, once we're implementing linked lists.

For now...
onto the STL List functionality!



Notes

Lists are also a *linear* data type, but you cannot *randomly access* items in the list because of how it is implemented.

Lists are implemented with Nodes that contain pointers to the previous and next items in the list.

2. STL List

- **push_back**
Insert an item at the end of the list
- **size**
Returns the amount of items in the list
- **empty**
Returns whether the list is empty or not (size == 0?)
- **clear**
Clears out all the elements of the list.
- **sort**
Sorts the elements of the list
- **reverse**
Reverses the order of elements in the list.

Notes

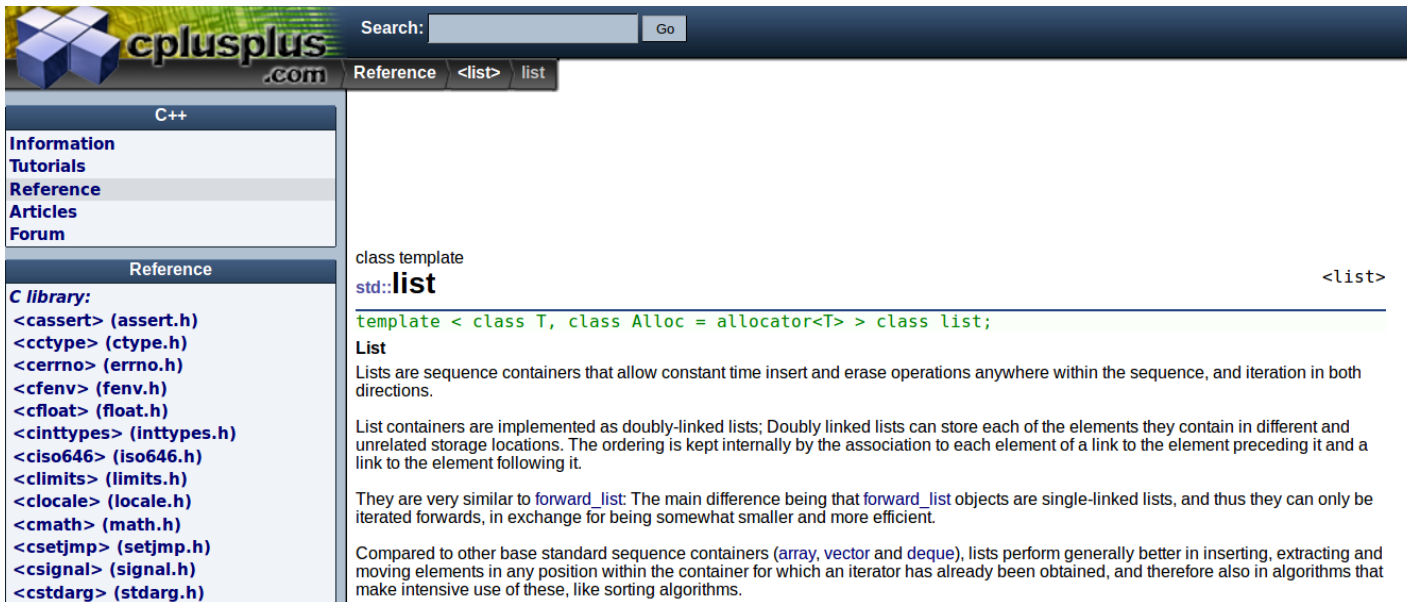
Functions:
(*"T" refers to the data-type the list stores*)

- `void push_back (T& item)`
- `size_type size()`
- `bool empty()`
- `void clear()`
- `void sort()`
- `void reverse()`

2. STL List

Access documentation about the STL List at **cplusplus.com**:

<http://www.cplusplus.com/reference/list/list/>



The screenshot shows the Cplusplus.com website interface. At the top is a search bar with the text "Search:" and a "Go" button. Below the search bar are navigation tabs: "Reference", "<list>", and "list". The left sidebar contains a menu with links to "C++", "Information", "Tutorials", "Reference", "Articles", and "Forum". Under the "Reference" section, there is a list of C library headers: <cassert> (assert.h), <cctype> (ctype.h), <cerrno> (errno.h), <cfenv> (fenv.h), <cfloat> (float.h), < cinttypes> (inttypes.h), <ciso646> (iso646.h), <climits> (limits.h), <locale> (locale.h), <cmath> (math.h), < csetjmp> (setjmp.h), <csignal> (signal.h), and <cstdarg> (stdarg.h). The main content area displays the documentation for the `std::list` class template. It includes the C++ code for the class template: `template < class T, class Alloc = allocator<T> > class list;`. Below the code, there is a description of the `list` container, stating that it is a sequence container that allows constant time insert and erase operations anywhere within the sequence, and iteration in both directions. It also mentions that `list` containers are implemented as doubly-linked lists, which can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it. A comparison is made with `forward_list`, noting that `forward_list` objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient. Finally, it states that compared to other base standard sequence containers (`array`, `vector`, and `deque`), `list`s perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

Notes

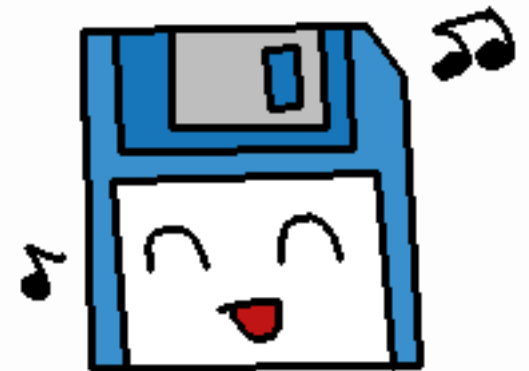
Functions:

("T" refers to the data-type the list stores)

- `void push_back (T& item)`
- `size_type size()`
- `bool empty()`
- `void clear()`
- `void sort()`
- `void reverse()`

2. STL List

Example Time!



2. STL List

Using a STL List...

```
list<string> states;  
  
states.push_back( "Maine" );  
states.push_back( "Hawaii" );  
states.push_back( "Kansas" );  
states.push_back( "Missouri" );  
states.push_back( "Washington" );  
states.push_back( "Alaska" );  
  
states.sort();  
states.reverse();  
  
for ( list<string>::iterator it = states.begin();  
      it != states.end(); it++ )  
{  
    cout << *it << endl;  
}
```

Declaring a list of strings

Adding strings to the list

Sort the list

Reverse the list

Use iterators to display
the contents of the list

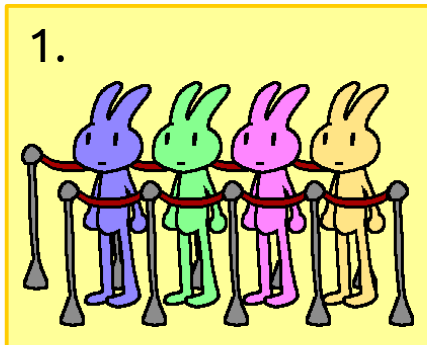
To iterate through the STL List, we must use an iterator. You probably haven't seen one of these before, but it's a topic for another time.

3. STL Queue

A queue is a type of **restricted-access** data structure.

It stores its contents in a linear order, but you're only able to **remove items at the front of the queue**, and to **add new items to the back of the queue**.

A queue is known as a **First In, First Out (FIFO)** structure.



Notes

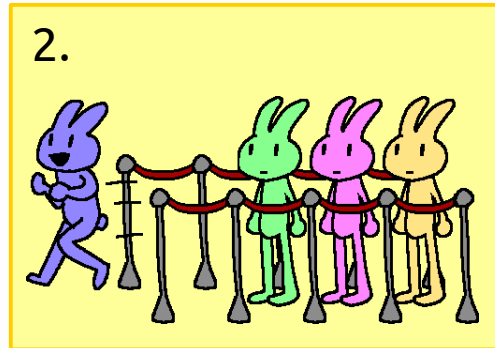
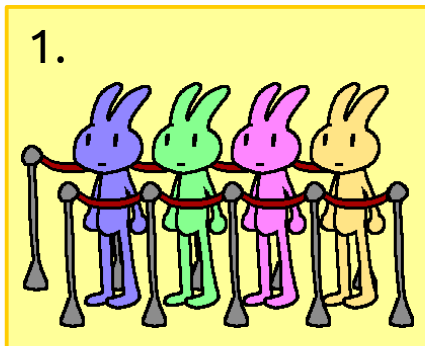
Queues are first-in-first-out.

3. STL Queue

The first item that enters the queue, who sits at the front of the line, is the first one to get removed, just like in a grocery-store line.

Notes

Queues are first-in-first-out.

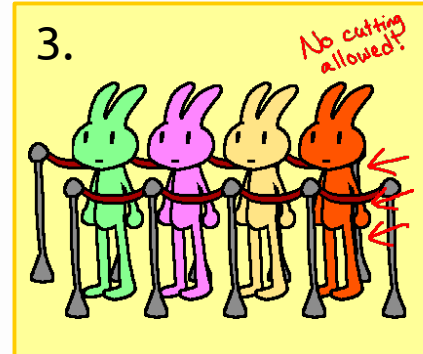
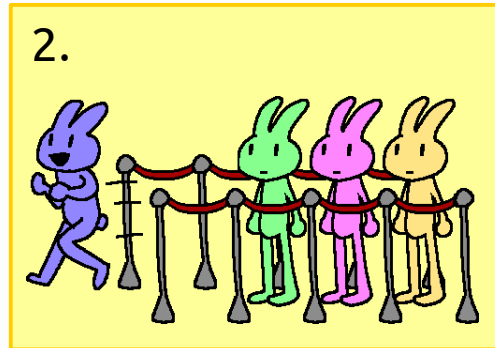
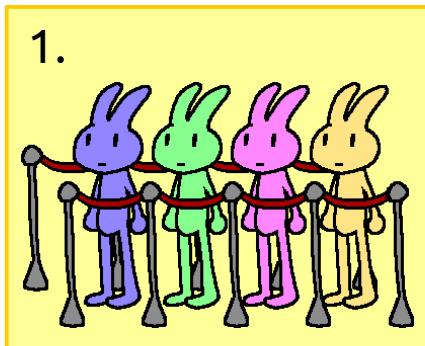


3. STL Queue

And when a new item is added to the queue, it enters at the end (or back) of the queue.

Notes

Queues are first-in-first-out.



3. STL Queue

Some handy queue functions are...

- **push**
Pushes an item to the back of the queue.
- **pop**
Removes an item from the front of the queue.
- **front**
Returns the item that is at the front of the queue.
- **size**
Returns the amount of items in the queue.
- **empty**
Returns whether the queue is empty or not.

Notes

Functions:

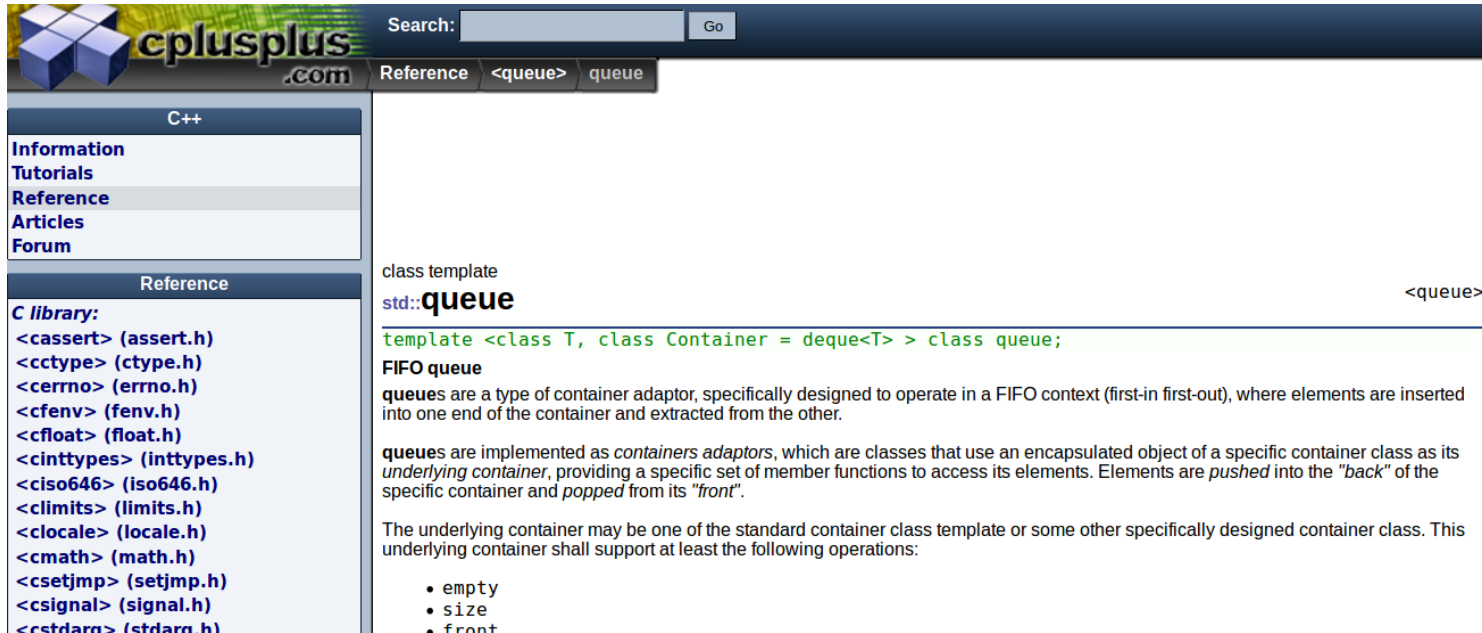
("T" refers to the data-type the queue stores)

- `void push(T& item)`
- `void pop()`
- `T& front()`
- `size_type size()`
- `bool empty()`

3. STL Queue

Access documentation about the STL Queue at **cplusplus.com**:

<http://www.cplusplus.com/reference/queue/queue/>



The screenshot shows the Cplusplus.com website. The top navigation bar includes a search box and a 'Go' button. Below the navigation bar, there are tabs for 'Reference', '<queue>', and 'queue'. The left sidebar contains a menu with 'C++' and 'Reference' sections. The 'C++' section includes links to 'Information', 'Tutorials', 'Reference', 'Articles', and 'Forum'. The 'Reference' section includes a list of C library headers: <cassert> (assert.h), <cctype> (ctype.h), <cerrno> (errno.h), <cfenv> (fenv.h), <cmath> (math.h), <climits> (limits.h), <locale> (locale.h), <math> (math.h), <setjmp> (setjmp.h), <signal> (signal.h), and <stdarg> (stdarg.h). The main content area displays the 'std::queue' class template documentation. It includes the class template declaration: `template <class T, class Container = deque<T> > class queue;`. Below this, it describes 'FIFO queue' and explains that 'queues' are a type of container adaptor designed to operate in a FIFO context. It also mentions that 'queues' are implemented as 'containers adaptors' and provides a list of operations: `empty`, `size`, and `front`.

Notes

Functions:

("T" refers to the data-type the queue stores)

- `void push(T& item)`
- `void pop()`
- `T& front()`
- `size_type size()`
- `bool empty()`

3. STL Queue

Example Time!



3. STL Queue

Using a STL Queue...

```
queue<string> commands;

commands.push( "mkdir newFolder" );
commands.push( "touch newFile" );
commands.push( "ping google.com -c 5" );
commands.push( "ping yahoo.com -c 3" );

while ( !commands.empty() )
{
    string command = commands.front();
    cout << endl << command << endl;
    system( command.c_str() );
    commands.pop();
}
```

Declaring a queue of strings

Pushing strings into the queue

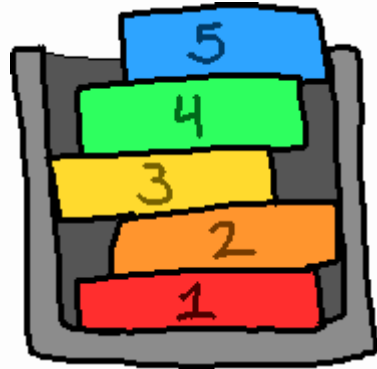
Checking if the queue is empty

Accessing the front-most item

Removing the front-most item

4. STL Stack

A Stack is a type of data structure that is linear, like a list or vector is, but it also restricts access to the internal data.



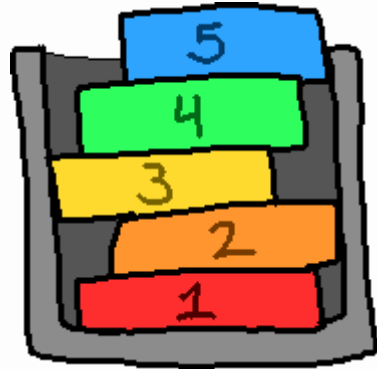
The main characteristic of a stack is that it is a First In Last Out (or) Last In First Out structure.

Notes

Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

At any time, you're only able to access one item from the stack – the top-most item.



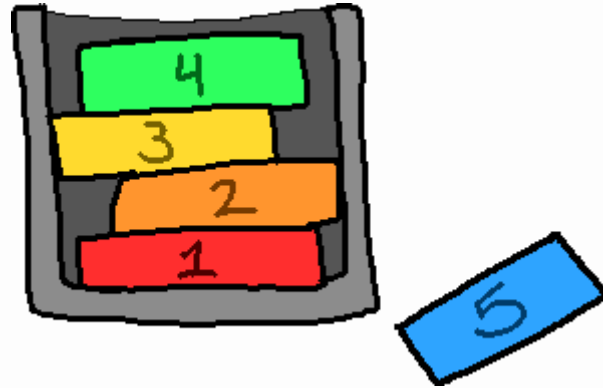
And, as items are pushed onto the stack, the older items are on the bottom, and the newer items are on the top.

Notes

Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

As you remove items from the stack, you pull the newest item that was added to the stack.



The first item on the stack is the last one to be removed.

Notes

Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

One example of using a Stack is to keep track of moves in a game of tic-tac-toe.



Notes

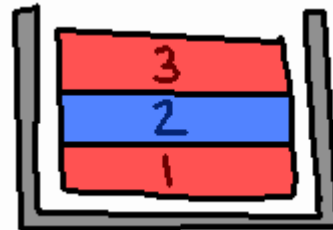
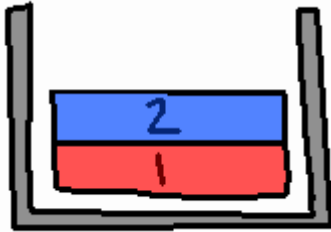
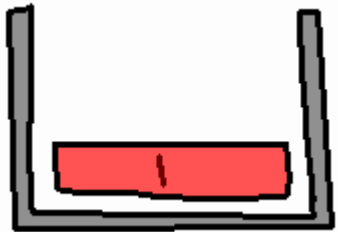
Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

One example of using a Stack is to keep track of moves in a game of tic-tac-toe.



Every time a move is made, we could push the game board's current state onto a stack...



Notes

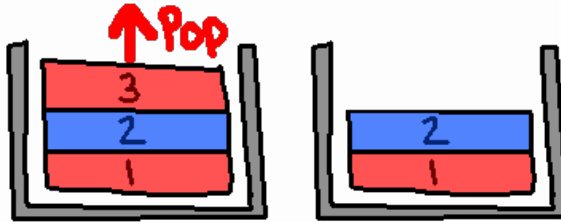
Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

One example of using a Stack is to keep track of moves in a game of tic-tac-toe.



Then if we want to undo a turn, we can pop the most recent state off the stack...



Notes

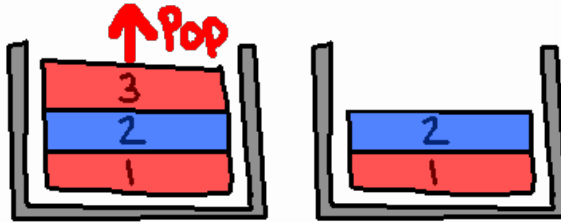
Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

One example of using a Stack is to keep track of moves in a game of tic-tac-toe.



Then if we want to undo a turn, we can pop the most recent state off the stack...



And the game board reverts to the state before.



Notes

Stacks are a restricted-access data type that is Last-In-First-Out

4. STL Stack

Some handy stack functions are...

- **push**
Pushes an item to the top of the stack.
- **pop**
Removes an item from the top of the stack.
- **top**
Returns the item that is at the top of the stack.
- **size**
Returns the amount of items in the stack.
- **empty**
Returns whether the stack is empty or not.

Notes

Functions:

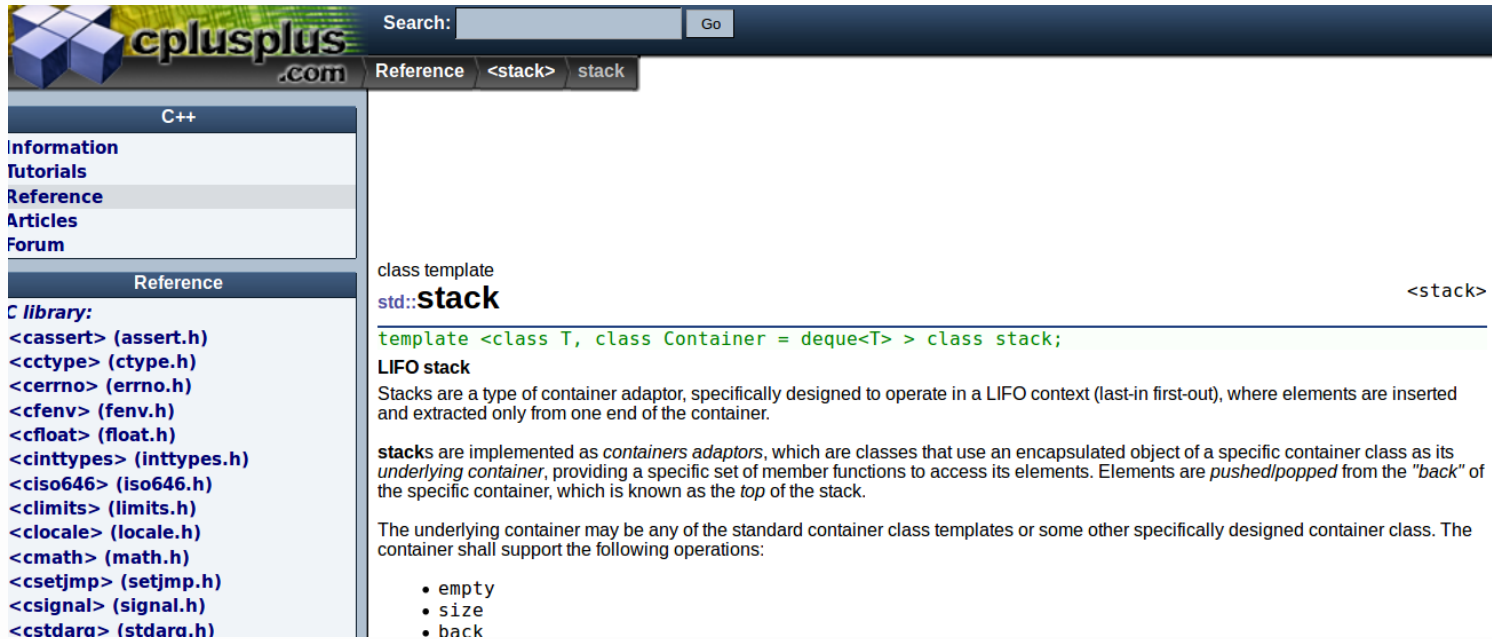
("T" refers to the data-type the stack stores)

- `void push(T& item)`
- `void pop()`
- `T& top()`
- `size_type size()`
- `bool empty()`

4. STL Stack

Access documentation about the STL Stack at **cplusplus.com**:

<http://www.cplusplus.com/reference/stack/stack/>



The screenshot shows the Cplusplus.com website interface. At the top, there is a search bar and a navigation menu with tabs for 'Reference', '<stack>', and 'stack'. The left sidebar contains a 'C++' section with links to 'Information', 'Tutorials', 'Reference', 'Articles', and 'Forum'. Below this is a 'Reference' section with a list of C library headers: <cassert> (assert.h), <cctype> (ctype.h), <cerrno> (errno.h), <cfenv> (fenv.h), <cfloat> (float.h), < cinttypes> (inttypes.h), <ciso646> (iso646.h), <climits> (limits.h), < clocale> (locale.h), < cmath> (math.h), < csetjmp> (setjmp.h), < csignal> (signal.h), and < cstdara> (stdara.h). The main content area displays the documentation for the 'std::stack' class template. It includes the header '<stack>', the template definition 'template <class T, class Container = deque<T> > class stack;', and a description of the LIFO stack. The text explains that stacks are container adaptors designed for LIFO operations, where elements are inserted and extracted from the 'back' of the container. It also mentions that stacks are implemented as 'containers adaptors' and that the underlying container can be any of the standard container class templates or a specifically designed one. A list of member functions is provided: empty, size, and back.

Search: Go

Reference <stack> stack

C++

Information
Tutorials
Reference
Articles
Forum

Reference

C library:
<cassert> (assert.h)
<cctype> (ctype.h)
<cerrno> (errno.h)
<cfenv> (fenv.h)
<cfloat> (float.h)
< cinttypes> (inttypes.h)
<ciso646> (iso646.h)
<climits> (limits.h)
< clocale> (locale.h)
< cmath> (math.h)
< csetjmp> (setjmp.h)
< csignal> (signal.h)
< cstdara> (stdara.h)

class template
std::stack <stack>

template <class T, class Container = deque<T> > class stack;

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

stacks are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed/popped* from the "back" of the specific container, which is known as the *top* of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

- empty
- size
- back

Notes

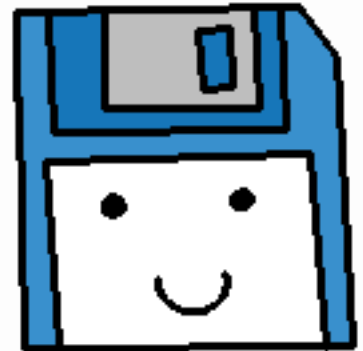
Functions:

(*"T" refers to the data-type the stack stores*)

- void push(T& item)
- void pop()
- T& top()
- size_type size()
- bool empty()

4. STL *Stack*

Example Time!



4. STL Stack

Using a STL Stack...

```
cin >> text;
stack<char> letters;

for ( unsigned int i = 0; i < text.size(); i++ )
{
    cout << "Letter " << i << " = " << text[i] << endl;
    letters.push( text[i] );
}

cout << endl << "Pop off stack: " << endl;
while ( !letters.empty() )
{
    cout << letters.top();
    letters.pop();
}
```

Declaring a stack of characters

Pushing characters onto the stack

Checking if the stack is empty

Accessing the top-most item

Removing the top-most item

5. STL Map

When we're using a plain-old array, we have a series of elements in order, starting at 0, going until (size – 1).



7 items in the array
=
Index 0 through 6 is valid

Notes

A map is also known as a dictionary or hash table.

A map contains key-value pairs.

5. STL Map

A value of 0, 1, 2, 3, 4, 5, or 6, which specifies an element's position, is known as an index, but we can also think of it like a “key”, which helps us locate the value we want



Notes

A map is also known as a dictionary or hash table.

A map contains key-value pairs.

5. STL Map

But with a data structure like a map, our keys don't have to just be integers, and they don't have to be array indices.

The **key** can be **any data type**, and it can point to a **value** of any data type as well.

Notes

A map is also known as a dictionary or hash table.

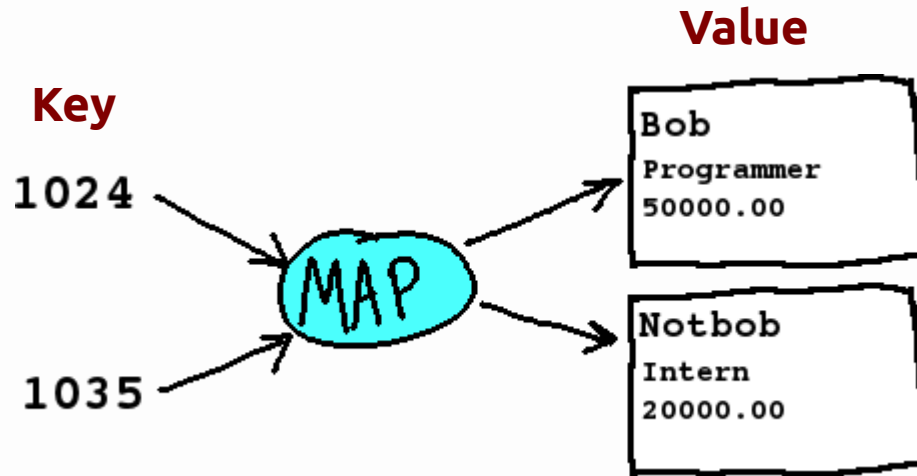
A map contains key-value pairs.

5. STL Map

For example, think of an employee ID, that points to an employee object in a program...

```
class Employee
{
    public:
        // ...

    private:
        string name;
        string jobTitle;
        float salary;
};
```



Notes

A map is also known as a dictionary or hash table.

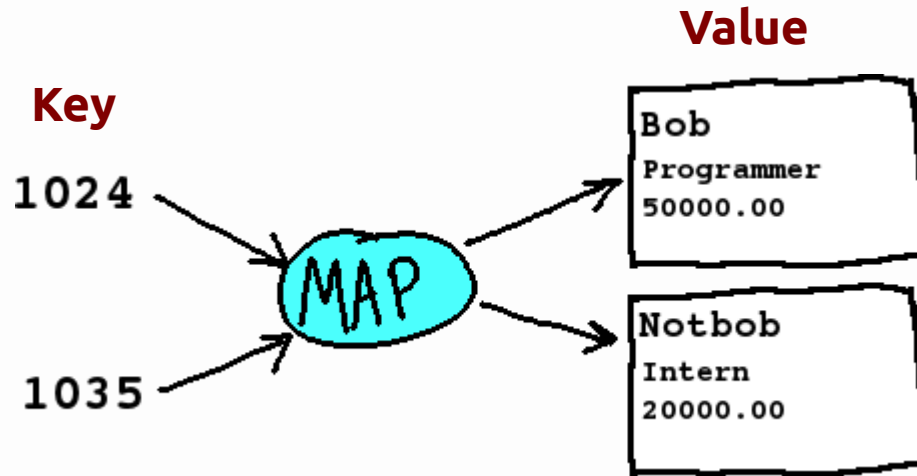
A map contains key-value pairs.

5. STL Map

Maps are built in a special way so that we can access elements by key, and it will access it quickly – search algorithm not required.

```
class Employee
{
    public:
    // ...

    private:
    string name;
    string jobTitle;
    float salary;
};
```



Notes

A map is also known as a dictionary or hash table.

A map contains key-value pairs.

5. STL Map

Some handy map functions are...

- **operator[]**
Access an element of the map, using a key
- **empty**
Returns whether the map is empty or not
- **size**
Get the amount of elements in the map

Notes

Functions:

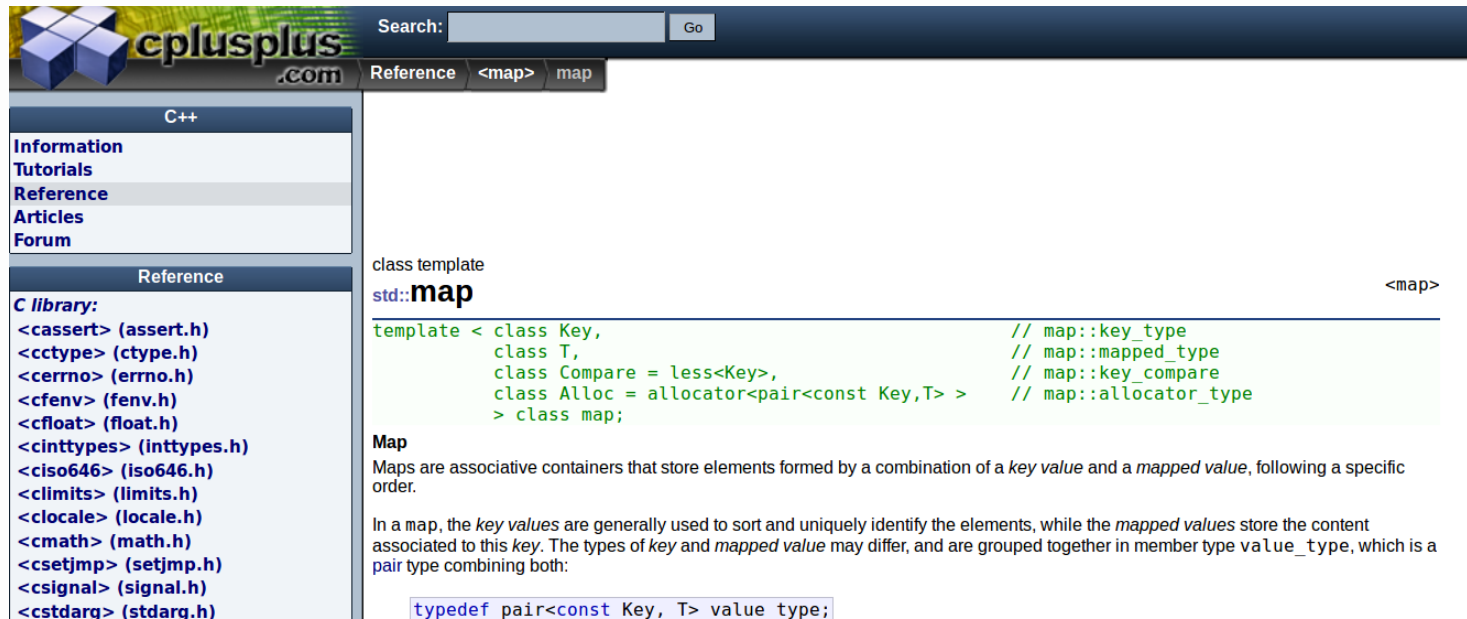
("T1" and "T2" refer to the key and value data-types)

- `T2& operator[]
 (const T1& key)`
- `size_type size()`
- `bool empty()`

5. STL Map

Access documentation about the STL Map at **cplusplus.com**:

<http://www.cplusplus.com/reference/map/map/>



The screenshot shows the Cplusplus.com website interface. At the top, there is a search bar and a 'Go' button. Below the search bar, there are tabs for 'Reference', '<map>', and 'map'. The left sidebar contains a navigation menu with links to 'C++', 'Information', 'Tutorials', 'Reference', 'Articles', and 'Forum'. Under the 'Reference' section, there is a list of C library headers: <cassert> (assert.h), <cctype> (ctype.h), <cerrno> (errno.h), <cfenv> (fenv.h), <cmath> (math.h), <cfloat> (float.h), <climits> (limits.h), <clocale> (locale.h), <cmath> (math.h), <csetjmp> (setjmp.h), <csignal> (signal.h), and <cstdint> (stdint.h). The main content area displays the 'std::map' class template. It includes the template definition: `template < class Key, class T, class Compare = less<Key>, class Alloc = allocator<pair<const Key,T> > > class map;` with comments for `map::key_type`, `map::mapped_type`, `map::key_compare`, and `map::allocator_type`. Below the code, there is a section titled 'Map' which describes it as an associative container. It states: 'Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order. In a map, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this key. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a *pair* type combining both:'. At the bottom, there is a typedef: `typedef pair<const Key, T> value_type;`

Notes

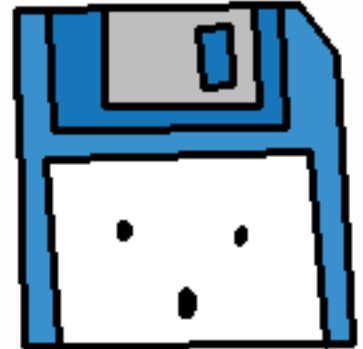
Functions:

("T1" and "T2" refer to the key and value data-types)

- T2& operator[] (const T1& key)
- size_type size()
- bool empty()

5. STL Map

Example Time!



5. STL Map

Using a STL Map...

```
map<string, int> employees;  
employees["Harry"] = 1024;  
employees["Ron"] = 256;  
employees["Hermione"] = 512;  
  
string key;  
cout << "Enter an employee name: ";  
cin >> key;  
  
cout << "That employee ID is "  
    << employees[ key ] << endl;
```

Declare a map, with string keys and int values

Adding key-value pairs

Accessing a value, given some key

Conclusion

We will eventually implement each of these structures ourselves to learn how these work, but sometimes it can be handy to view new concepts from the outside→in to begin with.

Data structures are all about writing structures (i.e., classes) to store data. Structures need a way to add data, access data, and remove data. We will be writing lots of add, access, and remove functions this semester!