Advanced C++ Topics
Alexander Warg, 2021
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
CONSTRUCTION AND DESTRUCTION

**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)

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

**Foo(Type a, Type b, Type c)** -> Normal Constructor
Rule Of Six

The 'default' operations:
- default constructor: \( X() \)
- copy constructor: \( X(\text{const } X&) \)
- copy assignment: operator\(=(\text{const } X&) \)
- move constructor: \( X(X&&) \)
- move assignment: operator\(=(X&&) \)
- destructor: \( ~X() \)

The default operations rules:
- If you can avoid defining any default operations, do
- If you define or \=delete any default operation, define or \=delete them all
- Make default operations consistent
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

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

Object
  cnt
  Car
    name()
  Boat
    name()
MULTIPLE INHERITANCE

- Object
  - cnt
- Car
  - name()
- Boat
  - name()
- Amphi
MULTIPLE INHERITANCE

![Diagram showing multiple inheritance relationships between Amphi, Car, and Boat objects.](image)
MULTIPLE INHERITANCE
Functions that operate on a *Generic Type* (e.g. *T*)

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

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

double a, b;
double x = max<double>(a, b);
```
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