Lecture 8

- Abstract classes
  - Pure virtual functions
- Virtual destructors
- Types of inheritance
  - `private`, `protected`, `public`
- Template classes
- The **Standard Template Library (STL)**
  - Containers
    - `vector`, `map`, `multimap`
  - Iterators
  - Algorithms

Abstract Classes

- Recall class **Shape**
  - It should be possible to draw a shape, but
  - Class **Shape** is too general to know how to draw an abstract shape
- **Solution**: define **Shape** as an **abstract class**
  - Abstract classes have at least one **pure virtual function**

```cpp
virtual void Shape::draw() const = 0;
```

- Pure virtual functions in an abstract class have no implementation
Abstract Classes

• Cannot create objects of an abstract class
  
  \[\text{Shape } s; \quad \text{// WRONG!!!}\]

• Can declare pointers for an abstract class
  
  \[\text{Square } sq;\]
  \[\text{Shape* } p = & \text{ sq;}\]

• Useful for dynamic binding
  
  - \text{Circle, Triangle, Rectangle} derived from \text{Shape}
    
    • Each must implement \text{draw} function

Virtual \textit{versus} Pure Virtual

• Regular virtual functions
  
  \[\text{virtual void Shape::draw()} \text{ const};\]
  
  • Must be implemented
  
  • Derived classes can override its implementation

• Pure virtual functions
  
  \[\text{virtual void draw()} \text{ const = 0;}\]
  
  • No implementation
  
  • Derived classes must provide an implementation

• Abstract classes can have data and concrete functions
  
  • Required to have one or more pure virtual functions
Abstract Classes *versus*
Concrete classes

• Abstract classes
  – Purpose: to be a base class (called abstract base classes)
  – Incomplete
    • Derived classes fill in "missing pieces"
  – Cannot make objects from abstract class
    • However, can have pointers and references

• Concrete classes
  – Can instantiate objects
  – Implement all functions they define

Virtual Destructors

• Base class pointer to derived object
  – Recall classes *Vehicle*, *Car*, and *Volvo*

  ```cpp
  Vehicle *Ptr = new Volvo("V50","Alpine",250,6);
  delete Ptr;
  ```

• Problem:
  – Destructor of *Vehicle* is called
  – The choice of destructor to call depends on the type of the pointer
    • The type of the object being pointed is not taken into account

• Solution: declare the destructor in *Vehicle* as virtual
Virtual Destructors

```cpp
virtual ~Vehicle();
```

- Declare base-class destructor virtual
  - Makes derived-class destructors virtual
  - Now, when `delete` used appropriate destructor called
    1. Derived-class destructor executes first
    2. Base-class destructor executes afterwards
      1. Destructor for `Volvo` called
      2. Destructor for `Car` called
      3. Destructor for `Vehicle` called

- Constructors cannot be virtual

Types of Inheritance + Access Specifiers

```
<table>
<thead>
<tr>
<th>Base-class Member</th>
<th>Access Specifier</th>
<th>Types of Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>Private</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Protected</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Public</td>
<td>x</td>
</tr>
</tbody>
</table>
```

Composition

class Engine {
    public:
        Engine(int numCylinders);
        // Starts this Engine
        void start();
    private:
        int cylinders;
};

class Car {
    public:
        Car() : e(8) { }
        void start() { e.start(); }
    private:
        Engine e;
};

Composition through Private Inheritance

class Engine {
    public:
        Engine(int numCylinders = 16)
            {cylinders = numCylinders;}
        void start() // Starts this Engine
            { cout <<" Vruuuuum...!" << endl; }
    private:
        int cylinders;
};

class Car : private Engine { // Car has-a Engine
    public:
        Car() : Engine(8) { }
        using Engine::start;
        void start() {Engine::start();}
};
Class Templates

• Class templates
  • Specify entire range of related classes
  • Generic programming
    – Class-template specializations
• Example: Stack  See stack1.h
  • LIFO (last-in-first-out) structure
  • Describe notion of stack generically
  • Instantiate type-specific version

```cpp
template< class T >
Class Stack {
  ...
}
```

```cpp
Stack< double > doubleStack(5);
doubleStack.push(5.6);
```

See Fig11_03.cpp
Tstack1.h
Class Templates and Nontype Parameters

```cpp
template< class T = String, int elements = 50 >
class Stack {
    private:
    int size;
    int top;
    T stackHolder[elements];

    public:
    ...
} //end class stack
```

```cpp
Stack< double, 100 > myStack;
Stack< > yourStack; //accept default template arguments
```

Think: What is a advantage of this solution?

---

Standard Template Library

- Powerful, template-based components
  - **Containers**: template data structures
    - Objects that contain other objects
  - **Iterators**: like “pointers”, cycle through the contents of a container
  - **Algorithms**: manipulate containers, searching, sorting, etc.

- Code reuse
  - Templates are used

- Performance
  - STL specification requires that containers and algorithms be implemented in a way that ensures optimal run time performance.
Containers

• Several types of containers
  – **Sequence containers**
    • Linear data structures
      – Dynamic array  
      – Linear list
  – **Associative containers**
    • Store key/value pairs
      – Map
  – **Container adapters**

A container automatically grows when elements are added and shrinks when elements are removed.

Containers

• Sequence containers
  – **vector** A dynamic array
  – **deque** A double-ended queue
  – **list** A linear list

• Associative containers
  – **set** no duplicates
  – **multiset** duplicates allowed
  – **map** Associates one value for each key
  – **multimap** Associates multiple values for each key

• Container adapters
  – **stack**
  – **queue**
    template <class T, class Container = deque<T>>
    class queue
  – **priority_queue**
**vector** Sequence Container

- **vector**
  
  // vector class-template definition
  #include <vector>

  – Data structure with contiguous memory locations
  • Access elements randomly with []
    – Access ( v[i] ) in constant time

- When memory exhausted
  – Allocates larger, contiguous area of memory

- **Declarations**

  vector<int> v1; // create a zero-length int vector
  vector<Point> v2(20);

- **Declarations**

  vector<char> v3(5, 'x');
  vector<int> v4(v1);
  // create a vector from an array a
  int a[10] = {0};
  vector<int> v5(a, a+10);

- **Subscripting []** (in constant time)

  v3[6] = 'b'; // Error!!
  – Access out of bounds (silent error)
  – Will not increase vector size

- **Expanding the vector**

  v1.push_back(8); // memory is allocated, if needed
  cout << v1.size(); // display number of elements stored

- **Remove last element**

  v1.pop_back();
Vector Example

```cpp
int main()
{
    vector<string> SS;
    SS.push_back("The number is 10");
    SS.push_back("The number is 20");
    SS.push_back("The number is 30");
    cout << "Loop by index:"
         << endl;
    for(int ii = 0; ii < SS.size(); ii++)
    {
        cout << SS[ii] << endl;
    }
    return 0;
}
```

Iterators

- Iterators similar to pointers
  ```cpp
  vector<int>::iterator it1;
  // Analogous to int*
  vector<int>::const_iterator it2;
  // Analogous to const int*
  ```
  - Operations supported
    ```cpp
    *it1 dereferences
    it1++ points to next element
    it2 = it1;
    it1 = it2; // compilation ERROR
    ```
Iterators

• Initialization
  – Create an iterator given a container \( c \)
    
    \[
    \text{vector<int> } c;
    \]
    
    • \( \text{c.begin()} \) returns iterator to first element
    • \( \text{c.end()} \) returns iterator to after last element
  
  – Container adapters do not support iterators

Iterators

• Iterating over a vector with an iterator

\[
\text{vector<int> } v;
\]

\[
\text{for (vector<int>::iterator it = v.begin(); it! = v.end(); ++it)}
\]

\[
\{ 
\text{cout << *it << endl;}
\}
\]
Iterators

```cpp
vector<char> v(10);
for(int i = 0; i < 10; i++)
v[i] = i + 'a';

vector<char>::iterator p = v.begin();
p += 2; // point to 3rd element
v.insert(p, 10, 'X');
```

- Inserts 10 copies of ‘X’ before the element pointed by p
- Elements after p are shifted
- Not available for arrays in C/C++

---

Const and Reverse Iterators

- **Reverse iterators**
  ```cpp
  vector<type>::reverse_iterator iterVar;
  ```
  - Visits elements in reverse order (end to beginning)
  ```cpp
  iterVar++; // walks backwards
  ```
  - Use `c.rbegin()` to get starting point
  - Use `c.rend()` to get ending point

- **Const iterators**
  ```cpp
  vector<type>::const_iterator iter;
  vector<type>::const_reverse_iterator iter;
  ```
  - `const_iterator` cannot modify elements “pointed” by the iterator
Iterators

```cpp
int main(){
    vector<string> SS;
    //add elements to SS
    vector<string>::const_iterator cii;
    for(cii=SS.begin(); cii!=SS.end(); cii++)
    {
        cout << *cii << endl;
    }
    //Reverse Iterator
    vector<string>::reverse_iterator rii;
    for(rii=SS.rbegin() ; rii!=SS.rend() ; ++rii)
    {
        cout << *rii << endl;
    }
    return 0;
}
```

vector Sequence Container

- **vector** functions
  - `v.push_back(value)`
    - Add element to end (found in all sequence containers).
  - `v.size()`
    - Current number of elements stored in the vector
  - `v.capacity()`
    - How much vector can hold before reallocating memory
  - `vector<type> v(a, a + SIZE)`
    ```cpp
    int a[10] = {0};
    vector<int> v5(a, a+10);
    ```
    - Creates `vector v` with elements from array `a` up to (not including) `a + SIZE`
vector Sequence Container

**vector functions**

- `v.insert(iterator, value)`
  - Inserts `value` before location of `iterator`
- `v.insert(iterator, array, array + SIZE)`
  - Inserts array elements (up to, but not including `array + SIZE`) into vector before `iterator`
- `v.erase(iterator)`
  - Remove element pointed by `iterator` from container
- `v.erase(iter1, iter2)`
  - Remove elements starting from `iter1` and up to (not including) `iter2`
- `v.clear()`
  - Erases entire container

See Fig21_14.cpp
Algorithms

- **STL** has algorithms used generically across containers
  - Operate on elements indirectly via iterators
    - `#include <algorithm>`
  - Often operate on a range of elements
    - Defined by pairs of iterators
  - Algorithms often return iterators
    - `find()`
      - Returns iterator to element, or `end()` if not found
  - Pre-made algorithms save programmers time and effort

### Algorithms Example

```cpp
bool even(int n) { return (n % 2? false: true); }

int main()
{
    int array[6] = { 1, 2, 3, 4, 5, 6 };
    vector<int> b(array, array+6);
    int c = count_if(b.begin(), b.end(), even);
    cout << "Number of evens: " << c << endl;
    vector<int> x(5, 0);
    // It only copies values 1,2,3,4
    copy(b.begin(), b.begin()+5, ++x.begin());
    reverse(x.begin(), x.end());
    vector<int>::iterator ptr;
    for(ptr = x.begin(); ptr != x.end(); ++ptr )
        cout << *ptr << ' ';
    // Output looks like: 4 3 2 1 0
}
```
Types of Algorithms

- Copying
- Searching unsorted sequences
- Replacing and removing elements
- Reordering a sequence
- Sorting
- Sorted Sequence Searching
- Merging sorted sequences
- Set operations
- Heap operations
- Minimum and maximum
- Permutations
- Miscellaneous

Find Algorithms

```cpp
int main() {
    int array[6] = { 1, 2, 3, 4, 5, 6 };  
    vector<int> b = (array, array+6); 
    vector<int>::iterator ptr; 
    ptr = find(b.begin(), b.end(), 10); 
    if (ptr != b.end()) 
        cout << *ptr << endl; 
    //find first even number 
    ptr = find_if(b.begin(), b.end(), even); 
    if (ptr != b.end()) 
        cout << *ptr << endl; 
    //find second even number 
    ptr = find_if(++ptr, b.end(), even); 
    if (ptr != b.end()) 
        cout << *ptr << endl; 
    return 0;
}
```
Numeric Algorithm

```cpp
text
int main() {
    int array[6] = { 1, 2, 3, 4, 5, 6 };  
    vector<int> b = (array, array+6);  
    vector<int> c(6);  
    //sum with predecessor
    adjacent_difference(b.begin(), b.end(),  
                        c.begin(), plus<int>());  
    for(vector<int>::iterator ptr = c.begin();  
        ptr != c.end(); ++ptr ) 
        cout << *ptr << ' ';  
    return 0;  
}
```

Algorithms

- In general, algorithms do not automatically increase the size of the target container
  - c should be large enough to hold the result

```cpp
text
int array[6] = { 1, 2, 3, 4, 5, 6 };  
vector<int> b = (array, array+6);  
vector<int> c;  
//sum with predecessor
adjacent_difference(b.begin(), b.end(),  
                    c.begin(), plus<int>()); 
```

Run time error!!
map container

- Associative container in which keys are mapped with values
  - A stored valued can be retrieved by its key
    - Key: person name
    - Value: telephone number
    - `operator[]` can be used
  - `multimap` container allows non unique keys
    - Key: City name
    - Value: list of clients who live in the city
    - `operator[]` cannot be used
  - Maps and `multimaps` are sorted by key
    - Pairs with same key are stored contiguously in a `multimap`
- `#include <map>`

```cpp
map<string, char> grades;
//if "John" is not in the map then it is inserted
grades["John"] = 'A';

map<string, char> iterator itr = grades.begin();
//Acess to the key
itr->first
//Acess to the value
itr->second

map<string, vector<int>> exams;
//add another 5 points for "Anna"
exams["Anna"].push_back(5);
```

Give a space, otherwise the compiler gives an error (confuses with the output operator `>>`)
**map Example**

```
typedef map <string, char> myList;
myList grade_list;
grade_list["John"] = 'A';
if(grade_list.find("Tim") == grade_list.end())
    cout<<"Tim is not in the map!"<<endl;
    // Should be John
    cout<<grade_list.begin()->first<< endl;
    // Should be A
    cout<<grade_list.begin()->second<< endl;

    grade_list.insert(myList::value_type("David", 'B'));
    grade_list.insert(make_pair("David", 'B'));
```

**multimap**

- **Insert**
  ```
  multimap<string, int> exams;
  exams.insert(make_pair("Anna", 5));
  ```

- **Find**
  ```
  multimap<string, int>::const_iterator itr;
  itr = exams.find("Anna");
  cout << itr->second << endl; //value
  ```
multimap

multimap<string, int>::const_iterator it1, it2;
int k = 0;
double sum = 0;
//it1 points to the first element with key "Anna"
it1 = exams.find("Anna");
//it2 points to the first element with
//key larger than "Anna"
it2 = movies.upper_bound("Anna");
do {
    ++k;
    sum += it1->second;
    it1++;
} while (it1 != it2);
double avg = sum / k;

Compute the average grade of Anna