Advanced C++

Ron Akkersdijk

Come along. Saxion.

Autumn 2014

SAXION

Overview

- ...
- Function Templates
- Template Examples
- Generic Algorithms
- Function Objects
- Iterator types
- Smart pointers
- ...

Algorithms

- Algorithms like find(), count(), count_if(), sort(), etc., describe what to do with the contents of a container, independent of the actual type of the container.
  - The STL already provides several of them
- Mechanisms used are:
  - *iterators* to visit elements of a container
  - ways to *compare* or *test* elements e.g.
    - *operators* (via *overloading*)
    - (pointers to) *functions*
    - *function objects* (aka *functors*)

Find algorithm

```
// How To: find first occurrence of some value
template <typename Iter, typename Type>
   Iter find( Iter from, Iter upto,      // the range
                  Type value )           // what
{
   for ( ; from != upto ; ++from) {      // for each
        if (value == *from)              // same?
             return from;                // got it!
   }
   return upto; // i.e. == "not found"
}
```

```
int  values[] = { 1, 2, 1, 2, 3, 2, 9 };
int* two = find( &values[0], &values[7], 2 );
// Uses:     find<int*,int>(...)
```

Using find()

```
int  searchValue;
cin >> searchValue;

int   values[] = { 1, 2, 1, 2, 3, 2, 9 };
int*  presult1;
presult1 = find(values, values+7, searchValue);
     // i.e find<int*,int>(int*, int*, int)

vector<int>  vec( values, values+7 ); // copy-constructor
vector<int>::iterator  presult2;
presult2 = find(vec.begin(), vec.end(), searchValue);
     // i.e. find<vector<int>::iterator,int>(...)

list<int>  ilist( values, values+7 );
list<int>::iterator  presult3;
presult3 = find(ilist.begin(), ilist.end(), searchValue);
     // i.e. find<list<int>::iterator,int>(...)
```

Count algorithm

```
// How To: count the occurrences of a value
template <typename Iter, typename Type>
   int  count( Iter from, Iter upto,    // the range
                Type value )            // what
{
   int  n = 0;                          // initially none
   for ( ; from != upto ; ++from) {     // for each
        if (value == *from)             // operator ==
             ++n;                       // increment count
   }
   return n;                            // return result
}
```

```
int  values[] = { 1, 2, 1, 2, 3, 2, 9 };
int  twos = count( &values[0], &values[7], 2 );
// Uses:     count<int*,int>(...)
```

Count_if algorithm

```
// How To: count the matching elements
template <typename Iter, typename Type>
   int  count_if( Iter from, Iter upto,    // the range
                  Type test )              // what
{
   int  n = 0;                          // initially none
   for ( ; from != upto ; ++from) { // for each
        if (test(*from))                // a function!
            ++n;                        // increment count
   }
   return n;                            // return result
}
inline bool gt2(int x) { return x > 2; }
int  values[] = { 1, 2, 1, 2, 3, 2, 9 };
int  gt2s = count_if( &values[0], &values[7], gt2 );
// Uses:    count_if<int*,bool(*)(int)>(...)
```

Using count_if

```
inline bool gt2(int x) { return x > 2; }
```

- This gt2 function (and others like it) has various limitations & drawbacks
  - Built-in datatype int
  - Built-in value 2
- An inline *expression* has no *address*, so the compiler will ignore the inline request!
  - Extra *function call overhead*

A Function Object (*Functor*)

```
template <typename T>   // The datatype of the ...
class GreaterThan {
   const T value;         // ... value we compare with
public:
   GreaterThan(T x) : value(x) {}
   bool operator() (T x) const { return x > value; }
};
```

temporary object

```
int  values[] = { 1, 2, 1, 2, 3, 2, 9 };
int  gt2s = count_if( &values[0], &values[7],
                                GreaterThan<int>(2) );
// Uses:    count_if<int*,GreaterThan<int>>(...)
```

- Now both the datatype (int) and the value (2) are determined by the call rather then the function
- and ....

A Function Object (*Functor*)

```
template <typename T>   // The datatype of the ...
class GreaterThan {
   const T value;         // ... value we compare with
public:
   GreaterThan(T x) : value(x) {}
   bool operator() (T x) const { return x > value; }
};
```

- and ...
- The temporary, anonymous, function object is passed *by value*, but smart compilers will do it *by reference*!
- Because the object is effective const the compiler may even pre-generate a reusable object!
- The test call inside count_if is replaced with the actual expression!   (*see next slide*)

The transformation of “test”

- The original code in count_if was

```
   if (test(*from)) ...
```

- When 'test' is a *functor* it means

```
   if (test.operator()(*from)) ...
```

- The () operator for GreaterThan was inline *by implication*, so the final code becomes

```
   if ((*from) > test.value) ...
```

- And so we have also eliminated any function call overhead!

Using sort()

```
#include <algorithm> // provides std::sort()
vector<string>  words;
   sort( words.begin(), words.end() );
```

- The 'words' vector will be sorted using the < operator for the string class (i.e. lexicographically).
- To get other sorting orders pass a suitable *comparator* function(object)

```
   sort( words.begin(), words.end(), howto );
```

A sneaky trick

- How To: The *partial ordering* of a set 8-)

```
template<typename T> class BiggerFirst {
   const T value;
public:
   BiggerFirst(T x) : value(x) {}
   bool operator () ( T val, T dummy ) const {
      return val > value;
   }
};

vector<int>  numbers; ...
sort( numbers.begin(), numbers.end()
                           , BiggerFirst<int>(20) );
// sorts into: ...bigger... (20) ...smaller...
```

Predefined Algorithms

```
#include <algorithm>
```

- There are a lot of predefined algorithms e.g.:
  - find(*from*,*upto*,*what*) find_if(*from*,*upto*,*how*)
  - count(*from*,*upto*,*what*) count_if(*from*,*upto*,*how*)
  - sort(*from*,*upto*) sort(*from*,*upto*,*how*)
  - fill(*from*,*upto*,*what*)
  - for_each(*from*,*upto*,*what*)
  - random_shuffle(*from*,*upto*)
  - copy(*from*,*upto*,*dest*) (see later)
  - etc

Predefined Functors

- A predefined collection of *functors*:

```
#include <functional>
```

- For instance

```
sort( ..., ..., greater<string>() );
```

- Various categories:
  - Arithmetic: plus<T>, minus<T>, ...
  - Relational: equal_to<T>, greater<T>, ...
  - Logical: logical_and<T>, logical_not<T>, ...
- And also ...

Predefined Adapters

- To extend or specialise unary or binary function(object)s:
  - Binders: bind1st en bind2nd
    - Convert a binary op into a unary op
  - Negators: not1 en not2
- For instance:

```
... = count_if( vector.begin(), vector.end()
                , not1(
                    bind2nd(
                      less_equal<int>(), 10 ) ) );
```

Questions?

Iterators

- Iterators exist in various flavours (similar to java *interfaces*):
  1. InputIterator
  2. OutputIterator
  3. ForwardIterator & ReverseIterator
  4. BidirectionalIterator
  5. RandomAcessIterator
- plus variants of them …

- An iterator offered by a container can belong to <u>multiple</u> flavours!

1. InputIterator

- An InputIterator is used to read data from a container
  - Used by e.g. find(), accumulate(), equal()

- Operations:
  - Compare iterators: == and !=
  - Has the *prefix* and *postfix* ++ operators
  - Reads data using the *dereference* operators * and ->

2. OutputIterator

- An OutputIterator is used to write data to a container
  - Used by e.g. copy()

- Operations:
  - See 1 InputIterator, plus
  - Write data using *dereference* operators * and ->

- Note: Actually it updates existing data!

3. ForwardIterator

- A ForwardIterator will visit the elements in the container in one order from begin() to end()
  - Used by e.g. adjacent_find(), swap_range() and replace()
- Operations:
  - Has the ++ operators ("*forward*")
  - Does NOT have the –– operator ("*backward*")
- Variant: A ReverseIterator visits the same elements but in the reverse order from rbegin() to rend()

4. BidirectionalIterator

- A BidirectionalIterator can visit the elements in the container in both orders
  - Used by e.g. place_merge(), next_permutation() en reverse()

- Operations:
  - Has the ++ operators ("*forward*") and
  - Has the -- operators ("*backward*")

5. RandomAccessIterator

- A RandomAccessIterator can visit the elements in the container in *random* order <u>in fixed time</u>
  - Used by e.g. binary_search(), sort_heap() en nth_element()
  - Provides:
    `i+n, i-n, i-i, i[n], i<i, i<=i, i>i, i>=i`
  - Example: int *ip;

Iterators

- Various Iterator flavours:
  1. InputIterator (read-only)
  2. OutputIterator (write-only)
  3. ForwardIterator & ReverseIterator (one direction)
  4. BidirectionalIterator (two directions)
  5. RandomAcessIterator (random order)

- And now some variants of them …

Const_Iterator

- A Const_Iterator will <u>not</u> modify the container in any way
- Needed if the container is declared const

```
template <typename Type>
int count_in(const vector<Type>& vec, Type value) {
     int count = 0;
     vector<Type>::const_iterator iter;
     for(iter = vec.begin() ; iter != vec.end() ; ) {
        if (*iter++ == value)
           ++count;
     }
     return count;
}
```

Intermezzo: Copy

```
// How To: Copy elements from ... to ...
template<typename Iter1, typename Iter2>
  void  copy( Iter1 from, Iter1 upto, Iter2 dest ) {
    for ( ; from != upto ; ++from, ++dest )
      *dest = *from; // Copy element to destination
}
```

```
int  values[] = { 1, 2, 1, 2, 3, 2, 9 };
vector<int>  v;             // an empty vector !
copy( &values[0], &values[7], v.begin() );
```

- It is actually a copy-replace algorithm!
- You can not “replace” items in an empty vector
- For this situation you will need a very special kind of iterator which actually inserts new items!!

Insert_Iterator

- An Insert_Iterator is used to insert new items. It always maintains it's relative place in a container.
- You obtain them via special “creator” *function templates*, that pass the actual container

```
copy(vector1.begin(), vector1.end(),
   back_inserter(vector2));    // append new items
```

```
copy(vector1.begin(), vector1.end(),
   front_inserter(vector2));  // prepend new items
   // (which is not supported by e.g. vector<T>)
```

```
copy(vector1.begin(), vector1.end(),
   inserter( vector2, vector2.begin() ));
```

Insert_Iterator

- A back_inserter uses *container*.push_back()
- A front_inserter uses *container*.push_front()
  - *Since vector<T> has no push_front you can not apply a front_inserter to a vector, but you can cheat by using an inserter at vector.begin()!*
- An inserter uses *container*.insert()
- They all work by very sneaky overloading of operators, especially the '=' operator
- Remember that:

```
*dest = *from;
```

- actually stands for:

```
dest.operator*().operator=( *from );
```

Istream_iterator

- An Istream_Iterator reads elements of some type from an input stream using the >> operator for that type

```
ifstream                         infile("advanced_Cpp");
istream_iterator<string>   istream_iter(infile);
// iterator now points to first string in infile
istream_iterator<string>   end_of_stream;
// this iterator points to "string after last string"
vector<string>                  words;

copy( istream_iter, end_of_stream,
              back_inserter(words) ); // append words
```

Ostream_iterator

- An Ostream_Iterator writes elements of some type to an output stream using the << operator for that type
- Two flavors:

```
ostream_iterator<Type>  identifier(ostream&);
ostream_iterator<Type>  identifier(ostream&,
                                  char* delimiter);
     // adds the delimiter after each element
```

Using both types

- Print the words from a file:

```
string fileName;
…
ifstream  infile( fileName.c_str() );
…
istream_iterator<string>  inpi(infile), eois;

ostream_iterator<string>  outs( cout, " " );
…
copy(inpi, eois, outs); // copies words
   // from the given input file to cout,
   // separating them with a single space
```

Using your own types

```
class MyType {
public:
   istream& operator>>(istream&,MyType&);
   ostream& operator<<(ostream&,const MyType&);
};

istream_iterator<MyType>  inpi(istream), eois;

ostream_iterator<MyType>  outs(ostream, "\n" );

copy(inpi, eois, outs); // copies MyType data
   // from some given istream
   // to some given ostream,
   // separating them with a new-line character
```

Final words

- The concept of *generic algorithms* does not match the OO principle that you call the method of some object to achieve some purpose
- In many cases the method provided by the container is preferable because it will usually yield better performance

```
list<string> words;
words.sort(); // list<T>::sort() is better than
sort( words.begin(), words.end() );
```

Questions?

About pointers (1)

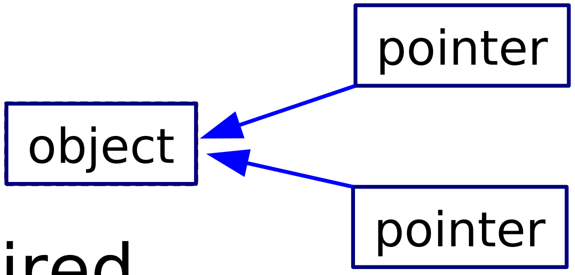
- The good things about pointers:

  - May produce compact code
  - Improve performance
  - Prevent redundancy
  - Provides dynamic binding (*polymorphism*)

- But they also have some drawbacks !

About pointers (2)

- Aspects causing undefined behaviour:
  - 1. Uninitialized pointers point into limbo
  - 2. Deleting an object may lead to *dangling pointers*



- Behaviour which is sometimes desired, but sometimes not:
  - 3. Deleting a pointer does not delete the associated object (possible *memory leak*)
  - 4. Copying pointers does not copy the associated object but makes it “shared”
- Possible solution: “*smart pointers*”

Class Handle<T>

- Purpose: Protect objects from users
  - Should be able to handle any type, so it will be implemented as a template class
  - Once an object “belongs” to a Handle only that handle should handle that object, not the program!
  - When copying a Handle the associated object will also be copied.
  - When deleting a Handle the associated object will also be deleted.
  - You can test whether a Handle refers to some object
  - Since C++11: unique_ptr<T>

Handle (1)

```
template <typename T>
class Handle {
private:
   T *objptr;     // the object "owned"
public:
   Handle(T* tp=0) : objptr(tp) {}

   operator bool() const { return objptr; } // if (x)

   T& operator  * () const {   // for: *ptr
      if (objptr) return *objptr;
      throw runtime_error("unbound handle");
   }
   T* operator -> () const  { // for: ptr->...
      if (objptr) return objptr;
      throw runtime_error("unbound handle");
   }
   ...
```

Using Handle (1)

- We can now do

```
void f() {

   Handle<Pet>  p = new Pet("fido", 10);
   if (p)                                   // operator bool
     cout << p->name << endl;               // operator ->

   Handle<int>  i = new int(7);
   cout << *i << endl;                      // operator *

}
// When leaving f, both p and i will be deleted,
// which in turn should delete the Pet and int
// objects created
```

Handle (2)

```
  ...

  ~Handle() { delete objptr; }  // delete 0 is valid!

  // call-by-value uses a copy-constructor!
  Handle(const Handle& s) : objptr(0)
    { if (s.objptr) objptr = new T(*s.objptr); }

  // assignment behaves like delete and copy
  Handle& operator = (const Handle& rhs) {
    if (&rhs != this) {        // not self?
      delete objptr;           // discard old object
      objptr = rhs.objptr      // anything to copy?
                 ? new T(*rhs.objptr) : 0;
    }
    return *this;
  }
}
```

Using Handle (2)

- We can now also do

```
void g(Handle<int> j) {
   ...
}
// at the end of g j will be deleted

void f() {
   Handle<int>  i = new int(7);
   g(i); // uses copy-constructor to clone an int
}
```

Handle

- Things to think about

```
Handle<int> i = new int(7);
Handle<int> j = i;
```

- Will `i == j` be true or false?
- Should `i == j` be true or false?
- Will `i != j` be true or false?
- Should `i != j` be true or false?

Handle

- Disadvantage:
  - Having an “exclusive” object may cause a lot of cloning when passing Handles as parameters (*call by value* uses the *copy-constructor*!)
- Solution:
  - Allow the object to be shared
  - Keeping track of the number of “handles” to it.

RefHandle

- Purpose: Protect objects from users
  - Like **`Handle<T>`** but:
    - Multiple RefHandles can refer to the same object
    - They maintain a shared “reference counter”
    - When copying a RefHandle the counter is incremented
    - When deleting a RefHandle the counter will be decremented
    - When the counter drops to zero the object will be deleted
    - Since C++11: shared_ptr<T>

RefHandle (1)

```
template <typename T>
class RefHandle {
private:
   T      *objptr;       // the shared object
   size_t *counter;      // counts "RefHandle"s !
public:
   RefHandle(T* tp=0)
     : objptr(tp), counter( new size_t(1) ) {}
   RefHandle(const RefHandle& s) // copy-constructor
     : objptr(s.objptr), counter(s.counter)
     { ++*counter; }             // one more RefHandle
   ~RefHandle() {
     if (--*counter == 0) {   // last RefHandle?
         delete objptr;       // cleanup everything
         delete counter;
     }
   }
   ...
```

RefHandle (2)

```
// assignment behaves like delete and copy
RefHandle& operator = (const RefHandle& rhs) {
  if (&rhs != this) {          // not self?
    // like destructor (give up current object)
    if (--*counter == 0) {
      delete objptr;
      delete counter;
    }
    // like copy-constructor (share other object)
    objptr = rhs.objptr;
    counter = rhs.counter;
    ++*counter;
  }
  return *this;
}
...
```

RefHandle (3)

- When multiple RefHandles share an object it can be desirable to “untangle” the sharing, i.e. give a RefHandle a private copy of the object.

```
// method to "untangle" RefHandles
void makeUnique() {
   if (*counter != 1) {  // not yet unique?
      --*counter;         // give up other object
      counter = new size_t(1);
      objptr = objptr ? new T(*objptr) : 0;
   }
}
```

Questions?

Calling C from C++

- In some situations we have to call C functions from C++ code
- Because of the *name-mangling* done by the C++ compiler, the C names don't match the C++ mangled versions
- By telling the compiler it is a C function this problem can easily be solved

```
extern "C" {
   void open(const char*,int,int);
   // now the compiler knows that the name 'open'
   // should not be mangled, making it C compatible.
}
```

Calling C from C++

- Usually this is already taken care of in the C header files!

```
// e.g. in cmath and math.h
#ifdef __cplusplus    // i.e. if used from C++
extern "C" {
#endif

// normal C declarations here
double sin(double);
double cos(double);
// etc.

#ifdef __cplusplus
}
#endif
```

C++ & pthreads

- Calling C++ functions from C requires a little but of work
- Calling C++ methods requires even more work
- For instance: Pthreads is a standard package for multi-threading on unix systems
- Since it has a C interface it can not directly handle C++ functions nor methods!
- Let's define a java style wrapper!

C++ & pthreads

- What should happen (the *program*):

```
class Runnable {    // de "interface"
public:
     virtual void     run() = 0;
};
```

- Who does the work (the *processor*):

```
class Thread : public Runnable {
public:
     Thread( Runnable* what = this );
     void  run() {}    // a dummy run method
     void  start();    // let's go!
};
```

SAXION

C++ & pthreads

- The, simplified, C interface provided by pthreads:

```
extern "C" {
   typedef   void*     thandle;
   typedef   void*     targs;
   typedef   void*     tresult;
   typedef   tresult ( *tfunc ) ( targs );

   thandle   createThread( tfunc, targs );
   void      exitThread( tresult );
   tresult   joinThread( thandle ); // wait for ...
}
```

- Note: the real pthreads interface is a lot more complex and the library provides a lot more!

C++ & pthreads

- However, **`createThread`** expects a C callback function, and can not handle C++ functions or methods
- So we need a work-around:

```
extern "C" {

  // A C++ function callable from C !
  void   cpphook( ... ) {
     ....
     // call regular C++ functions/methods here
     ....
  }

}
```

C++ & pthreads

- When we create a thread we let it run our cpphook function and pass the actual Runnable as a parameter

```
class Thread : public Runnable {
   thandle   tp;  // de pthread-handle
public:
   Thread::Thread( Runnable *rp )
   {
         tp = createThread( cpphook, rp );
   }
};
```

C++ & pthreads

- Of course our cpphook function knows the true nature of the parameter ...

```
extern "C" {
   tresult   cpphook( targs rp ) {  // targs is a void*
      // You can not dynamic_cast a void* !
      Runnable  *xp = static_cast<Runnable*>(rp);
      Runnable  *tp = dynamic_cast<Runnable*>(xp);
      if (tp)
         return  tp->run(); // the method to be run
      else
         throw runtime_error(
                    "cpphook: Not a Runnable!" );
   }
}
```

Questions?