Contents

1 Stacks

Definition. *Stacks*
A *stack* is an abstract data type where the last item that you add is the first one to be removed. You add and remove items from the top of the stack. To use a stack, you need to include it from the STL by writing `#include <stack>`.

1.1 Stack Operations

- Put something on top of the stack (`stack.push(item)`)
- Remove the top item (`stack.pop()`)
- Look at the top item, without removing it (`stack.top()`)
- Check to see if the stack is empty (`stack.empty()`)

2 Queues

Definition. *Queue*
A *queue* is an abstract data type where the first item that you add is the first one to be removed. Every queue has a *front* and a *rear*. You add items at the rear and remove them from the front. To use a queue, you need to include it from the STL by writing `#include <queue>`.

2.1 Queue Operations

- Insert an item at the rear of the queue (`queue.enqueue(item)`)
- Remove and return the top item from the front of the queue (`queue.dequeue()`)
- Determine if the queue is empty (`queue.empty()`)
- Determine the number of items in the queue (`queue.size()`)
- Get the value of the first item in the queue without removing it (`queue.getFront()`)

3 Inheritance

Main Idea. Inheritance is a way to form new classes using classes that have already been defined.

3.1 The Three uses of Inheritance

1. **Reuse**

   Reusing is when you write code *once* in a base class and reuse the same code in your derived classes.

2. **Extension**

   Extension is when you *add new behaviors or data* to a derived class that were not present in a base class.

3. **Specialisation/Overriding**

   Specialisation is when you *redefine an existing behaviour* (from the base class) with a new behaviour (in your derived class).

3.2 The "Virtual" Keyword

To override existing functions, we put the keyword "virtual" before the function *declarations* in both the original and replacement functions. You only want to use the "virtual" keyword for functions you intend to override in your subclasses. If a function is the same across all of the derived classes, then you don't want to make it a virtual function.

When you redefine a function in a derived class, the redefined version *hides* the base version of the function (when using your derived class). In other words, the derived class will default to always using the *most derived version (that it knows of)* a specialised method. You can still call a base class's version of a method that's been redefined by using `baseClass::method()`.

3.3 Inheritance and Construction

If you don't use an initializer list in your classes, C++ will add an implicit call to the member variable's constructor for you. In subclasses, C++ will always construct the base part first, and the derived part second.

3.4 Inheritance and Destruction

First, C++ runs the body of your outer object's destructor. Then, C++ destructs *all* member objects. In subclasses, C++ will destruct the derived part first, then the base part second.

Note (Summary). For construction, you can think of this as C++ needing the outer object to be made before it can make the derived object. For example, you can't make a `ShieldedRobot` without a `Robot`. For destruction, you can think of C++ needing to destroy the inner object first, lest it become ill-defined. For example, destroying the `Robot` class first could cause issues if `ShieldedRobot`'s destructor relies on some of `Robot`'s methods.

3.5 Inheritance and Initializer Lists

Rule. If a superclass requires parameters for construction, then you *must* add an initializer list to the subclass constructor! Furthermore, the *first item* in the initializer list must be the name of the base class, along with the parameters in parentheses.

3.6 Inheritance and Assignment Operators

Assigning one instance of a derived class to another is fine, *only if there are no dynamically allocated member variables.* If this is the case, then you *must* define assignment operators and copy constructors for the base and derived classes.

4 Polymorphism

Main Idea. The idea behind polymorphism is passing members of a derived class to a function that accepts parameters of the base class.

Definition. *Polymorphism*
The usage of a base pointer or base reference to access a derived object is called *polymorphism.*

This works because the derived class can do everything that the base class can, if not more. The function only sees the "base class parts" of the derived class, and has no idea that the parameter you passed in isn't actually an instance of the base class.

Definition. *Chopping*
If you don't pass the derived object by reference or by pointer, *chopping* occurs. Essentially, C++ will "chop off" all of the data/methods fo the derived class and *only* send the base parts of the variables to the function. The chopped variable will have no derived class parts.

For polymorphism to work, we use virtual functions. When you call a virtual function, C++ will automatically figure out the correct version of the function to use.

Note. When you omit the *virtual* keyword, C++ can't figure out the right version of the function to call, so it will default to calling the function defined in the base class.

4.1 When Should You Use the Virtual Keyword?

1. Use the virtual keyword in your base class *any time* you expect to redefine a function in a derived class.
2. Use the virtual keyword in your base class *any time* you redefine a function (for clarity, not required).
3. Always use the virtual keyword for the destructor in your base class (and in the derived classes, for clarity).
4. You *cannot* have a virtual constructor.

4.2 Polymorphism and Pointers

In general, you may point a superclass pointer at a subclassed variable. For example, you may point a `Person` pointer at a `Politician` variable. The reverse of this is *not* allowed.

Remember that C++ always calls the *most-derived* version of a function associated with a variable, as long as it's marked virtual!

4.3 Polymorphism and Virtual Destructors

You should *always* make sure that you use virtual destructors when you use inheritance/polymorphism. If you don't do this, the destructor for the base class will be called first, and the derived class's destructor will never be called (because C++ doesn't know it exists).

4.4 Pure Virtual Functions

You define *pure virtual functions* when you need a function in the base class for polymorphism, but never call it there. Your derived classes will need to redefine all pure virtual functions to actually do something. If a class has a pure virtual function, you can't even define a regular variable with this class.

4.5 Abstract Base Classes

Definition. *Abstract Base Class*
If you define *at least one* pure virtual function in a base class, then the class is called an *abstract base class.*

The derived classes of an abstract base class must provide code for *all* pure virtual functions, or the derived class will also become an abstract base class. Although you can't create a variable with an abstract base class type, you can still use them to implement polymorphism. This forces the programmer to implement certain functions and avoid bugs.

5 Recursion

Main Idea. Solve problems by making a function call itself to solve a smaller sub-problem.

5.1 The Two Rules of Recursion

1. Every recursive function must have a "stopping condition" or "base case". Your recursive function must be able to solve the simplest problem *without recursion.*

2. Every recursive function must have a "simplifying step". The function must pass a smaller sub-problem to itself to ensure that the algorithm will eventually reach the base case. You cannot pass the original problem back into the recursive function because it will just run forever.

Note. Recursive functions should *never* use global, static, or member variables. They should only ever use local variables and parameters.

Recursive functions should generally only access the current node/array cell passed into it.

6 Generic Programming

Main Idea. We want to write functions or classes in a manner so that it can process many different types of data.

6.1 Allowing Generic Comparisons

Comparison operators are basically the same thing as the assignment operator, except they compare things. You may either define them inside or outside a class. If the former, it should take a constant reference to another object of the class as a parameter. If the latter, it should take two constant references to objects of the class as parameters. In either case, the return type is a `bool`.

6.2 Templates

To turn any function into a "generic function", do this:

1. Add `template <typename ItemType>` above your function

2. Everywhere you use the data type in the function, replace it with `ItemType`.

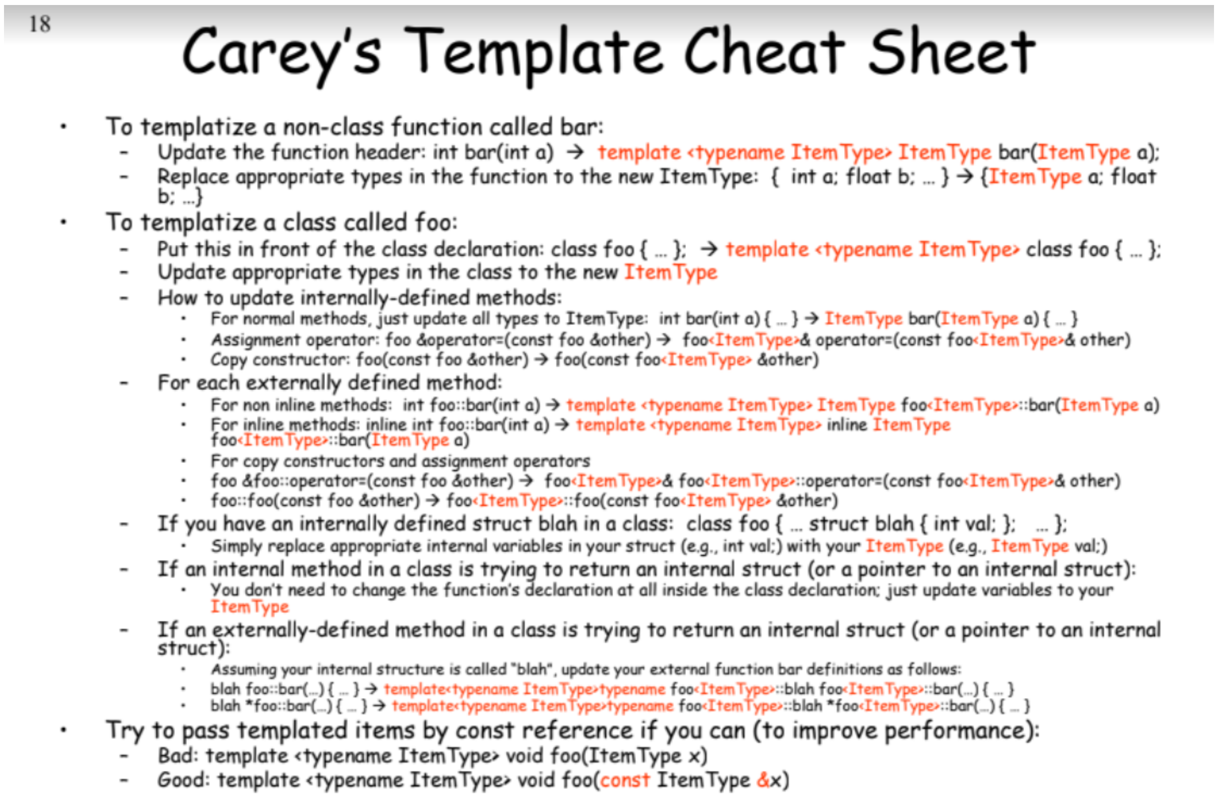
Note. Always put your templated functions in a header file. You put the *entire* template function body in the header file, not just the prototype.

6.3 Function Template Details

You *must* use the template data type to define the type of at least one formal parameter, otherwise you'll get an error. If a function has two or more templated parameters with the same type, you must pass in the same type of variable for both. For example, if we have a function `Data max(Data x, Data y)`, then passing in an integer and a float for `x` and `y` would cause an error.

You can template a function with multiple types by using `template <typename Type1, typename type2>` before the function header.

6.4 Templating Cheat Sheet



18

Carey's Template Cheat Sheet

- To templatize a non-class function called bar:
  - Update the function header: int bar(int a) → template <typename ItemType> ItemType bar(ItemType a);
  - Replace appropriate types in the function to the new ItemType: { int a; float b; ... } → {ItemType a; float b; ...}
- To templatize a class called foo:
  - Put this in front of the class declaration: class foo { ... }; → template <typename ItemType> class foo { ... };
  - Update appropriate types in the class to the new ItemType
  - How to update internally-defined methods:
    - For normal methods, just update all types to ItemType: int bar(int a) { ... } → ItemType bar(ItemType a) { ... }
    - Assignment operator: foo &operator=(const foo &other) → foo<ItemType>& operator=(const foo<ItemType>& other)
    - Copy constructor: foo(const foo &other) → foo(const foo<ItemType> &other)
  - For each externally defined method:
    - For non inline methods: int foo::bar(int a) → template <typename ItemType> ItemType foo<ItemType>::bar(ItemType a)
    - For inline methods: inline int foo::bar(int a) → template <typename ItemType> inline ItemType foo<ItemType>::bar(ItemType a)
    - For copy constructors and assignment operators
    - foo &foo::operator=(const foo &other) → foo<ItemType>& foo<ItemType>::operator=(const foo<ItemType>& other)
    - foo::foo(const foo &other) → foo<ItemType>::foo(const foo<ItemType> &other)
  - If you have an internally defined struct blah in a class: class foo { ... struct blah { int val; }; ... };
    - Simply replace appropriate internal variables in your struct (e.g., int val;) with your ItemType (e.g., ItemType val;)
  - If an internal method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - You don't need to change the function's declaration at all inside the class declaration; just update variables to your ItemType
  - If an externally-defined method in a class is trying to return an internal struct (or a pointer to an internal struct):
    - Assuming your internal structure is called "blah", update your external function bar definitions as follows:
    - blah foo::bar(...) { ... } → template<typename ItemType>typename foo<ItemType>::blah foo<ItemType>::bar(...) { ... }
    - blah *foo::bar(...) { ... } → template<typename ItemType>typename foo<ItemType>::blah *foo<ItemType>::bar(...) { ... }
- Try to pass templated items by const reference if you can (to improve performance):
  - Bad: template <typename ItemType> void foo(ItemType x)
  - Good: template <typename ItemType> void foo(const ItemType &x)

7 The Standard Template Library (STL)

7.1 Vector

A vector is basically the same as an array but you can change the size. You need to `#include <vector>` at the top of your file to use it.

Functionality of the Vector

- To add a new item to the end of the vector, use `vec.push_back(item)`
- You can use brackets to read or change an item, just like an array (only existing items)
- You can use `vec.front()` or `vec.back()` to read/write the first/last elements (if it's not empty)
- To remove an item from the back of a vector, use `vec.pop_back()`
- To get the number of elements in the vector, use `vec.size()`
- To determine if the vector is empty, use `vec.empty()`

7.2 List

The list class works just like a linked list. You need to `#include <list>` at the top of your file to use it. Like vector, the list class has `push_back`, `pop_back`, `front`, `back`, `size`, and `empty` methods. Furthermore, it has `push_front` and `pop_front` to add/remove items from the front of the list. Unlike vectors, you can't access list elements using brackets.

7.3 Iterators

To access elements, we need to use an iterator. The iterator syntax is

```
Container<ItemType>::iterator it;
```

We use the `begin()` and `end()` methods to point at items in our STL classes.

Note. The `end()` method points at the item *after* the last item in the container.

You can increment/decrement the iterator by using `it++` and `it--`, respectively. To iterate through a constant container, just use a constant iterator:

```
Container<ItemType>::const_iterator it;
```

7.4 Map

Maps (read: functions) allow us to associate two related values. You need to `#include <map>` at the top of your file to use it. Because there are two types involved, the syntax is `map<type1, type2> mapName;`.

If you want to search in both directions, you need to use two maps. The map basically creates a struct that holds both of your variables in the `first` and `second` member variables. To search for an association, we can use the `find()` function. Maps are automatically ordered for you.

7.5 Set

A set is a container that keeps track of *unique* items. You need to `#include <set>` at the top of your file to use it. You can `insert()` items into the set, and it will automatically ignore duplicate items. To erase an item of the set, use the `erase()` function. To get the number of elements in a set, use `size()`.

7.6 Iterator Gotchas!

If you add an item anywhere in a vector, you must assume that any iterators you had are invalidated. If you erase the item that the iterator is pointing to or any item that comes before it, the iterator is also invalidated.

For sets, lists, and maps, this doesn't happen—unless you delete the item that the iterator points to.

7.7 STL Algorithms

You will need to `#include <algorithm>` to use these. This provides many useful functions like `find()`, `set_intersection`, and `sort()`.

To use the `sort()` function, you pass in two iterators—one that points to the first item and one that points just past the last item you want to sort. This also works with arrays.

To use the `find()` function, you pass in two iterators and one item of the type you want to find—the iterators point to the first and just past the last item, and the last parameter is what you want to find. The function returns an iterator that points to the item it found (it will return the second iterator if the item is not found).

To use the `find_if()` function, you pass in two iterators and a predicate (that returns a boolean). The iterators point to the first and just past the last item, and the predicate is used to evaluate whether the current item satisfies the predicate. The function returns an iterator pointing at the first item that satisfies the predicate function. Thus we can use the `find_if()` function to locate items that meet specific requirements.