Programming Techniques for Scientific Simulations I

Introduction to templates and generic programming

Improving on the first week's assignment

- Quiz: How did you calculate the machine precision?
  1) Did you just have a main() function?
  2) Did you have three functions with different names?

```
epsilon_float()
epsilon_double()
epsilon_long_double()
```

  3) Did you have three functions with the same name?

```
epsilon(float x)
epsilon(double x)
epsilon(long double x)
```

  4) 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 “<“

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

- Some programming languages require concrete types for the function definition
  - C:

```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; }
```

```c
abs, labs, llabs, imaxabs, fabs, fabsf, fabsl, cabsf, cabs, cabsl
```

  - Fortran: `MIN(), AMIN(), DMIN(), ...`

Function overloading in C++

- Solves one problem immediately
  - We can use the same name

```
int min(int x, int y) { return x<y ? x : y; }
float min(float x, float y) { return x<y ? x : y; }
double min(double x, double y) { return x<y ? x : y; }
```

- Compiler chooses which one to use (so-called overload resolution)

```
min(1,3);   // calls min(int, int)
min(1.,3.); // calls min(double, double)
```

Details can be quite complicated...

- 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, double);
```

C++ versus C linkage

- How can three different functions have the same name?
  - Look at what the compiler does

```
c++ -c -save-temps -O3 min.cpp
```

  - 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 (so-called name mangling... Can be "demangled" by `c++filt`)
- C and Fortran functions just use the function name
  - Can declare a function to have C-style name by using

```
extern "C" { short min(short x, short y); }
```

Using macros (can be/is dangerous)

- We still need many functions (albeit with the same name)
- We could use preprocessor macros

```
#define min(x,y) (x < y ? x : y)
```

- However there are serious problems
  - No type safety
  - Clumsy for longer functions
  - Unexpected side effects

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

- Look at it:

```
c++ -E minmacro.cpp
```

- (In C this is common with the convention: `UPPERCASE` for macros)

Generic algorithms using templates in C++

- C++ (function) templates allow a generic implementation

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

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

- With function templates we get functions that
  - Work for many types `T`
  - Are as generic and abstract as the formal definition
  - Are a one-to-one translations of the abstract algorithm

Usage Causes Instantiation

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

Usage Causes Instantiation

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

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

Usage Causes Instantiation

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

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

Instantiation

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

Usage Causes Instantiation

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

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

Instantiation

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

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

Usage Causes Instantiation

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

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

Instantiation

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

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

Instantiation

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

Polymorphism

- **Definition**: Using many different types through the same interface

- What are the advantages?

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

Example: Generic Programming Process

- (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?
- Overlaps, similarities?

Example: Generic Programming Process

- (Simple) Family of algorithms: min, max
- Generic representation

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

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

- Minimal requirements, now?
- Overlaps, similarities?

Example: Generic Programming Process

- Possible implementation

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

- What are the requirements on `T`?

Example: Generic Programming Process

- Possible implementation

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

- What are the requirements on `T`?
  - Operator < , result convertible to bool
  - Copyable

Example: Generic Programming Process

- Possible implementation

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

- What are the requirements on `T`?
  - Operator < , result convertible to bool
  - Copyable, needed?

Example: Generic Programming Process

- Possible implementation

```
template <typename 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
  - -

So-called concept/named requirement

The problem of different types: manual solution

- What if we want to call `min(1, 3.141)`

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

- Now we need to specify the first argument (i.e., the return type) since it cannot be deduced

```
min<double>(1, 3.141);
min<int>(3, 4)
```

- More on this soon...

Templates in C++

- Templates
  - Function templates
  - Class templates (next week)
  - Variable templates (C++14)
- Introduced by the syntax
  - `template <...>`

Function templates

- Function templates
  - Type parameters: `typename` or `class` (most common case)
  - Non-type parameters (values, mostly integers)
  - Template-template parameters (passing a template as a template parameter)
- Function templates specialization
  - Concrete/special implementation for specific template parameters

```
// Specialization of min for std::string that returns the
// string smallest in length.
template <>
std::string const& min(std::string const& x, std::string const& y) {
  return (x.length() < y.length() ? x : y);
}
```

The primary template (i.e. the original template) would perform a lexicographical comparison.

  - Only full specialization
    - All template parameters need to be specified