Chapter 8. Templates

Programming Concepts in Scientific Computing

EPFL, Master class

October 16, 2024

Some more factorization

Some algorithms are not **type dependent**

Some more factorization

Some algorithms are not **type dependent**

```
int getMaximum(const int &a, const int &b) {
  if (a > b)
    return a;
  return b;
}
```

Some more factorization

Some algorithms are not **type dependent**

```
int getMaximum(const int &a, const int &b) {
  if (a > b)
    return a;
  return b;
}

double getMaximum(const double &a, const double &b) {
  if (a > b)
    return a;
  return b;
}
```

Some more factorization

Some algorithms are not **type dependent**

```
int getMaximum(const int &a, const int &b) {
  if (a > b)
    return a;
  return b;
}


double getMaximum(const double &a, const double &b) {
  if (a > b)
    return a;
  return b;
}
```

Can we use **types** as **parameters** ?

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Type matching !

```
  double a,b;
  double res_d = getMaximum(a,b);
```

Template Functions

```
template<typename T>
T getMaximum(const T & a, const T & b) {
  if (a > b)
    return a;
  return b;
}
```

Type matching !

```
  double a,b;
  double res_d = getMaximum(a,b);
```

or

```
  int c,d;
  int res_i = getMaximum(c,d);
```

Apply templates to **vector** class ?

Template Classes

Apply templates to **vector** class ?

Yes

Template Classes

```
template <typename T> class MyVector {
public:
  T &operator[](unsigned int dim) { return value[dim]; }

private:
  T value[3];
};
```

Template Classes

```
template <typename T> class MyVector {
public:
  T &operator[](unsigned int dim) { return value[dim]; }

private:
  T value[3];
};
```

Template Classes

```
template <typename T> class MyVector {
public:
  T &operator[](unsigned int dim) { return value[dim]; }

private:
  T value[3];
};
```

Template Classes

```
template <typename T> class MyVector {
public:
  T &operator[](unsigned int dim) { return value[dim]; }

private:
  T value[3];
};
```

Template Classes

```
template <typename T> class MyVector {
public:
  T &operator[](unsigned int dim) { return value[dim]; }

private:
  T value[3];
};
```

Template Classes

- MyVector is a templated class

Template Classes

- MyVector is a templated class
- Template parameters included in type definition

Template Classes

- MyVector is a templated class
- Template parameters included in type definition

```
MyVector<double> real_vector;
MyVector<int> integer_vector;
```

Template Classes

- MyVector is a templated class
- Template parameters included in type definition

```
MyVector<double> real_vector;
MyVector<int> integer_vector;
```

- Will be done during compilation

Template Classes

- MyVector is a templated class
- Template parameters included in type definition

```
MyVector<double> real_vector;
MyVector<int> integer_vector;
```

- Will be done during compilation
- Instanciated Template classes/functions: modified name (MyVectorInt, MyVectorDouble)

Template Classes

Template parameters can be:

- Other types (typename)
- Integers
- Enum values

Templates: Adding dimension to the vector type

```
template <typename T, int dim = 3> class MyVector {
...
private:
  T value[dim];
};
```

Templates: Adding dimension to the vector type

```
template <typename T, int dim = 3> class MyVector {
...
private:
  T value[dim];
};
```

Templates: Adding dimension to the vector type

```
template <typename T, int dim = 3> class MyVector {
...
private:
  T value[dim];
};
```

Using it:

```
MyVector<double> vector_3d;
MyVector<double, 2> vector_2d;
```

Type Matching: scalar product function

```
template <typename T, int dim>
T scalarProduct(const MyVector<T, dim> &v1, /
                const MyVector<T, dim> &v2  /
) {

  T res;
  for (int d = 0; d < dim; ++d)
    res += v1[d] * v2[d];
  return res;
}
```

Type Matching: scalar product function

```
template <typename T, int dim>
T scalarProduct(const MyVector<T, dim> &v1, /
                const MyVector<T, dim> &v2  /
) {

  T res;
  for (int d = 0; d < dim; ++d)
    res += v1[d] * v2[d];
  return res;
}
```

Using:

```
MyVector<double> vector1_3d;
MyVector<double> vector2_3d;

double res = scalarProduct(vector1_3d, vector2_3d);
```

Type matching: ostream « operator

```
template <typename T, int dim>
std::ostream &operator<<(std::ostream &stream, const MyVector<T,
  stream << "[";
  for (int d = 0; d < dim; ++d) {
    if (d != 0)
      stream << ", ";
    stream << vect[d];
  }
  stream << "]";
  return stream;
}
```

Template specialization

```
template <typename T> T getMaximum(const T &a, const T &b) {
  if (a > b)
    return a;
  return b;
}
```

Template specialization

```
template <typename T> T getMaximum(const T &a, const T &b) {
  if (a > b)
    return a;
  return b;
}
```

getMaximum working over vectors ?

Template specialization

```
template <typename T> T getMaximum(const T &a, const T &b) {
  if (a > b)
    return a;
  return b;
}
```

getMaximum working over vectors ?

```
MyVector<double> v1, v2;
MyVector<double> res = getMaximum(v1, v2);
```

Template specialization

Exceptions in type matching

Template specialization

Exceptions in type matching

```
template <>
MyVector<double, 3>
getMaximum<MyVector<double, 3>>(const MyVector<double, 3> &v1,
                                const MyVector<double, 3> &v2) {

  MyVector<double, 3> max;
  for (int i = 0; i < 3; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

Template specialization

Exceptions in type matching

```
template <>
MyVector<double, 3>
getMaximum<MyVector<double, 3>>(const MyVector<double, 3> &v1,
                                const MyVector<double, 3> &v2) {

  MyVector<double, 3> max;
  for (int i = 0; i < 3; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

Template specialization

Exceptions in type matching

```
template <>
MyVector<double, 3>
getMaximum<MyVector<double, 3>>(const MyV
                                const MyVector<doub
  MyVector<double, 3> max;
  for (int i = 0; i < 3; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

Template specialization

Exceptions in type matching

```
template <>
MyVector<double, 3>
getMaximum<MyVector<double, 3>>(const MyVector<double, 3> &v1,
                                const MyVector<double, 3> &v2) {

  MyVector<double, 3> max;
  for (int i = 0; i < 3; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

Template specialization

Exceptions in type matching

```
template <>
MyVector<double, 3>
getMaximum<MyVector<double, 3>>(const MyVector<double, 3> &v1,
                                const MyVector<double, 3> &v2) {

  MyVector<double, 3> max;
  for (int i = 0; i < 3; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

```
MyVector<double> v1, v2;
MyVector<double> res = getMaximum(v1, v2);
```

Templates specialization

Can we code-factor more ?

Templates specialization

Can we code-factor more ?

```
template <typename T, int dim>
MyVector<T, dim> getMaximum(const MyVector<T, dim> &v1,
                            const MyVector<T, dim> &v2) {

  MyVector<T, dim> max;
  for (int i = 0; i < dim; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

Templates specialization

Can we code-factor more ?

```
template <typename T, int dim>
MyVector<T, dim> getMaximum(const MyVector<T, dim> &v1,
                            const MyVector<T, dim> &v2) {

  MyVector<T, dim> max;
  for (int i = 0; i < dim; ++i) {
    max[i] = getMaximum(v1[i], v2[i]);
  }
  return max;
}
```

This is overloading

Template meta programming

```
int arithmetic(int i) {
  if (i == 1)
    return 1;
  return i + arithmetic(i - 1);
}
```

Template meta programming

```
int arithmetic(int i) {
  if (i == 1)
    return 1;
  return i + arithmetic(i - 1);
}
```

Let's call the function

```
  int a = 5;
  std::cout << arithmetic(a);
```

Template meta programming

```
template <int i> int arithmetic_template() {
  return i + arithmetic_template<i - 1>();
}

template <> int arithmetic_template<1>() {
  // ends recursion
  return 1;
}
```

Let's call the function

```
std::cout << arithmetic_template<5>();
std::cout << std::endl;
```

Template meta programming

Can we do this ?

```
int a = 5;
std::cout << arithmetic_template<a>();
std::cout << std::endl;
```

And why ?

Variadic template meta programming

```
template <typename T> T adder(T v) { return v; }

template <typename T, typename... Args>
// template ...Args is a variadic template
T adder(T first, Args... args) {
  return first + adder(args...);
}
```

Using it:

```
std::cout << adder(1, 2, 3, 4);
std::cout << adder(1., 2, 3., 4);
std::cout << adder(std::string("a"), std::string("b"));
```

Variadic template meta programming

```
template <typename T, typename... Args> // variadic
T adder(T first, Args... args) {
  T ret = first;
  ((ret += args), ...);
  return ret;
}
```

Using it:

```
  std::cout << adder(1., 2, 3., 4);
  std::cout << adder(std::string("a"), std::string("b"));
  std::cout << std::endl;
```

Function returning a pair

```
template <typename T1, typename T2> struct pair {
  T1 obj1;
  T2 obj2;
};

pair<int, int> foo() {
  // return two integers together
  return pair<int, int>{2, 3};
}
```

Using it:

```
  pair<int, int> p = foo();
  // fetching the integers
  int a = p.obj1;
  int b = p.obj2;
}
```

std::pair

```
std::pair<int, int> foo() {
  // return two integers together
  return std::make_pair(2, 3);
}
```

Using it:

```
std::pair<int, int> p = foo();
// fetching the integers
int a = std::get<0>(p);
int b = std::get<1>(p);
```

std::tuple

```
std::tuple<int, int, double> foo() {
  // return a tuple with 3 entries
  return std::make_tuple(2, 3, 3.14);
}
```

Using it:

```
std::tuple<int, int, double> p = foo();
// fetching the integers
int a = std::get<0>(p);
int b = std::get<1>(p);
double c = std::get<2>(p);
```

auto magic

```cpp
std::tuple<int, int, double> foo() {
  // return a tuple with 3 entries
  return std::make_tuple(2, 3, 3.14);
}
```

Using it:

```cpp
  // magical auto keyword
  auto [a, b, c] = foo();
```

Take away message

- **Templating** is a mechanism to **substitute** parameters during compilation
- **Template Classes** create types with parameters included in the type name
- **Template parameters** are other types, integers, enums
- Compiler does **type matching** to deduce template parameters
- **Specializations** allow exceptions
- **Overloading** or **Template** ? no rule: need programming experience
- **Meta-programming** allows calculation at compilation
- **Variadic templates** allow flexible expressions