Practical STL

Chuck Allison
Fresh Sources
chuck@freshsources.com

Objectives

- Understand Generic Programming
- Apply STL to real programming problems
- “Standard” STL
  - as defined in the draft C++ Standard (CD2)
  - Examples “work” with VC 5.0, BC 5.0

About Me...

- Owner, *Fresh Sources*
  - C++ Training & Consulting
- Consulting Editor, *C/C++ Users Journal*
- Contributing Member, *J16* (C++ Standards)
- Author, *C/C++ Code Capsules*, P-H, 1998.

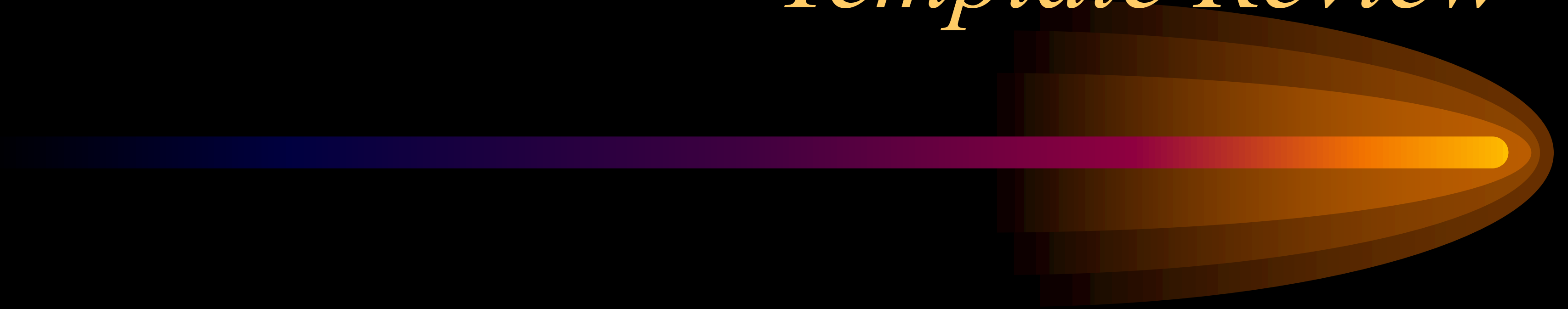
About you...

- Understand *pointers* and *references*
  - including pointers to functions
- Understand `class` mechanism, including:
  - constructors, initializer lists
- Understand *operator overloading*
  - member vs. non-member flavors
- Familiar with `template` facility
  - both *function* and *class* varieties

Agenda

- Template Review
- Algorithms
- Function Objects & Adapters
- Containers & Iterators
- Applications

Template Review



A swap function

```
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

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

main()
{
    int a = 1, b = 2;
    swap(a,b);
    cout << "a == " << a
         << ", b == " << b << endl;
}

// Output:
a == 2, b == 1
```

How to swap other types of objects?

Function overloading

```
void swap(float& x, float& y)
{
    float temp = x;
    x = y;
    y = temp;
}

void swap(string& x, string& y)
{
    string temp = x;
    x = y;
    y = temp;
}

// etc.
```

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

main()
{
    int a = 1, b = 2;
    float c = 100.0, d = 200.0;
    char *s = "hello", *t = "goodbye";

    swap(a,b);
    cout << "a == " << a << ", b == " << b << endl;
    swap(c,d);
    cout << "c == " << c << ", d == " << d << endl;
    swap(s,t);
    cout << "s == " << s << ", t == " << t << endl;
}

// Output:
a == 2, b == 1
c == 200, d = 100
s == goodbye, t == hello
```

Observations

- The logic is identical in each version of `swap`
- Only the types of the objects changes
- Repetitive editing is tedious and error-prone

A Better Way

- Capture the logic once
- Make the type a *parameter*

Macro Approach

```
// swap1.h:
#define genswap(T)      \
void swap(T& x, T& y) \
{                       \
    T temp = x;         \
    x = y;              \
    y = temp;           \
}
```

Using the genswap *macro*

```
#include <iostream>
#include "swap1.h"
using namespace std;

// Generate the needed code:
genswap(int)
genswap(float)
genswap(char*)

main()
{
     // (same as before)
}
```

More Observations

- I have to explicitly generate each version
  - I might forget one!
- `genswap()` looks funny w/o a semi-colon!
- Aren't function-like macros near-extinct?
- Can't the compiler help me out?

A Function Template

```
// swap2.h (Template version)
template<class T>
void swap(T& x, T& y)
{
    T t = x;
    x = y;
    y = t;
}
```

```
#include <iostream>
#include “swap2.h”
using namespace std;
                                  // no genswap calls
main()
{
    int a = 1, b = 2;
    float c = 100.0, d = 200.0;
    char *s = "hello", *t = "goodbye";

    swap(a,b);
    cout << "a == " << a << ", b == " << b << endl;
    swap(c,d);
    cout << "c == " << c << ", d == " << d << endl;
    swap(s,t);
    cout << "s == " << s << ", t == " << t << endl;
}

// Output:
a == 2, b == 1
c == 200, d = 100
s == goodbye, t == hello
```

About Function Templates

- compiler deduces necessary types from usage
- generates code on demand

A Class Template

```
template<class T>
class Set
{
public:
    Set();
    bool contains(const T&) const;
    void insert(const T&);
    void remove(const T&);
    void print(std::ostream&) const;
private:
    enum {LIMIT = 64};
    T elems[LIMIT];
    size_t nelems;
};
```

Instantiating Template Classes

```
Set<int> s1;
Set<float> s2;
Set<string> s3;
```

- You must provide template arguments

Explicit Instantiation

```
// Explicit requests for code generation
// without immediate variable declarations:

template class set<string>;
template void swap<string>(string&, string&);
```

Template Parameters

- Type vs. Non-type

```
template<class T, size_t N>
class Set {…};
Set<int, 100> s;
```

- Default parameters

```
template<class T, size_t N = 100>
class Set {…};
Set<int> s;  // same as Set<int,100>
```

Template Specialization

```
template<class T>
int comp(const T& t1, const T& t2)
{
    return (t1 < t2) ? -1 : (t1 == t2) ? 0 : 1;
}

// char* specialization (template<> optional, but good style)
template<>
int comp<const char*>(const char*& t1, const char*& t2)
{
    return strcmp(t1,t2);
}
```

Partial Specialization

```
template<class T, class U>
class A
{
public:
        A() {cout << "primary template\n";}
};

template<class T, class U>
class A<T*, U>          // used when 1st arg is a pointer
{
public:
        A() {cout << "<T*,U> partial specialization\n";}
};

template<class T>
class A<T, T>           // used when args are equal
{
public:
        A() {cout << "<T,T> partial specialization\n";}
};
```

"Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures."

-- Bjarne Stroustrup

Template Review Summary

- Templates implement *type abstraction*
  - types are parameters, independent of logic
  - support generic programming
- Templates also allow *non-type* parameters
- C++ supports *function templates* and *class templates*
- You override default template instantiation via a *specialization*

Lab 1

Algorithms

What's an Algorithm?

al•go•rithm: a rule of procedure for solving a mathematical problem (as of finding the greatest common divisor) that frequently involves repetition of an operation.

Euclid's Algorithm
(from Knuth, Vol. 1)

Algorithm E (Euclid's Algorithm). Given two positive integers `m` and `n`, find their greatest common divisor, i.e., the largest positive integer which evenly divides both `m` and `n`.

E1. [Find remainder.] Divide `m` by `n` and let `r` be the remainder.

E2. [Is it zero?] If `r = 0`, the algorithm terminates; `n` is the answer.

E3. [Interchange.] Set `m ← n`, `n ← r`, and go back to step E1.

Euclid's Algorithm
C Version

```
/* Euclid's Algorithm */
#include <assert.h>

int gcd(int m, int n)
{
    int r;
    assert(m > 0 && n > 0);

    while ((r = m % n) != 0)
    {
        m = n;
        n = r;
    }
    return n;
}
```

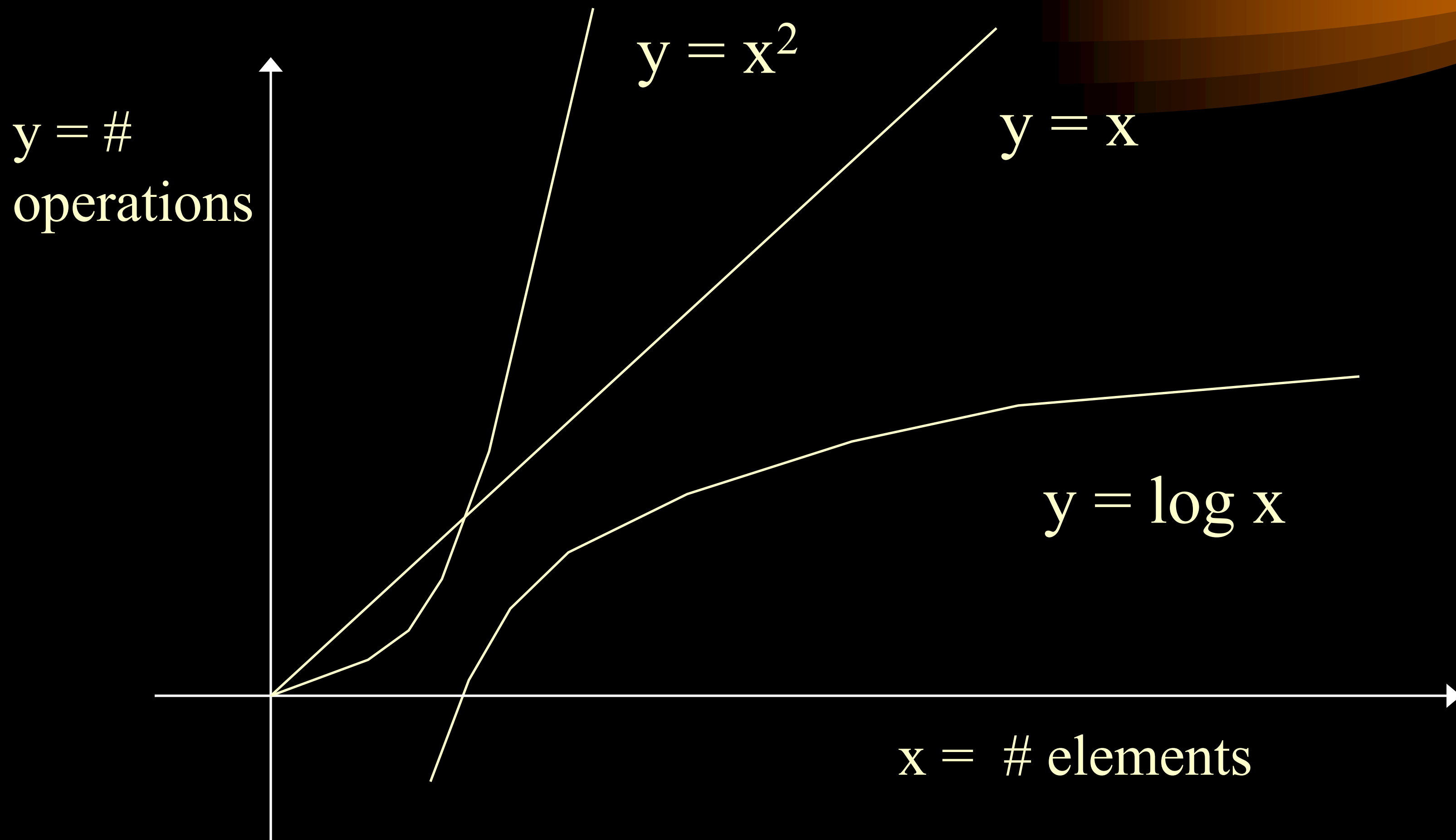
Properties of Algorithms

- Finite
  - `gcd()` terminates because 0 <= r < n
  - i.e., $\{r_i\}$ is a strictly decreasing sequence
- Optional Input
  - pre-conditions
  - m and n are positive integers
- Output
  - post-conditions
  - yields the `gcd`

Complexity

- A measure of efficiency
- Count of key operations (worst case)
  - e.g., comparisons in a sort
- "Big-oh" notation (*asymptotic* behavior)
  - O(1) *constant time* (hashing)
  - O(log n) *logarithmic* (binary search)
  - O(n) *linear* (simple search)
  - O(n log n) (quick sort, merge sort)
  - $O(n^2)$ *quadratic* (naïve sorts)

A Graphical Look at Asymptotic Complexity



Complexity
Sample Measures

- For 100,000 elements @ 1,000,000 ops/sec:
  - log n ==> .000012 sec
  - n ==> .1 sec
  - n log n ==> 1.2 sec
  - n^2 ==> 2.8 hrs.
  - n^3 ==> 31.7 yrs.
  - 2^n ==> > 1 century

Linear Search

```
// A linear search:
int search(int* data, int n, int x)
{
    for (int i = 0; i < n; ++i)
        if (data[i] == x)
            return i;
    return -1;
}
```

Average case: $\lceil n/2 \rceil$ **comparisons**

Worst case: **n comparisons**

Binary Search

```
// Binary search (assumes data is in ascending order)
int search(int* data, int n, int x)
{
    int begin = 0;
    int end = n;
    while (begin < end)
    {
        int mid = (begin + end) / 2;
        assert(mid < end);
        if (x == data[mid])
            return mid;
        else if (x < data[mid])
            end = mid - 1;
        else // x > data[mid]
            begin = mid + 1;
    }
    return -1;
}
```

Binary Search Complexity

- Divides list in half each time
- How many times can you halve a number and still have something left over?
- Answer: the number of times 2 is factor of a number (plus one, if the number is odd).
- $2^m = n \Rightarrow m = \log_2 n$
- Formula: $\lceil \log_2 n \rceil$

∴ Binary search is O(log n)

Generic Algorithms

- Independent of data type
- Template Functions

find

A Generic Algorithm

```
#include <algorithm>
main()
{
    int a[] = {45,23,89,12,78};
    int n = sizeof a / sizeof a[0];
    int* past = a + n;

    int* p = find(a, past, 12);
    if (p != past)
        cout << "found " << *p << " in position "
             << p - a << endl;
    else
        cout << "item not found\n";
}

// Output:
found 12 in position 3
```

Ranges

- `find` works on *sequences* of objects
- The subsequence, or *range* of interest is delimited by two pointers:
  - pointer to first element (`a`)
  - pointer "past-the-end" (`past`)
    - ANSI C notion
    - can't dereference it

Using `find` *with strings*

```
main()
{
    string a[] = {"Albert", "Charles", "Horatio"};
    int n = sizeof a / sizeof a[0];
    string* past = a + n;

    string* p = find(a, past, "Charles");
    if (p != past)
        cout << "found " << *p << " in position "
             << p - a << endl;
    else
        cout << "item not found\n";
}

// Output:
found Charles in position 1
```

Implementing `find`
(a first attempt)

```
template<class T>
T* find(T* start, T* past, const T& v)
{
    while (start != past)
    {
        if (*start == v)
            break;
        ++start;
    }
    return start;
}
```

How `find` works

- Compiler deduces type `T` from function call, and generates the code for a `T`-version of `find`
- `T` must support `operator==`

A User-defined Type

```
struct Person
{
    string name;
    int year;
    int month;
    int day;
    Person() : name("")
    {
        year = month = day = 0;
    }
    Person(const string& nm, int y, int m, int d)
        : name(nm)
    {
        year = y;
        month = m;
        day = d;
    }
};
```

```
bool operator==(const Person& p1, const Person& p2)
{
    return p1.name  == p2.name &&
           p1.year == p2.year &&
           p1.month == p2.month &&
           p1.day == p2.day;
}

ostream& operator<<(ostream& os, const Person& p)
{
    os << '{' << p.name << ',' << p.month
       << '/' << p.day  << '/' << p.year << '}';
    return os;
}
```

find *with* Person

```
main()
{
    Person a[3];
    a[0] = Person("Albert", 1901,1,20);
    a[1] = Person("Charles", 1897,3,11);
    a[2] = Person("Horatio", 1835,12,6);
    int n = sizeof a / sizeof a[0];
    Person* past = a + n;
    Person v("Charles", 1897,3,11);

    Person* p = find(a, past, v);
    if (p != past)
        cout << "found " << *p << " in position "
             << p - a << endl;
    else
        cout << "item not found\n";
}

// Output:
found {Charles,3/11/1897} in position 1
```

Adjusting the Match Criterion

- Equality isn't always what you want
- An alternate version, `find_if` takes a *predicate*
  - a function that takes a parameter of type `T` and returns a `bool`
  - in this case, a *unary* predicate

A Unary Predicate

```
// Use only the name for comparison
bool isCharles(const Person& p)
{
    return p.name == "Charles";
}
```

find_if

```
Person* p = find_if(a, past, isCharles);

if (p != past)
…

// Output:
found {Charles,3/11/1897} in position 1
```

Implementing `find_if`

```
template<class T, class Pred>
T* find(T* start, T* past, Pred p)
{
    while (start != past)
    {
        if (p(*start))
            break;
        ++start;
    }
    return start;
}
```

- Non-mutating
  - read-only
  - typically returns some value or set of values
- Mutating
  - writes to a new sequence, or
  - alters the original sequence
- Ordering
  - sorting, min/max, heap operations, etc.

Non-mutating Algorithms

```
for_each            count_if
find                mismatch
find_if             equal
find_first_of       search
adjacent_find       find_end
count               search_n
```

Mutating Algorithms

```
transform           generate
copy                generate_n
copy_backward       remove
swap                remove_if
iter_swap           remove_copy
swap_ranges         remove_copy_if
replace             unique
replace_if          reverse
replace_copy        reverse_copy
replace_copy_if     rotate
fill                rotate_copy
fill_n              random_shuffle
```

Ordering Algorithms

<u>Sorting</u>

```
sort
stable_sort
partial_sort
partial_sort_copy
nth_element
merge
inplace_merge
partition
stable_partition
```

<u>Set Operations</u>

```
includes
set_union
set_intersection
set_difference
set_symmetric_difference
```

<u>Heap Operations</u>

```
push_heap
pop_heap
make_heap
sort_heap
```

Ordering Algorithms continued...

Searching

- `binary_search`
- `lower_bound`
- `upper_bound`
- `equal_range`

Permutations

- `next_permutation`
- `prev_permutation`

Min/max

- `min`
- `max`
- `min_element`
- `max_element`
- `lexicographical_compare`

Sample Program

```
#include <algorithm>// most algorithms
#include <numeric>  // accumulate
#include <iostream>
#include <string>
#include <utility>  // struct pair
using namespace std;

void print_array(int [], int);

bool odd(int n) {return (n % 2) == 1;}

int calc_parity(int n) {return n%2;}

string parity[] = {"even", "odd"};

void print_parity(int n)
{
    cout << parity[n%2] << ' ';
}
```

```
main()
{
    int a[] = {1,2,3,4,5};
    const int nelems = sizeof a / sizeof a[0];
    int b[nelems];

    // copy
    copy(a, a + nelems, b);
    print_array(b, nelems);

    // test for equality:
    cout.setf(ios::boolalpha);
    bool test = equal(a, a + nelems, b);
    cout << "a == b? " << test << endl;

// Output:
1 2 3 4 5
a == b? true
```

```
    // reverse
    reverse(b, b + nelems);
    print_array(b, nelems);
    test = equal(a, a + nelems, b);
    cout << "a == b? " << test << endl;

    // sort
    sort(b, b + nelems);
    print_array(b, nelems);
    test = equal(a, a + nelems, b);
    cout << "a == b? " << test << endl;

// Output:
5 4 3 2 1
a == b? false
1 2 3 4 5
a == b? true
```

```
    // accumulate
    cout << "sum == "
         << accumulate(a, a + nelems, 0)
         << endl;

    // count
    int n3 = count(b, b + nelems, 3);
    cout << "# = 3's in b: " << n3 << endl;

    // count with predicate
    int nodd = count_if(b, b + nelems, odd);
    cout << "# odd's in b: " << nodd << end

// Output:
sum == 15
# = 3's in b: 1
# odd's in b: 3
```

```
    // find
    int* p = find(a, a + nelems, 2);
    if (p != a + nelems)
        cout << "found " << *p << endl;

    // find with predicate
    p = find_if(a + 1, a + nelems, odd);
    cout << "Start with first odd after second"
            " element: ";
    while (p != a + nelems)
        cout << *p++ << ' ';
    cout << endl;

// Output:
found 2
Start with first odd after second element: 3 4 5
```

```
    // for_each
    for_each(a, a + nelems, print_parity);
    cout << endl;

    // transform
    transform(b, b + nelems, b, calc_parity);
    print_array(b, nelems);

    // mismatch
    pair<int*, int*> p2 = mismatch(a, a + nelems, b);
    cout << "Mismatch in a[" << p2.first-a << "] vs. "
         << "b[" << p2.second-b << "]\n";
}

// Output:
odd even odd even odd
1 0 1 0 1
Mismatch in a[1] vs. b[1]
```

Words of Wisdom

"Algorithms are the *stuff* of Computer Science."

-- Robert Sedgewick

Algorithms Summary

- Generic
  - template functions
- Operate on *ranges*
  - “past-the-end” marker
- Default & Predicate Versions
- Non-mutating, Mutating, Ordering

Lab 2

Function Objects & Adapters

Desperately Seeking a More Powerful Predicate

- Recall `isCharles()`:

```
bool isCharles(const Person& p)
{
    return p.name == "Charles";
}
```

- Not very useful
  - need `isAlbert`, `isHoratio`, etc.

Still Seeking a More Powerful Predicate

- Need to simulate runtime definition of functions
  - Alas, C++ is not Lisp!
- Something that “acts like a function”
- Something we can initialize with different comparison keys

Function Objects

- A *function object*, is an object that “acts like a function”. i.e., it defines `operator()`
- In this example, we need to simulate a unary function with `operator()(const Person&)`
- Constructor takes comparison key

About `operator()`

- If `x` is an object of type `X` then

  `x()`

  is equivalent to

  `x.operator()`

- Can also take arguments, e.g.,

  `x(21)`

  invokes

  `X::operator()(int i)` on `x` with `i == 21`

*A Function Object that can Match an **Arbitrary Name***

```
struct byName
{
    string name;
    byName(const string& s) : name(s){}
    bool operator()(const Person& p)
    {
        return p.name == name;
    }
};

// Use like this:
Person* p = find_if(a, past, byName("Charles"));
```

How **`byName`** *works*

The following expression in `find_if`

```
if (p(*start))
```

is equivalent to

```
if (byName("Charles").operator()(*start))
```

which evaluates the expression

```
p.name == "Charles";
```

Standard Function Objects

Predicates

equal_to
not_equal_to
greater
less
greater_equal
less_equal
logical_and
logical_or
logical_not

Arithmetic

plus
minus
multiplies
divides
modulus
negate

Negaters

unary_negate
binary_negate

Member-related

mem_fun_t
mem_fun1_t
mem_fun_ref_t
mem_fun1_ref_t

Binders

binder1st
binder2nd

Pointer-related

pointer_to_unary_function
pointer_to_binary_function

Function Object Example

multiplies

- Does sums by default:

```
#include <numeric>
…
int sum = accumulate(a, past, 0);
```

- Use **`multiplies`** to do product:

```
#include <numeric>
#include <functional>
…
int prod = accumulate(a, past, 1,
                      multiplies<long>());
```

Implementation of **multiplies**

```
template<class T>
struct multiplies : binary_function<T, T, T>
{
    T operator()(const T& x, const T& y) const
    {
        return (x * y);
    }
};
```

Function Taxonomy

- Generators
  - no argument, return a single value (e.g., `rand`)
  - for theoretical completeness (not really used)
- Unary Functions
  - one argument (any type), return a single value
- Binary Functions
  - two arguments (possibly distinct types), return a single value

Using A Generator

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

void print(int x)
{
    cout << x << ' ';
}

main()
{
    int a[10];
    generate_n(a, 10, rand);
    for_each(a, a+10, print);
}
```

unary_function

A Base Class to Store Types

```
template<class Arg, class Rtn>
struct unary_function
{
    typedef Arg argument_type;
    typedef Rtn result_type;
};
```

binary_function

A Base Class to Store Types

```
template<class Arg1, class Arg2, class Rtn>
struct binary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Rtn  result_type;
};
```

Adaptable Function Objects

- Function Objects that derive from `unary_function` or `binary_function`
- Standard *function object adapters* use argument and return types

An Adaptable **byname**

Version 1

```
struct byName
{
    typedef bool return_type;
    typedef Person argument_type;
    string name;
    byName(const string& s) : name(s) {}
    bool operator()(const Person& p)
    {
        return p.name == name;
    }
};
```

An Adaptable **byname**
Version 2

```
struct byName : unary_function<Person, bool>
{
    string name;
    byName(const string& s) : name(s) {}
    bool operator()(const Person& p)
    {
        return p.name == name;
    }
};
// argument_type == Person
// result_type == bool
```

- Template functions
- Return new function objects from existing ones
- Used to create sophisticated function objects on demand
- Rarely have to actually write a function yourself!

Standard Function Object Adapters

```
unary_negate<Pred> not1(const Pred& pr)
binary_negate<Pred> not2(const Pred& pr)

binder1st<BF> bind1st(const BF& pr, const T& x)
binder2nd<BF> bind2nd(const BF& pr, const T& x)

pointer_to_unary_function ptr_fun(Result (*)(Arg))
pointer_to_binary_function ptr_fun(Result (*)(Arg1, Arg2))

mem_fun_t<R, T> mem_fun(R (T::*pm)())
mem_fun1_t<R, T, A> mem_fun1(R (T::*pm)(A arg))
mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)())
mem_fun1_ref_t<R, T, A> mem_fun1_ref(R (T::*pm)(A arg))
```

Function Object Adapter

Example - **`not1`**

```
// Find first Person that is not "Charles"
Person* p = find_if(a, past,
                    not1(byName("Charles")));
```

Implementation of **`not1`**

```
template<class UF>
unary_negate<UF> not1(const UF& uf)
{
    return(unary_negate<UF>(uf));
}
```

Implementation of **unary_negate**

```
template<class UF>
class unary_negate
    : public unary_function<UF::argument_type, bool>
{
public:
    explicit unary_negate(const UF& uf) : fn(uf) {}
    bool operator()(const UF::argument_type& fn) const
    {
        return (!fn(uf));
    }
protected:
    UF fn;
};
// Assumes return type is bool
```

How `not1` *works in* `find_if`

In `find_if`, the expression:

```
if (p(*start))
```

becomes

```
if (not1(byname("Charles"))(*start))
```

which further reduces to

```
if (!byname("Charles")(*start))
```

and finally to

```
if (!((*start).name == "Charles"))
```

An Alternative to **`byName`**

```
Person v("Charles", 1897, 3, 11);
find_if(a, past, bind2nd(equal_to<Person>(), v));
```

- You don't even have to write a function!
- `equal_to` is a predefined *binary* function object
- `bind2nd` binds `v` to the 2nd parm of `equal_to`
- But this still compares *complete objects*, so...

Comparing on Name Only

```
struct byName : binary_function<Person, string, bool>
{
    bool operator()(const Person& p,
                    const string& s) const
    {
        return p.name == s;
    }
};
…

find_if(a, past, bind2nd(byName(), "Charles"));
```

The Best Approach Yet!

Using **`ptr_fun`**

```
bool byName(const Person& p, const string& s)
{
    return p.name == s;
}


find_if(a, past, bind2nd(ptr_fun(byName), "Charles"));
```

Analysis

- The compiler deduces that `byName` is a binary function
- `ptr_fun` returns a `pointer_to_binary_function` (let's call it `pbf`), which stores the address of `byName`
- `bind2nd` creates a `binder2nd` (call it `b2`), a *unary* function object that stores the value "Charles", as well `pbf`
- The expression `p(*start)` in `find_if` invokes `b2(*start)`
- `b2(*start)` is really `b2.operator()(*start)`, which invokes `pbf(*start,"Charles")`, which is really `byName(*start,"Charles")`

Run that by me Again!

`ptr_fun(byName)`

⇒`pbf` // binary, stores `&byName`

`bind2nd(pbf, "Charles")`

⇒`b2` // unary, stores "Charles", `pbf`

`b2` becomes the predicate `p` in `find_if`

Yeah, and Then What Happens?

```
p(*start)
⇒b2.operator()(*start)
⇒pbf.operator()(*start, “Charles”)
⇒byName(*start, “Charles”)
⇒return (*start).name == “Charles”
```

Whew!

- Yeah, but you get used to it.

What about Efficiency?

- Everything is resolved as *compile-time*
- Functions can be *inlined*
- So don’t worry about it!

Try this one...

- Write an expression with standard function objects and adapters that searches a sequence of integers for the first element that is not less than 100.

Words of Wisdom

"When you need a new concept, create a new class."

-- adapted from Andrew Koenig

Function Objects and Adapters Summary

- Function objects overload `operator()`
- Since they are objects, they can save data
  - allows on-the-fly "function definition"
- Standard function objects are class templates
  - act like generic functions
- Function object adapters transform one function object into another
  - allows convenient construction of predicates

Lab 3

Containers and Iterators

Containers

- Objects that hold other objects
- Support insertion, deletion of elements
- Support iteration (visit each element)
- The ubiquitous Array is a container
  - PRO: random access
  - CON: fixed size

Vector
A Dynamic Array

```
main()
{
    vector<string> tokens;
    string token;

    // Read tokens; fill vector
    while (cin >> token)
        tokens.push_back(token);

    int ntok = tokens.size();
    cout << "There are " << ntok << "tokens:\n";
    for (int i = 0; i < ntok; ++i)
        cout << i << ": " << tokens[i] << endl;
}
```

```
// Input:
how now brown cow

// Output:
There are 4 tokens:
0: how
1: now
2: brown
3: cow
```

Vector Properties

- A class template
  - can hold objects of any *concrete* type
    - copyable: `T(const T&)`
    - assignable: `T& operator=(const T&)`
- Provides `operator[]` for random access
- Manages memory for you
  - grows as required

Recursive Argument Lists
A Sample Application

- Like C's `argc/argv` mechanism
- Arguments beginning with '@' are files
- Such *indirect files* can be nested

Sample Input

`C:> prog arg1 @more_args arg2`

File **`more_args:`**

```
arg3
arg4
```

Equivalent to :

`c:> prog arg1 arg3 arg4 arg2`

Class **Arglist**

Uses a Vector

```
// arglist.h
#include <stddef.h>
#include <string>
#include <vector>
using namespace std;

class Arglist
{
    vector<string> args;

    void expand(char *);
    void add(char *)
    {
        args.push_back(arg);
    }
```

```
public:
    Arglist(size_t, char **);
    size_t count() const
    {
        return args.size();
    }
    const string& operator[](size_t) const
    {
        return args[i];
    }
};
```

```
// arglist.cpp
#include <fstream>
#include "arglist.h"
using namespace std;

Arglist::Arglist(size_t arg_count, char **arg_vec)
{
    // Build complete argument vector
    for (int i = 0; i < arg_count; ++i)
        if (arg_vec[i][0] == '@')
            expand(arg_vec[i]+1);
        else
            add(arg_vec[i]);
}
```

```
// arglist.cpp (cont.)
void Arglist::expand(char *fname)
{
    ifstream f(fname);
    string token;

    while (f >> token)
        if (token[0] == '@')
            expand(token+1);
        else
            add(token);
}
```

Using Arglist

```
// echo.cpp: Echoes command-line args,
// 1-per-line

#include <iostream>
#include "arglist.h"

main(int argc, char* argv[])
{
    Arglist args(--argc, ++argv);
    for (int i = 0; i < args.count(); ++i)
        cout << args[i] << endl;
}
```

All Containers...

- Are *homogeneous*
- Provide `insert` and `erase` capability
- Support the following methods:

```
size_type size() const;
size_type max_size() const;
bool empty() const;
```

Standard Containers

- Basic Sequences
  - `vector, deque, list`
- Container Adapters
  - `queue, stack, priority_queue`
- Associative Containers
  - `set, multiset, map, multimap`

Basic Sequences

- **`vector`**
  - random access
  - optimal insertion/deletion at end (`push_back`)
- **`deque`**
  - random access
  - optimal insertion/deletion at both ends (`push_front`)
- **`list`**
  - sequential access only
  - doubly-linked; optimal insertion/deletion anywhere

All Sequences support...

```
void resize(size_type, T = T());
T& front() const;
T& back() const;
void push_back(const T&);
void pop_back();
```

Restricted Sequence Functions

```
// deque and list only:
void push_front(const T&);
void pop_front();


//deque and vector only:
T& at(size_type n);
```

Nagging Questions

- How does one navigate a `list`?
  - It doesn't have any traversal member functions!
- How does on use the standard algorithms on a sequence?
  - Don't tell us that all the algorithms have to be repeated as member functions!
- The answer is…

Iterators

- Generalization of a pointer
- Overload at least `operator!=`, `operator==`, `operator*`, `operator++`, `operator->`
- Some overload `operator--`, `operator[]`

Traversing a List

```
list<T> lst;
…
// insert some elements, then:
list<T>::iterator p = lst.first();
while (p != lst.end())
{
    // Process current element (*p), then:
    ++p;
}
```

- All sequences provide members `first` and `end`

Implementing **`find`**

```
template<class Iterator, class T>
Iterator
find(Iterator begin, Iterator end, const T& v)
{
    while (begin != end)
    {
        if (*begin == v)
            break;
        ++end;
    }
    return end;
}
```

- All algorithms are implemented in terms of *iterators*

Iterator Taxonomy

- Input
- Output
- Forward
- Bi-directional
- Random Access

Input Iterators

- Read-only access to elements
- Single-pass, forward traversal
- `find` expects an Input Iterator

The Real *Implementation of* **`find`**
(Documentation change only)

```
template<class InputIterator, class T>
InputIterator
find(InputIterator begin, InputIterator end,
     const T& v)
{
    while (begin != end)
    {
        if (*begin == v)
            break;
        ++end;
    }
    return end;
}
```

Output Iterators

- Write-only access to elements
- Single-pass, forward traversal

Forward Iterators

- Both read and write access
  - can therefore substitute for Input or Output Iterators
- Multiple-pass forward traversal
- `unique` expects a Forward Iterator

```
list<T>::iterator p = unique(lst.first(),
                             lst.end());
```

Bi-directional Iterators

- Can do everything a Forward Iterator can
- Also support backwards traversal
  - `operator--()`
  - `operator--(int)`
- `reverse` requires a Bi-directional Iterator

Traversing a List Backwards

```
list::iterator p<T> = lst.end();
while (p > lst.begin())
{
    --p;        // "advances" backwards
    // process *p, then:
    if (p == lst.begin())
        break;
}
```

A Better Way
Reverse Iterators

```
list::reverse_iterator<T> p = lst.rbegin();
while (p != lst.rend())
{
    // process *p, then:
    ++p;        // "advances" backwards
}
```

Random Access Iterators

- Support Pointer Arithmetic in constant time
  - `operator+, +=, -, -=, [], <, <=, >, >=`
- `sort` expects a Random Access Iterator

How do you Sort a List?

- Doesn't provide a Random Access Iterator
- Generic `sort` will fail on a `list`
- Provides its own `sort` member function
- Also `merge`, `remove`, and `unique`

What's Wrong with this Picture?

```
vector<int> v1;
…
// fill v1, then:
vector<int> v2;
copy(v1.begin(), v1.end(), v2.begin());
```

Iterator Modes

- Iterators work in *overwrite* mode by default
- Need an *insert* mode for cases like above
  - that calls appropriate, underlying insert operation

Insert Iterators

- Replace output calls (`operator*`, `operator=`, etc.) with appropriate *insert* function
- `back_insert_iterator`
  - calls `push_back`
- `front_insert_iterator`
  - calls `push_front`
- `insert_iterator`
  - calls `insert`

Helper Functions

- `back_inserter`
  - creates a `back_insert_iterator`
- `front_inserter`
  - creates a `front_insert_iterator`
- `inserter`
  - creates an `insert_iterator`

Insert Iterator Example

```
vector<int> v1;
…
// fill v1, then:
vector<int> v2;
copy(v1.begin(), v1.end(), back_inserter(v2));
```

Stream Iterators

- `ostream_iterator`
  - an Output Iterator

```
copy(v1.begin(), v1.end(),
        ostream_iterator<int>(cout, " "));
```

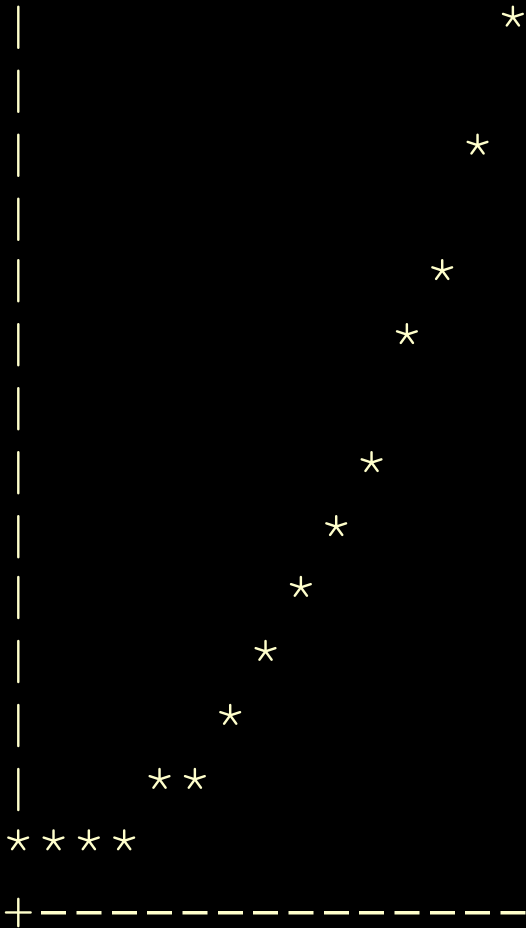
- `istream_iterator`
  - an Input Iterator

```
copy(istream_iterator<int>(cin),
        istream_iterator<int>(),
        back_inserter(v1));
```

Grid
A Sample Application

- Graphs `y = f(x)`
- Models the `x-y` plane as rows of characters
- In other words, a vector of vectors of char
- Illustrates `back_inserter`, reverse iterators, random acccess iterators

Grid Output



Grid
Source Code

```
// grid.cpp: Stores an x-y graph in a grid of chars
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

namespace
{
    typedef vector<char> row_type;
    typedef vector<row_type> grid_type;
    const int xmax = 15;          // # of columns
    const int ymax = xmax;        // # of rows
}
```

```
main()
{
    void print_grid(const grid_type&);
    double f(double);         // Function to graph
    grid_type grid;           // A vector of rows

    // Draw y-axis and clear 1st quadrant:
    grid.reserve(ymax);
    row_type blank_row(xmax);
    blank_row[0] = '|';
    for (int y = 0; y < ymax; ++y)
        grid.push_back(blank_row);

    // Draw x-axis:
    grid[0][0] = '+';
    for (int x = 1; x < xmax; ++x)
        grid[0][x] = '-';
```

```
    // Populate with points of f():
    for (int x = 0; x < xmax; ++x)
        grid[f(x)][x] = '*';     // row-oriented

    print_grid(grid);
}

double f(double x)
{
    // NOTE: Contrived to fix within grid!
    return x * x / ymax + 1.0;
}
```

```
void print_grid(const grid_type& grid)
{
    grid_type::const_reverse_iterator yp;
    for (yp = grid.rbegin(); yp != grid.rend(); ++yp)
    {
        // Print a row:
        copy(yp->begin(), yp->end(),
             ostream_iterator<char>(cout, ""));
        cout << endl;
    }
}
```

Container Adapters

- High-level abstract data types
  - `queue, stack, priority_queue`
- Use a sequence for implementation

```
stack< string, vector<string> > myStack;
```

- `stack` & `queue` use `deque` by default
- `priority_queue` uses a `vector`
- No iterators are provided
  - more restricted interface

Associative Containers

- `set`
  - stores unique elements
  - test for membership
  - `multiset` allows duplicates
- `map`
  - stores <key, value> pairs
  - keys must be unique
  - `multimap` allows duplicate keys
- Support fast (logarithmic), key-based retrieval
- Stored according to an ordering function
  - `less<T>()` by default
  - can use as a sequence

Set Example

```
#include <iostream>
#include <set>
#include <string>
using namespace std;

void main()
{
    // Populate a set:
    set<string> s;
    s.insert("Alabama");
    s.insert("Georgia");
    s.insert("Tennessee");
    s.insert("Tennessee");
```

```
    // Print it out:
    set<string>::iterator p = s.begin();
    while (p != s.end())
        cout << *p++ << endl;
    cout << endl;

    // Do some searches:
    string key = "Alabama";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
         << key << endl;

    key = "Michigan";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
         << key << endl;
}
```

```
// Output:
Alabama
Georgia
Tennessee

found Alabama
didn't find Michigan
```

Map Example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

void main()
{
    // Convenient typedefs:
    typedef map<string, string, greater<string>()>
            map_type;
    typedef map_type::value_type element_type;
```

```
// Insert some elements (two ways):
map_type m;
m.insert(element_type(string("Alabama"),
         string("Montgomery")));
m["Georgia"] = "Atlanta";
m["Tennessee"] = "Nashville";
m["Tennessee"] = "Knoxville";

// Print the map:
map_type::iterator p = m.begin();
while (p != m.end())
{
    element_type elem = *p++;
    cout << '{' << elem.first << ','
         << elem.second << "}\n";
}
cout << endl;
```

```
    // Retrieve via a key:
    cout << '"' << m["Georgia"] << '"' << endl;
    cout << '"' << m["Texas"] << '"' << endl;
}

// Output:
{Tennessee,Knoxville}
{Georgia,Atlanta}
{Alabama,Montgomery}

"Atlanta”
""
```

Words of Wisdom

"It is this arrangement of containment, iteration, and algorithms into separate concepts, coupled with the flexibility of templates, which makes the standard C++ library stand alone as the most powerful, general-purpose library ever conceived."

-- Me!

C/C++ Code Capsules, A Guide for the Practitioner, Prentice-Hall, 1998.

Containers & Iterators Summary

- **Containers** and their objects have *value semantics* (i.e., they are *concrete types*)
  - basic sequences: `vector, deque, list`
  - adapters: `queue, stack, priority_queue`
  - associative: `set, multiset, map, multimap`
- **Iterators** act like *pointers* into a container
  - 5 flavors: Input, Output, Forward, Bi-directional, Random Access
  - the *glue* between containers and algorithms

Lab 4

Applications

File Inclusion

- Like `#include`
- No limit on nesting depth
- Won't recursively process an open file
- How to track open files?
- What data structure?

include.cpp

```
// include.cpp: Nested file inclusion
#include <iostream>
#include <fstream>
#include <sstream>
#include <set>
#include <string>
using namespace std;

// List of active filenames:
set<string> files;

void include(const string&);
```

```
main(int argc, char* argv[])
{
    if (argc > 1)
        try
        {
            include(argv[1]);
        }
        catch (string& s)
        {
            cerr << s << endl;
        }
}
```

```
void include(const string& fname)
{
    // Open file; add to list:
    ifstream f(fname.c_str());
    if (!f)
        throw string("error opening file: ") + fname;
    files.insert(fname);

    // Process file:
    string line;
    while (getline(f, line, '\n'))
    {
        // Inspect first token:
        istringstream is(line);
        string word;
        is >> word;
```

```
        if (word == "#include")
        {
            // Attempt to include nested file:
            string nested;
            is >> nested;
            if (files.find(nested) == files.end())
                include(nested);
            // else file is ignored
        }
        else
            cout << line << endl;
    }

    // Remove filename from list:
    files.erase(fname);
}
```

Cross-reference Lister

- Lists each word together with the numbers of the lines where it appears in the file:

```
a        1    10    12
an       22
```

  (etc.)

- Words appear in alphabetical order, without duplication
- Associated numbers appear in ascending order, without duplication
- What data structure(s) will do the job?

xref.cpp

```
// xref.cpp:  Prints line #'s for each word
#include <iostream>
#include <iomanip>
#include <string>
#include <set>
#include <map>
using namespace std;

namespace        // same as file static
{
    typedef set<int> val_type;
    typedef map<string, val_type> map_type;
    const int WORD_WIDTH = 15;
}
```

```
main()
{
    // Utilities:
    string next_token(const string&, int&);
    void print_list(const val_type&);

    map_type m;          // the map
    string line;
    int lineno = 0;

    // Read each line:
    while (getline(cin,line,'\n'))
    {
        ++lineno;
```

```
    // Process each word in line:
    int pos = 0;
    string token;
    while (!(token = next_token(line,pos)).empty())
    {
        // Populate the map:
        m.insert(make_pair(token,val_type()));
        m[token].insert(lineno);
    }
}       // end outer while
```

```
    // Print results:
    cout << "No. of distinct words: ”
         << m.size() << endl;
    cout.setf(ios::right, ios::adjustfield);
    for (map_type::const_iterator p = m.begin();
         p != m.end();
         ++p)
    {
        cout << setw(WORD_WIDTH) << p->first
             << setw(0) << ": ";
        print_list(p->second);
        cout << endl;
    }
}     // end main
```

Duplicate Line Filter

- Like UNIX's `uniq`
- Except it processes *unsorted* files
- Need a way to detect duplicate lines without disturbing original line order

Sample Data

Input

```
each
peach
pear
plum
i
spy
tom
thumb
```
tom
thumb
```
in
the
cupboard
```
i
spy
```
mother
hubbard
```

Output

```
each
peach
pear
plum
i
spy
tom
thumb
in
the
cupboard
mother
hubbard
```

uniq2

- What data structure(s)?
  - Why won't a `set` suffice?
- Accesses lines through an *index*
- Sorts the index based on line content, then removes adjacent duplicates with the unique algorithm
- Re-sorts the surviving indexes numerically to adhere to original order

uniq2.cpp

```
(#includes omitted to save space)

namespace
{
    vector<string> lines;

    // Sort Predicates:
    bool less_by_idx(int a, int b)
    {
        return lines[a] < lines[b];
    }

    bool equal_by_idx(int a, int b)
    {
        return lines[a] == lines[b];
    }
}
```

```
main()
{
    vector<int> idx;

    // Read lines into memory:
    string line;
    int nlines = 0;
    for ( ; getline(cin, line, '\n'); ++nlines)
    {
        lines.push_back(line);
        idx.push_back(nlines);  // Identity index
    }
```

```
    stable_sort(idx.begin(), idx.end(), less_by_idx);

    // Remove indexes to duplicate lines:
    vector<int>::iterator uniq_end =
        unique(idx.begin(), idx.end(), equal_by_idx);

    // Restore correct order of remaining lines:
    sort(idx.begin(), uniq_end);

    // Output unique lines:
    int nuniq = uniq_end - idx.begin();
    for (int i = 0; i < nuniq; ++i)
        cout << lines[idx[i]] << endl;
}
```

```
if (now == practical_stl.end())
    exit(SUCCESS);
```