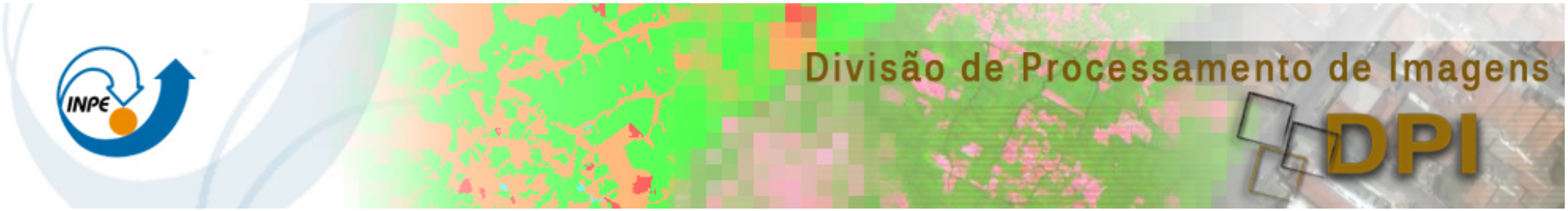

CAP- 390-1 Fundamentos de Programação Estruturada – Aula 9
STL

Lubia Vinhas

Templates in C++ - Syntax

< Content>

- Where Content can be:
  - class T / typename T
  - A data type, which maps to T
  - An integral specification
  - An integral constant/pointer/reference which maps to specification mentioned above
- Example:

```
template<yypename T>
void PrintTwice(T data)
{ cout<<"Twice: " << data * 2 << endl; }
```

Templates in C++ - Syntax

```
template<typename T>
void PrintTwice(T data)
{ cout<<"Twice: " << data * 2 << endl; }

{
   PrintTwice(4);
   PrintTwice(34.87);
}
```

- Two instances of PrintTwice will be generated by the compiler
  - No copy-pasting
- Effectively, for N number of data types, N instances of the same function will be created

Templates in C++ - Syntax

```
template<typename T>
T Twice(Tdata)
{
  return data * 2;
}
```

```
{
cout << Twice(10);
cout << Twice(3.14);
cout << Twice( Twice(55) );
}
```

- If a template function is instantiated for a particular data-type, compiler would re-use the same function' instance

Templates in C++ - Syntax

```
template<typename T>
T Twice(Tdata)
{
  return data * 2;
}
```

```
{
cout << Twice(10);
cout << Twice(3.14);
cout << Twice( Twice(55) );
}
```

- If a template function is instantiated for a particular data-type, compiler would re-use the same function' instance

Templates in C++

```
template<class T>
T Add(T n1, T n2)
{
   T result;
   result = n1 + n2;

   return result;
}
```

- The compiler will work in two phases:
  - once for basic syntax checks;
  - for each instantiation, compilation against the template data-types

  - T is having a default constructor (so that T result; is valid)
  - T supports the usage of operator + (so that n1+n2 is valid)
  - T has an accessible copy/move-constructor (so that return statement succeeds)

Templates in C++

```
template<class T>
double GetAverage(T tArray[],
                  int nElements)
{
   T tSum = T(); // tSum = 0

   for (int nIndex = 0;
         nIndex < nElements; ++nIndex)
   {
      tSum += tArray[nIndex];
   }

   // Whatever type of T is,
   // convert to double
   return double(tSum) / nElements;
}
```

```
int main()
{
   int  IntArray[5] = {100, 200, 400};
   float FloatArray[3] = { 1.55f, 5.44f};

   cout << GetAverage(IntArray, 3);
   cout << GetAverage(FloatArray, 2);
}
```

Templates in C++

```
template<class T>
GetAverage(T* tArray, int nElements){}
```

```
template<class T>
void TwiceIt(T& tData)
{
   tData *= 2;
   // tData = tData + tData;
}
```

- Try to identify the type deduction from this signatures

Templates in C++

```
template<class T1, class T2>
void PrintNumbers(const T1& t1Data,
                  const T2& t2Data)
{
    cout << "First value:" << t1Data;
    cout << "Second value:" << t2Data;
}

PrintNumbers(10, 100);    // int, int
PrintNumbers(14, 14.5);   // int, double
PrintNumbers(59.66, 150); // double, int
```

- Multiple types with function templates
- The order matters
- Compile will not perform automatic conversion

Templates in C++

```
template<class T1, class T2>
void PrintNumbers(
  const T1& t1Data,
   const T2& t2Data)
{}
```

- Explicit template argument specification
- You want only specific type to be passed, and not let the compiler intelligently deduce one or more template argument types solely by the actual arguments

Class templates

```
template<class T>
class Item
{
   T Data;
public:
   Item() : Data( T() )
   {}

   void SetData(T nValue)
   {
      Data = nValue;
   }

  T GetData() const
   {
     return Data;
   }

   void PrintData()
   {
      cout << Data;
   }
};

Item<int> item1;
item1.SetData(120);
item1.PrintData();

Item<float> item2;
float n = item2.GetData()
```

Examples from: http://www.codeproject.com/

Class templates

```
template<class T>
class Item
{
   T Data;
public:
   Item() : Data( T() )
   {}

   void SetData(T nValue)
   {
      Data = nValue;
   }

  T GetData() const
   {
     return Data;
   }

   void PrintData()
   {
      cout << Data;
   }
};

Item<int> item1;
item1.SetData(120);
item1.PrintData();

Item<float> item2;
float n = item2.GetData()
```

Examples from: http://www.codeproject.com/

Practice

- Write a templated version of a Pair data structure and experiment using it

The Standard Template Library (STL)

- A set of common used data structures and algorithms parameterized with types
- Some useful ones include
  - vector
  - map
  - stack, queue, priority_queue

STL - vectors

- an array with automatic resizing

```
std::vector<t> v;
std::vector<int> v2(200)
std::vector<T> v3(100)

v2.push_back(1);
v2.push_back(2);
v2.pop_back();

if (v2.empty()) std::cout << “Vector is empty!\n”;
```

Traversing a vector using an iterator

```
std::vector<int>::iterator it;

for (it=v2.begin(); it != v2.end(); ++it)
{
    std::cout *it;
}
```

Many other functions… and containers… and algorithms!

http://www.cppreference.com/wiki/stl/vector/start

http://en.cppreference.com/w/File:container-library-overview-2012-12-27.pdf