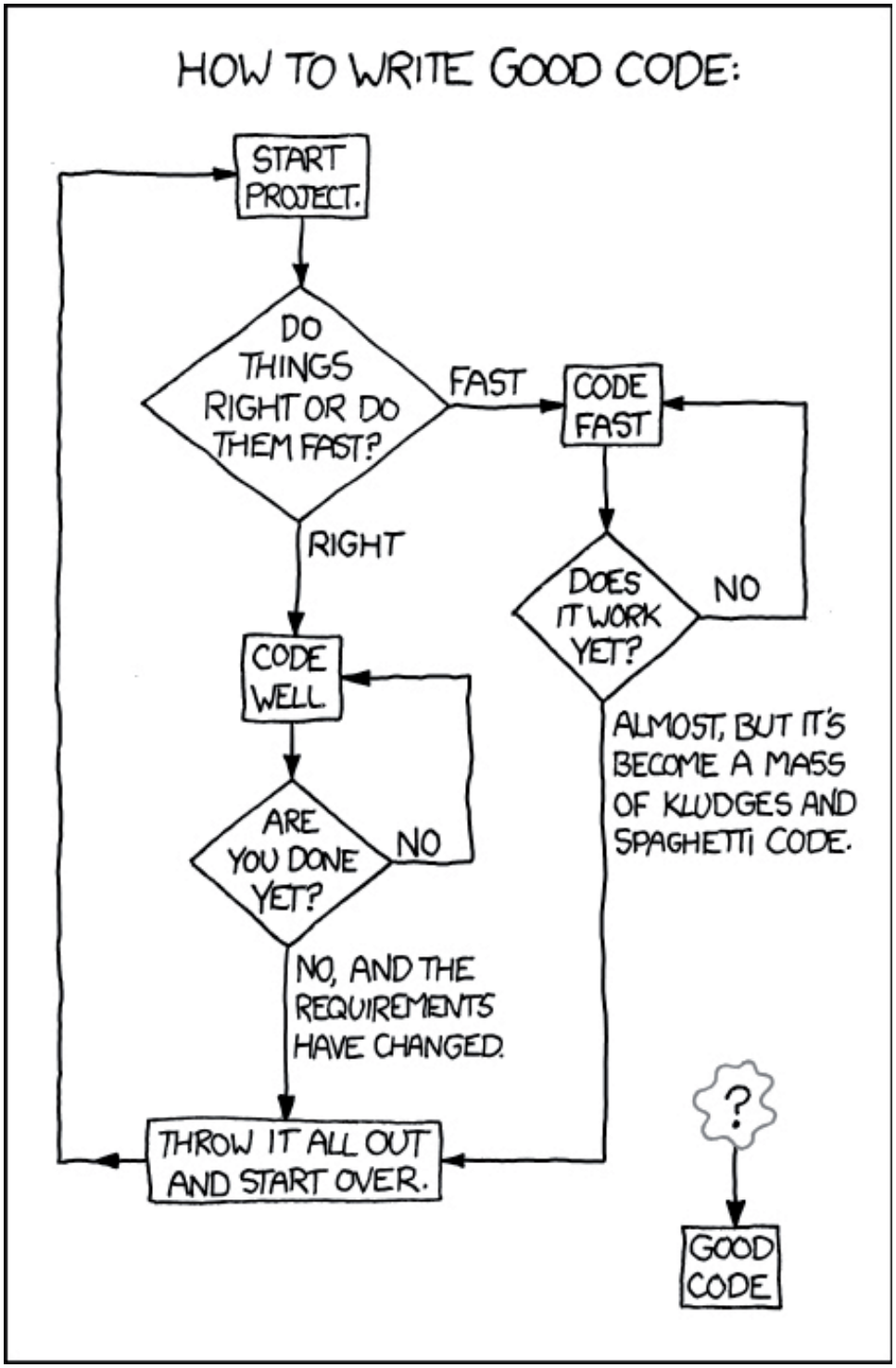
Object oriented analysis, design and coding

Statistical Methods in Particle Physics / WS 11

Niklaus Berger

Physics Institute, University of Heidelberg



http://xkcd.com/844/

Overview

- **Why**
  use objects?
- **How**
  to do that in C++?
- **Where**
  can and will it go wrong?
- **How**
  to avoid the most common pitfalls and solve common problems?

Why Objects?

For simple code, the challenge is in finding clever algorithms

- Has mostly been done before, in doubt look it up in Knuth, "The Art of Computer Programming"

For large projects, the challenge is getting things to work and keeping them that way

- Developer ≠ User
- Many developers
- Many environments
- Code changes over time
- There tends to be a lot of code

Some numbers...

Root:

- 12 core developers, O(100) contributors
- O(1 Million) lines of code
- O(1200) classes

Geant4:

- O(100) developers
- O(1 Million) lines of code
- O(2000) classes

Atlas codebase:

- O(1000) developers
- O(7 Million) lines of code
- O(10'000) classes

BES III partial wave analysis code:

- 1 core developer, O(3) contributors
- O(35'000) lines of code
- O(200) classes

Typical Ph.D. thesis in HEP:

- 1 core developer
- O(10'000) lines of code
- O(1-100) classes

Objects are a way of dealing with complexity

There are myriad books on the topic, the classic text is "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (usually referred to as the "Gang of four")

Core concepts of OO

- Data abstraction
- Encapsulation
- Modularity
- Polymorphism
- Inheritance and Composition

Data abstraction

Abstraction is what makes large programs possible, many layers...

- You need not know how a transistors works, to store a 0 or a 1
- You need not know how a number is represented in binary to add 4 and 7
- You need not know how your list of numbers is stored if you have a function returning the mean value
- You need not know how tracks are reconstructed if you can get the particle momentum

Abstraction is hiding lower levels from the user

Encapsulation

Encapsulation controls access to data and functions

- Allows you to control where in the program data can be manipulated
- Allows you to specify an interface to data and methods that users have to stick to
- Example: A list and an element counter, interface to make sure that they are always consistent
- Separate interface and implementation
- Interfaces allow collaboration between developers

Encapsulation is restricting access

Modularity

Modularity is breaking tasks down into sub-tasks

- Done anyway, but should be visible in code structure
- Concentrates everything needed for one task in one location (a module)
- Allows reuse of modules

Modularity is divide and conquer

Polymorphism

Polymorphism is sharing parts of the interface between different data types

- Allows unified treatment of different objects
- Example: Silicon hits and scintillator hits in a detector - they share a position (and related functionality) but not everything (pixel number, light yield...) - polymorphism allows to share position related functionality and unified treatment in a e.g. a track fit

Polymorphism is unifying/sharing interfaces

Inheritance is deriving specialized objects from more general ones

- Allows for the implementation of polymorphism
- Example: Derive all hits from space points
- Implements a “is a” relationship

Composition is having objects inside other objects

- Also allows for some polymorphism
- Example: A reconstructed particle contains the reconstructed track and the calorimeter cluster
- Implements a “has a” relationship

Inheritance and Composition represent object relationships

In general

- 2 files for every class
- a <Class>.h file with the interface:
  Say what your class can do
- a <Class>.C/.cpp file with the implementation:
  Say how your class does it

In the following: Focus on the interface - in the implementation, tell the compiler what class you have in mind, using the class name and :: before function names

We will also be moving away from interactive root...

Anatomy of a class

```
class Tracker
```

class keyword, class name

Most coding conventions suggest capitalized class names

Many old packages (Geant4, root) mark classes through prefix letters (G4, T). This is obsolete - if you have problems with your class names, use a namespace

Anatomy of a class

```
class Tracker{
public:
```

class keyword, class name
access modifier:
Three variants:

public: Visible from outside the class

private: Visible only within the class

protected: Visible within the class and derived classes

Only make public what you need to make public

Everything else is protected

Use private only when you know why

Anatomy of a class

```
class Tracker{
public:
   Tracker();
```

class keyword, class name
access modifier
Constructor
Method (functions in classes are called methods) called automatically upon creation of an object of this class

Used to initialize whatever needs to be initialised for objects of this class

Constructor does not have a return type and the same name as the class

Anatomy of a class

```
class Tracker{
public:
   Tracker();
   Tracker(unsigned int nlayers);
```

class keyword, class name
access modifier
Constructor
Method (functions in classes are called methods) called automatically upon creation of an object of this class

Used to initialize whatever needs to be initialised for objects of this class

Constructor does not have a return type and the same name as the class

Constructors can take arguments

There can be more than one constructor

Anatomy of a class

```
class Tracker{
public:
   Tracker();
   Tracker(unsigned int nlayers);
   ~Tracker();
```

class keyword, class name
access modifier
Constructor
Constructor
Destructor
Method to clean up when the class goes out of scope

Same name as class, with a tilde in front, no return value, no arguments

For every "new" in the constructor, there should be a "delete" in the destructor

If you do not create constructors and destructors, the compiler will add empty ones for you

Anatomy of a class

```
class Tracker{
public:
   Tracker();
   Tracker(unsigned int nlayers);
   ~Tracker();
   void DoSomething();
```

class keyword, class name
access modifier
Constructor
Constructor
Destructor
Method
Any function inside the class is called a method

They are not very different from normal functions, but will need an object of the class to exist

Inside any member function, there is a pointer called "this" pointing to the calling class

Anatomy of a class

```
class Tracker{
public:
   Tracker();
   Tracker(unsigned int nlayers);
   ~Tracker();
   void DoSomething();
protected:
   unsigned int nlayers;
   double gasTemperature;
```

class keyword, class name
access modifier
Constructor
Constructor
Destructor
Method
access modifier
data member
another data member
Variables inside a class are called member variables or simply members

They should always be protected (abstraction and encapsulation)

How to change them?

Anatomy of a class

```
class Tracker{
public:
  Tracker();
  Tracker(unsigned int nlayers);
  ~Tracker();
  void DoSomething();
  double GetGasTemperature(){return gasTemperature;};
  void SetGasTemperature(double T)
                  {gasTemperature =T;};
protected:
  unsigned int nlayers;
  double gasTemperature;
};
```

class keyword, class name
access modifier
Constructor
Constructor
Destructor
Method
Getter method
Setter method

access modifier
data member
another data member

Members are accessed through getters and setters

They are often implemented in the header (which is not nice, but extremely common)

The interface as a contract

Your interface is a contract with the compiler and the user about what your class is doing

- You are restricting yourself on purpose
- This is also where you put documentation for the user
- Especially if you make any assumptions on inputs
- Restrict yourself even further: Use "const"

Use the compiler to detect your errors before even running the program

Program defensively

Use const!

```
class Tracker{
public:
   Tracker();
   Tracker(unsigned int nlayers);
   ~Tracker();
   void DoSomething(vector<double> & const input);

   double GetGasTemperature() const
                  {return gasTemperature;};
   void SetGasTemperature(double T)
                  {gasTemperature =T;};
protected:
   const unsigned int nlayers;
   double gasTemperature;
   const double * p1;

   double * const p2;


   const double * const p3;
};
```

const has many uses, always implies that something does not change

DoSomething will not change input

GetGasTemperature will not change any class members

Changes a class member, so no const

nlayers will not change

the integer stored at p1 will not change

p2 will always point at the same double (which can change)

neither the pointer nor the value will change

Use const: Initialisation

```
In the .cpp file:

Tracker::Tracker(unsigned int _nlayers):
  nlayers(_nlayers){

  <Constructor Code>

}
```

All const members have to be initialized before any method (including the constructor) is run - do this just before the constructor starts

Making and checking assumptions

```
In the .h file
...
//Constructor: nlayers is the number of layers
// nlayers has to be larger than 2 and smaller than
// 100, otherwise some horrible things happen
Tracker(unsigned int nlayers);



In the .cpp file:
include <cassert>
...
Tracker::Tracker(unsigned int _nlayers):
  nlayers(_nlayers){

  assert(nlayers > 2 && nlayers < 100);

  <Constructor Code, where horrible things happen if
  nlayers has the wrong value>

}
```

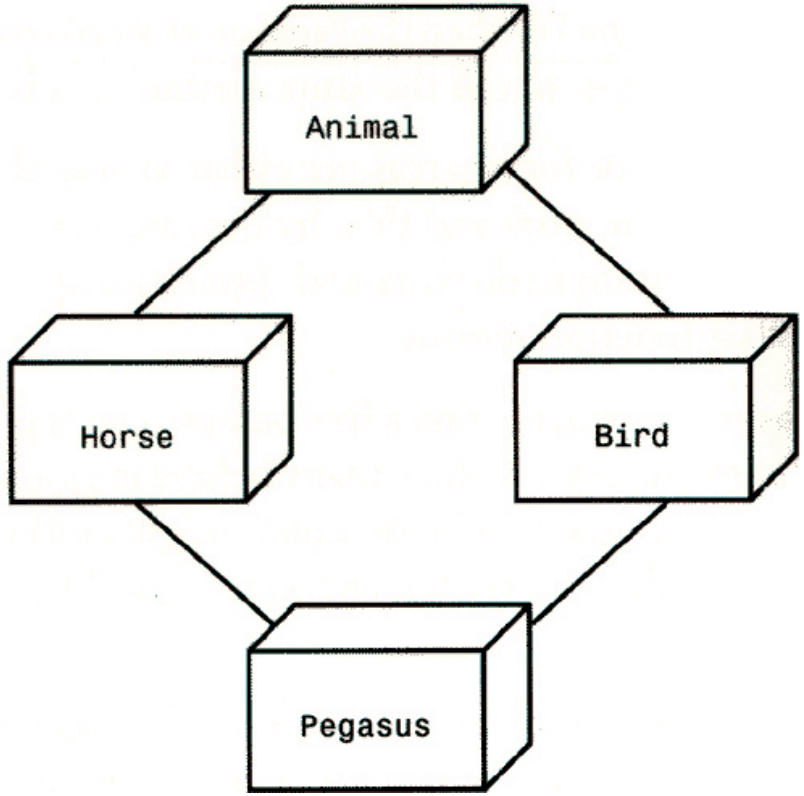
At some point you will be making assumptions - document them and check them

If you know that nlayers has to be positive, declare it as unsigned

Allows the use of assert

Will stop the program and print where this happened if the condition evaluates to false - not great error handling, but much better than a random crash

What about inheritance?



- Inheritance is basing one class on another; Cow and Camel are Animals
  TrackerHit and FibreHit are Spacepoints
  etc.
- Inheritance allows for easy code reuse
- Inheritance allows for multiple implementations of an interface
- Inheritance breaks encapsulation
- You can do a lot with inheritance, not all of it is sensible...
- As always: Think before you do...

Inheritance in code

```
class BaseClass{
public:
  Baseclass();
  ~Baseclass();
  SomeMethod();
protected:
  double someMember;
};


class DerivedClass: public BaseClass {
public:
  DerivedClass();
  ~DerivedClass();
};

In the main function:

BaseClass * myObject = new DerivedClass();
myObject->SomeMethod();
```

Declare a base class as usual...

Declare the derived class Inheritance comes with an access modifier - shoud the Methods of BaseClass be visible to the users of DerivedClass?

I can assign a DerivedClass to a BaseClass because derived Class is a BaseClass Object.

Inheritance in code

```
class BaseClass{
public:
  Baseclass();
  ~Baseclass();
  SomeMethod();
protected:
  double someMember;
};


class DerivedClass: public BaseClass {
public:
  DerivedClass();
  ~DerivedClass();
  AnotherMethod();
};

In the main function:

BaseClass * myObject = new DerivedClass();
myObject->SomeMethod();

myObject->AnotherMethod();
```

Declare a base class as usual...

Declare the derived class
Inheritance comes with an access modifier - shoud the Methods of BaseClass be visible to the users of DerivedClass?

I can assign a DerivedClass to a BaseClass because derived Class is a BaseClass Object. This will not work!

Virtual functions

```
class A{
public:
   A();
   virtual ~A();
   virtual Do()
      {cout << A;};
};

class B: public A{
public:
   B();
   ~B();
   Do()
      {cout << B;};
};

class C: public A{
public:
   C();
   ~C();
   Do()
      {cout << C;};
};
```

Use the virtual keyword in the base class and C++ will use the right Do() function

If your class has virtual functions, it should also have a virtual destructor (otherwise bad things may happen down the line)

```
A * a = new A();

A * b = new B();

A * c = new C();

a->Do();

b->Do();

c->Do();
```

A base class object

A derived object, pointer to A

A derived object; pointer to A

What is the output?

Virtual functions

```
class A{
public:
  A();
  virtual ~A();
  virtual Do()
    {cout << "A";};
};

class B: public A{
public:
  B();
  ~B();
  Do()
    {cout << "B";};
};

class C: public A{
public:
  C();
  ~C();
  Do()
    {cout << "C";};
};
```

Use the virtual keyword in the base class and C++ will use the right Do() function

If your class has virtual functions, it should also have a virtual destructor (otherwise bad things may happen down the line)

```
A * a = new A();

A * b = new B();

A * c = new C();

a->Do();

b->Do();

c->Do();

> ABC
```

A base class object

A derived object, pointer to A

A derived object; pointer to A

What is the output?

The runtime knows which function to call

This works via the "Virtual Function Table"

What if I do not want to write an implementation of Do() for A?

Virtual functions

```
class A{
public:
  A();
  virtual ~A();
  virtual Do()=0;
};

class B: public A{
public:
  B();
  ~B();
  Do()
    {cout << "B";};
};

class C: public A{
public:
  C();
  ~C();
  Do()
    {cout << "C";};
};
```

No implementation of Do() for A;
A::Do() is a purely virtual function

```
A * a = new A();

A * b = new B();

A * c = new C();


b->Do();

c->Do();

> BC
```

This will now fail - A is now an "abstract base class"

This is a great tool for specifying interfaces

We will use this in the exercises...

Initializing the base class

```
class A{
public:
  A(int n);
  virtual ~A();
  virtual Do()=0;
};

class B: public A{
public:
  B(int n);
  ~B();
  Do()
    {cout << "B";};
};


B implementation:

B::B(int n):A(n){
  <whatever else needs to be done>
}
```

Assume the base class constructor requires an argument

We would like to feed that through from the B constructor - how to implement?

You can explicitly call the constructor of the base class just before the derived constructor starts

Copying and assigning objects

Some care has to be taken whenever objects are copied or assigned

- Usually you do not actually need to copy and assign them, hand around a pointer instead
- The compiler will per default copy/assign all member variables (a shallow copy) - this is fine if there are no pointers...
- If you need to copy things that are pointed to, you have to provide a deep copy yourself...

This is fairly advanced stuff - no need to know this by heart, but need to know it is around...

Why copy constructors and such

```
class A{
public:
   A();
   ~A();
   SetName(string x){*name = x};
   string GetName() const {return *name};
protected:
   string * name;
};

Use case:

A myA;
myA.SetName("Alice");

A myOtherA = myA;

myOtherA.SetName("Bob");

cout << myA.GetName() << endl;
```

A class with a pointer member

Create an object of class A
Name it

Assign it...

Name the new object

What will be the output?

Why copy constructors and such

```
class A{
public:
   A(){new string;};
   ~A();
   SetName(string x){*name = x};
   string GetName() const {return *name};
protected:
   string * name;
};

Use case:

A myA;
myA.SetName("Alice");

A myOtherA = myA;

myOtherA.SetName("Bob");

cout << myA.GetName() << endl;

> "Bob"
```

A class with a pointer member

Create an object of class A
Name it

Assign it...

Name the new object

What will be the output?

Not necessarily what you want...

```
class A{
public:
   A(){name = new string();};
   A(const A& copy_from_me);
   A& operator= (A const& assign_from_me);
   ~A();
   SetName(string x){*name = x};
   string GetName() const {return *name};
protected:
   string * name;
};

Implementation:
A::A(const A& copy_from_me){
   name = new string();
   *name = copy_from_me.GetName();
}
A& A::operator= (A const& assign_from_me){
   if (this == &f) return *this;
   *name = assign_from_me.GetName();
   return *this;
 }
```

A class with a pointer member

Copy constructor
Assignment operator

Copy constructor - do things you would also do in constructor, then copy...

Assignment operator
Check for self-assignment
Copy what needs to be copied

Singletons

You want to make sure there is only one instance of the random number generator

- Make it a singleton!
- This is an useful and frequent use of a design pattern

- How?

Singleton implementation

```
class MySingleton{
public:
  static MySingleton& Instance()
  {
    static MySingleton singleton;
    return singleton;
  };

// Other non-static member functions

private:
  MySingleton() {};
  CMySingleton(const CMySingleton&);
  CMySingleton& operator=(const CMySingleton&);

};
```

There is no public constructor
The instance method is static, i.e. independent of an instance of MySingleton

Private constructor
Private copy constructor
Private assignment operators

Static creates something once per class (as opposed to once per object)

Other use: count objects of a specific class

Operator overloading

```
Vector3 v;
Vector3 r;
Matrix3x3 b;




r = A.multiply(v);

r = A*v;
```

Assume you create some vector or matrix classes...

and you want to calculate

r = A v

is ok, but it would of course be nicer, if you could write

You can, as C++ allows for operator overloading, i.e defining your own versions of +, -, *, /, << etc. - we have already seen the example for =

Operator overloading: Arithmetic

```
class Vector3{
public:
   Vector3;
   ~Vector3;
   Vector3& operator+=(const Vector3& rhs);
...
};
```

Start with the compound assignment operators +=, -=, *=, /=

Compound addition - implementation will be similar to the assignment operator (with the addition added...)

Operator overloading: Arithmetic

```
class Vector3{
public:
   Vector3;
   ~Vector3;
   Vector3 & operator+=(const Vector3& rhs);
   const Vector3 operator+(const Vector3& other);
...
};


Implementation:

const Vector3 Vector3::operator+(const Vector3
                                 &other) const {
    return Vector3(*this) += other;
}
```

Start with the compound assignment operators +=, -=, *=, /=

Now add the addition

Implementation then can make use of the += operator

Exercise: What happens here?

Use the standard library!

The C++ standard library provides very well written classes/templates for standard tasks
Use them!

- Containers:
  - Vector (dynamic array)
  - List (doubly linked list)
  - Map (associative array)
  - Set
  - Queue and Stack
- Strings
- Numerics
  - Complex numbers
  - Valarrays (vector with math)
- Algorithms: Do things with containers, e.g. for_each, sorting, searching...

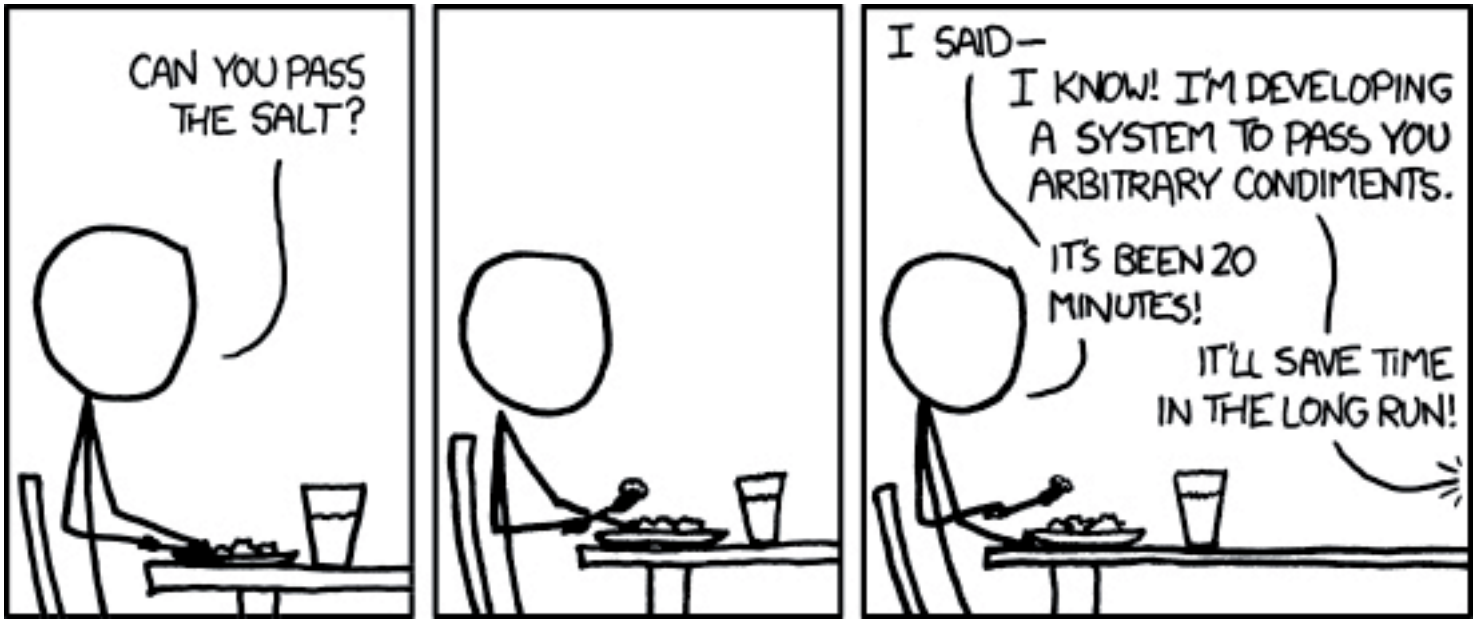
Fundamental Laws

Laws of nature also apply to code

- 2nd Law of Thermodynamics:
  Your code will get messier over time, if you do nothing about it
- Murphy's Law:
  If you can do it wrong, you will do it wrong
- Hofstadter's Law:
  It will take longer than you think, even if you take into account Hofstadter's Law

Generalities

- The best code is the one you newer write
- The compiler is your friend
- Program defensively
- If you do not test it, it will not work
- Make your code readable
- Your time is the most expensive resource (besides my time)
- Doing it right will save you time in the long run



I find that when someone's taking time to do something right in the present, they're a perfectionist with no ability to prioritize, whereas when someone took time to do something right in the past, they're a master artisan of great foresight. (http://www.xkcd.com/974/)

When collaborating:

- Use clean, small interfaces
- Do change them only in emergencies
- Document what you do, this includes:
  - Documenting your interface - tell the user what your methods do - there are nice tools like Doxygen, that create nice documents from code
  - Documenting your assumptions
  - Documenting your code such that a reader knows what your intentions where and why you do what you do
- Use a source control system (we all like git, but this is often fixed by the collaboration to cvs, svn etc.) - this is also very useful for e.g. your thesis
- Talk with your collaborators, do code reviews

You only become a better programmer by actually programming

- Write code
- Get it to work
- Have a thorough look at it - what would you have done differently?
- Do it differently!

This is the all important step: The technical term is refactoring and it means making your code nicer without changing the functionality

- We all should do this much more often
- You will get a feeling of when it is needed... - this is called "code smell"

Code Smell (from Wikipedia)

- Duplicated code: identical or very similar code exists in more than one location.
- Long method: a method, function, or procedure that has grown too large.
- Large class: a class that has grown too large. See God object.
- Too many parameters: a long list of parameters in a procedure or function make readability and code quality worse.
- Feature envy: a class that uses methods of another class excessively.
- Inappropriate intimacy: a class that has dependencies on implementation details of another class.
- Refused bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class.
- Lazy class / Freeloader: a class that does too little.
- Contrived complexity: forced usage of overly complicated design patterns where simpler design would suffice.

Code Smell (from Wikipedia) cont.

- Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture.
- Excessively short identifiers: the name of a variable should reflect its function unless it's obvious.
- Excessive use of literals: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts where possible, to facilitate localization of software if it is intended to be deployed in different regions.

Symptoms for the need to refactor

More signs for a need to refactor:

- You copy and paste blocks of code longer than three lines
- Relationships between classes stop making semantic sense
  (a histogram "is" a font)
- Your class inheritance hierarchy does hinder instead of help you
- You need run-time type ID
- You have to make too many assumptions
- Using your code becomes awkward

Refactoring techniques (Wikipedia)

- Techniques that allow for more abstraction
  - Encapsulate Field – force code to access the field with getter and setter methods
  - Generalize Type – create more general types to allow for more code sharing
  - Replace type-checking code
  - Replace conditional with polymorphism
- Techniques for breaking code apart into more logical pieces
  - Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
  - Extract Class moves part of the code from an existing class into a new class.
- Techniques for improving names and location of code
  - Move Method or Move Field – move to a more appropriate Class or source file
  - Rename Method or Rename Field – changing the name into a new one that better reveals its purpose
  - Pull Up – move to a superclass
  - Push Down – in OOP, move to a subclass

No Summary

Coding is a craft as many others

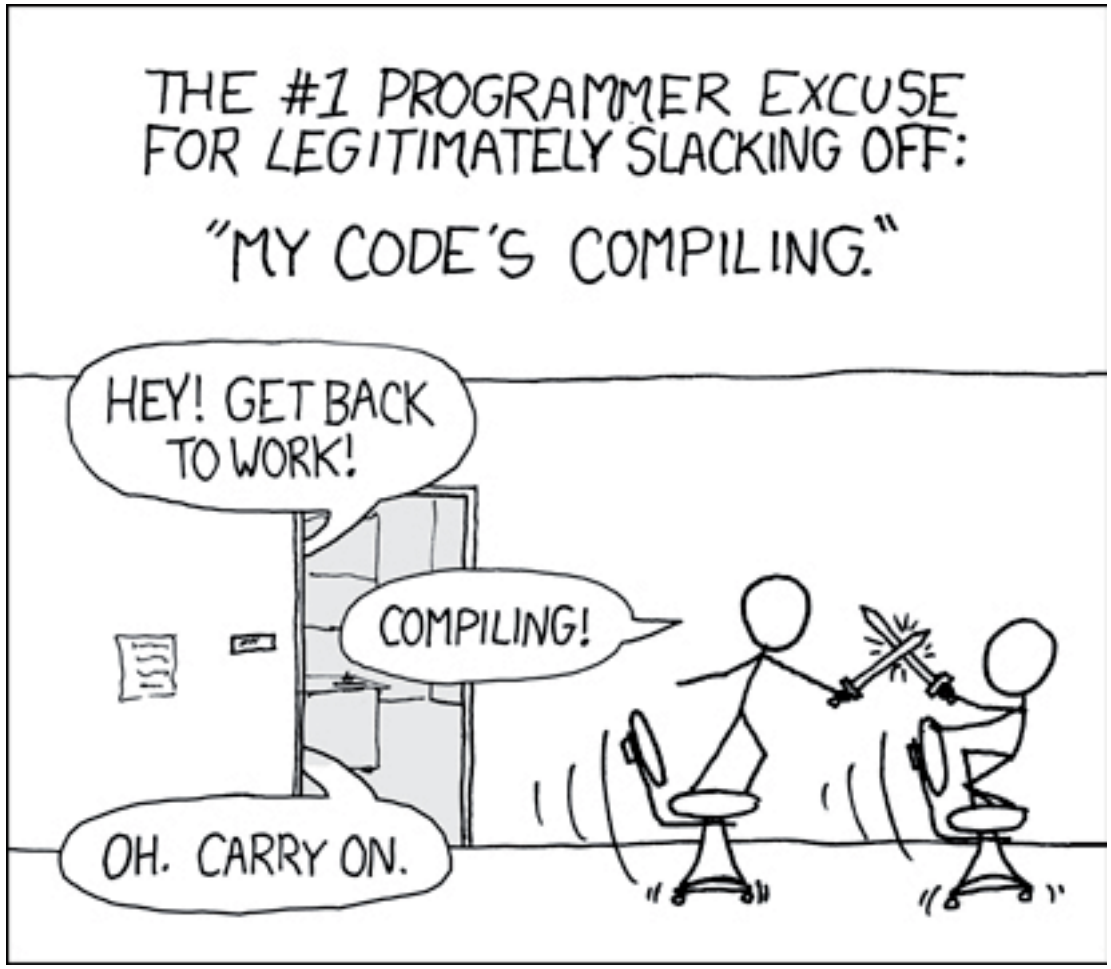
- Knowing some basics helps
- But only practice makes perfect

Producing working software in large collaborations is an extremely difficult problem

- There is no one safe method
- But we do it all the time in HEP
- Most of us were never properly trained
- Our software shows this
  (but most commercial stuff is no better)

Try to be better! Code! Learn!

And have some fun in the process...



http://xkcd.com/303/