C++: A Less Bad Systems Language

Sponsored by Green Hills Software

Green Hills make the world's highest performing compilers, most secure real-time operating systems, revolutionary debuggers, and virtualization solutions for embedded systems.

Green Hills® SOFTWARE

S15 Talk Series: Second Wave

- Programming languages
  - That aren't covered in the core curriculum
  - That are useful

- Next Week: Real-Time Operating Systems
  - Given by a Green Hills engineer
    - ie. someone who is extremely qualified to talk about this

- Schedule: http://cmucc.org/talks
  - Haskell, sed/awk, Verilog/FPGAs, and more

Computer Club

- We exist! We do things other than this!
- Open Hacking Hours
  - Every Saturday at 5pm, Cyert B6
  - Join us in hacking on all kinds of neat projects
    - Put skills from talk series to use!
- Retro Committee Meeting
  - This Saturday at 5pm, Cyert B6
  - Hack on and organize events related to retrocomputing

C++: A less bad language

- Quick poll:
  - How many of you know C?
  - How many of you know Java?
    - Some other object-oriented language?

- C++
  - A lot like C
    - Almost backwards-compatible
    - But a **lot** more features
      - Too many?

About this talk

- Too much C++ to cover this fast!
  - You should read a book
    - … but you won't
- Will try to mention:
  - Cool features
  - Dangerous features
- Gets increasingly vague
  - “less important” material

- Use other resources!
  - Nothing is covered in enough detail here
  - See final slides

Why C++

- Still low level
  - Can get almost all the performance benefits of C
    - And a few additional ones
- C-style linking
  - (albeit using disgusting hacks)
- Almost automatic memory management
  - … with a bit of work
- Powerful type system
  - Catch errors at compile time
- Elegant and extensive standard library
  - Still not nearly as large as Java, Python, etc

C++11

- Major change
- Some of this talk is C++11-specific
  - Hopefully this never matters to you
- May not be the default in your compiler
  - More on compilers later

Function Overloading

- Multiple implementations of same function name
  - Different number and/or types of arguments
- Name mangling
  - Function names are “mangled” by compiler to produce object files compatible with C linkers
  - Terrible hack, but works
  - Used for other C++ features as well
- Disable with `extern “C” { }`
  - To enable linking from C programs
  - Disables function overloading, other C++ features

References

- A “second name” for an existing variable
- A lot like a pointer
  - But it will always point to valid memory!
    - … hopefully

- Example:

```
int x = 5;
int& y = x;
assert(&y == &x);

int* z = nullptr;
y = *z; // don't do this!
```

Namespaces

```
namespace foo {
    void bar();
}

bar();          // will not compile!
foo::bar();     // will compile

{
    using namespace foo;
    bar();      // will compile
}

bar();          // will not compile
```

Object Orientation

- Aids in creating modular code
  - Better organization, reuseability
- Like a C struct, but with functions

- Inheritance
  - Let's ignore this for now

Objects in C++

- Probably the most obvious change from C
- How it works:
  - Header file:

```
class MyClass {
   int member_variable;
private:
   void internal_method();
public:
   void external_method();
};
```

  - Implementation file:

```
void MyClass::internal_method() { … }
void MyClass::external_method() { … }
```

Constructors

- Guaranteed to be called before before an object comes into scope
  - This is extremely useful— more on that later

- Header:
  - No return type
  - Name is the same as the class name
  - Any arguments
    - Overloading encouraged

- Invoked during variable declaration
  - eg for class MyClass:

```
MyClass test_object(...);
```

Constructors (cont)

- Initialization of member variables
  - Happens *before* constructor body
  - Initializer lists
    - Allow you to call something other than default constructor

- Example:

```
MyClass::MyClass(int var) :
member_variable(var)
{ }
```

Default Constructor

- The no-arguments constructor

- Invoked implicitly in many cases
  - Remember: A constructor is always called

- Implicitly defined default constructor
  - Just default constructs everything
  - Delete it by setting equal to delete

```
class MyClass {
    MyClass() = delete;
}
```

Copy Constructor

- Single augment, same type as class being constructed

- Implicit invocations
  - When passing by value
    - Pass by const ref to avoid!

- Implicitly defined copy constructor
  - Just copy constructs all elements
  - Delete it by setting equal to delete

```
class MyClass {
        MyClass() = delete;
}
```

Converting Constructors

- Single-argument constructors
- Invoked implicitly when a conversion is needed
  - Danger!
- `explicit` keyword
  - Disables implicit calls

Destructor

```
class MyClass {
   ~MyClass();
}
```

- Guaranteed to run when object goes out of scope
  - Or when dynamically-allocated memory is freed
- Chance to free resources, etc

Assignment Operator

```
MyClass& operator=(const MyClass& other);
```

- “Make this object the same as this other one”
    - ie. just like with an `int`

More Operators

- Methods with special meaning
  - Some automatically created and called

- Easy ones:
  - T operator+(const& T other) const;
  - T operator-(const& T other) const;
  - T operator==(const& T other) const;
  - etc

- Should be fairly self-explanatory
  - Implement them as makes sense for your class
    - Or not at all

Operators: Final thoughts

- Very convenient features
- Lots of potential headaches
  - Watch out for implicit calls
  - Be careful

Object Creation

- Static allocation:
  - MyClass test;
    - Invokes default constructor
  - MyClass test(...);
    - Invokes appropriate overload
- Dynamic allocation
  - `MyClass* test = new MyClass(...);`
  - `delete test;`

  - Do not mix malloc/free and new/delete!

Class Templates

- Idea: Type-generic data structures, etc
- Template arguments
  - inside < >
- Full class definition must be in header file
  - Cannot be compiled without instantiation
- Example:

```
template<typename T>
struct LinkedListNode {
  LinkedListNode<T> next*;
  T data;
}
```

Function Templates

- Same as class templates, but on a single function

- Example:

```
template<typename T>
struct LinkedListNode {
   LinkedListNode<T> next*;
   T data;
}
```

Inheritance

- Less important (than in Java, etc) due to templates
- Very brief overview:
  - General "base" class
    - eg. Dictionary
  - Specific "derived" class
    - eg. HashMap
  - Result:
    - Write functions, etc to operate on base class
      - Use them on the derived classes as well

- Good Object Oriented design is complicated
  - Read a book or take 15-214

Virtual Functions

- Allows derived class to override base class' implementation of a function
- Virtual function table
  - Created iff class contains a virtual function
  - Used to resolve function calls

- Pure virtual functions
  - No implementation
  - Makes containing class “abstract”
    - Cannot be instantiated
    - Useful only as a base class

Templates vs Inheritance

- Inheritance inherently hierarchal
  - Templates much more generic
- Type safety
  - Inheritance provides clear requirements
    - Enforced by compiler
  - Templates need whatever they use
    - Compiler checks at instantiation
    - Not always clear to programmer what is used
      - Nasty compilation errors
- Dynamic vs Static
  - Templates: Everything resolved at compile time
  - Inheritance: Resolved at runtime

Curiously Recurring Template Pattern

- In case you feel insufficiently confused...

```
template<class T>
class Base
{
    void foo();
};

class Derived : public Base<Derived>
{
     void foo();
};
```

http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Standard Library

- Collection of useful classes and functions
  - Data structures, file io, etc
- Lots of templates
- Very portable
  - Pretty much anywhere C++ runs

- http://en.cppreference.com/w/
  - Keep this up while writing code!

Boost

- The “less standard” library
  - Features often pushed into standard library
- Moves (relatively) fast
  - Often less elegant, user-friendly
- Encourages arguably terrible design
  - Uses debatable C++ features
    - Template metaprogramming
    - excessive templatization
    - functional programming
- Less bad with C++11?

- http://www.boost.org/

Exceptions

- Use throw keyword to indicate error
  - Throws an object (an “exception”)
  - Program control passes to nearest try-catch block
    - Up the call stack as needed
    - Destructors called along the way
- try-catch block
  - Case on exception object
  - Handle the error
- Crashes if no try-catch block is found

RAII

- Useful idiom for objects
  - Heavily used in standard library and Boost
- **R**esource **A**cquisition **I**s **I**nitialization

- Resource lifecycle:
  - Initialized when acquired
    - ie. in constructor
  - De-initialized when released
    - ie. in the destructor

RAII (cont)

- Advantages
  - No “used uninitialized” bugs!
  - No “forgot to free” bugs!
  - Elegance

- Disadvantages
  - Cannot initialize based on an if-else statement
  - Arrays automatically initialized with default constructor
  - etc

Smart Pointers

- RAII pointers
  - Allocate memory at declaration
  - Free memory when out-of-scope
- Reference counting
  - Keep track of reference count in copy constructor, etc
  - Enables sharing of smart pointer
    - Small overhead
- Standard Library
  - std::unique_ptr
    - No ref counting
  - std::shared_ptr
    - Ref counting

Move Constructors

- Usage example:
  - Want to pass ownership of a unique_ptr
- Move Constructors:
  - Similar to copy constructor
  - Leave original in undefined state
    - So we can steal resources, etc

- Implementation examples:
  - std::unique_ptr
    - Pass ownership of pointer
  - std::vector
    - Pass ownership of data field
      - (which is a dynamically-allocated array)

Move Constructors (cont)

- xvalue
  - expiring object
    - ie. one that can be moved from
  - When:
    - return values
      - Since they are about to go out of scope
    - `std::move(var)`
      - Make a xvalue representing var
      - var is left in undefined state
- Accepting xvalues as arguments
  - Overload will only match if an xvalue is provided as an argument
  - `void foo(T&& xval);`

Iterators

- Classes which “behave like pointers”
  - ie. implement `operator*`
- Forward iterator
  - Implement `operator++` “like a pointer does”
- Random Access Iterator
  - Implement `operator[]` “like a pointer does”

- Use in for loops:

```
std::vector<std::ifstream> v = get_files();
for(std::ifstream& file : v)
   do_something(v);
```

“Bad” Features

- Sometimes considered harmful
  - “Overly complicated code”
- My opinion
  - People should learn the language they work in
  - Lots of useful features
- A more moderate opinion
  - Google style guide:
  - http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

- Not entirely clear what these features are
  - … but the rest of this talk probably includes a lot

std::function and Lambdas

- std::function
  - Because C function pointer syntax is terrible
  - `std::function<ReturnType(ArgType1, ...)>`
- Lambdas
  - Lightweight functions
  - Declared like regular variables
- Details beyond the scope of this talk
  - And easy
  - Google it

Casting

- C casts considered harmful?
  - No checks
  - Syntax makes them easy to overlook
  - C++ provides better options

- reinterpret_cast<T>(T)
  - Casts between pointer types
  - Or pointers to arithmetic types
  - Covers most cases
- More casts
  - Lots of options: http://www.cplusplus.com/doc/tutorial/typecasting/

Compilers

- g++
  - like gcc, but for C++
  - Most common?

- clang++
  - Nicer error messages
    - **much** nicer
  - Less widespread
    - Dubious installation on Andrew machines
  - Problematic with gdb

Debugging

- Largely the same as C
  - Which is good!
    - Lots of tools, etc
- gdb
- valgrind

Additional Resources: Books

- Introduction:

  Accelerated C++: Practical Programming by Example
  - By Andrew Koenig
  - ISBN 860-1400402207
  - Good for learning C++ from scratch

- Advanced:

  Effective C++ and Effective Modern C++
  - By Scott Meyers
  - ISBN 978-0321334879 and 978-1491903995
  - Collection of tips for improving your C++

Additional Resources: Reference/ Overview

- API Reference
  - http://en.cppreference.com
- Language references/guides:
  - http://www.cplusplus.com/doc/tutorial/
    - More approachable
  - http://en.cppreference.com/w/cpp/language
    - More thorough