Writing Generic Functions

Lecture 20
Hartmut Kaiser
hkaiser@cct.lsu.edu
http://www.cct.lsu.edu/~hkaiser/spring_2015/csc1254.html
Code Quality

The only valid measurement of code quality: WTFs/minute

Good code.

Bad code.
Programming Principle of the Day

• Minimize Coupling - Any section of code (code block, function, class, etc.) should minimize the dependencies on other areas of code.

• This is achieved by using shared variables as little as possible.

• Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability

http://en.wikipedia.org/wiki/Coupling_(computer_programming)
Abstract

• So far we have concentrated on the abstractions provided to us by C++ and the Standard library

• Now we start looking into creating our own abstractions
  • This lecture talks about generic functions, which are functions with parameter types that we do not know until we call the functions
  • The next lectures will focus on abstract data types and later on object oriented programming
Generic Functions

• For all functions we’ve seen so far we knew the types of its parameters

• Seems natural, however we have already used (not written) functions which didn’t have that property

• For instance std::find()
  • Takes two iterators and a value
  • Usable for any appropriate type for any container
  • Implies we do not know types until we use the functions

• This is called a Generic Function
  • Key feature of C++
Generic Functions

• What exactly does it mean to have arguments of “any appropriate type”?
  • How can we know whether it will work for a given set of argument types?

• Two parts to that answer
  • Inside C++: the ways a function uses the arguments of unknown type constrains that arguments type
    • Function does x + y, this implies there is a defined operator + applicable to the types of ‘x’ and ‘y’
    • Implementation checks whether x + y is defined, and if yes, types of ‘x’ and ‘y’ are ‘appropriate’
Generic Functions

• Two parts to that answer
  • Outside C++: the way the Standards library constrains the argument types for its functions
    • Iterators: supports a collection of operations with well defined semantics
    • Function expecting iterators as arguments will use those in a way relying on the iterator semantics
  • Writing your own containers implies to write iterators exposing ‘appropriate’ operators and semantics
Median of Unknown Type

• Generic functions are implemented using *template functions*
  • Single definition for a family of functions (or types) that behave similarly
  • Types are parameters, relies on knowledge that different types still have common properties and behavior

• Template functions are written in terms of this common behavior
  • When function is used a concrete type is known, which allows to compile and link program
Median of Unknown Type

- That’s what we’ve used before:

```cpp
double median(vector<double> v)
{
    typedef vector<double>::size_type vec_sz;
    vec_sz size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    sort(v.begin(), v.end());
    vec_sz mid = size/2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```
Median of Unknown Type

That's what it looks like:

```
template <typename T>
T median(vector<T> v)
{
    typedef typename vector<T>::size_type vec_sz;
    vec_sz size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    sort(v.begin(), v.end());
    vec_sz mid = size/2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```
Median of Unknown Type

• For template functions, the type of T is deduced by the compiler when function is used:
  
  ```
  vector<int> vi; // = { 1, 2, 3, 4 }
  median(vi); // instantiates median with T == int
  
  vector<double> vd; // = { 1.0, 2.0, 3.0, 4.0 }
  median(vd); // instantiates median with T == double
  ```

  • The deduced template parameter type pervades the whole function:
    
    ```
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
    ```

    • As v is a vector<T>, v[mid] is of type T
Template instantiation

• When a template function is called the compiler instantiates a version of the function based on concrete types supplied
  • Concrete types are deduced from arguments
  • Return type cannot be deduced
  • Once deduced all occurrences of those types are ‘replaced’ by concrete ones

• Requires the compiler to see all of the code
  • Compiler needs access to all of the sources
  • Templates are often fully defined in header files
Generic Functions and Types

• What’s an ‘appropriate type’ when instantiating a template
  • Median: types stored in the passed vector need to support addition and division with normal arithmetic meaning

• But subtle problems may occur
  • This is ok:
    \[ \text{find(homework.begin(), homework.end(), 0);} \]
  • This is not (why?):
    \[ \text{accumulate(homework.begin(), homework.end(), 0);} \]
Generic Functions and Types

- std::find:

  ```cpp
template <typename Iterator, typename T>
  Iterator find(Iterator first, Iterator last, T const& val)
  {
    for (/**/; first != last; ++first)
      if (*first == val)
        break;
    return first;
  }
```

- std::accumulate:

  ```cpp
template <typename Iterator, typename T>
  T accumulate(Iterator first, Iterator last, T val)
  {
    for (/**/; first != last; ++first)
      val = val + *first;
    return val;
  }
```
Generic Functions and Types

- std::max is supposed to get arguments of the same type:

  ```cpp
  string::size_type maxlen = 0;
  maxlen = max(maxlen, name.size());
  ```

- Implementation:

  ```cpp
  template<typename T>
  T const& max(T const& left, T const& right)
  {
    return left < right ? right : left;
  }
  ```
Generic Functions and Types

• Why do we have that restriction (before C++11)?
• Unfortunately, this does not work:

```cpp
template <typename T1, typename T2>
const& max(T1 const& left, T2 const& right)
{
    return left < right ? right : left;
}
```

• But this does (C++11):

```cpp
template <typename T1, typename T2>
auto max(T1 const& left, T2 const& right) -> decltype(left < right ? right : left)
{
    return left < right ? right : left;
}
```
Generic Functions and Types

• And this does as well (C++14):

    template <typename T1, typename T2>
    auto max(T1 const& left, T2 const& right)
    {
        return left < right ? right : left;
    }
Data-structure Independence

• The median function we wrote can be called for any vector<T>, where T is an arithmetic type
  • Wouldn’t it be nice being able to use other containers as well (list, vector, map)?
  • Moreover, we would like to act on a part of the container, not always the full data set

• std::find:

```cpp
    find(c.begin(), c.end(), val);    // why?
    c.find(val);                  // why not?
    find(c, val);                  // why not?
```
Data-structure Independence

• Two questions, several answers:
  • By using iterators the library makes it possible to write a single ‘find’ function that can find a value in any contiguous part of any container
    • Even if we have to mention ‘c’ twice,
  • If we had c.find(val),
    • Then every container type ‘c’ needs to implement a member ‘find’
  • Moreover, if we had find(c, val),
    • Then we wouldn’t be able to use the algorithms for parts of a container
  • What about usage of rbegin()/rend() (i.e. reverse iterators)
Algorithms and Iterators

• Different containers expose iterators with different capabilities
  • std::vector exposes iterator allowing to ‘jump‘ to arbitrary element, std::list does not
  • Algorithms which rely on certain capabilities can’t be used with all iterators
  • All iterators expose similar functionality using similar names, i.e. operator++() for increment

• std::find uses only simple operations (all containers)

• std::sort uses more complex operations (vectors, and strings only)
Algorithms and Iterators

• Library defines five iterator categories
  • Each corresponds to a specific set of exposed container operations
  • Each library algorithm states what iterator category is compatible
    • Possible to understand what containers are usable with which algorithms
  • Each category corresponds to access strategy for elements, also limits usable algorithms
    • For instance: single pass algorithms, or random access algorithms
Sequential Read-Only Access

• std::find:

```cpp
template <typename Iterator, typename T>
Iterator find(Iterator first, Iterator last, T const& val)
{
    for (; first != last; ++first)
        if (*first == val)
            break;
    return first;
}
```

• Requires iterator operators: !=, ==, ++, *(for reading), -> for member access
  - Input iterators

• Not possible to store a copy of the iterator ‘to go back’

• All iterators we’ve seen so far are (at least) input iterators
Sequential Write-Only Access

- `std::copy`

```cpp
template <typename In, typename Out>
Out copy(In begin, In end, Out dest)
{
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

- Required iterator operators: `!=`, `==`, `++`, `*` (for writing)
  - Output Iterators, example: `std::back_inserter`

- Implicit requirements:
  - Do not execute `++it` more than once between assignments to `*it`
  - Do not assign a value to `*it` more than once without incrementing it

- Not possible to store a copy of the iterator to overwrite output
Sequential Read-Write Access

• std::replace

```cpp
template <typename For, typename X>
void replace(For beg, For end, X const& x, X const& y)
{
    while (beg != end)
    {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

• Required iterator operators: !=, ==, ++, * (for reading and writing), - for member access
  • Forward iterators

• Storing copy of iterator possible! Very handy!
Reversible Access

• `std::reverse`

```cpp
template <typename Bi>
void reverse(Bi begin, Bi end)
{
    while (begin != end) {
        --end;
        if (begin != end)
            swap(*begin++, *end);
    }
}
```

• Required iterator operators: `!=, ==, ++, --, *` (for reading and writing), `->` for member access
  • `Bidirectional iterators`

• The standard-library container classes all support bidirectional iterators.
  • Except C++11 `std::forward_list<>` (singly linked list)
Random Access

std::binary_search:

```cpp
template <typename Ran, class X>
bool binary_search(Ran begin, Ran end, X const& x) {
    while (begin < end) {
        // find the midpoint of the range
        Ran mid = begin + (end - begin) / 2;
        // see which part of the range contains x;
        // keep looking only in that part
        if (x < *mid)
            end = mid;
        else if (*mid < x)
            begin = mid + 1;
        else // if we got here, then *mid == x so we're done
            return true;
    }
    return false;
}
```
Random Access

- Required iterator operators:
  - !=, ==, ++, --, * (for reading and writing), -> for member access
  - Let’s assume p, q are iterators, n is integer
    - p + n, p - n, and n + p
    - p - q
    - p[n] (equivalent to *(p + n))
    - p < q, p > q, p <= q, and p >= q

- We’ve used std::sort

- Containers: std::vector, std::string
  - std::list is not random access (it’s bidirectional), why?
Iterator Ranges

- Algorithms take pair of iterators
  - `c.begin()` – refers to first element
  - `c.end()` – refers to first element after last (‘one off’)

- Allows simple handling of empty ranges, both iterators point to the one off element

- Allows for comparing iterators for equality (`!`/`=`, no need to define notion of iterators being larger/smaller than others

- Allows to indicate ‘out of range’, see `url_beg()`, where we returned end (off by one) iterator if nothing was found

- Only caveat for end iterators is that they can’t be dereferenced
Input/Output Iterators

• Why are they separate from forward iterators?
  • (no container requires them)

• One reason is
  • Not all iterators come from containers
    • back_inserter, usable with any container supporting push_back
    • Iterators bound to streams: using iterator operations those allow to access istreams and ostream
Input/Output Iterators

• The input stream iterator is an input-iterator type named `istream_iterator`:

```cpp
// read ints from the standard input and append them to v
vector<int> v;
copy(istream_iterator<int>(cin), istream_iterator<int>(),
backs_inserter(v));
```

• `istream_iterators` are templates!
  • Need to specify type to read from input
  • Same as for normal input operations, which are always typed as well
  • Default constructed `istream_iterator` denotes end of input
Input/Output Iterators

• The output stream iterator is an output-iterator type named ostream_iterator:

```cpp
// write the elements of v each separated from the other
// by a space
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " ") );
```

• ostream_iterator is a template as well
  • Need to specify type of required output
  • Second argument is separator (defaults to no separation)