CS 32 Week 6 Discussion 1I

Srinath

Outline

- Templates
- Standard Template Library (STL)
- Worksheet 6

Templates

Templates : Definition

An elegant way of handling generic types. Helps to adapt the same code pattern to more than one type

Generally 2 types

- Function Templates
- Class Templates

```
template <typename T>
T minimum( T a,  T b){
	if (a<b)
		return a;
	else
		return b;
}
```

Templates : Definition

An elegant way of handling generic types. Helps to adapt the same code pattern to more than one type

Generally 2 types

- Function Templates
- Class Templates

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int p, q;
p=10; q=15;
int r = minimum(p, q);

string x, y;
x= “Hi”; y = “Hello”;
string z = minimum(x, y);
```

The compiler looks at the matching pattern and writes appropriate code.

Templates : Definition

An elegant way of handling generic types. Helps to adapt the same code pattern to more than one type

Generally 2 types

- Function Templates
- Class Templates

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int p, q;
p=10; q=15;
int r = minimum(p, q);

string x, y;
x= “Hi”; y = “Hello”;
string z = minimum(x, y);
```

The compiler looks at the matching pattern and writes appropriate code.

Compiler generated code

```
int minimum( int a,  int b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
string minimum( string a,  string b){
        if (a<b)
                return a;
        else
                return b;
}
```

Templates : Definition

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
    m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
   If (m_first > m_second)
        return m_first;
  else
        return m_second;
}
```

Templates : Definition

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
   m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
   If (m_first > m_second)
        return m_first;
  else
        return m_second;
}
```

```
pair <int> coupleA(13, 17);
cout << couple1.getMax()<<endl;

pair <string> coupleB(“Hello”, “World);
```

Templates : Definition

Compiler generated code

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
    m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
   If (m_first > m_second)
        return m_first;
   else
        return m_second;
}
```

```
pair <int> coupleA(13, 17);
cout << couple1.getMax()<<endl;

pair <string> coupleB(“Hello”, “World);
```

```
…
    int m_first;
    int m_second;
…

pair<int>::pair(int first, int second){
   …
}

Int pair<int>::getMax(){
   …….
}

…
    string m_first;
    string m_second;
…

pair<string>::pair(string first, string second){
   …
}

string pair<string>::getMax(){
   …….
}
```

Templates : Definition

Compiler generated code

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
    m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
   If (m_first > m_second)
        return m_first;
   else
        return m_second;
}
```

```
pair <int> coupleA(13, 17);
cout << couple1.getMax()<<endl;

pair <string> coupleB(“Hello”, “World);
```

```
…
    int m_first;
    int m_second;
…

pair<int>::pair(int first, int second){
   …
}

Int pair<int>::getMax(){
   …….
}
```

```
…
    string m_first;
    string m_second;
…

pair<string>::pair(string first, string second){
   …
}

string pair<string>::getMax(){
   …….
}
```

No, getMax() for string won’t be generated as it is not called anywhere.

Compiler generated code

Templates : Definition

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
    m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
    If (m_first > m_second)
        return m_first;
    else
        return m_second;
}
```

```
pair <int> coupleA(13, 17);
cout << couple1.getMax()<<endl;

pair <string> coupleB(“Hello”, “World);
```

What else is generated?

```
…
    int m_first;
    int m_second;
…

pair<int>::pair(int first, int second){
    …
}

Int pair<int>::getMax(){
    …….
}

…
    string m_first;
    string m_second;
…

pair<string>::pair(string first, string second){
    …
}

string pair<string>::getMax(){
    …….
}
```

No, getMax() for string won’t be generated as it is not called anywhere.

Templates : Definition

Compiler generated code

```
template <typename T>
class pair {
        T m_first;
        T m_second;
    public:
        pair(T first, T second);
        T getMax();
};

template <typename T>
pair<T>::pair(T first, T second){
    m_first=first; m_second=second;
}

template <typename T>
T pair<T>::getMax(){
   If (m_first > m_second)
        return m_first;
   else
        return m_second;
}
```

```
pair <int> coupleA(13, 17);
cout << couple1.getMax()<<endl;

pair <string> coupleB(“Hello”, “World);
```

What else is generated?

- The destructors

```
…
    int m_first;
    int m_second;
…

pair<int>::pair(int first, int second){
   …
}

Int pair<int>::getMax(){
   …….
}
```

```
…
    string m_first;
    string m_second;
…

pair<string>::pair(string first, string second){
   …
}

string pair<string>::getMax(){
   …….
}
```

No, getMax() for string won't be generated as it is not called anywhere.

Templates : A successful call

Conditions for a successful template call

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

Will it compile ?

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int r = minimum(18, 12.5);
```

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

Will it compile ?

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int r = minimum(18, 12.5);
```

No, fails first condition

```
Ship s1;
Ship s2;
Ship s = minimum(s1, s2);
```

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

Will it compile ?

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int r = minimum(18, 12.5);
```

No, fails first condition

```
Ship s1;
Ship s2;
Ship s = minimum(s1, s2);
```

No, fails second condition

Can we fix it ?

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

Will it compile ?

```
template <typename T>
T minimum( T a,  T b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
int r = minimum(18, 12.5);
```

No, fails first condition

```
Ship s1;
Ship s2;
Ship s = minimum(s1, s2);
```

No, fails second condition

Can we fix it ?
- Yes, operator overloading

```
bool operator<(const Ship& s1, const Ship&s2){
        return s1.height < s2.height;
}
```

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

Will it compile ?

```
double r = minimum(18, 12.5);
```

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

Will it compile ?

`double r = minimum(18, 12.5);` Yes

Will it do what we want ?

Templates : A successful call

Conditions for a successful template call

- It has to match the specified pattern
- The generated code from pattern has to compile
- It has to do the right thing we wanted

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

Will it compile ?

`double r = minimum(18, 12.5);` Yes

Will it do what we want ? No, it returns 12 instead of 12.5

Templates : Allowed things

T ←→ T

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
double r = minimum(13, 12);
```

Templates : Allowed things

T ⟵⟶ T

T ⟵⟶ T&

```
template <typename T1, typename T2>
T1 minimum( T1& a,  T2& b){
	if (a<b)
		return a;
	else
		return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
	if (a<b)
		return a;
	else
		return b;
}
```

```
double r = minimum(13, 12);
```

Templates : Allowed things

T ⟵⟶ T

T ⟵⟶ T&

T ⟵⟶ const T&

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( T1& a,  T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( const T1& a,  const T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
double r = minimum(13, 12);
```

Templates : Allowed things

T ←→ T

T ←→ T&

T ←→ const T&

```
template <typename T1, typename T2>
T1 minimum( T1& a,  T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( const T1& a,  const T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
double r = minimum(13, 12);
```

Which way is better?

Templates : Allowed things

T ←→ T

T ←→ T&

T ←→ const T&

```
template <typename T1, typename T2>
T1 minimum( T1 a,  T2 b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
double r = minimum(13, 12);
```

```
template <typename T1, typename T2>
T1 minimum( T1& a,  T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

```
template <typename T1, typename T2>
T1 minimum( const T1& a,  const T2& b){
        if (a<b)
                return a;
        else
                return b;
}
```

Which way is better?

- const T &, as copying might be expensive sometimes
- and we are guaranteed our passed element is not modified

Templates :

```
template <typename T1, typename T2>
class pair {
        T1 m_first;
        T2 m_second;
    public:
        pair(){
            m_first = "";
            m_second = "";
        }
        pair(T1 first, T2 second);
};

template <typename T1, typename T2>
pair<T1, T2>::pair(T1 first, T2 second){
    m_first=first; m_second=second;
}
```

Will it compile ?

```
pair<string, string> p1;
```

Templates :

```
template <typename T1, typename T2>
class pair {
          T1 m_first;
          T2 m_second;
     public:
          pair(){
              m_first = "";
              m_second = "";
          }
          pair(T1 first, T2 second);
};

template <typename T1, typename T2>
pair<T1, T2>::pair(T1 first, T2 second){
    m_first=first; m_second=second;
}
```

Will it compile ?

`pair<string, string> p1;` Yes

`pair<string, double> p1;`

Templates :

```
template <typename T1, typename T2>
class pair {
          T1 m_first;
          T2 m_second;
     public:
          pair(){
              m_first = "";
              m_second = "";
          }
          pair(T1 first, T2 second);
};

template <typename T1, typename T2>
pair<T1, T2>::pair(T1 first, T2 second){
    m_first=first; m_second=second;
}
```

Will it compile ?

pair<string, string> p1; Yes

pair<string, double> p1; No, double can’t be assigned “”.

Can we fix it ?

Templates : Default Values

```
template <typename T1, typename T2>
class pair {
        T1 m_first;
        T2 m_second;
    public:
        pair(){
            m_first = “”;
            m_second = “”;
        }
        pair(T1 first, T2 second);
};

template <typename T1, typename T2>
pair<T1, T2>::pair(T1 first, T2 second){
    m_first=first; m_second=second;
}
```

Will it compile ?

`pair<string, string> p1;` Yes

`pair<string, double> p1;` No, double can’t be assigned “”.

Can we fix it ?
- Yes

```
……
m_first = T1();
m_second = T2();
…..
```

STL

STL : Standard Template Library

Most of the **Data Structures** and **Algorithms** handling various **Types** are already written for you, so don't reinvent the wheel :)

However, Programmers should have fair idea of what's happening under the hood for debugging, efficiency etc.

Reference :- https://www.cplusplus.com/reference/stl

Or just google

vector c++ STL …
queue c++ STL …
list C++ STL …

That should land you to good c++ site (https://www.cplusplus.com/reference/vector/vector/)

STL : Containers

Containers are which store elements of a certain type.

Examples : vector, list, queue, stack etc..

We have various functions at our disposal on top of them for efficient and effective usage

Defining : vector<int> v, list<string> l, queue<Ship> q; etc..

Make sure to include appropriate headers '**#include <vector>**', '**#include <list>**'

STL : Containers

Containers are which store elements of a certain type.

Examples : vector, list, queue, stack etc..

We have various functions at our disposal on top of them for efficient and effective usage

Defining : vector<int> v, list<string> l, queue<Ship> q; etc..

Make sure to include appropriate headers '**#include <vector>**', '**#include <list>**'

Iterator is kind of pointer to an element in the container, is helpful to access, modify, iterate .. the container

Defining : vector<int>::iterator it1, list<string>::iterator it2, queue<Ship>::iterator it3 …

STL : Containers

Containers are which store elements of a certain type.

Examples : vector, list, queue, stack etc..

We have various functions at our disposal on top of them for efficient and effective usage

Defining : vector<int> v, list<string> l, queue<Ship> q; etc..

Make sure to include appropriate headers ‘**#include <vector>**’, ‘**#include <list>**’

Iterator is kind of pointer to an element in the container, is helpful to access, modify, iterate .. the container

Defining : vector<int>::iterator it1, list<string>::iterator it2, queue<Ship>::iterator it3 …

v.begin()

STL : Containers

Containers are which store elements of a certain type.

Examples : vector, list, queue, stack etc..

We have various functions at our disposal on top of them for efficient and effective usage

Defining : vector<int> v, list<string> l, queue<Ship> q; etc..

Make sure to include appropriate headers '**#include <vector>**', '**#include <list>**'

Iterator is kind of pointer to an element in the container, is helpful to access, modify, iterate .. the container

Defining : vector<int>::iterator it1, list<string>::iterator it2, queue<Ship>::iterator it3 …

v.begin() - iterator to first element of the container
v.end()

STL : Containers

Containers are which store elements of a certain type.

Examples : vector, list, queue, stack etc..

We have various functions at our disposal on top of them for efficient and effective usage

Defining : vector<int> v, list<string> l, queue<Ship> q; etc..

Make sure to include appropriate headers '**#include <vector>**', '**#include <list>**'

Iterator is kind of pointer to an element in the container, is helpful to access, modify, iterate .. the container

Defining : vector<int>::iterator it1, list<string>::iterator it2, queue<Ship>::iterator it3 …

v.begin() - iterator to first element of the container
v.end() - iterator just passing the last element of the container (NOT the last element)

STL : Vector

Consider it as a **Dynamic Array,** store as many elements as you want, remove, modify, access … Don't worry about new memory allocation, de-allocation etc, all that is handled for you.

Elements are **stored** in **contiguous memory locations**, so **access** is **NOT expensive**.

https://www.cplusplus.com/reference/vector/vector/

STL : Vector

Consider it as a **Dynamic Array,** store as many elements as you want, remove, modify, access … Don't worry about new memory allocation, de-allocation etc, all that is handled for you.

Elements are **stored** in **contiguous memory locations**, so **access** is **NOT expensive**.

https://www.cplusplus.com/reference/vector/vector/

A little usage...

vector<int> V;

V[i] - access

V.at(i) - access

V.front() - access front(0 th) element

V.back() - access last element

Both the above have undefined behaviour when V is empty.

V.push_back(90) - insert at last

V.empty() - check if empty

V.size() - get size

V.erase(iterator it) - remove element pointed by iterator it

V.insert(iterator it, int element)

STL : List

It’s a dynamic storage, but elements are not necessarily stored in contiguous memory. As it’s a linked list, adding/removing elements becomes efficient.

Elements are **NOT** stored in **contiguous memory locations**, so **access** might be **expensive**.

https://www.cplusplus.com/reference/list/list/

STL : List

It’s a dynamic storage, but elements are not necessarily stored in contiguous memory. As it’s a linked list, adding/removing elements becomes efficient.

Elements are **NOT** stored in **contiguous memory locations**, so **access** might be **expensive**.

https://www.cplusplus.com/reference/list/list/

A little usage...

list<int> L;

L.front() - access front(0 th) element

L.back() - access last element

L.push_back(90) - insert at last

L.push_front(80) - insert at first

L.pop_back() - remove last

L.pop_front() - remove last

L.empty() - check if empty

L.size() - get size

L.erase(iterator it) - remove element pointed by iterator it

L.insert(iterator it, int element)

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

Traversing….

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

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

Traversing….

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

```
vector<int> v;
Int i=0;
for(i=0; i<v.size(); i++){
        cout << v[i] <<endl;
}
```

Can always do this for a vector, not for other containers.
Use iterators for other containers.

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

vector iterator it, can we do *(it+2) or it=it+2?

Traversing….

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

```
vector<int> v;
Int i=0;
for(i=0; i<v.size(); i++){
        cout << v[i] <<endl;
}
```

Can always do this for a vector, not for other containers.
Use iterators for other containers.

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

vector iterator it, can we do *(it+2) or it=it+2?

- Yes, contiguous memory

list iterator it, can we do *(it+2) or it=it+2?

Traversing….

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

```
vector<int> v;
Int i=0;
for(i=0; i<v.size(); i++){
        cout << v[i] <<endl;
}
```

Can always do this for a vector, not for other containers.
Use iterators for other containers.

STL : Iterator

Can be used to traverse the container
vector<int>::iterator it; for vector — **list<int>::iterator it;** for list
Next element : **it++;**
Prev element : **it--;**
Access element : ***it;**

vector iterator it, can we do *(it+2) or it=it+2?

- Yes, contiguous memory

list iterator it, can we do *(it+2) or it=it+2?

- No, NOT a contiguous memory
So use it++ two times

Traversing….

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

```
vector<int> v;
Int i=0;
for(i=0; i<v.size(); i++){
        cout << v[i] <<endl;
}
```

Can always do this for a vector, not for other containers.
Use iterators for other containers.

STL : Algorithm

Have many algorithms like **find, sort ..** for containers

need to add ‘**#include <algorithm>**’

https://www.cplusplus.com/reference/algorithm/

STL : Find

Find a particular element in a container, gives back its iterator

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 100)
```

STL : Find

Find a particular element in a container, gives back its iterator

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 100)

if (it != L.end())  // as find returns end() iterator if not found
        cout << “found” <<endl;
else
        cout << “not found” <<endl;
```

STL : Find

Find a particular element in a container, gives back its iterator

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 100)

if (it != L.end())  // as find returns end() iterator if not found
        cout << “found” <<endl;
else
        cout << “not found” <<endl;
```

```
class Chicken{
        string color;
        …
}
list<Chicken> L;
```

How do we find a “red” chicken ?

STL : Find

Find a particular element in a container, gives back its iterator

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 100)

if (it != L.end())  // as find returns end() iterator if not found
        cout << "found" <<endl;
else
        cout << "not found" <<endl;

class Chicken{
        string color;
        …
}
list<Chicken> L;
```

How do we find a "red" chicken ?

```
list<int>::iterator it = find(L.begin(), L.end(), "red") -
```

STL : Find

Find a particular element in a container, gives back its iterator

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 100)

if (it != L.end())  // as find returns end() iterator if not found
        cout << “found” <<endl;
else
        cout << “not found” <<endl;

class Chicken{
        string color;
        …
}
list<Chicken> L;
```

How do we find a “red” chicken ?

`list<int>::iterator it = find(L.begin(), L.end(), “red”)` - Won’t work, instead **pass a function** which makes this check

STL : Find using Predicate

You can **pass functions** to some of the STL methods to achieve certain functionality by using STL.

```
class Chicken{
        string color;
        …
}

bool isRedChicken(const Chicken& c){
        return c.color==”red”;
}

list<Chicken> L;
list<int>::iterator it = find(L.begin(), L.end(), isRedChicken)
```

- By default, this calls the passed function by passing the element of the container and expects a bool

STL : Find using Predicate

You can **pass functions** to some of the STL methods to achieve certain functionality by using STL.

```
class Chicken{
        string color;
        …
}

bool isRedChicken(const Chicken& c){
        return c.color==”red”;
}

list<Chicken> L;
list<int>::iterator it = find(L.begin(), L.end(), isRedChicken)
```

- By default, this calls the passed function by passing the element of the container and expects a bool

What if we want to insert a Chicken into list only if same color chicken doesn’t exist.
i,e checking condition changes dynamically? How to find a chicken of any given color?

STL : Finding an Object

Create an object and try to find that object in the container.

```
class Chicken{
	string type;
	string color;
	…
}

list<Chicken> L;
bool insert(string mytype, string mycolor){
	???
	…
}
```

STL : Finding an Object

Create an object and try to find that object in the container.

```
class Chicken{
	string type;
	string color;
	…
}

list<Chicken> L;
bool insert(string mytype, string mycolor){
	Chicken c1(mytype, mycolor);
	list<int>::iterator it = find(L.begin(), L.end(), c1)
	…
}
```

STL : Finding an Object

Create an object and try to find that object in the container.

```
class Chicken{
        string type;
        string color;
        …
}

list<Chicken> L;
bool insert(string mytype, string mycolor){
        Chicken c1(mytype, mycolor);
        list<int>::iterator it = find(L.begin(), L.end(), c1)
        …
}
```

Are we done, will it work?

STL : Finding an Object

Create an object and try to find that object in the container.

```
class Chicken{
        string type;
        string color;
        …
}

list<Chicken> L;
bool insert(string mytype, string mycolor){
        Chicken c1(mytype, mycolor);
        list<int>::iterator it = find(L.begin(), L.end(), c1)
        …
}
```

Are we done, will it work?

- No, As we haven’t specified what it means to be equality of two Chicken’s

STL : Finding an Object

Create an object and try to find that object in the container.

```
class Chicken{
        string type;
        string color;
        …
}

list<Chicken> L;
bool insert(string mytype, string mycolor){
        Chicken c1(mytype, mycolor);
        list<int>::iterator it = find(L.begin(), L.end(), c1)
        …
}
```

Are we done, will it work?

- No, As we haven't specified what it means to be equality of two Chicken's

Operator overloading again!!

```
bool operator==(const Chicken& a1, const
Chicken& a2){
        return a1.color == a2.color;
}
```

STL : Sort

Sort elements in a container, or part of it

void **sort(iterator begin, iterator end)**

```
list<string> s;
sort(s.begin(), s.end());

vector<int> v;
sort(v+2, v+12)
```

STL : Sort

Sort elements in a container, or part of it

void **sort(iterator begin, iterator end)**

```
list<string> s;
sort(s.begin(), s.end());

vector<int> v;
sort(v+2, v+12)

vector<Chicken> CV;
sort(CV.begin(), CV.end())
```

Will it work?

STL : Sort

Sort elements in a container, or part of it

void **sort(iterator begin, iterator end)**

```
list<string> s;
sort(s.begin(), s.end());

vector<int> v;
sort(v+2, v+12)

vector<Chicken> CV;
sort(CV.begin(), CV.end())
```

Will it work?

- No, We don’t know how to compare two Chickens

STL : Sort using Predicate

```
void sort(iterator begin, iterator end)
```

```
bool compareChicken(const Chicken& c1, const Chicken& c2){
        return c1.weight < c2.weight;
}

vector<Chicken> CV;
sort(CV.begin(), CV.end(), compareChicken)
```

STL : Sort using Predicate

void **sort(iterator begin, iterator end)**

```
bool compareChicken(const Chicken& c1, const Chicken& c2){
        return c1.weight < c2.weight;
}

vector<Chicken> CV;
sort(CV.begin(), CV.end(), compareChicken)
```

Will operator overloading work here?

STL : Sort using Predicate

void **sort(iterator begin, iterator end)**

```
bool compareChicken(const Chicken& c1, const Chicken& c2){
        return c1.weight < c2.weight;
}

vector<Chicken> CV;
sort(CV.begin(), CV.end(), compareChicken)
```

Will operator overloading work here?
- Yes, overload the ‘<’ operator