Software↵
Languages.Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Software↵
Languages.Lab

Chapter 5 - User-Defined Type Constructors

Structuur van Computerprogramma's 2

Templates

Why Templates?

Suppose we have written a function:

```
void sortints(int a[]);
```

and now we need a function:

```
void sortstrings(std::string a[]);
```

What can we do?

Most of **`sortstrings`** will be duplicated from **`sortints`**
(only the type of the things we compare/move will be different)

Is it possible to write one function such as:

```
void sort(T a[]);
```

which will work for **any** type `T` such as:

```
std::string sa[]; sort(sa);
int ia[]; sort(ia);
```

Template Functions (I)

```
template<TemplateParameterDeclarationList>
ReturnType
FunctionTemplateName(ParameterDeclarationList){
  StatementList
}
```

```
#ifndef BUBBLE_SORT_H
#define BUBBLE_SORT_H

#include <iostream>
#include <string>

template<typename T>
void swap(T& x, T& y) {
   T tmp(x);
   x = y;
   y = tmp;
}

//...
```

A (**function** or **class**) **template** is a type constructor which, given concrete parameter types, returns, at compile-time, a concrete (function or class) type

Template Functions (2)

```
//...

template<typename T>
void bubble_sort(T a[], unsigned int total_size) {
  for (unsigned int size = total_size - 1; (size > 0); --size)
    // find largest element in 0..size range
    // and store it at size
    for (unsigned int i = 0; (i < size); ++i)
      if (a[i + 1] < a[i])
        swap(a[i + 1], a[i]);
}
#endif
```

Notice how templates introduce **variation points** in your implementation (**software variability**) and that they facilitate **polymorphism**

What are the (hidden) requirements on `T` **?**

Instantiating Template Function for type `std::string`

```
FunctionTemplateName<TemplateParameterList>
```

```
#include "bubble_sort.h"

int main(unsigned int argc, char* argv[]) {
   unsigned int size = argc - 1;
   std::string *args = new std::string[size];

   std::cerr << "size=" << size << std::endl;

   for (unsigned int i = 0; (i < size); ++i)
      args[i] = std::string(argv[i + 1]);

   bubble_sort<std::string>(args, size);

   for (unsigned int i = 0; (i < size); ++i)
      std::cerr << "args[" << i << "]=" << args[i] << std::endl;
}
```

Causes the compiler to create a **template instance** where T is bound to the type `std::string`

Instantiating Template Function for type `int`

```
#include "bubble_sort.h"
#include <stdlib.h> // for atoi()

int main(unsigned int argc, char* argv[]) {
  unsigned int size = argc - 1;
  std::cerr << "size=" << size << std::endl;

  // now args[] is an int array
  int *args = new int[size];

  for (unsigned int i = 0; (i < size); ++i)
    args[i] = atoi(argv[i + 1]); // why i+1?

 // use same bubble_sort, now instantiated for int
  bubble_sort<int> (args, size);// explicit instantiation for int
  bubble_sort(args, size);       // implicit - compiler deduces T
  for (unsigned int i = 0; (i < size); ++i)
    std::cerr << "args[" << i << "]=" << args[i] << std::endl;
}
```

Only use implicit instantiation if it is clear to which type it will be matched !

Overloading Template Functions

```
template<typename T>
T                        // general case
maximum(T x1, T x2) {
   return (x1 > x2 ? x1 : x2);
}

template<typename U>
U*                       // compare what is pointed to, not the pointers
maximum(U* p1, U* p2) {
   return (*p1 > *p2 ? p1 : p2);
}

const char*              // special case for C strings
maximum(const char* s1, const char* s2) {
   return (strcmp(s1, s2) > 0 ? s1 : s2);
}

double d1(1.23);
double d2(4.5);
maximum(d1, d2);
maximum(&d1, &d2);
maximum("abc", "abcd");
```

Overloading and Specialization

Essentially the same rule as before (i.e. use best match):

- To resolve an overloaded function call, more specialized functions (or function templates) that better match the actual call's parameters are to be preferred

In a readable program, a call's resolution should be clear from this principle alone

Template Classes

Template classes are type constructors
a.k.a. **parameterized types**

```
#ifndef ARRAY_H
#define ARRAY_H
#include <assert.h>
// safe arrays: subscripts are checked
template<typename T>
class Array {
public:
   Array(unsigned int size);          // gang of three: custom ctor
   Array(const Array&);               // gang of three: custom cctor
   ~Array();                          // gang of three: custom dtor

   unsigned int size() const;

   // overloaded operator[], will check legality of index
   T& operator[](unsigned int i);     // why two versions of [] ?
   const T& operator[](unsigned int i) const;
private:
   T* data_;                          // a regular unchecked T array
   unsigned int size_;                // size of data_ array
   Array& operator=(const Array&);    // we forbid assignment !
};
```

Within the class and member definition you refer to **`Array`** and not **`Array<T>`** !

Array Implementation: Constructors

```
// constructor
template<typename T>
Array<T>::Array(unsigned int size) :
  data_(new T[size]), size_(size) { }
```

Indicates that this is a **template function**

Indicates that it is a member function of a **template class**

```
// copy constructor
template<typename T>
Array<T>::Array(const Array& a) :
   data_(new T[a.size()]), size_(a.size()) {
   for (unsigned int i = 0; (i < size_); ++i)
      data_[i] = a[i];
}
```

Within the class and member definition you refer to `Array` and not `Array<T>` !

```
// destructor
template<typename T>
Array<T>::~Array() {
   delete[] data_;
}
```

Array Implementation: Inspectors

```
template<typename T>
unsigned int Array<T>::size() const {
   return size_;
}
```

```
// supports e.g. x = a[k]
// but not a[k] = 5 if a is an array of constants
template<typename T>
const T&
Array<T>::operator[](unsigned int i) const {
   assert(i < size_); // abort if bad index, use exceptions!
   return data_[i];
}
```

Array Implementation: Indexing for Assignment

```cpp
// non-const operator[]: return non-const reference
// can be used as in a[k] = ..

template<typename T>
T&
Array<T>::operator[](unsigned int i) {
   assert(i < size_);
   return data_[i];
}
#endif
```

Array: Usage Example

```
#include <string>
#include "array.h"

Array<std::string> f(Array<std::string> a) {
   // to test copy-ctor
   return a;
}

int main(int argc, char *argv[]) {
   unsigned int size = argc - 1;
   Array<std::string> a(size);

   for (unsigned int i = 0; i < size; ++i)
      a[i] = std::string(argv[i + 1]);

   for (unsigned int i = 0; i < size; ++i)
      std::cout << f(a)[i] << std::endl;
}
```

Creates an **`Array`** of **`size`** elements of type **`std::string`**
Note that the initialization of **`a`** refers to the number of elements (similar to `[ ]`) and not to the value to be stored !

You could create a multidimensional array with: **`Array<Array<int> > a(10);`**
`a` is an **`Array`** of **`10`** elements, where each element is an **`Array`** of **`10`** **`int`**'s

Note the space between '`> >`', this is to avoid confusion by the compiler with '`>>`'!
Access it with **`a[2][4];`** similar to `(a.operator[](2)).operator[](4);`

Software Languages Lab