More C++

David Chisnall

March 17, 2011
Exceptions
A more fashionable goto

- Provides a second way of sending an error condition up the stack until it can be handled
- Lets intervening stack frames ignore errors
- Comes from languages with garbage collection
- Huge problem with languages that don’t have it! (like C/C++)
The Basic Idea

```c
// Throw an exception
throw x;
try
{
    somethingThatMayThrowAnException();
}
catch (int a)
{
    // Exceptions don’t have to be objects
}
catch(...)
{
    cleanup();
    throw;
}
```
Differences from Java

- Can throw anything that supports copying, not just objects
- No `finally` statement.
- Can be emulated with `catchall` and `rethrow`
How it Works: The Old Way

- Every `try` block pushes a handler function and the current instruction and stack pointer onto the exception stack
- Exiting the `try` block pops the handler
- Exceptions run by calling the handler then setting the instruction and stack pointers.
Why It Doesn’t (Usually) Work That Way Now

- Exceptions are meant to be unusual
- This imposed a cost for every try block
- Even when no exception is thrown.
‘Zero Cost’ Exceptions

- Each stack frame contains some metadata saying where the exception handlers are
- Often in the same format as debug info
- Throw function visits walks the stack reading this metadata
- Reconfigures stack for each handler
Exception Performance

- ‘Zero-cost’ exceptions cost nothing when entering a try block
- Throwing an exception is *very* slow
- Can be hundreds of times more expensive than a function call
Exception Specifiers

// May only throw a std::bad_alloc exception
int foo(void) throw(std::bad_alloc);

// May not throw any exceptions
int bar(void) throw();

- Exception specifiers are enforced at run time
- Violation invokes a callback function
- Typically aborts the program
- Could just abort the thread
// Class on the stack, destructor will be called
SomeClass f;
// Simple stack allocation, no cleanup needed
int b;
// Pointer. Just leaks!
SomeClass *f = new SomeClass();
// Exception thrown, maybe from some called function
throw a;
• Non-virtual methods in C++ are really just functions that are in a restricted namespace
• C++ generalises this, allowing all declarations to be inside namespaces
• Namespaces can be imported into the current context, or have elements from them used directly
• All of this works via the name mangling mechanism
namespace a
{
    namespace b
    {
        void function(void);
    }
    void function(void);
}; // Note the semicolon!

// Calling
a::b::function();
a::function();
using namespace a;
function();
using a::b::function;
function();
Namespaces in Use

- Anonymous namespace declarations not exported
- Standard C headers come in namespaced versions
- `#include <stdio.h>` is the same as C
- `#include <stdio>` gives the same functions in the `std::` namespace

```c
// Put all functions from some C lib into a namespace
namespace somelib
{
  #include <somelib.h>
};
```
Interoperability With C

- C doesn’t support name mangling
- C doesn’t support classes
- C usually defines the interoperable ABI for a platform - other languages can call C, but often not C++
Turn Off Name Mangling

extern "C"
{
// Header that isn’t designed for use by C++
#include <some_c_header.h>
}

// Turn off name mangling for this function
extern "C" int fortytwo(void) throw()
{
    return 42;
}

- `extern "C"` specifies C linkage
- Used when declaring a C function, or exporting a function for use by C code
- Disables overloading and namespaces for that function
- Functions exported to C should usually not throw exceptions!
C++ Headers

```c
#ifdef __cplusplus
extern "C" {
# define NOTHROW throw()
#else
# ifdef __GNUC__
# define NOTHROW __attribute__((nothrow))
# else
# define NOTHROW
# endif
# endif

void someFunction(void) NOTHROW;

#endif
```

Templates

Metaprogramming For C++

- C++ templates are written in the C++ template language
- C++ code with some placeholders
- Placeholders filled in when template is used
- A big part of the reason C++ executables are so huge
**Templates for Type-Generic Use**

```cpp
template <typename T> T max(T a, T b) {
    return a > b ? a : b;
}
int a = max(1, 2);
std::vector<int> vector;
```

- Template parameter defines the type
- Instantiated with the type where used
- Used for all of the STL collections
Templates for Exception-Safe Pointers

```cpp
std::auto_ptr<SomeClass> p(new SomeClass());
```

- p is an on-stack instance of `std::auto_ptr`
- The `std::auto_ptr` class wraps the pointer
- Assigning the pointer removes ownership from the `auto_ptr`
- The `auto_ptr`'s destructor deletes the object
- Exception-safe cleanup of pointers
Smart Pointers

- Template, wraps a pointer
- Copy constructor increments reference count
- Destructor decrements reference count
- Object freed when last smart pointer destroyed
- *Very* hard to get right
- Provided by Boost, C++0X
Templates for Compile-Time Calculation

template <unsigned int n> struct fib
{
    static const unsigned int result = fib<n-1>::result + fib<n-2>::result;
};
template <> struct fib<1>
{
    static const unsigned int result = 1;
};
template <> struct fib<0>
{
    static const unsigned int result = 0;
};
int main(void)
{
    printf("%d\n", fib<9>::result);
}
Template Performance

- Instantiated at compile time
- Not instantiated multiple times
- Template recursive Fibonacci is $O(n)$
- Nontrivial templates take a lot of time to compile
- Linker will try to remove identical template instantiations
- Sometimes, it fails (e.g. `std::vector<uintptr_t>` and `std::vector<void*>`)
Questions?