Journal of Engineering Education Transformations ,
Volume 29 , No.3, January 2016, ISSN 2349-2473, eISSN 2394-1707

An Approach to Introductory Programming

Abhiram G. Ranade
Department of Computer Science and Engineering, IIT Bombay
ranade@cse.iitb.ac.in

Abstract: Computer programming is a compulsory subject in most engineeringcurricula, and also in several science curricula. For most studentsit is also the first subject in theireducation in which they can actually build something. Programs can bewritten to do useful computation, and also to explore othersubjects such as science, engineering, and even art. Computerprogramming has the potential to empower students and unleash theircreative abilities.

We have developed an approach to teaching programming which emphasizesthese aspects. We use the C++ programming language, augmented with agraphics library and some linguistic devices we have developed. Wehave found that our augmentations are very useful in explaining manyprogramming concepts such as recursion, and of course enablevisualization and graphical interaction. In addition to teaching thesyntax of C++ we show how interesting programs from science,engineering, operations research can be developed with relativelylittle effort. We feel that this approach improves studentparticipation, excitement, and learning.

Abhiram G. Ranade
Department of Computer Science and Engineering, IIT Bombay
ranade@cse.iitb.ac.in

Our proposed curriculum has been described at length in the book "Anintroduction to programming through C++", recently published by McGrawHill.

Keywords: Introductory programming, C++, pedagogy, graphics.

1. Introduction

Most bachelors' degree programs in Computer Science or Information Technology begin with a course in Computer Programming. It is thefirst encounter of the students with their major, and if taught wellcan lead to "love at first sight". Computer programming is also acompulsory subject in most engineering curricula, and also in several science curricula. Given the importance of computers to thesestudents, a well-designed course in programming will greatly help them too.

Computer programming is perhaps the first subject in all of education in which a student canactually build something. You can write programs to do usefulcomputation, but also to explore other subjects such as science,engineering, and even art. Computer programming has the potential toempower students and unleash their creative abilities, irrespective oftheir degree major.

The goal of this paper is to present an approach to introductoryprogramming education. Of course, many approaches have been proposedin the literature for this. These range from the classicimperative-first

to functions-first or objects-first or models first (Bennedsen, 08),or top-down or bottom-up. There are approaches that recommendstaying away from computers and learning to first reason aboutprograms (Dijkstra 88). There also are approaches that appear to teach programmingas a byproduct while apparently teaching robotics (Lawhead 02), or graphics (Cooper et al 00, Resnick et al 09)orgeographical information systems (Meyer 03). There are also aspects such as thechoice of language, availability of appropriate tools and IDEs.

A cursory glance at the different approaches indicates that eachapproach really has its own definition of "programming". For example,the objects first approach seems to be oriented more towards businessdata processing, or GUI programming. The imperative-firstapproach seems to be more oriented towards systems programming. Thefunctional programming approach seems to emphasize the elegance ofclean mathematical notation and semantics in writing programs.

In what follows, we first present what we think is the "core" ofprogramming, what we believe the first course must focus on. We thendiscuss the various issues that go into deciding how to teachprogramming. We finally present our approach, and also our experiencewith it.

2. The "core" of computer programming

For an introductory course, we believe that it is appropriate todefine "programming" as the "act of expressing in a programminglanguage the computations that you know how to perform manually".Thus we do not include in programming the task of designing cleveralgorithms. But we do include issues such as understanding thespecification and reasoning about them. The attributes of a programmost relevant for an introductory course are as follows.

- Generality: The same program should be able to solve instances of different sizes. A key part of learning to program is how to think about and describe the computation in general, e.g. "as many iterations are needed as the rows of a matrix".

- Matching the structure of computations to the structure of the program: A program may perform millions of arithmetic operations, however its textual size will typically be much smaller. The key to making programs compact is to understand the patterns in the computation, and match those patterns by the constructs (iteration, recursion) in the programming language.

- Naturalness/readability: A program is not only to be executed on the computer, but also to be read by other programmers. Thus it needs to be easy to understand.

- Extensibility: Once we write some code, it is often desirable to build up on it, or modify it to suit other purposes. Our expression and our programming language should facilitate such reuse.

We do not include speed in the above list. Surely, our program mustrun fast. Speed is more an issue for an algorithm design course.

We believe that there will be general agreement on the points madeabove. The key pedagogical question then is:

- How do we get to the heart of a subject quickly and keep the focus on it rather than get lost in the details?"

Using languages such as C++ to teach programming seems to require usto first talk about a lot of technical details before we can doanything interesting. Consider the standard introductory program:

```
#include<iostream>

using namespace std;

int main(){

cout<< "Hello world!" <<endl;

}
```

To a beginner almost all the words in this except "Hello" and "world" must appear like some mysterious mumbo jumbo --potentially leading to intimidation or to boredom. And on top of it,the program accomplishes precious little. Programming in a language such as C++ seems to require us to master a lot of conceptually trivial information. Indeed, a well-known textbook of introductory programming, very commonly used in India,devotes several initial chapters on such information and includesa whole chapter on printing statements before

anything ofconsequence or excitement is attempted.

Contrast this with approaches in which programming is taught in a language such as Scheme (Abelsonand Sussman 96). In the very first lecture, a studentcan get to program interesting computations, e.g. finding squareroots, greatest common divisors.

3. Instructional vs. professional languages

Over the years many languages have been invented to ease teaching ofprogramming. Some of them such as Logo (diSessa and Abelson81,Papert 80) contain graphics primitivesand were actually designed to teach children to program. They arecharacterized by simple syntax and small number of primitives. Schemeappears similar, but is actually a full-fledged language that has beenused for developing complex projects. However, even Scheme hasn'tfound universal favour with educators who often consider suchlanguages to be "not real".

At the other end of the spectrum we have approaches that teach objectoriented programming -- possibly because professional programmerssupposedly only use object oriented programming. But this is notconsidered easy, even by the proponents of the approach. The reasons are several. For example, organizing simple introductory programs into classes is often very artificial and verbose. Expectinga student to actually develop classes very early requiresunderstanding function abstraction (for developing memberfunctions/methods) even before control structures are understood.This can appear unmotivated and overwhelming.

4. Programming in the context of an application

We never write programs in a vacuum -- programs are always written tosolve problems from some domain. The domain could bebasic mathematics, or day-to-day life. If you wish to write moreinteresting programs, you need to deal with more interesting domains.

To help in this, one popular idea is to teach programming inconjunctionwith some application, e.g. robotics (Lawhead et al 02), or graphics. TheScratch system (Resnick et al, 09) uses 2 dimensional graphics, while the Alice system (Cooper et al, 00)uses 3 dimensional graphics from which to draw programming examples. A geographical information system has also been used as a domain. (Meyer03).

These approaches are attractive also because they force the student towork alongside an existing system. This is useful because in themodern workplace programmers hardly develop programs for scratch, butrather work to enhance or modify existing programs.

A drawback of the approach is that the domain chosen may not appeal toevery student, or learning the domain (e.g. robotics principles, 3dgraphics principles) may place additional learning burden on thestudent.

5. Motivational aspects

The first point is that real life examples are very important in learning. We could even say that if we cannot give goodreal life motivating examples, there is no point in teaching anythingincluding language constructs. The term "real life" in this contextshould not be taken to mean "commercially important", but should be understood more as "what can be drawn from the experience of the student,based on what she has learned so far, say till junior college, and is learning right now".

Second, it is important to keep the fun in learning. Students shouldlike what they learn. This will happen, for example, if they can workwith pictures or develop games. Rather than fight the fun lovingnature of students, we think it is better to channel that energyconstructively.

6. Our approach

Our approach is based on the following ideas: (a) Stress the usage of programming language constructs in real lifeproblems rather than the syntax and abstract semantics, (b) Usegraphics (as well as high school math including geometry) as a domainfor illustrating programming concepts as well as a source ofinteresting programming exercises, (c) Keep the focus on theinteresting ideas, from the first day of the course.

We use the C++ language, augmented with a library we developed, simplecpp. The biggest component of simplecpp is a two-dimensional graphicslibrary. Two kinds of graphics are supported: so-calledturtle graphics and more standard coordinate

based graphics. Turtle graphics is adapted from Logo (diSessa and Abelson, 81,Papert 80). The basic idea in turtle graphics is:students get to program the on-screen movement of a symbolic animal,the turtle. The turtle has a pen that draws on the screen as theturtle moves, so that the purpose of turtle graphics is to move around the turtle so that interesting pictures get drawn. In addition, we have moreconventional graphics which allows students to create and manipulateon-screen shapes. It is possible to create reasonably excitingdrawings and animations, e.g. Hilbert space filling curves, the snake game, bouncing balls, planets rotating around the sun and so on.

Another important component of simplecpp is a repeat statement. The repeat statement has the form:

```
repeat(count)
  {statements to be repeated}
```

As you might guess the statements to be repeated are repeated asmany times as the value of count. A repeat statement is translatedinto a for loop using C++ preprocessor macros which getloaded automatically. The main reason for defining this statement isthat it can be introduced in the very first lecture! The standardlooping statements in C++ are far too complex and need substantialpreparation before they can be introduced. Therepeat is easily understood, and using it students can start writing interesting programs from day 1.

A. The first lecture in the course

The first lecture in any course is extremely important: it is important to introduce the core ideas rather than bore students with unnecessary detail. Thefirst lecture sets the tone for the course: to a student it signals whether the course will have interesting ideas, orwhether it will just be lot of boring information.

Here is the first program that we show to students in the firstlecture.

```
#include<simplecpp>

main_program{

turtleSim();

forward(100); left(90);

forward(100); left(90);

forward(100); left(90);

forward(100);

wait(5);

closeTurtleSim();

}
```

Several things are to be noted. First we only include the simplecpp library, which in turn includes iostream and issues using commands. Thus we only need to explain to the students why we need to include simplecpp, other explanations can comelater in the course. Next, we have a macro main_program whichexpands to intmain(), so we don't need to explain what int and main mean and why main has parentheses () following it. Thiswill get explained later -- after we discuss functions, when the students canunderstand everything.

The body of the program is perhaps most interesting. It opens theturtle simulator window, which already has a turtle at the center ofthe screen (a red triangle, as is customary). The command forward(100) causes the turtle to move forward 100 pixels. Thecommand left(90)causes the turtle to turn left by 90 degrees.Thus the above code causes the turtle to draw a square (because of itspen). After that the program waits for 5 seconds, and thenterminates.

Note that the first program creates expectations in minds of students, e.g. "Can I draw other kinds of polygons?" Some students might alsoask if they need to write 50 forward statements if they wish todraw a 50 sided polygon. The repeat statement can then be introduced. Indeed the second program of the first lecture could be

```
main_program{

turtleSim();

repeat(10){

forward(100); right(36);

// will draw a decagon.
```

```
 }

}
```

The turning angle, 36 degrees, is easily calculated from the high school geometry theorem "The exterior angles of a polygon add up to 360 degrees."

We have found that the students spontaneously understand nested repeatstatements, e.g. as in the program below.

```
main_program{

turtleSim();

repeat(4){

repeat(10){forward(5); penUp();

forward(5); penDown();}

left(90);

 }

}
```

Many students correctly guess that this will cause a square to bedrawn, using dashed lines.

Notice that on the very first day we can accomplish many things using the above ideas. Wecan create a great amount of excitement. We can force students tothink algorithmically: they need to figure out the turning angles.They also need to use repeatstatements properly to draw complex figures. This requires matching the pattern in the drawingwith the pattern of repeat statements in the program. This isof course a very fundamental programming activity! And we have got toin on day 1.

Most students will have already heard that a circle is a limiting case of an n sided polygon, as n becomes large. On a 1000 by 1000 pixel screen, choosing n=100 is enough to draw reasonably nicelooking circles. Thus we can ask students to draw patterns involving circles too.

B. Utility of repeat and graphics

The repeat statement and graphics are useful for providinginteresting exercises for several initial weeks. For example, after discussingdata types and assignment statements, we can write a programcontaining code such asthe following

```
inti=1;

repeat(40){

forward(i*10);

left(90);

i = i + 1;

}
```

As you might guess, this draws a spiral. Note that without therepeat, this code would have to wait for the looping constructs to betaught.

Graphics is useful in explaining difficult concepts such asrecursion. That is because many pictures have a recursive structure. A simple example is a tree -- it consists of smallertrees on top of a trunk. It can be easily drawn using a recursivefunction. Here is perhaps the simplest possible recursive function for drawing trees.

```
void tree(int levels){

if(levels > 0){

forward(levels*10);

left(15);

tree(levels-1);

right(30);

tree(levels-1);

left(15);

forward(-levels*10);

 }

}
```

This would have to be called as, say,tree(5).

Touse graphical objects in the coordinate graphics

system, it isnecessary to use constructors and member functions. For example, hereis the code for creating a rectangle and moving it.

```
Rectangle r(xc,yc,L,H);

// center coordinates, Length, Height

r.move(deltax, deltay);
```

This can be explained to students even without explaining objects:"the first statement creates a rectangle namedr, the secondstatement moves it." Thus the students get introduced to constructorsand the dot notation well before object-oriented programming isintroduced.

The graphics functionality is implemented using a class hierarchy. Thus the graphics library itself can serve as an example when discussing object oriented programming.

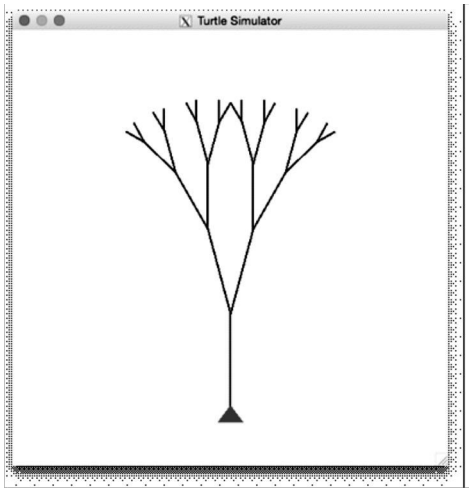

Fig. 1Result of calling tree(5)

C. Overall pedagogical approach

We use a simple principle in introducing new ideas: first presenta real life motivating example in which the new idea is needed. Weuse this in introducing looping statements, functions, object orientedprogramming, practically everything. Once the students understand whysomething is needed, we feel they have less difficulty in remembering, understanding and applying it. We also try to give graphical uses - pictures oftenhelp remembering more than text.

We develop substantial programs for applications drawn from math,science, engineering, operations research, and even topics that are more like Art.We thus believe that our treatment better integrates programmingwith the math and science skills (not to mention general worldlyskills!) that the students already have. We feel this synergisticallybenefits the learning of computer programming and the other sciences.We also find that this appeals to our audience which includes nonCS/IT majors.

Finally, we do not teach as per any ideology (e.g. objects first), butgo by increasing order of complexity of constructs. Thus we beginwith the imperative subset of C++, then talk about functions anddiscuss some of the functional programming related ideas. Classdesign comes last, though we do introduce graphics classes for useearly on.

7. Experience

Our approach was developed while the author was teaching theintroductory programming course at IIT Bombay. The second offering ofthe course was taught with a draft of a book written as per the ideasdiscussed here. About an year ago the book was published by McGraw Hill Education (Ranade 2014).

The book has been used in the introductory programmingcourse in two offerings last year. It is currently being used thethird offering at IIT Bombay, and is also being used in VishwakarmaInstitute of Technology, Pune. The book was also used in variousofferings of a Massively Open Online Course (MOOC) byProf. D. B. Phatak and Prof. SupratikChakraborty of IIT Bombay.

The author is happy to report that the feedback has been very positive.

The simplecpp library isavailable from the author's webpage and the publisher's webpage, for Unix, Mac OS, and Windows operating systems.

8. Concluding remarks

We believe that programming is a unique subject which must be taughtwith excitement. We believe this can be done without sacrificingrigour, without inventing "teaching languages". We feel that C++ is a good language for teaching. Its complexity need not overwhelm novices if we provide some syntactic sugar(e.g. the repeat statement) and deemphasize obscure features. Besides being a commercially/industrially popular language, it has evolved over time to include the best features from other languages (e.g. lambda expressions from Scheme/Lisp).

Finally, we feel that in this age of touch screens, graphical inputoutput is invaluable to get the learner's attention, and is also agreat learning aid.

References

Harold Abelson and Gerald J. Sussman. (1996) Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition.

J. Bennedsen. (2008) Teaching and learning introductory programming - a model based approach. PhD Thesis.

Stephen Cooper, Wanda Dann, and Randy Pausch.(2000) Alice: A 3-d tool for introductory programming concepts. J. Comput. Sci. Coll., 15(5):107116.

E.W. Dijkstra. (1988) On the cruelty of really teaching computing science. EWD 1036.

Andrea diSessa and Harold Abelson.(1981) Turtle Geometry: the computer as a mediumfor exploring mathematics. MIT Press, Cambridge, MA, USA.

R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, andM. Felleisen. (2002) Drscheme: a programming environment for scheme. J. Functional Programming, 12(2):159182.

P. Lawhead, M. Duncan, C. Bland, M. Goldweber, M. Schep, D. Barnes, andR. Hollingsworth. (2002) A road map for teaching introductory programming using LEGO. In ITiCSE-WGR '02 Working group reports from ITiCSE on Innovationand technology in computer science education, pages 191201. ACM.

B. Meyer. (2003) The Outside-In Method of Teaching Introductory Programming. In Ershov Memorial Conference, volume 2890 of Lecture Notes in Computer Science,pages 6678.

Seymour Papert. (1980) Mindstorms: Children, Computers, and Powerful Ideas. BasicBooks, Inc., New York, NY, USA.

Abhiram Ranade.(2014) An Introduction to Computer Programming through C++.McGraw Hill Education.

Mitchel Resnick, John Maloney, Andr´esMonroy-Hern´andez, Natalie Rusk, Evelyn Eastmond, Karen Brennan, AmonMillner, Eric Rosenbaum, Jay Silver, BrianSilverman, and YasminKafai. (2009) Scratch: Programming for all. Communications of the ACM, 52(11):6067. time to include the best features from other languages (e.g. lambda expressions from Scheme/Lisp).