Templates

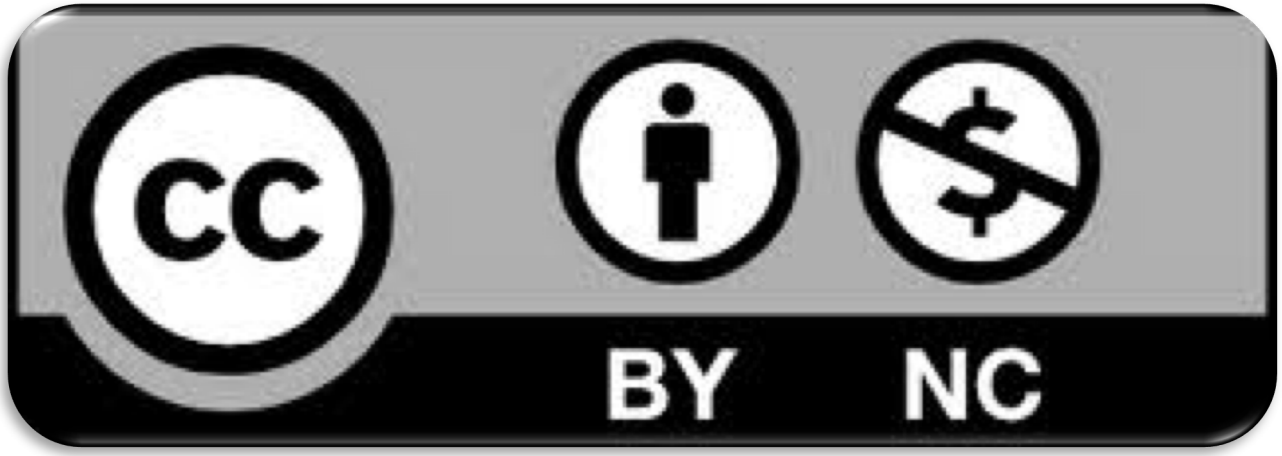
Alessandro SAVINO
Politecnico di Torino (Italy)

alessandro.savino@polito.it

www.testgroup.polito.it

License Information

This work is licensed under the Creative Commons BY-NC License



To view a copy of the license, visit:
http://creativecommons.org/licenses/by-nc/3.0/legalcode

Disclaimer

- **We disclaim any warranties or representations as to the accuracy or completeness of this material.**
- **Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.**
- **Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.**

Goal

- **This lecture presents a global overview of the basic concepts behind the C++ templates.**

Prerequisites

- **Basic knowledge of C++ programming language**

Homework

- **None**

Outline

- **Template Definition**
- **Template Basic Concepts**
- **Function Template**
- **Class Template**

Outline

- **Template Definition**
- **Template Basic Concepts**
- **Function Template**
- **Class Template**

Rel. 06/05/2019

Template Definition

- **Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types.**

Outline

- Template Definition
- Template Basic Concepts
- Function Template
- Class Template

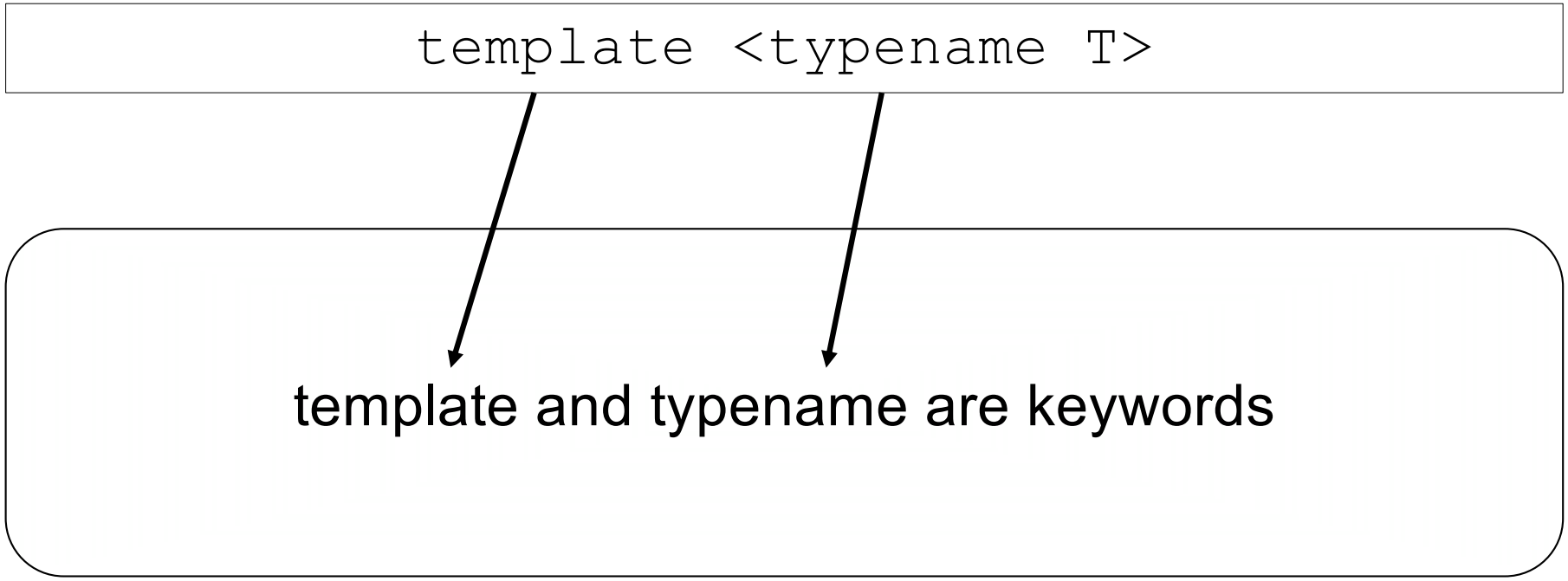
Template Basic Concepts

- **The basic code for the definition of a “template” is:**

```
template <typename T>
```

Template Basic Concepts

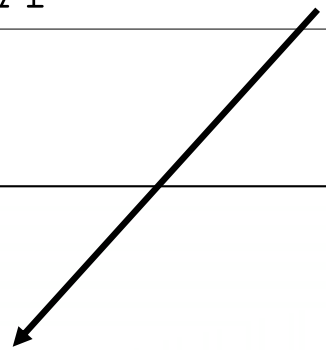
- **The basic code for the definition of a “template” is:**



Template Basic Concepts

- **The basic code for the definition of a “template” is:**

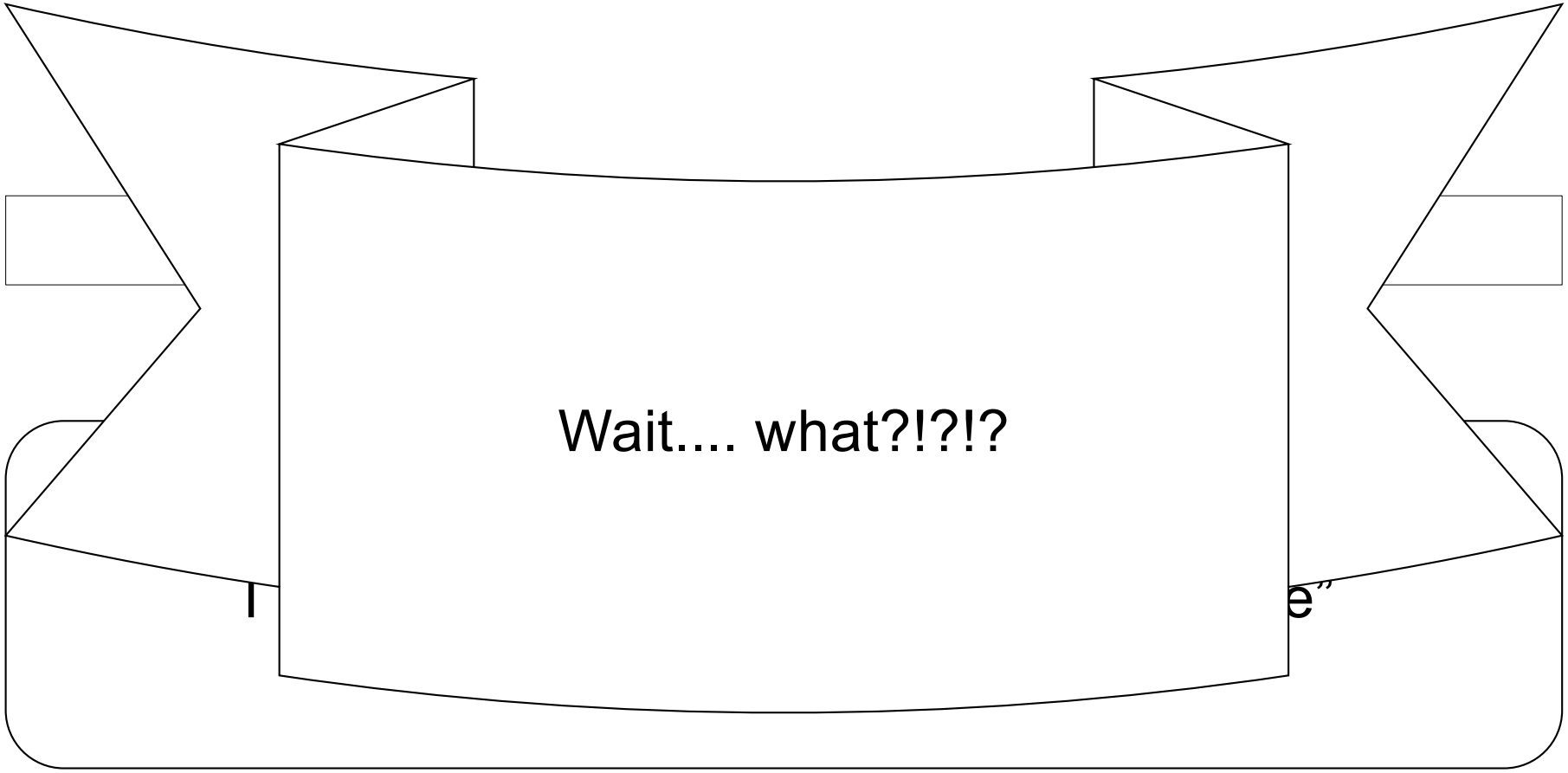
```
template <typename T>
```



T will be the label to refer to the “template”

Template Basic Concepts

- **The basic code for the definition of a “template” is:**



Template Basic Concepts

- **The template keyword usage is ever within a more complex definition.**
  - **Function template**
  - **Class template**

Template Basic Concepts

- **The template keyword usage is ever within a more complex definition.**
  - **Function template**
  - **Class template**

```
template <typename T>
int compare(const T& v1, const T& v2)
```


T is now a template type for a function template

Rel. 06/05/2019

Outline

- Template Definition
- Template Basic Concepts
- Function Template
- Class Template

Function Template

- **Function template: you can write a function that takes arguments of arbitrary types.**

```
template <typename T>
int compare(const T& v1, const T& v2)
```

Function Template

- **This is a function template that compares two values of the same type:**

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

Function Template

- **This is a function template that compares two values of the same type:**
  - **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;
}
```

Function Template

- **When you call a template function, the compiler deduces what types to use instead of the template parameters and instantiates ("writes") a function with the correct types.**

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

Function Template

- **When you call a template function, the compiler deduces what types to use instead of the template parameters and instantiates ("writes") a function with the correct types.**

```
void f() {
    cout << compare(1, 0) << endl;
    string s1
                ello";
```

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

Function Template

- **When you call a template function, the compiler deduces what types to use instead of the template parameters and instantiates ("writes") a function**

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

```
    string s1 = "hello";
    string s2 = "world";
    cout << compare(s1, s2) << endl;
}
```

Function Template

- **A template usually puts some requirements on the argument types. If these requirements are not met the instantiation will fail.**

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

Function Template

- **A template usually puts some requirements on the argument types. If these requirements are not met the instantiation will fail.**

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

If class Rectangle implements operator<, everything is ok.

Function Template Drawbacks

- **The only requirement placed on the type T in the compare template is that objects of T can be compared with <.**

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

Function Template Drawbacks

- **Th**

The following implementation puts more requirements on T and isn't good... why?

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

Function Template Drawbacks

- **Th**

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

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


Explicit instantiation

- **In every case you want to be sure the compiler is able to understand which type you are referring to, you can set it upon call:**

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

Explicit instantiation

- **In c**

This is true also when you get an error from the compiler (i.e., because the types of the arguments don't match)

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

Function Template

- **This is a function template that compares two values of the different type:**

```
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;
}
```

Function Template

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

```
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;
}
```

Function Template

- **Thi**

In this case, the comparison has to be feasible (e.g., int vs double?)

```
template <typenam 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**

```
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**
  - **Notice that in this case you have only one type so they 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**
  - **Notice that in this case you have only one type so they must match**

```
template <typename T>
T compare(const T& v1, const T& v2) {
}
```

It must make sense... otherwise you need to decouple (at least) the parameter type from the return type...

Return as Template

- **It is possible to resort to template also as return value for a function**

```
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;
}
```

Outline

- **Template Definition**
- **Template Basic Concepts**
- **Function Template**
- **Class Template**

Class Templates

- **In a similar way, a class can be programmed to deal with a generic data type**

```
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

- **In a similar way, a class can be programmed to deal with a generic data type**

```
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

- **In a similar way, a class can be programmed to deal with a generic data type**

```
template <typename T>
class Rectangle {
  public:
    Rectangl        { initData(0,0); };
    Rectar          st T &w, const T &l){
        i             l); };
  ...
```

Notice the same definition methodology for functions and classes...

Class Templates

- **In a similar way, a class can be programmed to deal with a generic data type**

... and the encapsulated data definition...

```
template <typename T>
    Rec              nst T &w, const T &l){
                 ta(w, l); };
  ...
private:
  T m_width, m_length;
...
```

Class Templates

- **In a similar way, a class can be programmed to deal with a generic data type**

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

.. and the way T is managed across methods definition.

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, but then the template information must be repeated**

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

Class Templates

- **It is possible to write the definition of a class member function outside the class definition, but then the template information must be repeated**

Take care of:

1. properly respect the syntax: template info + return type + class name + :: + member function
2. class name must include the typename(s)

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

Class Templates

- **What happen if a need to return a class template object?**

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

Class Templates

- **What happen if a need to return a class template object?**

No deviations from what you've learned so far...

```
    T g                 c;
    ...
};
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.**

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

Where to put templates

- **Since template do not have a proper implementation (the compiler is going to take care of it), you do not have a .cpp to be compiled!**
- **You should put everything in a .hpp file to be included when needed as any other kind of .h file.**


Малые Автюхи, Калинковичский район, Республики Беларусь