Teaching the Programmer's "Bag of Tricks"

Brian J. Resnick, P.E.
University of Cincinnati

Abstract

Prior to entering academia, the author provided supplemental programming education to the new hires for a manufacturer of an embedded system application. Over a twenty year period, he observed the skill set of graduates from a variety of educational institutions, and discovered that they understood the syntax but were unable to conceive or express a solution to many of the problems at hand. They had limited exposure to the problem-solving techniques of the programmer.

This paper presents an approach for teaching programming that evolved from the experiences of enabling those new hires to perform their job. It describes how to prepare to learn a language, presents visualization techniques and problem assignments that take a student from zero to full-speed, and concludes with a series of programming axioms and background information that is essential to every programmer. The author applied this approach to one course of Data Structures in C++ for third-year students who had previously received one or two courses in other programming languages. Selected comments from a survey of the students are included.

Introduction

Computing Curricula 2001 [1], a joint task force report from the Computer Society of the Institute for Electrical and Electronic Engineers and the Association for Computing Machinery, describes six different strategies for teaching introductory computer science. The two most widely used are the traditional Imperative-first paradigm and Objects-first, the rapidly growing approach that emphasizes objects and object-oriented design. Both place programming first and start the student with a mainstream general-purpose language.

For each strategy, Curricula 2001 provides a guideline for a two-course sequence, but strongly suggests their three-course curriculum because "the two-course sequence is no longer sufficient to cover the fundamental concepts". Two or three courses may be the recommendation, but the reality is, as Hankley [2] notes, that most engineers receive just one service course wherein they learn the syntax of a language.

This limited and language-centric education results in today's engineer, someone who can cause a program to print "Hello World", count to ten in a loop, and sum the elements of an array, but who does not know when to use an array or what to put into the array to solve a problem.

There are two problems: one is the single course limitation, and the second is the content. Piper, Castle, Fuller and Awyzio [3] address the second problem by proposing a hybrid approach to the core curriculum for the University of Wollongong, which combines the Imperative-first and

Algorithms-first approaches from Curricula 2001 to achieve "a proper balance" between the problem solving skills and the mechanics of the language.

Is there a way to apply this hybrid concept in a single course? As a practicing engineer in industry, the author has had the opportunity to provide supplementary programming education to many engineering new hires. That twenty-year experience has exposed the missing elements, the missing knowledge base of the new hires, and has evolved into a "crash course" on programming that focuses on solution techniques and "how things work" – the programmer's bag of tricks.

This paper is a condensed version of that course. It begins by describing how to prepare to learn a language. Following the preparation is a technique for visualizing variables and memory that takes programming out of the realm of being invisible and untouchable. The very first program has the student doing something useful; solving an algebraic word problem. Continuing with the same word problem, the student implements functions and visualizes how they work. Next, is a confrontation with the harsh realities of computing, along with the introduction of various data types. Combining those data types into objects and visualizing those objects proves helpful as the student solves that same algebra problem using objects. Useful applications work with lists of objects, and once again, visualization drives home the concept. A menu system problem culminates the discourse on lists. A traffic light controller problem illustrates the state machine problem-solving technique. The course concludes with a series of series of programming axioms and background information that is essential to every programmer.

The author, a new tenure-track faculty member, applied this approach to a recent lecture course on Data Structures in C++ for third-year Electrical and Computer Engineering Technology students. The paper concludes with selected comments from students who replied to an optional survey, and a summary of the teaching experience.

The next section begins the crash course. The lesson of the notebook is crucial to the success of the new graduate. The organized new hire, the one who records the details and does not ask the same question twice, will go far.

Preparation for Learning a Language

Help the student become an organized and disciplined learner. Have them acquire a composition book and begin a programmer's notebook. Follow established guidelines for laboratory notebooks, such as those provided by Rice University [4]. Number the pages, skip a few pages in the front for a table of contents, and divide the rest of the book into sections: computer hardware, development environments, languages, programming concepts, and miscellaneous. Record everything important about programming and computing, and be sure to include the trivial stuff such as how to turn on the computer, the directory system of the project, and passwords.

Research the reason for the creation of the language, determine the perceived advantages and disadvantages, and record this information in the language section of the notebook. Later, after applying the language, update the advantages and disadvantages based on personal experience. This information will help with language selection activities of the future.

Record the unique terminology of the language, and relate it to equivalent concepts from other languages. Doing so will help develop the ability to translate between languages.

At the introduction of each new tool of the language, record the basic how-to information in the development environments section of the notebook:

> For *editors*, record how to start, open a file, create a new file, insert, overwrite, delete, copy, paste, save, exit, and exit without saving.
>
> For *compilers*, include how to start, compile one file, compile many files, and necessary compiler options.
>
> For *linkers*, record how to start, list which files to link, select libraries, and specify the output and options.
>
> For the *executable*, include how to launch and supply any runtime requirements.

With notebook in hand, the student is ready to learn programming. The presumption of this paper is that the student is new to programming, and the following discourse is elementary by design. However, the author has found that even experienced programmers have "ah-ha" moments with this material.

Visualize Memory

A concept often neglected in teaching a language is that variables are physical entities, they take up space, and they reside in a physical location. Use visualization techniques to eliminate the thought that programs are invisible and cannot be touched. One technique that works well is to use a cardboard box to represent the memory space of a variable.

A variable is a box with a slip of paper in it. The box has a name and an indication of the type of data it can hold. It can hold only one value. The slip of paper has a numeric value written on it.

Begin with floating-point [5] numbers, known as rational numbers in algebra, because engineering students are comfortable with them. An integer, the data type traditionally taught first, has unintuitive limitations with division and maximum/minimum values. Introduce that complexity after the student is capable of doing something useful with the language. Strings, integers, characters, etc. are advanced data types.

Introduce the syntax of the language that instantiates a floating-point variable. Relate the variable's name and data type specification to those on the box. Relate the action of the assignment operator to that of inserting a slip of paper with a value on it. Remember that the box can only hold one slip of paper. The action of inserting a new slip destroys the existing slip.

Algebraic Word Problems

The very first program should be useful and should solve a well-known problem. For instance:

> Convert the temperatures of 32 and 212 degrees Fahrenheit to degrees Celsius.

The well-known formula, where "c" is degrees Celsius and "f" is degrees Fahrenheit, is:

$$c = \frac{f - 32}{1.8}$$

Rewrite the formula to match the syntax of the language, and solve for both answers:

f = 32

c = (f – 32) / 1.8 = 0 ← by substituting for "f" and simplifying

f = 212

c = (f – 32) / 1.8 = 100 ← by substituting for "f" and simplifying

Present the syntax of an actual complete program that implements these steps. Walk through the program using boxes and slips of paper. Make sure the names on the boxes are the same as the variables in the program. When demonstrating the part of the equation to the right of the assignment operator, just read the value from the slip of paper and use it (but do not remove the paper from the box). When demonstrating a variable to the left of the assignment operator, replace the slip of paper with a new piece.

Compile and run the program. Add appropriate output statements to display the results. The student is now capable of writing a program that solves a real problem. Prepare the student for functions by noting the inflexibility of the program and the duplication within the program.

Visualize Functions

The engineering student has a history with functions that extends back to their first algebra class when "y = x + 2" became "y = f(x)" and "f(x) = x + 2". Make the connection. Draw the parallel between algebra functions and programming functions.

Rewrite the first algebra formula as a function:

$$c = DegreesFtoC(f) = \frac{f - 32}{1.8}$$

Introduce the syntax for an equivalent function call in the program. Be sure to make the variable names within the function different from those outside the function, because using the same name leads to the belief that they must be the same. For instance, use *f* and *c* in the main body, and *fahr* and *celsius* in the function as in this example written in C:

```
double DegreesFtoC (double fahr)
{
        double celsius;
        celsius = (f – 32.) / 1.8;
        return celsius;
}
```

```
main ()
{
        double c;
        double f;
        f = 32.;
        c = DegreesFtoC (f);
        f = 212.;
        c = DegreesFtoC (f);
}
```

Visually, a function is like a worker at a desk with an In-box and an Out-box; the boss puts work into the In-box, the worker performs the job, and then puts the results in the Out-box.

The "DegreesFtoC" worker has one In-box named *fahr* and an Out-box named *celsius*. To illustrate the function call, copy the value from box *f* onto a slip of paper and place it in box *fahr*. Note the word **copy**. This is a pass-by-value example; the worker has no access to *f*. The function can alter *fahr*, but *f* does not change.

The worker at the function desk reads the value from box *fahr*, performs the mathematics, and writes the result onto a slip of paper for insertion into the *celsius* box. To complete the function call, copy the value from the *celcius* box into box *c*.

Stress the point that the function worker is very, if not completely, isolated; that the worker's access to information outside the desk is only via the In-boxes. Violating this rule creates bad programming habits, and defeats the purpose of encapsulation. Later, once the programmer has formed good habits, information access may expand to the rare instance of a global variable.

Now, that the student is able to program solutions to algebra problems, it is time to introduce other data types and the hard realities – the limiting factors – of computing.

Numbers Have Limits and Other Realities

Unlike the variables used in algebra, values in computer memory have limits. Numbers can be too big or too small to fit into the box. Variables are just a collection of ones and zeros, bits and bytes that have a limited range and precision.

Hand-held calculators transparently switch between integers and rational numbers. As a result, there is no intuitive understanding of the difference between these data types, or that there is a preferred data type. Integers are for counting, indexing, and selecting. Floating-point numbers are for math. Switching between data types can also occur in programming languages, but a good programmer limits such activities by selecting the appropriate data type for each variable.

Integers and floating-point data types come in multiple sizes. When memory size is not a limiting factor, select the data type size that provides the fastest operation. For integers, this means matching the bit size of the compiler, that is when using a 32-bit compiler, use 32-bit integers. For floating-point data types, match the native data type of the math library. In Java for instance, this means selecting double instead of float. Selecting the wrong data type can

significantly slow down the execution of the program because the compiler will insert extra "hidden" data type conversions.

Character data is an anomaly because it is not a basic data type of the processor. The handling of text, strings of characters, is an invention of the language. Capabilities vary from language to language. Programmers use character data mostly for output. Except for name and address kinds of fields, user input of text is rare, because interpreting text is not easy. It is far easier for the programmer to provide a menu of selections than it is to interpret natural language input.

Individual variables are interesting and necessary, but the real power of expression comes when the programmer combines variables into a new data type perfectly suited for the problem.

Visualize Objects

Every thing – every noun – is an object. Every thing has data. For maintainability and reusability reasons, the data must be indirectly accessible. Encapsulation is an organizational technique that ties together the various pieces of data and the functions for setting, getting, and processing the data. In programming terms, a class encapsulates the object's data and the functions that service that data.

Visually, a complex object is a bunch of variable boxes glued together side-by-side, and a class is a set of function workers at their desks. The assembly of boxes represents one instance of the class. Each instance has a unique name. Each instance has the same assembly of boxes as any other instance of that class, but the values in the boxes may be different.

The function workers are very finicky. In addition to their in- and out-boxes, they need to know the name of the object being worked on. They refuse to work on objects instantiated by a different class because they do not understand their assembly of boxes.

To make the transition to objects, take a familiar assignment and convert it into an object problem. For instance, have the student create a temperature object with four capabilities: getting and setting the temperature value as degrees Celsius, and doing the same in degrees Fahrenheit.

Note the power of encapsulation: as long as the access capabilities convert between the desired temperature unit and the object's internal unit for temperature, the user of the class does not know or care what the internal unit is. Have the student change the internal units from Fahrenheit to Celsius to Kelvin, and realize that the application program does not change.

Even when the language does not support classes, a good programmer will use the concepts of encapsulation by grouping related variables and functions together with a common naming scheme, and when possible pass the object as a whole to its functions.

Visualize Lists

By design, class functions deal with data from a single instance of the class. Application programs, on the other hand, are users of classes, and typically deal with many instances of the class at the same time. A list is a convenient way to organize the collections of objects.

Visually, a list of variables is a stack of boxes, and a list of objects is a stack of box assemblies. The whole stack has one name, and each item in the stack is individually accessible with some combination of the stack name and an index into the stack.

An alternative visualization is a spreadsheet or data table; the columns are member variables, and the rows are the instances. When a programmer sees a table of data in a word problem, the first thought should be a "list of objects".

A programmer's education of lists begins with arrays, and expands to linked lists and/or helper classes that implement lists. It is important that the programmer separate the concept of a list from the particular method of implementation.

Processing Lists

Application programs typically provide the same processing to every item in the list. The thought process is "for each item in the list do such and such". The programmer translates that thought into a set of instructions that operate on one item in the list, and then loops back and does the next item; loops process lists. Introduce the looping capabilities of the language.

Sometimes not every item in the list needs to be processed, show the student how to skip to the next item. Early termination of the looping is useful when searching for a particular item; show the student how to break out of a loop.

Some languages, such as C#, Java 1.5, and Visual Basic, offer a "for each item in the list" capability. These powerful capabilities hide the underlying issues regarding the servicing of lists:

Lists Cannot Be Endless

The process that services the list needs to know how many items are in the list. There are two basic implementation methods: 1) pass the value that is the quantity of items to the process, and 2) the process knows how to recognize the end of the list.

Method #1 is typically used when the type of data being processed is unknown – such as a message buffer for a transmit routine. The sender has data in a known structure, but the generic transmit routine has no knowledge of that structure, and thus, no way to know the size, and no means to detect the end of the message. The sender must pass the quantity of bytes to send to the transmit routine.

Use method #2 when arrays of known objects are involved. Mark the end of the array with a special value in the last entry. Null terminated strings in C/C++ are a good example of this method.

Lists May Need To Grow

The quantity of objects that an application must process is typically an unknown. The program must adjust to the amount of data, acquiring memory space from the operating system as needed and releasing it when done. Programmers experience this concept by

implementing the nuts and bolt of a linked list project. An address book program is a good exercise, especially when connected with file I/O for saving and reloading the data.

The discussion on lists takes many hours. The author uses the following assignment to tie all of the pieces together, and give the student an opportunity to apply list processing.

What to Do With Lists

Viewing a list of objects as a *program* and each object in the list as an *instruction* is a powerful concept that lets the programmer create a new *language* for expressing the solution to a problem. Taking this view is a major step forward in learning how to apply lists.

A menu system for a simple user interface is a good example. Each instruction includes a prompt for the selection and a function to call if the user makes that selection. The menu system receives the list of instructions, displays the prompts, accepts and validates user entry, and then calls the appropriate function.

This problem is deceptively simple, extremely useful, and it exercises all aspects of list processing: objects, lists of objects, unknown length lists, and loops.

Another important solution technique that the students need to learn is the application of the state machine.

State Machines

Sometimes an object represents a complex situation with multiple states and well-defined transitions from state to state. A traffic light controller is good example; the light remains green on the primary road until sensing a car on the secondary road, which triggers the orderly sequence of green to yellow to red for the primary road, followed by a short green, yellow, red sequence for the secondary road, and finally back to green for the primary.

A programmer solves this type of problem with a state machine, and implements the state machine with enumerations and switch statements, or with inheritance.

The author uses the traffic light controller problem in the classroom. The student creates a simulation of a traffic light controller that displays the condition of the signal lights on both the primary and secondary roads for each second of the simulation. At predefined times during the simulation a car "appears" on the secondary road triggering a change in state. Require a minimum "ON time" for the primary road's green light, and have the appearances of the secondary cars timed to exercise that minimum.

Thus ends this mini crash course in programming. The next series of sections are fine points of programming, axioms, and other information that all programmers should know. Distribute the content throughout the course at the appropriate "point of learning".

Essential Background

Knowing how a computer works is the foundation for debugging difficult problems. What may seem like unimportant trivia is actually very valuable. For example, calling a function places the

parameters for the call, the return address, and the automatic variables of that function onto the stack. With this knowledge, it is easy to see how exceeding the bounds of an array on the stack could clobber the return address, and then cause the program to crash.

Knowing how the tools work together helps in the design of efficient programs. For instance, a compiler converts a program source file into an object code file, which for a language such as C/C++ contains machine code and memory requirements. A library is a collection of compiled functions that a program may use. A linker ties together various object files and libraries to produce an executable. The operating system, as it loads the executable into a particular location in memory, relocates the relative addressing within the executable to match that physical location. The executable not only contains code, but it also reserves memory space for persistent variables and fills that space with their initial values. For un-initialized non-automatic variables in C/C++, bytes of zero fill the space, thus "initializing" those variables to zero.

Familiarity with bits and bytes, Boolean logic, discrete mathematics, hexadecimal numbers, and ASCII codes rounds out the minimum background requirements for an effective programmer. Again, teach these throughout the course, but do give one warning upfront:

> The rate of change in the field of programming has kept up with Moore's law [6]. As the power doubles, so does the "behinds the scenes" activities of the human interfaces and computer interfaces; from character graphics to pixel graphics to animations, from modems to Ethernet to wireless, from static libraries to dynamic libraries to remoting. Keeping up with the advances and expanding the background information is an ongoing activity for the programmer.

Testing

Programming is causing the computer to do the right thing. More importantly, programming is preventing the computer from doing the wrong thing. Test the boundary conditions and make sure the program knows right from wrong.

Name That Number

Using the same numeric value in multiple places within a program is a maintenance nightmare. Replace the number with a name, and use that name throughout the program. Doing so improves clarity, readability, and reduces errors when the value needs to be changed.

The best way to represent a group of terms, such as the days of the week or the points of a compass, is to use enumeration. Not only does each term receive a name, the collection itself has a name, and variables of this new data type can be instantiated, which remarkably improves readability and maintainability.

Do Not Type It Twice

Whenever a programmer begins to type the same line or collections of lines for a second time, there is an instinctive reaction to place those lines into a subroutine, thus improving maintainability by reducing future edits to one location. The programmer also senses when two

or more lines are almost the same, and then determines how to place those lines into a subroutine with parameters.

Clarity of Expression Is Of Utmost Importance

Accurate, informative comments, meaningful variable names, and a consistent easy-to-read format are critical to the maintainability of a program and the hallmark of a great programmer. Poor programmers blame tight schedules and budgets for failing in these endeavors; devious programmers believe this kind of information hiding is a job security technique.

Set precedence and make clarity a major component of the grade. Checking for clarity is time consuming, but it provides a great insight into the comprehension level of the student.

Every Protocol Needs Version Control

A protocol is the format used to transfer data between two entities. Common examples are file formats and message formats. Invariably, an application that creates data files will catch on. As the application grows, new versions of the file format will become necessary. "Old" versions of the program will hang around forever, and need to gracefully ignore or reject newer data formats. "New" versions of the program need to process all formats; new and old.

A good programmer designs each format to have some form of identification and a revision number because this permits the receiver to recognize the format and adjust accordingly. For instance, the first data written to a file should include file type identification and a revision number. The receiver of this file can check this information and reject the file as a whole, or process it in accordance with the rules the revision.

Rewrite Three Times

Incredibly detailed functional specifications, top-down design, bottom-up implementation, and fast prototyping are just some of the methods instituted by software management gurus to improve the productivity of a programmer. There are just too many details, too many options, to get it right the first time. The latest management craze, Extreme Programming [7] accepts this notion and even describes itself as the process of experimentation and improvement.

The great American painter, Andrew Wythe, makes scores of sketches and studies in preparation to painting a finished work. The same process is true for programming. The programmer needs to experiment, to step back and look at the program, and then make improvements. It takes a minimum of three rewrites to make a program near perfect. Plan schedules accordingly.

Shortcuts

Do not worry about language shortcuts. For experienced programmers, the shortcuts are a convenience and may provide for truly elegant expression. Newbie's should record the shortcut in their notebook, and proceed with a style that provides the most comprehension.

Student Comments

The following comments are from third and fourth year Electrical Engineering Technology (EET) students and Computer Engineering Technology (CET) students at the University of Cincinnati who took a one-quarter course in Data Structures using C++ taught in the style of this paper:

- "Learning problem solving techniques is a much more valuable experience than learning a language syntax. Learning the syntax of a programming language will not make you a programmer. It is obvious that one must learn the syntax and nuances of a language to make a program work, but without the underlying understanding of how to attack and solve a problem any language is useless."

- "It is much easier to learn a programming language if it is treated like a tool and not as the actual solution."

- "Learning a programming concept first may make it easier to teach/learn the implementation. However, programming isn't an easy concept to teach. Many people who do not understand the implementation may be completely confused if a concept about programming is presented and they have no background of how to use it."

- "I personally enjoy the low level details because I like to know how things work. There is a much broader understanding that EET's especially should have."

- "Knowing what the processor is actually doing is important information to learn not only for programmers but any IT professional. It's a good base for teaching programming because learning how a processor works and what methods require less memory or less CPU time will benefit a person who writes long programs on the job some day. Also, when troubleshooting a computer with multiple programs running, knowing the concepts of the processor may help that individual see which processes are causing problems."

Conclusion

The ultimate goal of the programming education is for the student to view the computer as a tool that they know how to apply. How many courses it takes to achieve this level of understanding and the content of those courses is an endless debate in the educational community.

My approach to teaching programming grew from years of experience bringing new hires "up to speed" as quickly as possible. It focuses on the application of problem solving techniques and understanding how things work. The syntax of the language is a secondary issue. I created all of the assignments before mainstream languages had classes, and it is important to note that the approach works well with both object-oriented and non-object-oriented languages.

My university experience with this approach is limited to a single one-quarter course in Data Structures using C++ for third-year students. The following are my observations:

- The problems and exercises were appropriate and resonated with the students.

- The course involved three one-hour lectures and one three-hour lab per week. The next preparation will integrate the lecture and lab into two three-hour sessions per week that hopefully will allow for a more immediate application of the lecture material.

- The course received mixed reaction from the students. Those truly interested in programming saw their technique improve dramatically. Those students who did not like to solve problems found the course to be too difficult. Those in the middle said the course helped with their understanding in other Computer Engineering Technology courses, which in and of itself is an important success.

Bibliography

[1] Computer Society of the Instituite for Electrical and Electronic Engineers and the Association for Computing Machinery. *Computing Curricula.* 14 Dec. 2001. 17 Nov. 2004 <http://www.computer.org/education/cc2001/final/index.htm>

[2] Hankley, William. "Software Engineering Emphasis for Engineering Computing Courses: An Open Letter to Engineering Educators". *2004 ASEE Conference.* Paper 2305 on Conference CD.

[3] Piper, Ian et al. "A Hybrid Approach to the Core Curriculum". *32nd ASEE/IEEE Frontiers in Education Conference.* Session F3G-13.

[4] Caprette, David. *Guidelines for Keeping a Laboratory Record.* 25 April 2003. Rice University. 27 Dec. 2004 <http://www.ruf.rice.edu/~bioslabs/tools/notebook/notebook.html>

[5] Hollash, Steve. *IEEE Standard 754 Floating Point Numbers.* 04 Nov. 2004. 25 Dec. 2004 <http://stevehollasch.com/cgindex/coding/ieeefloat.html>

[6] *Moore's Law.* Intel. 15 Oct. 2004 <http://www.intel.com/research/silicon/mooreslaw.htm>

[7] Wells, Don. *Extreme Programming.* 28 February 2004. 20 Dec. 2004 <http://www.extremeprogramming.org/>

Biography

BRIAN J RESNICK – Is an Assistant Professor of Electrical and Computer Engineering Technology at of the University of Cincinnati. Software engineering, embedded systems, automation, motion control, simulations, and data communications are among his areas of expertise. He has over 30 years of industrial experience, is a member of ASEE, IEEE and Tau Beta Pi, is a Professional Engineer, and holds eight U.S. Patents.