Lecture 12

C++ Programming

Arne Kutzner
Hanyang University / Seoul Korea

C++
Templates

Generic Programming

- We already know linked lists as data structures:
  - ```
    class ListOfInt {
     public:
           int data;
           ListOfInt * next;
    };
    ```
  - ```
    class ListOfDouble {
     public:
           double data;
           ListOfDouble * next;
    };
    ```

Apart from the datatypes for list-items (int and double) both definitions are equal

- So far we have to create for every datatype a separate **`ListOf`** ***`[Datatype]`*** definition

Reuse of Code

- This redundancy is inconvenient and opposes to the principle of “reuse of code”
- Possible solution:
  - We wrap atomic data using special objects and use polymorphism. (This approach is used in older editions of Java)
  - However, this results in well known problems regarding:
    - Type-safeness
    - Performance issues (E.g. we have to check during runtime whether we get data of some required datatype)

Class Templates

- **Templates** offer a convenient way to overcome this problem:

```
template <typename TYPE>
class ListOf {
   public:
      TYPE data;
      ListOf<TYPE> * next;
};
```

Example is in file `linkedList-generic.cpp`

Template Declaration

- General form of Template declaration:

```
template <class identifier>
    class_definition;
```

  or

```
template <typename identifier>
    class_definition;
```

- Both forms are identical
- In the class definition **`identifier`** is some form of placeholder that can be used at positions, where we expect some type specification

Multiple Type Identifiers

- A template declaration can have more than one argument
- For multiple arguments we use some comma separated list where each type-argument is introduced by the keyword **class**

```
template <class identifier_1,
class identier_2, …>
   class_definition;
```

Instantiation of Templates

- In order to “instantiate” some class template use the following syntax:
  `class_name<type_list>`
- **`type_list`** is some comma separated list of type names, e.g.
  `<int, int, double>`
- The number of type names has to match the number of arguments in the head of the template declaration

Nested Template usage

- In the context of template instantiations you can use the template syntax in nested form.
  Example:
  `ListOf<ListOf<int>*> * ptr;`

Function Templates

- You can also specify templates for creating generic functions. Example:

TP can be used like a normal type in the function declaration

```
template <class TP>
TP max(TP d1, TP d2)
{ return (d1>d2) ? d1 : d2; }
```

Type Inference Mechanism

- With function templates C++ can infer (automatically elaborate) the type of the template parameter by the type of the arguments.
  Examples: **integer literals**
  - `max(1, 2)` is identical to `max<int>(1, 2)`
  - `max(1.0, 2.0)` is identical to `max<double>(1.0, 2.0)`
  - `max('A', 'B')` is identical to `max<char>('A', 'B')`

Template specialization

- For some call **`max("cat","dog")`** we get the instantiation
```
const char* max(const char* d1, const char* d2) {
    return (d1>d2) ? d1 : d2;
}
```
- The syntax is okay, but the semantic is not as intended
- How to fix?
  – We need to be able to create a special version of the function to be called in the case of the type **`char*`**
  – Such special version is called a "**specialization**"

Template Specialization (cont.)

- For our example the specialization looks as follows:

The empty parameter list indicates that the following code is a specialization of some template

```
template <>
const char* max(const char* d1, const
char* d2) {
    return (strcmp(d1,d2)>0) ? d1 : d2;
}
```

Integral Values as Template Parameter

- Besides types we can use integral values as parameter of templates. Example:

```
template <int NUM>
   class_definition;
```

- Like constants in the class definition

Example of Application of Integral Parameter

- ```
  template <long N>
  class Factorial {
  public:
      long Value(void) {
          return N * fn_1.Value();
      }
  private:
      Factorial<N-1> fn_1;
  };
  ```

  Integer arithmetic in the context of integral parameter is possible

  specialization for the factorial of 0

  ```
  template <>
  class Factorial<0> {
  public:
      long Value(void) { return 1; }
  };
  ```
  ```

Example of Application of Integral Parameter (cont.)

- Main-function that calls our weird template:

```
int main(void) {
   Factorial<15> f;
   cout << f.Value() << endl;
}
```

How to design a "templated" class?

- Design an ordinary class which does the job for some specific type (e.g. **`ListOfInt`**)...
  - ...then elaborate the positions of the specific type...
  - ...and replace the type name with some generic identifier (e.g. **`TP`**)...
  - ...and finally place the keyword **`template`** together the type parameter (e.g. **`TP`**) in front.