Templates

Templates

- Allow to abstract the code from the type used
- Suppose we want to write code which would be identical for type int and type double, except that in one case all variables would be of type int, in the other case all of type double
- We write the code with respect to a generic type T, which acts merely as a placeholder
- T is replaced at compile time by the appropriate type
- Both functions and classes can be defined with templates parameters

Template Functions

(see: http://www.cplusplus.com/doc/tutorial/functions2/)

Template Functions

- Suppose we want to implement a function which computes the cube of a number
- We want to define the function for the types float, double and int
- We can do that using overloading, but it is a waste of code, which is always the same, except for the type

Templates

- Using overloads, we need to repeat the same code

```
int cube(int x) { return x*x*x; }
float cube(float x) { return x*x*x; }
double cube(double x) { return x*x*x; }
```

- Using templates, we write the code only once

```
template <typename MyType>
MyType cube(MyType x) { return x*x*x; }
```

- **template** and **typename** are keywords
- The keyword **class** can be used instead of **typename** (it is equivalent)
- **MyType** is an identifier acting as a placeholder for a type

std::swap

- **swap** is a simple template function provided by the Standard Template Library (STL)
- It swap two values of the same type
- It requires **#include <algorithm>**
- See:

  http://en.cppreference.com/w/cpp/algorithm/swap

std::swap - Implementation

```
template <typename T>
void swap(T& a, T& b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

usage example – std::swap

```
#include <algorithm>
#include <iostream>
int main()
{
    int a = 5, b = 3;
    // before
    std::cout << a << ' ' << b << '\n';
    std::swap(a, b); // the type is auto-inferred
    // after
    std::cout << a << ' ' << b << '\n';
}
```

Problem

• Create an integer array A with entries 0, 1, ..., 99 in this order
• Reverse the order of the entries of A by executing 50 swaps (A[0] should be swapped with A[99], A[1] with A[98] etc., etc.).
• Print all entries of A to the screen to check if the swapping worked.

• Hint: A[0]=A[99]; A[99]=A[0]; … will NOT work. (why?)

Problem

Write a program that generates a random permutation of 0, 1, ..., 99. One possible method:

- Create an integer array Perm of length 100 with entries 0,1, ..., 99 in this order.
- For each i=99,98,...,1 do the following:
  - Determine a random index j in the range 0,...,i. Hint: Use j=rand()%(i+1);
  - Swap the entries Perm[i] and Perm[j]

std::max

- **max** is a simple template function provided by the Standard Template Library (STL)
- It returns the maximum of two values of the same type
- It requires **#include <algorithm>**
- See:

  http://en.cppreference.com/w/cpp/algorithm/max

std::max - Implementation

```
template<class T>
const T& max(const T& a, const T& b)
{
    return (a < b) ? b : a;
}
```

Usage example – std::max

```
#include <algorithm>
#include <iostream>
int main()
{
    int a = 5, b = 3;
    double c = 3.4;
    // the type is auto-inferred
    std::cout << std::max(a, b) << '\n';
    // the type must be specified explicitly
    std::cout << std::max<double>(a, c) << '\n';
}
```

Problem

Let Perm[i], i=0,...,n-1, be a permutation of 0,1,...,n-1. We say that Perm has a fixed point if there is an index i with Perm[i]=i.

Write a program that generates 10,000 random permutations of 0, 1, ..., 99, and determines how many of these permutations have no fixed point.

To determine if there are no fixed point, write a template function

Template Matching

```
template <typename T>
T cube(T x) { return x*x*x; }
```

- This code is valid for any generic type T which supports the operator *
- When it is compiled, the syntax is checked, but no binary code generation happens: because it is applicable to any type, it should generate a large number of variations
- Only the variations actually used are generated (i.e. **instantiated**)

Example - Template Instantiation

```
template <typename T>
T cube(T x) { return x*x*x; }

int main()
{
  int x = 2;
  double y = 3.0;
  cout << cube(x) << endl << cube(y) << endl;
}
```

- Here binary code for *cube* is instantiated only for T=int and T=double, which are the only implementations of the function used

Templates Matching

```
int cube(int x) { return x*x*x; }

template <typename MyType>
MyType cube(MyType x) { return x*x*x; }

int main()
{
   int x = 2;
   double y = 2;
   cout << cube(x) << cube(y) << endl;
}
```

- Here cube is explicitly overloaded for int, but could also be obtained instantiating the template
- The overload is more specific, hence it is considered a better match

Template classes

- Classes can be defined with respect to templates (like functions)
- The syntax is:

```
template <typename T1>
class MyClass
{
    // ...
};
```

Example - Template Class

```
template <class T>
struct MyArray
{
    T *m_p;
    MyArray() : m_p(NULL) {}
    MyArray(int size) : m_p(new T[size]) {}
    const T& operator[](unsigned i) const { return m_p[i]; }
    T& operator[](unsigned i) { return m_p[i]; }
    ~MyArray() { delete [] m_p; }
};

void main()
{
    MyArray<int> x(100);
    x[0] = 4;
}   // here x goes out of scope, the destructor is called and memory is
    deallocated
```

- MyArray is an example of rudimentary container class, with very basic functionality. It manages memory. The copy semantic is not well defined.

Templates – Multiple Arguments

- A template class or function can have many template arguments
- Example:

```
template <typename T1, typename T2>
void foo(T1 x, T2 y)
{
   // do something with an object of type T1
 and one of type T2
}
```

std::pair

- **pair** is a simple template class provided by the Standard Template Library (STL)
- It bundles together a pair of values, which may be of different types (T1 and T2). The individual values can be accessed through its public members first and second.
- It requires **#include <utility>**
- See:

  http://www.cplusplus.com/reference/utility/pair/

  http://www.cplusplus.com/reference/utility/make_pair/

Example - pair

```
#include <utility> // std::pair
#include <iostream> // std::cout
int main ()
{
   // we use the default constructor, then the template function
   // make_pair, and the assignent operator
   std::pair<int,double> x = std::make_pair(10,20.4);
   // we use the constructor
   std::pair<int,int> y(4, 5);
   // << is not defined for a pair, so we output manually
   std::cout << "x: " << x.first << ", " << x.second << "\n";
   std::cout << "y: " << y.first << ", " << y.second << "\n";
   return 0;
}
```

std::pair - Implementation

```
template<class T1, class T2>
struct pair
{
    pair() {}
    pair(const T1& a, const T2& b)
      : first(a)
      , second(b)
    {
    }
    T1 first;
    T2 second;
};

template<class T1, class T2>
pair<T1,T2> make_pair(const T1& a, const T2& b)
{
    return pair<T1,T2>(a,b);
}
```

std::complex

- **complex** is a template class provided by the Standard Template Library (STL)
- The argument T is limited to *float*, *double* and *long double*
- It defines all the operators necessary to do complex arithmetic (+, -, *, /, ...)
- It requires **#include <complex>**
- It defines **literals** for imaginary numbers (C++14)
- See: http://en.cppreference.com/w/cpp/numeric/complex
- In the user guide note that it lists member functions and non-member functions

Example - complex

```
#include <iostream>
#include <iomanip>
#include <complex>
#include <cmath>
int main()
{
   // note 'using' is local, to avoid defining the suffix i everywhere
   using namespace std::complex_literals;

   // format output
   std::cout << std::fixed << std::setprecision(1);

   std::complex<double> x(1, 2); // 1+2i (use constructor)
   std::complex<double> y = 1i * (1.0 - 1i); // 1+i (use literals)

   // arithmetic operators (e.g. +, -, *, /) and mathematical functions
   // are overloaded for complex<T>
   std::cout << "x = " << x << '\n';
   std::cout << "y = " << y << '\n';
   std::cout << "x*y = " << x*y << '\n';
   std::cout << "exp(x) = " << std::exp(x) << "\n";
}
```

Templates

- Templates is one of the most advanced concepts of C++
- There is much more about templates, which we will not discuss
- As beginners, you will use template classes written by others (e.g. the STL)
- Compilation errors are difficult to understand
- It takes some experience to write template classes on your own and master them.

More on Templates ...

- Templates of templates
- Non type template arguments
- Nested templates
- Partial specialization
- Explicit instantiation
- The **using** keyword
- SFINAE
- Template enablers
- Variadic arguments
- Traits
- Metaprogramming