

# First Year Computer Science

Viera K. Proulx, Richard Rasala, Jeff Raab

Northeastern University, Boston, MA 02115

[vkp@ccs.neu.edu](mailto:vkp@ccs.neu.edu), [rasala@ccs.neu.edu](mailto:rasala@ccs.neu.edu), [jmr@ccs.neu.edu](mailto:jmr@ccs.neu.edu)

<http://www.ccs.neu.edu/teaching/EdGroup>

The goals of the first year computer science course should be to introduce the student to the basic ideas of the field, the key concepts, techniques used by practitioners, and the basic skills needed to function in the field. We build camps that fight over the various issues: programming vs. general overview, theory vs. practice, basic skills vs. design, programming in the small vs. programming in context. We present here a comprehensive view of the topics and skills students should learn, organize them into categories, and suggest a pedagogy that can be used to teach these topics and skills.

The topics and skills can be organized into the following categories:

- Basic programming skills
- Techniques for encapsulation and building program or data components
- Design of program components and programs
- Algorithmic patterns for problem solving
- Computer Science overview - the types of applications
- Program analysis techniques
- Presentation and documentation skills

## Basic programming skills

Here the student learns the basic syntax of the language and the semantics of various constructs. The focus is on understanding what is the meaning of various statements, how the compiler interprets the statements, and how are the statements processed at run time. Students learn these skills in the context of small exercises and pattern tutorials that focus on only a few of the features at a time.

- identifiers, variables, basic data types
- initialization, assignment, basic operations (arithmetic, logic, relational, String)
- functions, arguments, parameter passing mechanisms
- representation of data
- defining classes, using objects
- basic data organization: arrays, pointers, structures, dynamic arrays, linked lists
- class interactions: composition, inheritance
- templates
- file input and output
- non-text data and file types: sound, graphics

## Techniques for encapsulation and building program or data components

The focus is on learning the mechanics for representing encapsulation. Students learn about the syntax for function headers, class definitions, using member functions, etc., and again learn how the compiler interprets these constructs and what effect they have at run time. Again, a series of short exercises and pattern tutorials are needed to support students' learning and practice of these techniques.

- functions with arguments and return values
- classes and objects

- class composition and inheritance
- interfaces
- templates
- operator overriding

## **Design of program components and programs**

It is not sufficient to understand the mechanics of creating encapsulations. Learning what to encapsulate is a more serious task. So, while the list of topics here may be the same as in the previous section, the focus is on learning to decide when we need a function, what the function should do, how a class is to be designed, what objects will be needed in a program, when templates are appropriate, etc.

The learning strategy here follows the standard learning theory:

- see the design feature in action
- use the design feature
- modify the design feature
- implement a design feature from a given specification
- design your own component or program

## **Algorithmic patterns for problem solving**

Rather than listing the specific algorithms students should study, we focus on the problem solving strategies illustrated in these algorithms. For example, when learning about the divide and conquer strategy, different algorithms help student learn when to apply this method, and what are the possible pitfalls. Similarly, student should be able to recognize a linear traversal pattern regardless of whether we traverse an array, a list of input data, or a linked list. The pedagogical approach should combine the presentation of several examples of a particular algorithmic pattern, description of this pattern, and analysis of its advantages and drawbacks. Some other algorithmic patterns may be presented in a similar manner.

- linear traversal with or without exit
- divide and conquer - logarithmic reduction
- recursion – efficient recursion vs. exponential explosion
- heuristic selection
- hashing as a storage-lookup technique
- relationship between algorithms and data structures

## **Computer Science overview - the types of applications**

The following categories describe a number of techniques used in applications that appear in the real world. Students should learn about the breadth of computer science by working on a number of scaled down examples of real applications. In each instance, they should understand the underlying principles used to solve the particular problem. These techniques should be clearly highlighted, so the student are aware of the transfer of the basic skills to the problem at hand. Students begin to see the need for understanding the basic techniques and can see the power of using these techniques in the context of real examples.

We list the techniques that occur repeatedly in the most common applications:

- cumulative results
- scaling, transformation and table lookup, conversions, filters, encryption
- search methods
- organizing data - sorting, linking, referencing
- visualization of data/information, modeling
- graphics, animation, morphing

- simulations
- generators and little languages - formal grammars, rewriting
- data-driven programs
- menu-driven programs
- event driven applications; interactions

For each of the CS applications student should see real examples, work on a scaled down problem that illustrates the key problem solving strategies and learn these techniques in apprentice style. In addition, student should be able to learn more about the problem in supplementary reading or during the lectures.

## **Program analysis techniques**

Students need to be exposed to the formal methods for measuring the correctness and effectiveness of their programs and algorithmic solutions. Additional formal training in discrete mathematics is a suitable co-requisite of the first computer science course. If such course is not available, some of these topics may need to be covered in the CS course, or some of these topics may be covered in an outline form and be reinforced in later courses.

Many of these techniques can be presented in the context of the pattern exercises used to practice basic skills. Others should be presented with the algorithmic patterns for problem solving.

- methods for space/time usage estimates
- complexity analysis
- testing methods (unit, composite, regression test)
- logical reasoning, proofs, pre-conditions, post-conditions

## **Presentation and documentation skills**

Students need to write programs that are well documented, readable, and correct. They should also be able to explain what they and their programs do. In the context of a first year course, this includes:

- writing code with good organization, style, and documentation, specifically
  - inline comments
  - external comments
  - use of white space
  - identifier naming
  - indentation
- documenting program specification and design
  - be able to describe the overall purpose of a class
  - explanation of member variables including any constraints that will be enforced
  - description of member functions including pre-conditions and post-conditions
- writing reports on what they have accomplished and what pitfalls they encountered

This can be learned not only by enforcing discipline when students are writing code, but also by asking them to read well written sample code and to review code written by their peers. Additional documentation methods (JavaDocs, UML) may be included, depending on the instructor's preferences.