Programming
in an undergraduate CS curriculum

Bjarne Stroustrup
Texas A&M University
www.research.att.com
www.stroustrup.com/Programming
Overview

• Aims
• Curriculum issues
• A first programming course
  – Details
  – Experience
My background

- Industrial research (24 years)
- Industrial research management (7 years)
- Academic research and graduate education (6 years)
- Freshman programming (4 years)
- Language design, implementation, and large scale use
Problems

Famous CS professor (proudly): 
“*We don't teach programming; we teach computer science*”

Industrial manager: 
“They can’t program their way out of a paper bag”
Problems

Another CS professor: “I never code”

Another industrial manager: “We don’t hire CS graduates; it’s easier to teach a physicist to program than to teach a CS graduate physics”
Problems

CS professor (about a student):
“*He accepted a job in industry*”

Another CS professor:
“*Sad; he showed so much promise*”
Why do I bother with beginning programming?

- We (the computing community) have a serious problem
  - Many (good) students really can’t program their way out of a paper bag
    - Often, their education encourages them to think of programming as a menial task to be avoided
  - Large sections of industry despair and focus on tools and management techniques for exploiting low-level labor
    - Leading to bloated, expensive, inflexible buggy systems
  - Costs are increasing
    - Development, maintenance, testing
Is it really that bad? (yes)

• “Things” have improved:
  – My computer doesn’t crash every day
  – I haven’t had any bills for $0.00 lately

• But the structure of code is often really bad
  – Million line programs are quite common
  – Layers upon layers of checking
    • yet still incomplete and insecure
  – Code bloated so as to defy analysis/understanding
  – Massive testing
    • hard to do completely and systematically because of lack of understanding
  – Year-long beta programs
    • Yet still bugs and usability problems
  – “32% of all projects succeeding which are delivered on time, on budget, with required features and functions” – Jim Johnson, chairman of The Standish Group

• No, I don’t offer a miracle cure
What are the right questions?

• What should we teach our future software developers?
  – Why?
  – How?
  – “future software developers”?

• Somebody else’s problems
  – “computer literacy”
  – “computers for poets”
  – “minimal skills needed for web development using XYZ™ v2007 tools”
  – Web designer
  – …
Software and curriculum

• Assume a set of liberal arts and general science classes
  – We need history, literature, economics, physics, math, etc.

• Assume a core CS curriculum
  – We need algorithms, data structures, machine architecture, “systems”, …

• Make sure there is room for software
  – Programming, languages, basic project management, etc.

• Make sure there are plenty of opportunities to explore further CS topics
  – Security, parallel computation, human-computer interfaces, theory, graphics, compilers, …
Computer science

• “Computer science is not programming!”
  – Correction: not just programming
  – CS involves programming in critical ways

• “Software development is not computer science”
  – Correction: not just computer science
  – Software development (under any name) needs (parts of) CS

• I consider CS inherently an applied field
  – We can argue whether it is science or engineering
Curriculum design strategy

• Work backwards from requirements
  – After degree: ready for industry
    • or grad school: masters (my ideal)
    • or grad school: Ph.D. (secondary concern)
  – After two years: ready for internship
  – After the first programming class: ready for first real-world program (assuming supervision)
Software curriculum

• First two years
  – First programming course
    • using C++
  – Data structures and algorithms
  – Languages
    • using at least two that’s not C++
  – Software design studio
    • using at least two languages

• Second two years
  – Specialization (“software” is one choice)
  – Capstone project

• Repeated presentation with increasing integration of key topics
Freshman programming class criteria

• Work backwards from what a student will need to start a first real project (under supervision)
  – A professional attitude
  – Objects, values, types, and control structures
  – Error handling and resource handling
  – Simple numerical calculation and string manipulation
  – Keyboard/screen and file I/O
  – Algorithms and data structures
  – Procedural, object-oriented, and generic programming
  – Use of libraries
  – Basic graphics/GUI
  – Basics of low-level programming (C style)
    • Arrays and pointers
First language choice – C++

- Direct industry demand for “more system knowledge”
- Direct engineering college demand for C/C++
- Desire for “real-world” language and tools
- Desire to demonstrate a broad set of techniques and facilities
- Desire for portability
  - Windows, Mac, Linux, Unix, embedded platforms
- Desire to cover “higher level” techniques
  - OOP, generic programming, invariants, resource management, …
- C++ strengths
  - Wide availability, wide industrial use, formal standard, non-proprietary, supports several programming “paradigms,” libraries
- C++ weaknesses
  - Complexity
  - type safety
  - No acceptable textbook
A freshman programming class

• 14 or 15 weeks
  – Lectures (up to 180) + Lab sessions (with TAs; up to 25) + office hours (1-to-1)
  – 22-to-25 lectures (depending on the need for review, etc.)

• Ordered by programming technique needs
  – Not by language-manual order
    • It’s a programming course using C++ (not a C++ language course)

• “Depth first”
  – Not “bottom up”
  – Not “complete first”

• Concrete before abstract
  – Example based (Examples illustrate programming/design principles)

• Constant emphasis on the structure of code
  – Correctness, simplicity, maintainability
Philosophy - programming

• Programming involves theoretical and practical skills
  – A bit like learning to play an instrument
  – A bit like learning a foreign language
  – A bit like learning to play a sport

• Such examples takes about a decade to master at a professional level
  – Programming is a bit simpler, but think of “professional level” as completing a masters course.

• Cannot be achieved by just reading a book
  – “correspondence course in bicycling”
  – Experimentation with feedback is essential

• Cannot be learned just by rote
  – “you don’t learn a language by reading a dictionary”
  – Idioms, techniques, and concepts are important
Philosophy - students

• Don’t spend time on what students already know
  – “if the traffic light is green, cross”

• Don’t spend time on what the students don’t yet need
  – And can look up in a manual (preferably online)

• Students live up or down to your expectations
  – Challenge them

• If you have a large class, you’ll “loose a few”
  – Make that the “tail end” not the “best and brightest”
Course outline

• Part I: The Basics
  – Simple I/O, iteration, selection, types, error handling, debugging, program organization, functions, classes

• Part II: Input and Output
  – Stream I/O, Graphical output, object-oriented programming, GUI

• Part III: Data and algorithms
  – Implementing vector, pointers, generic programming, containers

• Part IV: Broadening the view
  – as needed and if time allows
  – Examples of application domains, the C language programming

• First and last (and occasionally in-between)
  – Programming in society, …, ideals and history
Computers, people, and programming

- Programming is a way to change the world
  - Hopefully for the better
Part I: The Basics

• String and numeric I/O, selection, iteration, computation, types
  – 3 chapters
• Error handling
  – Very basic exception handling, pre- and post-conditions
• Program organization
  – a complete program (a calculator); 2 chapters
• Function technicalities
  – declarations and definitions, scope, argument passing, header files …
• Class technicalities
  – members, constructors, destructors, invariants, …

• Lots and lots of examples
  – Few explanations – if any – are complete from a programming language point of view
Example from week 2

/*
read a bunch of strings into a vector of strings, sort
them into lexicographical order (alphabetical order),
and print the strings from the vector to see what we have.
*/

vector<string> words;
string s;
while (cin>>s && s!="quit") // && means AND
    words.push_back(s);

sort(words.begin(), words.end()); // sort the words we read

for (int i=0; i<words.size(); ++i)
    cout<<words[i]<< "\n";
Part II: Input and Output

• Stream I/O
  – 2 chapters

• Graphical output
  – 5 chapters, incl. an introduction to OOP/OOD

• GUI
  – just one chapter, event-driven programming
Example from week 7

Simple_window win20(pt,600,400,"16*16 color matrix");

Vector_ref<Rectangle> vr;  // use like vector
   // but imagine that it holds references to objects
for (int i = 0; i<16; ++i) {  // i is the horizontal coordinate
   for (int j = 0; j<16; ++j) {  // j is the vertical coordinate
      vr.push_back(new Rectangle(Point(i*20,j*20),20,20));
      vr[vr.size()-1].set_fill_color(i*16+j);
      win20.attach(vr[vr.size()-1]);
   }
}

// new makes an object that you can give to a Vector_ref to hold
// Vector_ref is built using std::vector, but is not in the standard library
Example from week 7

The output
Part III: Data and algorithms

• Implementing vector
  – 3 chapters, incl. use of free store, exceptions, and templates
  – Use of pointers, array, and some C style

• Generic programming and the STL
  – Algorithms (find, accumulate, …)
  – Containers (vector, list, map, …)
Example from week 10 or 11

```cpp
// print unique words from iname to oname in order
istream is(iname); // input from file iname
ostream os(ofname); // output to file oname
if (!is || !os) error("couldn't open file");

istream_iterator ii(is); // iterator for input stream from file iname
ostream_iterator eos; // end of stream
set<string> b(ii,eos); // read words from input into a set

ostream_iterator oo(os,"\n"); // use newline as output separator
copy(b.begin(),b.end(),oo); // copy words to output ordered by the set
```
Example from week 9 or 10

```c
char* cat(const char* p, char c, const char* q)
   // concatenate p and q separated by c
{
   int lp = strlen(p);
   int lq = strlen(q);
   char* r = new char[lp+lq+2];
   strcpy(r,p);
   r[lp] = c;
   strcpy(r+lp+1,q);
   r[lp+lq+1] = 0;
   return r;
}
char* p = cat("someone", '@', "somewhere");
cout << p;  // "someone@somewhere"
delete[] p;
```
Ideals and history

• Ideals matter
• History matters
• How?
Part IV: Broadening the view

(as needed and if time allows)

• Text manipulation
  – iostreams, string, and map review, regular expressions

• Numerics
  – linear algebra examples using Matrix library

• Embedded systems programming
  – What’s special about embedded systems programming
  – low-level coding, memory management
  – coding standards

• Testing
  – too theoretical

• C language programming
Projects

• “programming is learned by writing programs”
  – Brian Kernighan

• So, we need
  – Drills
    • For every chapter/lecture; done immediately, by all
  – Exercises
    • A few, too few done
  – Individual projects
    • We don’t have the time; some students do them voluntarily
  – Group projects
    • Just one; about three weeks

• But we have found it hard to get that done
  – Lack of time
  – Problems synchronizing “labs” with lectures
Surprises
((only) to me, coming from industry)

Many students

– did not know how to work hard
  • Had breezed through high school
– study for the test (only)
  • And unfortunately, our testing is pathetic
– thought they knew programming better than the professor
  • Many had known more than their high school teachers
– thought they were smarter than the professor
  • Many had been smarter than their high school teachers
– thought that they knew what they needed to learn
  • The idea of gradually assembling a portfolio of skills and knowledge was alien
– thought they would be ready to write real code immediately
  • Why on earth? (image problem)
Practical problems

• “In theory, practice is easy”
  – Trygve Reinskov (and others)

• Scale
  – We teach between 120-240 students at a time
  – 60% have programmed before; 40% has not

• Common complaints
  – “The order of topics is illogical and confusing”
  – “Principles go straight over our heads”

• Textbook, other material, (was) missing

• Attendance
  – The book plus (posted) quality slides made that problem worse

• Teaching assistants and professors
  – Old habits die hard

• Grading and testing
  – Culture and costs conspire against us!
Software problems

• Getting ready on the first day
• Installing FTLK
• Very minor problems
  – Using Microsoft, Apple, and etc. Linux and Unix distributions
    • TAs easily handle that
• Non-problems
  – “buffer overruns”
  – Pointer problems
Progress!

• For three years we ran EE (new course) and CS (old course) programs side by side
  – The new course won in the impression of professors and students
  – Now we use only the new course
• We think we are seeing progress
  – And hope for more as we are addressing known problems
• People taking our students as interns report improvement
• I have no way of proving that this is not a variant of the Hawthorne effect
• The text itself seems to be well received
  – Email, blogs, “reviews”
Questions?

• www.stroustrup.com/Programming