Building Real Software
My Background

• Research Scientist at Yale in Dedis group

• Large scale distributed systems:
  • P2P VPN in C#
  • Anonymity software C++
  • Decentralized / Automated general purpose computing grid supporting Hadoop, MPI, and Condor
Relevance

- Academic projects
  - More than single use projects
  - Encourage use by external parties
- Industry projects
  - Bugs will haunt you
  - Job security
Talk Overview

- Understanding the project life cycle
  - Starting the project
  - Building the project
  - Living with the project
- Improving your skills
  - Coding
  - Advanced C++
  - Advanced programming
- Recommended reading
Before Starting the Project

• Know your team – have roles
• Pick your basic tools
  • Documentation
  • Doxygen
  • Wiki
  • revision history
  • Git
  • Mercurial
  • Subversion
  • Web / web services
• Group management
• Third-party services (github, bitbucket, Google code, SourceForge)
Starting the Project

• Understand the problem
• Understand the clients (users) expectations
• What are the inputs and outputs?
• Who will be using this?
• Don’t solve problems you don’t need to.

Design a solution

• Enumerate the different modules
• Establish interfaces and relationships
• Make your own mini-libraries (well defined separate parts of the solution)
• Don’t over engineer
Starting the Project

• Examine your solution

• Compare the definition of your problem to your proposed solution

• Did you consider if any aspects of your solution might have many different implementations (or representations?)

• Do your interfaces make sense to the uninitiated (someone other than you and your colleagues)
Starting the Project

• Which language are you going to use?
  • The language you know best is not always the best choice
  • You will always need tool applications: tests, data gather, parsing, etc – use this as opportunities to test new languages
  • Learning different programming languages can teach you new ways to solve problems in languages you consider an expert in
Starting the Project

• Deciding which libraries to reuse
  • Many people write and maintain libraries
  • Many other people use these every day
  • Popular libraries tend to be written with higher quality than you will be able to attain in an afternoon of hacking
  • In general, write as little code as possible
  • Libraries are less likely to have bugs than your own code
  • More people to help you if you find a bug in a library
  • You’ll complete the job much more quickly by not having to write more code
  • You’ll have much less code to maintain
  • Avoid dead libraries – Unless you want to adopt them
Starting the Project

• Revisit your solution
  • You’ve made a lot of decisions since the original design – does your solution still make sense?
  • Does your choice in libraries mean you’ll need to construct interfaces so that different libraries can support the same functionalities?
• Delegate responsibilities to each individual
• Begin…
Building the Project

- Write tests first
  - Express the behavior you expect from your interfaces (modules) *before* you even write your first line of code
  - Unit tests enable to ensure functionality *before* you have completed the entire solution
  - When you make a modification, you can use unit tests to focus on ensuring (or minimizing the risk) that the expected behavior for your application has not changed without doing full scale integration tests
Building the Project

• Experiment
  • Programming languages are flexible!
  • If you can imagine a solution, with enough ingenuity you can make it a reality
  • If you do not understand something, write a toy application, I have 100s

• Write it correctly the first time
  • Don’t leave hacked code in your solution
  • You will forget
  • Someone will use it and find a bug
  • You will spend a long time trying to figure out the bug
Building the Project

• Use it yourself
  • The client may relay errors that occur, If you have not used it, it may be meaningless to you
  • If you cannot use it, no one else will be able to

• Checking the Project
  • Check for memory leaks (valgrind) using both the unit tests and a sample running of your solution
  • Ensure that the solution solves the needs of the client
  • Use debuggers (gdb or those built into the IDE)
Living with the Project

• Handling Bugs
  • Calm down!
  • Verify their existence
  • Write a test case that exhibits the bug
  • Narrow down the location of the bug via debuggers or code analysis
  • Solve, verify test passes and the original process no longer triggers the bug
  • Commit the test to the repository – bugs tend to crop up again and again
Living with the Project

• Keep the project focused
  • You may be tempted to introduce code redesigns or new methodologies – DON’T
  • Your code should look like it was written with a single mind or idea
  • Do not mix new concepts into existing code
  • Exception: you proliferate the changes throughout the entire solution
  • If you have a better solution, try to take advantage of the fact that you have code modularity
Living with the Project

• What if the project does need a redesign
  • Remove legacy components
  • Truly general solutions tend to invite bloat, confusion, and bugs!
  • Try to keep as many of the existing modules as possible
Advanced C++

• C++ 0x/11 is introducing a lot of timely features
  • Scoped / Shared pointers
  • Hashables
  • Foreach (ranged for)
  • Lambda (nameless) functions
  • Thread support

• Use “stack” allocations as much as possible!
  • Heap allocations take time
  • Require you to remember to delete it
  • Unfortunately cannot represent different implementations of the same interface
Advanced C++

• Using a stack versus a heap:

Stack:

Person david(“David”);
Banana banana;
david.eat(banana);

Heap:

Person *david = new Person(“David”);
Fruit *banana = new Banana();
david->eat(banana)

• Deleting

• Stack – there is no need to delete, it is clear
• Heap – we definitely need to delete david, but when we feed David the banana, does he delete it?

• Fruit is abstract, we cannot create a Fruit on the stack, but we can pass a Fruit by reference: Person::Eat(const Fruit &fruit);
Advanced C++

• Good function prototypes
  • Pass by reference instead of by pointer
  • A pass by pointer may imply optional and the user may pass a null pointer:
    • Method(const Class *value); // value can be null
    • Cannot pass a null pointer via reference
    • Method(const Class &value); // value cannot be null
  • Always include “const” unless the calling function needs to modify the state of passed reference
  • Do not pass simple data types (int, char, bool) by by const reference … it does not do anything good and may actually inhibit some optimizations.
Advanced C++

• Casting!
  • Example:
    
    ```
    int x = 5;
    double y0 = (int) x;
    double y1 = static_cast<int>(x);
    ```
  
  • Do not use C style casting it is difficult to find
  
  • Understand the different C++ casting mechanisms
Exceptions vs Error Codes

Exceptions:
try {
    do_something();
} catch(BadInput &e) {
} catch(IOError &e) {
} catch(exception &e) {
    throw;
}

• Force us to evaluate on error
• Are expensive
• Contain more information
• Standardized

Error Codes:
int result = do_something();
if(result == BadInput) {
} else if(result == IOError) {
} else if(result != Expected) {
}

• Cheap
• Easily ignored
• Difficult to propagate
• Not standardized
Exceptions vs Error Codes

• Use exceptions only for exceptional behavior
  • Exceptions are not free
  • In my C# project, the use of a try statement masked a race condition due to timing delays
  • Make your own exceptions where appropriate
  • Try to avoid default exception handlers

• Why use error codes?
  • Some languages do not support exceptions (C)
  • Historically, not all C++ implementations supported exceptions
  • Services which cannot pay the penalty incurred by exceptions
Advanced Programming

- Asynchronous programming
  - Avoid long running tasks in your primary thread
  - Events and Callbacks
  - State machines
- Is a vs Has a
  - Defining the relationship between two closely related modules
  - Is a – can be used in place of
  - Has a – needs to use but should not be used as
  - Should try to use Has a as much as possible
  - Good interfaces can enable you to use Has a while looking like Is a (much safer)
- Is a behavior can change if the internal behavior of the parent changes
Is A vs Has A

• A square is by definition a rectangle
• In OO, making a square inherit a rectangle makes for messy interfaces:

```cpp
class rectangle {
    public:
        void setLength(double length);
        void setWidth(double width);
}
```

• A square should only have setSide, this is a confusing Api!
Advanced Programming

- Inheritance
  - Avoid multiple inheritance of non-pure classes
  - Make consistent
- Ignore early optimizations
  - Write well-organized code and do it quickly
  - Compilers can do amazing amounts of optimization
- Comment your interfaces
  - If you’re passing a pointer, make it clear who will take ownership of that pointer (if not using shared pointers)
  - Use good naming convention:
    - DoSomething(int a) or Increment(int amount)
Advanced Programming

• Code commenting
  • If you have to comment your code – maybe it is too complicated – try to simplify
  • Sometimes complicated code is unavoidable
  • Be concise, leave reminders
  • Most code *should* be self-commenting
A Few Great Books

- Kernighan and Ritchie – The C Programming Language
- Scott Meyers – Effective C++
- Scott Meyers – More Effective C++
- Gamma, Helm, Johnson, and Vlissides – Design Patterns: Elements of Reusable Object-Oriented Software
Important Points

- Test
- Test
- And test – It will save you when you’ve been working for hours on end
  - it's much easier to do make test && ./test
  - than start an application, feed it input, and see if you get what you expect
- Write comments for all interfaces
- Don’t change your coding style in the middle of a project
- Test