High Level Parallel Programming

C++ Templates

Alessandro Savino and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Templates

- In problem solving data structures often include different types of data
- In C++ it is possible to make code functionality independent from the specific type T
  - For example we would like to write function
    - swap(T& a, T& b)

    Which is independent from the type T

Templates

- The same functionality should be made available for all suitable types T
  - Templates reply to the following questions
    - How to avoid massive code duplication?
    - How to account for user-defined types?

Function templates

- Function template
  - It is possible to write a function that takes arguments of arbitrary types
  - T is a template parameter, which must be a type

```
template <typename T>

int compare (const T& v1, const T& v2) {
  if (v1 < v2) { return -1; }
  if (v2 < v1) { return 1; }
  return 0;
}
```

T a label which will be substituted by the correct object type

Same implementation we would have for integer or float types

The compiler will generate a new version of the function (using polymorphism) when this is required

Function templates

The template is **never** processed at tun-time

- When you call a template function, the compiler
  - Deduces what types to use instead of the template parameters
  - Instantiates (“generates”) a function with the correct types

T is an int
The compiler instantiates
int compare(const int&, const int&)

```
void f() {
  cout << compare(1, 0) << endl;
  string s1 = "hello";
  string s2 = "world";
  cout << compare(s1, s2) << endl;
}
```

T is a string
the compiler instantiates
int compare(const string&, const string&)

Function templates

- A template can be used with “personal” types
  - Templates usually put some requirements on the argument types
  - If these requirements are not met the instantiation will fail

If class Rectangle implements operator<, everything is ok otherwise we receive an error at compilation time

```
void f() {
  Rectangle r1(2,3),r2(3,4.5);
  cout << compare(r1, r2) << endl;
}
```

Drawbacks

- Requirements on the type T
  - The previous compare template requires that objects T must be compared with <
  - Now < and ==

```
template <typename T>

int compare (const T& v1, const T& v2) {
  if (v1 < v2) { return -1; }
  if (v2 <= v1) { return 1; }
  return 0;
}
```

This implementation puts more requirements on T
This isn't good... why?

The arguments are passed by value, so it has to be possible to copy objects of T
Objects of T must implement comparisons with < and ==

Explicit instantiation

- In every case you want to be sure the compiler is able to understand to which type T is referring to
  - It is possible to set the type upon call

Type specification
Useless in this case

```
void f() {
   cout << compare<int>(1, 0) << endl;
   string s1 = "hello";
   string s2 = "world";
   cout << compare<string>(s1, s2) << endl;
}
```

Type specification
Useless in this case

Function templates

- This is a function template that compares two values of the different type
  - The differentiation **must** make sense
  - The operation **must** be feasible

```
template <typename T, typename Z>
int compare(const T& v1, const Z& v2) {
  if (v1 < v2) { return -1; }
  if (v2 < v1) { return 1; }
  return 0;
}
```

Return as template

- It is possible to resort to template also as return value for a function
  - Notice that in this case you have only one type
  - Return values and the type T must match

```
template <typename T>
T compare(const T& v1, const T& v2) {
  if (v1 < v2) { return -1; }
  if (v2 < v1) { return 1; }
  return 0;
}
```

Return as template

- It is possible to resort to template also as return value for a function
  - You can also resort to different types to distinguish parameters and return
  - Again, return values and the type R must match

```
template <typename T, typename R>
R compare(const T& v1, const T& v2) {
  if (v1 < v2) { return -1; }
  if (v2 < v1) { return 1; }
  return 0;
}
```

Class templates

- In a similar way, a class can be programmed to deal with a generic data type
  - Definition is like functions

```
template <typename T>
class Rectangle {
  public:
    Rectangle()  { initData(0,0); };
    Rectangle(const T &w, const T &l){
       initData(w, l); };
    ...
  private:
    T m_width, m_length;
    ...
}
```

Class templates

- Class members implementations should be inline

```
template <typename T>
class Rectangle {
  public:
    Rectangle()  { initData(0,0); };
    Rectangle(const T &w, const T &l){
       initData(w, l); };
    ...
  private:
    T m_width, m_length;
    ...
}
```

Class templates

- It is possible to write the definition of a class member function outside the class definition
  - The template information must be repeated

File with extension hpp (or h)

```
template <typename T>
class Rectangle {
  public:
    ...
    void setW(const T &w);
    T getW() const;
    ...
};
```

The compiler will instantiate two separate classes, one for each type
These classes will share nothing

File with extension cpp

```
template <typename T>
void Rectangle<T>::setW(const T &w) {...}
template <typename T>
T Rectangle<T>::getW() const { ... }
```

Class templates

- With class templates, the compiler cannot deduce template parameter types from the class instantiation
  - The types must be explicitly supplied when an object is created
  - The compiler never guesses the type

Without the keyword <double> the compiler cannot argue what to do

```
int main() {
  Rectangle<double> r1, r2(12.4, 5), r3;
  ...
```

<double> is mandatory

Proper constructors required

Where to define templates

- Templates do not have a proper implementation (the compiler is going to take care of it)
  - There is no .cpp to compile
  - Everything should be inserted in a .hpp/.h file
    - The .hpp file can be included, when needed, as any other kind of .h file