Advanced C++ Topics

Alexander Warg, 2018
Overview

WHAT IS BEHIND C++
- Language Magics
- Object Life Time
- Object Memory Layout

INTRODUCTION TO TEMPLATES
- Template Function
- Template Class

WHAT I DO NOT EXPLAIN
- Standard C++ Library
Some more keywords
- new, delete, class, virtual, mutable, explicit...

Stricter type system
- e.g. no automatic conversion from void *
- custom class types
- strictly typed enums (C++11)

Function overloading / Operator overloading
- multiple functions with the same name but different arguments

Extensible hierarchical type system
- classes and inheritance

Generic programming via templates
Constructors: Special Member Functions for object initialization
- Same name as the class
- No return type
**Constructors:** Special Member Functions for object initialization
- Same name as the class
- No return type

**Destructors:** Special Member Functions for object destruction
- Name: ~Classname()
- No return type
- No arguments
CONSTRUCTORS (CLASS FOO)
**Constructors (Class Foo)**

Foo() -> Default Constructor

No arguments

Generated by Compiler if no other Constructors
CONSTRUCTORS (CLASS FOO)

**Foo()** -> Default Constructor
  - No arguments
  - Generated by Compiler if no other Constructors

**Foo(Type x)** -> Conversion Constructor
  - Is used to cast type Type to Foo (implicitly)
  - (see keyword `explicit`)
CONSTRUCTORS (CLASS FOO)

**Foo()** -> Default Constructor
   No arguments
   Generated by Compiler if no other Constructors

**Foo(Type x)** -> Conversion Constructor
   Is used to cast type Type to Foo (implicitly)
   (see keyword `explicit`)

**Foo(Foo const &o)** -> Copy Constructor
   Always generated by Compiler if not provided
   (related to `operator = (Foo const &o)`, see later)
**CONSTRUCTORS (CLASS FOO)**

**Foo()** -> Default Constructor  
No arguments  
Generated by Compiler if no other Constructors

**Foo(Type x)** -> Conversion Constructor  
Is used to cast type Type to Foo (implicitly)  
(see keyword *explicit*)

**Foo(Foo const &o)** -> Copy Constructor  
Always generated by Compiler if not provided  
(related to *operator = (Foo const &o)*, see later)

**Foo(Type a, Type b, Type c)** -> Normal Constructor
**Foo(Foo &&o) -> Move Constructor**

(related to `operator = (Foo &&o)`, see later)
**CONSTRUCTORS (CLASS FOO) C++11**

**Foo(Foo &&o)** -> Move Constructor  
(related to **operator = (Foo &&o)**, see later)

**Type &&x** -> RValue refernce  
Fast optimized move operations  
Support for **perfect forwarding**
Implicit type conversion
- among integer types (incl. enum)
- conversion ctor
- conversion operator
- from pointers/references of derived classes to pointers/references to base classes

Explicit type conversion (casts)
C++ has three (actually four) types of casts
- static_cast<type>(...)
- reinterpret_cast<type>(...)
- dynamic_cast<type>(...)
- const_cast<type>(...)
**Virtual Functions**
Support for Overriding functions in C++

**Pure Virtual Functions** (Abstract Function)
class A { void func() = 0; }
<A> cannot be instantiated (is abstract)

**Multiple Inheritance**
class A : public B, public C {...};
Virtual deletion ...
MULTIPLE INHERITANCE
MULTIPLE INHERITANCE

Car

name()
MULTIPLE INHERITANCE

<table>
<thead>
<tr>
<th>Car</th>
<th>Boat</th>
</tr>
</thead>
<tbody>
<tr>
<td>name()</td>
<td>name()</td>
</tr>
</tbody>
</table>
MULTIPLE INHERITANCE
MULTIPLE INHERITANCE

```
POLYMORPHISM II

Object
  cnt

Car
  name()

Boat
  name()

Amphi
```
MULTIPLE INHERITANCE
MULTIPLE INHERITANCE
Functions that operate on a *Generic Type* (e.g. $T$)
Functions that operate on a *Generic Type* (e.g. *T*)

```c
int max(int a, int b)
{
    return a>b?a:b;
}
```
Functions that operate on a *Generic Type* (e.g. \( T \))

\[
\begin{align*}
\text{int } & \textbf{max}(\text{int } a, \text{int } b) \\
& \{ \textbf{return } a>b?a:b; \} \\
\text{int } & a, b; \\
\text{int } & x = \textbf{max}(a, b); 
\end{align*}
\]
Functions that operate on a \textit{Generic Type} (e.g. $T$)

```c
int max(int a, int b)
{ return a>b?a:b; }

int a, b;
int x = max(a, b);

double a, b;
double x = max(a, b);
```
Functions that operate on a *Generic Type* (e.g. *T*)

```cpp
template< typename T >
T max(T a, T b)
{
    return a>b?a:b;
}

int a, b;
int x = max<int>(a, b);

double a, b;
double x = max<double>(a, b);
```
Functions that operate on a *Generic Type* (e.g. $T$)

```cpp
template< typename T >
T max(T a, T b)
{ return a>b?a:b; }

int a, b;
int x = max(a, b);

double a, b;
double x = max(a, b);
```
Classes with members of *Generic Types* (e.g. $T$)
Classes with members of *Generic Types* (e.g. \textit{T})

class List_item
{
    List_item * _next, * _prev;
    void * _data;
};
Templates – Class Templates

Classes with members of *Generic Types* (e.g. `T`)

```cpp
template< typename T >
class List_item
{
    List_item *next, *prev;
    T *data;
};
```
DON'T DO THIS...

**Too Much operator overloading**
- Keep usual semantics
- Avoid implicit conversion operators

**using namespace <X>** in Header Files

**#define ...**
- Use enum's for constant values
- Use templates for functions