A Whirlwind Tour of C++

Robert P. Goddard
Applied Physics Laboratory
University of Washington

Part 1: 3 November 2000
Part 2: 17 November 2000
Part 3: 15 December 2000
Introduction

- Scope: ISO/ANSI Standard C++ language and standard library
- What is in it, what isn’t, and why?
- Emphasis is on objectives of language, and compromises made to achieve them.
- Audience: Programmers. Some familiarity with C, and concepts like object, class, and type will be helpful.
- Fast and shallow
Agenda

- Overview of language features
- Overview of standard library
- Objectives of the language
- Classes and objects
- Templates
- Standard Template Library
- Exceptions
- I/O Streams
- Numerics
- etc.
Overview of the Language

- Loose types
- adding operators
- parallel
- introspection
- processes
- threads
- serialization
- I/O
- C
- C++
- //
- namespaces
- RTTI
- bool
- exceptions
- Overloading operators and functions
- references
- templates
- classes
- multiple dispatch
Overview of the Library

- inter-process communication
- C Standard Library
  - drawing
  - GUls
  - parallel
  - security
  - threads
  - database
  - serialization
- C++ Standard Library
  - complex
  - numeric_limits
  - valarray
  - locales
  - iostreams
  - STL
  - allocators
  - string
  - locales
  - matrices

A Whirlwind Tour of C++
Aims of the Language

- C++ makes programming more enjoyable for serious programmers.
- C++ is a general-purpose programming language that
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - *supports generic programming
  (Stroustrup, *Design and Evolution of C++*, + *)
General Rules

- C++’s evolution must be driven by real problems.
- Don’t get involved in a sterile quest for perfection.
- C++ must be useful now.
- Every feature must have a reasonably obvious implementation.
- Always provide a transition path.
- C++ is a language, not a complete system.
- Provide comprehensive support for each supported style.
- Don’t try to force people.
Design Support Rules

- Support sound design notions.
- Provide facilities for program organization.
- Say what you mean.
- All features must be affordable.
- It is more important to allow a useful feature than to prevent every misuse.
- Support composition of software from separately developed parts.
Language-technical Rules

- No implicit violations of the static type system.
- Provide as good support for user-defined types as for built-in types.
- Locality is good.
- Avoid order dependencies.
- If in doubt, pick the variant of a feature that is easiest to teach.
- Syntax matters (often in perverse ways).
- Preprocessor usage should be eliminated.
Low-level Programming Support Rules

- Use traditional (dumb) linkers.
- No gratuitous incompatibilities with C.
- Leave room for lower-level language below C++ (except assembler).
- What you don’t use, you don’t pay for (zero-overhead rule).
- If in doubt, provide means for manual control.
C++ Is a Better C

- C code compiles and runs as C++, except:
  - All functions must be declared (with prototype) before used.
  - No old-style (K&R) function headers.
  - No implicit int.
  - New keywords.
  - Use extern “C” for C linkage (default C++ linkage includes signatures for overloading).
- All but last can be fixed and remain C.
Features Inherited From C

- Built-in types (float, int, etc.), operators, expressions.
- Syntax for pointers, arrays, structures, const, etc.
- Syntax for functions (C prototypes from C++).
- Syntax for decisions and control.
- File organization: header (.h), source (.c, .cc, .C, .cpp, .cxx, etc.).
- Preprocessor (but used less).
- External linkage model.
- Standard conversions (including narrowing).
- Efficiency: You pay only for what you use.
Concrete Data Type Usage:
“Just like built-in types”

```
#include "Complex.h"
#include <iostream>
using namespace std;

int main()
{
    const Complex x(1.0, 2.0);
    Complex y(3.0);
    Complex z = x*y + exp(z);
    y *= x;
    z.imag(x.real());
    cout << "w=" << w << " z=" << z << endl;
}
```
Concrete Data Type: Class Definition

    // In Complex.h:
    class Complex {
        double _re, _im; // private
    public:
        Complex( double r, double i=0) //default
            : _re(r), _im(i){}
        Complex() : _re(0), _im(0) {}
        double real() const { return _re; } //inline
        double imag() const { return _im; } //inline
        void real(double r) {_re = r; }  //inline
        void imag(double i) {_im = i; }  //inline
        // Declarations of member functions
        Complex& operator+=( const Complex& );
        Complex& operator*=( const Complex& );
    };
Concrete Data Type: Extended Interface

// Complex.h continued:
inline Complex& Complex::operator+=( // Member operator
    const Complex& rhs )
{
    _re += rhs._re;
    _im += rhs._im;
    return *this;
}
inline Complex operator+( // Non-member operator
    const Complex& lhs, const Complex& rhs )
{
    Complex result( lhs ); // invokes copy constructor
    result += rhs;
    return result;
}
Complex exp( const Complex& ); // Non-member declaration
ostream& operator<<( ostream&, const Complex& );
Concrete Data Type:
External Definitions

    // In Complex.cc:
#include <Complex.h>
#include <cmath>
    // Member operator
Complex& Complex::operator*=( const Complex& rhs )
{ double temp = _re*rhs._re - _im*rhs._im;
  _im = _re*rhs._im + _im*rhs._re;
  _re = temp;
  return *this;
}
    // Non-member function (overloaded)
Complex exp( const Complex& x )
{ using std::exp, std::cos, std::sin;
  double mag = exp( x.real() );
  return Complex( mag*cos(x.imag()), mag*sin(x.imag()) );
}
Concrete Data Type: External Definitions cont.

// In Complex.cc, continued:
#include <iostream>

// Overloaded inserter for text output
ostream& operator<<( ostream& s, const Complex& x )
{
    s << "(" << x.real() << "," << x.imag() << ")";
    return s;
}
Polymorphism: Usage

// A client of abstract class Shape, // which has derived classes Circle, // Rectangle, etc.
#include "Shape.h"

void drawAllShapes( Shape *first, Shape *last )
{
    for ( Shape *s = first; s != last; ++s )
        s->draw(); // Which draw() gets called?
}
Polymorphism Via Inheritance

```cpp
class Shape { // abstract class, in Shape.h
public:
    virtual void rotate(int) = 0;
    virtual void draw() const = 0; // pure virtual
}; // Abstract class cannot be instantiated.

// In Circle.h:
#include "Shape.h"
class Circle : public Shape {
public:
    Circle(const Point& p, int r); // constructor
    void rotate(int) { }; // overrides
    void draw() const; // definition is in Circle.cc
private:
    Point center; int radius;
};

// etc. for Rectangle, ...
```

A Whirlwind Tour of C++
Polymorphism Via Inheritance: How it works

- For each class that has any virtual functions, there is a list of addresses (vtbl) of v. f. implementations.
- Each derived class inherits the vtbl of its base class, but it substitutes its own v.f. implementations.
- Each object of a class having a virtual function contains a pointer (vptr) to the vtbl for the object’s class.
- A v.f. call is doubly indirect, through vptr and an address in the vtbl.
Inheritance Rules of Thumb

- Public inheritance means “is a”. Must obey the Liskov Substitution Principle.
- Private or protected inheritance means “is implemented in terms of”. Rarely a good idea.
- Cleanest form of public inheritance: Pure abstract base class (“interface”). No state, pure virtual functions.
- Inheritance implies tight coupling from derived class to base class. When in doubt, choose containment instead.
- Polymorphism requires client access via references or pointers. Passing by value will “slice” objects back to base class part.
- Multiple inheritance is useful but tricky. It helps if base classes are “interfaces” (except perhaps one).
Inheritance Rules of Thumb (2)

- Consider refactoring if “bad smells” detected:
  - Inheritance more than 3 levels deep.
  - Clients fall into categories that use different interfaces.
  - Class has too many responsibilities, or not enough.
  - Code is duplicated, with only minor changes.
  - Parameter list, class, or method too big or too small.
  - Switch statements.
  - Inappropriate intimacy with another class.
  - Divergent changes for different causes.
  - Changes require editing many classes.

(Lots more: See Fowler, *Refactoring: Improving the Design of Existing Code*)
Special Member Functions

class Foo {
public:
    Foo();    // Default Constructor
    Foo( const Foo& );   // Copy Constructor
    Foo& operator=( const Foo& );   // Assignment
    ~Foo();    // Destructor
};

• These 4 are created by compiler if you don’t.
• Default versions call same operator for base classes and all data members.
• Destructor must release any resources held in object.
• Assignment must release resources held by target.
• To disable one of these, declare it private.
Template Class: Usage

// Use by client:
#include "Complex.h"       // Complex<T>, next slide
#include <algorithm>        // for std::transform
void client_fun()
{
    Complex<double> cd( 1.0, 3.0 );
    Complex<float> cf;          // = 0
    Complex<double> ecd = exp( cd );
    Complex<double> cdarray[10];  // all 0
    // ... set cdarray values
    // Substitute exp(x) for x in cdarray
    std::transform( cdarray, cdarray+10, cdarray, exp );
}
Template Class Definition

// T can be double, float, long double, Rational, ...
template <class T>
class Complex {
  T _re, _im; // private
public:
  Complex( const T& r=0, const T& i=0) // 0,1,2 args
    : _re(r), _im(i){}
  T real() const { return _re; } // getters
  T imag() const { return _im; }
  void real( const T& r ) { _re = r; } // setters
  void imag( const T& i ) { _im = i; }
  // Constraint: operator+= is defined for T
  Complex& operator+=( const Complex& )
    {_re += rhs._re; im += rhs._im; return *this; }
  Complex& operator*=( const Complex& );
};
// Template Function:  
// Template Function Definition

namespace Complex
{
template<class T>
Complex<T> operator+(  
    const Complex<T>& lhs, const Complex<T>& rhs )
{
    Complex<T> result( lhs );  // copy constructor!
    result += rhs;  // member operator+=
    return result;
}

T must be copy constructible and must define
operator +=.

Such constraints are implicit. C++ has no syntax to
specify them, but compilation fails if they aren’t
met.
Template Specialization

// Assume double version requires an algorithm
// that differs from that for float etc.
#include "Complex.h"

template<>
Complex<double> exp( const Complex<double>& x )
{
    Complex<double> result;
    // Set it using code specific to T=double
    return result;
}

A Whirlwind Tour of C++
Templates: When To Use

- to express algorithms that apply to many argument types
- to express containers (and iterators)
- to specify policy
- instead of inheritance when run-time efficiency is at a premium
- instead of inheritance when no common base class can be defined
Polymorphism via Virtual Functions vs. Templates

<table>
<thead>
<tr>
<th>Virtual Functions</th>
<th>Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-time selection</td>
<td>Compile-time selection</td>
</tr>
<tr>
<td>Common base class</td>
<td>Any type or constant</td>
</tr>
<tr>
<td>Object contains vptr</td>
<td>No memory overhead</td>
</tr>
<tr>
<td>Call is indirect</td>
<td>No run-time overhead</td>
</tr>
<tr>
<td>Cannot be in-line</td>
<td>Can be in-line</td>
</tr>
<tr>
<td>One copy of each method</td>
<td>Can cause code bloat</td>
</tr>
<tr>
<td>Client knows interface (base class and signature)</td>
<td>Client must know actual type and template</td>
</tr>
</tbody>
</table>
Templates: Other Topics

- Member templates
- Partial specialization
- Non-type template arguments
- Default template parameters
- Using templates to specify policy (traits etc.)
- Recursive templates vs. iteration
- Order of specializations (how compiler chooses)
- When and how are specializations generated?
Standard Template Library (STL)

Containers
- hold values (elements), create iterators

Allocators
- manage memory

Iterators
- provide access to elements in containers

Algorithms
- operate on multiple elements via iterators

Function Objects
- implement operations on elements

pair<T1,T2>

iterator_traits

iostream etc
STL Example: Read a File 1

// This version uses vector as expandable int[]
#include <fstream>
#include <vector>
void ReadIntegers( const char *filename, 
   std::vector<int>& result ) 
{ ifstream s( filename );
   int val;
   while ( s >> val )
      result.push_back( val ); // It grows!
} // ifstream destructor closes file

- Vector access is like an array: result[i].
- Memory is released by destructor.
// This version uses iterators, STL copy
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
void ReadIntegers2( const char *filename,
    vector<int>& result )
{
    ifstream s( filename );
    copy( istream_iterator<int>( s ),
      istream_iterator<int>(),
      back_inserter(result) );
}
// Generalize contained type and container type
// #include & using as before

template < class T, template<class> class C >
void ReadAny( const char *filename,
        C<T>& result )
{ ifstream s( filename );
    copy( istream_iterator<T>( s ),
        istream_iterator<T>(),
        back_inserter(result) );
}
Standard Template Library (STL)

**Containers**
- Hold values (elements), create iterators

**Algorithms**
- Operate on multiple elements via iterators

**Allocators**
- Manage memory

**Iterators**
- Provide access to elements in containers

**Function Objects**
- Implement operations on elements

**iostream etc**

**pair<T1,T2>**

**iterator_traits**
STL Sequence Containers

- `vector<T>`: Random access, grows at back. Use this instead of arrays.
- `list<T>`: Bidirectional access, grows anywhere.
- `deque<T>`: Random access, grows at front or back.
- `queue<T>` (adapter): Grow at back, access and remove at front.
- `stack<T>` (adapter): Push, pop, top at front.
- `priority_queue<T>` (adapter): Add anywhere, access and remove highest priority. T ordered.
STL Associative Containers

- `map<Key,T>`: Access unique T by specifying key; add or remove anywhere. Key must be ordered. Holds pair<const Key,T>.
- `multimap<Key,T>`: Like map, allowing duplicate keys (equivalence classes).
- `set<T>`: Unique values with order. Add or remove anywhere, query for presence. T must be ordered.
- `multiset<T>`: Like set, allowing duplicate values (equivalence classes).
Almost STL Containers

- `basic_string<C>`: String of character type
- `valarray<T>`: Vector optimized for numeric computation
- `bitset`: Fixed-size packed array of booleans
- `built-in arrays`: Can be used with most STL algorithms. Iterator is pointer.
- `iostreams`: STL provides `istream_iterator`, `ostream_iterator`. 
STL vector: Array-like Access

template <class T, class A=allocator<T> >
class std::vector {
public:
   typedef T& reference;
   typedef const T& const_reference;
   typedef long size_type; // maybe
   reference operator[]( size_type n ); // unchecked
   const_reference operator[]( size_type n ) const;
   reference at( size_type n ); // checked
   const_reference at( size_type n ) const;
   reference front(); reference back(); // + const
   // ...
template <class T, class A=allocator<T> >
class std::vector {
public:
    typedef T* iterator; // maybe
    iterator begin();
    iterator end(); // to one past last

typedef const T* const_iterator; // maybe
    const_iterator begin() const;
    const_iterator end() const;
    // etc. for reverse_iterator
};
STL Containers: Member Types

- **value_type, reference, const_reference**: Behave like T, T*, const T*.
- **size_type, difference_type**: Type of subscripts, difference between iterators.
- **iterator, const_iterator, reverse_iterator, const_reverse_iterator**: Each container defines a set of iterator types.
- **key_type, mapped_type, key_compare**: Associative containers only.
STL Containers: Access

- **begin()**: Iterator for first element.
- **end()**: Iterator for one-past-last element.
- **rbegin(), rend()**: Reverse iterators.
- **front(), back()**: Reference to first and last elements.
- **operator[] (size_type)**: Subscripting, unchecked.
- **at(size_type)**: Subscripting, checked.
- All return const version if container is const.
STL Containers: Modification

- `push_back(T&)`, `pop_back()` : Add or remove from end.
- `push_front(T&)`, `pop_front()` : Add or remove at front (list, deque only).
- `insert(iterator& p, T& x)`;
  `template <class iterator2>
  insert(iterator& p, iterator2& first, iterator2& last)`: Add value(s) before p.
- `erase (iterator& p)`,
  `erase(iterator& first, iterator& last)` : Remove element(s).
- `clear()`: `erase( begin(), end() )`
STL Containers: Other ops

- `size()`, `empty()`, `max_size()`:
  Cardinality.
- `capacity()`,
  `reserve(size_type)`: vector only.
- `resize(size_type, T val=T())`:
  vector, list, deque.
- `swap()`:
  Swap elements of 2 containers.
- `==`, `!=`, `<`:
  Lexographic comparisons.
STL Containers: Constructors

- `container()`: Empty container.
- `container(size_t n, const T& x=T()):` n copies (not associative containers)
- `template <class iterator2> container(iterator2 first, iterator2 last):` Copy elements from a range.
- `container(const container& c):` Copy elements from another container.
- `~container():` Destroy container and all elements.
STL Containers: Assignments

- `operator=(const container& x)`: Copy all elements from x.
- `assign( size_t n, T& x )`: Assign n copies of x (not for associative containers).
- `template <class iterator2> assign( iterator2& first, iterator2& last )`: Assign from range.
- All assignments destroy existing elements.
STL Containers: Associative

- **operator[] (k)**: Reference to element with unique key k.
- **find(k)**: Iterator to element with key k.
- **lower_bound(k), upper_bound(k)**: Iterators to first and last+1 elements with key k.
- **equal_range(k)**: pair(lower_bound(k), upper_bound(k)).
- **key_comp()**: Copy of key comparison object.
- **value_comp()**: Copy of mapped_value comparison object.
## Container Summary

<table>
<thead>
<tr>
<th></th>
<th>[]</th>
<th>List Operations</th>
<th>Front Operations</th>
<th>Back (Stack) Operations</th>
<th>Iterators</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector list deque</td>
<td>const</td>
<td>O(n)+</td>
<td>const</td>
<td>const+</td>
<td>Ran</td>
</tr>
<tr>
<td></td>
<td>const</td>
<td>const</td>
<td>const</td>
<td>const</td>
<td>Bi Ran</td>
</tr>
<tr>
<td>stack queue priority_queue</td>
<td></td>
<td></td>
<td>const O(log(n))</td>
<td>const const O(log(n))</td>
<td>Bi Bi Bi</td>
</tr>
<tr>
<td>map multimap set multiset</td>
<td>O(log(n))</td>
<td>O(log(n))+</td>
<td>O(log(n))+</td>
<td>O(log(n))+ O(log(n))+</td>
<td>Bi Bi Bi</td>
</tr>
<tr>
<td>string array valarray bitset</td>
<td>const</td>
<td>O(n)+</td>
<td>O(n)+</td>
<td>const+</td>
<td>Ran Ran Ran</td>
</tr>
</tbody>
</table>

A Whirlwind Tour of C++
STL Contained Type Concepts

- **Regular Type**: Assignable, Default Constructible, Equality Comparable. Required for many algorithms, generally a good idea for elements of containers.

- **Ordering**: Less Than Comparable, Strict Weakly Comparable (refinement). Required for some algorithms & roles, e.g. as key in associative container.

- Built-in numeric types are *models* of all of these, plus Totally Ordered (further refinement).
STL Container Concepts

- Hold any "regular type" as copies.
- Responsible for construction & destruction of elements.
- Responsible for memory management via plug-in allocators.
- Const correct, exception safe.
- Uniform access syntax via iterators.
- Associated types declared via member typedefs.
- Array-like access syntax where it makes sense.
- Common syntax for adding and removing elements (subsets depending on container type).
Standard Template Library (STL)

Containers
- hold values (elements),
- create iterators

Allocators
- manage memory

Algorithms
- operate on multiple elements via iterators

Iterators
- provide access to elements in containers

Function Objects
- implement operations on elements

pair<T1,T2>

Iterator_traits

iostream
- etc

A Whirlwind Tour of C++
STL Algorithms: Definitions

- The STL algorithms are template functions that operate on ranges of values.
- All are defined in header `<algorithm>`.
- Ranges are specified using *iterators*.
- For some algorithms, operations on individual elements are specified using *function objects*, which behave like unary or binary functions.
- Each algorithm requires a specific class of iterator and may constrain the value type.
STL Algorithms: Example Use

```cpp
#include "Club.h"    // my own class
#include <list>
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

// Club has a member function of the form
string Club::name() const { ... }

void printClubNames( const list<Club>& lc )
{ transform( lc.begin(), lc.end(),
            ostream_iterator<string>( cout, "\n" ),
            mem_fun_ref( &Club::name ) );
}
```
STL transform Specification

```cpp
template <class In, class Out, class Op>
Out transform( In first, In last, Out res, Op op );

template <class In, class In2, class Out, class Op>
Out transform( In first, In last, In2 first2, Out res,
   Op op );
```

- For each element in the range \([first, last)\), apply the operation and place the result in the range starting at \(res\).
- Two-input form: Second argument of op comes from range starting at \(in2\).
- Op defines function call operator of the form
  
  \[
  *res = op( *first ) \text{ or } *res = op( *first, *first2 )
  \]
STL transform Implementation

template <class In, class Out, class Op>
Out transform( In first, In last, Out res, Op op )
{
    while ( first != last )
        *res++ = op( *first++ );
    return res;
}

template <class In, class In2, class Out, class Op>
Out transform( In first, In last, In2 first2, Out res, Op op )
{
    while ( first != last )
        *res++ = op( *first++, *first2++ );
    return res;
}
STL Algorithms: Nonmodifying Sequence Ops

for_each       mismatch
find            equal
find_if         search
find_first_of    find_end
adjacent_find    search_n
count            
count_if
STL Algorithms: Modifying Sequence Ops

- copy
- copy_backward
- swap
- iter_swap
- swap_ranges
- replace
- transform
- replace_if
- replace_copy
- replace_copy_if
- fill
- fill_n
- generate
- generate_n
- remove
- remove_if
- remove_copy
- remove_copy_if
- unique
- unique_copy
- reverse
- reverse_copy
- rotate
- rotate_copy
- random_shuffle
STL Algorithms: Sorted Sequence Ops

- sort
- stable_sort
- partial_sort
- partial_sort_copy
- nth_element
- lower_bound
- upper_bound
- equal_range

- binary_search
- merge
- inplace_merge
- partition
- stable_partition
STL Algorithms: Set and Heap Operations

- includes
- set_union
- set_intersection
- set_difference
- set_symmetric_difference
- make_heap
- push_heap
- pop_heap
- sort_heap
STL Algorithms: Extrema and Permutations

min
max
min_element
max_element
lexicographical_compare
next_permutation
prev_permutation
STL Algorithm Concepts

- Input and output sequences accessed via iterators.
- Iterator types required (forward, bidirectional, random access, etc.) depend on algorithm.
- Element types required (Less Than Comparable, etc.) depend on algorithm.
- Operations on elements specified via function object: Function address, object with operator() defined, or adapter.
- Decisions specified via predicate: Function object returning bool.
Containers hold values (elements), create iterators

Algorithms operate on multiple elements via iterators

Allocators manage memory

pair<T1,T2>

Iterators provide access to elements in containers

Function Objects implement operations on elements

iterator_traits

A Whirlwind Tour of C++
## Iterator Operations and Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Out</th>
<th>In</th>
<th>For</th>
<th>Bi</th>
<th>Ran</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>*=p</td>
<td>*=p</td>
<td>*=p</td>
<td>*=p</td>
<td>*=p</td>
</tr>
<tr>
<td>Access</td>
<td>-&gt;</td>
<td>-&gt;</td>
<td>-&gt;</td>
<td>-&gt;</td>
<td>-&gt; [ ]</td>
</tr>
<tr>
<td>Write</td>
<td>*=p=</td>
<td>*=p=</td>
<td>*=p=</td>
<td>*=p=</td>
<td>*=p=</td>
</tr>
<tr>
<td>Iterate</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++ -- +</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- += -=</td>
</tr>
<tr>
<td>Compare</td>
<td>==</td>
<td>!=</td>
<td>==</td>
<td>!=</td>
<td>== != &lt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&gt; &gt;= &lt;=</td>
</tr>
</tbody>
</table>
Iterator Sources

- Container operations: `begin()`, `end()`, etc.
- Operations on other iterators: `=, ++, --, +, -`
- Results of some STL algorithms: `find` etc.
- Many STL modifying algorithms return iterator just past last modification.
- Pointers into arrays are valid iterators.
- Template classes defined in `<iterator>`: `back_inserter`, `front_inserter`, `inserter`.
- Stream iterators: `ostream_iterator`, `istream_iterator`, `ostreambuf_iterator`, `istreambuf_iterator`
- Your own, e.g. checked iterators.
STL Iterator Concepts

- Iterator Categories: Input, Output, Forward, Bidirectional, Random Access
- Assignable, Default Constructible, Equality Comparable. Random Access iterators are also Strict Weakly Comparable.
- Associated types (value_type, pointer, reference, etc.) defined as typedefs in iterator_traits template class.
- Dereference via unary * – unless iterator is "singular" (e.g. past end of container).
- Increment via ++. Maybe --, +, - too.
- Reading or writing through iterator might do other things, e.g. I/O, appending.
Standard Template Library (STL)

Containers
- hold values (elements), create iterators

Algorithms
- operate on multiple elements via iterators

Allocators
- manage memory

Pair
- pair<T1,T2>

Iterators
- provide access to elements in containers

Function Objects
- implement operations on elements

Iterator traits

A Whirlwind Tour of C++
STL Function Objects

- Passed as input argument to some STL algorithms to specify operation on individual elements of ranges.
- Behave like a unary or binary function.
- Most commonly, use the address of a function.
- OR, define an object that defines operator().
- OR, use an adapter that constructs a function object for you.
- STL’s function objects and adapters are in `<functional>`.

A Whirlwind Tour of C++
template <class T>
struct logical_not: public unary_function<T,bool> {
    bool operator()( const T& x ) const
    {
        return !x;
    }
};

template <class T>
struct less: public binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const
    {
        return x < y;
    }
};

- These are *predicates* (return bool) defined in STL.
- Base classes define types.
- Similar ones for operators == != > >= <= && || !
Function Object Usage

- Example: Compare two sequences, looking for the first element of one that is less than the corresponding element of the other.

```cpp
void f( vector<int>& vi, list<int>& li )
{
    typedef list<int>::iterator LI;
    typedef vector<int>::iterator VI;
    pair<VI,LI> pl = mismatch(
        vi.begin(), vi.end(), li.begin(),
        less<int>() );
    // ...
}
```
STL Function Objects: Predicates and Arithmetic

- `equal_to`
- `not_equal_to`
- `greater`
- `less`
- `greater_equal`
- `less_equal`
- `logical_and`
- `logical_or`
- `logical_not`
- `plus`
- `minus`
- `multiplies`
- `divides`
- `modulus`
- `negate`
STL Function Objects: Binders, Adapters, Negaters

bind2nd()
bind2st()
mem_fun()
mem_fun_ref()
ptr_fun()
not1()
not2()

Each of these adapter functions is a template function that returns a function object as its result. Typically the call to an adapter function is placed in-line in the argument list of a call to an STL algorithm.
STL Adapters: Example Use

```cpp
// Count the number of clubs whose name contains a given string (e.g. "National")
bool contains( const string& outer,
               const string& inner )
{ return outer.find( inner ) != string::npos; }

int countClubsNamed( const list<Club>& lc,
                      const string& namepart )
{ return count_if( lc.begin(), lc.end(),
                   bind_second( ptr_fun(contains), namepart ) ); }
```

- Adapters support composition of function objects.
- An adapter is a high-order function: Takes function argument and produces a new function.
Exceptions

```
#include <stdexcept>

class Complex::DivZero : public std::exception {
    const char *what() const // override
    {
        return "Complex divide by zero";
    }
}

Complex& Complex::operator/=(
    const Complex& den )
{
    if ( den == Complex(0.0,0.0) )
        throw ComplexDivZero();
    // ... do the calculation ... 
    return *this;
}
```
Exceptions (2)

// Client code
#include <stdexcept>
...

try {
    // ... calculation involving Complex etc.
}
catch ( std::exception& e ) {
    cerr << "std::exception caught"
         << e.what() << endl;
    throw;     // re-throw same exception
}

Exceptions (3)

- Throw at any level unwinds call stack until a matching catch is found. At each level, destructors for local variables are called before leaving function.
- Any type may be thrown & caught. Normally throw a special-purpose class, often based on std::exception hierarchy.
- Use only for bona fide errors, not for control.
- Advantage over status flag: Cannot be ignored.
- Advantage over longjump: Calls destructors.
- But: Writing exception-safe code is hard.
Streams: Class Relationships

\textit{ios\_base}: locale independent format state

\textit{basic\_ios\<>}: locale dependent format state
stream state

\textit{basic\_iostream\<>}: formatting ($<<,>>$, etc.)
setup/cleanup

\textit{basic\_streambuf\<>}: buffering

\textit{locale}: format information
character buffer
real destination/source

A Whirlwind Tour of C++
Streams

- Template arguments are character type and character traits (default char_traits<Ch>).
- ostream, istream, iostream are typedefs for basic_ostream<char> etc.
- Predefined ostream’s: cout, cerr, clog.
- Predefined istream: cin.
- File I/O: ifstream, ofstream, fstream.
- String I/O: istringstream, ostringstream, stringstream.
Streams: Example Usage

#include "Date.h" // Date with operator<<
#include <iostream>
using namespace std;

// Inserter usage
float score; Date today;
cout << "The score on " << today
    << " is " << setprecision(3) << score
    << endl;

// Extractor usage
string answer;
cin >> answer;
Streams: Advantages over C

- Type safe.
- Extensible to I/O of user-defined types.
- Extensible to I/O to/from any destination.
- Buffering strategy can be modified via plug-in.
Numerics

- `complex<T>`: Concrete type, same rep as Fortran, the usual library functions and operations, template is underlying numeric type.
- Standard math functions: Same as C but includes float, long double versions + convenience stuff like min, max.
- `numeric_limits<T>`: Machine dependencies.
- `valarray, slice, slice_array, gslice`: Vectors optimized for numeric computation.
Where to Get More Information

- Book list posted at:
  http://staff.washington.edu/aganse/staffmtg.html