### Advanced C++ Libraries and Introduction to Template Metaprogramming

**Jean-Paul Rigault**
University of Nice - Sophia Antipolis
Engineering School — Computer Science Department
SOPHIA ANTIPOLIS, France
Email: jpr@polytech.unice.fr

---

#### Lectures and Labs

<table>
<thead>
<tr>
<th>Time</th>
<th>Lectures</th>
<th>Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:00-1:30</td>
<td>1. Introduction</td>
<td>Lab 5: boost::any</td>
</tr>
<tr>
<td>1:30-2:00</td>
<td>2. Template reminder and complements</td>
<td>Lab 3: Regex</td>
</tr>
<tr>
<td>2:00-2:30</td>
<td>3. Smart pointers</td>
<td>Lab 4: Type traits</td>
</tr>
<tr>
<td>2:30-3:00</td>
<td>4. Type traits and high order programming</td>
<td>Lab 2: Logger (optional)</td>
</tr>
<tr>
<td>3:30-4:00</td>
<td>5. Boost containers and algorithms</td>
<td></td>
</tr>
<tr>
<td>4:00-4:30</td>
<td>6. Strings and regular expressions</td>
<td></td>
</tr>
<tr>
<td>1:00-1:30</td>
<td>7. Boost serialization</td>
<td></td>
</tr>
<tr>
<td>1:30-2:00</td>
<td>8. Introduction to the metaprogramming library (MPL)</td>
<td></td>
</tr>
<tr>
<td>2:00-2:30</td>
<td>9. Computing with types: the Boost metaprogramming library MPL</td>
<td></td>
</tr>
</tbody>
</table>

---

#### Part 1

**Introduction**
Generative Programming

Generative programming
- Programming the generation of programs
- A very old practice in Computer Science

Metaprogramming
- Generative programming where the generative language and the target language are the same
- The language must have reflection capabilities

Template (meta)programming
- Metaprogramming à la C++
  - The generative language is the template subset of C++
  - The target language is full C++

C++ STL Extensions: TR1

- Non normative extensions to the standard
- Candidate for integration into next standard
- TR1 is now part of current C++ Standard (C++ 2011)
- Available (partially) with most current implementations
- For the most part, already in Boost

- General utilities (reference wrappers, smart pointers)
- Function objects (mem_fn, bind)
- Type traits
- Numerical facilities (random numbers, special functions)
- Containers (tuple, array, unordered associative containers)
- Regular expressions (regex)
- Improved C99 compatibility

C++ STL Extensions: Boost

Open source
- Boost Software License
- Extensions of the STL
  - Respect basic philosophy
  - Many sub-libraries
    - Different themes
    - Different sizes
- Work on most C++ compilers
- Large library
  - Usable by pieces...
  - Some parts integrated into TR1

- String and text processing
- Containers, iterators, algorithms
- Function objects, high order programming
- Generic and template metaprogramming
- Preprocessor metaprogramming
- Concurrent programming (Thread)
- Maths and numerics
- Correctness and testing
- Data structures, graphs (Graph)
- Image processing (CIL)
- Input/output (ASSO, serialization...)
- Inter-language support (Python)
- Memory handling
- Parsing (Spirit)
- Programming facilities...
C++ Extensions: Towards C++11

- The current version of C++ (C++11) will propose many extensions to C++
- Some of these extensions are already available with some compilers
- Extensions of C++11 used in the examples of this course (C++14, 7.x or more recent)
  - Inclusion of the Technical Report 1 (TR1) into the standard library
  - Fix for the >> problem: A<>>int
  - Static assertion: static_assert(c, "message")
  - c must be a compile time expression convertible into bool
  - We use our own macro static assert(c) where the message is empty
  - Uniform initialization syntax: vector<int> v = {1, 2, 3, 4};
  - Auto typing: auto x = expr
  - Range-based for loop:
    for (auto e : container) {...}
  - Lambda expressions
  - True null pointer:
    nullptr
  - Defaulted and deleted member functions:
    A(const A&) = delete; // class not copy constructible
    A() = default; // provide the default default constructor

References

- Generative Programming: Methods, Tools, and Applications
  Krzysztof Czarnecki, Ulrich Eisenacke – Addison Wesley, 2000
  [General]
- C++ Templates: The Complete Guide
  David Vandevoorde, Nicolai Josuttis – Addison Wesley, 2003
  [General]
- Modern C++ Design: Generic Programming and Design Patterns Applied
  Andrei Alexandrescu – Addison Wesley, 2001
  [General]
- The C++ Standard Library Extensions: A Tutorial and Reference
  Pete Becker – Addison Wesley, 2006
  [TR1]
- Beyond the C++ Standard Library
  Bjorn Karlsrud – Addison Wesley, 2006
  [Boost]
- C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond
  David Abrahams, Aleksey Gurtovoy – Addison Wesley, 2005
  [Boost mpl]
- Proposed Draft Technical Report on C++ Library Extensions
  ISO/IEC PDR 19769, 2005
  [TR1]
- Boost Library Documentation
  http://www.boost.org/
  [Boost]
- The NPL Reference Manual
  [Boost mpl]
C++ Templates

Class and function templates overview
- Definition and use
- Template parameters

Template specialization
- Full specialization of class and function templates
- Partial specialization of class templates
- Consequences of specialization on name lookup
- Turing-completeness of C++ templates

Overloading resolution
- Overview
- Argument (type) deduction
  • SFINAE: Substitution Failure Is Not An Error

Function Templates

Definition of function templates

```cpp
template <typename T>
const T& Min(const T& a, const T& b) {
    return a < b ? a : b;
}
```

Constraints on template parameters (T) are implicit (“Duck Typing”)

Use of function templates
- Parameterized overloading
  ```cpp
  int x, y, z; . . .; z = Min(x, y);
  string s1, s2, s; . . .; s = Min(s1, s2);
  ```
- Explicit instantiation
  ```cpp
  int u = Min<double>(x, 5.7);
  ```

Class and Function Templates

Functions and classes templates parameterized (principally) with types
- Two different mechanisms for functions and classes
- Although somewhat homogenized by ANSI C++

Static resolution (compile and link time)
- The template parameters must be static entities
- A type driven macro-processing facility
- A full programming language on top of (template-less) C++

Class Templates

Definition of a class template
```cpp
// File: List.h

template <typename T>
class List {
    public:
        List();
        void append(const T&);
    . . .
};
```

Definition of the members of a class template
```cpp
// File: List.cpp

template <typename T>
List<T>::List() {
    . . .
}
```

Use of a class template
```cpp
// File: xxx.cpp

List<int> aList;
. . .
aList.append(3);
```
Instances of class templates and implicit conversions

Since there exists an implicit conversion from \texttt{int} to \texttt{double}, shouldn't be a \texttt{List<int>} implicitly convertible into a \texttt{List<double>}?

Since a French is a Human (suppose), shouldn't a set of French be implicitly convertible into a set of Human?

The answer is NO in both cases
- Should the answer be yes, substitutability principle, contravariance rule, and static type checking would be broken...

Two instances of the same class template define different types as soon as their (effective) template parameters are different

Breaking static type checking?

Assume French (and English) derives from Human

Hence, French * is implicitly convertible into Human *

Suppose we allow implicit conversion from set\texttt{<French *}> to set\texttt{<Human *>

Static typing is broken!

Member Function Template (1)

Member template of a regular class

class A {
    public:
        template <typename U> void mt(U u);
    
    template <typename U> void A::mt(U u) { ... }

Just a parameterized overloaded member function

A a;
a.mt(3); // A::mt\texttt{(int)}
a.mt("hello"); // A::mt(const char *)
a.mt(3.5); // A::mt\texttt{(double)}

Member Function Template (2)

Member template of a class template

template <typename T>
class A {
    public:
        template <typename U> void mt(U u);
    
        template <typename T>
template <typename U> void A<T>::mt(U u) { ... }

A<int> a;
a.mt("hello"); // A<int>::mt(const char *)

Member templates cannot be virtual
Template parameters of templates (1)

- ISO C++ unifies the template parameter possibilities for template functions and classes
- A generic (template) parameter may be
  - A type (built-in or class)
  - A static (compile-time) constant of any integral type
    - This includes enumerations and also pointers to members and pointers to extern variables and functions
  - Another class template
- Default values are possible for template parameters
  - But for class templates only, not for function templates (in C++03)

Template parameters of templates (2)

- Pointers as template parameters

```
template <typename T, void (*SORT)(int, T[])> class Sorter {...};
```

```
extern void quicksort(int, double[]); Sorter<double, quicksort> s(...);
```

- Parameterize internal implementation (algorithms)
- The pointed object must be extern

Template parameters of templates (3)

- Class templates as template parameters

```
template <typename U>
    template <typename, typename> class Container = std::vector;

    class AClass {
        Container<string, std::allocator<string>> cS;
        Container<int, std::allocator<int>> cI;
        // ...
    };

template <typename T, typename A>
    class My_Container {...};

    AClass<int, My_Container> a1; // use My_Container
    AClass<double, std::list> a2; // use std::list
    AClass<int> a3; // use default std::vector

    Parameterize internal implementation (data structures)
```

Template parameters of templates (4)

- Default values for template parameters

```
template <typename T = int, int N = 10>
    class Fixed_Array {...};

    Fixed_Array<double, 100> fa1;
    Fixed_Array<double> fa2; // Fixed_Array<double, 100>
    Fixed_Array<> fa3; // Fixed_Array<int, 10>
```

- Rules are similar to default value for function parameters
Template parameters of templates (5)

Forbidden effective template parameters

- Local types (in C++ 2003)
- Constants of type real
- Non-external pointers...

```
template <typename T>
class A { ... };

void f() {
    class L { ... };
    A<
        c
    > a; // NO in C++03
}
```

```
template <typename T>
class B { ... };

const double PI = 3.141592;
B<PI> bpi; // NO
```

```
template <typename T>
const T& Min(const T& a, const T& b) { ... }
```

```
template<>
const char *Min<> (const char *s1, const char *s2) {
    return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

```
template <typename U>
class List<U*>{ ... };
```

The specializations must be in the same namespace as the template definition

Class Template Partial Specialization

Partial specialization of a class template

```
template <typename T>
class List { ... };
```

```
template <typename U>
class List<U*>{ ... };
```

the contents and interface of List<U*> can be totally different from those of the generic List<T>

Function Template Partial Specialization (1)

Both function and class templates can be fully specialized

- This includes member templates
- Only class templates can be partially specialized
  - Function and member templates cannot...
  - ... but they can be overloaded

```
template <typename T>
const T& Min(const T& a, const T& b) { ... }
```

```
template <typename T>
T* Min(T* a, T* b) { ... }
```

```
const T& Min(T* a, T* b) { ... }
```

"C++11 will allow partial specialization of function templates"

The specializations must be in the same namespace as the template definition

Template Full Specialization

Full specialization of a function template

```
template <typename T>
const T& Min(const T& a, const T& b) { ... }
```

```
template <>
class List<> { ... };
```

a specialization of List<T> when T is a C-string

```
template<>
class List<char*> { ... };
```

the contents and interface of List<char*> can be totally different from those of the generic List<T>
Consequences of Specialization on Name Lookup (1)

- Templates are parsed twice
  - At the template definition
    - Verify basic syntax
  - Look up non-dependent names since they must have been already defined
  - At instantiation point
    - Where the compiler inserts the substituted template definition
    - Look up dependent names since their resolution required knowledge of effective template parameters
  - 2 phases lookup

Independent name
- A name that does not depend on any template parameter

Dependent name
- A name that depends on (some) template parameters
- Template \( \text{A} < \text{T} > \)
- Template \( \text{B} < \text{T} > \)
- Template template <typename T>

Consequences of Specialization on Name Lookup (2)

- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // compile error
}
- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // OK
  A<T>::_i = 12; // OK
}
- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // OK
}
- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // OK
}
- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // OK
}
- Template template <typename T>
- Template struct A { int _i; }
- Template void f() {
  _i = 12; // OK
}

TURING-completeness of C++ Templates (1)

- C++ offers two levels of languages
  - C++ without templates, a classical programming language
    - with run-time semantics
  - The template mechanism itself
    - with compile-time semantics
- Both levels are Turing-complete
  - They make it possible to compute any Computable Function
  - Thus C++ Templates constitute a programming language whose programs are evaluated at compile-time
  - The power of C++ templates comes (in particular) from
    - integral template parameters
    - template specialization
Computing the Fibonacci series at compile time with templates

```cpp
template <int N> struct Fibo {
    static const long long value =
        Fibo<N-1>::value + Fibo<N-2>::value;
}; // Generic form

template <> struct Fibo<0> {
    static const long long value = 1;
}; // Specialization to stop recursion

template <> struct Fibo<1> {
    static const long long value = 1;
}; // Specialization to stop recursion

int main() {
    cout << Fibo<40>::value << endl;
}
```

Note that the computational complexity of $Fibo<N>$ is linear.

```
<table>
<thead>
<tr>
<th>Fibo(40)</th>
<th>Compilation</th>
<th>Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative</td>
<td>0.24</td>
<td>0.00</td>
</tr>
<tr>
<td>Recursive</td>
<td>0.24</td>
<td>2.34</td>
</tr>
<tr>
<td>Template</td>
<td>0.22</td>
<td>0.00</td>
</tr>
</tbody>
</table>
```

Intel Xeon at 2.33 GHz (times in seconds)

Overloading Resolution (1)

**Overview**
Given a function call, find a unique function with the same name matching the call

- $f(a_1, a_2, \ldots, a_n)$
  - $n$ is known, as well as the argument types $T_1, T_2, \ldots, T_n$
  - Context and return type play no role

**Overloading resolution principle**

1. Identify candidate functions using name lookup
2. Among candidate functions, select the viable ones, the ones that can correspond to the call
3. Among viable functions, select the best match for the call
   - Relies on ranking possible argument conversions

Overloading Resolution (2)

**Implicit conversion strength**

1. Exact match: strict exact match (identity) or `value` transformations
2. Qualification adjustment: adding `const` or `volatile`
3. Integral and floating point promotions
4. Integral and floating point conversions, pointer conversions, inheritance conversions
5. User defined conversions: constructor, cast operator
   - at most one for each argument

The best viable function is the unique function, if it exists, such that
- the conversion applied to each argument is no worse than for all the other viable functions
- there is at least one argument for which this function has a strictly better conversion that the other viable functions
Remarks on overloading resolution when template functions are candidates

- A template function is not a real function, but a potential infinity of them.
- Thus we need to select template instances.
- However, the richness of C++ implicit conversions makes the number of candidate instances huge, if not infinite.
- Hence, the number of candidate instances has to be reduced.

Argument deduction

- The process of matching (substituting) the call arguments with the function template parameters.
- Only the following conversions are accepted:
  1. Exact match
  2. Qualification adjustment
  3. Derived to base (inheritance) conversions

Overloading vs. Function (Full) Specialization

Do not confuse specialization and overloading!

- The supported conversions are not the same...
- `template <typename T>
  bool is_equal(const T& t1, const T& t2); // generic`
- `template <>
  bool is_equal(const string& t1, const string& t2); // specialize`
- Consider
  `bool b = is_equal("hello", "bonjour");`
- Specialization (specialize) does not match
  - It would require a user-defined conversion `const char* -> string` forbidden in argument deduction context
- Overloading (override) does match since user-defined conversion is valid here
  - and `override` does match since a user-defined conversion is valid here.

Overloading Resolution (3)

Overloading Resolution (4)

Overloading resolution when template functions are candidates

- Candidate functions
  - Add the function template instances for which argument deduction succeeds: parameter substitution.
  - Select template specializations, if any, instead of generic form.
  - Viable functions: unchanged.
  - Candidates issued from templates are automatically viable.
  - Best viable function
    - In case of ambiguity for the best viable between a template and a non template instance, select the non template.

It is not an error for argument deduction (parameter substitution) to fail

- This simply means that no instance of the template function can match the call.
- But other functions—templates or not—may do so.

SFINAE: Substitution Failure is Not An Error

- Template programming often takes advantage of it.
- See in particular `boost::enable_if` later.

Advanced C++ Libraries

Part 3

Shared Pointers
Working with Pointers

- Drawbacks of regular (C) pointers
  - The Resource Acquisition Is Initialization idiom (RAII)
  - Smart pointers and RAII; the infamous auto_ptr
- Boost and TR1 RAII pointers
  - boost::scoped_ptr: a pure RAII smart pointer
  - tr1::unique_ptr: a replacement for auto_ptr
  - tr1::shared_ptr: a pointer to safely share (and delete) resources
  - tr1::weak_ptr: coping with circular data structures
- Smart pointers and inheritance
- Pros and cons of smart pointers

Drawbacks of regular (C) pointers

```cpp
void f() {
    int *pi = new int[3];
    int *pj = new int[3];
    // . . .
    delete pi;
    if (*pi == 0)
        throw Exc();
    else
        *pi = 12;
    // . . .
}
```

pointeurs to individual objects and to arrays are not distinguishable
when dynamically allocated, explicit deletion required...
... but when? this certainly compiles and runs...
but this may crash, and even if it does not, it is incorrect

Resource Acquisition Is Initialization (RAII)

- Use destructor to release a resource at the end of its scope
  ```cpp
class Lock {
    Mutex& _m;
public:
    Lock(Mutex& m) : _m(m) {_m.lock();}
    ~Lock() {_m.unlock();}
};
void f() {
    Lock l(the_mutex);
    ifstream is("foo.txt");
    // . . .
}
```
  - ifstream() automatically flushes the buffer and releases all
  - ~ofstream() automatically deletes the mutex

Smart Pointers and RAII

```cpp
template<typename T>
class SP {
    T *pt; // encapsulated pointer
public:
    SP(T *pt) : _pt(pt) {
        // possibly other constructors,
        ~SP() if (needed) delete _pt;
        // other operations: *, ->, casts..
    }
    ~SP() {
        // work with pa as with a pointer
        ~SP() automatically deletes the A object if needed
    }
};
class A { . . .};
void f() {
    SP<A> pa(new A());
    // . . .
}
```

The condition for deleting the object may vary with the nature or role of the smart pointer
The object must be allocated with new
~SP() automatically deletes the A object if needed
Smart Pointer and RAI
The infamous auto_ptr (1)

In the standard library since 1998
Copyable
but with special (bizarre!) copy semantics
the pointer origin of the copy is set to zero and thus becomes invalid
the object pointed to is not sharable through several auto pointers
auto_ptr cannot be put into STL containers
The only “safe” copy context is returning an auto_ptr by value from a function
Its use should be restricted to legacy code and only when copying is not needed.

Smart Pointer and RAI
The infamous auto_ptr (2)

#include <memory>
class A { ... };
void f(std::auto_ptr<A> ap);
void someFunction() {
    std::auto_ptr<A> ap1(new A());
    // work with ap1 as if it were a pointer
    std::auto_ptr<A> ap2(ap1); // ownership transfer (move)
    // the internal pointer of ap1 has been set to null
    f(ap2); // ownership transfer (move)
    // the internal pointer of ap2 has been set to null
}

The A object is deleted correctly (and only once) on exiting the function

Smart Pointer and RAI
The infamous auto_ptr (3)

std::auto_ptr<A> h() {
    std::auto_ptr<A> ap(new A());
    //...
    return ap;
}
void someFunction() {
    //...
    std::auto_ptr<A> ap1;
    ap1 = h();
    //...
    return;
}

Nothing is deleted here since ap has been set to null by the function return
Ownership of the A object transferred to ap1 in h(), ap set to null
The A object is deleted correctly (and only once) on exiting this function

Boost Scoped Pointer

A strict auto pointer, used only for RAI, not copyable
The object pointed to cannot be shared

#include <boost/scoped_ptr.hpp>
void f(boost::scoped_ptr<A> ap); // NO: compile error
void someFunction() {
    boost::scoped_ptr<A> ap1(new A());
    // work with ap1 as if it were a pointer
    boost::scoped_ptr<A> ap2(ap1); // NO
    ap2 = ap1; // NO
}

The A object is deleted correctly on exiting the function
TR1 Smart Pointers

- **TR1** borrows and adapts two smart pointers from Boost
  - `shared_ptr<T>`: reference counting smart pointer, copiable
  - `weak_ptr<T>`: a sort of pointer observer, to break circular data structures
  - Boost has other smart pointers which are not part of TR1 (nor of C++11)
    - `scoped_ptr` replaced by `std::unique_ptr`
    - `scoped_array` and `shared_array`: no so useful
    - `intrusive_ptr`: sharing objects with an embedded reference count

C++11 Unique Pointer (1)

- A strict auto pointer, used only for RALI, not copyable
  - The object pointed to cannot be shared
- Usage identical to `boost::scoped_ptr`
  ```cpp
  #include <memory>
  
  void f(std::unique_ptr<A> ap); // NO: compile-time error
  
  void someFunction() {
    std::unique_ptr<A> ap1(new A());
    // work with ap1 as if it were a pointer
    std::unique_ptr<A> ap2(ap1); // NO
    ap2 = ap1; // NO
  }
  ```
  - The A object is deleted correctly on exiting the function

C++11 Unique Pointer (2)

- By default, the pointed object is deleted by `operator delete`
  - The pointed object must have been allocated by `new`
- It is possible to pass a `deleter` function (or function-object) to the `unique_ptr` constructor
  ```cpp
  class A { . . . };
  void a_deleter(A *pa) { . . . }
  . . .
  void f() {
    A a; // or any other kind of allocation
    unique_ptr<A> pa(&a, a_deleter);
    // . . .
  }
  ```
  - Destruction of `pa` calls `a_deleter()` on the pointed object
  - In this case, `delete` and `~A()` are not used for deleting the pointed object
- This makes it possible to release resources which are not allocated by `new` (or even which are not related to memory)
- `unique_ptr` supports C++11 move semantics (ownership transfer)

TR1 Smart Pointers

Constructions of `shared_ptr`

```cpp
#include <memory> // for C++11
#include <tr1/memory> // for C++03 with TR1
using namespace std;
using namespace std::tr1; // for C++03 with TR1

// Empty shared pointer
shared_ptr<T> pt; // owns nothing

// Pointer to a dynamically allocated object
shared_ptr<T> pt(new T(constructor parameters)); // constructor explicit
// preferred
auto pt = make_shared<T>(constructor parameters); // preferred
// This constructor allocates a new reference count for the pointed object
// Destruction is performed by `operator delete` on pointed object

// Pointer to a resource which is not dynamically allocated, or which is not memory based, or...
void a_deleter(T *p) { . . . } // or any other kind of allocation
shared_ptr<T> pt(pt, a_deleter);
// Destruction of pt calls a_deleter() on the pointed object
// delete and ~T() are not used for deleting the pointed object
```
**TR1 Smart Pointers**

**shared_ptr**

- **Smart pointer with reference counting**
  - The reference count is handled by the smart pointer, outside the object
    - The `shared_ptr` constructor allocates the reference count
    - Copyable: copy operations update the reference count
    - The `shared_ptr` destructor decrements the reference count and deletes the pointed object when the count becomes 0

```
        p1
    shared_ptr
        |     
        v     
Object

2 reference count
```

- **shared_ptr are copiable objects**
  - Copy constructor and copy assignment
  - `shared_ptr` can be put into STL containers

- **Copy with conversion**
  - A `shared_ptr<U>` is implicitly convertible into a `shared_ptr<T>` provided that `U*` is implicitly convertible into `T*`
  - This is possible only if `U` derives from `T` (or if `T` is `void`)

- **Comparison operations**
  - Equality, inequality, relational operators

- **Display operation**
  - `operator<<` for `ostream`

**Specific shared pointer member functions on shared_ptr (2)**

- `sp1.swap(sp2) or swap(sp1, sp2)`
  - Exchange the two pointed objects

  ```
  T* p = sp.get()
  ```

  - Return the internal pointer

  ```
  T& rt = *sp (operator*)
  ```

  - Return a reference to the pointed object

  ```
  sp->f() (operator->)
  ```

  - Return the internal pointer so that a member can be selected

  ```
  long n = s.use_count();
  ```

  - Return the current reference count

  ```
  if (sp.unique()) . . .
  ```

  - Return whether the reference count of sp is 1 (unique owner)

**Specific shared pointer member functions on shared_ptr (3)**

- `sp.reset()`
  - Release ownership (decrement reference count)
  - Equivalent to `shared_ptr().swap(*this)`

- `sp.reset(p)` (p is a pointer to some object, possibly of different type)
  - Replace currently owned object by the one pointed by p

  ```
  p must be convertible into the type of the internal pointer of sp
  ```

  - Equivalent to `shared_ptr(p).swap(*this)`

- `sp.reset(p, d)` (p is a pointer as above, d is a deleter)
  - Replace currently owned object by the one pointed by p with deleter d

  ```
  p must be convertible into the type of the internal pointer of sp
  ```

  - Equivalent to `shared_ptr(p, d).swap(*this)`

- `sp.get_deleter()`
  - Return the address of the current deleter or null pointer if not any
TR1 Smart Pointers
Operations on shared_ptr (4)

- **Conversion of shared pointers**
  - A shared_ptr<U> is implicitly convertible into a shared_ptr<T> provided that U* is implicitly convertible to T*
  - `spt = static_pointer_cast<T>(spu)`
    - `spt` is a shared_ptr<T>, `spu` is a shared_ptr<U>
    - U* must be convertible into T* using static_cast
  - `spt = dynamic_pointer_cast<T>(spu)`
    - same as before but using dynamic_cast
    - if the cast fails, return an empty shared pointer for `spt`
    - U and T must be polymorphic types
  - `spt = const_pointer_cast<T>(spu)`
    - same as before but using const_cast
    - avoid it!

TR1 Smart Pointers
Circular Data Structures and Shared Pointers

If there are no other references to these objects there is no way to delete them ⇒ Memory Leak

TR1 Smart Pointers
Shared Pointer Observers: weak_ptr (1)

- **A weak pointer is an observer of a shared pointer**
  - When the shared pointer releases its resource, it sets the observing weak pointer(s) to null
    - Thus a weak pointer never contains a dangling pointer
    - A weak pointer does not interfere in any way with the reference count
  - **It is impossible to access the resource directly through a weak pointer**
  - The weak pointer must first be transformed into a shared pointer

TR1 Smart Pointers
Shared Pointer Observers: weak_ptr (2)

- **Weak pointer operations**
  - Weak pointers are copiable
  - Weak pointers are transformable to and from shared pointers
  - A weak_ptr<U> is implicitly convertible into a weak_ptr<T> provided that U* is implicitly convertible to T*
  - The `get()`, `swap()`, `reset()` operations are similar, *mutatis mutandis*, to their counterparts in shared_ptr
  - There are no dereferencing operators (such as `operator*` or `operator->`) for weak_ptr

Advanced C++ Libraries
57
©2009-2014 — Jean-Paul Rigault

Advanced C++ Libraries
58
©2009-2014 — Jean-Paul Rigault

Advanced C++ Libraries
59
©2009-2014 — Jean-Paul Rigault

Advanced C++ Libraries
60
©2009-2014 — Jean-Paul Rigault
Obtaining a `weak_ptr` from a `shared_ptr`

Use the constructor or assignment operator

```cpp
shared_ptr<A> pa(new A());
weak_ptr<A> wp = pa; // implicit conversion
```
- Since the constructor is not explicit, the corresponding conversion is implicit
- It is always possible to observe a shared pointer

Verifying that the observed `shared_ptr` is still owning a resource

The expired operation

```cpp
if (wpexpired()) ... // no resource owned
```

Obtaining a `shared_ptr` from a `weak_ptr`

Use the constructor or assignment operator

```cpp
weak_ptr<A> wp;
shared_ptr<A> sp(wp); // constructor is explicit
sp = wp; // assignment
```
- Throws `bad_weak_ptr` if `wp` is expired

Use `weak_ptr` operation `lock()`

```cpp
weak_ptr<A> wp;
shared_ptr<A> sp = wp.lock();
```
- No exception is thrown if `wp` is expired
- Instead return an `empty` shared pointer (thus owning nothing)

Example of `weak_ptr`: a doubly linked list (1)

```cpp
template <typename T> class DList {
public:
    struct DCell  {
        T _t;
        shared_ptr<DCell> _next;
        weak_ptr<DCell>  _prev;
        DCell(const T& t) : _t(t) {}
        ~DCell() { cout << "DCell destructor\n"; }
    };
private:
    shared_ptr<DCell> _head;
    shared_ptr<DCell> _tail;
public:
    shared_ptr<DCell> append(const T &t);
    shared_ptr<DCell> insert_before(shared_ptr<DCell> where, const T &t) ;
};
```
Example of weak_ptr: a doubly linked list (2)

```cpp
template <typename T>
shared_ptr<DList<T>::DCell>
DList<T>::append(const T& t) {
    shared_ptr<DCell> newcell(new DCell(t));
    if (not _head)
        _head = _tail = newcell;
    else {
        _tail->next = newcell;
        newcell->prev = _tail; // implicit conversion
        _tail = newcell;
    }
    return newcell;
}
```

Example of weak_ptr: a doubly linked list (3)

```cpp
template <typename T>
shared_ptr<DList<T>::DCell>
DList<T>::insert_before(shared_ptr<DCell> where, const T& t) {
    if (not where)
        return append(t);
    shared_ptr<DCell> newcell(new DCell(t));
    newcell->_prev = where->_prev;
    newcell->_next = where;
    // where->_prev->next = newcell; // DOES NOT COMPILE
    shared_ptr<DCell> prev = where->_prev.lock();
    if (prev)
        prev->_next = newcell;
    else
        _head = newcell;
    where->_prev = newcell;
    return newcell;
}
```

The problem

```cpp
struct A {
    ~A() { cout << "A destructor\n"; }
    shared_ptr<A> f() {
        return shared_ptr<A>(this);
    }
};
```

What if the A object is already owned by a shared pointer?

```cpp
int main() {
    shared_ptr<A> pa1(new A());
    shared_ptr<A> pa2(pa1->f());
}
```

At first, the A object is owned by pa1

When calling pa1->f() a fresh shared pointer is returned, with its own reference count set to 1; thus the A object is now owned by two completely different shared pointers

Thus A is deleted twice!

The solution

```cpp
struct A : std::enable_shared_from_this<A> { // CRTP
    ~A() { cout << "A destructor\n"; }
    shared_ptr<A> f() {
        return this->shared_from_this();
    }
};
```

```cpp
int main() {
    shared_ptr<A> pa1(new A());
    shared_ptr<A> pa2(pa1->f());
}
```

Base class contains an observer (a weak_ptr) of the shared pointer on the A object
When calling pa1->f(), if there exists already a shared pointer owning the A object, shared_from_this() returns a copy of this shared pointer
Otherwise shared_from_this() throws bad_weak_ptr

Example of weak_ptr: observer (1)

```cpp
struct A {
    ~A() { cout << "A destructor\n"; }
    shared_ptr<A> f() {
        return shared_ptr<A>(this);
    }
};
```

Example of weak_ptr: observer (2)

```cpp
struct A {
    A() : enable_shared_from_this(*this) {
        cout << "A constructor\n";
    }
    ~A() { cout << "A destructor\n"; }
    shared_ptr<A> f() {
        return this->shared_from_this();
    }
};
```

When calling A::f(), if there exists already a shared pointer owning the A object, shared_from_this() returns a copy of this shared pointer
Otherwise shared_from_this() throws bad_weak_ptr
It is possible to construct a `XXX_ptr<T>` from an `XXX_ptr<U>` provided that `U` is implicitly convertible into `T`.

This is the case when `U` derives from `T`.

This is implemented as a constructor template in the smart pointer classes:

```
template <typename T>
template <typename U>
shared_ptr<T>::shared_ptr(
    const shared_ptr<U>& spu);
```

Which destructor is called?

- `boost::scoped_ptr` and `std::unique_ptr` calls `~A()`.
- `std::shared_ptr` calls directly `~B()` and accepts `~A()` to be protected.
- Identical behavior for the deleter.

protected Destructor Idiom for `shared_ptr`

```
class A {
    // ... 
    public:
        ~A();
    };

class B : public A {
    // ... 
};

shared_ptr<A> pa1(new A());
delete pa1->get(); // not a good idea, but allowed
A a = *pa1; // likely crash or incorrect behavior
```
Pros and Cons of Smart Pointers

**Pros**
- Safe deallocation, especially for `shared_ptr`
  - No memory leak, no crash due to spurious deletions
- Inheritance and virtual functions still operational
- Easy to use: simply replace `A*` with `shared_ptr<A>`

**Cons**
- Problem with circular structures (`weak_ptr`)
- Strange copy: ownership transfer or move semantics
- Size penalty: shared and weak twice the size of a regular pointer
- Speed penalty for copy operations?

---

Pros and Cons of Smart Pointers

Size of smart pointers

<table>
<thead>
<tr>
<th></th>
<th>sizeof</th>
<th>factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular pointer (T*)</td>
<td>8</td>
<td>×1</td>
</tr>
<tr>
<td>auto_ptr</td>
<td>8</td>
<td>×1</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>8</td>
<td>×1</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>16</td>
<td>×2</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>16</td>
<td>×2</td>
</tr>
</tbody>
</table>

Intel Xeon (64 bits)

---

Pros and Cons of Smart Pointers

Tips and Traps

- **Avoid auto_ptr**
- If copy or sharing is not needed, use `scoped_ptr` or `unique_ptr`
- If sharing is needed or in case of doubt, use `shared_ptr`
  - but take care of circular pointer chains (`weak_ptr`)
- Never mix `smart_ptr` and regular C pointers...
  - ...unless you are sure that the regular pointers do not (and will not in future evolutions) claim ownership of the objects
  - Restrict the use of `share_ptr::get()` to pass a regular pointer to a legacy C function (e.g., system programming) or to the case above
- **Shared pointer are not guaranteed thread safe in the C++03 standard**
  - but they are some thread safe implementations (e.g., g++)
  - however, this does not solve the problem of concurrent updates of the pointed object: mutexes are still required for this

---

Advanced C++ Libraries and Introduction to Template Metaprogramming

**Part 4**

Type Traits and High Order Programming
Consulting and Manipulating Types

- **Boost and tr1 type_traits**
- Control of template instantiation: std::enable_if
- High order programming
  - Boost and the tr1 bind adaptor
  - Boost and tr1 function
  - C++11 lambda expressions and closures

Boost and tr1 Type Traits (1)

- Example
  ```cpp
  // Using C++11
  #include <type_traits>
  using namespace std;
  template <typename T>
  class A {
    static_assert(
      not
      is_void<T>::value,
      "cannot instantiate A"
    );
    T*_pt;
  };
  ```

- Boost vs tr1/C++11
  - Identical functionality...
  - ... but notational differences
  - Do not use both simultaneously!

- Set of classes (in fact structs) templates to consult or even transform type properties at compile time

Boost and tr1 Type Traits (2)

- Simplified implementation of is_void
  ```cpp
  // Generic form: most types are not void
  template<typename T>
  struct is_void {
    static const bool value = false;
  };

  // Full specialization: only void is void!
  template <>
  struct is_void< void > {
    static const bool value = true;
  };
  ```

- Convention for type traits classes
  ```cpp
  template <typename T>
  struct property {
    static const Tval value = val;
    typedef Ttype type;
  };

  if (property<MyType>::value) . . .
  typedef property<MyType>::type MyType2;
  static_assert(property<MyType>::value, "property is false");
  ```

Boost and tr1 Type Traits (3)
Boost and **TR1** Type Traits

**Integer to Type (1)**

```cpp
template <typename T, T val>
struct integral_constant {
    static const T value = val;
    typedef T value_type;
    typedef integral_constant<T, val> type;
};
```

- **Type** T must be integral
- Specializations:
  ```cpp
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

All type predicates further (properties of the forms `is_XXX` `has_XXX`) derive either from `true_type` or `false_type`.

**Example of use**

```cpp
template <typename T>
struct A {
    T _t;
    void save() {
        if (is_pointer<T>::value)
            _t->save();
        else
            _t.save();
    }
};
```

**Dispatching at compile time**

```cpp
template <typename T>
struct A {
    T _t;
    void save(true_type&) { _t->save(); }
    void save(false_type&) { _t.save(); }
    void save() {
        save(
            integral_constant<bool, true>(),
            integral_constant<bool, false>(),
            is_pointer<T>::value
        );
    }
};
```

Check at run-time!

Anyhow, it does not compile!

- Compiler requires that the 2 branches of `if` be compilable

**Property Query**

- **Primary categories** (bool value)
  - `is_void<T>`
  - `is_integral<T>`
  - `is_float<T>`
  - `is_array<T>`
  - `is_pointer<T>`
  - `is_reference<T>`
  - `is_member_pointer<T>`
  - `is_member_function_pointer<T>`
  - `is_enum<T>`
  - `is_union<T>`
  - `is_class<T>`
  - `is_function<T>`
- **Composite categories** (bool value)
  - `is_arithmetic<T>`
  - `is_fundamental<T>`
  - `is_object<T>`
  - `is_array<T>`
  - `is_member_pointer<T>`
  - `is_pointer<T>`
  - `is_reference<T>`
  - `is_member_pointer<T>`
- **Type properties** (bool value)
  - `is_const<T>`
  - `is_volatile<T>`
  - `is_signed<T>`
  - `is_unsigned<T>`
  - `is_parenthesized<T>`
  - `is_abstract<T>`
  - `is_constructible<T>`
  - `is_copyable<T>`
  - `is_copy_assignable<T>`
  - `is_assignable<T, U>`
  - `has_virtual_destructor<T>`
- **Type properties** (unsigned value)
  - `alignment_of<T>`
  - `rank<T>`
  - `extend<T>`

**Relations and Transformations**

- **Relations** (bool value)
  - `is_same<T1, T2>`
  - `is_base_of<Base, Derived>`
  - `is_convertible<From, To>`
- **Transformation** (type)
  - `remove_const<T>`
  - `remove_volatile<T>`
  - `remove_cv<T>`
  - `add_const<T>`
  - `add_volatile<T>`
  - `add_cv<T>`
  - `remove_reference<T>`
  - `add_reference<T>`
  - `remove_pointer<T>`
  - `add_pointer<T>`
  - `remove_extent<T>`
  - `remove_all_extents<T>`
  - `aligned_storage_LENGTH<Align>`
  - etc.
Boost and TR1 Type Traits
Exemple: Call Parameter (a.k.a. Call Traits)

```cpp
template <typename T>
struct A { unsigned f(T t) { } };

// If T is a scalar, passing by value is good, but if T's are big objects
// passing by reference (to a constant) is better

template <typename T>
struct A { unsigned f(T t) { } };

// This is not a complete solution: see boost::call_traits
```

Control of Template Instantiation:

std::enable_if (1)

```cpp
template <typename Iter>
typename Iter::value_type
sum(Iter first, Iter last) {
    typedef typename Iter::value_type T;
    T s = T();
    for (Iter it = first; it != last; ++it)
        s += *it;
    return s;
}
```

Suppose that we wish to use this algorithm only when T is a numeric type.

How to remove `sum()` from candidate functions when T is not numeric?

- Static assertions would provoke compilation errors
- Conditional compilation does not work
- But we can provoke a substitution failure... SFNAE (which is not an error...)

Control of Template Instantiation:

std::enable_if (2)

```cpp
// enable_if makes it possible to control overloading resolution

// Beware: boost::enable_if and std::enable_if(C++11) are slightly different: here we use std::enable_if

// The enable_if templates from C++11

// template <typename Iter>
typename Iter::value_type
sum(Iter first, Iter last) {
    typedef typename Iter::value_type T;
    T s = T();
    for (Iter it = first; it != last; ++it)
        s += *it;
    return s;
}
```

Control of Template Instantiation:

std::enable_if (3)

```cpp
// Now, sum() is a valid candidate if and only if T is an arithmetic type...
```
Control of Template Instantiation:

std::enable_if (5)

- Removing a candidate from the overloading set (1)
  - Trying to print a whole container?
    ```cpp
template <typename Container>
ostream& operator<<(ostream& os, const Container& cont)  
{
  typename Container::const_iterator Iter;
  for (Iter it = cont.begin(); it != cont.end(); ++it)
  
xos << *it << ' ';
  return os;
}
```
  - This does not work!
    - This function is always candidate, since Container can be any type
      - hence possible compile-time errors
    - Some containers already have operator<< (e.g. string), hence ambiguity

Control of Template Instantiation:

std::enable_if (6)

- Removing a candidate from the overloading set (2)
  ```cpp
  template <typename Container>
  typename std::enable_if<
    not is_same<Container, string>::value,
  ostream&
  >::type
  operator<<(ostream& os, const Container& cont)
  {
    // Same as before...
  }
  ``
  - This removes the function from the overload set when the container is string, allowing to correctly lookup string std::operator<<
  - However this function is still candidate for all types (except string), hence possible compile-time errors
  - We need to remove the function also when Container is not a container type: e.g. it has no type member value_type or const_iterator

Control of Template Instantiation:

std::enable_if (7)

- Introspection for a member type (1)
  ```cpp
  template<typename C>
  struct has_type_<value_type>
  {
    struct _yes {char c;};
    struct _no {_yes a[2];};
    template <typename T>
    static _yes _dummy(typename T::value_type*);
    template <typename >
    static _no _dummy(...);
    static const bool value = sizeof(_dummy<C>(0)) == sizeof(_yes);
  };  
  The sizeof trick: Nothing is evaluated (but types and purely type dependent operations such as overloading) within sizeof
  - Hence function _dummy needs not be defined
  - Note that constant 0 is convertible into a pointer to member
  ``

Control of Template Instantiation:

std::enable_if (8)

- Introspection for a member type (2)
  ```cpp
  #define DEF_HAS_MEMBER_TYPE(TYPE) \
  template<typename C>  \
  struct has_type_##TYPE \
  { \
  struct _yes {char c;}; \
  struct _no {_yes a[2];}; \
  \
  template <typename T> \
  static _yes _dummy(typename T::TYPE*);
  \
  template <typename > \
  static _no _dummy(...);
  \
  static const bool value = sizeof(_dummy<C>[0]) == sizeof(_yes);
  };
  ```
  - Hence function _dummy needs not be defined
  - Note that constant 0 is convertible into a pointer to member
Removing a candidate from the overloading set (3)

```
DEF_HAS_MEMBER_TYPE(value_type); // struct has_type_value_type
DEF_HAS_MEMBER_TYPE(const_iterator); // struct has_type_const_iterator

template <typename Container>
typename std::enable_if<
    not is_same<Container, string>::value
    and has_type_value_type<Container>::value
    and has_type_const_iterator<Container>::value,
    ostream&
>::type
operator<<(ostream& os, const Container& cont)
{
    // Same as before...
}
```

Now the function appears in the overload set only if type Container is not string and if it has member types value_type and const_iterator

STL (2003) function objects and function adaptors

```
mem_fun, mem_fun_ref, ptr_fun, bind1st, bind2nd...
```

Boost and TR1 function adaptors

```
bind, mem_fn
```

Reference wrappers: ref, cref

Direct manipulation of function objects: function

C++11 lambda expressions

```
list<int> l;
for_each(l.begin(), l.end(), printer<int>());
```

```
list<int>::iterator it = find_if(l.begin(), l.end(),
    bind2nd(greater<int>(), 5));
```

```
struct Figure {
    void draw() const;
    int rotate(int angle);
};
```

```
dequeue<Figure *> dq
for_each(dq.begin(), dq.end(), mem_fun<&Figure::draw>);
```

```
list<int> li;
transform(dq.begin(), dq.end(), back_inserter(li),
    bind2nd(mem_fun<&Figure::rotate>, 3.141592));
```
High Order Programming

STL Function Objects and Adaptors (3)

- **Drawbacks of STL (2003) function adaptors**
  - Distinction between different natures of functions
    - regular functions
    - member functions
    - well-formed function objects
  - Functions must have no more than 2 parameters
  - Difficult to compose functions
  - Terrible syntax
    ```cpp
    bind2nd(mem_fun(&Figure::rotate), 3.141592));
    ```
  - Possible problems with parameters passed by reference...
  - Problem with containers of smart pointers

Examples of function objects (φ) created by `bind`

```cpp
int f(int a, int b) { return a - b; }

bind(f, _1, 3) → φ(x) = f(x, 3)
bind(f, 4, 3) → φ(y) = f(4, 3)
bind(f, _1, _2) → φ(x, y) = f(x, y)
bind(f, _2, _1) → φ(x, y) = f(y, x)
```

```cpp
int n = 4, m = 3;
binderf, _1, 3)(n) == 1
bindf, _1, _2)(n, m) == 1
bindf, _1, _1)(n) == 0
bindf, _2, _1)(n, m) == -1
bindf, _1, _1)(3) → does not compile in C++03
bindf, _1, _1)(n + 1) → does not compile in C++03
```

High Order Programming

Boost and tr1 bind Adaptor (1)

```cpp
bind<Return>(f, p1, p2, ..., pn);
```

- `<Return>` is optional if the compiler can deduce it
  - if present, `Return` designates the return type of the function object

- The callable `f` may be
  - a pointer to a regular function or a function object with `n` parameters
  - a pointer to an instance member function with `n-1` parameters (not accounting for this which will become the first parameter of the function object)

- The parameters `p2` may be
  - either an expression which will be bound to the corresponding parameters of `f`
  - or a placeholder such as `_1`, `_2`, `_3`... which corresponds to a free variable, thus a parameter of the produced function object

- The function object produced will have
  - as many parameters as the number of different placeholders used: they must be used in order `_1`, `_2`, ... (without jumps)
  - its parameters passed by reference to a variable: thus the effective parameter cannot be a literal value nor an expression yielding a value (no longer true in C++11)

Usage of `bind`

```cpp
struct Figure {
  void draw() const;
  int rotate(double angle);
};

dqueue<Figure *> dq;
for_each(dq.begin(), dq.end(), bind(&Figure::draw, _1));

list<int> li;
transform(dq.begin(), dq.end(), back_inserter(li),
  bind(&Figure::rotate, _1, 3.141592));
```
Function composition with bind

struct Figure {
    void draw() const;
    int rotate(double angle);
    unsigned diameter() const;
};

decque<Figure *> dq;
sort(dq.begin(), dq.end(),
    bind(less<unsigned>(),
        bind(&Figure::diameter, _1),
        bind(&Figure::diameter, _2)));

Binding data members

map<int, string> m;
typedef map<int, string>::value_type pair_type;
void print_string(const string& s);
for_each(m.begin(), m.end(),
    bind(sprint_string,
        bind<string>(&pair_type::second, _1)));
**High Order Programming**

**Boost and TR1 bind Adaptor (8)**

- **Compatibility of `bind` with previous STL**
  - STL 2003 `functors` defines nested types for the result type and the parameter type(s)
  - The STL 2003 `functionals` expect and use these nested types
    ```cpp
    list<char *> lcstr;
    int c = count_if(lcstr.begin(), lcstr.end(),
        not1(bind2nd(ptr_fun(strcmp), "hello")));
    ``
  - `bind` does not define these nested types
    ```cpp
    int c = count_if(lcstr.begin(), lcstr.end(),
        not1(bind(strcmp, _1, "hello")));
    error: no type named 'argument_type' . . .
    ```

- **Problem with `bind`**
  - Creating and storing a function object to use it later?
    ```cpp
    void foo(int a, int b) { . . . }
    ?? fobj = bind(foo, _1, 3); // type unknown
    fobj(5); // ??
    - The type is determined with respect to the context of call
    ```

- **Use Boost or TR1 `function`**
  ```cpp
  function<void(int)> fobj = bind(foo, _1, 3);
  fobj(5);
  - Any sort of callable can be converted into an instance of function<>
    (with the suitable signature)
  - A powerful replacement for pointer to functions...
  ```

- **Construction of function objects using `function`**
  ```cpp
  function<int(int, int)> f;
  - The default constructor yields an empty function object
  if (f.empty()) ... // true
  if (not f) ... // true
  f(3, 3); // throw bad_function_call
  - The function object can be initialized with
    - a regular function or a function object of the same signature
    - a reference wrapper to a function object of the same signature
    - a member function the first parameter of which must be the class of this member
  - The function objects created with `function` are copiable
  - Up to 10 parameters (library configuration)
  - Some compilers do not accept the above syntax for the signature (g++ does)
    - Alternative forms:
      ```cpp
      function1<void, int> f1; function2<void, int, int> f2; ...
      ```

**High Order Programming**

**Boost and TR1 `function` (1)**

- **Examples of construction of function objects using `function`**
  ```cpp
  int myplus(int, int);
  function<int(int, int)> f1 = &myplus;
  function<int(int, int)> f2 = std::plus<int>();
  int a = f1(3, 4);
  int b = f2(a, 5);
  f1 = f2;
  struct Figure {
    double rotate(double);
    // ...
  };
  function<double(Figure, double)> f3 = &Figure::rotate;
  double x = f3(Figure(), 3.5);
  function<double(Figure*, double)> f4 = &Figure::rotate;
  x = f4(new Figure(), 3.5);
  ```
Function objects with state
As usual, function objects may have internal attributes
Thus they accumulate information each time they are called
However always remember that function objects are copied by value

```
struct Accumulate {
    int _total;
    Accumulate() : _total(0) {}
    int operator()(int a) {
        _total += a;
        return _total;
    }
};
Accumulate acc;
```

```
function<int(int)> facc1 = acc;
function<int(int)> facc2 = acc;
cout << facc1(10) << ' ';
cout << facc2(10) << endl;
// 10 10
cout << acc._total << endl;
// 0
```

function and bind
These two functions are compatible

- One can create a function<...> as the result of bind
  `function<int(int)> f1 = bind(plus<int>(), _1, 3);`
- One can also bind the arguments of a function<...>
  `function<int()> f2 = bind(f1, 5);`
### Advantages of function
- Replacement of pointers to functions
- Compatible with (nearly) all ways of representing something like a function in C++
- Compatible with STL 2003 for 1 and 2 argument functions
- Function objects may have state

### Drawbacks
- Be careful with copying function objects
- Cost: 4 times bigger than a regular pointer to function
- Does not solve the overloading problem

```cpp
int g(int i);
double g(double x);
function<int(int)> fg = &g; // does not compile
function<int(int)> fg = static_cast<int(*)(int)>(&g);//!!!
```

### Problems with bind
- The syntax of `bind` is improved but still ugly
- It is still necessary to define dedicated function objects (at global or namespace scope)

```cpp
struct printer {
    template <typename T>
    void operator()(const T& t) {cout << t << ' ';
}
}
```

```cpp
list<double> lx;
for_each(lx.begin(), lx.end(), printer());
```

### Two different and incompatible implementations of lambda
- `boost::lambda`, a library implementation (obsolete)
- `lambda expressions and closures` as part of core C++11

### Example of C++11 lambda expression
```
list<double> lx;
for_each(lx.begin(), lx.end(),
    [] (double x) { cout << x << ' '; });
```

### Anatomy of a lambda expression
```
[...](P_1 p_1, ..., P_n p_n) -> T { ... }
```

**Capture of “more global” variables**  
**Parameter list**  
**Return type**  
**Body**

- Defines a function: `T anonymous(P_1 p_1, ..., P_n p_n)`
- Part of syntax in *Magenta* is optional

### Return type
- Optional if the compiler can deduce it unambiguously

### Capture list
- Make variables in local scope available within the lambda body
- Several forms:
  - `[]` do not capture anything
  - `[=]` capture everything, by value
  - `[&]` capture only `x, y` by value
  - `[&x, &y]` capture only `x, y` by reference
  - `[&x, &y]` capture `x` by reference, `y` by value, and nothing else...
Capture list example

```cpp
int suffix_list(list<string>& ls, int suffix) {
    int local_suffix = suffix;
    transform(ls.begin(), ls.end(), ls.begin(),
              [local_suffix] (const string& s) {
                  ++local_suffix;
                  return s + boost::lexical_cast<string>(local_suffix);
              });
    return local_suffix;
}
```

Note that this must be captured to be able to access class members

```cpp
struct A {
    int _a;
    A() : _a(0) {}    
    void f() {
        cout << "[" << (return ++_a) << "]" << endl;
    }
};
```

Evaluation of High Order Programming in TR1 and C++11

- **bind, and function** are a real improvement over STL 2003 constructs
  - Respect the STL philosophy
  - Simple to use
  - Powerful (e.g., function composition)
- **However,**
  - delicate incompatibilities between `bind` and STL 03 function objects
  - still problems with overloaded functions
- **C++11 lambdas are a totally new construct**
  - Powerful, simple enough
  - The function is defined where it is needed
    - Possibility of local function
    - Compatibility with `bind/function`

Containers and Algorithms

- **TR1 containers (also in Boost)**
  - `tuple`
  - `array`
  - Unordered associative containers
- **Boost “polymorphic” containers**
  - `any`, `variant`
  - `pointer containers`
- **Other Boost containers**
  - multi-arrays, multi-index, bidirectional maps
  - property maps
  - intrusive containers
Extension of the pair<T, U> structure
tuple<T1, T2, ..., Tn>
  Ti are types (possibly references)
  n ≥ 0 (guaranteed maximum n ≤ 10)
    • no real maximum if variadic templates supported

Tuples properties
  Copiable, assignable, provided T1 are
  Comparable (==, <, ...), provided T1 are
  tuple<T, U> interoperable with pair<T, U>

Examples of tuples
typedef tuple<int, double> Doublet;
Doublet t1(3, 2.5);
Doublet t2 = t1; // Copy constructor
typedef tuple<int&, double> Tref;
int i = 0;
Tref tr1(i, 3.5);
tr1 = t2; // OK
assert(i == 3);

Tuple accessors
tuple_size<Tuple>::value
  • Number of elements in the Tuple type
typed tuple<int, string, double> Triplet;
static_assert(tuple_size<Triplet>::value == 3, "");
tuple_element<1, Tuple>::type
  • Type of the element of index I in the Tuple type
tuple_element<1, Triplet>::type s; // s is a string
get<I>(t)
  • Return a reference (possibly const) onto the element of index I in the Tuple t
Triplet t; // all zeroes of their type
get<2>(t) = 3.5; // t == (0, "", 3.5)

Tuple convenience functions
make_tuple(t1, t2, ..., tn)
  • make a tuple<T1, T2, ..., Tn> where Ti is the type of ti
tuple<int, double, string> t = make_tuple(3, 4.5, "hello");
tie(t1, t2, ..., tn)
  • equivalent to make_tuple(ref(t1), ref(t2), ..., ref(tn)) where ref is the C++ (non const) reference wrapper
  • ti must be (non const) references
  • Usually used on the left of an assignment
int i;
string s;
tie(i, s) = make_tuple(3, "hello"); // i == 3, s == "hello"
  • Special marker ignore prevents copy for the corresponding element
tie(i, ignore) = make_tuple(3, "bye"); // string unchanged
Defining a function `print_tuple` is easy with some template metaprogramming:

```
template <typename Tuple, int I>
struct print_helper {
    static void print(ostream& os, const Tuple& t) {
        print_helper<Tuple, I - 1>::print(os, t);
        os <<", " << get<I>(t);
    }
};

template <typename Tuple>
struct print_helper<Tuple, 0> {
    static void print(ostream& os, const Tuple& t) {
        os << get<0>(t);
    }
};

template <typename Tuple>
void print_tuple(ostream& os, const Tuple& t) {
    os << "( ";
    if (tuple_size<Tuple>::value > 0)
        print_helper<Tuple, tuple_size<Tuple>::value - 1>::print(os, t);
    os << " )";
}
```

Defining stream operator `<<` for tuples:

```
template <typename Tuple, int I>
struct print_helper {
    static void print(ostream& os, const Tuple& t) {
        print_helper<Tuple, I - 1>::print(os, t);
        os <<", " << get<I>(t);
    }
};

template <typename Tuple>
void print_tuple(ostream& os, const Tuple& t) {
    os << "( ";
    if (tuple_size<Tuple>::value > 0)
        print_helper<Tuple, tuple_size<Tuple>::value - 1>::print(os, t);
    os << " )";
}
```

Using pairs as tuples:

```
pair<int, string> p(10);
get<1>(p) = "hello";
```

```
int i;
string s;
tie(i, s) = make_pair(12, "bye");
assert(tuple_size<pair<int, string>>::value == 2);
```

Using tuples as pairs:

```
tuple<int, string> t(make_pair(12, "bye"));
```

Simple array class:

```
array<int, 4> a2 = {{1, 2, 3, 4}};
```

```
array<int, 4> a2 = (1, 2, 3, 4); // C++11?
```
TR1 Containers
Unordered Associative Containers

- `unordered_map<K, T>, unordered_multimap<K, T>`
  - Fast search and retrieval (average O(1), worst case linear)
  - Does not require an order relation over K
  - Use hash coding
    - Special hash<T> functor with predefined specializations for usual types (in `<functional>`)
  - Same interface as the corresponding ordered associative collections
    - plus some extra functions or parameters to handle hashing configuration

Boost “Polymorphic” Containers

- Objects of different types in the same container
  - `boost::variant`
    - Discriminated union of a fixed and finite set of types
    - Type safe storage and retrieval
  - `boost::any`
    - Discriminated union of an unbounded set of types
    - Typesafe storage and retrieval

- Pointer containers
  - Alternative to using containers of `shared_ptr`

Boost “Polymorphic” Containers

- `boost::variant` (1)

  - C and C++ unions are not discriminated
    - `union Real_Int {double x; int i;}`
    - `Real_Int u;`
    - `u.x = 3.141592;`
    - `int a = u.i; // a == -57999238 ???`
  - C++ unions cannot contains members with constructors
    - `union {int i; string s;}; // NO`
  - Boost variant class template
    - `variant<double, int> v(3.141592);`
    - `int a = get<int>(v); // throw boost::bad_get`
    - `int *pa = get<int>(&v); // pa is null`

- `boost::variant` (2)

  - Properties of `boost::variant`
    - Variants are copiable (if their elements are)
    - Variants are comparable (if their elements are)
    - Implicit conversions are possible only if unambiguous
    - `which()`
      - return the index of the type currently occupying the variant
    - `type()`
      - return the `type_info` of the type currently occupying the variant
    - `variant<double, int> v(3);`
    - `assert(v.which() == 1);`
    - `assert(v.type() == typeid(int));`
Visitor for boost::variant

```cpp
struct ostream_visitor : public static_visitor<void> {
    ostream& _os;
    ostream_visitor(ostream& os) : _os(os) {}

    template <typename T>
    void operator()(T& t) const {
        _os << t;
    }
};
```

- Note that passing `t` by reference avoids most implicit conversions

```cpp
variant<double, int, char> v('a');
apply_visitor(ostream_visitor(cout), v); // -> 'a'
v = 2.71828;
apply_visitor(ostream_visitor(cout), v); // -> 2.71828
```

Boost “Polymorphic” Containers

boost::any

- Store a value of an almost arbitrary type
- Allow safe storage, retrieval, and copy
- A sort of unbounded union
- Constraints on the type of value
  - Copy constructible
    - This one is mandatory
  - Assignable
    - If not assignable, strong exception guarantee may be lost
  - Non-throwing destructor
    - As always!

```cpp
#include <boost/any.hpp>
class any {
public:
    any();
    any(const any&);
    ~any();
    any& swap(any&);
    any& operator=(const any&);
    template <typename V> any(const V&); // conversion
    template <typename V> any& operator=(const V&); // conversion
    bool empty() const;
    const type_info& type() const;
};
```

Extracting the current value

```cpp
any a;
T t = any_cast<T>(a);
- The previous expression throws exception `bad_any_cast` if a does not contain a value of type `T`
T *pt = any_cast<T>(&a);
- The previous expression returns 0 if a does not contain a value of type `T`
```

Example

```cpp
any a;
a = string(“hello”); // see next slide
a = 12;
int i = any_cast<int>(a); // OK
string *s = any_cast<string>(&a); // null
```
## Boost “Polymorphic” Containers

### `boost::any (4)`

- **Putting pointers into any**
  - any is not considered empty when holding a null pointer
  - any destructor won’t destroy the object pointed to
  - polymorphism is not honored
- **Be careful when putting char* into any**
  - Put string instead
- **By contrast, it is safe to put shared_ptr into any**

### Boost “Polymorphic” Containers

#### Pointer Containers

- Alternative to using containers of `shared_ptr` when
  - sharing is not needed, or
  - shared pointer overhead is not acceptable
- **Pointer containers contain heap allocated objects**
  - They take ownership of them, and destroy them when needed
- **Performance optimization**
  - but requires strict ownership

## Other Boost Containers

### Other Boost Containers (1)

- **Multi-arrays**
  - Arrays with more than 1 dimension
  - Contiguous in memory
- **Multi-index**
  - Allow to index containers with different strategies
- **Bidirectional maps**
  - `bimap<X, Y>`: two opposite `std::maps`
  - Interoperable with `std::map(left and right views)`

### Other Boost Containers (2)

- **Intrusive containers**
  - Store objects themselves, not copies
  - **Advantages**
    - No memory management
    - Less memory used
    - An object may belong to several containers simultaneously
    - Fast iteration
    - Better exception guarantee
    - Better predictability on insert/erase (no memory...)
  - **Drawbacks**
    - The objects must already contain some predefined members, e.g., the next and previous members for a doubly linked list
    - No automatic life time management
    - Intrusive containers are non copyable and non assignable
    - Analyzing thread safety is harder with intrusive containers
Other Boost Containers (3)

- **Property maps**
  - A generic representation of (name, value) pairs

- **Circular buffer**
  - STL compliant: same properties as sequences
  - Contiguous memory
  - Fast insertion, removal, and iteration

- **Dynamic bitset**
  - Dynamic extension of std::bitset

- **GIL, the Generic Image Library**
- **GRAPH, generic components and algorithms**

Advanced C++ Libraries and Introduction to Template Metaprogramming

Part 6

String Utilities and Regular Expressions

**Synopsis**

```cpp
template <typename Source, typename Target>
Target lexical_cast(const Source& arg);
```

Convert `arg` into type `Target` using stream operators

- Source must be `Output Streamable` (operator `<<`)
- Target must be `Input Streamable` (operator `>>`). Default and Copy Constructible

If conversion does not work, throw `bad_lexical_cast`

**Example**

```cpp```
#include <boost/lexical_cast.hpp>
int i = lexical_cast<int>("-123"); // atoi()
string s = lexical_cast<string>(i + 2); // itoa??
```
Miscellaneous Boost Libraries

Format

**Motivation**
- Type-safe printf-like format operations
- Much richer than printf
  - Works with any (Output Streamable) type
  - Possibility of reordering elements
  - More formatting options...

**Example**

```
#include <boost/format.hpp>
boost::format f("%05d %5.2f %s\n");
float x = 5.3;
string s = "hello";
out << f % 3 % x % s;
```

String Algorithms

**Namespace boost::algorithm**
- A lot of supplementary algorithms for strings
  - Convert strings to upper/lower case
  - Trim strings
  - Search for, replace, or erase sub-strings
  - Split strings, join substrings
  - Iterators on substrings
  - Interoperability with regular expressions

Tokenizer (1)

**String tokenization**
- Splitting a string into substring with respect to character or substrings considered as separators

**Many ways to do it using Boost**
- regular expressions token iterator (see further)
- split() algorithm in string algo
- string tokenizer
  - the result of tokenization is a sort of iterable container, compatible with the STL

Tokenizer (2)

**String tokenization: example with boost::tokenizer**

```
#include <boost/tokenizer.hpp>
char_separator<char> sep(";");
string s = getenv("PATH");
tokenizer<char_separator<char>> tok(s, sep);
list<string> li;
copy(tok.begin(), tok.end(), back_inserter(li));
```

Separators are individual characters
- The separators can be dropped (by default) or kept in the tokens
- Empty fields are dropped by default, but can be kept:

```
char_separator<
```
Regular expression support in TR1 and Boost
Regular expression syntax(es)
Defining regular expressions
Matching, searching, and replacing
Regular expression iterators
Regular expressions and UNICODE

Both Boost and TR1 define regular expression support (regex)
As of gcc-4.8.x, GNU compilers do not implement regex
If Boost is present, tr1::regex is in fact boost::regex
The definition in TR1 and the Boost implementation are almost identical
Boost supports more regular expression syntaxes than tr1
As an extension, Boost regex supports UNICODE...
provided that the (free) IBM library ICU is installed
We use boost::regex in the following examples
#include <boost/regex.hpp>

TR1 supports 6 different (similar but incompatible) syntaxes
- POSIX Basic Regular Expression (BRE) and Extended Regular Expression (ERE) syntaxes
- POSIX awk, grep, and egrep syntaxes
- ECMAScript regular expression syntax, which is the default (and the richest)
We use ECMAscript
We do not describe the whole ECMA syntax!
See the documentation for complements and differences, or Pete Becker’s book

Usual elements
- Individual characters: a b ... A B ... ; : ...
- Wildcard: . (dot) stands for any character except newline
- Character class: [a-zA-Z0-9] [^a-zA-Z0-9]
- Repetition: a* a? [0-9]+ [ab]{3} [0-9]{1,5}
- Group: (ab)*
- Alternative: (a|b)+
- Back reference: ([a-z]*)([0-9]+)\2\1
Predefined character classes

- Digits: \d \[":digit:"\]
- Non digit: \D \[":d:"\]
- Spaces: \s \[":space:"\]
- Non space: \S \[":s:"\]
- Alphanum: \w \[":w:"\]
- Non alphanum: \W \[":W:"\]

The exact definition depend on regex traits (in particular on the encoding and locale)

Examples of regular expressions within C/C++ literal strings

- Beware: within a literal string, character \ must be doubled
  - "[a-zA-Z_][a-zA-Z0-9_]*" identifier
  - "[[alpha]][[alpha]][[alnum]][*]" idem
  - "[[alpha]][[alnum]][*]" idem
  - "[-+]\d*(\.\d*)?" a float in fixed representation
  - "((\w+))\s+\1" a duplicated word separated by spaces
  - "((\w+)|\d+)+\1" a duplicated word or integer separated by spaces

Classes to encapsulate (compile...) a regular expression

```cpp
template <typename charT, 
typename traits = regex_traits<charT>>
class basic_regex;
typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

traits define necessary internal operations on charT

Construction of a regex from a C or C++ string

```cpp
regex re(\"[a-zA-Z_][a-zA-Z0-9_]*\")
```

These constructors are explicit

A second parameter (flags) is optional: it mainly allows to choose the syntax of regular expressions and some parameters such as case sensitiveness...

```cpp
regex re(\"[a-zA-Z_][a-zA-Z0-9_]*\", 
regex_constants::perl);
```

Note on greediness

- By default, repetition operators * and + choose the longest match
  - thus, with string "hello new world", the group (\"\w+\s\") matches the whole word hello plus the following space and not, for instance, h followed by no space, or he followed by no space...
  - It is possible to require a non greedy match (see later)
TR1 and Boost Regular Expressions (regex)

Matching (2)

- Parameters of regex_match
  - Many forms of regex_match
    - Using char*, strings, iterators...
  - The last parameter (optional) contains match flags
    - They correspond to handling of some special conditions such as
      beginning or end of line, or depending on particular forms of regular
      expressions
  - All other matching functions take the same sort of flags as last
    parameter
  - Some forms include a match_results parameter to match sub-
    expressions (see regex_search)

Searching for sub-patterns: regex_search

```cpp
string s = "10u, 11u, 12, 13U, 14U, 15u";
regex re("(\d+u)|(\d+U)\";
bool b = regex_search(s, re); // true
```

Identifying sub-patterns

```cpp
smatch m; // an array of sub-matches for string
if (regex_search(s, m, re)) {
    if (m[1].matched)
        cout << "use form in u
    if (m[2].matched)
        cout << "use form in U
}
```

- The search stops at the first match
- Output => use form in U

Match results and sub-matches

Class match_results

- An array of sub-matches
  - m[0] the current match
  - m[1] sub-match for sub-expression 1 in regex
  - m[2] sub-match for sub-expression 2 in regex, etc.

- Several forms depending on the type of strings that are scanned
  - cmatch for char*
  - wcmatch for wchar_t*
  - smatch for std::string
  - wsmatch for std::wstring

Class sub_match

- Each sub-match is a pair of iterators delineating the sub-match, plus a
  boolean matched

Match results and sub-matches (cont.)

```cpp
string s = "foo = 12; bar = 0;";
regex re("(\w+) = (\d+)\";
smatch m;
bool b = regex_search(s, m, re);
```

```cpp
m[0] = foo;
m[1] = 12;
m[2] = bar;
m[3] = 0;
```
**TR1 and Boost Regular Expressions (regex)**

**Searching (4)**

- **Iterating on sub-matches**

  ```cpp
  string s = "10U, 11u, 12, 13U, 14U, 15u";
  regex re("(\d+u)|(\d+U)\d+U")
  smatch m;
  int cntu = 0, cntU = 0;
  string::const_iterator it = s.begin();
  string::const_iterator end = s.end();
  while (regex_search(it, end, m, re)) {
    if (m[1].matched)
      cntu++;
    else
      cntU++;
    it = m[0].second;
  }
  ```

- **Revisiting greediness: greedy repetition**

  ```cpp
  string s = "aaa 12 bbbb 163";
  smatch m;
  regex re(".*\d+U")
  string::const_iterator it = s.begin();
  string::const_iterator end = s.end();
  while (regex_search(it, end, m, re)) {
    if (m[1].matched) // useless
      cout << m[1].str() << endl;
    it = m[0].second;
  }
  Output =>
  3
  ```

**TR1 and Boost Regular Expressions (regex)**

**Searching (5)**

- **Revisiting greediness: non greedy repetition**

  ```cpp
  string s = "aaa 12 bbbb 163";
  smatch m;
  regex re(".*?\d+U")
  string::const_iterator it = s.begin();
  string::const_iterator end = s.end();
  while (regex_search(it, end, m, re)) {
    if (m[1].matched) // useless
      cout << m[1].str() << endl;
    it = m[0].second;
  }
  Output =>
  12
  163
  ```

**TR1 and Boost Regular Expressions (regex)**

**Replacing**

- **Text substitution: regex_replace**

  ```cpp
  regex re("(colo)(u)(r)\", regex::icase);
  string s = "Colour colours color Colourize";
  cout << regex_replace(s, re, "$1$3")
  Output =>
  Color colors color Colorize
  ```
Using STL algorithms on sub-match collections

```cpp
struct regex_sum {
    int _sum;
    regex_sum() : _sum(0) {} // M will be a match results

template <typename M>
void operator()(const M& m) {
    _sum += lexical_cast<int>(m[1].str());
}

string s = "1, 2, 3, 4, 5, 6, 7, 8, 9, 10";
regex re("(\d+),?");
regex_iterator it(s.begin(), s.end(), re);
regex_iterator end;
foreach(it, end, regex_sum())._sum << endl; // 55
```

Regex token iterators (regex_token_iterator)

```cpp
Regex token iterators (regex_token_iterator)

Similar to regex_iterator but can be used to split a string according to separators which are themselves regular expressions

```cpp
vector<string> split(const string& s, const string& sep) {
    regex resep(sep);
    sregex_token_iterator it(s.begin(), s.end(), resep, -1);
    sregex_token_iterator end;
    vector<string> res;
    while (it != end)
        res.push_back(*it++);
    return res;
}
```

```cpp
vector<string> v = split("aaa;--bbbb::;ccccc;;;-", "[-;:]+ ");
v => aaa bbbb cccc
```

Unicode and C++

C++03 does not know about Unicode
- wchar_t is not guaranteed to be wide enough for Unicode representation
- There is no guarantee that the system will consider wchar_t as Unicode

There exist third party Unicode libraries
- Boost is able to use the ICU Unicode library from IBM
- UTF-8, UTF-16, or UTF-32 are supported through class boost::u32regex

C++2011 knows a (little) about Unicode
- It can represent literal strings in UTF-8, UTF-16, or UTF-32
- But it lacks general facilities for conversions, iterating, etc.

A simple example of Unicode regex

```cpp
#include <boost/regex/icu.hpp>
// other usual includes
int main() {
    u32regex re =
        make_u32regex("(/?[^\/]*)*/[^/]*");
    smatch m;
    string s = "/été/être";
    if (u32regex_match(s, m, re) {
        cout << m[2].str() << endl; // => être
    }
```
Part 7

Boost Serialization Library

Serialization Library

Overview
- A simple example
- Pointer tracking
- Polymorphic class hierarchies
- STL collections

Serialization Library

Overview: Motivation

- Save and restore C++ objects to or from files (archives)
- Minimal modification of classes
- Handle object sharing
  - Pointer tracking and sharing
- Handle polymorphism
- Compatible with STL collections

Serialization Library

Overview

- Archive types
  - Text archives (*.txt)
    - A compact, hardly readable format
  - XML archives (*.xml)
    - Readable (?) and verbose
    - More information than in a text archive
  - Binary archive (*.bin)

- Data streams
  - Output stream: used to save data
    - `oar << data;`
  - Input data stream: used to restore (load) data
    - `iar >> data;`
  - Operator &: save or restore according to the type of ar
    - `ar & data;`
Serialization Library
A Simple Example (1)

- A class to serialize

```cpp
class Date {
    private:
        int _year, _month, _day;

    public:
        Date() : _year(0), _month(0), _day(0) {};
        Date(int y, int m, int d) : _year(y), _month(m), _day(d) {};
        // other functions and operators
};
```

Serialization Library
A Simple Example (2)

- Intrusive mode

```cpp
#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/xml_iarchive.hpp>

#define NVP(x) BOOST_SERIALIZATION_NVP(x)

class Date {
    friend class boost::serialization::access;
    private:
        int _year, _month, _day;
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & NVP(_year) & NVP(_month) & NVP(_day);
    }
    public:
        Date() : _year(0), _month(0), _day(0) {};
        Date(int y, int m, int d) : _year(y), _month(m), _day(d) {};
        // other functions and operators
};
```

Serialization Library
A Simple Example (3)

- Non intrusive mode

```cpp
#include <boost/archive/xml_oarchive.hpp>
#include <boost/archive/xml_iarchive.hpp>

#define NVP(x) BOOST_SERIALIZATION_NVP(x)

class Date {
    private:
        public:
            int _year, _month, _day;
        public:
            Date() : _year(0), _month(0), _day(0) {};
            Date(int y, int m, int d) : _year(y), _month(m), _day(d) {};
            // other functions and operators
};
```

Serialization Library
A Simple Example (4)

- Main program (both for intrusive and non intrusive)

```cpp
int main() {
    Date dat(2009, 11, 24);
    {
        // Saving
        ofstream ofs("archive.xml");
        boost::archive::xml_oarchive oar(ofs);
        oar << NVP(dat); // destructors close archive
    }
    // Later... or in another program
    {
        // Restoring
        ifstream ifs("archive.xml");
        boost::archive::xml_iarchive iar(ifs);
        Date dat1; // Note: default construction
        iar >> NVP(dat1);
        assert(dat == dat1); // destructors close archive
    }
}```
Serialization Library
A Simple Example (5)

Format of the XML archive

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>

<boost_serialization
    signature="serialization::archive" version="5">

<dat
    class_id="0" tracking_level="0" version="0">
    <_year>2009</_year>
    <_month>11</_month>
    <_day>24</_day>
</dat>

</boost_serialization>
```

Serialization Library
A Simple Example (6)

Assigning XML tags

When saving/restoring an object into/from an XML archive, one must provide a tag.
This is done through a “name-value pair”

```cpp
ar & make_nvp("my_tag", a_variable);
```

Usually, the tag is identical to the variable name: one can then use a predefined macro

```cpp
ar & BOOST_SERIALIZATION_NVP(a_variable);
```

Since this macro name precludes readability, we always define

```cpp
#define NVP(x) BOOST_SERIALIZATION_NVP(x)
```

Tags are not used for text archives

However, one can still use the previous macro, which ignores the tag in this case

Serialization Library
Polymorphic Class Hierarchies (1)

```cpp
#include <boost/serialization/export.hpp>

class A {
    friend class boost::serialization::access;
    int _a;
public:
    A(int a) : _a(a) {} 
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & NVP(_a);
    }
    virtual void f() {cout << "A::f" << endl;}
};

class B : public A {
    friend class boost::serialization::access;
    int _b;
public:
    B(int a, int b) : A(a), _b(b) {} 
    template <typename Archive>
    void serialize(Archive& ar, const unsigned int version) {
        ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(A);
        ar & NVP(_b);
    }
    virtual void f() {cout << "B::f" << endl;}
};
```

```cpp
BOOST_CLASS_EXPORT(B)
```
int main() {
    A* pa = new B();
    ofstream ofs("poly.xml");
    boost::archive::xml_oarchive oar(ofs);
    oar & BOOST_SERIALIZATION_NVP(pa);
}

A* qa;
{
    ifstream ifs("poly.xml");
    boost::archive::xml_iarchive iar(ifs);
    iar & BOOST_SERIALIZATION_NVP(qa);
}

qa->f(); // virtually calls B::f()

#include <boost/serialization/list.hpp>
#include <boost/serialization/deque.hpp>

list<A> la = {A(), A(), A()};
deque<A *> dqa = {new A(), new B(), new B()};
oar & NVP(la) & NVP(dqa);

list<A> lal;
deque<A *> dqal;
lar & NVP(lal) & NVP(dqal);

for_each(lal.begin(), lal.end(), bind(&A::f, _1));
for_each(dqal.begin(), dqal.end(), bind(&A::f, _1));
// This one is polymorphic!

Motivation: the Boost Metaprogramming Library (mpl)
Characteristics of (template) metaprogramming in C++ mpl
Example: dimensional analysis
Overview of the mpl: compile time containers, iterators, and algorithms
Evaluation of C++ metaprogramming
Motivation – Boost MPL

- Type traits elements, `enable_if` are examples of meta functions
  - A meta-function operates on types (arguments or results)
  - Thus its evaluation is performed at compile-time
  - It may, or may not, be present/used at run time
- To make metaprogramming more general and more accessible, Boost has introduced the `mpl`
  - Metafunctions, lambdas, and high order programming to manipulate and compute with types (and integers) at compile-time
  - Compile time containers, iterators, and algorithms (inspired by the STL)

Characteristics of (Template) Metaprogramming in C++ MPL (1)

- Metadata
  - They are types
  - Thus they are immutable (no assignment)
- Programming style
  - Functional programming
  - No assignment
  - No loops, no primitive control structures... except recursion and template specialization

Characteristics of (Template) Metaprogramming in C++ MPL (2)

- Metadata declaration
  - A nullary metafunction
    ```cpp
    struct mf0 { typedef some_type type; }
    ```
  - A metafunction with 2 parameters
    ```cpp
    template <typename T1, typename T2>
    struct mf2 { typedef some_type type; }
    ```
  - Numerical metafunctions
    ```cpp
    template <typename T> struct nf1 {
      static const int value = ...;
    }
    ```
  - Metaprogramming
    ```cpp
    int f(mf0::type x) ...
    mf2<int, double>::type g() ...
    if (is_pointer<T>::value) ... 
    ```

Characteristics of (Template) Using the MPL

- Namespace `boost::mpl`
- The example programs of this course declare
  ```cpp
  using namespace std;
  namespace mpl = boost::mpl;
  using namespace placeholders;
  ```
  - Thus the elements of the `mpl` are in `mpl`:
    ```cpp
    mpl::vector<...> mpl::int_<3> ... 
    ```
    - On the slides we shall forget `mpl`:
    - The placeholders are simply _1, _2...
- The `mpl` elements reside in a lot (a lot!) of `.hpp` files
Example: Dimensional Analysis

**Purpose**
- Represent physical quantities together with their physical dimensions
  - Mass, length, time, velocity, energy, etc.
- Enforce dimension constraints at compile time
  - Adding a mass to a length must not compile
- Track dimensions at run time
  - The product of a mass by an acceleration is a force

**Idea**
- Represent physical dimensions as types
- Associate its dimension type with each value
- However, type computation is needed because of 2 and 3 above...

---

Example: Dimensional Analysis

**Representing Physical Dimensions (1)**

### Système International (SI)
- 7 fundamental dimensions: mass (kg), length (m), time (s), electrical intensity (A), temperature (K), amount of substance (mol), and luminous intensity (cd)
- Here, we shall limit ourselves to the first 3: mass (M), length (L), and time (T)

**It is possible to represent a dimension by an array of integers**
```c
typedef int dimensions[3]; // 7 in reality
const dimension mass = {1, 0, 0};
const dimension length = {0, 1, 0};
const dimension time = {0, 0, 1};
```
- A force (MLT²) would then be
  ```c
  const dimension force = {1, 1, -2};
  ```
- However, these are run time arrays, *not types*!

---

Example: Dimensional Analysis

**Representing Physical Dimensions (2)**

### Integral sequence wrapper
```c
    vector<int_<e1>, int_<e2>, …, int_<en>>
```
- A vector of types, each type being associated with an integer
  - each element similar to integral_constant from type traits
  - int_<e1> can be replaced by bool_<1>, long_<…>
- The elements e1, e2, …, en must be compile time constants, here of type int
- The overall result vector is a type
- There is one different type for different values of the template parameters
- Note the use of variadic templates, a C++11 extension

---

Example: Dimensional Analysis

**Representing Physical Dimensions (3)**

### Fundamental dimensions
```c
typedef vector_c<int, 1, 0, 0> Mass; // M
typedef vector_c<int, 0, 1, 0> Length; // L
typedef vector_c<int, 0, 0, 1> Time; // T
```
### Derived dimensions
```c
typedef vector_c<int, 0, 0, 0> Scalar;
typedef vector_c<int, 1, 1, -2> Force;
typedef vector_c<int, 0, 1, -1> Velocity;
typedef vector_c<int, 0, 1, -2> Acceleration;
```
- All these types are different
- `vector_c` is an example of (meta)sequence, a sequence of types
Example: Dimensional Analysis
Representing Physical Quantities

// D: the physical dimension type
// note that D is not directly used in the class

template <typename D>
struct quantity
{
  explicit quantity(double x = 0.0) : _value(x) {}

double value() const {return _value;}

private:
  double _value;
};

quantity<Force> f(10.0); // a force of 10N

Example: Dimensional Analysis
Add, Subtract, Compare...

template <typename D>
quantity<D> operator+(quantity<D> x, quantity<D> y)
{
  return quantity<D>(x.value() + y.value());
}

quantity<Length> len1(3.5), len2(3.5), l;
quantity<Force> f(10.0);
quantity<Length> len;
len = len1 + len2; // OK
len = len1 + f;  // does not compile

Example: Dimensional Analysis
Multiply and Divide (1)

template <typename D1, typename D2>
quantity<???> operator*(quantity<D1> x, quantity<D2> y)
{
  return quantity<???> (x.value() * y.value());
}

We need a type like D1+D2, where addition is element-wise...

Example: Dimensional Analysis
Multiply and Divide (2)

(Meta)-algorithm transform

transform<S1, S2, BinOp>
S1 and S2 are two type sequences with the same length
BinOp is a binary metafunction
The metafunction transform yields a new sequence produced by applying
BinOp to the pairs of elements, one from each sequence

Meta-function plus

plus<int_<i1>, int_<i2>>
I1 and I2 are types representing integral values [int, long, ...]
The metafunction plus yields a type representing the sum of the values associated
with I1 and I2
    plus<int_<i1>, int_<i2>> is plus<int_<i1+i2>>
Of course there exists also minus, mult, div, etc.
Example: Dimensional Analysis
Multiply and Divide (3)

Dimensions of the result of multiplication

```cpp
template <typename D1, typename D2>
struct mult_dimensions {
    transform<D1, D2, plus<_1, _2>>;
};
```

Multiply operator

```cpp
template <typename D1, typename D2>
quantity<typename mult_dimensions<D1, D2>::type>
operator*(quantity<D1> x, quantity<D2> y) {
    typedef typename mult_dimensions<D1, D2>::type dim_res;
    return quantity<dim_res>(x.value() * y.value());
}
```

Example: Dimensional Analysis
Multiply and Divide (4)

Use of multiply

```cpp
quantity<Mass> m(10.0);
quantity<Acceleration> g(9.81);
cout << m * g << endl; // OK
```

But...

```cpp
quantity<Force> f;
f = m * g; // does not compile
```

The type built by transform is not a specialization of vector_c
- although it contains the same integer values...
- We need an implicit conversion...

Example: Dimensional Analysis
Multiply and Divide (5)

```cpp
template <typename D>
struct quantity {
    explicit quantity(double x = 0.0) : _value(x) {};

double value() const {return _value;}

    template <typename D1>
    quantity(const quantity<D1>& q) :
        _value(q.value()) {
        static_assert(equal<D, D1>::type::value,
            "Bad conversion of dimensions");
    }

private:
    double _value;
};
```

mpl::equal checks two type sequences for equality, element-wise

Overview of MPL

- Data types
  - Integral wrappers
    - transform integers into types
  - Metafunctions
    - for integral wrappers
    - arithmetic
    - logical
    - bitwise
    - for type selection
    - for high order programming
    - miscellaneous

- Type sequences
  - containers for types
  - possibly extensible (insertion)
  - possibly with lazy evaluation
  - associated metafunctions

- Iterators
  - position in sequence

- Algorithms
  - apply an operation on a sequence
Overview of MPL
Data Types

- Integral constant
  - Nullary metafunction that returns itself
  - Associated with an integral compile-time constant
  - Example: `int_<N>` is a class such that `int_<N>::value == N`
  - `int_<N>::type` is `int_<N>`
  - `int_<N>::value_type` is `int_<N>`
  - Also `bool_<TF>`, `long_<N>`, `size_t<N>`, `integral_c<T,N>`

- Numeric metafunctions
  - Example: `plus<T1, T2, ...>` is a class such that
    - `T1`, `T2`... must be integral constant types
    - `plus<T1, T2, ...>::type` is the integral constant type corresponding to the sum of the values of `T1`, `T2`,...
  - Arithmetic: `plus`, `minus`, etc.
  - Comparison: `less`, `equal_to`, etc.
  - Logical: `and`, `or`, `not`
  - Bitwise: `bitand`, `shift_left`....

Overview of MPL
Sequences of Types (1)

- Sequences
  - `vector`, `list`, `deque`
  - `vector_c` for integral constant wrappers
  - `set`, `map` (associative sequences)
  - `range_c`

- Sequence metafunctions
  - Indexing: `at`, `at_c`
  - Iterator interface: `begin`, `end`
  - Accessors: `back`, `front`, `size`, `value_type`
  - Insertion: `insert`, `insert_range`, `push_back`, `pop_back`
  - Pop Front
  - Erasing: `clear`, `erase`
  - Associative sequences: `erase_key`, `has_key`, `key_type`

Overview of MPL
Sequences of Types (2)

- `vector` and `vector_c`
  - `typedef vector<int_<0>, int_<1>, int_<2>> v012; // not nice`
  - `typedef vector_c<int, 0, 1, 2> vc012; // nicer`
  - `STATIC_ASSERT(equal<v012, vc012, equal_to<_,_>>::value);`
  - `STATIC_ASSERT(not is_same<v012, vc012>::value);`

- The 2 types are different, although they correspond to the same integer sequence

- Extending a sequence
  - `typedef push_back<v012, int_<3>> v0123;`
  - `typedef push_back<vc012, int_<3>> vc0123;`
  - `STATIC_ASSERT(equal<v0123, vc0123, equal_to<_,_>>::value);`
  - `STATIC_ASSERT(not is_same<v0123, vc0123>::value);`

Overview of MPL
Sequences of Types (3)

- Indexing
  - `typedef at_c<v0123, 2> int2;`
  - `STATIC_ASSERT(at_c<v0123, 2>::type::value == 2);`

- `range_c`
  - `range_c<int, N1, N2>` creates a sequence of consecutive integers from `N1` (included) to `N2` (excluded)
  - `typedef range_c<int, 0, 4> r0123;`
  - `STATIC_ASSERT(size<r0123>::type::value == 4);`
  - `STATIC_ASSERT(not equal<v0123, v0123, equal_to<_,_>>::value);`

- A `range_c` sequence is not extensible
Overview of MPL
Sequences of Types (4)

**Iterators**

typedef list<char, short, int, long, long long> types;
typedef begin<types>::type types_0;
STATIC_ASSERT(is_same<deref<types_0>::type, char>::value);
typedef advance<types_0, int_<2>>::type types_2;
STATIC_ASSERT(is_same<deref<types_2>::type, int>::value);

**Metafunctions**
- move: advance, next, prior
- compute distance: distance
- iterator dereferencing: deref
- sort of iterator: iterator_category

Overview of MPL
Algorithms: for_each (1)

**Traversing the compile time/run time boundary: for_each**

```cpp
for_each<seq>(fobj);
```
- `seq` is an MPL sequence, and `fobj` a function object
- `for_each` type in `seq` *for_each* invokes `fobj`, passing to it an instance of the type
  - the instance is *value initialized*
  - since value initialization does not work for references, `seq` cannot contain reference types (nor classes which are not default constructible...)

Overview of MPL
Algorithms: for_each (2)

**Example: printing the elements of a sequence of types**

```cpp
template <typename T>
struct type_printer {
  void operator()(T) {
    cout << typeid(T).name() << ' ';
  }
};
template <typename T> class A {};
typedef vector<int, double, string*, A<int>> types;
for_each<types>(type_printer());
```
- Because of value initialization, type `A` must be complete (and default constructible) and `types` must not contain any reference
- The result is not nice: with `g++` it is `id Ps 1AIiE`
Fixing the `typeid()` problem (case of g++)

```cpp
template <typename T>
string type_to_string()
{
    int status;
    const char *name = typeid(T).name();
    char *demangled = abi::__cxa_demangle(name, 0, 0, &status);

    string res(status == 0 ? demangled : name);
    if (demangled)
        free(demangled); // !! __cxa_demangle uses malloc()!!
    return res;
}
```

However, `typeid()` looses reference indications...

Second form of initialization problem

```cpp
for_each<seq, transf>(fobj);
```

- `seq` is an MPL sequence, `transf` an MPL transformation metafunction, and `fobj` a function object
- for each type `T` in `seq`, `fobj` is invoked with a parameter the type of which is the result of the transformation of `T` by `transf` (`transf<T>::type`)

Using a visitor pattern

```cpp
struct visit_type { // a completely generic visitor
    void operator()(Visitor) const {
        Visitor::visit();
    }
};

template <typename T>
struct print_visitor {
    static void visit() {
        cout << type_to_string<T>() << ' ';
    }
};
```

Note that `print_visitor<T>` does not instantiate any `T` object nor that `visit_type` require any one; only the type is passed along

The value initialization problem disappears

```cpp
template <typename Seq>
struct print_sequence {
    print_sequence() {
        for_each<Seq, print_visitor<>>(visit_type());
        cout << endl;
    }
};
```

```cpp
typedef vector<int, double&, string, A<int>> types;
print_sequence<types>();
```

The result is

```cpp
int double std::string* A<int>
```

The reference indication is lost; not yet perfect forwarding...
Using if_ and logical operators: an example

```cpp
typedef transform<
    types,
    if_<
        or_<
            boost::is_scalar<_1>,
            boost::is_reference<_1>
        >,
        identity<_1>, // then
        add_reference<_1> // else
    >::type ref_types;
```

Lazy evaluation

```cpp
Lazy evaluation
- Computing the elements of a sequence on demand only
- Here, searching an element first construct the whole transformed sequence
- With the following, the transformed sequence is constructed on demand (the process stops when found)
```

Just an example: applying the same metafunction twice

```cpp
template <typename F, typename T>
struct twice : apply<F, typename apply<F, T>::type> {}

struct pointerize { // a metafunction class
    template <typename T>
    struct apply {
        typedef T *type;
    };
};

twice<pointerize, double> ppx; // double** ppx

typedef vector<int, double, string> types;
typedef transform<types, twice<pointerize, _> >::type pptypes;
at_c<pptypes, 2>::type pps; // string **pps
```

Evaluation of Template Programming (1)

Pros
- Powerful type manipulation
- Computing and dispatching at compile time speeds up execution
- Domain specific languages on top of C++
- Type aware macro-processing

Cons
- Nasty syntax, terrifying error messages
- Difficult to debug, no development environment
- Programming limitations: data can be types or integers only
- Functional style is not natural to everybody
Type traits is nice and simple enough

- **bind and function** are very convenient
  - Used by other libraries like Boost or C++11 threads

- Removal of *Concepts* from C++11 will tend to increase usage of
  - `enable_if` and other "metaprogramming tricks"
  - But *Concept (Lite)* will resurrect in C++14...