G64OOS (Spring 2014)

Lecture 06

OO Programming in C++ (2/2)

Peer-Olaf Siebers

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Motivation

- Get you prepared for the coursework :-)
  - Strengthen our knowledge about polymorphism
  - Introduce abstract classes and explain when to use them
  - Introduce you to some good coding practices
  - Provide you with an overview of the Standard Library (STL)

Polymorphism (Revision)

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Using Polymorphism and Derived Classes

- Polymorphism means that some code or operations or objects behave differently in different contexts
  - Inheritance means the Dog object inherits attributes and capabilities from the Mammal object
  - Polymorphism allows derived objects to be treated as if they where base objects
  - You can use polymorphism to declare a pointer to Mammal and assign to it the address of a Dog object you create on the heap
    - Mammal *pMammal=new Dog;
  - You can then use this pointer to invoke any member function on Mammal; the functions that are overridden in Dog will call the correct function if we use **virtual member functions**

Virtual Member Functions

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    ~Mammal(){cout<<"Mammal destructor\n";}

    void speak()const{cout<<"Mammal speak\n";}
    void sleep()const{cout<<"Mammal sleep\n";}
};

class Dog:public Mammal{
public:
    Dog(){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    void speak()const{cout<<"Dog speak\n";}
};

int main(){
    // Mammal pointer that points to a dog object
    Mammal *pDog=new Dog;
    pDog->sleep();
    pDog->speak();
    delete pDog;
    return 0;
}
```

- We have a Mammal pointer
  - speak() is not overridden
  - Dog destructor is not called

```
Mammal constructor
Dog constructor
Mammal sleep
Mammal speak
Mammal destructor
```

Virtual Member Functions

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    ~Mammal(){cout<<"Mammal destructor\n";}

    virtual void speak()const{cout<<"Mammal speak\n";}
    void sleep()const{cout<<"Mammal sleep\n";}
};

class Dog:public Mammal{
public:
    Dog(){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    void speak()const{cout<<"Dog speak\n";}
};

int main(){
    // Mammal pointer that points to a dog object
    Mammal *pDog=new Dog;
    pDog->sleep();
    pDog->speak();
    delete pDog;
    return 0;
}
```

- The keyword "virtual" signals that the derived class will probably want to override the virtual function

```
Mammal constructor
Dog constructor
Mammal sleep
Dog speak
Mammal destructor
```

Virtual Member Functions

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    virtual ~Mammal(){cout<<"Mammal destructor\n";}

    virtual void speak()const{cout<<"Mammal speak\n";}
    void sleep()const{cout<<"Mammal sleep\n";}
};

class Dog:public Mammal{
public:
    Dog(){cout<<"Dog constructor\n";}
    ~Dog(){cout<<"Dog destructor\n";}

    void speak()const{cout<<"Dog speak\n";}
};

int main(){
    // Mammal pointer that points to a dog object
    Mammal *pDog=new Dog;
    pDog->sleep();
    pDog->speak();
    delete pDog;
    return 0;
}
```

- If any member functions in your class are virtual then the destructor should also be virtual!
  - This will override the Mammal destructor with a Dog destructor
  - Otherwise you have a memory leak as only the type of object that the pointer is supposed to point to (in our case Mammal) will be deleted

```
Mammal constructor
Dog constructor
Mammal sleep
Dog speak
Dog destructor
Mammal destructor
```

Runtime Binding

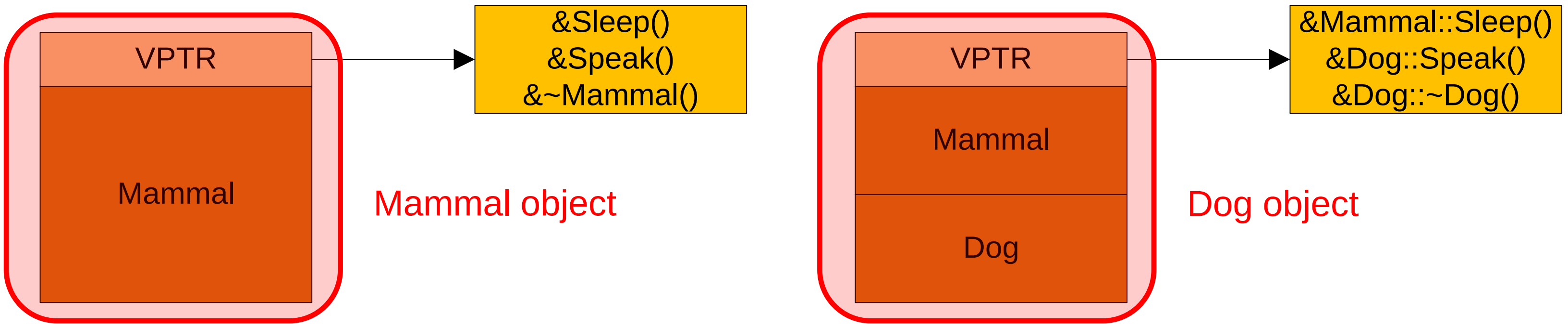
```cpp
#include <iostream>
using namespace std;

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    virtual ~Mammal(){cout<<"Mammal destructor\n";}
    virtual void speak()const{cout<<"Mammal speak\n";}
};

class Cat:public Mammal{
public:
    Cat(){cout<<"Cat constructor\n";}
    ~Cat(){cout<<"Cat destructor\n";}
    void speak()const{cout<<"Cat speak\n";}
};

int main(){
    int size=3;
    Mammal *array[size];
    Mammal *ptr;
    int choice;
    for(int i=0;i<size;i++){
        cout<<"1=cat; 2=mammal: ";
        cin>>choice;
        switch(choice){
            case 1: ptr=new Cat; break;
            default: ptr=new Mammal; break;
        }
        array[i]=ptr;
    }
    for(int i=0;i<size;i++){
        array[i]->speak();
        delete array[i];
    }
    return 0;
}
```

- It is impossible to know at compile time which object will be created and therefore which speak() method will be invoked
- The pointer "prt" is bound to its object at runtime; this is called **late binding** or **runtime binding**

```
1=cat; 2=mammal: 2
Mammal constructor
1=cat; 2=mammal: 1
Mammal constructor
Cat constructor
1=cat; 2=mammal: 2
Mammal constructor
Mammal speak
Mammal destructor
Cat speak
Cat destructor
Mammal destructor
Mammal speak
Mammal destructor
```

Runtime Binding

- How does late binding work?
  - When a virtual member function is created in an object the object must keep track of that member function
  - Compilers build a virtual function table (v-table) - one for each type and each object of that type keeps a v-table pointer (v-ptr)

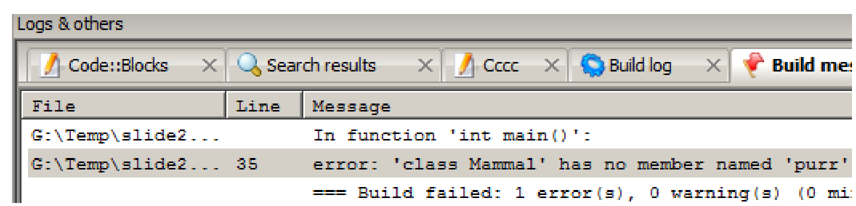


  - When the Dog constructor is called the v-ptr is adjusted to point to the virtual function overrides in the Dog object

Dynamic Casting

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    virtual ~Mammal(){cout<<"Mammal destructor\n";}
    virtual void speak()const{cout<<"Mammal speak\n";}
};

class Cat:public Mammal{
public:
    Cat(){cout<<"Cat constructor\n";}
    ~Cat(){cout<<"Cat destructor\n";}
    void speak()const{cout<<"Cat speak\n";}
    void purr()const{cout<<"Cat purrs\n";}
};

int main(){
    int size=3;
    Mammal *array[size];
    Mammal *ptr;
    int choice;
    for(int i=0;i<size;i++){
        cout<<"1=cat; 2=mammal: ";
        cin>>choice;
        switch(choice){
            case 1: ptr=new Cat; break;
            default: ptr=new Mammal; break;
        }
        array[i]=ptr;
    }
    for(int i=0;i<size;i++){
        array[i]->speak();
        array[i]->purr();
        delete array[i];
    }
    return 0;
}
```

- What happens if you want to add a member function to Cat that is inappropriate for Mammal?
- Calling "purr()" using your pointer to Mammal will produce a compiler error



Dynamic Casting

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

class Mammal{
public:
    Mammal(){cout<<"Mammal constructor\n";}
    virtual ~Mammal(){cout<<"Mammal destructor\n";}
    virtual void speak()const{cout<<"Mammal speak\n";}
};

class Cat:public Mammal{
public:
    Cat(){cout<<"Cat constructor\n";}
    ~Cat(){cout<<"Cat destructor\n";}
    void speak()const{cout<<"Cat speak\n";}
    void purr()const{cout<<"Cat purrs\n";}
};

int main(){
    int size=3;
    Mammal *array[size];
    Mammal *ptr;
    int choice;
    for(int i=0;i<size;i++){
        cout<<"1=cat; 2=mammal: ";
        cin>>choice;
        switch(choice){
            case 1: ptr=new Cat; break;
            default: ptr=new Mammal; break;
        }
        array[i]=ptr;
    }
    for(int i=0;i<size;i++){
        array[i]->speak();
        Cat *pRealCat=dynamic_cast<Cat *>(array[i]);
        if(pRealCat) pRealCat->purr();
        delete array[i];
    }
    return 0;
}
```

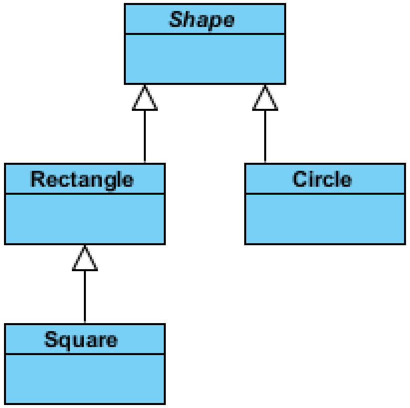
- Solution: Cast your base class pointer to your derived type
  - Using the "dynamic_cast" operator ensures that when you cast, you cast safely; base pointer is examined at runtime; if conversion is proper your new cat pointer is fine, else your new cat pointer will be pointing to "null"

```
1=cat; 2=mammal: 2
Mammal constructor
1=cat; 2=mammal: 1
Mammal constructor
Cat constructor
1=cat; 2=mammal: 2
Mammal constructor
Mammal speak
Mammal destructor
Cat speak
Cat purrs
Cat destructor
Mammal destructor
Mammal speak
Mammal destructor
```

Abstract Data Types (Abstract Classes)

Abstract Data Types

- Abstract Data Type (ADT) = Abstract Classes
  - Represents a concept rather than an object
    - Shape represents a concept
    - Rectangle, circle and square represent objects
  - Exists only to provide an interface for the classes derived from it
  - It is not valid to create an instance of an ADT

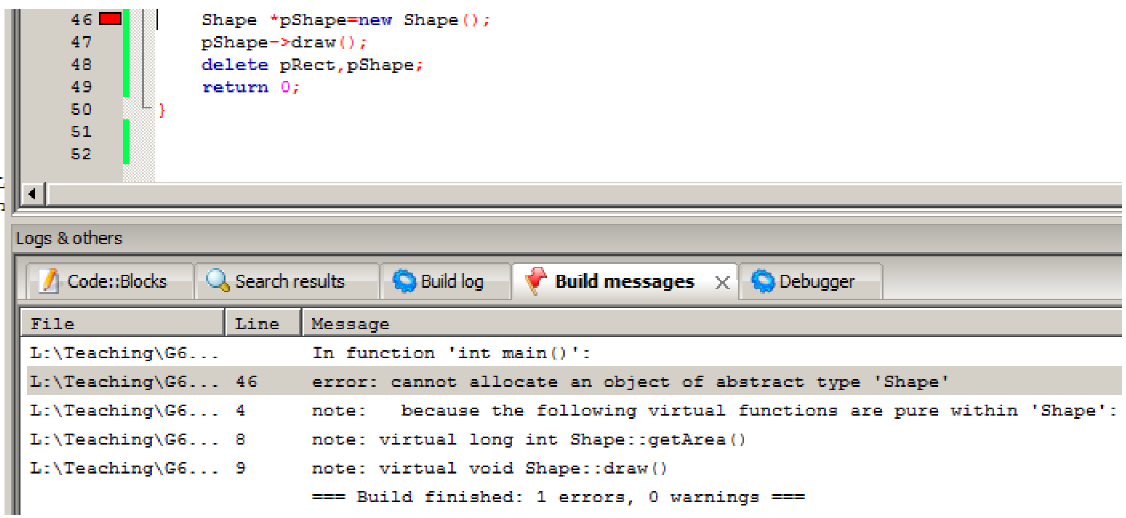


Abstract Data Types

- ADTs can be created by using **pure virtual functions** (virtual functions that **must** be overridden in the derived class)
  - virtual void draw()=0;
- Any class with one or more pure virtual functions is an ADT and cannot be instantiated
- Any class that derives from an ADT inherits the pure virtual functions and **must override** them if it wants to instantiate objects

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

class Shape{
public:
    Shape(){}
    virtual ~Shape(){}
    virtual long getArea()=0;
    virtual void draw()=0;
};

class Rectangle:public Shape{
protected:
    int width;
    int length;
public:
    Rectangle(int newLen,int newWidth):length(newLen),width(newWidth){}
    virtual ~Rectangle(){}
    virtual long getArea(){return length*width;}
    virtual void draw();
};

void Rectangle::draw(){
    for(int i=0;i<length;i++){
        for(int j=0;j<width;j++){
            cout<<"x";
        }
        cout<<"\n";
    }
    cout<<"\n";
}

class Square:public Rectangle{
public:
    Square(int newLength);
    Square(int newLength, int newWidth);
    ~Square(){}
};

Square::Square(int newLength):Rectangle(newLength,newL
Square::Square(int newLength,int newWidth):Rectangle(n

int main(){
    Shape *pRect=new Rectangle(4,6);
    pRect->draw();
    Shape *pShape=new Shape();
    pShape->draw();
    delete pRect,pShape;
    return 0;
}
```

- The pure virtual functions in an ADT are never implemented (although technically it is possible)
- Rectangle must override **both** pure virtual functions or it will also be an ADT



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

class Shape{
public:
    Shape(){}
    virtual ~Shape(){}
    virtual long getArea()=0;
    virtual void draw()=0;
};

class Rectangle:public Shape{
protected:
    int width;
    int length;
public:
    Rectangle(int newLen,int newWidth):length(newLen),width(newWidth){}
    virtual ~Rectangle(){}
    virtual long getArea(){return length*width;}
    virtual void draw();
};

void Rectangle::draw(){
    for(int i=0;i<length;i++){
        for(int j=0;j<width;j++){
            cout<<"x";
        }
        cout<<"\n";
    }
    cout<<"\n";
}

class Square:public Rectangle{
public:
    Square(int newLength);
    Square(int newLength, int newWidth);
    ~Square(){}
};

Square::Square(int newLength):Rectangle(newLength,newLength){}
Square::Square(int newLength,int newWidth):Rectangle(newLength,newWidth){if(length!=width){cout<<"Error --> not a square!\n";}}

int main(){
    Shape *pRect=new Rectangle(4,6);
    pRect->draw();
    Shape *pSquare=new Square(4,6);
    pSquare->draw();
    delete pRect,pSquare;
    return 0;
}
```

```
xxxxxx
xxxxxx
xxxxxx
xxxxxx

Error --> not a square!
xxxxxx
xxxxxx
xxxxxx
xxxxxx
```

Good Coding Practice

Good Coding Practices

- File organisation
  - Header file(s)
    - Contains the data members and prototypes
  - Implementation file(s)
    - Contains the implementation
  - Main program file
    - Only contains a trigger for the program to kick off

- Good coding practice
  - Have one .cpp file for every .h (or .hpp) file
  - Group multiple logically related classes in one set of .h/.cpp files

Good Coding Practice

Shape.h

```
#include <iostream>

using namespace std;

/* Class definitions */
class Shape
{
    protected:
    int centre[2];

    public:
    // -- Constructor with Centre given, c must be a 2-element array
    Shape(int c[]);

    void move(int d[]);
    virtual float area() = 0;
    virtual float circumference() = 0;
    virtual int max_x() = 0;
};

class Rectangle: public Shape
{
    int height;
    int width;

    public:
    // -- Constructor with Centre and side given
    Rectangle(int c[], int h, int w);

    float area();
    float circumference();
    int max_x();
};
```

Definition of Abstract Base Class (ABC)

Definition of concrete Subclass

Constructor

Definition of polymorph functions that were pure virtual in ABC

Good Coding Practice

Shape.cpp

```
#include <iostream>
#include "Shape.h"

using namespace std;
/* Class implementations */
Shape::Shape(int c[])
{
    centre[0] = c[0];
    centre[1] = c[1];
}

void Shape::move(int d[])
{
    centre[0] += d[0];
    centre[1] += d[1];
}

// -- Rectangle constructor
Rectangle::Rectangle(int c[], int h, int w): Shape(c), height(h), width(w) {}

void Rectangle::print() const
{
    for (int i=0; i<height; i++){
        for (int j=0; j<width; j++)
            cout << "#";
        cout << endl;
    }
}

float Rectangle::area() const
{
    return height*width;
}

float Rectangle::circumference() const
{
    return 2*(height + width);
}

int Rectangle::max_x() const
{
    return centre[0] + width/2;
}
```

Include Class definitions!!!

Rectangle implementation

Polymorph functions implementation

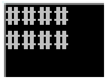
Good Coding Practice

TestShape.cpp

```
#include <iostream>
#include "Shape.h"

using namespace std;

int main()
{
    int L0[2] = {0,0};
    Rectangle *R = new Rectangle(L0, 2, 4);

    R->print();

    // -- Clean up after yourself
    delete R;
}
```

Include Shape header again

Create a Rectangle pointer



Good Coding Practice

- Whitespace
- Comments
- Const correctness
- Symmetry

Good Coding Practice

- Whitespace

```
// -- Bad use of whitespace
void Rectangle::printCircumferences() const
{for (int r=0; r<2; r++) {for (int i=0; i<height; i++) cout << "-";
cout << "#"; for (int j=0; j<width; j++) cout << "-"; cout << "#";}}
```

```
// -- Proper use of whitespace
void Rectangle::printCircumferences() const
{
    for (int r=0; r<2; r++) {
        for (int i=0; i<height; i++)
            cout << "-";
        cout << "#";

        for (int j=0; j<width; j++)
            cout << "-";
        cout << "#";
    }
}
```

```
// -- Very bad use of whitespace
void Rectangle::printCircumferences() const {for (int r=0; r<2; r++) {for (int i=0; i<height; i++) cout << "-"; cout << "#"; for (int j=0; j<width; j++) cout << "-"; cout << "#";}}
```

Good Coding Practice

- Comments
  - Should state why something is done, not what is done
  - Could be automatically harvested to generate documentation
    - Doxygen/Doxys and SandCastle

Good Coding Practice

- Const correctness
  - The const keyword allows you to specify whether or not a variable is modifiable; you can use const to prevent modifications to variables and const pointers and const references to prevent changing the data pointed to (or referenced)
  - The primary purpose of constness is to provide documentation and prevent programming mistakes; const allows you to make it clear to yourself and others that something should not be changed
  - Can be completely designed during header file construction

  - For more advice see
    - http://www.cprogramming.com/tutorial/const_correctness.html

Good Coding Practice

- Const correctness
  - Const variables
    - "int const x = 5;" is the same as "const int x = 4;“
  - Const pointers
    - "const int *p_int;“
      - The pointer may be changeable but you can't touch what p_int points to
    - "int x; int * const p_int = &x;"
      - The address pointed to cannot be changed; therefore the pointer has to be initialised when it is declared
  - Const functions
    - "int Loan::calcInterest() const {return loan_value * interest_rate;}"
      - Guarantees that the function will not change the object; the function itself can still be used by non-const objects

Good Coding Practice

- Symmetry
  - Symmetry is good practice in many situations, e.g. report writing, or when presenting good coding practices in a lecture

```cpp
// -- Bad symmetry
class Library
{
  vector<Book> books;
  vector<CD> cds;
  vector<Shelf> shelfs;
  bool open;
  bool staffed;

  public:
  bool isStaffed():
  bool checkIfLibraryOpen();

  vector<Book> getBooks();
  vector<Shelf> returnShelfs();
  vector<CD> libraryCDs();

};
```

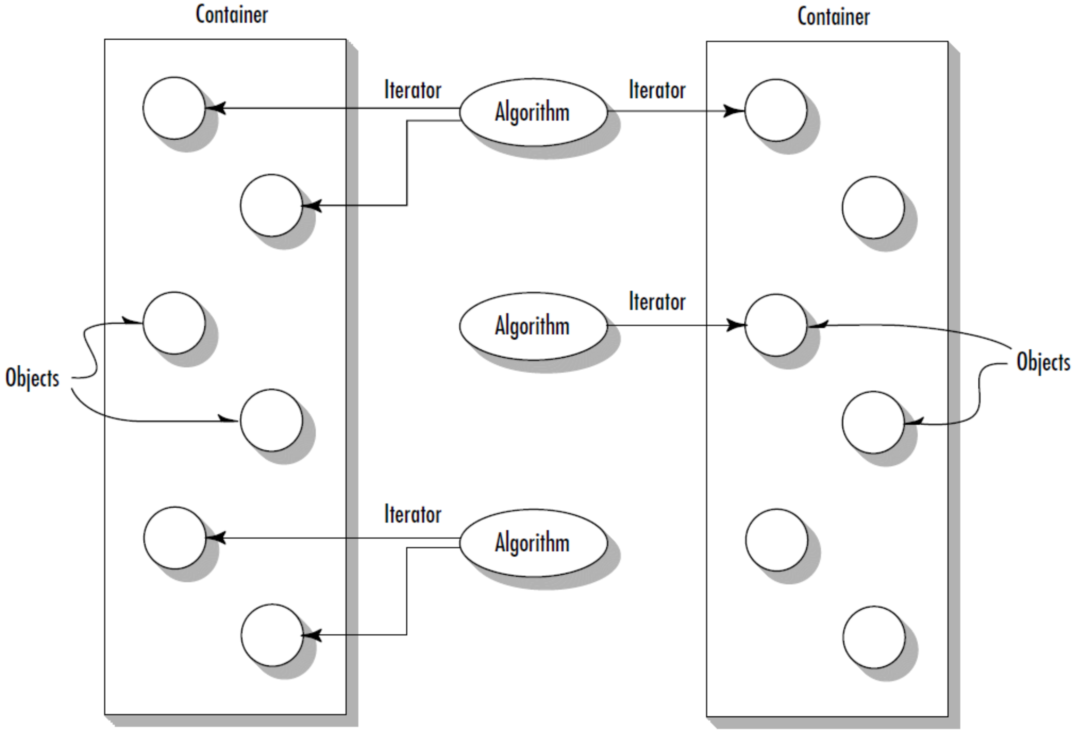
```cpp
// -- Good symmetry
class Library
{
  vector<Book> books;
  vector<CD> cds;
  vector<Shelf> shelfs;

  bool open;
  bool staffed;

  public:
  vector<Book>  getBooks();
  vector<CD>    getCDs();
  vector<Shelf> getShelfs();

  bool isOpen();
  bool isStaffed():
};
```

Whitespace used to improve symmetry

Standard Template Library

Standard Template Library (STL)

- STL is a collection of classes that provides
  - Template Containers
  - Iterators
  - Algorithms



Algorithms use iterators to act on objects in containers

Containers

- A container is a way to store data - whether the data consists of build-in types or of class objects
- A container usually include functions for
  - Creating an empty container
  - Insert a new object into the container
  - Remove an object from the container
  - Report the current number of objects in the container
  - Empty the container
  - Provide access to the stored objects
  - Sort the elements (optional)

Containers

- Three basic categories
  - Sequence containers (vector; deque; list)
    - Maintain the ordering of elements inside the container; you can chose the position of the element you insert
  - Associative containers (set; multiset; map; multimap)
    - Automatically sort their input when inserted into the container
  - Container adaptors (stack; queue; priority queue)
    - Predefined containers that are adapted for specific use

Sequence Containers

- Vector
  - Dynamic array; allows random access to elements; removing or inserting elements from the end of vector is generally fast
- Deque
  - Double ended queue class; implemented as dynamic array that can grow from both ends
- List
  - Each element in the container contains pointers that point at the next and previous element in the list
  - Inserting elements in a list is very fast if you know where you want to insert them

Associative containers

- Set
  - Stores unique elements only
  - Elements are sorted according to their value
- Multiset
  - A set that allows duplicate elements
- Map (associative array)
  - Set where each element is a key/value pair
  - Key is used for sorting and indexing the data
- Multimap (dictionary)
  - A map that allows duplicate keys
  - Some words can have multiple meanings

Container Adaptors

- Stack
  - Elements operate in a FILO context
  - Use deque as default container but can also use vector or list
- Queue
  - Elements operate in a FIFO context
  - Use deque as default container but can also use list
- Priority queue
  - Queue where elements are kept sorted
  - Removing an element from the front returns the highest priority item in the priority queue

Containers

- Some useful member functions
  - push_front() push_back(): Inserts a new element at the beginning/end of the container effectively increasing the container size by one .
  - pop_front() pop_back(): Removes first/last element of container, effectively reducing the container size by one and invalidating all iterators and references to it

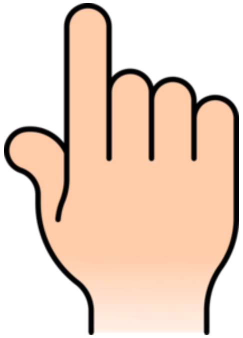
- The vector and deque containers provide
  - [] Subscripting access without bounds checking (vect[…])
  - at Subscripting access with bounds checking (vect.at(…))

Containers

- Some useful member functions
  - empty	Boolean indicating if the container is empty
  - size	Returns the number of elements
  - insert	Inserts an element at a particular position
  - erase	Removes an element at a particular position
  - clear	Removes all the elements
  - resize	Resizes the container
  - front	Returns a reference to the first element
  - back	Returns a reference to the last element

Iterators

- What?
  - Objects that can iterate over a container class without the programmer having to know how the container class is implemented
  - Iterators make it easy to step through each element of a container without having to understand how the container class is implemented
- How?
  - An iterator is a pointer to a given element in a container with a set of overloaded operators to provide a set of well-defined functions



Iterators

- Operators
  - "*": Dereferencing the iterator (returns the element that the iterator is currently pointing at)
  - "++": Moves the iterator to the next element in the container (most iterators also provide "--" to move to previous element)
  - "=="; "!=": Basic comparison operators to determine if two iterators point to the same element (to compare the values that two iterators are pointing at iterators need to be dereferenced first)
  - "=": Assign the iterator to a new position (typically the start or end of the container's elements)

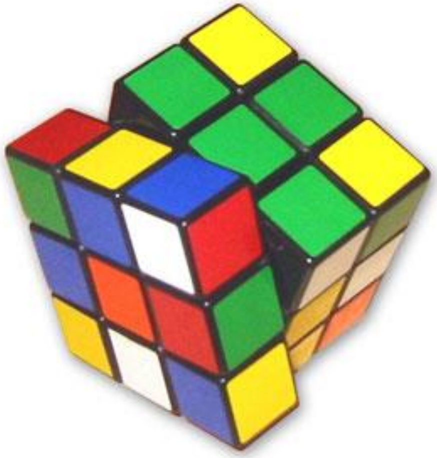
Iterators

- Each container includes four basic functions for use with "="
  - begin() returns iterator representing the beginning of elements in the container; cbegin() returns const iterator
  - end() returns iterator representing the element just past the end of elements; cend() returns const iterator

- All containers provide (at least) two types of iterators
  - "container::iterator" provides a read/write iterator
    - for(vector<int>::iterator i=rData.begin();i!=rData.end() ; ++i)*i=0;
  - "container::const_iterator" provides a read-only iterator
    - for(vector<int>::const_iterator i=rData.begin(); i!=rData.end();++i)cout<<*i;

Algorithms

- An algorithm is a function that does something to the items in a container (or containers)
- Algorithms are stand-alone template functions (global functions that operate using iterators)
- You can use algorithms with built-in C++ arrays or with container classes



Algorithms

- Small choice of algorithms
  - find — Returns first element equivalent to a specified value
  - count — Counts the number of elements that have a specified value
  - equal — Compares the contents of two containers and returns true if all corresponding elements are equal
  - search — Looks for a sequence of values in one container that corresponds with the same sequence in another container
  - copy — Copies a sequence of values from one container to another (or to a different location in the same container)
  - swap — Exchanges a value in one location with a value in another
  - fill — Copies a value into a sequence of locations
  - sort — Sorts the values according to a specified ordering
  - for_each — Executes a specified function for each container element

Example: Vector & Iterator & Algorithm

- Example: Vector & Algorithm

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

int main(){
    vector<int> vect;
    vect.push_back(7);
    vect.push_back(-3);
    vect.push_back(6);
    vect.push_back(2);
    vect.push_back(-5);
    vect.push_back(0);
    vect.push_back(4);

    sort(vect.begin(), vect.end());
    vector<int>::const_iterator it;
    for(it=vect.begin();it!= vect.end();it++)
        cout << *it << " ";
    cout << endl;
    reverse(vect.begin(),vect.end());
    for(it=vect.begin();it!= vect.end();it++)
        cout << *it << " ";
    cout << endl;
}
```

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Summary

- What did you learn?

References

- Slides are based on
  - Sams Teach Yourself C++ in 24 Hours (chapter 17-18)