

Containers

- ❖ Objects that hold other objects
 - E.g., stack, queue, vector, list
- ❖ Type(s) of stored objects
 - Homogeneous or heterogenous?
- ❖ Storage semantics
 - Value or reference?

ArrayInt Example

Auto-adjust array

- ❖ Acts as vector of int
- ❖ Homogeneous (all ints)
- ❖ Value semantics
 - accessed by [] (subscript operator)
- ❖ Note (if array implementation)
 - What about `p=&x[2]` if `realloc`?

List of Shapes

- ❖ Container for “drawing”
- ❖ Contains shapes
 - Of varying types (line, circle, etc.)
 - Store pointers (object size varies)
- ❖ Sequential access (drawing)

ShapeList

```
class ShapeList
{public:
    ShapeList(): count(0) { }
    ~ShapeList();
    void add(Shape* sp);
private:
    enum {size = 20};
    Shape* array[size];
    int count;
    friend class ShapeListIter;
};
```

ShapeList

Array of Shape pointers

```
class ShapeList
{public:
    ShapeList(): count(0) { }
    ~ShapeList();
    void add(Shape* sp);
private:
    enum {size = 20};
    Shape* array[size];
    int count;
    friend class ShapeListIter;
};
```

ShapeList

Constructor inits count

```
class ShapeList
{public:
    ShapeList(): count(0) { }
    ~ShapeList();
    void add(Shape* sp);
private:
    enum {size = 20};
    Shape* array[size];
    int count;
    friend class ShapeListIter;
};
```

CS481 - Object-Oriented Programming

ShapeList

Friend class (later)

```
class ShapeList
{public:
    ShapeList (): count(0) { }
    ~ShapeList ();
    void add (Shape* sp);
private:
    enum {size = 20};
    Shape* array[size];
    int count;
    friend class ShapeListIter;
};
```

ShapeList

```
void ShapeList::add(Shape* sp)
{
    ...
    assert(count < size);
    ...
    array[count++] = sp;
}
```

ShapeList

```
ShapeList::~~ShapeList()
{
    ...
    for (int ii=0; ii < count; ii++)
        delete array[ii];
}
```

Iterators

- ❖ Serves as pointer or index
 - Validity checks and “EOF”
- ❖ Provides access to contents
 - Of a specific container
 - ◆ Bound at iterator creation
- ❖ Can have multiple instances

ShapeListIter

```
class ShapeListIter
{
    ...
private:
    ShapeList& list;
    int index;
};
```

ShapeListIter

Reference to list object

```
class ShapeListIter
{
    ...
private:
    (ShapeList& list;
    int index;
};
```

CS481 - Object-Oriented Programming

ShapeListIter

Index into list

```
class ShapeListIter
{
...
private:
  ShapeList& list;
  int index;
};
```

ShapeListIter

```
class ShapeListIter
{public:
  ShapeListIter(ShapeList& sl)
    : list(sl), index(0) { }
  ShapeListIter
    (const ShapeListIter& sli)
    : list(sli.list),
      index(sli.index) { }
  void reset() { index = 0; }
...
}
```

ShapeListIter

Constructor inits ref & index

```
class ShapeListIter
{public:
  ShapeListIter(ShapeList& sl)
    : list(sl), index(0) { }
  ShapeListIter
    (const ShapeListIter& sli)
    : list(sli.list),
      index(sli.index) { }
  void reset() { index = 0; }
...
}
```

ShapeListIter

Copy constructor dupes iterator

```
class ShapeListIter
{public:
  ShapeListIter(ShapeList& sl)
    : list(sl), index(0) { }
  ShapeListIter
    (const ShapeListIter& sli)
    : list(sli.list),
      index(sli.index) { }
  void reset() { index = 0; }
...
}
```

ShapeListIter

Function resets index to beginning

```
class ShapeListIter
{public:
  ShapeListIter(ShapeList& sl)
    : list(sl), index(0) { }
  ShapeListIter
    (const ShapeListIter& sli)
    : list(sli.list),
      index(sli.index) { }
  void reset() { index = 0; }
...
}
```

ShapeListIter

```
ShapeListIter& operator++()
{ index++; return *this; }
ShapeListIter operator++(int)
{ ShapeListIter sli(*this);
  index++; return sli; }
const Shape* operator*()
{ return (index < list.count) ?
  list.array[index]: NULL; }
...
}
```

CS481 - Object-Oriented Programming

ShapeListIter

++op bumps index, returns iterator ref

```
ShapeListIter& operator++()
{ index++; return *this; }
ShapeListIter operator++(int)
{ ShapeListIter sli(*this);
  index++; return sli; }
const Shape* operator*()
{ return (index < list.count) ?
  list.array[index]: NULL; }
...
```

ShapeListIter

op++ copies iter, returns iterator object

```
ShapeListIter& operator++()
{ index++; return *this; }
ShapeListIter operator++(int)
{ ShapeListIter sli(*this);
  index++; return sli; }
const Shape* operator*()
{ return (index < list.count) ?
  list.array[index]: NULL; }
...
```

ShapeListIter

* returns pointer [index] or NULL

```
ShapeListIter& operator++()
{ index++; return *this; }
ShapeListIter operator++(int)
{ ShapeListIter sli(*this);
  index++; return sli; }
const Shape* operator*()
{ return (index < list.count) ?
  list.array[index]: NULL; }
...
```

Iterator Usage

```
ShapeList list;

list.add
  (new Circle(Point(1,3),2));
Shape* ptr;
ShapeListIter iter1(list);
while ((ptr = *iter1) != NULL)
{ ptr->draw(); iter1++; }
```

Iterator Usage

Declare list, add elements

```
ShapeList list;

list.add
  (new Circle(Point(1,3),2));
Shape* ptr;
ShapeListIter iter1(list);
while ((ptr = *iter1) != NULL)
{ ptr->draw(); iter1++; }
```

Iterator Usage

Create/initialize iterator

```
ShapeList list;

list.add
  (new Circle(Point(1,3),2));
Shape* ptr;
ShapeListIter iter1(list);
while ((ptr = *iter1) != NULL)
{ ptr->draw(); iter1++; }
```

Iterator Usage

Increment iterator and access elements

```
ShapeList list;

list.add
  (new Circle(Point(1,3),2));
Shape* ptr;
ShapeListIter iter1(list);
while ((ptr = *iter1) != NULL)
{ ptr->draw(); iter1++; }
```

Iterator Usage

Looks like pointer ("smart pointer"?)

```
ShapeList list;

list.add
  (new Circle(Point(1,3),2));
Shape* ptr;
ShapeListIter iter1(list);
while ((ptr = { *iter1 } != NULL)
{ ptr->draw(); iter1++; }
```

A Problem

- ❖ Common container functions
 - E.g., list, queue, vector, stack
- ❖ Different contents
 - E.g., int, double, user-defined
- ❖ How reuse container code?
 - Common base class vs template

Common Base Class

- ❖ As in Shape example
 - Other shapes are derived classes
- ❖ Container deals with base
 - Treats all content as base class objects
- ❖ Common base for all content?
 - Require multiple inheritance?

Templates

- ❖ Parameterized types
- ❖ Blueprint for class/function
 - Common generic description
 - Made specific by parameters
- ❖ Instantiated as needed
 - When compiler detects use

Template (Class)

```
template<class T>
class array
{
  enum { size=100 };

  T A[size];

public:
  T& operator[](unsigned int idx);
};
```

Template (Class)

```
template<class T>
class array
{
    enum { size=100 };

    T A[size];

public:
    T& operator[](unsigned int idx);
};
```

Declare template class with parameter(s)

Template (Class)

```
template<class T>
class array
{
    enum { size=100 };

    T A[size];

public:
    T& operator[](unsigned int idx);
};
```

"T" (any identifier) represents type parameter

Template (Class)

```
template<class T>
T& array<T>::operator[]
(unsigned int idx)
{
    assert(idx < size);
    return A[idx];
}
```

```
array<int> ia; array<float> fa;
ia[3]=27; fa[10]=ia[3]+2.5;
```

Template (Class)

```
template<class T>
T& array<T>::operator[]
(unsigned int idx)
{
    assert(idx < size);
    return A[idx];
}
```

```
array<int> ia; array<float> fa;
ia[3]=27; fa[10]=ia[3]+2.5;
```

Out-of-line member function definition

Template (Class)

```
template<class T>
T& array<T>::operator[]
(unsigned int idx)
{
    assert(idx < size);
    return A[idx];
}
```

```
array<int> ia; array<float> fa;
ia[3]=27; fa[10]=ia[3]+2.5;
```

Use template name with parameter as type name

Template Location

- ❖ In header file
 - All declarations and definitions
- ❖ Used when needed
 - Template instantiated when called for
- ❖ What about duplicate copies?
 - Merged by compiler or linker

Template Arguments

- ❖ Types `typename`
 - E.g., `template<class T>`
 - Note: T may be built-in type
- ❖ Compile-time constants
 - E.g., `template<class T, int size=100>`
 - Not commonly used? (constructor arg?)

Function Templates

- ❖ Generates function (global)
- ❖ Type appears in arg list
 - No implicit conversions

```
template<class T>
void swap(T& a, T& b)
{ T temp(a); a=b; b=temp; }
```

Template Bloat(?)

- ❖ Code generated for instance
 - Each different version of template
 - Multiple copies of similar code
- ❖ Common vs. Unique Feature
 - Base class for common functionality
 - Derived class for unique portions

Container Types

- ❖ Bag
- ❖ Set
- ❖ Vector
- ❖ Queue
- ❖ Deque
- ❖ Stack
- ❖ Ring
- ❖ List
- ❖ Map
- ❖ Tree (binary?)

STL

- ❖ Standard Template Library
 - Now part of C++ standard library
 - Defines containers, iterators, & "generic" algorithms
- ❖ Developed at H-P
 - Available via `ftp`

STL Generic Algorithm

```
#include <algorithm.h>

int data[100];

// code that adds elements to the data array

int *where; // sometimes int* where;

where = find(data, data+100, 14);

if (where != &data[100])
// then where points to 1st occurrence of 14
else // no 14 found in the data
```

CS481 - Object-Oriented Programming

STL Container

```
#include <algorithm.h>
#include <vector.h>

vector<float> nums;    // creates empty container
nums.push_back(3.0);
nums.push_back(7.0);
nums.push_back(14.0); // etc.

vector<float>::iterator where;
where = find(nums.begin(), nums.end(), 14.0);

if (where != nums.end())
    // then where "points to" the element containing 14
else // the element is not in the vector
```

Function Templates

- ❖ Generates function (global)
- Not a class

Implementation of find()

```
// part of the code in algorithm.h

template<class Iterator, class T>
Iterator find(Iterator first, Iterator last, T& value)
{
    while (first != last && *first != value)
        first++;

    return first;
}
```

Vector Interface

```
// part of the declaration of class 'vector'

template<class T>
class vector
{
public:
    typedef T* iterator;
    ...
    T front();
    T back();
    void push_back(T);
    void pop_back();
};
```