Vectors

First-class mechanism for representing lists

Standard Template Library

- What is it?
  - Collection of container types and algorithms supporting basic data structures
- What is a container?
  - A generic list representation allowing programmers to specify which types of elements their particular lists hold
    - Uses the C++ template mechanism
- String class is part of the STL

STL Container Classes

- Sequences
  - deque, list, and vector
    - Vector supports efficient random-access to elements
- Associative
  - map, set
- Adapters
  - priority_queue, queue, and stack

Vector Class Properties

- Provides list representation comparable in efficiency to arrays
- First-class type
- Efficient subscripting is possible
  - Indices are in the range 0 ... size of list - 1
- List size is dynamic
  - Can add items as we need them
- Index checking is possible
  - Through a member function
- Iterators
  - Efficient sequential access

Example

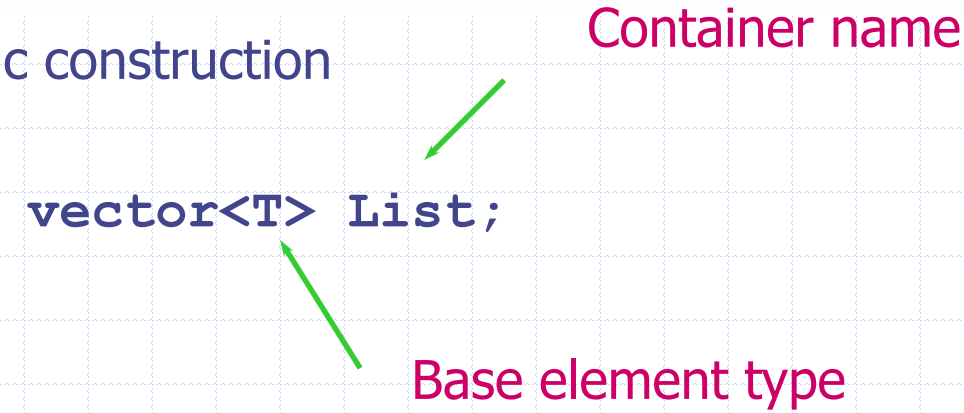
```
#include <vector>
#include <iostream>
using namespace std;
int main() {
   vector<int> A(4, 0); // A: 0 0 0 0
   A.resize(8, 2);      // A: 0 0 0 0 2 2 2 2
   vector<int> B(3, 1); // B: 1 1 1
   for (int i = 0; i < B.size(); ++i) {
       A[i] = B[i] + 2;
   }                    // A: 3 3 3 0 2 2 2 2
   A = B;               // A: 1 1 1
   return 0;
}
```

Some Vector Constructors

- **`vector()`**
  - The default constructor creates a vector of zero length
- **`vector(size_type n, const T &val = T())`**
  - *Explicit* constructor creates a vector of length **`n`** with each element initialized to **`val`**
- **`vector(const T &V)`**
  - The copy constructor creates a vector that is a duplicate of vector **`V`**.
    - Shallow copy!

Construction

- Basic construction

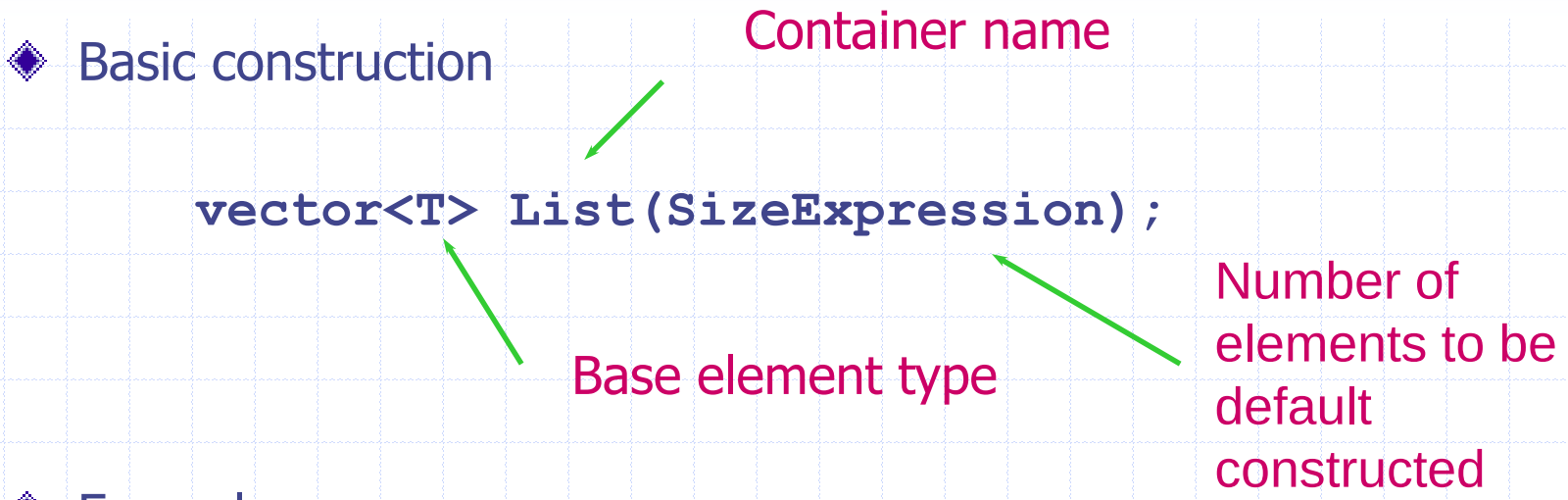


- Example

```
vector<int> A;              // 0 ints
vector<float> B;            // 0 floats
vector<Rational> C;         // 0 Rationals
```

Construction

- Basic construction

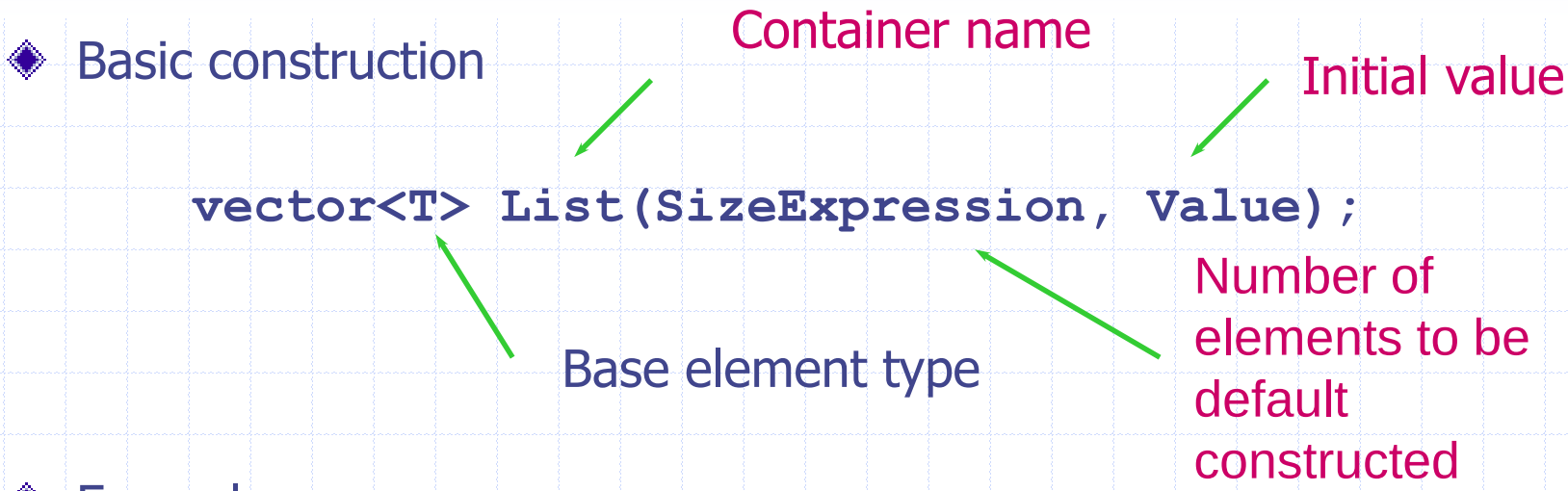


- Example

```
vector<int> A(10);        // 10 ints
vector<float> B(20);      // 20 floats
vector<Rational> C(5);    // 5 Rationals
int n = PromptAndRead();
vector<int> D(n);         // n ints
```

Construction

- Basic construction



- Example

```
vector<int> A(10, 3);       // 10 3s
vector<float> B(20, 0.2);   // 20 0.2s
Rational r(2/3);
vector<Rational> C(5, r);   // 5 2/3s
```

Vector Interface

- `size_type size() const`
  - Returns the number of elements in the vector

```
cout << A.size();      // display 10
```

- `bool empty() const`
  - Returns true if there are no elements in the vector; otherwise, it returns false

```
if (A.empty()) {
    // ...
```

Vector Interface

- `vector<T>& operator = (const vector<T> &V)`
  - The member assignment operator makes its vector representation an exact duplicate of vector V.
    - Shallow copy
  - The modified vector is returned

```
vector<int> A(4, 0); // A: 0 0 0 0
vector<int> B(3, 1); // B: 1 1 1
A = B;               // A: 1 1 1
```

Vector Interface

- `reference operator [](size_type i)`
  - Returns a reference to element `i` of the vector
    - Lvalue
- `const_reference operator [](size_type i) const`
  - Returns a constant reference to element `i` of the vector
    - Rvalue

Example

```
vector<int> A(4, 0);                        // A: 0 0 0 0
const vector<int> B(4, 0);                  // B: 0 0 0 0

for (int i = 0; i < A.size(); ++i) {
  A[i] = 3;                                 // lvalue
}                                           // A: 3 3 3 3

for (int i = 0; i < A.size(); ++i) {
  cout << A[i] << endl;                     // rvalue
  cout << B[i] << endl;                     // rvalue
}
```

Vector Interface

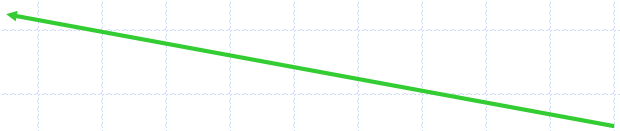
- `reference at(size_type i)`
  - If `i` is in bounds, returns a reference to element `i` of the vector; otherwise, throws an exception
- `const_reference at(size_type i) const`
  - If `i` is in bounds, returns a constant reference to element `i` of the vector; otherwise, throws an exception

Example

```
vector<int> A(4, 0);          // A: 0 0 0 0

for (int i = 0; i <= A.size(); ++i) {
  A[i] = 3;
}                             // A: 3 3 3 3 ??

for (int i = 0; i <= A.size(); ++i) {
  A.at(i) = 3;
}                             // program terminates
                              // when i is 4
```



Vector Interface

- `void resize(size_type s, T val = T())`
  - The number of elements in the vector is now `s`.
    - To achieve this size, elements are deleted or added as necessary
      - Deletions if any are performed at the end
      - Additions if any are performed at the end
      - New elements have value `val`

```
vector<int> A(4, 0); // A: 0 0 0 0
A.resize(8, 2);      // A: 0 0 0 0 2 2 2 2
A.resize(3,1);       // A: 0 0 0
```

Function Examples

```
void GetList(vector<int> &A) {
   int n = 0;
   while ((n < A.size()) && (cin >> A[n])) {
       ++n;
   }
   A.resize(n);
}
```

```
vector<int> MyList(3);
cout << "Enter numbers: ";
GetList(MyList);
```

Examples

```
void PutList(const vector<int> &A) {
   for (int i = 0; i < A.size(); ++i) {
       cout << A[i] << endl;
   }
}
```

```
cout << "Your numbers: ";
PutList(MyList)
```

Vector Interface

- `pop_back()`
  - Removes the last element of the vector
- `push_back(const T &val)`
  - Inserts a copy of `val` after the last element of the vector

Example

```
void GetValues(vector<int> &A) {
   A.resize(0);
   int Val;
   while (cin >> Val) {
       A.push_back(Val);
   }
}
```

```
vector<int> List;
cout << "Enter numbers: ";
GetValues(List);
```

Overloading >>

```
istream& operator>>(istream& sin, vector<int> &A) {
   A.resize(0);
   int Val;
   while (sin >> Val) {
       A.push_back(Val);
   }
   return sin;
}
```

```
vector<int> B;
cout << "Enter numbers: ";
cin >> B;
```

Vector Interface

- `reference front()`
  - Returns a reference to the first element of the vector
- `const_reference front() const`
  - Returns a constant reference to the first element of the vector

```
vector<int> B(4,1);  // B: 1 1 1 1
int& val = B.front();
val = 7;             // B: 7 1 1 1
```

Vector Interface

- **`reference back()`**
  - Returns a reference to the last element of the vector
- **`const_reference back() const`**
  - Returns a constant reference to the last element of the vector

```
vector<int> C(4,1);  // C: 1 1 1 1
int& val = C.back();
val = 5;             // C: 1 1 1 5
```

Iterators

- Iterator is a pointer to an element
  - Really pointer abstraction
- Mechanism for sequentially accessing the elements in the list
  - Alternative to subscripting
- There is an iterator type for each kind of vector list
- Notes
  - Algorithm component of STL uses iterators
  - Code using iterators rather than subscripting can often be reused by other objects using different container representations

Vector Interface

- `iterator begin()`
  - Returns an iterator that points to the first element of the vector
- `iterator end()`
  - Returns an iterator that points to immediately *beyond* the last element of the vector

```
vector<int> C(4);  // C: 0 0 0 0
C[0] = 0; C[1] = 1; C[2] = 2; C[3] = 3;
vector<int>::iterator p = C.begin();
vector<int>::iterator q = C.end();
```

Iterators

- To avoid unwieldy syntax programmers typically use typedef statements to create simple iterator type names

```
typedef vector<int>::iterator iterator;
typedef vector<int>::reverse_iterator reverse_iterator;
typedef vector<int>::const_reference const_reference;
```

```
vector<int> C(4);  // C: 0 0 0 0
iterator p = C.begin();
iterator q = C.end();
```

Iterator Operators

- `*` dereferencing operator
  - Produces a reference to the object to which the iterator `p` points

```
*p
```

- `++` point to next element in list
  - Iterator `p` now points to the element that followed the previous element to which p points

```
++p
```

- `--` point to previous element in list
  - Iterator `p` now points to the element that preceded the previous element to which p points

```
--p
```

```
typedef vector<int>::iterator iterator;
typedef vector<int>::reverse_iterator reverse_iterator;
vector<int> List(3);

List[0] = 100; List[1] = 101; List[0] = 102;

iterator p = List.begin();
cout << *p;                                  // 100
++p;
cout << *p;                                  // 101
--p;
cout << *p;                                  // 100
reverse_iterator q = List.rbegin();
cout << *q;                                  // 102
++q;
cout << *q;                                  // 101
--q;
cout << *q;                                  // 102
```

Vector Interface

- `insert(iterator pos, const T &val = T())`
  - Inserts a copy of `val` at position `pos` of the vector and returns the position of the copy into the vector
- `erase(iterator pos)`
  - Removes the element of the vector at position `pos`

Explicit Two-Dimensional List

- Consider definition

```
vector< vector<int> > A;
```

- Then

`A` is a **`vector< vector<int> >`**

  - It is a vector of vectors

`A[i]` is a **`vector<int>`**

  - **`i`** can vary from `0` to **`A.size() - 1`**

`A[i][j]` is a **`int`**

  - **`j`** can vary from `0` to **`A[i].size() - 1`**