Numerical Software Engineering 101/201

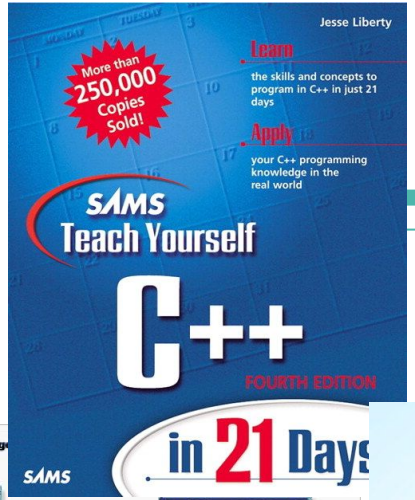
Scientific Software Club 2/13/17

Papers

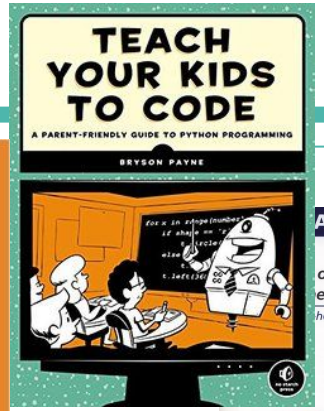
- *Best Practices for Scientific Computing*, Wilson et al.
- *Good Enough Practices in Scientific Computing*, Wilson et al.
- *Barely Sufficient Software Engineering: 10 Practices to Improve your CSE Software*, Heroux and Willenbring

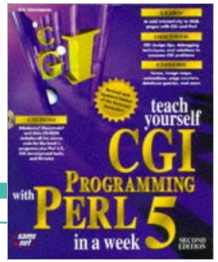
Misconception: Coding is unimportant! It’s not like I’m a software engineer…

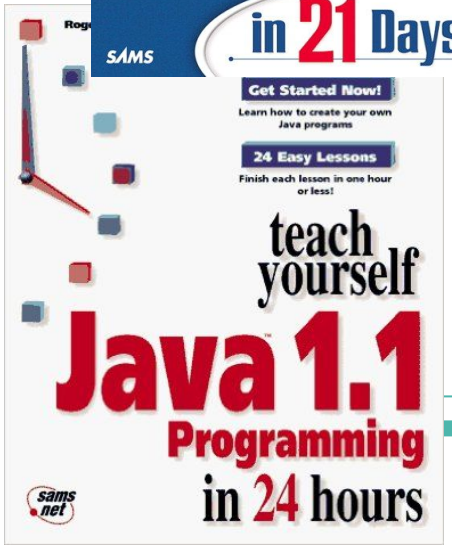
(The crucial part is getting the numerical algorithm, proper data, good results, etc)

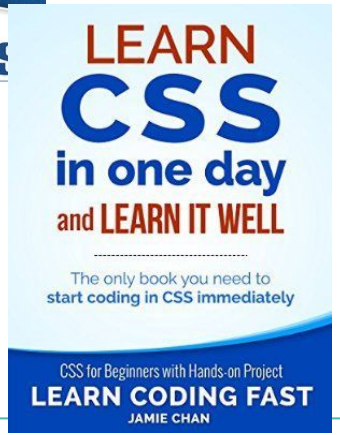

Jesse Liberty
More than 250,000 Copies Sold!
Learn
the skills and concepts to program in C++ in just 21 days
Apply
your C++ programming knowledge in the real world
SAMS
Teach Yourself
C++
FOURTH EDITION
in 21 Days
SAMS


coursera
3
Programming Lessons


TEACH YOUR KIDS TO CODE
A PARENT-FRIENDLY GUIDE TO PYTHON PROGRAMMING
BRYSON PAYNE

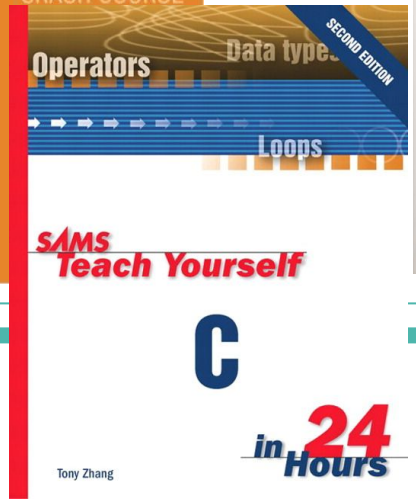

teach yourself
CGI
PROGRAMMING
with PERL 5
in a week

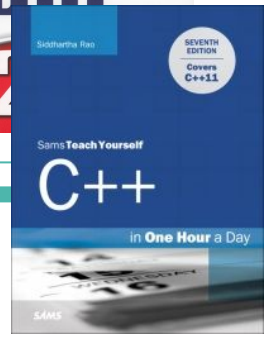

Get Started Now!
Learn how to create your own Java programs
24 Easy Lessons
Finish each lesson in one hour or less!
teach yourself
Java 1.1
Programming
in 24 hours
sams net

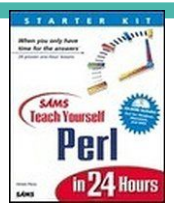

LEARN
CSS
in one day
and LEARN IT WELL
The only book you need to start coding in CSS immediately
CSS for Beginners with Hands-on Project
LEARN CODING FAST
JAMIE CHAN


MySQL
CRASH COURSE
SAMS


ARTER KIT
only have
e answers
hour lessons
SAMS
Teach Yourself
COBOL
CD-ROM Includes Fujitsu COBOL V3 Starter Set
Thane Hubbell
SAMS


Operators
Data types
Loops
SECOND EDITION
SAMS
Teach Yourself
C
in 24 Hours
Tony Zhang


Siddhartha Rao
SEVENTH EDITION
Covers C++11
Sams Teach Yourself
C++
in One Hour a Day
SAMS


STARTER KIT
SAMS
Teach Yourself
Perl
in 24 Hours

The (Relative) Truth: Coding is an important part of research and a skill that takes years to hone

(*Teach Yourself Programming in Ten Years* by Peter Norvig)

Topics

- Code Level Management
- Data Management
- Directory Level Management
- Project Level Management
- Working with Others
- Documentation and Technical Writing

Code Level Management

Comment Succinctly (Design, Not Mechanism)

```
double AreaRectangle(double x, double y){
    /* AreaRectangle calculates the area of a
    rectangle with dimensions x and y */

    /* Return -1 if bad input*/
    if(x < 0 || y < 0){
        printf("x and y must be positive numbers");
        return -1;
    }
    /* Return the product of x and y */
    return x*y;
}
```

Comment Succinctly

```
/*
runN4SID runs the system identification algorithm n4sid

~~~~INPUT~~~~
data: N x K time domain signal, N = number samples, K = dimension of data
p: includes measurement frequency in Hz, model size to fit
~~~~~~~~~~~~~

~~~OUTPUT~~~
Fitted system model, saved in results folder as system.csv
~~~~~~~~~~~~~
*/
void runN4SID(double data, params p){
        ...
}
```

Name Intelligently

- Fits in with earlier example, but having descriptive function and variables is extremely important
- A headache for numerical calculations
  - Generally, code might be ugly, but make sure function is named well!

Name Variables Intelligently

```
void calcStuff(...){
    A = getMatrix(...);
    [U, D, V] = svd(A);
    [X, Y] = getData(...);
    [E, Z] = eig(X*A*Y);
    w = getWeights...();
    [S, N] = sumEV(W, w);
    B = convolveMatrix(A, N, S)
    I = [ identity(N); identity(N)];
    C = I*B + I*A;
    [Q, R, P] = qr(C);
    ....
    (you get the point)
}
```

```
class Central2D {
public:
   Central2D(float w, float h,     // Domain width / height
             int nx, int ny,     // Number of cells in x/y (without ghosts)
             float cfl = 0.45) :  // Max allowed CFL number
       nx(nx), ny(ny),
       nx_all(nx + 2*nghost),
       ny_all(ny + 2*nghost),
       dx(w/nx), dy(h/ny),
       cfl(cfl) {}

   static constexpr int nghost = 3;   // Number of ghost cells
   const int nx, ny;          // Number of (non-ghost) cells in x/y
   const int nx_all, ny_all;  // Total cells in x/y (including ghost)
   const float dx, dy;         // Cell size in x/y
   const float cfl;            // Allowed CFL number
   // Array accessor functions
   int offset(int ix, int iy) const { return iy*nx_all+ix; }

   float& u1(int ix, int iy) { return u1_[offset(ix,iy)]; }            // Solution values
   float& u2(int ix, int iy) { return u2_[offset(ix,iy)]; }
   float& u3(int ix, int iy) { return u3_[offset(ix,iy)]; }
   float& f1(int ix, int iy) { return f1_[offset(ix,iy)]; }            // Fluxes in x
   float& f2(int ix, int iy) { return f2_[offset(ix,iy)]; }
   float& f3(int ix, int iy) { return f3_[offset(ix,iy)]; }
   float& g1(int ix, int iy) { return g1_[offset(ix,iy)]; }            // Fluxes in y
   float& g2(int ix, int iy) { return g2_[offset(ix,iy)]; }
   float& g3(int ix, int iy) { return g3_[offset(ix,iy)]; }
   float& ux1(int ix, int iy) { return ux1_[offset(ix,iy)]; }           // x differences of u
   float& ux2(int ix, int iy) { return ux2_[offset(ix,iy)]; }
   float& ux3(int ix, int iy) { return ux3_[offset(ix,iy)]; }
   float& uy1(int ix, int iy) { return uy1_[offset(ix,iy)]; }           // y differences of u
   float& uy2(int ix, int iy) { return uy2_[offset(ix,iy)]; }
   float& uy3(int ix, int iy) { return uy3_[offset(ix,iy)]; }
   float& fx1(int ix, int iy) { return fx1_[offset(ix,iy)]; }           // x differences of f
   float& fx2(int ix, int iy) { return fx2_[offset(ix,iy)]; }
   float& fx3(int ix, int iy) { return fx3_[offset(ix,iy)]; }
   float& gy1(int ix, int iy) { return gy1_[offset(ix,iy)]; }           // y differences of g
   float& gy2(int ix, int iy) { return gy2_[offset(ix,iy)]; }
   float& gy3(int ix, int iy) { return gy3_[offset(ix,iy)]; }
   float& v1(int ix, int iy) {return v1_[offset(ix,iy)]; }            // Solution values at next
   float& v2(int ix, int iy) {return v2_[offset(ix,iy)]; }
   float& v3(int ix, int iy) {return v3_[offset(ix,iy)]; }

   // Diagnostics
   void solution_check();
   // Array size accessors
   int xsize() const { return nx; }
   int ysize() const { return ny; }

   // Read / write elements of simulation state
   float&       operator()(int i, int j) {
       return u1_[offset(i,j)];
   }

   const float& operator()(int i, int j) const {
       return u1_[offset(i,j)];
   }
   // Wrapped accessor (periodic BC)
   int ioffset(int ix, int iy) {
       return offset( (ix+nx-nghost) % nx + nghost,
                      (iy+ny-nghost) % ny + nghost );
   }

   float& uwrap1(int ix, int iy)  { return u1_[ioffset(ix,iy)]; }
   float& uwrap2(int ix, int iy)  { return u2_[ioffset(ix,iy)]; }
   float& uwrap3(int ix, int iy)  { return u3_[ioffset(ix,iy)]; }


   void run(float tfinal);
   // Call f(Uxy, x, y) at each cell center to set initial conditions
```

Decompose Programs into Functions

- Try to keep functions short
- Modularity makes code base more flexible, more easily modifiable
- Saves lines of code
- Practically speaking, humans can only remember a few things at a time!

Decomposing Programs into Functions

```
void calcStuff(...){
    Node root;
    ...
    Node data;
    ...
    bool checkchild = 0;
    for(i = 0; i < root.numchildren; i++){
        if(root.child[i] == data){
            checkchild = 1;
        }
    }
    ...
}
```

```
void calcStuff(...){
    Node root;
    ...
    Node data;
    ...
    bool checkchild = isChild(root, data);
    ...
}
```

Eliminate Duplication

```
double calcValues(...){
    ...
    X = getvalue(...);
    return X;
}

double calcValuesFilter(...){
    ...
    X = getvalue(...);
    X = filter(X);
    return X;
}
```

VS

```
double calcValues(... , bool Filter){
    ...
    X = getvalue(...);
    if( Filter == true){
        X = filter(X);
    }
    return x;
}
```

Keep Semantics Consistent

```
void scaleVec(vec v, double n){
    ...
    }

void filterEigenVecs(Matrix M){
    ...
    }

void find_all_keys(keys K){
    ...
    }
```

VS

```
void scaleMatrix(double n, matrix m){
    ...
    }

void filterEigVals(Matrix M){
    ...
    }

void findAllKeyrings(rings R){
    ...
    }
```

Use Data Structures (If necessary)

```
void doStuff(...
double timestep, int size...
date d, int dimx, int dimy...
int numthreads){
...
}
```

VS

```
void doStuff(metatdata d){
    ...
}

class metadata{
    double timestep;
    int size;
    date d;
    int dimx;
    int dimy;
    int numthreads;
}
```

Incremental Changes

- Emphasized in two papers
  - Decompose a large task into small components
  - Test the correctness of components
- Programmers are most productive working in small steps
  - + Course Correction

Defensive Programming

- Assert (or Try/Catch)
- Unit Testing
  - What if no “useful” unit tests?
  - Numeric Unit Tests
- Automated Testing and Continuous Integration
  - (to be covered in the future)

Abstractions

- Computer Systems Researchers often talk about getting the right “abstractions”
  - “Abstraction” decrease the complexity of your software by making the low-level details hidden from the user
- Defining a convenient way to interact with your code base is hard!
  - Takes practice… cannot be quantified
  - What do you expose to the user (one of which will surely be yourself)?

Data Level Management

Save Raw and Intermediate Data

- Raw data D >> Intermediate Forms >> Result (yes or no)
  - You don’t just want to save the yes/no!
- Save Raw and Intermediate Forms
  - Saves time, extra processing, etc

Format Data Well

- Create data you wish to see in the world
  - Neatly labeled columns, information on format, etc
  - Important, especially if your data format changes down the road
- Space is cheap!
  - One variable per column, one observation per row, etc
  - Don’t cram!

Manage Your Metadata

- What is “Metadata”?
  - In short: Data about Data Set
- Might include date produced, units, etc
- You’ll need it later!

Publish Data

- (If you think others might want to use it)
- “Your data is as much a product of your research as the papers you write”
- Figshare, Dryad, Zenodo

Directory Level Management

Directory Names

- Your project should NOT be named “foo” or “a”
- Subdirectories should also be descriptive
  - Documentation in “docs”
  - Source in “src”
  - Scripts in “bin”
  - Etc…
- Should include a “data” and “results” folder
  - Make a distinction between what goes in each folder, as your results will surely contain data!
  - Idea: every output goes in “results”, every input goes in “data”

Directory Names

- ❏ README
- ❏ LICENSE
- ❏ Tests
  - ❏ testSightings.py
- ❏ data
  - ❏ birdcount.csv
- ❏ doc
  - ❏ notebook.md
  - ❏ changelog.txt
- ❏ results
  - ❏ summarized-results.csv
- ❏ src
  - ❏ Sightings.py

Subdirectories (Don’t make too many)

- src
  - helpers
    - datastructs
      - graph
        - graphsearch
          - methods
            - dfs.py

Don’t Repeat Previous Work

- Use external libraries as much as possible
  - Optimized code and saves development time
- Use google, github, cppreference, etc

Project Level Management

Version Control

- Discussed Earlier This Semester
- Git, CVS, Mercurial, etc
  - Git preferred (Github, Bitbucket)
- Commit often, Commit early
- Don’t add large data dumps/files!
  - Makes version control slow, impractical
  - We will discuss later in semester how to manage this stuff

Adding Features, Refactoring

- Add features incrementally
  - Constantly check correctness
  - Don’t expect to add 1k+ lines and have your code work the first time
- Refactoring is a natural part of coding
  - Don’t avoid it
  - End up with bloated code

To use an IDE or not to use an IDE...

- I'm not sure!
  - What if like Microsoft Visual Studio, Eclipse, PyCharm?
  - Problem: code should be accessible to everyone
  - Getting libraries integrated into an IDE can be painful
    - For numeric libraries, even more annoying
    - Software makes this easier e.g. Intel Parallel Studio XE, Nividia NSIGHT, etc
  - If you're prototyping and know IDE's debugging and profiling tools well, why not
  - Mismatch between IDE environment and deployment environment

Issue-Tracking Software

- Common Mistake
  - “I need to refactor A, B, C and debug I, J, K
  - (One seminar and one nap later) “What was I supposed to do again?”
- Many out there (Wikipedia lists ~ 50)
  - Bugzilla, Apache Bloodhound, Planbox, etc etc

Apache™ Bloodhound

Login Preferences Help/Guide

Search anything. Try #EF-492. Search Create Ticket

Wiki Tickets Apps

Available Reports Custom Query

{2} Active Tickets by Version (106 matches)

This report shows how to color results by priority, while grouping results by version.

Last modification time, description and reporter are included as hidden fields for useful RSS export.

Max items per page
100
Update

Results (1 - 100 of 106) 1 2 →

	Ticket	Summary	Component	Version	Type	Owner	Status	Created
empty	#161	Install fails when trac-hacks is down	installer		defect	gjm	assigned	Aug 16, 2012
	#146	Inline editing of objects	dashboard		enhancement	olemis	accepted	Jul 17, 2012
	#151	Users should be able to watch objects	dashboard		enhancement	nobody	new	Jul 18, 2012
	#156	Local copy of bloodhound part of Apache repo for browse functionality	siteadmin		task	nobody	new	Aug 7, 2012
	#159	Incorporate Trac 1.0 into vendor branch	trac core		task	nobody	new	Aug 11, 2012
	#27	Request that the ThemeEngine alters their implementation to work for Trac 0.13	plugins		task	nobody	assigned	Apr 4, 2012
	#53	Update styling for login page to bootstrap	dashboard		task	olemis	accepted	May 1, 2012
	#194	Examine workflow for Bloodhound site	siteadmin		task	nobody	new	Aug 31, 2012
	#101	Generic conversion for admin table listings and add forms	dashboard		defect	nobody	new	Jun 18, 2012
	#181	Make arrows for reply to and follow-up comments point in correct direction on ordering change	dashboard		defect	nobody	new	Aug 29, 2012
	#47	Investigate enhanced selection of active ticket module (initially for quick ticket)	dashboard		enhancement	nobody	new	Apr 30, 2012
	#48	Split QuickCreateTicketDialog for more flexible approach	dashboard		enhancement	nobody	new	Apr 30, 2012
	#60	Alert text width in quick ticket.	dashboard		enhancement	nobody	new	May 3, 2012
	#67	Highlight tickets outputted by reports	ui design		enhancement	olemis	accepted	May 14, 2012

Note See TracReports for help on using and creating reports.

Download in other formats:

RSS Feed | Comma-delimited Text | Tab-delimited Text | SQL Query

Powered by Apache™ Bloodhound. Standing on the shoulders of Trac 0.13dev-r1360726.

Working with Others

Industry vs Academia

- In industry, a group of experienced engineers is often assigned to manage a single piece of software
- In academia, a single person might manage multiple pieces of software

Getting a Second Look

- Just as research ideas need a second look, so does a potential code base
- Pair Programming is extremely beneficial
  - Could be a problem if you're the only one working on a project
- Coding with others ultimately makes you a better programmer

Documentation and Technical Writing

Create Barely Sufficient Documentation

- Somewhat covered earlier last semester
  - Documentation generation via Sphinx, Doxygen, etc
- You are writing the documentation for yourself as well as others!

Document All Work You've Done

- Not just the code you plan to release; code you've written but not used, ideas you've tried (both successful and unsuccessful), etc

Reports and Papers

- Writing a paper or technical report? Put it under version control as well
- Formal Approach: Treat paper/report writing as programming.
- Save you time and effort town the road

Figures

- One script per figure
- Don’t manually change parameters; input them into functions
- Automation
  - Don’t be tempted to manually adjust window size and click the “save as” button in MATLAB

Conclusions

Conclusions: Takeaways

- Following software engineering best practices saves development time, headaches, and user-friendliness
- Developing (and maintaining) software is hard!

Conclusions: Questions

- Why put in all this effort if no one else is going to use my code?
- Considering the time spent improving non-essential parts of my code, will the time saved from following best practices be greater than the extra development time invested?