C++ Containers
(Standard Template Library)
Why Container Classes?

• Many programs use arrays, vectors, lists, queues, stacks, sets to store information.
• Both C++ and Java provide container classes that automatically manage memory, i.e. they allocate additional memory when more elements are added.
• The supported container classes greatly reduce the amount of code and programming needed and improve productivity.
• Container classes and OOP are closely related:
  − Containers hold objects of different derived classes
  − Polymorphism properly invokes the correct methods
Container Class (For Code Reuse)

• A container needs to be able to hold items of different types (i.e. classes). Examples
  − list of strings, integers, floating points, student objects
  − queues of undergraduates, graduate students, staff and faculty
  − maps: name → address, student ID → name, course title → classroom

• C++ standard template library (STL) and Java container classes provide such functionality.

• A question that will need to be answered: How do we write containers that work for a variety of class types and primitive types?
Selecting a container class

- random or sequential accesses?
- allow unique or duplicate items?
- $O(1)$ or $O(N)$ for array-like access (using [index])
- efficient insert / delete?
  - front
  - end
  - middle
- Java containers cannot store primitive types (int, char, float ...), they can store objects only. Primitive types, however, have corresponding object types (e.g. Integer, Boolean) that can be held in containers.
- C++ containers can store primitives.
# Efficiency

<table>
<thead>
<tr>
<th>operation</th>
<th>vector</th>
<th>deque</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-like access</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>insert/delete at front</td>
<td>$O(N)$</td>
<td>$O(1)+$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert/delete at end</td>
<td>$O(1)+$</td>
<td>$O(1)+$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert/delete in middle</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

$N$: current number of items
Two suggestions when using containers

• If code you are writing can ever exist in a multithreaded environment
  – Make sure the container is thread safe or add your own synchronization
  – Make sure actions on objects stored in the container are thread safe

• If you have the choice of using a Java or C++ container or writing your own, use the supplied one
  – Even if yours and their’s are both O(N), their constant will almost certainly be smaller than yours
  – If thread safe, smart people will have spent lots of time tuning this to avoid unnecessary synchronization
C++ Vector
// VectorBasic.cc From Prof Kak

#include <iostream>
#include <vector>
using namespace std;

void print( vector<int> );

int main() {
    vector<int> vec;
    vec.push_back( 34 );
    vec.push_back( 23 );  // size is now 2
    print( vec );        // 34 23

    vector<int>::iterator p;
    p = vec.begin();
    *p = 68;
    *(p + 1) = 69;
    // *(p + 2) = 70;       // WRONG
    print( vec );        // 68 69
    vec.pop_back();      // size is now 1
    print( vec );        // 68

    vec.push_back(101);
    vec.push_back(103);  // size is now 3
    int i = 0;
    while ( i < vec.size() )
        cout << vec[i++] << "  ";
    cout << endl;        // 68 101 103
    vec[0] = 1000;
    vec[1] = 1001;
    vec[2] = 1002;
    print( vec );        // 1000 1001 1002

    return 0;
}

void print( vector<int> v ) {
    cout << "\nvector size is: ";
    cout << v.size() << endl;
    vector<int>::iterator p = v.begin();
    while ( p != v.end() )
        cout << *p++ << "  ";
    cout << endl;
    cout << endl;
}


// VectorBasic.cc From Prof Kak

#include <iostream>
#include <vector>
using namespace std;

void print( vector<int> );

int main()
{
    vector<int> vec;
    vec.push_back( 34 );
    vec.push_back( 23 );     // size is now 2
    print( vec );                   // 34 23

    // Function prototype for a
    // print function that takes a
    // vector of ints as an argument.

    // Declare a vector of ints
    // called vec.

    // Add to the back of the vector
    // 34 and 23.
Iterators

- Iterators are easy ways to traverse a collection of objects
- To be safe, unless allowed or specified by the documentation:
  - Don’t assume an order for how objects are visited
  - Don't change what is being iterated on - be especially careful of adds and deletes
  - Don't assume iterators are thread safe
    - CopyOnWriteArrayList is
    - Vector iterator is not
```cpp
vector<int>::iterator p;
p = vec.begin();
*p = 68;
*(p + 1) = 69;
// *(p + 2) = 70;          // WRONG
print( vec );             // 68 69
vec.pop_back();           // size is now 1
print( vec );             // 68
```

Note the pointer based notation. $p$ really an iterator.

Attempted access of an undefined element.
vec.push_back(101);
vec.push_back(103);    // size is now 3

int i = 0;
while (i < vec.size())
    cout << vec[i++] << " " << endl; // 68 101 103
vec[0] = 1000;
vec[1] = 1001;
vec[2] = 1002;
print( vec );  // 1000 1001 1002
void print( vector<int> v ) {
    cout << "\nvector size is: ";
    cout << v.size() << endl;
    vector<int>::iterator p = v.begin();
    while ( p != v.end() )
        cout << *p++ << "  ";
    cout << endl << endl;
}

Iterators *necessary* in an OO language for encapsulation

- If an iterators were not provided, you would have to know how elements are physically stored or linked to access them
- With iterators, you only need an interface to access them
- It will give you elements, but you don’t need to know the details of how the elements are stored
- Same is true with vector through pointer -- allowing you to say (as was done erroneously in an earlier slide) that \((p+2) = \ldots\) would imply something *more than you can know from what your program has already done* about the storage allocated.
C++ Vector

• contiguous memory
• efficient access (array-like [index])
• efficient insert / delete at the end (allocate more memory occasionally)
• inefficient insert / delete at the front (will move all elements down to do the insertion)
• automatic expand / shrink allocated memory (allocate more than necessary to reduce allocation / release / copy overhead)
• occasional copying of the whole vector
C++ Iterator

pointer notation (really an iterator) to traverse a vector or other STL (standard template library) container

```cpp
vector<int> v;
cout << "\nv vector size is: " << v.size() << endl;
vector<int>::iterator p = v.begin();
while ( p != v.end() ) {
    cout << "*p << " " ";
    p++;
}
```

Reallocation storage can cause iterators to be invalid.
C++ List
```cpp
#include <iostream>   // for cout, endl
#include <string>
#include <list>
using namespace std;

void print(list<string>& li) {
    typedef list<string>::const_iterator CI;
    cout << "The number of items in the list: " << li.size() << endl;;
    for (CI iter = li.begin(); iter != li.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl << endl;
}

int main() {
    list<string> animals; // (A)

    animals.push_back("cheetah"); // (B)
    animals.push_back("lion");  // (C)
    animals.push_back("cat");   // (D)
    animals.push_back("fox");   // (E)
    animals.push_back("elephant"); // (F)
    animals.push_back("cat"); // duplicate cat  // (G)

    print(animals); // cheetah lion cat fox
                     // elephant cat

    animals.pop_back(); // (H)
    print(animals); // cheetah lion cat fox
                     // elephant

    animals.remove("lion"); // first occurrence of lion  // (I)
    print(animals); // cheetah cat fox elephant

    animals.push_front("lion"); // (J)
    print(animals); // lion cheetah cat fox elephant
    animals.pop_front(); // (K)
    print(animals); // cheetah cat fox elephant

    animals.insert(animals.end(), "cat"); // (L)
    print(animals); // cheetah cat fox elephant cat

    animals.sort(); // (M)
    print(animals); // cat cat cheetah elephant fox
    animals.unique(); // (N)
    print(animals); // cat cheetah elephant fox

    // another list needed for demonstrating splicing and merging:
    list<string> pets; // (O)
    pets.push_back("cat");
    pets.push_back("dog");
    pets.push_back("turtle");
    pets.push_back("bird");

    animals.splice(animals.begin(), pets, pets.begin()); // (P)
    print(animals); // cat cat cheetah elephant fox
    print(pets); // dog turtle bird

    pets.sort(); // (Q)
    animals.merge(pets); // (R)

    cout << pets.empty() << endl; // true  // (S)
    print(animals); // bird cat cheetah // (T)
                     // dog elephant fox
                     // turtle
    return 0;
}
```

```cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;

void print(list<string>&);
int main() {
    list<string> animals; // (A)

    animals.push_back("cheetah"); // (B)
    animals.push_back("lion"); // (C)
    animals.push_back("cat"); // (D)
    animals.push_back("fox"); // (E)
    animals.push_back("elephant"); // (F)
    animals.push_back("cat"); // duplicate cat

    print(animals); // cheetah lion cat fox
        // elephant cat

    animals.pop_back(); // (H)
    print(animals); // cheetah lion cat fox elephant

    animals.remove("lion"); // first occurrence of lion
    print(animals); // cheetah cat fox elephant
animals.push_front("lion");
print(animals); // lion cheetah cat fox elephant
animals.pop_front();
print(animals); // cheetah cat fox elephant

animals.insert(animals.end(), "cat");
print(animals); // cheetah cat fox elephant cat

animals.sort();
print(animals); // cat cat cheetah elephant fox

animals.unique();
print(animals); // cat cheetah elephant fox

Only examines adjacent elements so most useful on sorted lists
//another list needed for demonstrating
//splicing and merging:
list<string> pets; // (O)
pets.push_back("cat");
pets.push_back("dog");
pets.push_back("turtle");
pets.push_back("bird");
animals.splice(animals.begin(), pets, pets.begin());
print(animals); // cat cat cheetah elephant fox
print(pets); // dog turtle bird

pets.sort(); // bird dog turtle

animals.merge(pets);

cout << pets.empty() << endl; // true

print(animals); // bird cat cat cheetah
                // dog elephant fox
                // turtle
typedef creates a type CI used to declare variables (e.g., iter below)

```cpp
void print(list<string>& li) {
    typedef list<string>::const_iterator CI;
    cout << "The number of items in the list: ";
    cout << li.size() << endl;
    for (CI iter = li.begin(); iter != li.end(); iter++) {
        cout << *iter << " ";
    }
    cout << endl << endl;
}
```

With a const_iterator, iterator can be changed, but not what it points to
C++ Vector of objects
```cpp
#include <iostream>
#include <vector>
#include <algorithm>  // for sort() Algorithm is a collection of templates
using namespace std;

class X {
    int p;
public:
    X() {
        p = 42;
    }
    X(int q) {
        p = q;
    }
    int getp() const {
        return p;
    }
    void changeState(int pp) {
        p = pp;
    }
};
```

Zero arg constructor

getters and setters for \( p \)
Overloaded comparison operators

```cpp
bool operator<(const X& x1, const X& x2) {
    return x1.getp() < x2.getp();
}

bool operator==(const X& x1, const X& x2) {
    return x1.getp() == x2.getp();
}
```

• Could be member functions
• Would declare as

```cpp
bool operator==(const X& x2) const;
```

which “==” is used?

later called by sort which passes a `const this` pointer
forward declaration of print. Why can’t it be declared as a member function?

```cpp
void print(vector<X>);

int main() {
    vector<X> vec;
    X x1(2);
    X x2(3);
    X x3(5);

    vec.push_back(x1);
    vec.push_back(x3);
    vec.push_back(x2);

    print(vec); // 2 5 3
    x2.changeState(1000);

    // change made to x2 above does not affect copy
    // of x2 in vec;
    print(vec); // 2 5 3
```
// vector elements initialized by X’s no-arg
// constructor:
vector<X> vec_2(5);
print(vec_2); // 42 42 42 42 42
vec_2.resize(7);
print(vec_2); // 42 42 42 42 42 42 42

// uninitialized increase in the vector capacity
vec_2.reserve(10);
cout << vec_2.capacity() << endl; // 10
print(vec_2); // 42 42 42 42 42 42 42
// size still returns 7;
cout << vec_2[8].getp() << endl; // undefined

• reserve(n) gives the vector a capacity of at least n, it may be more
• If capacity is already n, essentially a no-op
// set up vector for sorting
vec_2[0] = X(12);
vec_2[1] = X(36);
vec_2[2] = X(3);
vec_2[3] = X(56);
vec_2[4] = X(2);

sort(vec_2.begin( ), vec_2.end( ));
print(vec_2); // 2 3 12 36 42 42 56
vec_2.clear( );
print(vec_2); // nothing printed, empty

cout << vec_2.capacity( ) << endl; // 10
return 0;

• sort is a member of the algorithms Standard Template Library

• sort(vec_2.begin(), vec_2.end()); causes a template to be expanded (if not done already) that creates a sort that operates on the range given.

• sort operates on the elements of the vector via an iterator
void print(vector<X> v) {
    cout << "\nvector size is: ";
    cout << v.size() << endl;
    vector<X>::iterator p = v.begin();
    while (p != v.end()) {
        cout << (*p++).getp() << " ";
    }
    cout << endl << endl;
}

Note that p is an iterator, not a pointer. "*" and "++" are over-loaded operators that get the element currently indicated by the iterator or move to the next element, respectively. More natural in a C/C++ context than .get() and .next()
Why Create New Containers?

• need efficient ways to store and access objects
• reuse the same implementation for different classes
Creating New C++ Containers
C++ Binary Search Tree
Binary Search Tree

- A tree contains a group of nodes.
- A node can have a left child and a right child.
- If a node has no child, the node is called a leaf node.
- Each node has a unique value.
- When a new value is inserted into the tree, if the value already appears, the insertion has no effect.
- If the value is smaller than a node, the value is inserted as the left child of the node. If the node already has a left child, the value is inserted as a child of the child (recursively).
• insert the first value, 17
• insert 9
• insert 23

• insert 4
• insert 11

• insert 31
Traverse Tree (In-Order)

if (left child is not empty)
{ visit left child; }
print value;
if (right child is not empty)
{ visit right child; }

⇒ The output values are sorted.
4, 9, 11, 17, 23, 31
#include <iostream>
#include <string>
using namespace std;

template <class TPL> class BinaryNode {
    BinaryNode * bn_left;
    BinaryNode * bn_right;
    TPL bn_content;
public:
    BinaryNode(TPL content);
    virtual ~BinaryNode();
    void insertContent(TPL content);
    bool searchContent(TPL content) const;
    void print(int depth) const;
};

This is the .h declarations for the template

View TPL as a declaration of a symbol that has type Class, just as int i is a variable of type int.
template <class TPL> class BinaryNode {
    BinaryNode * bn_left;
    BinaryNode * bn_right;
    TPL bn_content;

public:
    BinaryNode(TPL content);
    virtual ~BinaryNode( );
    void insertContent(TPL content);
    bool searchContent(TPL content);
    void print(int depth) const;
};

Two ways to declare this: class TPL or typename TPL

Some say use class if expecting a class name, typename if expecting Template type, typedef or primitive, i.e., is a usage hint.
Some people prefer `typename` to avoid overloading of `class`. If you have template parameters to a template, **must** use `typename`.

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

template <class TPL> class BinaryNode {
    BinaryNode * bn_left;
    BinaryNode * bn_right;
    TPL bn_content;

public:
    BinaryNode(TPL content);
    virtual ~BinaryNode( );
    void insertContent(TPL content);
    bool searchContent(TPL content) const;
    void print(int depth) const;
};
```
Some history about `class, typename`

For details see

http://stackoverflow.com/questions/213121/use-class-or-typename-for-template-parameters

and


From Lippman: When C++ designed, the decision was made to not make an additional keyword to declare types that are parameters to templates. Why add another keyword to the language? Now, consider the following:

template <class T> class Demonstration {
public:
    void method( ) {
        T::A *aObj; // expression or declaration?
    }
};
template <class T> class Demonstration {
public:
    void method( ) {
        T::A *aObj; // expression or declaration?
    }
};

Programmer clearly wants to declare a pointer to aObj that is of type A that is defined within template parameter type T.

C++ compilers will interpret this as a static (class) variable from the class T (T::A) multiplied times a local variable aObj, with the result discarded.
How to fix this?

template <typename T> class Demonstration {
public:
    void method() {
        typename T::A *aObj; // expression or declaration?
    }
};

Create a new keyword *typename* and change the code to the above. Instructs the compiler to treat what follows as a declaration.

Once the keyword was introduced the ISO standards committee decided to revisit the use of *class* and allow *typename* to be used there, as well.
template <class TPL> 
BinaryNode<TPL>::BinaryNode(TPL content) { 
    bn_content = content; 
    bn_left = NULL; 
    bn_right = NULL; 
} 

This is the .cpp stuff for the template.

In particular, this declares a constructor for the BinaryNode<TPL> class
template <class TPL> void BinaryNode<TPL>::insertContent(TPL content) {
    if (content == bn_content) { return; } // no duplicates
    if (content < bn_content) {
        if (bn_left == NULL) {
            BinaryNode<TPL> *newnode = new BinaryNode<TPL>(content);
            bn_left = newnode;
        } else {
            bn_left->insertContent(content);
        }
    } else {
        if (bn_right == NULL) {
            BinaryNode<TPL> *newnode = new BinaryNode<TPL>(content);
            bn_right = newnode;
        } else {
            bn_right->insertContent(content);
        }
    }
}
template <class TPL> bool BinaryNode<TPL>::searchContent(TPL content) const {
    if (content == bn_content) {return true;}
    if (content < bn_content) {
        if (bn_left == NULL) {return false;}
        return bn_left->searchContent(content);
    } else {
        if (bn_right == NULL) {return false;}
        return bn_right->searchContent(content);
    }
}
template <class TPL> void BinaryNode<TPL>::print(int depth) const {
    if (bn_left != NULL) {bn_left->print(depth+1);}  
    for (int identcnt = 0; identcnt < depth; identcnt++)  
    {  
        cout << "\t";    
    }  
    cout << bn_content << endl;   
    if (bn_right != NULL) {bn_right->print(depth+1);}  
}

template <class TPL> BinaryNode<TPL>::~BinaryNode( ) {  
    if (bn_left != NULL) {delete bn_left;}  
    if (bn_right != NULL) {delete bn_right;}  
}

More of the .cpp file
class Student {
    string s_name;

public:
    Student(string name): s_name(name) {} 
    Student(const Student& orig): s_name(orig.s_name){} 
    Student( ) {s_name = "";} 
    bool operator<(const Student& arg2) {
        return (s_name < arg2.s_name);
    }
    bool operator==(const Student& arg2) {
        if (s_name == arg2.s_name) {return true;} 
        else {return false;} 
    }
    friend ostream& operator<<(ostream& os, const Student& stu);
};

ostream& operator<<(ostream& os, const Student& stu) {
    os << stu.s_name << endl;
    return os;
class User {
    int u_age;
public:
    User(int age): u_age(age) { }
    User(): u_age(0) { }
    bool operator<(const User& arg2) { 
        return (u_age < arg2.u_age);
    }
    bool operator==(const User& arg2) { 
        return (u_age == arg2.u_age);
    }
    friend ostream& operator<<(ostream& os, const User& user);
};

ostream& operator<<(ostream& os, const User& usr) { 
    os << usr.u_age << endl;
    return os;
}
int main(void) {
    BinaryNode<int> bnint(5);
    bnint.insertContent(3);
    bnint.insertContent(6);
    bnint.insertContent(4);
    bnint.insertContent(7);
    bnint.insertContent(6);
    bnint.insertContent(9);
    bnint.print(0);
    cout << "______" << endl;
}

Root of the tree
Student stu1("John");
Student stu2("Mary");
Student stu3("Tom");
Student stu4("Amy");
Student stu5("Ted");

BinaryNode<Student> bnstu(stu1);
bnstu.insertContent(stu2);
bnstu.insertContent(stu3);
bnstu.insertContent(stu4);
bnstu.insertContent(stu5);
bnstu.print();

cout << "__________" << endl;

The main fct.
User usr1(21);
User usr2(28);
User usr3(19);
User usr4(17);
User usr5(22);
User usr6(20);
User usr7(18);
BinaryNode<User> bnusr(usr1);
bnusr.insertContent((User)3);
bnusr.insertContent(usr2);
bnusr.insertContent(usr3);
bnusr.insertContent(usr4);
bnusr.insertContent(usr5);
bnusr.insertContent(usr6);
bnusr.insertContent(usr7);
bnusr.print(0);
cout << "__________" << endl;
User usr1(21);
User usr2(28);
User usr3(19);
User usr4(17);
User usr5(22);
User usr6(20);
User usr7(18);

BinaryNode<User> bnusr(usr1);
bnusr.insertContent((User)3);
bnusr.insertContent(usr2);
bnusr.insertContent(usr3);
bnusr.insertContent(usr4);
bnusr.insertContent(usr5);
bnusr.insertContent(usr6);
bnusr.insertContent(usr7);
bnusr.print(0);
cout << "___________" << endl;

from stackoverflow: “This is legal because C++ interprets any constructor that can be called with a single argument of type T as a means of implicitly converting from Ts to the custom object “type.”
The main function

cout << bnint.searchContent(11) << " "
   << bnint.searchContent(10) << endl;
Student stu6("Ted");
cout << bnstu.searchContent(stu6)<< " "
   << bnstu.searchContent(stu2)<< endl;
cout << bnusr.searchContent(usr2)<< " "
   << bnusr.searchContent(usr4)<< endl;
The template language is Turing complete

- Any computable problem can be solved by C++'s Template language
- I don't suggest you exploit this, in general
#include <iostream>

template <int N> struct Factorial {
enum { val = Factorial<N-1>::val * N }
};

template<> struct Factorial<0> {
enum { val = 1 }
};

int main() {
// Value generated at compile time.
// Most compilers limit recursion depth
std::cout << Factorial<4>::val << "\n";
}
Instantiating templates

• Declare and define the template in the same file, as we did here
• This will ensure that the template is *instantiated*.
• If the template is not clearly defined to the C++ compiler, it will not be instantiated
  – classes intended to be created by the template will not exist
• Template instantiation is difficult for the C++ system
  – A template for a type can only be instantiated once, otherwise multiple copies of static variables will exist or there will be linker errors
  – Every template must be instantiated once for each use with a type
Two common models

• The *Borland* model
  – each compilation unit (file) that uses a template instantiation creates an instantiation.
  – The equivalent of a *common* block is created for class statics
  – The linker collapses these into a single version

• The AT&T C++ *Cfront* model
  – Create a repository where template instantiations live
  – Only allow one/per template<type>
  – Problems when multiple programs live in the same directory or one program spans multiple directories