Advanced C++

Ron Akkersdijk

Come along. Saxion.

Autumn 2014

SAXION

Overview

- ...
- Datatype convertions
- Class Templates
- Function Templates
- Template Examples
- Generic Algoritms
- Function Objects
- ...

C++ paradigma's

- The C++ language supports multiple paradigma's
  - Procedural Programming (C heritage)
  - Object Oriented Programming
  - **Generic Programming**
  - Functional Programming

C++ templates

- Purpose:
  - To describe <u>generic</u> solutions, for <u>arbitrary</u> types
- Usage:
  - To prevent writing the same code multiple times for different types
- Application:
  - Container classes like:

    vector<V>, list<V>, map<K,V>, …
  - Algorithms, Adapters, Problem Solving classes
- Anything where you describe the “how-to” without the “with-what”

SAXION

Templates & OOP

- There exists a conceptual relation between Templates and Object Oriented Programming
- A *program* is a collection of *objects*
  - *Objects* have values
- *Classes* are recipies for *objects*
  - *Classes* omit the actual values
  - *Classes* describe the operations on objects
- *Templates* are recipies for *classes*
  - *Templates* omit the actual types

A generic Box

- The generic description for a class Box

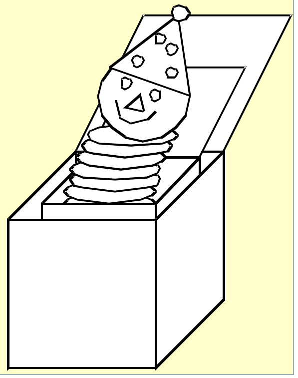
```
template <typename Type>   // For some type
   class Box               // a Box class
{
private:
    Type   content;        // the contents
    bool   isopen;         // the current state
public:
    Box(Type);             // a box with something
    bool   isOpen();       // is the box open?
    void   open();         // open the box
    Type   getContent();   // give the contents
    void   close();        // close the box
};
```

Using Box<T>

- The actual T may be anything that is a type
  - The compiler “writes” the code on demand!

```
Box<int>        ibox(8); // a template class
Box<string>     sbox("Hello world");

class Jack { ... };
Jack    jack;
Box<Jack>             box1(jack);
Box<Jack*>            box2(&jack);
Box<Box<Jack> >       box3(box1);
```



NB extra space needed here (>>)

- Note: std::string ≡ std::basic_string<char>

The compiler made 5 new classes for us

Java *generics* works differently

- A java **ArrayList<T>** instance will always be an instance of class **ArrayList<Object>**
- Where you use ArrayList<T> the compiler adds code
  - For: *put*(Object x)

```
if ( ! (x instanceof T) )   throw ...
arraylist_put( x );
```

  - For: T *get*()

```
Object  x = arraylist_get();
if ( ! (x instanceof T) )   throw ...
return  (T) x; // typecast Object x to T x
```

  - Therefore that T must always be a type!

Class Template Syntax

- Class template definition:

```
template <typename T1, typename T2, ... >
   class C  { ... }; // i.e. class C<T1,T2,...>
```

- Method implementation template:

```
template <typename T1, typename T2, ... >
     type C<T1,T2,...>::method(type p1, …)
          { ... }
```

The template arguments are part of the name of the class!

- but often written as an all-in-one definition

Class Template Syntax

- template definition & implementation:

```
template <typename T1, typename T2, ... >
   class C  {
     …
     type method(type p1, type p2, ...) {
       …
     }
};
```

- Note: not all parameters have to be typenames
- Note: parameters can have defaults
- Note: old programs often use “class” rather then “typename”

Class Template Syntax

- Beware:

```
template <typename T>
    C<T>::C(T x)  { ... };
```

  - Is the constructor: `C<T>(...)` for multiple class<u>es</u> `C<T>`

plurals!

- Versus:

```
template <typename T>
    C::C(T x)  { ... };
```

  - is the recipe for the constructor<u>s</u>: `C(...)` for <u>one</u> class C (for various types T)

- So C<int>, C<string>, ... and C can coexist!

Function Templates

- The template mechanism also applies to functions and methods

```
template<typename T> // for some type T
T  doubleIt( T x )    // i.e. doubleIt<T>(...)
{
   return  x * 2;
}
```

- Used as (the compiler infers the type)

```
doubleIt( 7 );      // T=int because 7 is an int
doubleIt( 8.5 );    // T=double
doubleIt<float>( 10 );// int implied, float wanted
doubleIt( m );      // with e.g. Magic m
                    // provided you can say: m * 2
```

- *Note: No template defaults unless C++11*

Explicit Specialization

- You can overrule a template by explicitly giving an alternative implementation:

```
// alternative implementation for: doubleIt<Magic>
template <>                   // no "typename T" here!
Magic  doubleIt( Magic x ) // replaced all T's
{
   return  x.clone(); // different implementation!
}
```

- Note: Also applies to template classes
- Note: ALL template parameters must be specified (e.g. map<K,V>)

Friend Functions

- Because friend *functions* exist independent from a *class template*, getting the code right can be tricky. The simple solution is:

```
template<typename T>  class C {
public:
   friend void f(const C<T>& c2) {      // inline
      ...
   }
   // The compiler will “generate” a matching
   // friend function f for every class C<T>

   friend void g(const C<T>&);
   // The compiler will think this g will
   // be a NON-template function defined later
};
```

Questions?

Java Autoboxing (again)

```
template <typename T>
   class AutoBox {
      T  data;         oops!        // contains a T value
   public:
      AutoBox(T x=0) : data(x) {}              // store T
      operator T() const { return data; }  // fetch T
};

typedef AutoBox<bool>       Boolean;
typedef AutoBox<int>        Integer;
typedef AutoBox<double>     Double;
typedef AutoBox<void*>      Pointer;
...etc...
```

- Also see *Datatype Conversions* before

Templates & array's

- Usage: `Array<int,10>`, `Array<int>` or `Array<>`

```
template <typename T=int, unsigned N=10>
   class Array {     // like vector<T>
         T* data;  // points to "array of T"
      public:
         Array( unsigned size = N )
               : data( new T[size] ) {}
};
```

- Using new[] here is not fully generic (the actual type for T would need a *default constructor* !)
- and not very efficient because the actual content will always be *constructed* elsewhere!

A support class

```
template <typename T>
class ArraySupport { // a generic support class
public:
   T* allocate(size_t); // mimic new T[n]
     // without calling default constructors
   void construct(T*, const T&);
     // mimic a copy-constructor
     // by copying an existing object of type T to
     // the non-initialized area
   void destruct(T*);    // mimic delete T*
   void deallocate(T*); // mimic delete[]
     // without invoking destructors
};
```

- This way we delay the “construction” of objects until it is actually needed. Ditto for “destruction”

A support class

```
#include <cstdlib>   // malloc/calloc/free from C

template <typename T>
T* ArraySupport<T>::allocate(size_t n) { // new T[n]
     // get enough space for an array of T[n]
     return (T*) malloc( n * sizeof(T) ); // c++ style
     // or use:  calloc( n, sizeof(T) ); // java style
}

template <typename T>
void ArraySupport<T>::deallocate(T* a) { // delete[]
     // release the space occupied by the array
     free( (void*) a );
}
```

A support class

```
#include <cstring>    // for: memcpy from C

template <typename T>   // fake copy-constructor
T* ArraySupport<T>::construct(T* dest, const T& from)
{
     memcpy( (void*) dest, (const void*) (&from),
                                    sizeof(T) );
}

template <typename T>   // fake delete T*
void ArraySupport<T>::destruct(T* o) {
     o->~T(); // just call the destructor
}
```

Container example

- Example (**`Array.h`**)
- A simple, fixed size, **`Array<T>`** container with
  - **`size()`**, **`at(`*`index`*`)`**, **`[`*`index`*`]`** and also the
  - **`begin()`** and **`end()`** methods for iterators
- And an **`Array<T>::iterator`** with
  - **`==`**, **`!=`**, **`++i`**, **`i++`**, **`--i`**, **`i--`**, **`*i`** and i**`->`**… operators

Questions?

Template pitfalls

- Defining methods inside the class definition will make them inline by implication, leading to code bloat when large methods are called in many places in the source code
- When defining the methods outside the class definition people often forget the type specifier

```
// Box.tcc - implementation
template <typename T>
  Box<T>::Box(T x) : content(x) {}
```

don't forget that <T>

Template Limitations

- Class template parameters can have *defaults*, but this is not allowed for template functions (*unless your compiler fully supports C++11*)
- Solution:
  - Define multiple function templates

```
template <typename T>
   type somefunc(T ...) ...
```

```
template <typename T, typename V>
   type somefunc(T ..., V ...) ...
```

```
template <typename T, int N>
   type somefunc(T ..., int n=N) ...
```

Template Limitations

- If the actual type of some T is “unusable” the compiler can only detect the problem where it occurs in the (final) source code
- For instance:
  - You can not document constraints on the actual type as in: `template <typename T : int>`
    - to demand that T must “inherit” from “int”
  - or: `template <typename T :: size()>`
    - to demand that T has a “size()” method
- So, you will get a very cryptic error message

Questions?

About Instantiation

- A template is just an incomplete recipe so it can not be compiled
- Templates will be “instantiated” when used, which may occur in multiple source files, possibly leading to code duplication
- The actual mechanism is compiler dependent
- There exist multiple compilation models:
  - *Separation Compilation Model*
  - *Inclusion Compilation Model*
  - *Explicit Instantiation Model*

Separation Compilation

- Tries to create unique instantiations of templates

```
// Box.h
export template <typename Type>
   class Box {
      …
};
```

- Getting it right is complex & error-prone
- Not supported by many compilers
- Note: “export” is deprecated since C++11

Inclusion Compilation

- The header file also contains the methods

```
// Box.h
template <typename T>
  class Box {
     T contents;
   public:
     Box(T x) : contents(x) {…} // implicit inline
};
```

- Each .o file may (potentially) hold an instantiation
- Unused inline methods don't produce code
- Large inline methods may lead to code bloat

Inclusion Compilation

- A variant moves the methods out of the class

```
// Box.h - class and methods
template <typename T>
   class Box {
   public:
      Box(T);            // no longer inline
};
// define the methods outside the class
template <typename T>
  Box<T>::Box(T x) : content(x) {}
```

- Note: Many compilers only instantiate the methods really used

Inclusion Compilation

- Or, a more “conventional” arrangement:

```
// Box.h - definition
template <typename T>
  class Box {
     ...              // methods no longer inline
};
#include "Box.tcc" // reads the implementation!!
```

```
// Box.tcc - implementation
template <typename T>
  Box<T>::Box(T x) : content(x) {}
```

- Note: To prevent problems with your IDE, it might be wise to give the “`cc`” file another extension e.g. “`tcc`”, but that may cause other problems in turn

Explicit Instantiation

- Controlled by the programmer:
  - Explicitly request instantiation in “this” file:

```
template class Box<Jack*>;
template class Box<Box<Jack*>*>;
```

  - Suppress instantiation in “this” file:

```
extern template class Box<Jack*>;
extern template class Box<Box<Jack*>*>;
```

- Possible drawback:
  - to play safe it will also instantiate unused methods

Questions?