Advanced Software Engineering with C++ Templates

Lecture 3: Templates II and Standard Library II

Thomas Gschwind <thg@zurich.ibm.com>
Agenda

- Templates II
  - Specialization
  - Traits
  - Default Values

- Standard Library II
  - Design
  - Algorithms
  - Function Operators
pvector<string>

- Implement the persistent vector data type.
- Experiment with the persistent vector and use it in combination with different data types. What do you observe? Why do you observe that behavior? How can it be changed?
- What happens if we pass the pvector around?
A Persistent Vector Class (cont’d)

```cpp
template<typename T>
class pvector {
  string filename;
  vector<T> v;

  void readvector() {
    ifstream ifs(filename);
    for(;;) {
      T x; ifs >> x; if(!ifs.good()) break;
      v.push_back(x);
    }
  }

  void writevector() {
    ofstream ofs(filename);
    typename vector<T>::iterator fst=v.begin(), lst=v.end();
    while(fst!=lst) ofs << *fst++ << endl;
  }
}
...
```

What if we use `string` as type parameter?

Reads a string up to the next whitespace

 Writes a string with and without whitespace
pvector for string?

- Use partial specialization
- Partial specialization allows us to change the implementation of a template for a specific class
- Very easy to implement but very repetitive

```cpp
template<>
class pvector<string> { 
    string filename;
    vector<string> v;
    void readvector() { ... }
    void writevector() { ... }
    ... // repeat all the other methods as is 
}
```
Sidebar: Inheritance

- The following shows a sample child class inheriting from a sample base class
  - Typically, we use “: public” to indicate “extends”
  - Typically, functions to be overridden in the child are declared virtual
  - We will come back to the gory details (do not use inheritance yet)

```cpp
struct base {
    virtual void print() const { cout << "base" << endl; }
};
struct child : public base {
    virtual void print() const { cout << "child" << endl; }
};
void test(const base &b) {
    b.print();
}
```

For inheritance to work as “expected” we typically use pointers or references.
pvector for string? (cont’d)

- Use inheritance
  - Need to change readvector and writevector in parent class to be virtual
  - Implied dynamic dispatch although unnecessary
  - We need a new class with a new name pvectorstring?

```cpp
class pvectorstring : public pvector<string> {
    virtual void readvector() { ... }
    virtual void writevector() { ... }
};
```

- Of course, we can fix the name problem

```cpp
template<typename T> class pvector : public pvector_base<T> {
    // repeat constructors
};
template<> class pvector<string> : public pvector_base<string> {
    // repeat constructors, specialize readvector, writevector
};
```

Not bad, but we can do better than that
pvector for string? (cont’d)

- Factor out the persistence logic into a separate interface
  - Yes, not so repetitive
  - Yes, the persistence logic can be reused

```cpp
template<
typename T>
struct pvector_serialize {
    virtual void readvector(string fn) = 0;
    virtual void writevector(string fn) = 0;
};
```
pvector with inheritance based serializer

- Pass serializer to pvector in constructor
  - But how will the serializer get hold of the vector to store the elements?

```cpp
#include <vector>

template<typename T> class pvector {
    string filename;
    vector<T> v;
    pvector_serializer<T> *serializer;

public:
    pvector(string fname, pvector_serializer<T> *ser)
    : filename(fname), serializer(ser) {
        serializer->readvector();
    }

    ~pvector() { serializer->writevector(); }

    ...}
```
pvector for string? (cont’d)

- Current solution is rather coarse grained
- Would be better to just factor out the read and write function
- Gives better reuse (left as exercise)

```cpp
template<typename T> struct element_serializer {
    virtual void read(ifstream &i, T &elem) { ... };
    virtual void write(ofstream &o, const T &elem) { ... };
};
```

- Trouble is we always use virtual method calls although
  - We know the type T
  - Typically, we know the serializer to be used upfront
Templates: Trait Classes

- Templates can also be used to provide polymorphism
  - Same functionality, same everything, just more efficient
- Inheritance allows to define an interface to be provided by subtypes
- Templates allow us to define an interface to be used (traits)

```cpp
template<typename T> struct persistence_traits {
    static void read(ifstream &i, T &elem) {...}
    static void write(ofstream &o, const T &elem) {...}
};
```

- With inheritance subtypes may override members
- The traits class is specialized for different types (override members)

```cpp
template<> struct persistence_traits<string> {
    static void read(ifstream &i, T &elem) {...}
    static void write(ofstream &o, const T &elem) {...}
};
```

Th. Gschwind. Fortgeschrittene Programmierung in C++. 11
pvector: Refactoring

- First, let’s refactor the pvector class to move the read and write methods into external classes...
pvector: Using the Trait class

```cpp
template<
typename T>
struct persistence_traits {
    static void read(ifstream &i, T &elem) {...}
    static void write(ofstream &o, const T &elem) {...} }

template<>
struct persistence_traits<string> {
    static void read(ifstream &i, T &elem) {...}
    static void write(ofstream &o, const T &elem) {...} }

template<
typename T>
class pvector {
    void writevector() {
        ofstream ofs(filename);
        vector<T>::iterator fst=v.begin(), lst=v.end();
        while(fst!=lst) persistence_traits<T>::write(ofs, *fst++); }
```

Depending on the type of container in use — can be determined by the compiler

Th. Gschwind. Fortgeschrittene Programmierung in C++. 13
pvector for string? (cont’d)

- Factor out the persistence logic in a separate class
  - Yes, not so repetitive
  - Yes, the persistence logic can be reused
  - Yes, everything determined at compile time

- But, what if we want to use different persisters?
  - Now, let’s use our traits class in a polymorphic way
  - Sort of like we are used from object-oriented programming
  - Except that types are resolved during compile time
Traits are similar to Inheritance but Different

- Inheritance allows us to use different implementations
- Similarly, allow to specify the use of different implementations, independently of the trait’s type
- The key difference is that all type inference is known, verified, and resolved during compile time

```cpp
template<typename T, typename P>
class pvector {
  void writevector() {
    ofstream ofs(filename);
    vector<T>::iterator fst=v.begin(), lst=v.end();
    while(fst!=lst) P::write(ofs,*fst++);
  }
};

pvector<string, persistence_traits<string>> > avector;
```
pvector for string? (cont’d)

- Make the persistence class generic
  - Yes, now we can even specify the persister
  - Yes, the code is as readable as generic Java code
  - Yes, BUT creating a new object becomes tedious
Templates: Default Values

- Always having to specify the persistence trait is tedious
- Like with function parameters, C++ allows to assign a default value to template parameters
- Now, we can only complain about certain style issues

```cpp
template<typename T, typename P = persistence_traits<T> >
class pvector {
  void writevector() {
    ofstream ofs(filename);
    vector<T>::iterator fst=v.begin(), lst=v.end();
    while(fst!=lst) P::write(ofs,*fst++);
  }
}
```

```cpp
pvector<string, persistence_traits<string> > avector;
```
pvector: Beauty Contest

- Typically, as part of a parameterized class, we include certain typedefs for convenience and readability.
- We will come back to the usefulness when we talk about binders.

```cpp
template<
    typename T,
    typename P=persistence_traits<T> >
class pvector {

    ...

public:
    typedef P persister;
    typedef typename vector<T>::iterator iterator;

    ...

    void writevector() {
        ofstream ofs(filename);
        iterator fst=v.begin(), lst=v.end();
        while(fst!=lst) persister::write(ofs,*fst++);
    }
};
```

*typename helps the compiler to identify that the following is a typename.*
Agenda

- Templates II
  - Specialization
  - Traits
  - Default Values

- Standard Library II
  - Design
  - Algorithms
  - Function Operators
Algorithms

- Containers alone are useful but not so useful
  - Typically we search for elements, remove them, iterate over them, ...

- Examples
  - `find(...)`, `x`
    - Locates `x` in a sequence/container
    - Returns an iterator/pointer to that element
  - `find_if(...)`, `pred`
    - Similar to `find`, but iterator/pointer to first element fulfilling `pred`

- How shall we will solve it
  - Naive implementation of `find()`
  - Generalizing the `find()` function
  - Extend it to `find_if()`
C++ is based on C, and tries to be close to C

We start out reusing pointers for algorithms

```c
int *find(int *array, int n, int x) {
    int *p=array;
    for (int i=0; i<n; i++) {
        int val=*p;
        if (val==x) return p;
        p++;
    }
    return NULL;
}
```

Problem 1: Can only be used for `int` arrays

Problem 2: Assumes knowledge about the implementation (`int*`)
### find (V0.2)

- **Addressing 1:** replace the int* with a template pointer

```cpp
template<typename T>
T *find(T *array, int n, const T &x) {
    for (int i=0; i<n; i++) {
        T val=*array;
        if (val==x) return array;
        array++;
    }
    return NULL;
}
```

- **Problem 2:** Still assumes knowledge about the implementation (T*)
Iterators (cont’d)

- Work similar to a pointer
- Implement the following operations (at least)
  - `T operator*();`
  - `Ptr &operator++();` // ++prefix
  - `const Ptr &operator++(int);` // postfix++
  - `bool operator==(const Ptr &)` const;
  - `bool operator!=(const Ptr &)` const;
  - ...
find (V0.3)

- Addressing 2: We do not know the exact type of iterator
- Addressing 2: Iterator becomes another template parameter

```c++
template<
typename Iter,
typename T>
Iter find(Iter begin, int n, const T &x) {
    for (int i=0; i<n; i++) {
        T val=*begin;
        if (val==x) return begin;
        begin++;
    }
    return 0; // compiler error
}
```

Here the iterator’s value type is converted to a value of type T. If we do not want that, we have to write:

```c++
auto val=*begin; // C++11, or
typename Iter::value_type val=*begin;
```

- Problem 3: Return value if the element is not found
- Problem 4: We may not always know the number of elements (iterate over list, iterate over partial set, etc.)
find (V0.5)

- Addressing 3: Exception? Makes the algorithm clumsy to use
- Addressing 3: Special NULL iterator? Where do we get it from?
- Addressing 4: Pass in an end iterator? Recycle end as NULL iterator?
- Look for x in [begin, beyond)

```cpp
template<typename Iter, typename T>
Iter find(Iter begin, Iter beyond, const T &x) {
    for (; begin<beyond; begin++) {
        if (*begin==x) return begin;
    }
    return beyond;
}
```

- Problem 5: Efficiency
Postfix versus Prefix Increment Operator

- Postfix `var++` returns the old value and increments `var`
- Prefix `++var`, increments `var` and returns its value

```cpp
T &Iterator2::operator++() {  // prefix
    this->pos++;
    return *this;
}

T Iterator2::operator++(int) {  // postfix
    T t(*this);
    this->pos++;
    return t;
}
```

- Hence postfix `++/--` copy the iterator and are less efficient
- Prefer prefix `++/--` operator over the postfix operator
find (V0.6)

- Addressing 5: Use prefix ++ operator

```cpp
template<
typename Iter, typename T>
Iter find(Iter begin, Iter beyond, const T &x) {
    for (; begin<beyond; ++begin) {
        if (*begin==x) return begin;
    }
    return beyond;
}
```

- Can be used for any container, any data type, any iterator, and works well! :-)

- Problem 6: Style (as always)
find (V1.0)

```cpp
template<typename Iter, typename T>  
Iter find(Iter begin, Iter beyond, const T &x) {  
  while ((begin!=beyond) && (*begin!=x)) ++begin;  
  return begin;  
}
```

- Perfectly optimal and generic function
- It doesn’t get better than that
- Any questions?
find_if (V0.1)

```
template<typename Iter>
Iter find_if(Iter begin, Iter beyond,
        void(*pred)(typename Iter::value_type)) {
    while (begin!=beyond) {
        if(pred(*begin)) break;
        ++begin;
    }
    return begin;
}
```

- Need to always be a function, looks clumsy
- Function always called through a function pointer
- Better solution, C++ allows functions to be template arguments as well
Pointer to Functions

- Functions can be viewed as variables
- Variables that are constant and have the code of the function assigned to them
- As consequence, we can use the pointer to a function representing its address.
- Can also be achieved by interfaces but we do not always want to create a new interface
- Sounds theoretical? The benefits gained by this, certainly are not.
find_if (V1.0)

```cpp
template<
typename Iter,
typename Pred>
Iter find_if(Iter begin, Iter beyond, Pred pred) {
    while (begin!=beyond) {
        if (pred(*begin)) break;
        ++begin;
    }
    return begin;
}
```

- **pred** may be a
  - Function (as before)
  - Lambda expression (again as before), or a
  - Function object (we will soon come to this)
Some More Algorithms (#include <algorithm>)

- Non mutating
  - for_each, find, adjacent_find, count, mismatch, ...

- Mutating
  - copy, swap, transform, replace, fill, unique, ...

- Sorting
  - sort, nth_element, binary_search, partition, ...

- Sets
  - includes, set_union, set_intersect, set_difference, ...

Decide which procedures you want; use the best algorithms you can find.

Bjarne Stroustrup
for_each

```cpp
template <typename In, typename Op>
Op for_each(In first, In last, Op f);
```

- Calls `f` for each element in `[first, last)`
- O(n)
- Attention!
  - There is a controversy whether this algorithm may change its elements
  - It is listed as non-mutable algorithm
  - Arguments have been made whether this applies only to the element order or the elements (first to last) itself
  - It is not enforced by the compiler
Output with for_each

```cpp
template <typename In, typename Op>
Op for_each(In first, In last, Op f);
```

- f can be an unary function

```cpp
template <typename T>
void my_print(const T &x) {
    cout << x << endl;
}

int main(int argc, char *argv[]) {
    // ...
    for_each(v1.begin(), v1.end(), my_print);
}
Traditionally, C++ does not offer lambda functions (delegates, anonymous classes, etc.)

- C++ uses functors (functions and function objects)
- Functors are defined in the global context and not always where needed

```cpp
int cnt;
for_each(v.begin(), v.end(), [](int i) {
    cout << i << endl; // ++cnt;
});
```

[] indicates the environment that should be visible from within the lambda expression

- [&cnt] would pass a reference to the local variable cnt
- [=cnt] indicates a value
- [&] and [=] allow to indicate the entire current scope
- Careful, [=] can generate a lot of copies
Maximum with for_each

```cpp
template <typename In, typename Op>
Op for_each(In first, In last, Op f);
```

- f is another unary function

```cpp
int my_max_val;

template <typename T>
void my_max(const T &x) {
  if(x>my_max_val) my_max_val=x;
}

int main(int argc, char *argv[]) {
  // ...
  my_max_val=*v1.begin();
  for_each(v1.begin(), v1.end(), my_max);
}
```
Maximum with for_each II

```cpp
template <typename In, typename Op>
Op for_each(In first, In last, Op f);
```

- `f` can also be an instance of a class that implements `operator() / 1`
- A so-called function object

```cpp
template <class T> struct my_max: public unary_function<T,void> {
    T max;
    my_max(const T& x) : max(x) {}
    void operator() (const T& x) { if(x>max) max=x; }
}

int main(int argc, char *argv[]) {
    // ...
    my_max<T> my_max_obj(*v1.begin());
    cout << for_each(v1.begin(), v1.end(), my_max_obj).max;
}
```
Related: Range For (foreach) Operator (C++11)

- Copied over from Java (which copied it over from C# (which ...))
- Easier to use than C++’s standard for function
- Useful in combination with initializer lists

```cpp
void print(const vector<int> &v) {
    for(const auto &i : v) {
        cout << i << endl;
    }
}
```
transform

- Transform input and write it to out
- O(n)
Summary

- Templates II
  - Specialization
  - Traits
  - Default Values

- Standard Library II
  - Design
  - Algorithms
  - Function Operators
Exercise 1

- Adapt your persistent pvector to make use of the new persister trait. Also implement a pset that implements a persistent set (based on std::set).
Exercise 2

Simple Spell Checker

- Implement a simple spell checker. The spell checker takes two files as command line arguments, a dictionary file containing a list of correctly spelled words and a file whose content is to be checked. Upon startup, your program stores the words contained in the dictionary file in a set<string>. Then it reads every word in the file to spell check, checks whether each word is correctly spelled (i.e., contained in the dictionary file) and if not displays it on cout.

- Additionally, your program should allow the user to correct them or insert them into the dictionary if spelled correctly but not in the dictionary. Make use of your persistent set implementation from exercise 1.
**Exercise 2 – An Example**

<table>
<thead>
<tr>
<th>dict. txt</th>
<th>text. txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>correctly spelled words are stored in the dictionary</td>
<td>a comprehensive dictionary is important.</td>
</tr>
</tbody>
</table>

C: \> check dict.txt text.txt

a
(a)dd, (i)gnore, (r)eplace? i comprehensive
(a)dd, (i)gnore, (r)eplace? i is
(a)dd, (i)gnore, (r)eplace? i important
(a)dd, (i)gnore, (r)eplace? i
Exercise 3

- Change your RPN calculator such that the minimum function computes the minimum of all the numbers stored on the stack. Make use of the std::for_each routine.

Hints
- don’t forget to create a copy of your old RPN calculator for the 1st colloquium
Exercise 4

- Implement a template based version of the Connect 4. Connect 4 builds on a playing field composed out of 7 columns each having 6 rows. When a player puts a stone into a column, gravity pulls the stone towards the lowest unoccupied column. The player who first has 4 stones in a row (horizontally, vertically, diagonally) wins.

- After each turn display the game field using simple ASCII graphics. Implement the game in such a way that players can be exchanged easily using templates. The precise interfaces to follow will be published at the lecture’s homepage. It is important that you follow these interfaces religiously.
Exercise 5

- Implement a computer player. The computer player of this version does not have to be intelligent. At a minimum, however, the computer player should be able to identify whether he can win the game by placing a stone. Let your computer player compete against computer players from your colleagues.
Next Lecture

- Separate Compilation
- More Templates
- More Algorithms

Have fun solving the examples!
See you next week!