CS 32 Week 6 Discussion 2C

UCLA CS

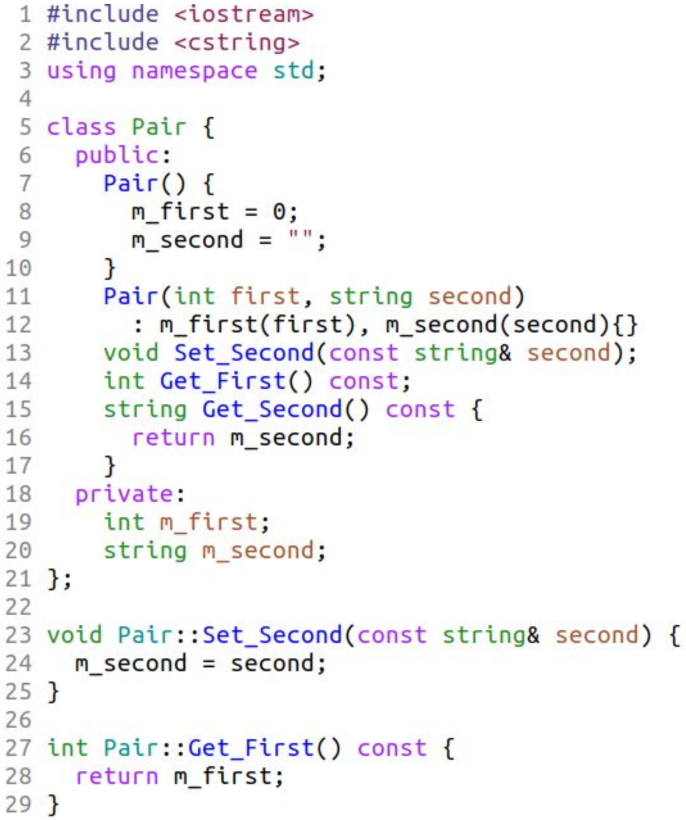
Yiyou Chen / Katie Chang

Topics

- C++ Template.
- C++ Standard Template Library (STL):
  1. Containers: queue, stack, vector, list
  2. iterator
- C++ standard library algorithm:
  1. find and find_if
  2. sort

These make C++ programmers look “lazier” than C programmers!

Template

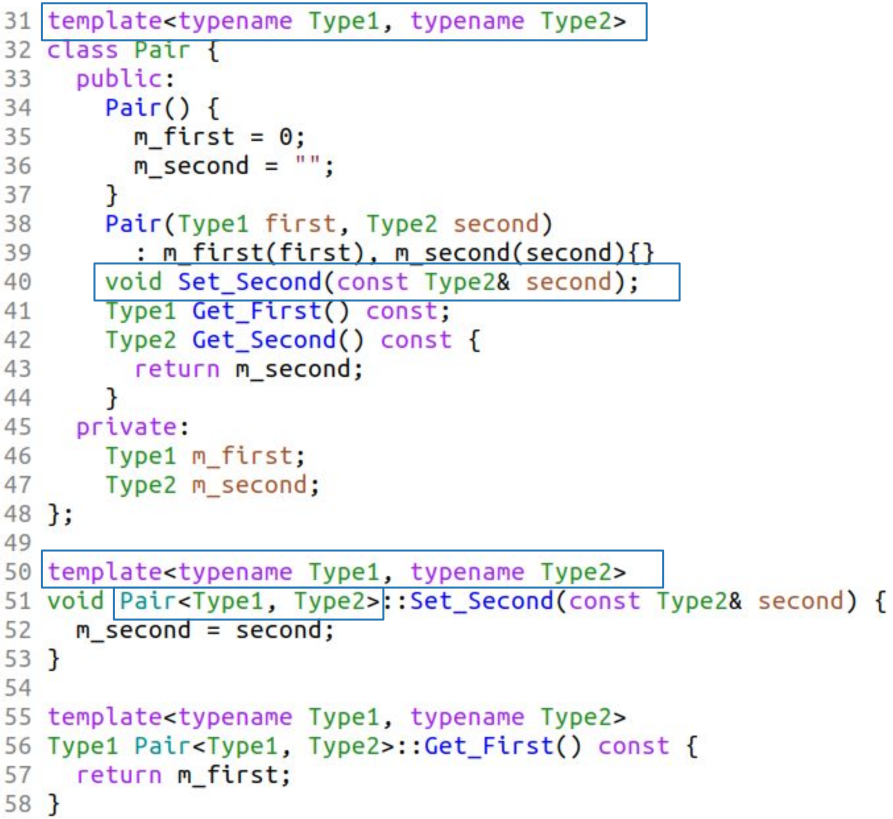


```
#include <iostream>
#include <cstring>
using namespace std;

class Pair {
  public:
    Pair() {
      m_first = 0;
      m_second = "";
    }
    Pair(int first, string second)
      : m_first(first), m_second(second){}
    void Set_Second(const string& second);
    int Get_First() const;
    string Get_Second() const {
      return m_second;
    }
  private:
    int m_first;
    string m_second;
};

void Pair::Set_Second(const string& second) {
  m_second = second;
}

int Pair::Get_First() const {
  return m_first;
}
```

This code compiles and the objects are pairs of the form (int, string).

What if we want to modify it to pairs of general forms (FirstType, SecondType)?

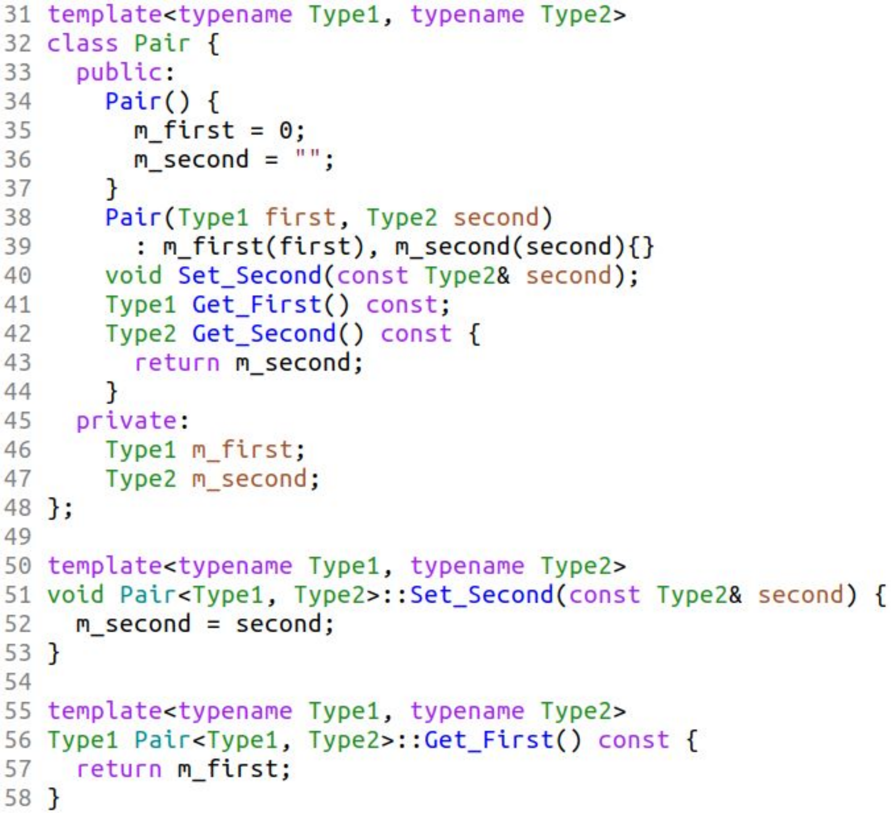
Template



```
template<typename Type1, typename Type2>
class Pair {
  public:
    Pair() {
      m_first = 0;
      m_second = "";
    }
    Pair(Type1 first, Type2 second)
      : m_first(first), m_second(second){}
    void Set_Second(const Type2& second);
    Type1 Get_First() const;
    Type2 Get_Second() const {
      return m_second;
    }
  private:
    Type1 m_first;
    Type2 m_second;
};

template<typename Type1, typename Type2>
void Pair<Type1, Type2>::Set_Second(const Type2& second) {
  m_second = second;
}

template<typename Type1, typename Type2>
Type1 Pair<Type1, Type2>::Get_First() const {
  return m_first;
}
```

Pass by constant reference since we don’t know if the copying will be expensive.

Template



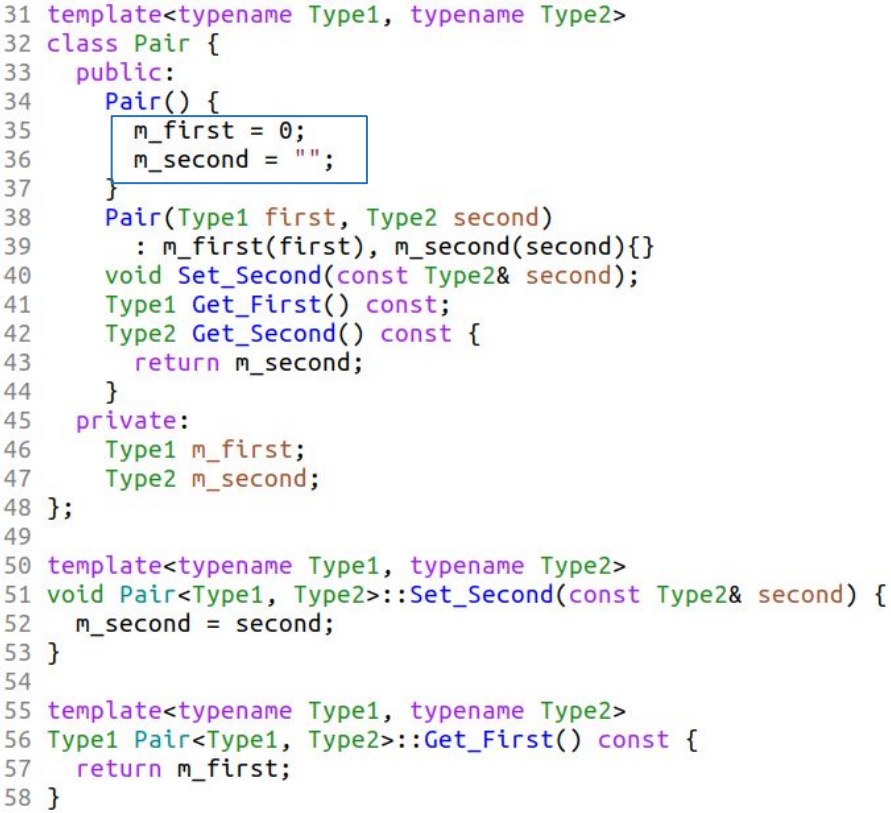
```
template<typename Type1, typename Type2>
class Pair {
  public:
    Pair() {
      m_first = 0;
      m_second = "";
    }
    Pair(Type1 first, Type2 second)
      : m_first(first), m_second(second){}
    void Set_Second(const Type2& second);
    Type1 Get_First() const;
    Type2 Get_Second() const {
      return m_second;
    }
  private:
    Type1 m_first;
    Type2 m_second;
};

template<typename Type1, typename Type2>
void Pair<Type1, Type2>::Set_Second(const Type2& second) {
  m_second = second;
}

template<typename Type1, typename Type2>
Type1 Pair<Type1, Type2>::Get_First() const {
  return m_first;
}
```

Create (int, string) Pair.

Pair<int, string> p;
Pair<int, string> p(1, “hi”);

Is there any possible runtime issue with this code?

Template



```
template<typename Type1, typename Type2>
class Pair {
  public:
    Pair() {
      m_first = 0;
      m_second = "";
    }
    Pair(Type1 first, Type2 second)
      : m_first(first), m_second(second){}
    void Set_Second(const Type2& second);
    Type1 Get_First() const;
    Type2 Get_Second() const {
      return m_second;
    }
  private:
    Type1 m_first;
    Type2 m_second;
};

template<typename Type1, typename Type2>
void Pair<Type1, Type2>::Set_Second(const Type2& second) {
  m_second = second;
}

template<typename Type1, typename Type2>
Type1 Pair<Type1, Type2>::Get_First() const {
  return m_first;
}
```

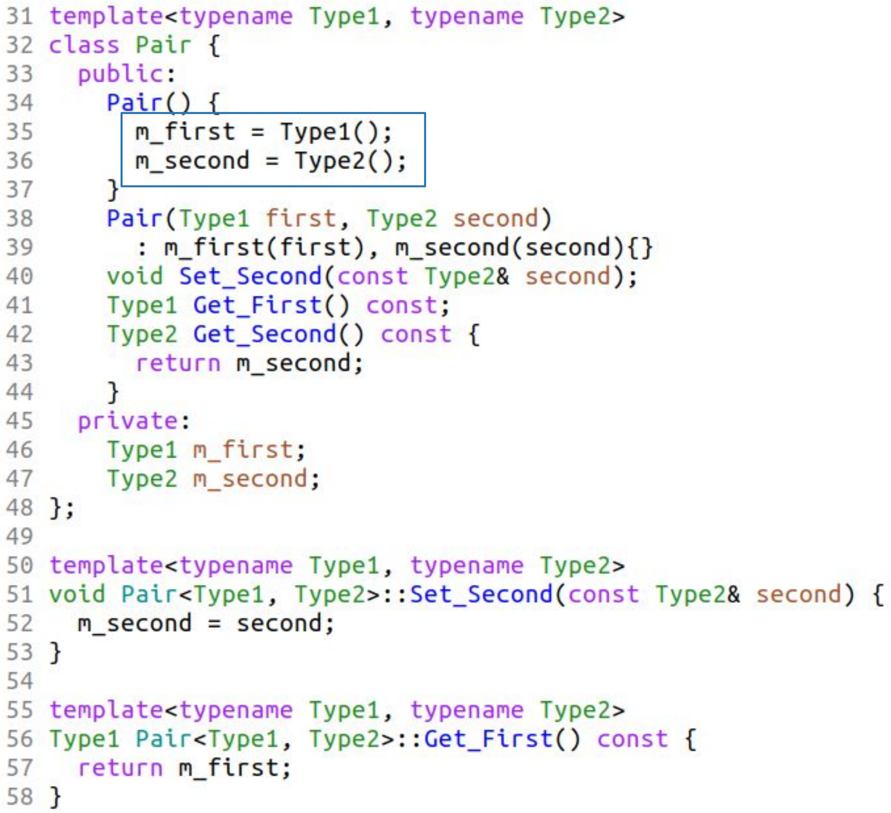
Is there any possible runtime issue with this code?

Pair <int, double> p;

Incorrect!
double m_second=“” .

How to fix it?

Template



```cpp
template<typename Type1, typename Type2>
class Pair {
  public:
    Pair() {
      m_first = Type1();
      m_second = Type2();
    }
    Pair(Type1 first, Type2 second)
      : m_first(first), m_second(second){}
    void Set_Second(const Type2& second);
    Type1 Get_First() const;
    Type2 Get_Second() const {
      return m_second;
    }
  private:
    Type1 m_first;
    Type2 m_second;
};

template<typename Type1, typename Type2>
void Pair<Type1, Type2>::Set_Second(const Type2& second) {
  m_second = second;
}

template<typename Type1, typename Type2>
Type1 Pair<Type1, Type2>::Get_First() const {
  return m_first;
}
```

Pair <int, double> p;

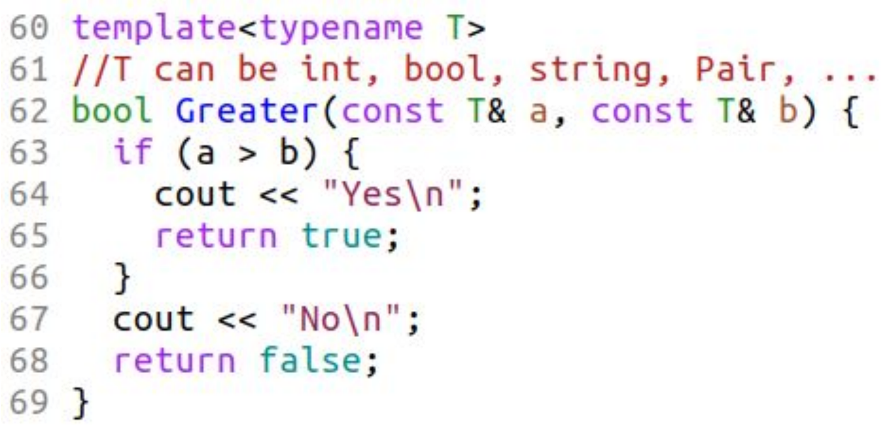
Correct!

Type() creates a Type object using the default constructor.

int(), double() are 0 by default.

string() is “” by default.

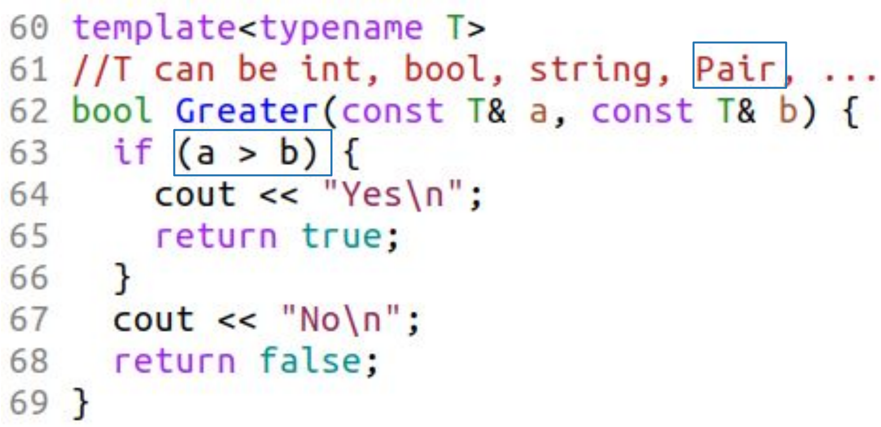
Template



```
template<typename T>
//T can be int, bool, string, Pair, ...
bool Greater(const T& a, const T& b) {
  if (a > b) {
    cout << "Yes\n";
    return true;
  }
  cout << "No\n";
  return false;
}
```

Is there any possible runtime issue with this code?

Template

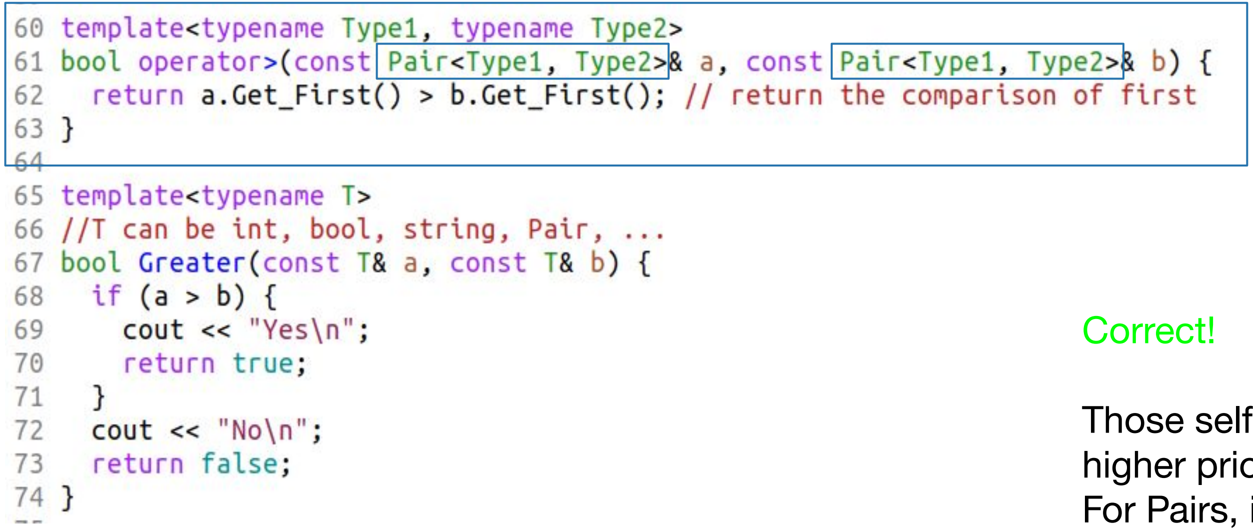


```
template<typename T>
//T can be int, bool, string, Pair, ...
bool Greater(const T& a, const T& b) {
  if (a > b) {
    cout << "Yes\n";
    return true;
  }
  cout << "No\n";
  return false;
}
```

Is there any possible runtime issue with this code?

Incorrect!
Pair<int, int> p1, p2;
cout<<Greater(p1, p2)<<endl;
‘>’ is not defined for Pairs.

Template



```
template<typename Type1, typename Type2>
bool operator>(const Pair<Type1, Type2>& a, const Pair<Type1, Type2>& b) {
  return a.Get_First() > b.Get_First(); // return the comparison of first
}

template<typename T>
//T can be int, bool, string, Pair, ...
bool Greater(const T& a, const T& b) {
  if (a > b) {
    cout << "Yes\n";
    return true;
  }
  cout << "No\n";
  return false;
}
```

Correct!

Those self-implemented functions have higher priority.
For Pairs, it will use the overloaded >.

Standard Template Library (STL)

Pros: greatly reduces the length of the code and amount of work since we no longer have to implement some data structures by ourselves.

Cons: some “programmers” don’t really know what data structures are used for STL and simply use them, which sometimes makes their programs slow.

In reality, almost all C++ programmers use STL for most data structures.

STL: containers

In my opinion, the best way to learn and verify operations of a STL container is looking up online. A good site is cplusplus.com: https://www.cplusplus.com/reference/stl

Define a container of type CType with elements of type EType.

Ctype<EType> c;

E.g. queue<int> q; vector<double> v; list<Pair<int, string>> l; …..

Iterator: a container “pointer” that “points” to the elements of the container.

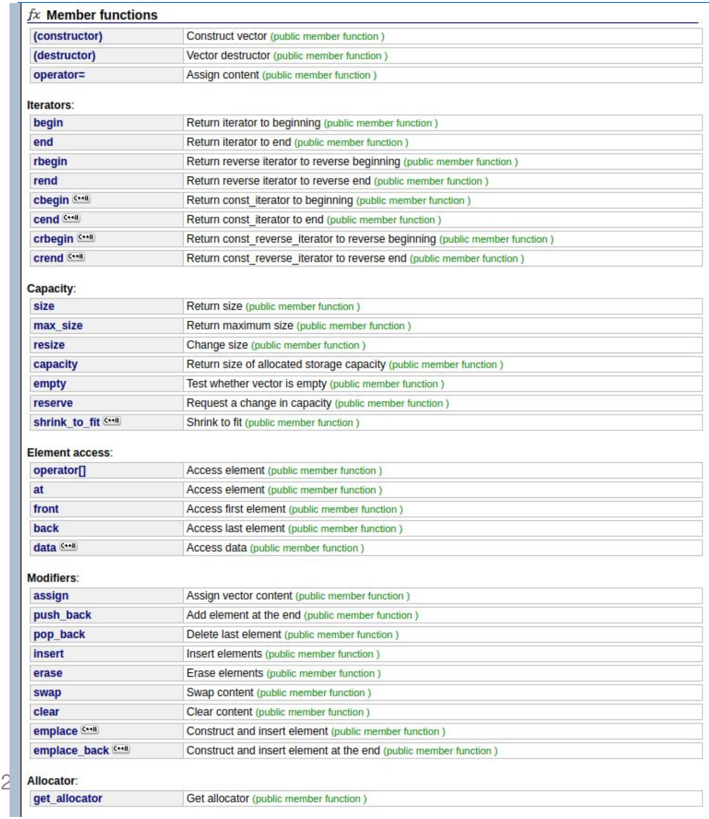
For a container c, usually
c.begin() returns an iterator to the location of its first element
c.end() returns an iterator to the location just passing the last element

Standard Template Library (STL)

queue<type> q: queue. #include <queue>
stack<type> s: stack. #include <stack>
vector<type>v : dynamic array (size not fixed). #include <vector>
list<type> l: linked list. #include <list>

Reminder: **generally, check the size of the container before popping, erasing and accessing elements.**

STL: vector



fx **Member functions**

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

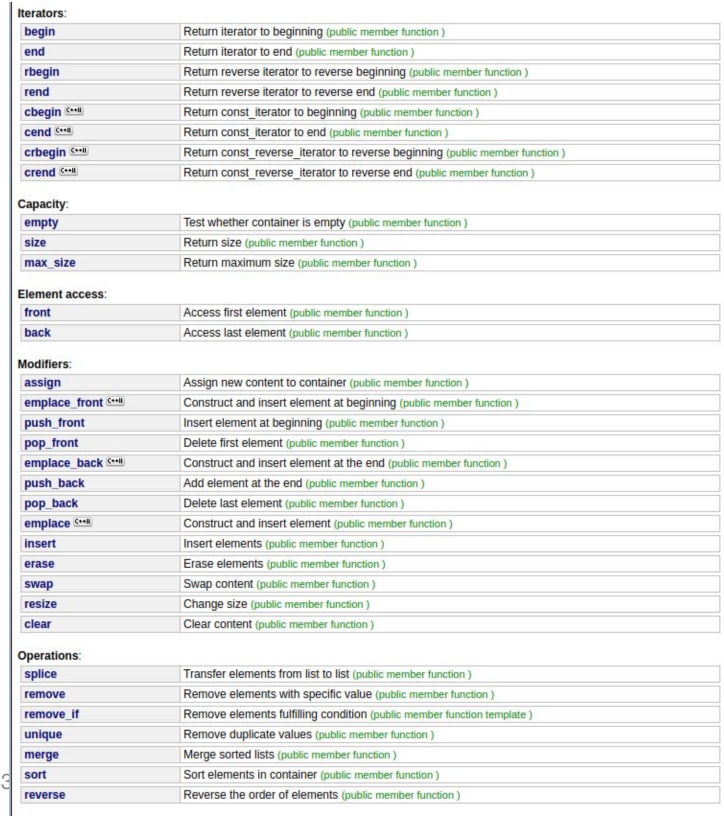
Allocator:

get_allocator	Get allocator (public member function)

Popular member functions:
vector <EType> v;
v[int n]
EType& v.at(int n)
EType v.front()
EType v.back()
void v.push_back(EType e)
bool v.empty()
int v.size()
Iterator v.erase(Iterator it)
Iterator v.insert(Iterator it, EType e)

Please read the documentations to learn more ways to use them.

STL: list



Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
emplace_front C++11	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace C++11	Construct and insert element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Operations:

splice	Transfer elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

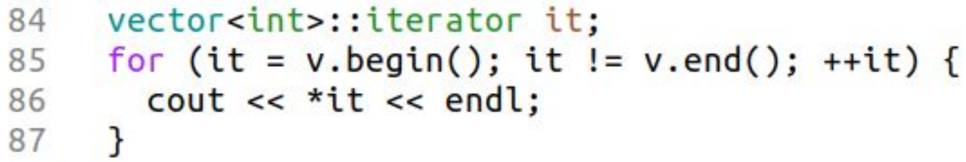
Popular member functions:
list <EType> l;
EType l.front()
EType l.back()
void l.push_back(EType e)
void l.pop_back()
void l.push_front(EType e)
void l.pop_front()
bool l.empty()
int l.size()
Iterator l.erase(Iterator it)
Iterator l.insert(Iterator it, EType e)

Please read the documentations to learn more ways to use them.

STL: iterator

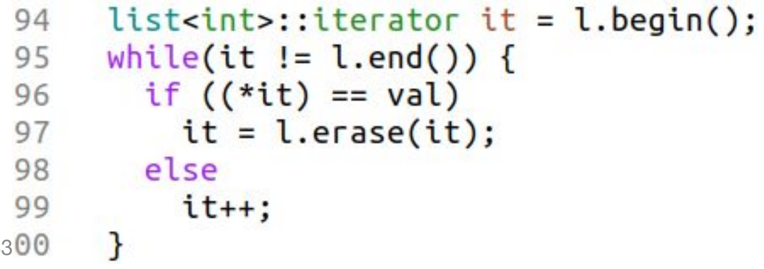
Create an iterator of container CType with elements EType:
Ctype<EType>::iterator it;
Move to next element: it++
Get the element value: *it

E.g. Traverse elements of an integer vector v.



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

E.g. Remove elements with value val in an integer list l.



```
list<int>::iterator it = l.begin();
while(it != l.end()) {
  if ((*it) == val)
    it = l.erase(it);
  else
    it++;
}
```

STL: iterator

Q: given a vector iterator it, can we do *(it+2) and it = it + 2?
Given a list iterator it, can we do *(it+2) and it = it + 2?
Why?

STL: iterator

Q: given a vector iterator it, can we do *(it+2) and it = it + 2?
Given a list iterator it, can we do *(it+2) and it = it + 2?
Why and how to resolve the issue?

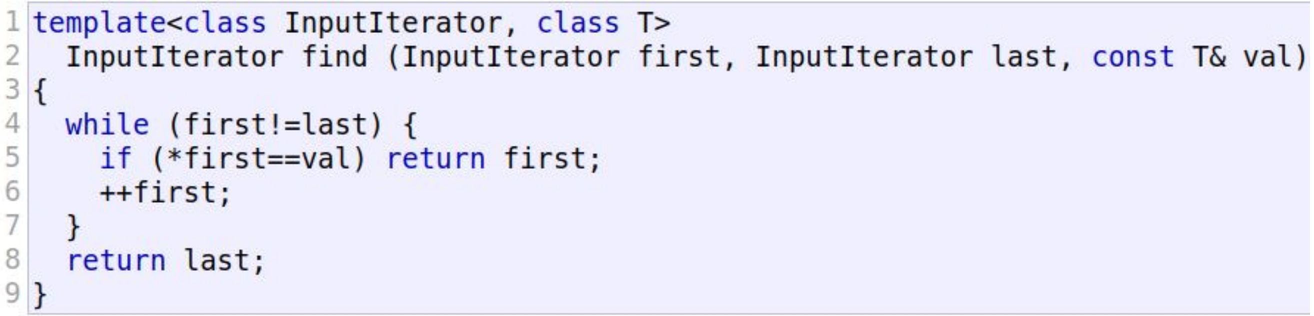
Yes for vector iterator since it's a dynamic array with contiguous space allocation. It + 2 points to 2 elements after it.
No for list iterator since for linked list the space is not contiguous. It + 2 doesn't move by 2 elements.
For list iterator, to get to 2 elements after the current it, one can do it++; it++;

Std library: algorithm

#include <algorithm>
This allows us to use implemented algorithms including find() and sort(), which can apply to STL containers and arrays.

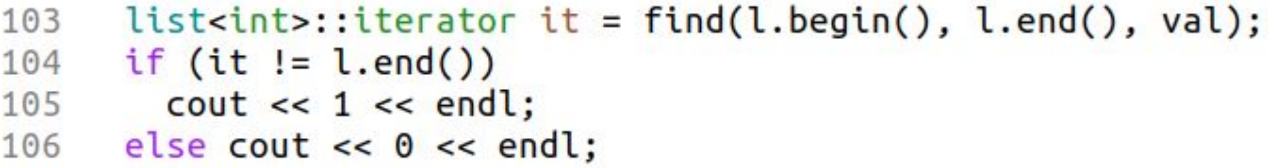
Std library: find by value

#include <algorithm>
Find by value:
iterator find(iterator begin, iterator end, EType value)
It returns end if value not found, an iterator to the element if found



```
template<class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val)
{
  while (first!=last) {
    if (*first==val) return first;
    ++first;
  }
  return last;
}
```

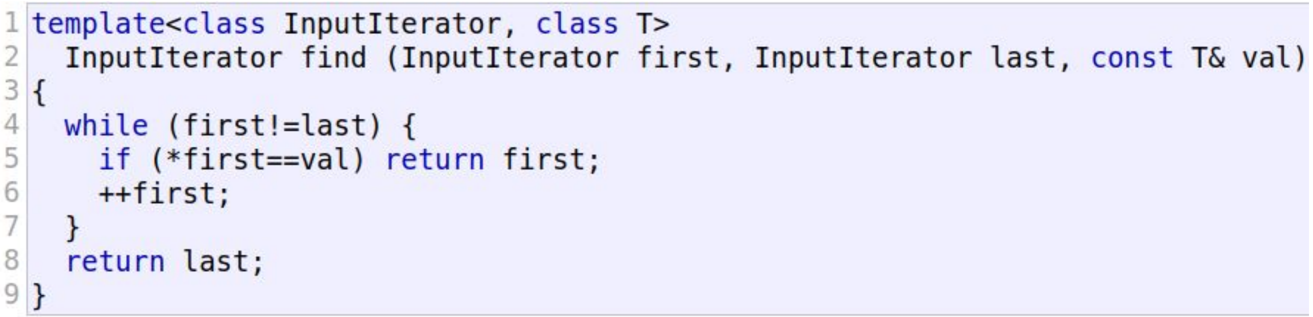
E.g. check if val is in an integer list l.



```
list<int>::iterator it = find(l.begin(), l.end(), val);
if (it != l.end())
  cout << 1 << endl;
else cout << 0 << endl;
```

Std library: find by value

#include <algorithm>
Find by value:
iterator find(iterator begin, iterator end, EType value)
It returns end if value not found, an iterator to the element if found

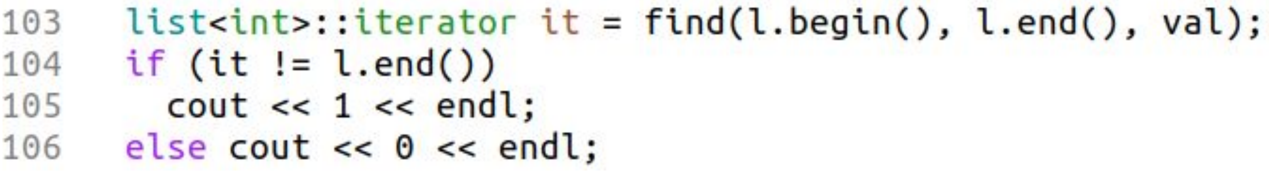


```
template<class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val)
{
  while (first!=last) {
    if (*first==val) return first;
    ++first;
  }
  return last;
}
```

Sometimes, value is not precisely defined.

For instance, for a self-defined class object, seach by value may not be well defined.

E.g. check if val is in an integer list l.



```
list<int>::iterator it = find(l.begin(), l.end(), val);
if (it != l.end())
  cout << 1 << endl;
else cout << 0 << endl;
```

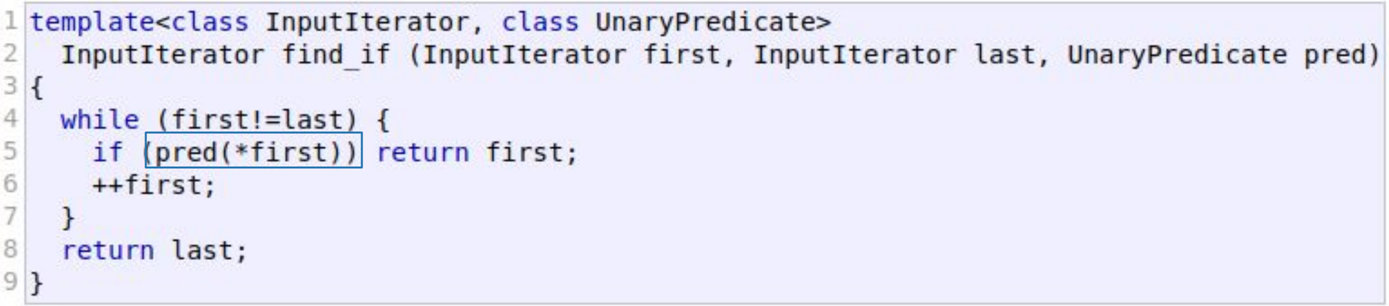
Std library: find by predicate

#include <algorithm>

Find by a predicate function:

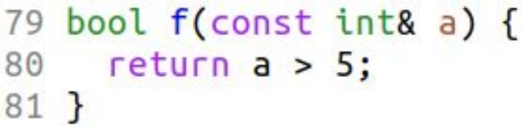
iterator find_if(iterator begin, iterator end, bool predicate function f)

It returns an iterator to the first element satisfying a predicate f, and end if not found.

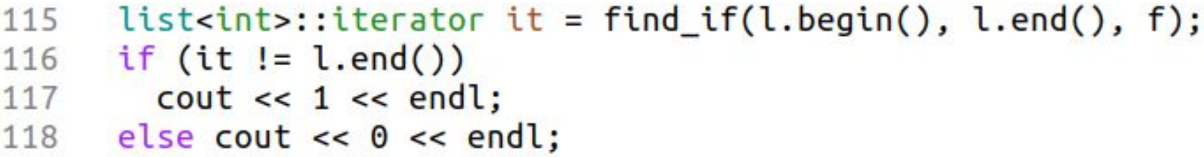


```
template<class InputIterator, class UnaryPredicate>
  InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
{
  while (first!=last) {
    if (pred(*first)) return first;
    ++first;
  }
  return last;
}
```

E.g. check if there is an element with value > 5 in an integer list l.



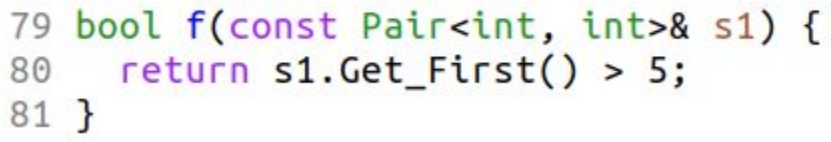
```
bool f(const int& a) {
    return a > 5;
}
```



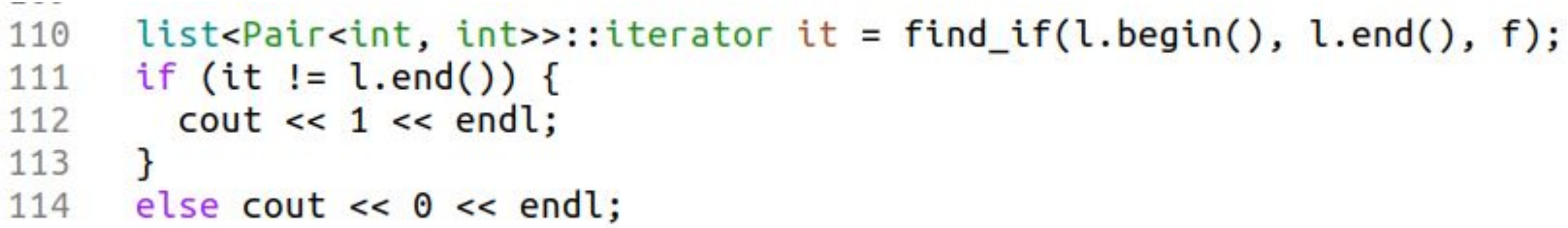
```
list<int>::iterator it = find_if(l.begin(), l.end(), f);
if (it != l.end())
  cout << 1 << endl;
else cout << 0 << endl;
```

Std library: find by predicate

Check if there’s a Pair in a list such that its first element is greater than 5.



```
bool f(const Pair<int, int>& s1) {
  return s1.Get_First() > 5;
}
```

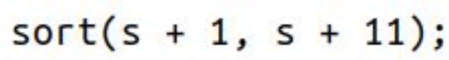


```
  list<Pair<int, int>>::iterator it = find_if(l.begin(), l.end(), f);
  if (it != l.end()) {
    cout << 1 << endl;
  }
  else cout << 0 << endl;
```

Std library: sort by value

#include <algorithm>
Sort by value:
void sort(iterator begin, iterator end)

E.g. sort an array of strings (alphabetical order) s[] from s[1] to s[10].



```
sort(s + 1, s + 11);
```

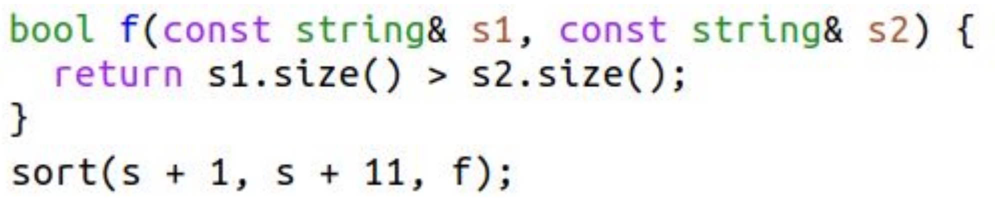
Sometimes, value is not precisely defined.

For self-defined class objects, ‘<’ is not well defined.

Std library: sort by predicate

#include <algorithm>
Sort by a predicate function:
void sort(iterator begin, iterator end, bool predicate function f)

E.g. sort an array of strings by length (larger length comes first, same length preserves original order).

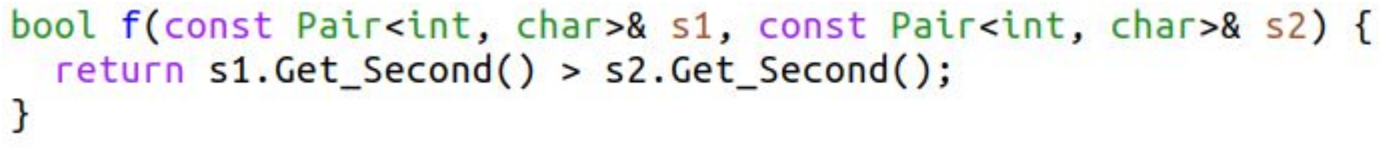


```
bool f(const string& s1, const string& s2) {
  return s1.size() > s2.size();
}
sort(s + 1, s + 11, f);
```

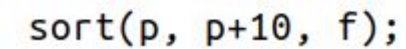
Interpretation the predicate:
changes the order of s1 and s2 if and only if f returns true.

Std library: sort by predicate

Sort an array of pairs<int, char> in decreasing order by comparing the values of their second elements.



```
bool f(const Pair<int, char>& s1, const Pair<int, char>& s2) {
  return s1.Get_Second() > s2.Get_Second();
}
```



```
sort(p, p+10, f);
```