Generic programming and templates

function overloading

How do we compute the minimum of 2 variables?

```
int min(int a, int b) {
  return (a < b ? a : b);
}
```

But what do we do when we also want to know the minimum of 2 floating variables?

```
int min_int(int a, int b) {
  return (a<b ? a : b);
}

float min_float(float a, float b) {
  return (a<b ? a : b);
}
```

C++ allows the function name to be overloaded and just distinguished by its arguments,

```
float min(float a, float b) {
  return (a<b ? a:b);
}

double min(double a, double b) {
  return (a<b ? a:b);
}
```

How does the compiler deal with the same function name? Check the output of the .s and .o files!

function overloading

We need however to be careful:

```
min(1,2); // OK
min(4., 5.); // OK
//min(5, 3.1); // does not compile, call to 'min' is ambiguous
```

C++ allows for a generic implementation:

```
template <class T>
inline T min(T x, T y) {
  return (x < y ? x : y);
}
```

- is generic : templates are the prime example of generic programming in C++
  - works for any type *T*
  - close to the abstract notion of *min*
  - condition : the operator < needs to be defined for the type *T* and resulting in a bool
  - *copy* construction for type T: we need to copy the result!
  - is concise (no code doubling) and is efficient (can be *inlined*)

How does this work? Check out the output of the .s and the .o files!

Usage causes instantiation: whenever min is called for an unknown type T,
the compiler generates a new version of min specific for the type T

advantage: evaluated at compile time so no performance drawbacks when executing the code (but the compile time might increase)

different types

What if we want to use different types?

```
template <typename R, typename U, typename T>
R const min(U const& x, T const& y) {
  return (x < y ? static_cast<R>(x) : static_cast<R>(y));
}
```

The compiler cannot infer the type R from the function arguments, hence we need to specify it:

```
min<int>(a,b);      // OK
min<double>(a,b);   // OK
```

note: in C++11 a variadic template name is allowed

template specialization : sometimes we want to have a particular implementation for certain values

```
template <class T>
inline T Mymin(T x, T y) {
  return (x < y ? x : y);
}

template <>
char Mymin(char x, char y) {
  char x1= ((x>='a')&&(x<='z')) ? x + 'A'-'a' : x;
  char y1= ((y>='a')&&(y<='z')) ? y + 'A'-'a' : y;
  return x1 < y1 ? x : y;
}
```

Note the syntax with template<>

the use of typename

consider the code snippet based on Ch 8 in Accelerated C++:

```
template <class T>
T median(std::vector<T>& v) {

  typedef typename std::vector<T>::size_type vec_sz;       // NOVELTY

  vec_sz size = v.size();

  if (size == 0)
    throw std::domain_error("median of an empty vector");

  std::sort(v.begin(), v.end());                           // note the use of iterators

  vec_sz mid = size/2;

  return size%2 ? v[mid] : (v[mid] + v[mid+1])/2;

}
```

in the typedef the use of typename tells the implementation that std::vector<T>::size_type is the name of a type even though the implementation does not know what type T is. Whenever you have a type that depends on a template parameter and you want to use a member of that type, you must use typename

STL and Iterators

the function sort defined in <algorithm.h> takes 2 iterators as function arguments. Since we did not specify the type of these iterators, it is clear that sort is based on templates.

This is good design practice: we want functions such as sort to be *independent of the data structure*

We also want to be able to act on the relevant part of the container rather than having to use the entire container. The solution is provided by iterators

One distinguishes 5 iterator categories:

- *input iterator*: sequential access in one direction, input only
- *output iterator*: sequential access in one direction, output only
- *forward iterator*: sequential access in one direction, input and output
- *bidirectional iterator*: sequential access in one directions, input and output
- *random-access iterator*: efficient access to any element, input and output

Examples

Sequential read-only access can be seen in the following implementation of find, which looks for the first element where a value x is found in a sequence:

```
template <class InputIterator, class X>
InputIterator myFind(InputIterator begin, InputIterator end, const X& x) {
  while (begin != end) {
    if (*begin == x) return begin;
    ++begin;
  }
  return end;
}
```

requirements: an input iterator should support *++*, *==*, *!-*, unary *** (as well as *->*)

Sequential write-only access can be seen in the following implementation of copy

```
template <class InputIterator, class OutputIterator>
OutputIterator myCopy(InputIterator first, InputIterator last, OutputIterator dest) {
  while (first != last) *dest++ = *first++;
  return dest;
}
```

we only need to evaluate **dest = value*, hence this is an output iterator. It should also support ++. An additional requirement is that no double increment operations without intermediate assignment is allowed (otherwise a gap would be created in the output)

checkout the example programs of this week for the remaining iterator specifications

the power of the STL

what is the output of the following program?

```
int main()
{
  vector<string> data;
  copy(istream_iterator<string>(cin), istream_iterator<string>(), back_inserter(data));
  sort(data.begin(), data.end());
  copy(data.begin(), data.end(), ostream_iterator<string>(cout," ")); cout << "\n";
  // unique_copy(data.begin(), data.end(), ostream_iterator<string>(cout," "));
}
```

on input : bayern won against dortmund

on input: all the works by Goethe

template meta programming

In 1994 Erwin Unruh wrote a program that prints all prime numbers based on template meta programming only!

```cpp
#include <iostream>

template <int p, int i>
class is_prime {
public:
    enum { prim = ( (p % i) && is_prime<p, i - 1>::prim ) };
};

template <int p>
class is_prime<p, 1> {
public:
    enum { prim = 1 };
};

template <int i>
class Prime_print {      // primary template for loop to print prime numbers
public:
    Prime_print<i - 1> a;
    enum { prim = is_prime<i, i - 1>::prim };
    void f() {
        a.f();
        if (prim)
        {
            std::cout << "prime number:" << i << std::endl;
        }
    }
};

template<>
class Prime_print<1> {   // full specialization to end the loop
public:
    enum { prim = 0 };
    void f() {}
};


#ifndef LAST
#define LAST 18
#endif

int main()
{
   Prime_print<LAST> a;
   a.f();
}
```

```
output:
prime number:2
prime number:3
prime number:5
prime number:7
prime number:11
prime number:13
prime number:17
```

try to figure out how it works!

the C++ compiler with template support is *Turing complete* (aka a computer in its own)

The concept of C++ template metaprogramming as a serious programming tool was first made popular by Todd Veldhuizen, who used and developed this in the Blitz++ library

template type deduction

(Scott Meyers, Effective Modern C++, ch 1)

A templated function and its function call can be written as

```
template<typename T>
void f(ParamType param);

f(expr);
```

The type *T* and *ParamType* have to be separately determined from *expr*, for instance

```
template<typename T>
void f(const T& param);
int x = 27;  f(x);      // x is an int; call f with an int; T is int, ParamType is const int&
```

but what happens with *const* variables, and with *references* (*pointers* behave the same way)? Let

```
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int
```

we first consider the case of references

```
template<typename T>
void f(T& param);       // param is a reference

f(x);                   // T is int,       param's type is int&
f(cx);                  // T is const int, param's type is const int&
f(rx);                  // T is const int, param's type is const int&
```

hence, if ParamType is a reference or a pointer

1. if *expr* is a reference, ignore the reference part
2. match the pattern of *expr* with *ParamType* to determine *T*

template type deduction

now consider pass-by-value

```
template<typename T>
void f(T param);          // param is now passed by value

f(x);                     // T's and param's types are both int
f(cx);                    // T's and param's types are again both int
f(rx);                    // T's and param's types are still both int
```

in this case, *param* is always copied. Note that *param* is never const - it is a copy of cx and rx, and nothing says that this copy cannot be modified.

there are further details to know about template type deduction : read the book of Scott Meyers if you want to know more.

argument dependent lookup

- Also known as Koenig lookup, although he was not the inventor
- originally thought of for the operators >> and <<
- extends the number of namespaces where the function is looked for depending on the namespace of the arguments of the function

```
namespace Lecture {
  class Student {};
  void foo(Student& s, int i) {};
}

int main() {
  Lecture::Student s;
  foo(s, 100); // calls Lecture::foo
}
```

- is essential for the following “hello, world!” program

```
#include <iostream>

int main() {
  std::string s = "Hello, world!";
  std::cout << s << "\n";   // standard example of ADL
}
```

the << is replaced by operator<<(…) , not by std::operator<<(…) . So it is only through the argument (std::string) and argument dependent lookup that the code behaves as expected

- can be tricky: what is really called when using max(), swap(), … ? depending on the arguments the output of std::swap(a,b) and swap(a,b) may be different: if a and b are members of a class N, then N::swap(N::a, N::b) will be called by the latter, but not by the former. Note that the STL relies heavily on overloading swap().

example: integration

- the simplest possible integration scheme of a 1-dimensional function is to replace *dx* by a finite amount *h*

- we wish to write an integration library for arbitrary functions

```
// the integration routine
template<typename F, typename Prec>
double integrate(F& f, Prec a, Prec b, size_t steps)
{
  double s = 0;
  double h = (b-a)/steps;
  for (size_t i = 0; i < steps; ++i)
    s += f(a + i*h);          // this implies that () must be defined on f with arguments of type double
  return h*s;
}
```

- it is however not very accurate: locally of order h^2, globally of order h

- evaluating the function in the middle of the interval is already more accurate

- in the exercises you will implement the trapezoid and the Simpson integration scheme. Ideally, you can specify the integration method as a further argument to the integrate function. Then the implementation of the integrator can be shielded from the user; the user only needs to know the interface

example: power method

- repeated action of the Hamiltonian on a trial vector results in the largest eigenvalue and corresponding eigenvector
- the power method is hence a simple eigenvalue solver

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $v = y/\|y\|_2$
(4) $y = Av$
(5) $\theta = v^* y$
(6) **if** $\|y - \theta v\|_2 \leq \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

(from www.netlib.org)

- used in the PageRank algortihm and by Twitter
- basis for Arnoldi and Lanczos (Krylov subspace methods) iterations

example: power method

- generic implementation:

```
Op A;
V v,y;
T theta, residual, tolerance;


// initialize …                          // line 1

do {
  v = y / norm(y);                      // line 3
  y = A * v;                            // line 4
  theta = dot(v, y);                    // line 5
  v *= theta;                           // line 6
  v -= y;                               // line 6
  residual = norm(v);                   // line 6
} while (residual > tolerance * abs(theta));
```

- we need:
  - V must be a vector in the Hilbert space over T
  - A must be a linear operator in the Hilbert space
  - action of linear operator on vector (line 4)
  - dot product of a vector (line 5) and its norm (line 3)
  - multiplication and division of a vector with a scalar (line 3 and line 6)
  - addition, subtraction (also with assignment) of vectors (line 6)
  - copy constructor, assignment, … of vectors
  - etc