Templates

Michael Hanke

School of Engineering Sciences

Program construction in C++ for Scientific Computing

Outline

The max Function: C Style

- Problem: Computation of the maximum of 2 expressions:

```
#define MAX(a,b) ((a) < (b) ? (b) : (a))
```

- `a` and `b` can be replaced by expressions of any type/class.
- Only requirement: The operator < must be defined.
- Consider the assignment

```
t = MAX(f(args1),g(args2));
```

- Problem: Depending on the outcome of the comparison, one of `f` or `g` must be evaluated twice!
- Side effects may become important.

The max Function

- A more efficient way would be to use inlined functions:

```
inline double max(double a, double b) {
  if (a < b) return b;
  else return a;
}
```

- If we would need the comparison of values with other types, a (large) set of functions with different signature must be written. *Inefficient!*
- The way out are parametrized functions.

The max Function: C++ Style

```
template<class T>
inline const T& max(const T& a, const T& b) {
  if (a < b) return b;
  else return a;
}
```

- This is an example of a function template.
- The type(s) are a parameter of the function.
- Such a programming style is called *generic programming*.
- Note: instead of `class`, the keyword `typename` can be used.

Usage of the max Function

```
class C {
  public:
    bool operator<(const C& b) const {...}
};
C x, y, z;
z = max(x,y);
double r = max(2.5,3.1);
```

- The compiler creates two instances of the max function (polymorphism at compile time):

```
const C& max<C>(const C&, const C&)
const double& max<double>(const double&, const double&)
```

- The interface to the class parameter must implement all requirements of the template.

Instance Generation

- In order to generate an instance, only a minimal number of type conversations take place:
  - A function parameter that is a reference (or pointer) to a const can be passed a reference (or pointer) to a non-const object.
  - For non-reference parameters, an array will be converted to a pointer to its first element.
- Consequently, in our example, the following call will give a compile time error:

```
max(2.1,3);
```

- Explicit instantiation: `max<double>(2.1,3)`.

Instance Generation (cont)

- A more general way:

```
template<class T1, class T2, class T3>
inline const T1& max(const T2& a, const T3& b) {
  if (a < b) return b;
  else return a;
}
```

- Since T1 cannot be deduced from the calling sequence during instantiation, the types have been provided explicitely:

```
max<C1,C2,C3>(a,b)
```

- Trailing parameters can be omitted if uniquely deductable: `max<C1>(a,b)`.

Code Generation

- Since instantiation is done during compile time, the complete source code must be available. Therefore, templates are always defined in a header file, not in the implementation!
- If a template function is used in many different source files with the same signature and not inlined, each object file contains the corresponding code many times!
  - This may lead to huge executables containing repeated copies of identical code.
- Explicit instantiation:

```
extern template d;  // instantiation declaration
template d;         // instantiation definition
```

  There can be many declarations in a program, but only one definition!
- *A note on compiler technology*: Template programming is extremely rich (examples later). Compiler quality in this respect is very different!

Template Classes

- In a previous lecture, we developed a `Point` class for describing points in the plane:

```
class Point {
  private:
    double x;
    double y;
  public:
    double X() const { return x; }
    double Y() const { return y; }
    void zero() { x = y = 0.0; }
};
```

A Template Point Class

- This class can easily generalized if we are interested in other accuracies, e.g., `float`:

```
template<class T>
class Point {
  private:
    T x;
    T y;
  public:
    T X() const { return x; }
    T Y() const { return y; }
    void zero() { x = y = 0.0; }  // ??
};
```

A Generalized Point Class

Idea: Write a class for points in any dimension

```
template<class T = double, int d = 2>
class Point {
  private:
    T coord[d];
  public:
    Point() { zero(); }  // etc
    ...
    void zero() {
      for (int i = 0; i < d; i++)
        coord[i] = T(0);
    }
};
```

Usage of Point Class

- Point<double,2>: Corresponds to our old class
- Point<double>: Identical to the previous instantiation
- Point<>: dto
- Point<int,3>: Model for $\mathbf{Z}^3$
- Point<Point<>,3>: *Guess what is this?*

Specialization

- The zero function is obviously not very useful for copying points of `double`s, `int`s etc.
- Instead, a simple `memset` (or `bzero`, `fill`, `cblas_dcopy`) would be sufficient (and much faster).
- This can be handled by *partial specialization*:

```
#include <cstring>
template<int d>
class Point<double,d> {
  private:
    double coord[d];
  public:
    Point() { zero(); }  // etc
    ...
    void zero() {
      memset(coord,0,d*sizeof(double));
    }
};
```

Specialization (cont)

- Functions can also be specialized:

```
template<>
inline void Point<double,2>::zero() {
  memset(coord,0,2*sizeof(double));
}
```

- Functions must always be completely specialized (in contrast to classes).

Standard Template Library

- The Standard Template Library (STL) contains a huge set of predefined (template) classes.
- They include container classes (e.g., vectors, lists, I/O routines), standard algorithms (even random numbers!), complex numbers, strings, and many more.
- It is usually very convenient (and bug free!) to use these libraries.
- *Use your own implementation of standard objects and algorithms only if you have a good reason for doing so!*
- A good reason might be efficiency.

Sequential Containers

- The sequential containers are designed to provide very fast sequential access to their elements.
- The difference lies
  - in the costs to add or delete elements to the container;
  - in the costs to perform nonsequential access to the elements.

container	property
`vector`	Flexible size array. Fast random access.
`deque`	Double-ended queue. Fast random access. Fast append
`list`	Doubly linked list. Bidirectional sequential access
`forward_list`	Singly linked list. Tuned for performance
`array`	Fixed size array. Shall replace C-style arrays
`string`	Similar to vector. Intended for character strings.

The vector Class

- The include file for the vector class is vector:

```
#include <vector>
```

- The elements can be of any type. When defining a variable of a container class, the type must be provided in < .. >:

```
vector<double> v;
vector<int> u;
vector<Point> polygon;
```

  The number of elements in these vectors is 0.

- Vectors can be created having a certain size:

```
vector<double> v(10);          // zeros(10,1)
vector<int> u = {1,2,3,4,5}; // integer vector with
                               // 5 elements
vector<double> w(4,1.1);      // 1.1*ones(4,1);
```

- Copy constructors, assignment etc work as expected.

The vector Class: Selected Funtions

function	description
size()	length of vector
empty()	returns true if size() == 0
assign({...})	replace elements by the given values
assign(int,val)	
clear()	Remove all elements
resize(int)	set the new length
push_back(...)	Add an element
pop_back()	remove the last element

The vector Class: Element Access

function	
v[i]	Index without bounds check
v.at(i)	Index with bounds check
data()	Pointer to the first element

It is easy to traverse a vector:

```
for (int i = 0; i < v.size(); i++)
  std::cout << v[i] << "  ";
  std::cout << std::endl;
```

The `array` Class

- The vector class is very easy to use.
- However, growing arrays require a great deal of memory management. This leads to inefficiencies compared to C-style arrays.
- The memory management of `vector` can be tuned to a certain amount.

- In the C++11 standard, a *fixed-size class for vectors* has been introduced: `array`

```
#include <array>
array<double, 10> a
```

- The latter definition introduces an array of 10 elements. *The length of the array must be known at compile time!*
- All members of vector are defined for array, too (of course, with the exception of operations changing the size of the array).

Constructors etc

- In contrast to C-style arrays, the default constructors correspond to deep copy.
- There is no need to define its own deep copy.

Example: Domain Revisited

```
class Domain {
  public:
    Domain(const Curvebase&, const Curvebase&,
           const Curvebase&, const Curvebase&);
    Domain(const Domain&);
    Domain &operator=(Domain &);
    ~Domain();
    void generate_grid (int m, int n);
    // more members
  private:
    Curvebase *sides[4];
    double *x_, *y_;
    int m_, n_;
    bool check_consistency();
    // more members
};
```

Example (cont)

```
class Domain {
  public:
    Domain(const Curvebase&, const Curvebase&,
           const Curvebase&, const Curvebase&);
    Domain(const Domain&);
    Domain &operator=(Domain &);
    ~Domain();
    void generate_grid (int m, int n);
    // more members
  private:
    Curvebase *sides[4];
    vector<double> x_, y_;
    int m_, n_;
    bool check_consistency();
    // more members
};
```

Example (cont)

Old version:

```
Domain::grid_generation(int m, int n) {
  if (m <= 0 || n <= 0) ; // Do something meaningful
  else {
    if (m_ > 0) { // There exists already a grid!
      delete [] x_;
      delete [] y_;
    }
    m_ = m; n_ = n;
    x_ = new double[m_*n_];
    y_ = new double[m_*n_];
    // Fill x_[] and y_[] with values!
  }
}
```

Example (cont)

New version:

```
Domain::grid_generation(int m, int n) {
  if (m <= 0 || n <= 0) ; // Do something meaningful
  else {
    x_.resize(m_*n_);
    y_.resize(m_*n_);
    // Fill x_[] and y_[] with values!
  }
}
```

A Special Construct

- Sometimes it is necessary to call basic functions which do not have a C++ interface (for example, MPI or numerical libraries).
- They can be used by including the function declaration in

```
extern "C" { ... }
```

- Example:

```
extern "C" {
void cblas_daxpy(const int N, const double alpha,
                 const double *X,
                 const int incX, double *Y,
                 const int incY);
}
```

Motivation

- Let a[N] be a C-style array. Then we can traverse the elements of a in two ways:

```
double a[N];
for (int i = 0; i < N; i++) do_something(a[i]);
for (double *p = a; p < a+N; p++) do_something(*p);
```

- For vector, array, deque, string the first possibility is also available.
- *What about the second version?*
- *What about other container classes?*

Here, iterators are useful.

Iterators

- Iterators are used for traversing objects of container classes.
- Many operations for container classes require iterators as parameters.
- Many functions from the standard library (so-called *algorithms*) that handle containers require iterators as parameters.

How to Use Iterators

```
vector<double> a(N);
vector<double>::iterator i;
for (i = a.begin(); i < a.end(); i++) do_something(*i)
```

- *An iterator is a pointer to an element in a container object.*
- The *first element* can be obtained by calling `begin()`.
- A pointer to an *element after the last* is obtained via `end()`.
- A pointer to the *next element* is obtained by the increment operator.
- In C++11, i need not be explicitly defined:

```
for (auto i = a.begin(); i < a.end(); i++)
```

Reverse Iterators

- In order to go through the container object backwards, one could use the decrement operator.
- It is, however, usually more efficient to use a reverse iterator:

```
(auto rit = a.rbegin(); rit != a.rend(); rit++)
```

Note: Decrementing a reverse iterator means traversing the object *forwards*!

Types of Iterators

Classification according to access/traversing:

- *class*::iterator
- *class*::const_iterator
- *class*::reverse_iterator
- *class*::const_reverse_iterator

Classification according to category:

- forward iterator
- bidirectional iterator
- random-access iterator

In principle, all meaningful operations are defined.

Final Remarks

- Container classes are very convenient to use.
- *Container classes may have a bad impact on efficiency!* In particular, reallocation is very time-consuming.
- When choosing a container class, *think twice!*
- Sometimes, it is much more efficient to reinvent the wheel.
- Compare the comments in: Agner Fog, *Optimizing software in C++*, `http://www.agner.org/optimize`.

Summary

- Template functions and classes
- Instantiation, specialization, partial specialization
- Some container classes from STL
- Iterators

- What comes next
  - Efficiency considerations