

# Two-days lecture on the STL

*Masa*

Thierry Géraud

URL <http://www.lrde.epita.fr/people/thierry.geraud>

Email [thierry.geraud@lrde.epita.fr](mailto:thierry.geraud@lrde.epita.fr)



(c) EPITA Research and Development Laboratory, 2002.

14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre cedex

Tél. +33 1 53 14 59 47, Fax +33 1 53 14 59 22

# Outline

- First Part:
  - Genericity v. Inheritance 3-5
  - C++ Generic Features 6-10
  - Generic Algorithms 11-15
  - Understanding STL 16-18
- Second Part:
  - Function Objects 19-21
  - Defining an Ordered Pair 22-24
  - Adapting Data to STL Containers 25-28
  - Traits 29-31
  - Quick Tour 32-35
- Third Part:
  - Container Concepts 36-51
  - Practising Containers 52-56
  - Predicates and Iterators 57-58
  - Algorithms 59-61
  - Use Cases 62-64
- Fourth Part:
  - Predifined Function Objects 65
  - Adaptators 66-69
  - Template Meta-Programs 70-73
  - References 74-75

# Genericity v. Inheritance

1/3

“Classical” OO program

```
#include <iostream>

class A {                                // an abstract class
public:
    virtual void m() = 0;                // a polymorphic method
    //...
};

class C1 : public A {                    // a concrete class
public:                                   // inheriting from the former
    virtual void m() {
        std::cout << "C1::m" << std::endl;
    }
};

class C2 : public A {                    // another concrete class
public:
    virtual void m() {
        std::cout << "C2::m" << std::endl;
    }
};

void foo(A& arg) {                        // say, foo is general
    arg.m();                              // binding this call to a method
}                                          // has a (negligible?) cost

int main() {
    C1 c;
    foo(c);
}
```

# Genericity v. Inheritance

2/3

Equivalent “generic” program

```
#include <iostream>

class C1 {                                // a concrete class
public:                                    // without inheritance
    void m() {
        std::cout << "C1::m" << std::endl;
    }
};

class C2 {                                // another concrete class
public:
    void m() {
        std::cout << "C2::m" << std::endl;
    }
};

template<class T>
void foo(T& arg) {                        // foo is generic
    arg.m();                              // this call can be inlined!
}

int main() {
    C1 c;
    foo(c);
}
```

# Genericity v. Inheritance

3/3

Genericity:

- use of the `template` keyword in C++
- types should be known at compile-time (this feature is also a limitation)
- no inheritance is required
- programs are efficient

Severe limitation = `foo`'s signature is not restrictive at all:

- we have: `' $\forall$  type T, void foo(T&)'`
- we would have preferred:  
`' $\forall$  type T with { void m(); }, void foo(T&)'`

Definitions:

A *concept* is a set of requirements over types.

for instance, **myconcept** = { void m(); }

A *refinement* of a concept is a concept with more requirements than the primary one.

for instance, **otherconcept** = { void m(); void n(); } is a refinement of **myconcept**

# C++ Generic Features

1/5

Functions can be generic:

```
template<typename T>
T sqr(T val) {
    return val * val;
}

int main() {
    int i = 2, j;
    j = sqr(i);           // calls sqr<int>

    float f = 3.14, g;
    g = sqr(f);          // calls sqr<float>

    g = sqr<float>(i);    // explicitly calls sqr<float>
}
```

- The compiler instantiates as many versions of the generic function as it is required.
- Calls are usually implicit; the compiler deduces the parameters' types from the types of function arguments.

# C++ Generic Features

2/5

Classes can be generic:

```
template<unsigned dim, class T>
class vec {          // within this class, 'vec' means 'vec<dim,T>'
public:
    vec() {}
    vec(T val) {
        for (unsigned i = 0; i < dim; ++i)
            data[i] = val;
    }
    T operator[](unsigned i) const { return data[i]; }
    T& operator[](unsigned i)      { return data[i]; }
    vec operator+(const vec& rhs) const {
        vec tmp = *this;
        for (unsigned i = 0; i < dim; ++i)
            tmp[i] += rhs[i];
        return tmp;
    }
private:
    T data[dim];
};
```

Sample use:

```
typedef vec<3,float> RGB;
typedef vec<2,int>   Point2D;

int main() {
    RGB color;
    Point2D p, p2;
    void(p + p2);    // correct
    void(color + p); // error at compile-time
}
```

# C++ Generic Features

3/5

Classes and functions can be *specialized*.

For instance, this particular class...

```
template<class T>
class vec<0,T> {                               // 0 means 'unknown size'
public:
    vec(unsigned dim) { data = new T[dim]; }
    ~vec() { delete[] data; }
    // ...
    T operator[](unsigned i) const { return data[i]; }
    T& operator[](unsigned i)      { return data[i]; }
    // ...
private:
    T* data;
};
```

...replaces the global definition of `vec<dim,T>`.

Now, an example involving function specialization:

```
template<int a> inline
double f(double x) {
    return sin(a * x);
}

template<> inline
double f<0>(double) {
    return 0;
}

int main() {
    double d = f<3>(16);
}
```

# C++ Generic Features

4/5

Class parameters can have default values; e.g.:

```
template<class T1 = void*, class T2 = T1>
struct my_pair {
    T1 first;
    T2 second;
};
```

```
my_pair<Point2D>      means    my_pair<Point2D,Point2D>
my_pair<Point2D*>    means    my_pair<Point2D*,Point2D*>
my_pair<>             means    my_pair<void*,void*>
```

Parameters can be meta-entities:

```
template<class T,
         template<class, class> class P = std::pair>
class value {
public:
    // ...
    bool is_ok() const { return data.first; }
    T    get()    const { return data.second; }
private:
    P<bool,T> data;
};

int main() {
    value<float,my_pair> val;
    // ...
    if (val.is_ok()) cout << val.get(); else cout << '?';
}
```

# C++ Generic Features

5/5

Generic entities are more powerful than macros!

```
template<class T>
struct plain_type {
    typedef T ret;
};

template<class T>
struct plain_type<T*> {
    typedef typename plain_type<T>::ret ret;
};

int main() {
    typedef int** a_type;
    plain_type<a_type>::ret i = 51;
    // compilation gives "warning: unused variable 'int i'"
}
```

- Templates can be recursive.
- Templated code is handled by the compiler.  
An error can then be pointed out at a precise code line.
- Genericity leads to stronger typed programs.
- Macros' side-effects are no longer a problem.

```
#define SQR(a) ((a)*(a))
int i = 1, j = SQR(++i);
```

# Understanding Generic

## Algorithms

1/5 — data genericity

*Nota bene* : “Data genericity”, presented here, is the most common use of genericity.

The following code is generic *vis-a-vis* image data:

```
template<class T>
class image2d {
    // ...
};

template<class T>
void println(const image2d<T>& input) {
    for (unsigned irow = 0; irow < input.nrows(); ++irow)
        for (unsigned icol = 0; icol < input.ncols(); ++icol)
            cout << input(irow, icol) << ' ';
    cout << endl;
};
```

but is dedicated (so restricted) to 2D images.

*Drawback*: `println` is **not** generic *vis-a-vis* image structures.

# Understanding Generic

## Algorithms

2/5 — OO iterators

Particular objects, *iterators*, are defined to browse structures. Here, `image2d_const_iterator` is dedicated to 2D images.

```
template<class T>
struct const_iterator { // abstract class
    virtual T operator*() = 0;
    virtual void operator++() = 0;
    virtual void begin() = 0;
    virtual bool end() = 0;
};

template<class T>
class image2d_const_iterator : public const_iterator<T> {
public:
    image2d_const_iterator(const image2d<T>& ima) :
        ima(ima) {
    }
    virtual T operator*() {
        return ima(irow, icol);
    }
    virtual void operator++() {
        if (++icol == ima.ncols()) { icol = 0; ++irow; }
    }
    virtual void begin() {
        irow = icol = 0;
    }
    virtual bool end() {
        return irow == ima.nrows();
    }
private:
    const image2d<T>& ima;
    unsigned irow, icol;
};
```

# Understanding Generic

## Algorithms

3/5 — OO structures

The following code is now generic *vis-a-vis* both image data and image structures:

```
template<class T>
struct image { // abstract class
    virtual const_iterator<T>& create_const_iterator() const = 0;
    // ...
};

template<class T>
class image2d : public image<T> {
public:
    virtual image2d_const_iterator<T>& create_const_iterator() const {
        return *new image2d_const_iterator<T>(*this);
    }
    // ...
};

template<class T>
void println(const image<T>& input) {
    const_iterator<T>& i = input.create_const_iterator();
    for (i.begin(); ! i.end(); ++i) // this loop mimics how
        cout << *i << ' '; // buffer are browsed :)
    cout << endl;
};
```

*Drawback:* within the loop, three calls are due to virtual methods and thus have a cost at run-time that we do **not** want to pay. We want efficient algorithms.

# Understanding Generic Algorithms

4/5

— iterators and structures w/o abstract classes

First, let us forget our abstract classes:

```
template<class T>
class image2d_const_iterator
{
public:
    image2d_const_iterator(const image2d<T>* ima,
                          unsigned irow, unsigned icol) :
        ima(ima), irow(irow), icol(icol) {
    }
    T operator*() { // no more virtual
        return (*ima)(irow, icol);
    }
    void operator++() { // no more virtual
        if (++icol == ima->ncols()) { icol = 0; ++irow; }
    }
    // ...
};

template<class T>
class image2d {
public:
    typedef image2d_const_iterator<T> const_iterator;
    const_iterator begin() const { // begin() and end()
        return const_iterator(this, 0, 0); // have moved from
    } // iterator classes
    const_iterator end() const { // to structure
        return const_iterator(this, nrows, 0); // ones
    }
    // ...
};
```

# Understanding Generic

## Algorithms

5/5 — full genericity

Then, the following code is *fully generic and efficient*:

```
template<class I>
void println(const I& input) {
    typename I::const_iterator i;
    for (i = input.begin(); i != input.end(); ++i)
        cout << *i << ' ';
    cout << endl;
};

int main() {
    image2d<float> ima1;    image3d<RGB> ima2;
    println(ima1);        println(ima2);
}
```

Moreover:

- `println` also works on STL containers!

```
int main() {
    std::vector<int> v; // ...
    println(v);
}
```

- `image2d` can also use STL algorithms!

```
int main() {
    image2d<float> ima;
    std::vector<int> v(ima.size()); // ...
    std::copy(ima.begin(), ima.end(), v.begin());
}
```

# What's worth remembering?

- Genericity is not related to inheritance and *inclusion* polymorphism, i.e., to the use of sub-classing and of virtual methods.

Genericity is called *parametric* polymorphism.

- Algorithms can be fully generic *vis-a-vis* their input while remaining efficient.
- STL describes concepts; classes that conform to these concepts are called models.

and:

- We can define our own algorithms and apply them on STL containers.
- We can define our own structure types (containers) and take advantage from STL algorithms.

Actually:

- the classes `image2d<T>` and `image2d_const_iterator<T>` are *models* of the STL **Sequence** and **InputIterator** *concepts*,
- and `println` is written following the STL style!

however :

- The most common use of STL is to take benefits from containers and algorithms to handle our data types (e.g., `Point2D`).

# STL and std

The *Standard Template Library* (STL for short):

- is a code library of *containers*, *algorithms*, and related tools such as *iterators*,
- was first written by Alexander Stepanov,
- has been adopted as part of the ANSI/ISO C++ standard,
- is now widely available through several high-quality versions.

The *C++ Standard Library*:

- includes most of STL classes,
- features much more tools,
- is located in the `std` namespace.

# Containers

## in a design process

Classes usually are related together. For instance:

```
class B {
    // ...
};

class A {
    // ...
private:
    std::list<B*> bs;
};
```

Every instance of A, through `bs`, encloses pointers to instances of B.

For that, one must choose a proper *container* whose type depends on the properties we want when we handle its data.

Here, `std::list` means that:

- we do not need a random access to data,
- we do not need to keep data sorted,
- we can have constant time front insertion,
- and so on.

# Function Objects 1/3 — 1st Attempts

Consider a function as a parameter:

```
double sqr(double x) {
    return x * x;
}

template<double (*fun)(double)>
void print(double x) {
    cout << x << " -> " << fun(x) << endl;
}

int main() {
    double x = 3.14;
    print<sqr>(x);
}
```

Trying to be generic, code becomes like obfuscated:

```
template<class T>
T sqr(T x) {    // we can have now sqr<int>, sqr<float>, and so on
    return x * x;
}

template<class T, T (*fun)(T)>
void print(T x) {
    cout << x << " -> " << fun(x) << endl;
}

int main() {
    double x = 3.14;
    print<double, sqr<double> >(x);
}
```

Argh...

# Function Objects

2/3 — Soluce

Imagine that `sqr` is a type whose instances behave like functions:

```
struct sqr {
    template<class T>
    T operator()(T x) const {
        return x * x;
    }
};

template<class Fun, class T>
void print(Fun fun, T x) {
    cout << x << " -> " << fun(x) << endl;
}

int main() {
    double x = 3.14;
    print(sqr(), x);
    print(sqrt, x);
}
```

Please note that `print` works both on function objects *and* functions.

So change your mind!

```
float line(float x) { return 2 * x + 1; }
```

can be advantageously rewritten into:

```
template<class Coef>
struct line
{
    Coef a, b;
    line(Coef a, Coef b) : a(a), b(b) {}
    template<class T>
    T operator()(T x) const { return a * x + b; }
};
```

# Function Objects

3/3 — Test

How does this program work?

```
template<class T>
struct objfun {
    const T& self() const { return static_cast<const T&>(*this); }
};

struct power : public objfun<power>
{
    unsigned p;
    power(unsigned p) : p(p) { }
    float operator()(float x) const { return pow(x, p); }
    string name() const {
        stringstream s;
        s << "power[" << p << ']';
        return s.str();
    }
};

struct name {
    template<class T>
    static string of(T*) { return "<unknown>"; }
    template<class T>
    static string of(const objfun<T>& t) { return t.self().name(); }
};

template<class Fun, class T>
void print(Fun fun, T x) {
    cout << name::of(fun) << '(' << x << ")=" << fun(x) << endl;
}

int main() {
    power sqr(2);
    print(sqr, 3.14); // gives 'power[2](3.14)=9.8596'
    print(sqrt, 3.14); // gives '<unknown>(3.14)=1.772'
}
```

# Defining An Ordered Pair

## 1/3 — Class Definition

Element ordering is handled by an object function.

```
#include <utility> // for std::pair

#include <string>
using std::string; // 'string' now means 'std::string'

#include <iostream>
using std::ostream;
using std::cout;
using std::endl;

template<class T, class Cmp>
class OrdPair {
public:
    OrdPair() {} // no default ctor would have been created
    OrdPair(const T& first_, const T& second_) {
        Cmp cmp;
        _p = cmp(first_, second_) ?
            pair_t(first_, second_) :
            pair_t(second_, first_);
    }
    // OrdPair(const OrdPair&) and operator=(const OrdPair&)
    // do not seem to be defined...

    T first() const { return _p.first; }
    T second() const { return _p.second; }

    template<class T2, class Cmp2>
    bool operator==(const OrdPair<T2,Cmp2>& rhs) const;

private:
    typedef std::pair<T,T> pair_t;
    pair_t _p;
};
```

# Defining An Ordered Pair

## 2/3 — Extra Code and Sample Use

A method of a generic class can also feature its own parameters:

```
template<class T, class Cmp>
template<class T2, class Cmp2> inline
bool OrdPair<T,Cmp>::operator==(const OrdPair<T2,Cmp2>& rhs) const {
    return _p.first == rhs.first() && _p.second == rhs.second();
}
```

A generic class transmits its parameters to procedures:

```
template<class T, class Cmp> inline
ostream& operator<<(ostream& ostr, const OrdPair<T,Cmp>& p) {
    return ostr << p.first() << ' ' << p.second();
}
```

Finally, we have:

```
struct string_cmp {
    bool operator()(const string& lhs, const string& rhs) const {
        return lhs < rhs;
    }
};

int main() {
    string a = "xyz", b = "abc";
    OrdPair<string,string_cmp>
        p(a, b),
        p2(p), // default cpy ctor
        p3 = p, // default cpy ctor
        p4;
    p4 = p; // default assign op
    cout << (p3 == p2 ? "true" : "false") << endl;
}
```

# Defining An Ordered Pair

## 3/3 — Remarks and Rules

- An ordering object function has to be defined for `std::string` and that's a pity since:
  - string ordering truly sounds reasonable,
  - C++ favors definitions having a default behavior.
- The default copy constructor of `OrdPair` is valid because:
  1. `std::pair` owns a copy constructor,
  2. this latter constructor is based on the copy constructor of `std::string`,
  3. `std::string` copy constructor actually exists.

(Thus, an error might occur in a piece of code being *far* from the one which is responsible for compilation!)
- The comparison operator relies on `T == T2`;
  - since a comparison *is* used, the compiler checks that `string::operator==(string)` exists or can be instantiated,
  - if this comparison was not used, there is no need for `string::operator==(string)` to exist. (when some generic code is not used, only a syntax check is performed)

# Adapting Client Data to STL containers

1/4 — Our Data

A record (`Person::record`) *maps* an id to a person.

```
typedef unsigned id_t;

struct Person
{
    // start of object data
    const id_t id;
    string name;

    Person(const string& name) : id(++cur_id), name(name) {
        record[id] = this;
    }
    ~Person() {
        record.erase(id);
    }

    bool operator<(const Person& rhs) const {
        return this->id < rhs.id;
    }

    // start of class data
    static id_t cur_id;

    typedef std::map<const id_t, Person*> record_t;
    static record_t record;

    static Person* fetch(id_t id) {
        record_t::iterator i = record.find(id);
        return i == record.end() ? 0 : i->second;
    }

    // ...
}
```

Question: what about C++ default methods here?

# Adapting Client Data to STL containers

2/4 — Our Problem

```
// w/o impl:

Person(const Person&);
Person& operator=(const Person&);

}; // end of class Person

Person::record_t Person::record;
id_t              Person::cur_id = 0;

ostream& operator<<(ostream& ostr, const Person& p) {
    return ostr << '(' << p.id << ', ' << p.name << ')';
}

int main() {
    Person me("theo"),
           al("alexander"),
           su("root"),
           *p = Person::fetch(3);
    if (p != 0)
        cout << *p << endl;
}
```

`Person::record` stores persons with increasing ids.

Now, how to build an address book whose entries are sorted by name then id?

We cannot use the data type `std::set<Person*>` since it stores pointers... with increasing values!

# Adapting Client Data to STL containers

3/4 — Our Aim

We want:

- a “name then id” sort when `Person*` is the sorting key type (it is *our* default rule),
- to disable this default rule for some particular cases.

Let us have a brief look to STL bowels:

```
template<class T>
struct less {
    bool operator()(const T& lhs, const T& rhs) const {
        return lhs < rhs;
    }
};

namespace std {
    template< class T, class Cmp = less<T> >
    class set {
        // ...
        void xxx() { // a method that needs to compare two Ts:
            if (cmp(t1, t2)) // ...
        }
    private:
        Cmp cmp;
    };
}
```

`std::set` uses an object function to keep data ordered. Its type is a class parameter (`Cmp`) whose default value is `less<T>`. The behavior of the default object function is to rely on the “<” operator of class `T`.

# Adapting Client Data to STL containers

4/4 — Solutions

While using `std::set<Person*>`, specialize the default object function for `Person*`, following either:

```
template<>
struct less<Person*> {
    bool operator()(const Person* lhs, const Person* rhs) const {
        return lhs->name < rhs->name ||
            (rhs->name == lhs->name && lhs->id < rhs->id);
    }
};
```

Or:

```
struct less_name_id { // this struct factors generic code...
    template<class T>
    bool operator()(const T* lhs, const T* rhs) const {
        return lhs->name < rhs->name ||
            (rhs->name == lhs->name && lhs->id < rhs->id);
    }
};
template<> // ...and inheritance does the job
struct less<Person*> : public less_name_id {
};
```

Alternatively, define a particular object function type:

```
struct cmp_name_id {
    template<class T>
    bool operator()(const T* lhs, const T* rhs) const {
        return lhs->name < rhs->name ||
            (rhs->name == lhs->name && lhs->id < rhs->id);
    }
};
```

and use the data type `set<Person*, cmp_name_id>` which disables the comparison with `less`.

# Traits

## 1/3 — State of the Problem

We want to define a generic addition for our previously defined vector class:

```
template<unsigned dim, class T, class U>
vec<dim,?> operator+(const vec<dim,T>& lhs,
                    const vec<dim,U>& rhs) {
    vec<dim,?> tmp;
    for (unsigned i = 0; i < dim; ++i)
        tmp[i] = lhs[i] + rhs[i];
    return tmp;
}
```

The problem is to replace “?” by a proper type.

We only know that:

float	+	float	should give	float
float	+	int	should give	float
int	+	float	should give	float
int	+	int	should give	int
int	+	unsigned	should give	int

etc.

When we want a set of properties that depends on types, we use *traits*.

# Traits

2/3 — Solution

From both types  $T$  and  $U$ , we can deduce an addition result type by the means of a typedef enclosed in a dedicated structure:

```
template<class T, class U>
struct plus_traits {          // structure declaration
};

template<class T>
struct plus_traits<T,T> {    // specialization: T + T gives T
    typedef T ret;
};

// particular specializations:

template<> struct plus_traits<float,int>      { typedef float ret; };
template<> struct plus_traits<int,float>      { typedef float ret; };
template<> struct plus_traits<int,unsigned>  { typedef int  ret; };
// ...
```

and if we need recursive constructs like `vec< n, vec<m,T> >`, we can define:

```
template<unsigned dim, class T, class U>
struct plus_traits< vec<dim,T>, vec<dim,U> > {
    typedef  vec< dim, typename plus_traits<T,U>::ret >  ret;
};
```

# Traits

3/3 — About STL

STL also defines traits.

E.g., in `stl_iterator.h`:

```
template <class Iterator>
struct iterator_traits {
    typedef typename Iterator::iterator_category iterator_category;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
};
```

```
template <class T>
struct iterator_traits<T*> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

Or in `std/traits.h`:

```
template <class charT>
struct string_char_traits {
    // ...
};

// quoted from std/bastring.h
// template <class charT,
//         class traits = string_char_traits<charT>,
//         class Allocator = alloc >
// class basic_string {
//     // ...
// };
```

Question: what is it for?

# Quick Tour of STL containers

1/2

Basic containers:

<code>vector&lt;T&gt;</code>	dynamic array,
<code>deque&lt;T&gt;</code>	double-ended queue,
<code>list&lt;T&gt;</code>	doubly-linked list,
<code>set&lt;T&gt;</code>	mathematical set,
<code>map&lt;Key,Value&gt;</code>	dictionary (or associative array).

Variations of set:

`multiset<T>`,  
`hash_set<T>`, `hash_multiset<T>`.

Variations of map:

`multimap<Key,Value>`,  
`hash_map<Key,Value>`, `hash_multimap<Key,Value>`

`hash_*` means that a *hash function* applied on keys is used to store and retrieve values; *Warning*: these explicitly hashed containers do not belong to `std`.

`multi*` means that several values can share the same key.

# Quick Tour of STL containers

2/2

Last, some containers are defined as adaptations of basic ones (by default, they are built from `deque`):

<code>stack&lt;T&gt;</code>	last-in first-out structure (LIFO),
<code>queue&lt;T&gt;</code>	first-in first-out structure (FIFO), also known as 'heap',
<code>priority_queue&lt;T&gt;</code>	priority queue.

## Important warning

Several other tools are extensions (from SGI); they are *not part of the C++ standard!*

Let us mention:

<code>slist&lt;T&gt;</code>	singly-linked list,
<code>hash&lt;T&gt;</code>	hash function,
<code>hash_*set&lt;T&gt;</code>	hashed sets,
<code>hash_*map&lt;K,V&gt;</code>	hashed maps,
<code>rope&lt;T&gt;</code>	efficient scalable string.

So, *just forget them* when you write portable code if you do not want to provide the files they are defined in!

# Today's conclusion

1/2

Consider the following programs.

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

int main() {
    string s;
    list<string> l;
    while (getline(cin, s))
        l.push_front(s);
    l.sort();
    for (list<string>::const_iterator i = l.begin(); i != l.end(); ++i)
        cout << *i << endl;
}
```

and:

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

struct println {
    template<class T>
    void operator()(const T& t) const {
        cout << t << endl;
    }
};

int main() {
    int a[] = {1, 2, 3};
    for_each(a, a + sizeof(a) / sizeof(int), println());
    // with l, 'for_each(l.begin(), l.end(), println())' is still ok
}
```

Question: what are the concepts behind this code?

# Today's conclusion

2/2

Now, enjoy this one.

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

int main()
{
    list<string> l;
    copy(istream_iterator<string>(cin),
        istream_iterator<string>(),
        back_inserter(l));
    l.sort();
    copy(l.begin(), l.end(), ostream_iterator<string>(cout, "\n"));
}
```

- What about expressiveness?
- What about efficiency?
- What about std tools?

# Basic Concepts

*concept*

**Assignable**

**Default Constructible**

**Equality Comparable**

**LessThan Comparable**

**Strict Weakly Comparable**

*requirement*

assignment operator  
and copy constructor

constructor w/o argument  
(by default or user defined)

equality operator

$<$ ,  $<=$ ,  $>$ , and  $>=$  operators  
defining partial ordering

refinement of  
**LessThan Comparable**  
defining equivalence classes

# Container Concepts

1/15

## **Container (Input Iterator)**

→ *object that stores elements*

- - - - - is refined into - - - - -

## **Forward Container (Forward Iterator)**

→ *elements are arranged  
in a definite order*

- - - - - is refined into - - - - -

## **Reversible Container (Bidirectional Iterator)**

→ *elements are browsable  
in a reverse order*

- - - - - is refined into - - - - -

## **Random Access Container (Random Access Iterator)**

→ *elements are retrievable w/o browsing  
(amortized constant time access to arbitrary elements)*

# Container Concepts

2/15

## Forward Container

----- is refined into -----

### Sequence

→ *variable-sized container with elements in a strict linear order*

----- is refined into -----

### Front Insertion Sequence

→ *first element access in amortized constant time*

### Back Insertion Sequence

→ *last element access in amortized constant time*

# Container Concepts

3/15

## Forward Container

----- is refined into -----

## Associative Container

→ *element retrieving is based on key*

----- is refined into -----

### **Simple Associative Container**

→ *elements are their own  
keys*

### **Pair Associative Container**

→ *elements are  
(key,value) pairs*

and/or

### **Unique Associative Container**

→ *each key is unique*

### **Multiple Associative Container**

→ *several elements can  
have the same key*

and/or

### **Sorted Associative Container**

→ *elements are sorted in  
ascending order by key*

### **Hashed Associative Container**

→ *a hash table stores  
elements; a hash function  
based on key does not  
imply a meaningful order!*

# Container Concepts

4/15

Their models from std:

<b>Forward Containers</b>	all
<b>Reversible Containers</b>	vector, list, set...
<b>Random Access Containers</b>	vector, deque, hash_*
<b>Front Insertion Sequences</b>	list, deque
<b>Back Insertion Sequences</b>	list, deque (remember, slist is not std!)
<b>Associative Containers</b>	set-based, map-based
<b>Unique Associative C.</b>	[hash_]set, [hash_]map
<b>Multiple Associative C.</b>	*multi*
<b>Simple Associative C.</b>	set-based
<b>Pair Associative C.</b>	map-based
<b>Sorted Associative C.</b>	[multi]set, [multi]map
<b>Hashed Associative C.</b>	hash_*

# Container Concepts

5/15

a.c.t stands for “amortized constant time”.

## Container

is a refinement of **Assignable** and **Default Constructible**.

Associated types (typedefs):

	<i>type of</i>
value_type	elements
reference	value_type&
const_reference	const value_type&
...	
size_type	container size
iterator	read-write iterator
const_iterator	read-only iterator

Methods:

	<i>return</i>	<i>complexity</i>
begin() →	iterator † or const_iterator ‡	a.c.t.
end() →	idem	a.c.t.
size() →	size_type	less than $O(n)$
empty() →	bool	$O(1)$
swap(other) →	bool	a.c.t.

† if the targeted container is mutable.

‡ if the targeted container is constant.

# Container Concepts

6/15

`a.begin()`:

- points to the first element of `a` if this container is not empty,
- or is equal to `a.end()` otherwise.

`a.end()` points to a *past-the-end* element, that is, a virtual element being after the last container element.

By analogy, in the C string below:

```
char name[] = "john doe",
```

the *true* last character is 'e' and a past-the-end element really does exist (it is the character '\0').

- A past-the-end element allows to stop iterating on the elements of a container whose size is unknown.
- While searching a particular element in a container, it also allows to return a result that indicates that no element was found.

Sample use:

```
list<int> l;  
l.push_front(51);  
list<int>::const_iterator i = l.begin();  
cout << *i << endl;  
cout << *(--l.end()) << endl;
```

# Container Concepts

7/15

## Forward Container

As a refinement of **Container**, the **Forward Container** concept has extra requirements. As a consequence, models of this latter concept have more functionalities (methods and/or typedefs) than models of the former one.

Extra methods are comparison operators:

`a == b` contents equality  
`a != b` defined as “ `!(a == b)` ”

`a < b` lexicographical contents comparison  
`a > b` defined as “ `b < a` ”  
`a <= b` defined as “ `!(b < a)` ”  
`a >= b` defined as “ `!(a < b)` ”

When using these methods, `value_type` is required to respectively be **Equality Comparable** and/or **LessThan Comparable**.

# Container Concepts

8/15

## Reversible Container

Extra typedefs:

	<i>type of</i>	
<code>reverse_iterator</code>		read-write backward iterator
<code>const_reverse_iterator</code>		read-only backward iterator

Extra methods:

	<i>return</i>	<i>complexity</i>
<code>rbegin()</code>	→ <code>reverse_iterator</code> Or <code>const_reverse_iterator</code>	a.c.t.
<code>rend()</code>	→ <code>idem=</code>	a.c.t.

`a.rbegin()` points to the last element of the usual — non-reverse!— browsing. If `a` is not empty, we have:  
`*(l.rbegin()) == *(--l.end())`.

`a.rend()` is a past-the-end iterator in reverse browsing; it points to a virtual element located “before” the first element considering the usual browsing. If `a` is not empty, we have:  
`*(--l.rend()) == *(l.begin())`.

## Random Access Container

Extra Method(s) have an a.c.t.:

	<i>return</i>
<code>operator[] (size_type)</code>	→ <code>reference</code>
<code>operator[] (size_type) const</code>	→ <code>const_reference</code>

# Container Concepts

9/15

## Sequence

Notations:

<i>variable</i>	<i>type</i>	<i>meaning</i>
a	X	a sequence with type X
n	size_type	a number of elements
t	value_type, or T	an element
p Or q and	iterator_type	an iterator on a
i, j		two iterators on a container (with type Y) [i,j[ being valid

Extra methods:

X(n,t)	ctor, fills with n elts t
X(n)	equiv. to X(n,T())
X(i,j)	ctor, fills with the elts in [i,j[
insert(p,t)	inserts t <i>before</i> p
insert(p,n,t)	equiv. to n times insert(p,t)
insert(p,i,j)	inserts the elts in [i,j[ <i>before</i> p
erase(p)	erases the elt pointed by p
erase(p,q)	erases the elts in [p,q[
front()	get front elt, equiv. to *(a.begin())

# Container Concepts

10/15

## Front Insertion Sequence

Extra methods:

`push_front(t) → void` insert `t` at front  
equiv. to `a.insert(a.begin(),t)`

`pop_front() → void` remove front elt  
equiv. to `a.erase(a.begin())`

Models: `list`, `deque`.

## Back Insertion Sequence

Extra methods:

`back() → [const_]ref.` get back elt  
equiv. to `*(--a.end())`

`push_back(t) → void` insert `t` at back  
equiv. to `a.insert(a.end(),t)`

`pop_back() → void` remove back elt  
equiv. to `a.erase(--a.end())`

Models: `vector`, `list`, `deque`.

All front-based and back-based methods are  $O(1)$ .

# Container Concepts

11/15

## Associative Container

Warning: a model of **Associative Container** is a priori *not* a model of **Sequence**.

Extra typedef and notation:

<i>variable</i>	<i>type</i>	<i>meaning</i>
k	key_type	a key
I		[const_]iterator for short

Extra methods:

find(k) → [const_]iter.	find first elt whose key is k
count(k) → size_type	number of elts whose key is k
equal_range(k) → pair<I,I>	all elts whose key is k
erase(k) → void	erases all elts whose key is k
erase(p) → void	erases the elt pointed by p
erase(p,q) → void	erases the elts in [p,q[

Models: \*set, \*map

Sample code:

```
typedef multiset<int> X;
typedef X::iterator I;

const int val[] = {1, 1, 1};
X a(val, val + 3);

pair<I,I> r = a.equal_range(1);
a.erase(r.first, r.second); // equiv. to a.erase(1)
```

# Container Concepts

12/15

## Unique Associative Container

Extra methods:

	<code>X(i,j)</code>	ctor, fills with the elts in <code>[i,j[</code> that have a unique key
<code>insert(t) → pair&lt;I,bool&gt;</code>		inserts <code>t</code> if its key is new for <code>a</code>
<code>insert(i,j) → void</code>		equiv. to <code>insert(t)</code> for every <code>t</code> in <code>[i,j[</code>

`a.count(k)` is always 0 or 1.

If `a` is not empty() and if `k` is represented in `a`,

`r = a.equal_range(k)` always leads to `r.first == --r.second`.

Models: `set`, `hash_set`, `map`, `hash_map`.

## Multiple Associative Container

Extra methods:

	<code>X(i,j)</code>	ctor, fills with the elts in <code>[i,j[</code>
<code>insert(t) → I</code>		inserts <code>t</code>
<code>insert(i,j) → void</code>		equiv. to <code>insert(t)</code> for every <code>t</code> in <code>[i,j[</code>

Models: `multiset`, `hash_multiset`, `multimap`, `hash_multimap`.

# Container Concepts

13/15

## Simple Associative Container

Constraint:

`key_type` and `value_type` have to represent the *same* type.

Models: `set`, `multiset`, `hash_set`, `hash_multiset`.

## Pair Associative Container

Extra typedef:

`mapped_type`     *type of*  
                  elements associated with keys

Constraint:

`value_type` has to be `pair<const key_type, mapped_type>`.  
A model of this concept stores pairs; a value is (key,element)!

Models: `map`, `multimap`, `hash_map`, `hash_multimap`.

# Container Concepts

14/15

## Sorted Associative Container — 1/2

Refinement of both **Associative Container** and **Reversible Container**.

Operation complexity is never worse than logarithmic.

Extra typedef:

	<i>type of</i>
key_compare	function (object) used to cmp keys
value_compare	function (object) used to cmp values it is induced by the key comparison!

Extra methods:

	X(c)	ctor, c being the key cmp.
	X(i,j,c)	ctor, fills with the elts in [i,j[, c being the key cmp.
key_comp() →	key_cmp.	returns c
value_comp() →	value_cmp.	returns a value comparison function (object)
lower_bound(k) →	I	returns an iterator pointing to the 1st elt whose key is not < k
upper_bound(k) →	I	returns an iterator pointing to the 1st elt whose key is > k
insert(p,t) →	I	equiv. to insert(t), p is only a hint!

Models: set, multiset, map, multimap.

# Container Concepts

15/15

## Sorted Associative Container — 2/2

`a.value_comp()(t1, t2)` is equiv. to `a.key_comp()(k1, k2)` when `k1` and `k2` are the keys respectively associated to `t1` and `t2`.

`is_sorted(a.begin(), a.end(), a.value_comp())` always returns true.

## Hashed Associative Container

Extra typedef:

*type of*  
`hasher` function (object) `h` used to hash keys  
`key_equal` a binary predicate `u` for key equality

Extra methods:

`X(n)` ctor, the container is empty but the hasher contains at least `n` buckets  
`X(n,h)` ctor, `h` being the hash function  
`X(n,h,u)` ctor, `u` being the key equal. pred.  
`X(i,j,n)` equiv. to “`X a(n); a.insert(i,j);`”  
...idem for `X(i,j,n,h)` and `X(i,j,n,h,u)`  
`hash_func()` returns `h`  
`key_eq()` returns `u`  
  
`bucket_count()` returns the number of buckets  
`resize(n)` increases bucket count

Models: `hash_set`, `hash_multiset`, `hash_map`, `hash_multimap`.

# Practising Containers

## 1/5 — Common Mistakes

The program below can neither be compiled nor be linked!

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

class Toto {
public:
    Toto(int i) : i(i) {}
    ~Toto();
    const int i;
};

void foo() {          // 1
    list<Toto> l;     // 2

    Toto t1(1);
    l.push_back(t1); // 3

    Toto t2(2);
    l.front() = t2;  // 4

    Toto t3(3);
    l.remove(t3);    // 5

    l.sort();        // 6
}                    // 7

int main() {
    foo();
}
```

# Practising Containers

## 2/5 — Common Mistakes

The program below can be compiled but is erroneous; two lines have to be fixed!

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

int main() {
    vector<int> v;
    for (int i = 0; i < 10; ++i)
        v[i] = i;
    list<int> l;
    copy(v.begin(), v.end(), l.begin());
}
```

The program below gives a strange result!

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

int main() {
    map<string,float> var;
    var["pi"] = 3.14159;
    cout << var["e"] << endl;
    cout << var.size() << endl;
}
```

# Practising Containers

## 3/5 — Common Mistakes

The program below does not compile for one reason!

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

typedef map<int,int> map_t;

void foo(const map_t& m) {
    cout << m[100] << endl;
}

int main() {
    map_t m;
    m[100] = 9;
    foo(m);
}
```

It does not compile with the new `foo` version below and now for two reasons!

```
void foo(const map_t& m) {
    for (map_t::iterator i = m.begin();
         i != m.end(); ++i)
        cout << *i << endl;
}
```

# Practising Containers

## 4/5 — Common Mistakes

The program below does not compile and, once fixed, gives “not found”!

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

int main() {
    set<char*> s;
    s.insert("one");
    s.insert("two");

    string one = "one";

    set<char*>::const_iterator i;
    i = s.find( one.c_str() );

    if (i == s.end())
        cout << "not found!" << endl;
    else
        cout << "found" << endl;
}
```

# Practising Containers

## 5/5 — Common Mistakes

What's the point with the program below?

```
struct B {};  
  
struct A {  
    ~A() {  
        l.clear();  
    }  
    A& add(B* b) {  
        l.push_back(b); return *this;  
    }  
    list<B*> l;  
};  
  
void foo(A a) {  
    // does nothing  
}  
  
int main() {  
    A a;  
    a.add(new B).add(new B);  
    foo(a);  
}
```

and with:

```
struct A {  
    ~A() {  
        for (list<B*>::iterator i = l.begin(); i != l.end(); ++i)  
            delete *i;  
    }  
    // ...  
};  
  
?
```

# Predicates

**Predicate** refines **Unary Function**. Here is a simple model:

```
bool isNull(int i) {
    return i == 0;
}
```

or more sophisticated (a model of the refinement **Adaptable Unary Function**):

```
template<class T>
struct is_equal_to : public unary_function<T,bool> {
    is_equal_to(const T& value) :
        value(value) {
        count = 0;
    }
    bool operator()(const T& t) const {
        bool res = (t == value);
        if (res) ++count;
        return res;
    }
    const T value;
    mutable unsigned count;
};

int main() {
    list<int> l;
    // ...
    unsigned count =
        for_each(l.begin(), l.end(), is_equal_to<int>(7)).count;
}
```

**Binary Predicate** refines **Binary Function**:

```
template<class T>
bool isEqual(const T& x, const T& y) {
    return x == y;
}
```

# Iterators

Required by algorithms.

*concept*

*meaning*

**Input Iterator**

read-only iterator, `*i`, `++i`, `i++`

**Output Iter.**

read-write iterator, `*i = t`

**Forward Iter.**

for **Forward Containers**

**Bidirectional Iter.**

`--i`, `i--`

**Random Access Iter.**

arithmetics works!

Trick: you can inherit typedefs when defining your own iterator.

```
template<class T>
class image2d_const_iterator : public std::iterator_traits<T*> {
    // ...
};
```

There is no need to include particular headers for iterators.

```
#include <set> // for both set and multiset
#include <hash_set> // not std! for both hash_set and hash_multiset
#include <utility> // for std::pair
#include <functional> // for std function objects
#include <algorithm> // for all algorithms
```

# Non-Mutating Algorithms

1/2

Non-mutating algorithms operate on constant containers.

Abbreviations:

<i>parameter</i>	<i>abbrev.</i>
InputIterator	InputI
ForwardIterator	FwdI
UnaryFunction	UnaryF
EqualityComparable	EqCmp
Predicate	Pred
BinaryPredicate	BinPred
pair<InputI1,InputI2>	PairI12

Algorithms:

<i>name</i>	<i>return</i>	<i>args</i>
for_each	UnaryF	InputI first, InputI last, UnaryF f
find	InputI	InputI first, InputI last, EqCmp value
find_if	InputI	InputI first, InputI last, Pred pred
adjacent_find	FwdI	FwdI first, FwdI last
adjacent_find	FwdI	FwdI first, FwdI last, BinPred pred
find_first_of	InputI	InputI first1, InputI last1, FwdI first2, FwdI last2
find_first_of	InputI	InputI first1, InputI last1, FwdI first2, FwdI last2, BinPred comp

# Non-Mutating Algorithms

2/2

<i>name</i>	<i>return</i>	<i>args</i>
count	int-like	InputI first, InputI last, EqCmp value
count_if	int-like	InputI first, InputI last, Pred pred
mismatch	PairI12	InputI1 first1, InputI1 last1, InputI2 first2
mismatch	PairI12	InputI1 first1, InputI1 last1, InputI2 first2, BinPred pred
equal	bool	InputI1 first1, InputI1 last1, InputI2 first2
equal	bool	InputI1 first1, InputI1 last1, InputI2 first2, BinPred pred
search	FwdI1	FwdI1 first1, FwdI1 last1, FwdI2 first2, FwdI2 last2
search	FwdI1	FwdI1 first1, FwdI1 last1, FwdI2 first2, FwdI2 last2, BinPred pred
search_n	FwdI	FwdI first, FwdI last, Int count, EqCmp value
search_n	FwdI	FwdI first, FwdI last, Int count, EqCmp value, BinPred pred
find_end	FwdI1	FwdI1 first1, FwdI1 last1, FwdI2 first2, FwdI2 last2
find_end	FwdI1	FwdI1 first1, FwdI1 last1, FwdI2 first2, FwdI2 last2, BinPred pred

# Mutating Algorithms 0/0

Mutating algorithms operate on mutable (non-constant) containers.

You have understood the STL way of defining algorithms, so you do not really need any exhaustive list for mutating algorithms. Just get a good electronic reference manual!

However...

you still need to learn how to get some *flexibility* from all these algorithms.

# Use Cases

1/3

Hint:

never forget that `for_each` returns your function object.

```
struct adder {
    adder() :
        sum(0.) {
    }
    void operator()(double x) {
        sum += x;
    }
    double sum;
};

int main() {
    vector<double> v(8);
    generate(v.begin(), v.end(), rand);
    adder result = for_each(v.begin(), v.end(), adder());
    cout << result.sum << endl;
}
```

# Use Cases 2/3

Is it some good code?

```
template<class T>
struct in_range { // : public unary_predicate<T,bool> {
    in_range(const T& low, const T& high) :
        low(low), high(high) {
    }
    bool operator()(const T& t) const {
        return !(t < low) && t < high;
    }
    T low, high;
};

int main() {
    vector<int> v(10);
    generate(v.begin(), v.end(), rand);
    vector<int>::iterator i = v.begin();
    while (i != v.end()) {
        i = find_if(i, v.end(), in_range<int>(0, 846930887));
        if (i != v.end())
            cout << *i++ << endl;
    }
}
```

The answer is “not too bad” but...

# Use Cases

3/3

...this one is shorter!

```
#include <functional>
// ...

int main() {
    vector<int> v(10);
    generate(v.begin(), v.end(), rand);

    vector<int>::iterator i = v.begin();
    while (i != v.end()) {
        i = find_if(i, v.end(),
                   compose2(logical_and<bool>(),
                             bind2nd(greater_equal<int>(), 0),
                             bind2nd(less<int>(), 846930887)));
        if (i != v.end())
            cout << *i++ << endl;
    }
}
```

# Predefined Function Objects

## *Arithmetic operations.*

`plus`, `minus`, `multiplies`, `divides`, `modulus`, and `negate`:

- are parameterized by `T`, type of arguments and returns,
- are adaptable,
- are binary except `negate` which is unary.

## *Comparisons.*

`equal_to`, `not_equal_to`, `less`, `greater`, `less_equal`, and `greater_equal`:

- are parameterized by `T`, type of arguments,
- return `bool`,
- are adaptable and binary.

## *Logical operations.*

`logical_and`, `logical_or`, and `logical_not`:

- are parameterized by `T`, type of arguments, which must be convertible to `bool`,
- return `bool`,
- are adaptable and binary.

# Fun. Object Adaptors

1/3

Function object adaptors transform function(s) or function object(s) into a function or a function object, and then allows to adapt objects to algorithms.

Abbreviations:

<i>name</i>	<i>abbrev.</i>
function	fun.
function object	funobj.
predicate	pred.
adaptable	adp.
unary	una.
binary	bin.
pointer	ptr.

Adaptors:

<i>name</i>	<i>input</i>	<i>output</i>
binder1st	adp. bin. funobj.	adp. una. funobj.
binder2nd	adp. bin. funobj.	adp. una. funobj.
ptr_fun	fun. ptr.	funobj.
ptr._to_una._fun.	fun. ptr.	una. funobj.
ptr._to_bin._fun.	fun. ptr.	bin. funobj.
unary_negate	adp. una. pred.	adp. una. pred.
binary_negate	adp. bin. pred.	adp. bin. pred.
unary_compose	2 adp. una. funobj.s	1 adp. una. funobj.
binary_compose	1 adp. bin. funobj. + 2 adp. una. funobj.s	1 adp. una. funobj.

# Fun. Object Adaptors

2/3

Finding the first non-zero value in a container:

```
list<int> l;  
// ...  
list<int>::iterator first_nonzero =  
    find_if(l.begin(), l.end(),  
            binder1st(not_equal_to<int>(), 0));  
assert(first_nonzero == l.end() || *first_nonzero != 0);
```

Let us denote by  $i$  an iterator between `l.begin()` and `l.end()`.  $*i$  is the only argument passed to the predicate in the `find_if` algorithm.

A *binary* function object is created on the fly and has for type `not_equal<int>`. Its first argument is bound to 0 by `binder1st` which then returns a *unary* function object.

This latter object is *the* predicate and iteratively performs  $0 \neq *i$ .

Applying `-fabs` to every element of a container:

```
vector<double> v;  
// ...  
transform(v.begin(), v.end(), v.begin(),  
          compose1(negate<double>(), ptr_fun(fabs)));
```

`fabs` becomes a unary function object due to the call to `ptr_fun` constructor. `compose1` creates from two unary function objects, say  $f$  and  $g$ , a unary function object which represents  $f \circ g$ .

# Fun. Object Adaptors

3/3

Now, it is your turn to explain the two code snippets given below.

```
set<char*> s;
// ...
set<char*>::iterator ok_item =
    find_if(s.begin(), s.end(),
            not1(bind2nd(ptr_fun(strcmp), "OK")));
assert(ok_item == s.end() || !strcmp(*ok_item, "OK"));
```

```
char str[128] = //...
```

```
const char* p =
    find_if(str, str + 128,
            compose2(not2(logical_or<bool>()),
                    bind2nd(equal_to<char>(), ' '),
                    bind2nd(equal_to<char>(), '\n')));
assert(p == str + 128 || !(*p == ' ' || *p == '\n'));
```

# Member Function Adaptors

Sometimes, it is also great to adapt methods.

```
struct A {
    virtual void print() const = 0;
};

struct C1 : public A {
    virtual void print() const { cout << "C1 "; }
};

struct C2 : public A {
    virtual void print() const { cout << "C2 "; }
};

int main() {
    vector<A*> as;
    as.push_back(new C2);
    for_each(as.begin(), as.end(), mem_fun(&A::print));

    vector<C1> cs(3);
    for_each(cs.begin(), cs.end(), mem_fun_ref(&C1::print));
    cout << endl;
}
```

The program above gives: C2 C1 C1 C1.

# Template Meta-Programs

1/4

$n!$  computed at run-time:

```
unsigned fact(unsigned n) inline {
    return n == 1 ? 1 : n * fact(n-1) };
};

int main() {
    cout << fact(7) << endl;
}
```

Then, computed at compile-time by a *meta-program*:

```
template<unsigned n>
struct fact {
    enum { ret = n * fact<n-1>::ret };
};

template<>
struct fact<1> {
    enum { ret = 1 };
};

int main() {
    cout << fact<7>::ret << endl;
}
```

# Template Meta-Programs

2/4

Program in a functional language (**caml**) to sort lists:

```
let rec sort lst =
  match lst with
  [] -> []
  | head :: tail -> insert head (sort tail)
and insert elt lst =
  match lst with
  [] -> [elt]
  | head :: tail -> if elt <= head
                    then elt :: lst
                    else head :: insert elt tail
;;
```

C++ static "list":

```
struct empty_list {};
```

```
template<class head_, class tail_ = empty_list>
struct list {
  typedef head_ head;
  typedef tail_ tail;
};
```

C++ static "if":

```
template<bool cond, class then_type, class else_type>
struct if_;
```

```
template<class then_type, class else_type>
struct if_<true, then_type, else_type> {
  typedef then_type ret;
};
```

```
template<class then_type, class else_type>
struct if_<false, then_type, else_type> {
  typedef else_type ret;
};
```

# Template Meta-Programs

3/4

C++ static sorted insertion:

```
template<class elt, class lst>
struct insert_sorted {
    typedef lst::head head;
    typedef lst::tail tail;
    typedef if_< leq< elt, head >::ret,
                prepend< elt, lst >::ret,
                prepend< head, insert_sorted<elt,tail>::ret >::ret
                >::ret ret;
};

template<class elt>
struct insert_sorted< elt, empty_list > {
    typedef list<elt> ret;
};
```

C++ static sort:

```
template<class lst>
struct sort
{
    typedef insert_sorted<
        lst::head,
        sort< lst::tail >::ret
    >::ret ret;
};

template<>
struct sort<empty_list> {
    typedef empty_list ret;
};
```

typename keywords have been removed to increase code readability and, finally, code is very close to a functional programming version.

# Template Meta-Programs

4/4

Quoted from SGI implementation of the STL:

```
template <class _RandomAccessIter>

inline void sort(_RandomAccessIter __first,
                _RandomAccessIter __last) {

    __STL_REQUIRES(_RandomAccessIter,
                   _Mutable_RandomAccessIterator);
    __STL_REQUIRES(typename
                   iterator_traits<_RandomAccessIter>::value_type,
                   _LessThanComparable);

    if (__first != __last) {
        __introsort_loop(__first, __last,
                         __VALUE_TYPE(__first),
                         __lg(__last - __first) * 2);
        __final_insertion_sort(__first, __last);
    }
}
```

`__STL_REQUIRES` checks the appropriate requirements over types of function arguments; thus, error messages are readable!

Checking parameter bounds is made possible in C++ due to... meta-programming.

# References

1/2

## *About the C++ language:*

Stanley Lippman and Rene Lajoie. C++ Primer.  
a very readable book to learn C++

Scott Meyers. Effective C++.  
good advice and common traps

Scott Meyers. More Effective C++.  
more advice and traps

Bjarne Stroustrup. The C++ Language.  
only interesting as a reference guide

International Standard ISO/IEC 14882:1998(E) — Programming languages — C++.  
the C++ standard; purchasable from [webstore.ansi.org/](http://webstore.ansi.org/).

## *About software engineering:*

James Rumbaugh, Ivar Jacobson, and Grady Booch.  
The Unified Modeling Language Reference Manual.  
UML notation

John Lakos. Large-Scale C++.  
how to handle large projects

Erich Gamma, Richard Helm, Ralph Johnson,  
and John Vlissides. Design Patterns.  
program macro-structures

# References

2/2

## *About STL:*

David Musser and Atul Saini. STL Tutorial and Reference Guide.

a rather good book to begin with

Matthew Austern. Generic Programming and the STL.  
*the book*

Scott Meyers. Effective STL.

good advice and common traps

## *Webography:*

[www.cetus-links.org](http://www.cetus-links.org)

a lot of good links about object-orientation

[www.sgi.com/tech/stl](http://www.sgi.com/tech/stl)

reference guide and source code of STL by SGI

[www.stlport.org](http://www.stlport.org)

a *true* ANSI/ISO C++ library

[www.google.com](http://www.google.com)

with keywords "ANSI", "C++" and "draft";

how to get a *free* draft of the ANSI C++ standard

[www.oonumerics.org](http://www.oonumerics.org)

a lot of materials about scientific computing with C++