Trends in C++ and Java

rOOts 2002

Angelika Langer
Training / Consulting
http://www.AngelikaLanger.com

rOOts 2002

C++ and Java trends

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com

C++ and Java trends

- C++ standard library extensions
- Java generics

status of C++

- standardized in 1998
- 5-year freezing period
- committee now considers extensions

- language mostly stable
- library will be extended

example of language change

- what does the following snippet of code mean?

```
template <class T> class X {
 ...
 friend class T;
};
```

- the class used as template argument is friend of the generated class, i.e. `MyClass` is friend of `X<MyClass>` ?

friends and templates

- can't use a template parameter in an *elaborated type specifier*
  - means you can't use a template parameter as the class name in a friend class declaration
  - you can say things like "`friend class X<T>`" though

- the friend declaration is either ill-formed (compile-time error) or declares a new type `T` in the scope surrounding `X`
  - committee not sure what it means (open issue 245)

- it definitely does not mean "`T` is a friend of `X`"
  - committee considers language change

real-life example: stream manipulator

- manipulators are objects that can be inserted or extracted from a stream
- manipulate the stream; example: `endl`

- need an overloaded shift operator for the manipulator type:

```
template <class Ostream, class Manip>
Ostream& operator<< (Ostream& os, const manipBase<Manip>& m)
{ return m.manipulate(os); }
```

manipulator base class

- responsible for iostream-specific duties
  - error/exception handling, exception mask, stream state, ...
- relies on derived class's `fct()` function for actual manipulation

```
template <class Manip> class manipBase {
public:
 template <class Stream>
 Stream& manipulate(Stream& str) const
 {
   //...
   // call Manip::fct()
   static_cast<const Manip&>(*this).fct(str);
   //...
   return str;
 }
};
```

static downcast

intended use

"curiously recurring template"

```
class multi : public manipBase<multi> {
public:
   multi(char c, size_t n) : how_many_(n), what_(c) {}
private:
   const size_t how_many_;
   const char what_;
public:
   template <class Ostream>
   Ostream& fct(Ostream& os) const
   {  for (unsigned int i=0; i<how_many_; ++i)
         os.put(what_);
      os.flush();
      return os;
   }
};
```

```
cout << multi('*',100) << endl;
```

minor flaw

- static downcast not entirely safe
- undefined behavior in following case:

different classes

```
class mendl : public manipBase<multi> {
public:
   mendl(unsigned int n) : how_many_(n) {}
private:
   const unsigned int how_many_;
public:
   template <class Ostream>
   Ostream& fct(Ostream& os) const { ... }
};
```


an elegant solution

- make ctors/dtors private and declare `Manip` a friend

```
template <class Manip> class manipBase {
public:
 template <class Stream>
 Stream& manipulate(Stream& str) const
 { ... as before ... }
private:
 manipBase() {}
 manipBase(const manipBase&) {}
 ~manipBase() {}
 manipBase &operator=(const manipBase&) {}
 friend class Manip;
};
```

solution explained

- ctor of `mendl` implicitly calls private base class ctor
- accessible only to friends
- but `mendl` is not a friend `manipBase<multi>`, whereas `multi` would be

```
class mendl : public manipBase<multi> {
public:
   mendl(unsigned int n) : how_many_(n) {}
   ...
};
```


catch

- friend declaration doesn't make the template parameter a friend
- instead makes a class named `Manip` a friend (or is ill-formed)

```
template <class Manip> class manipBase {
public:
 template <class Stream>
 Stream& manipulate(Stream& str) const
 { ... as before ... }
private:
 manipBase() {}
 ... as before ...
 friend class Manip;
};
```

library extensions

- non-profit organization BOOST
  - has been collecting ideas and implementations of conceivable extensions to the standard library
  - free download from www.boost.org

- many proposals for library extensions are now taken from the BOOST library

proposed library extension

- C99 extensions
- rational numbers
- regular expressions
- smart pointers
- hash tables
- random numbers
- type traits
- threads

C99 extensions

- typedefs based on the 1999 C Standard header `<stdint.h>`
- supplies typedefs for standard integer types such as `int32_t` or `uint_least16_t`
- use in preference to `<stdint.h>` for enhanced portability

rational numbers

- standard library supports complex numbers
- natural extension to support rational numbers
- in a similar manner to the standard complex class

```
rational<int> half(1,2);
rational<int> one(1);
rational<int> minus_half(-1,2);

assert(rational_cast<double>(half) == 0.5);
assert(half + half == one);
assert(abs(minus_half) == half);
```

regular expressions

- a form of pattern-matching used in text processing
  - known from Unix utilities grep, sed and awk, and programming language perl
- C++ regex provides POSIX C API's, but goes beyond
  - can cope with wide character strings
  - offers search and replace operations
- `reg_expression` represents a "machine readable" regular expression
  - closely modeled on `std::basic_string`
  - a string plus actual state-machine required by regular expression algorithms

smart pointers

- 4 smart pointer classes in BOOST
  - designed to complement the standard library `auto_ptr` class

`scoped_ptr` / `scoped_array`

- simple sole ownership of single objects / arrays
  - guarantees deletion of object on destruction or via `reset()`
  - no transfer of ownership; mimics a built-in pointer

`shared_ptr` / `shared_array`

- object / array ownership shared among multiple pointers
  - reference counted pointer

Alexandrescu's smart pointer

- the configurable smart pointer class template from the Loki library
  - uses not just template type parameters, but also template template parameters

```
template
<
 typename T,
 template <class> class OwnershipPolicy = RefCounted,
 class ConversionPolicy = DissallowConversion,
 template <class> class CheckingPolicy = AssertCheck,
 template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPointer;
```


Alexandrescu's Smart Pointer

- ownership policy
  - deep copy, destructive copy, no copy
  - reference counted (thread-safe or not)
- conversion policy
  - allow or disallow implicit conversion to underlying pointer type
- checking policy
  - reject null
  - no check
- storage policy
  - default storage (does `delete`)
  - array storage (does array `delete[]`)
  - heap storage (calls `free()`)

random numbers

- random numbers needed for
  - numerics (simulation, Monte-Carlo integration)
  - games (non-deterministic enemy behavior)
  - security (key generation)
  - testing (random coverage in white-box tests)

- proposal separates number generators from distributions
  - number generator
    - generates a sequence of numbers uniformly distributed on a given range
  - distribution
    - maps one distribution (e.g. uniform distribution provided by some generator) to another

random number generators

- non-deterministic random number generator
  - based on some stochastic process; truly-random numbers
- pseudo-random number generator
  - based on some algorithm and internal state; deterministic

- proposal has implementations of various algorithms
  - example: `minstd_rand`
    - linear congruential pseudo-random number generator

```
x(n+1) := (a * x(n) + c) mod m
```

    - `a`, `c`, and `m` have sensible default values; `x(0)` is the seed

distributions - examples

`uniform_smallint`

- discrete uniform distribution on a small set of integers (much smaller than range of underlying generator)
- example: drawing from an urn

`bernoulli_distribution`

- Bernoulli experiment: discrete boolean valued distribution with configurable probability
- example: tossing a coin (p=0.5)

`uniform_on_sphere`

- uniform distribution on a unit sphere of arbitrary dimension
- example: choosing a random point on Earth (assumed to be a sphere) where to spend the next vacation

random number generator - example

```
minstd_rand generator;

uniform_01<minstd_rand> uni(generator);
for(int i = 0; i < 10; i++)
   cout << uni() << '\n';

generator.seed(std::time(0));

uniform_smallint<minstd_rand> die(generator,1,6);
for(int i = 0; i < 10; i++)
   cout << *die++ << '\n';
```

- pseudo random number generator
- uniform distribution in [0,1)
- change seed
- discrete uniform distribution
- use STL iterator interface

hash tables

- first proposal in 1995 rejected for reasons of timing
- today: 3 independently written implementations
  - SGI, Dinkumware, and Metrowerks
- similar, but not identical and different from current proposal
- differences include:
  - iterator can be forward (reduced overhead, but slower) or bidirectional (same as for tree-based container)
    - proposal allows for both
  - lookup in bucket via equality or comparison
    - proposal suggest equality

hash table interface

- hash function and equality are separate functions
  - Dinkumware packages them into one structure

```
template <class Value,
          class Hash = hash<Value>,
          class Pred = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class hash_set;
```

hash policy control

```
double load_factor() const;
```

- returns average number of elements per bucket

```
double max_load_factor() const;
```

- returns maximum load factor
  - container automatically increases number of buckets as necessary to keep the load factor below this number

```
void set_max_load_factor(double z);
```

- changes the container's maximum load load factor

```
void rehash(size_type n);
```

- changes the number of buckets so that it is at least n

bucket interface

- expose the bucket structure
  - lets users investigate how well hash function performs
    - test how evenly elements are distributed within buckets
    - see if element in a bucket have any common properties
- enable optimized algorithms
  - iterators might have an underlying segmented structure
    - if buckets are singly linked lists
  - algorithms can exploit that structure with an explicit nested loop
- interface includes:

```
size_type bucket_count() const;
size_type bucket_size(size_type n);
local_iterator begin(size_type n);
local_iterator end(size_type n);
```

type traits

- sometimes templates are not quite as "generic" as one would wish
- problem: not all types are created equally
  - some categories of types may need special handling
- example: a version of `std::pair` that can hold reference types

std::pair

- can't be instantiated for a reference type
  - reference to reference not allowed in C++

```
template <typename T1, typename T2>  struct pair  {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;

  pair(const T1& nfirst, const T2& nsecond)
  :first(nfirst), second(nsecond) { }

  ...
};
```

type traits "magic"

type of `T1`	type of constructor argument
`T`	`const T &`
`T &`	`T &`
`const T &`	`const T &`

`add_reference<const T1>::type`

- if `T` is a reference type then leaves `T` unchanged
- otherwise converts `T` to a reference type

```
template <typename T>
struct add_reference{ typedef T& type; };
template <typename T>
struct add_reference<T&>{ typedef T& type; };
```


change std::pair

```
template <typename T1, typename T2> struct pair {
  typedef T1 first_type;
  typedef T2 second_type;
  T1 first;
  T2 second;
  pair(add_reference<const T1>::type nfirst,
       add_reference<const T2>::type nsecond)
  :first(nfirst), second(nsecond) { }
};
```

- can also be achieved by partial specialization of the pair class

type traits for optimizing code performance

- classic example: algorithm `std::copy`
  - if types being copied are PODs, then `std::memcpy` can be used to copy the data, rather than a slower "object by object" copy

- key idea:
  - use helper function that is overloaded for PODs and non-PODs

helper function

- version for non-POD types
  - performs regulat object-by-object copy

```
namespace detail{
 template <bool b> struct copier
 {  template<typename I1, typename I2>
    static I2 do_copy(I1 first, I1 last, I2 out)
    {  while(first != last)
       {  *out = *first;
          ++out;
          ++first;
       }
       return out;
    }
 };
}
```

helper function overloaded

- version for PODs
  - uses memcpy

```
namespace detail{
 template <> struct copier<true>
 {
   template<typename I1, typename I2>
   static I2* do_copy(I1* first, I1* last, I2* out)
   {
      memcpy(out, first, (last-first)*sizeof(I2));
      return out+(last-first);
   }
 };
}
```


optimized std::copy

- same semantics as `std::copy`
- calls `memcpy` where appropriate

```
template<typename I1, typename I2>
inline I2 copy(I1 first, I1 last, I2 out)
{
   typedef typename remove_cv<
        typename std::iterator_traits<I1>::value_type>::type v1_t;
   typedef typename remove_cv<
        typename std::iterator_traits<I2>::value_type>::type v2_t;
   return detail::copier<
         is_same<v1_t, v2_t>::value
         && is_pointer<I1>::value
         && is_pointer<I2>::value
         && has_trivial_assign<v1_t>::value
       >::do_copy(first, last, out);
}
```

type traits “magic”

`remove_cv<T>::type`

- creates a type the same as T but with any top level cv-qualifiers removed

`is_same<T,U>::value`

- true if T and U are the same type

`is_pointer<T>::value`

- true if T is a regular pointer type

`has_trivial_assign<T>::value`

- true if T has a trivial assignment operator
  - if `T::operator=(const T&)` is equivalent to `memcpy`

remove_cv

- works via template specialization

```
template <typename T> struct remove_cv
{ typedef typename cv_traits_imp<T*>::unqualified_type type; };
template <typename T> struct remove_cv<T&>
{ typedef T& type; };
```

```
template <class T> struct cv_traits_imp
{};
template <class T> struct cv_traits_imp<T*>
{ typedef T unqualified_type; ... };
template <class T> struct cv_traits_imp<const T*>
{ typedef T unqualified_type; ... };
template <class T> struct cv_traits_imp<volatile T*>
{ typedef T unqualified_type; ... };
template <class T> struct cv_traits_imp<const volatile T*>
{ typedef T unqualified_type; ... };
```

has_trivial_assign

```
template <typename T> struct has_trivial_assign {
private:
   typedef typename remove_cv<T>::type cvt;
public:
   static const bool value = is_POD<T>::value
      || HAS_TRIVIAL_ASSIGN(cvt);
};
```

```
template <typename T> struct is_POD {
   ...
   static const bool value = is_scalar<cvt>::value
      || IS_POD(cvt);
};
...
```

- macros like `IS_POD` and `HAS_TRIVIAL_ASSIGN` must be specialized for each user-defined POD type

threads

- goals: portable, safe, efficient
- supports:
  - synchronization primitives
    - mutex, recursive mutex, scoped lock
  - thread management and thread specific storage
    - thread, thread specific pointer
- intended extensions (not yet available and not yet proposed):
  - more advanced synchronization concepts
    - read/write mutexes, barriers

synchronization - example

```
class counter {
public:
    counter() : count(0) { }
    int increment() {
        mutex::scoped_lock scoped_lock(mutex);
        return ++count;
    }
private:
    mutex mutex;
    int count;
};
```

a thread-safe class

critical region

```
mutex io_mutex;

void change_count(void*) {
    int i = c.increment();
    mutex::scoped_lock scoped_lock(io_mutex);
    std::cout << "count == " << i << std::endl;
}
```

a function using
the synchronized operation

thread management - example

- change count in 4 parallel threads

```
counter c;

int main(int, char*[]) {
    const int num_threads = 4;
    thread_group thrds;
    for (int i=0; i < num_threads; ++i)
        thrds.create_thread(&change_count, 0);
    thrds.join_all();
    return 0;
}
```

output:

```
count == 1
count == 2
count == 3
count == 4
```

thread specific memory - example

```
thread_specific_ptr<int> value;

void increment() {
   int* p = value.get();
   ++*p;
}
void thread_proc() {
   value.reset(new int(0));
   for (int i=0; i<10; ++i)
   {   increment();
       int* p = value.get();
       assert(*p == i+1);
   }
}
```

initialize thread-specific storage

```
int main(int argc, char* argv[]) {
   thread_group threads;
   for (int i=0; i<5; ++i)
      threads.create_thread(&thread_proc);
   threads.join_all();
}
```


conclusion

- evolution rather than revolution
  - sound proposals for useful library extensions standardize common practice
- there is more on the committee's wish list
  - text processing, numerics, graphics, system programming, networking, language bindings, and multi-language programming
- open-source projects complement the standard
  - International Components for Unicode (ICU) (initiated by IBM)
  - cross-platform C, C++ and Java APIs for supporting I18N
  - interesting alternative to the standard C++ locales and facets

conclusion

- templates play an important role
  - used intensely in modern C++ libraries
    - except ICU which is a Java port
  - more daring use of templates demonstrated
    - for Generative Programming by Eisenecker/Czarnecky and
    - in Loki library by Alexandrescu

C++ and Java trends

- C++ standard library extensions
- Java generics

Java Generics

- add generic types and methods to Java
- benefits:
  - expressiveness and safety
  - make type parameters explicit and making type casts implicit
  - crucial for using libraries such as collections in a flexible, yet safe way

- parameterized type
  - class or interface that has type parameters
- type variable
  - placeholder for a type, i.e. the type parameter

parameterized types

- instantiations of parameterized types look like C++ templates

- examples:

```
Vector<String>
Seq<Seq<A>>
Seq<String>.Zipper<Integer>
Collection<Integer>
Pair<String,String>
```

- primitive types cannot be parameters
  - `Vector<int>` is illegal

benefit of parameterized types

- today: no information available about the type of the elements contained in a collection

cast might fail

```
void append(Vector v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = (StringBuffer) (v.get(idx));
    buf.append(suffix);
  }
}
```

- future: parameterized type provides more information and performs cast implicitly

cannot fail

```
void append(Vector<StringBuffer> v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = v.get(idx);
    buf.append(suffix);
  }
}
```


parameterized classes or interfaces

- definition of a parameterized class
  - type variables T1 and T2 act a parameters

```
class Pair <T1,T2> {
  private T1 t1;
  private T2 t2;
  ...
}
```

- type variable can have optional *bounds*
  - a bound consist of a class and/or several interfaces
  - if no bound is provided `Object` is assumed

```
class AssociativeArray <Key implements Comparable, Value> {
  ...
}
```

shared type identification

- all instantiations of a parameterized type have the same runtime type
  - type parameters are not maintained at runtime and do not show up in the byte code

```
Vector<String>  x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
return x.getClass() == y.getClass();
```

true

raw types

- ray type: parameterized class without its parameters
  - variables of a raw type can be assigned from values of any of the type's parametric instances
  - reverse assignment permitted to enable interfacing with legacy code

```
Vector rawVector = new Vector();
Vector<string> stringVector = new Vector<String>();

rawVector = stringVector;

stringVector = rawVector;
```

fine (rawVector = stringVector;)

compiler warning: assignment deprecated (stringVector = rawVector;)

raw types

- access to fields of a raw type

```
class Cell<Type> {
 private Type value;
 public Cell (Type v)    { value=v; }
 public Type    get()    { return value; }
 public void set(Type v) { value=v; }
}
```

```
Cell rawCell = new Cell<String>("abc");
... rawCell.value ...;
... rawCell.get();
rawCell.put("def"); // deprecated
```

fine, value has type Object

compiler warning:
unchecked access to field

generic methods

- method declarations can have a type parameter section like classes have

```
static <Elem> void swap(Elem[] a, int i, int j) {
  Elem temp = a[i]; a[i] = a[j]; a[j] = temp;
}
```

```
<Elem implements Comparable<Elem>> void sort(Elem[] a) {
  for (int i = 0; i < xs.length; i++)
  for (int j = 0; j < i; j++)
    if (a[j].compareTo(a[i]) < 0) <Elem>swap(a, i, j);
}
```

- no special syntax for invocation
  - type parameters are inferred from arguments

```
swap(ints, 1, 3);
sort(strings);
```

do we really benefit ?

```
void append(Vector<StringBuffer> v, char[] suffix) {
  for(int idx=0; idx<v.size(); ++idx) {
    StringBuffer buf = v.get(idx);
    buf.append(suffix);
  }
}
```

- raw type can be assigned to instantiated type
  - creates compiler warning, but is permitted

```
Vector files = new Vector();
// fill with Strings, not StringBuffers !!!
Vector<StringBuffer> tmp = files;
append(tmp, ".txt");
```

assignment of raw type permitted

implicit cast can fail

conclusion

- minor impact on the language
  - designed for backward compatibility
- major impact on the platform libraries
  - resign of collections framework likely
- availability: not yet announced
  - definitely not in J2SE 1.4
  - draft version of specification (dated April 2001)

trends in C++ and Java

- evolution instead of revolution
  - both language are fairly mature
- language adjustments circle around templates / generics
- library extensions cover smaller utilities and features

- big difference between C++ and Java
  - major business frameworks are developed for Java
  - distributed component architecture (J2EE with EJB)
  - service oriented architectures (JINI, WebServices)
  - not so sure about Java's role in the real-time domain

big difference between C++ and Java

- major business frameworks are developed for Java
  - distributed component architecture (J2EE with EJB)
  - service oriented architectures (JINI, WebServices)
  - RogueWave's XML for C++ Web Services

- not so sure about Java's role in the real-time domain
  - are there any real-time virtual machines ?