Lambda Functions for C++0x

Jaakko Järvi    John Freeman

March 19, 2008
This talk answers
- What are lambda functions?
- Why does C++ need them?
- Why are library solutions not good enough?
- How is it non-trivial to add them as a language feature?
- What is your proposal?
Outline

1. Introduction
2. Motivation
3. Proposal
4. Future
5. Conclusion
Introduction

- Regular functions consist of three parts
  - Declaration, i.e. name and type
    
    ```
    int increment(int i)
    ```
  - Definition, i.e. executable code
    
    ```
    { return i + 1; }
    ```
  - Use, as either function pointer or call
    
    ```
    increment(0);
    ```
- Sometimes useful to combine all three
  - Short functions used only once
  - Arguments to higher-order functions, like `for_each`
  - Callbacks in GUI frameworks
- Lambda functions let us do this
  - Ubiquitous in functional languages
  - Increasingly useful in object-oriented languages
Widely adopted Standard Template Library (STL) begs for lambdas

Ideally, select algorithm, specify iterators, and throw in function object

\[
\text{transform(a.begin(), a.end(), b.begin(), result.begin(), plus<int>());}
\]

Sometimes needs a little tinkering

\[
\text{string round(double n, int places);} \\
\text{...} \\
\text{transform(x.begin(), x.end(), y.begin(), bind2nd(ptr_fun(round, 2)));}
\]
Realistically, nice and compact function objects are only that way in textbook examples.

Very simple and common situations are not supported at all, requiring either a verbose custom function object...

```cpp
int bar(int a, int b, int c);

... class bar_caller {
    int a_, b_;

public:
    bar_caller(int a, int b) : a_(a), b_(b) {}
    int operator()(int c) const {
        return bar(a_, b_, c);
    }
};

... transform(x.begin(), x.end(), y.begin(), bar_caller(a, b));
```
... or a cumbersome mishmash of STL

```cpp
transform(angles.begin(), angles.end(),
    sines.begin(),
    compose1(negate<double>(),
        compose1(ptr_fun(sin),
            bind2nd(multiplies<double>(), pi / 180.0))));

// −sin(x * pi / 180)
```
Lambda Libraries

- Several libraries try to embed a lambda “sublanguage”
- Tangible improvements in many cases
- STL

```cpp
transform(angles.begin(), angles.end(),
    sines.begin(),
    compose1(negate<double>(),
              compose1(ptr_fun(sin),
                        bind2nd(multiplies<double>(),
                                 pi / 180.0))));
```

- Boost Lambda Library

```cpp
transform(angles.begin(), angles.end(),
    sines.begin(),
    bind(sin, _1 * pi / 180.0));
```
Lambda Libraries

Problems:

- Long compilation times
- Fail in unexpected ways
  
  ```cpp
  for_each(a.begin(), a.end(), cout << " " << _1);
  for_each(a.begin(), a.end(), _1.foo());
  ```
- Cryptic and extremely long error messages
  
  ```cpp
  error: const struct boost::lambda::lambda_functor<
      boost::lambda::placeholder<1>
    > has no member named foo
  ```
- 95% solution
Outline

1. Introduction
2. Motivation
3. Proposal
4. Future
5. Conclusion
Lambda Expressions

- What we want is to define a function from within an expression
- Such “local” functions need access to the current *environment*

```
for_each(a.begin(), a.end(), bind1st(plus<int>(), x));
```

- Function + Environment = Closure
  - Function objects define closures
    - `bind1st(plus<int>(), x)` // *x* stored as a member variable
  - So do lambda expressions, but avoid problems of the library solutions
    - `[x] (int y) → int { return x + y; }`
Design Goals

- Safety
  - Issues with storing environment

- Conciseness
  - Want to trim syntax everywhere possible
  - Primary tool is type deduction

...
Variables defined outside but used within a lambda expression, i.e. *free variables*, need to be stored in the closure somehow

Two choices: copy or reference

- Neither is universally good

A copy is generally safe, but sometimes impossible or inefficient

```cpp
std::ostream out;
... [] () { return out << "Hello, World!"; } ... // std::ostream is uncopyable
vector<int> v(1000000);
... [] (int i) { return v[i]; } ... // copies million—element vector
```

A reference is generally executable, but sometimes unsafe

```cpp
A a;
... return [] () { return a; } ... // returns reference to local variable
```
The programmer knows best: explicit *capture-list*

... [&out] () { return out <<= "Hello, World!"; } ...
... [&v] (int i) { return v[i]; } ...
... return [a]() { return a; } ...
Return Type Deduction

- Potentially ambiguous

```cpp
[](bool start) { if (start) return A(); else return Z(); }
```

- Restrict to single return expression

```cpp
transform(a.begin(), a.end(), b.begin(), [](int x) { return x; }); // copy a to b
```
We would like to omit argument types as well.

Besides conciseness, additional advantage is polymorphism.

template <typename F, typename T1, typename T2>
void distribute(F f, const pair<T1, T2>& p) {
    f(p.first); f(p.second); // f(a,b) = f(a), f(b)
}

... 

distribute([](x) { return cout << x; }, make_pair("SAC", 2008));
Argument Type Deduction

- But still need to type check lambda body
  
  ```cpp
  (\[](x) { return cout << x; }) ("SAC"); // OK
  (\[](x) { return cout << x; }) (non_streamable_object); // error
  ```

- Argument types need to be determined at some point

- Too late at call site: compromises modular type checking

```cpp
template <typename F, typename T1, typename T2>
void distribute(F f, const pair<T1, T2>& p) {
    f(p.first); f(p.second); // f(a,b) = f(a), f(b)
}
```

```cpp
distribute(\[](x) { return cout << x; }, make_pair(1, non_streamable_object));
```
Argument Type Deduction

- Parameter types must be inferred at definition site
- C#: if a lambda is passed to another function, we can use the type of the formal parameter

```csharp
int origin(Func<int, int> f) { return f(0); }
origin((x) => return -x);
```

- Except C++ does not have a special function type (like `Func`) to match against

```cpp
template<typename F>
int origin(F f) { return f(0); }
origin([] (x) { return -x; });
```

- Must use concepts (constrained templates)
Concepts

- Concepts are new in C++0x, to allow modular type checking of templates

```cpp
concept Negateable<typename T> {  
    T operator− (T); // constraint  
}
```
Concepts

- Concepts are new in C++0x, to allow modular type checking of templates

```cpp
concept Negateable<.typename T> { 
    T operator− (T); // constraint
}

template <typename T>
    requires Negateable<T> // requires—clause
T& negate(T& t) { return −t; }
```
Back to lambdas

```cpp
template <typename F>

int origin(F f) { return f(0); }
```
Back to lambdas

```cpp
template <typename F>
    requires Callable1<F, int>
int origin(F f) { return f(0); }
```
Back to lambdas

template <typename F>
    requires Callable1<F, int>
int origin(F f) { return f(0); }

concept Callable1<typename F, typename T1> {
    typename result_type;
    result_type F::operator() (T1); // argument types: T1
}

Implementation:
During constraint checking for lambda
Build list of argument types
Type-check lambda body
If successful, inject function into closure
Resume normal type-checking
Back to lambdas

template <typename F>
  requires Callable1<F, int>
int origin(F f) { return f(0); }

concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type F::operator() (T1); // argument types: T1
}

origin([] (x) { return −x; });
Argument Type Deduction

- Back to lambdas

```cpp
template <typename F>
    requires Callable1<F, int>
int origin(F f) { return f(0); }

concept Callable1<typename F, typename T1> {
    typename result_type;
    result_type F::operator() (T1); // argument types: T1
}

origin([] (x) { return -x; });
```

- Implementation:
  - During constraint checking for lambda
    - Build list of argument types
    - Type-check lambda body
    - If successful, inject function into closure
  - Resume normal type-checking
1. Introduction
2. Motivation
3. Proposal
4. Future
5. Conclusion
Function objects are very important for generic programming

Current syntax for defining them is too verbose, and is a hurdle for the effective use of the Standard Library, and other generic libraries

Library solutions are inadequate because of limitations that do not allow supporting all common cases, and because of several subtleties

A language solution is required, and can provide a safe, efficient, and fairly economical anonymous function feature for C++

Lambda expressions are now part of the working draft of the next revision of ISO C++