# Templates and generic programming

## Improving on last week's assignment

- How did you calculate the machine precision?
  - Did you just have a main() function
  - Did you have three functions with different names?
    - `epsilon_float()`
    - `epsilon_double()`
    - `epsilon_long_double()`
  - Did you have three functions with the same name?
    - `epsilon(float x)`
    - `epsilon(double x)`
    - `epsilon(long double x)`
  - Or did you have just one function that could be used for any type?
    - `epsilon()`

## Generic algorithms versus concrete implementations

- Algorithms are usually very generic:
  - for min() all that is required is an order relation "<"

$$\min(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

- Most programming languages require concrete types for the function definition
  - C:
    - `int min_int(int a, int b) { return a<b ? a : b;}`
    - `float min_float (float a, float b) { return a<b ? a : b;}`
    - `double min_double (double a, double b) { return a<b ? a : b;}`
    - …
  - Fortran:
    - `MIN()`, `AMIN()`, `DMIN()`, …

## Function overloading in C++

- solves one problem immediately: we can use the same name
  - `int min(int a, int b) { return a<b ? a : b;}`
  - `float min (float a, float b) { return a<b ? a : b;}`
  - `double min (double a, double b) { return a<b ? a : b;}`
- Compiler chooses which one to use
  - `min(1,3);` // calls min(int, int)
  - `min(1.,3.);` // calls min(double, double)
- However be careful:
  - `min(1,3.1415927);` // Problem! which one?
  - `min(1.,3.1415927);` // OK
  - `min(1,int(3.1415927));` // OK but does not make sense
  - or define new function `double min(int,float);`

## C++ versus C linkage

- How can three different functions have the same name?
  - Look at what the compiler does
    - 
```
cd PT
cvs update -d
cd week3
g++ -c -save-temps -O3 min.C
```
  - Look at the assembly language file min.s and also at min.o
    - 
```
nm min.o
```
- The functions actually have different names!
  - Types of arguments appended to function name
- C and Fortran functions just use the function name
  - Can declare a function to have C-style name by using `extern "C"`
    `extern "C" { short min(short x, short y);}`

## Using macros (is dangerous)

- We still need many functions (albeit with the same name)
- In C we could use preprocessor macros:
  - `#define min(A,B) (A < B ? A : B)`
- However there are serious problems:
  - No type safety
  - Clumsy for longer functions
  - Unexpected side effects:

```
min(x++,y++);  // will increment twice!!!
               // since this is: (x++ < y++ ? x++ : y++)
```

- Look at it:
  - `c++ -E minmacro.C`

## Generic algorithms using templates in C++

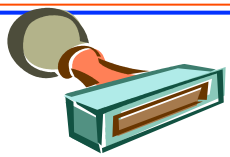
- C++ templates allow a generic implementation:

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

$$\min(x,y) \text{ is } \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

- Using templates we get functions that
  - work for many types `T`
  - are optimal and efficient since they can be inlined
  - are as generic and abstract as the formal definition
  - are one-to-one translations of the abstract algorithm

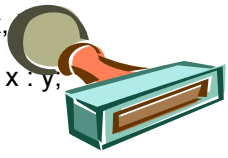
## Usage Causes Instantiation



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

```
int x = min(3, 5);
int y = min(x, 100);
```

```
// T is int
int min<int>(int x,
{
    return x < y ? x : y;
}
```



```
float z = min(3.14159f, 2.7182f);
```

```
// T is float
float min<float>(float x, float y)
{
    return x < y ? x : y;
}
```

## Discussion

"What is Polymorphism?"

Our definition:

*Using many different types through the same interface*

## Generic programming process

- Identify useful and efficient algorithms
- Find their generic representation
  - Categorize functionality of some of these algorithms
  - What do they need to have in order to work **in principle**
- Derive a set of (minimal) requirements that allow these algorithms to run (efficiently)
  - Now categorize these algorithms and their requirements
  - Are there overlaps, similarities?
- Construct a framework based on classifications and requirements
- Now realize this as a software library

## Generic Programming Process: Example

- (Simple) Family of Algorithms: min, max
- Generic Representation

$$\min(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

$$\max(x,y) = \begin{cases} x & \text{if } x > y \\ y & \text{otherwise} \end{cases}$$

- Minimal Requirements?
- Find Framework: Overlaps, Similarities?

## Generic Programming Process: Example

- (Simple) Family of Algorithms: min, max
- Generic Representation

$$\min(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

$$\min(x,y) = \begin{cases} x & \text{if } y < x \\ y & \text{otherwise} \end{cases}$$

- Minimal Requirements yet?
- Find Framework: Overlaps, Similarities?

## Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool

## Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool
  - Copy construction: need to copy the result!

## Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool
  - that's all!

## Concepts

- A concept is a set of requirements, consisting of valid expressions, associated types, invariants, and complexity guarantees.
- A type that satisfies the requirements is said to model the concept.
- A concept can extend the requirements of another concept, which is called refinement.
- A concept is completely specified by the following:
  - Associated Types: The names of auxiliary types associated with the concept.
  - Valid Expressions: C++ expressions that must compile successfully.
  - Expression Semantics: Semantics of the valid expressions.
  - Complexity Guarantees: Specifies resource consumption (e.g., execution time, memory).
  - Invariants: Pre and post-conditions that must always be true.

## Assignable concept

- Notation
  - X — A type that is a model of Assignable
  - x, y — Object of type X

| Expression | Return type | Semantics | Postcondition |
|---|---|---|---|
| x=y; | X& | Assignment | X is equivalent to y |
| swap(x,y) | void | Equivalent to<br>{<br>X tmp = x;<br>x = y;<br>y = tmp;<br>} | |

## CopyConstructible concept

- Notation
  - X — A type that is a model of CopyConstructible
  - x, y — Object of type X

| Expression | Return type | Semantics | Postcondition |
|---|---|---|---|
| X(y) | X& | | Return value is equivalent to y |
| X x(y); | | Same as<br>X x;<br>x=y; | x is equivalent to y |
| X x=y; | | Same as<br>X x;<br>x=y; | |

## Documenting a template function

- In addition to
  - Preconditions
  - Postconditions
  - Semantics
  - Exception guarantees
- The documentation of a template function must include
  - Concept requirements on the types
- Note that the complete source code of the template function must be in a header file

## Argument Dependent Lookup

- Also known as Koenig Lookup
- Applies only to *un*qualified calls

  abs(x) ~~std::abs(x)~~
- Examines "associated classes and namespaces"
- Adds functions to overload set
- Originally for operators, e.g. operator<<(std::ostream&, T);

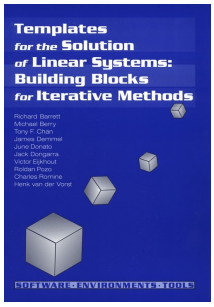
```
namespace lib  {
    template <class T> T abs(T x)
        { return x > 0 ? x : -x; }

    template <class T>
    T compute(T x) {
        …
        return abs(x);
    }
}
namespace user {
    class Num {};
    Num abs(Num);
    Num x = lib::compute(Num());
}
```

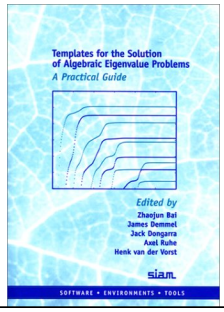
?

## Examples: iterative algorithms for linear systems

- Iterative template library (ITL)
  - Rick Lee *et al*, Indiana
- generic implementation of iterative solvers for linear systems from the “Templates” book




- Iterative Eigenvalue Template Library (IETL)
  - Prakash Dayal *et al*, ETH
- generic implementation of iterative eigensolvers. partially implements the eigenvalue templates book




## The power method

- Is the simplest eigenvalue solver
  - returns the largest eigenvalue and corresponding eigenvector

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* $x = v$

- Only requirements:
  - $A$ is linear operator on a Hilbert space
  - Initial vector $y$ is vector in the same Hilbert space
- Can we write the code with as few constraints?

## Generic implementation of the power method

- A generic implementation is possible

```
OP A;
V v,y;
T theta, tolerance, residual;
…
do {
  v = y / two_norm(y);          // line (3)
  y = A * v;                    // line (4)
  theta = dot(v,y);             // line (5)
  v *= theta;                   // line (6)
  v -= y;
  residual = two_norm(v); // ||θ v - Av||
} while(residual>tolerance*abs(theta));
```

## Concepts for the power method

- The triple of types (T,V,OP) models the Hilbertspace concept if
  - `T` must be the type of an element of a field
  - `V` must be the type of a vector in a Hilbert space over that field
  - `OP` must be the type of a linear operator in that Hilbert space
- All the allowed mathematical operations in a Hilbert space have to exist:
  - Let `v`, `w` be of type `V`
  - Let `r`, `s` of type `T`
  - Let `a` be of type `OP`.
  - The following must compile and have the same semantics as in the mathematical concept of a Hilbert space:
    - `r+s`, `r-s`, `r/s`, `r*s`, `-r` have return type `T`
    - `v+w`, `v-w`, `v*r`, `r*v`, `v/r` have return type `V`
    - `a*v` has return type `V`
    - `two_norm(v)` and `dot(v,w)` have return type `T`
    - …
  - Exercise: complete these requirement