Making New Pseudo-Languages with C++

Build You a C++ For Great Good

A 10,000 Metre Talk by David Williams-King
Agenda

1/4 Introduction
2/4 Polymorphism & Multimethods
3/4 Changing the Behaviour of C++
4/4 Metaprogramming & Frontends
Introduction

• About me
  – CBoard member for nearly 10 years
    http://cboard.cprogramming.com/
  – C++ game engine developer

• Most large-scale C++ projects have their own idioms, and invent their own “dialect” of C++

• Thinking about this explicitly is useful
C++ Language Specifications

- Pre-standard: iostream.h, ad-hoc libraries
- C++98: first standard
- TR1 (C++03): regular exp, smart pointers, hash tables, etc (just library changes)
  - Boost: major C++ library which influenced TR1
- C++11 (C++0x): second major standard, syntax changes (template >>), auto type inference, etc
- C++14 (upcoming): auto return types, better lambdas, etc.
C++ ecosystems

• Major C++ compilers
  – Borland C++ Builder
  – Microsoft Visual Studio C++ (MSVC)
  – GNU Compiler Collection (g++)
  – LLVM (clang)
  – IBM's xlc++, Intel's icc, EDG front-end (Coverity...)

• Boost: high-quality C++ libraries
  – Atomics, message-passing, serialization, regexes, preprocessors (Wave), co-routines, random number generators, shared pointers, embedded Python, ...
Agenda

1/4 Introduction
2/4 Polymorphism & Multimethods
3/4 Changing the Behaviour of C++
4/4 Metaprogramming & Frontends
Partially-Specified Behaviour

• Polymorphism through template types

```c++
// from GCC 4.9's bits/stl_set.h
namespace std
{
  template<typename _Key,
           typename _Compare = std::less<_Key>,
           typename _Alloc = std::allocator<_Key> >
    class set
    {
      // ...
```

• Polymorphism through inheritance, interface specification, composition, etc

• Polymorphism through virtual functions!
Virtual Functions

- Overriding a method with a new version
  - crops up in C code, in the runtime linker, etc.
  - Some languages do this everywhere (Smalltalk, Java, etc.)
  - C++ lets you opt in with “virtual”

- Normal function calls are bound statically; virtual function calls are bound dynamically
Multimethods

- Call a function polymorphically based on the types of multiple different classes
  - e.g. collisions in a game
  - a.k.a. multiple dispatch (double dispatch)
- “Report on language support for Multi-Methods and Open-Methods for C++” -- Stroustrup
- Can emulate with visitor design pattern
  - polymorphic source method creates a visitor class which has accept(Foo), accept(Bar), etc
  - Target class hierarchy has polymorphic visit(Visitor)
Visitor Design Pattern

```cpp
struct Visitor {
    virtual ~Visitor() {}  
    virtual void visit(const Foo1 &f) = 0;
    virtual void visit(const Foo2 &f) = 0;
};

struct Foo {
    virtual void accept(Visitor &v) { v.visit(*this); } 
    virtual void collide(const Foo &other);
};
class Foo1 : public Foo {};
class Foo2 : public Foo {};

void Foo1::collide(const Foo &other) {
    struct NewFooFunction : public Visitor { /* ... */ } f;
    other.accept(f);
    // one level of polymorphism because collide is virtual;
    // another level because of the visitor's overloading
}
```
Agenda

1/4 Introduction
2/4 Polymorphism & Multimethods
3/4 Changing the Behaviour of C++
4/4 Metaprogramming & Frontends
Operator overloading

- Simple operator overloading

```
Point operator + (const Point &other) const
    { return Point(x + other.x, y + other.y); }
```

- External operator overloading

```
std::ostream &operator << (std::ostream &o, const Foo &f)
    { o << f.getName(); return o; }
```

- Type-conversion overloading

```
operator std::string () const
    { return StreamAsString() << x << ',' << y; }
```
StreamAsString

• Use \texttt{\textless\textgreater} operator anywhere a string is expected

\begin{verbatim}
void print(const std::string &s);
print(StreamAsString() \textless\textgreater \text{“Answer: “} \textless\textgreater 42);
\end{verbatim}

• How?
  \begin{itemize}
  \item \texttt{std::ostringstream}
  \item \texttt{template operator \textless\textgreater}
  \item \texttt{operator std::string()}
  \end{itemize}
#include <sstream>
#include <string>

class StreamAsString {
private:
    std::ostringstream stream;
public:
    template <typename T>
    StreamAsString &operator << (const T &data) {
        stream << data;
        return *this;
    }

    operator std::string() const {
        return stream.str();
    }
};
Memory Management

- C-style arrays, unchecked accesses, unsafe
- New Standard Template Library containers like std::vector, std::map, std::unordered_map, etc.
  - they can do bounds-checking and auto-resizing
- Automatic memory management with smart pointers and reference counting (C++03/Boost)
- Program-wide memory management with allocator pools
Smart Pointers

• How to write a smart pointer implementation:
  – catch dereferences (operator *, operator ->)
  – catch copying (operator =, copy constructor)
  – provide comparisons, conversions (operator bool)

• std::shared_ptr, std::weak_ptr
  – shared_ptr does ref counting
  – weak_ptr can be converted to shared but doesn't count towards the reference count
Agenda

1/4 Introduction
2/4 Polymorphism & Multimethods
3/4 Changing the Behaviour of C++
4/4 Metaprogramming & Frontends
Metaprogramming

- C++ is Turing-complete (obviously)
  - So is the preprocessor: [http://stackoverflow.com/questions/3136686/is-the-c99-preprocessor-turing-complete](http://stackoverflow.com/questions/3136686/is-the-c99-preprocessor-turing-complete)
  - So are templates (see Modern C++ Design by Andrei Alexandrescu -- the library is called Loki)
- Basic ideas like singleton, factories, pools
- But also typelists, traits, multimethods, functors
Object Messages/Event Systems

- A class wants to announce a state change without knowing who is interested
  - common in GUI toolkits and game engines
- Ways of implementing this:
  - observer design pattern (quite klunky)
  - event class with functors (Boost.Signals, templates)
  - global event managing system (my favourite)
  - separate pre-processing pass (e.g. Qt moc)
Serialization/Marshalling

• Turn an object into a string and back again (for sending over a network, storing on disk, etc)

• Boost.Serialization example:

class C {
private:
    friend class boost::serialization::access;

    template <typename Archive>
    void serialize(Archive &ar, const unsigned ver) {
        ar & x;  // like << and >> combined together
        ar & y;
    }

private:
    int x, y;
};
Reflection

- Want the ability to query the functions of an unknown class, call a function by name, instantiate a class by name at runtime
  - powerful when combined with serialization
- One example: Qt's Meta-Object Compiler (moc)
  - extra pre-processing pass that constructs a meta-object for relevant classes
  - also generates plumbing for object messages
Synthesis

- add events to objects (Boost.Signals, etc)
- store events in templated thread-safe queues
- automatically serialize and deserialize events (Boost.Serialization)
- send events over the network asynchronously (Boost.Asio)
- manage memory with shared pointers
- define events in XML or Lua ....
The End.
References (1/3)

• More about C++ in general
  – CBoard http://cboard.cprogramming.com/
  – Boost! Learn it!! http://boost.org/
  – Misc: function pointers http://www.newty.de/fpt/

• Slide references
  – Images from Learn You a Haskell for Great Good http://learnyouahaskell.com/
References (2/3)

- Metaprogramming and language extensions
  - Book: Modern C++ Design by Andrei Alexandrescu (will turn you into a template wizard!)
    - Or get the code online
  - Qt Meta-Object system
    http://qt-project.org/doc/qt-4.8/metaobjects.html
  - Boost http://boost.org/
    - Especially Boost.Signals, for event systems:
      http://www.boost.org/doc/libs/1_56_0/doc/html/signals/tutorial.html#idp426643280
  - My rant about Qt signals/slots (Boost is much better!) http://elfery.net/blog/signals.html
References (3/3)

• Serialization
  – Google's protocol buffers
    https://github.com/google/protobuf/

• Multimethods
  – “Report on language support for Multi-Methods and Open-Methods for C++”
  – For stuff that actually exists, see “Multiple Dispatch” on Wikipedia
    http://en.wikipedia.org/wiki/Multiple_dispatch

• Design patterns
  – Visitor, Observer, Composition; Event Notifier:
    • http://www.marco.panizza.name/dispenseTM/slides/exerc/eventNotifier/eventNotifier.html
(backup slides)
Undefined Functions

- Convention: prototype a method but don't define the function body (to create an abstract class)
- C++ canonized this with pure virtual functions

```cpp
class C {
public:
    virtual void foo() = 0;
};
```

- Effective way to define abstract classes
C++11 Virtual Function Features

- New virtual function controls
  - `override`: this function must override a base-class function (like Java 5's `@Override` annotation)
  - `final`: can't be overridden (like Java's final)
  - `default`: use default code for default constructor, copy-constructor, assignment operator, or destructor
  - `delete`: prevent function from being called

```cpp
virtual void foo() override;
virtual void foo() final;
virtual void foo() = default;
virtual void foo() = delete;
```
Function Pointers

- [http://www.newty.de/fpt/](http://www.newty.de/fpt/)
- Original C function pointers are straightforward:

```c
void print(const char *s) {
    puts(s);
}

void (*func)(const char *) = &print;
func("Hello");
(*func)("Hello");
```
Function Pointers

- Pointers to member functions must specify scope

```cpp
class C {
public:
    int add(int i) const { return i+i; }
    int mul(int i) const { return i*i; }
};

int (C::*func)(int) = &C::add;
C c, *p = &c;
int result1 = (c.*func)(5);
int result2 = (p->*func)(5);
int result3 = (*this.*func)(5);
```