An Internet Course in Software Development with C++ for Engineering Students

Yosef Gavriel, Robert Broadwater
Department of Electrical and Computer Engineering
Virginia Tech

Abstract

This paper discuss an experimental course in software development in C++ for graduate and undergraduate students in engineering, that was completely delivered through the Internet during the 1997-1998 time period. The course assumed no previous knowledge or background in C or C++ programming. Topics covered included: fundamental types in C++, basic operations, control structures, programming style and software craftmanship, functions, classes, constructors and destructors, object oriented analysis, object oriented design, operator overloading, inheritance, polymorphism memory management exception handling, function and class templates, and template libraries.

I. Introduction

At the time the course was developed, on-campus servers were having problems handling the large number of students enrolled in some courses that made extensive use of the Internet. This course was to be offered to both on-campus and off-campus students. To avoid slow server response, the course was designed so that the majority of the materials to be used by the students (i.e., the lectures) could be downloaded and utilized on the student's machines.

II. Course management

The students were able to use the Internet to ask questions and participate in discussion groups, download course files, submit homework, and take tests. Course material was available in a web site for viewing and download. The Internet interactions included chat rooms, virtual office hours, and a Frequently Asked Questions bulletin board. Student were able to review assignments, work through hypertext lectures relating to the reading assignments, review material via interactive questions, do homework, and have the capability to copy code directly from lectures and homework to a Windows based C++ compiler. Programming examples were motivated from small and simple circuits from introductory courses in electricity.

III. Course materials

The lectures were written in a hypertext language that provided jumps and popup windows. The popup windows were extremely useful for explaining code. For instance, consider the following line of code which is defined at global scope in a file:

int n;

where the variable n is hot spotted to provide a jump into a popup window with the following explanation:

A global int which is available to all functions defined below this point in the file.

At the end of each lecture a series of hot spotted fill-in-the-blank and true/false questions were provided. These questions were designed to provide a review of the major new concepts introduced in the lecture. An example of such a question is

When you write code, one way you communicate with other programmers is to embed _____ in the code.

By picking on the hot spotted blank in the above statement, the student would see the word "comments" appear in a popup window. All together, over 400 hot spotted questions were developed for the course.

IV. Software development practices and principles

Throughout the course, emphasis was placed on nine software development practices and principles [1,2]:

- Experiment in small programs
- Step through the code line-by-line
- Programming is first of all communication
- Analyze and plan before you act
- Clearly understand your program
- Trail-and-error development
- No-duplication principle
- Keep-it-simple principle
- Objects are characterized by behavior

The students were encouraged to experiment with each new concept in the smallest program possible. In writing the program the student should be experienced with all parts of the program except the new concept that is to be mastered. In other words, design a number of small experiments in which mistakes will be made, and from which understanding will be increased. Learning new programming concepts can be viewed as a "process of carefully planning small mistakes." [2]

Simple use of the debugger was introduced early in the course. This aided the students in their experimentation with new concepts. Also, it was emphasized that good professional programmers step through their code line-by-line in the debugger.

It was emphasized that programming gurus write readable code. Programming is first of all communication with other programmers, and second of all communication with the machine. If

the programmer is the only person that will ever look at the code, then he is still involved in an exercise of communicating with himself.

There is always tension between thinking (planning to write code) and acting (writing the code). However, experience has shown that the problem for students new to programming is not an excess of thinking prior to writing. Students were told not to write code until a plan was in place that was clearly understood. Furthermore, if they were deeply involved in coding and experienced a major new understanding of the problem, the students were told that the best decision may be to begin coding from scratch.

A strong intellectual grasp of a program should be obtained prior to relinquishing the program to the compiler. If a student resorts to praying that their program compiles and runs OK, then their intellectual grasp has fallen short.

Software is subject to iterative revision and improvement. Once a program compiled successfully, then the student's job was not finished until testing was performed.

Code that is used in more than one location or code that is common to more than one type of object should be maintained in a central location, either a function or a base class. A set of calculations should occur only once.

Often algorithms are selected to improve efficiency at the expense of complexity. In general, this should only be done if the decrease in execution time is really needed. Student's were encouraged to avoid code complexity and to write the simplest, most understandable code.

From experience, objects should only expose behavior. Attributes often need to be changed or extended, but the "correct behavior" should be a constant. Students were encouraged to characterize objects by behavior and not attributes.

V. Pencil and paper type circuit solutions with objects

DC circuits were used to illustrate the concept of using a set of interacting objects to solve a problem. It should be possible to use the objects in a manner analogous to pencil and paper solutions. As to be shown in the next section, with the addition of a base class and the concept of topology management, problems too large for hand calculations may still be solved by the computer.

Consider the dc circuit shown in Figure 1. Assume that classes of type VoltageSource and Resistor have been defined. Instantiating objects corresponding to those of the circuit

VoltageSource e(100);
Resistor r1(10), r2(10);

From the above, the voltage source "e" is a 100 volt source and each resistor is a 10 ohm resistor. Note that the constructors can check for unreasonable values. A C++ statement may now be

written, just as we would with pencil and paper, to solve for the current flow through the voltage source

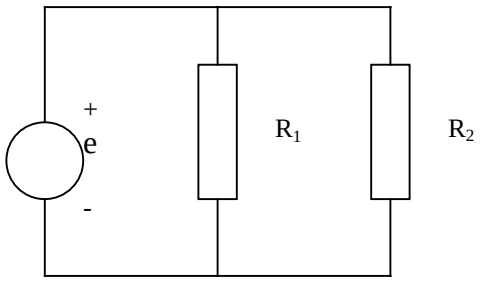


Figure 1: Example illustrating DC circuit without inheritance

e.i() = e.v() * (r1.g() + r2.g());

where e.v() returns the voltage of the voltage source ri.g() returns the conductance of the resistor r_i, i = 1,2 and e.i() returns a reference to the current through the voltage source.

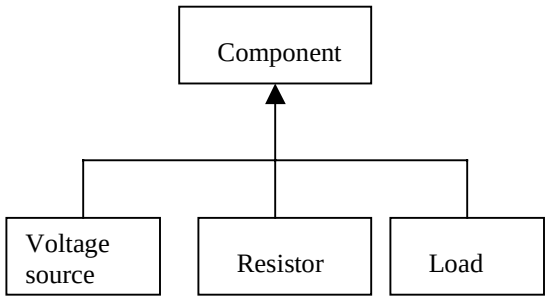


Figure 2: DC circuit modeling classes

VI. Algorithm for automatic circuit solution

Following the introduction of objects to model the various components in the dc circuit, a base class, the Component class, was defined. Specific components, such as voltage sources, resistors, and loads (loads were modeled as current sources), then inherit from the Component class, as shown in Figure 2. The Component class defined the following pure virtual functions corresponding to voltage and current for the component,

virtual double v() = 0;
virtual double i() = 0;

As long as any component that inherits from the Component class overrides the above two virtual functions, then the algorithm for solving the circuit will continue to work without re compiling any of the existing circuit modeling classes.

The Component class included four pointers of type Component that were used to manage a doubly linked list of components. Two of the pointers were declared as static and were used to store the start and end of the doubly linked list. With this design, it was now possible to solve large radial circuits using an algorithm that involved two iterations. Consider the following main program which uses the classes to solve the circuit shown in Figure 3.

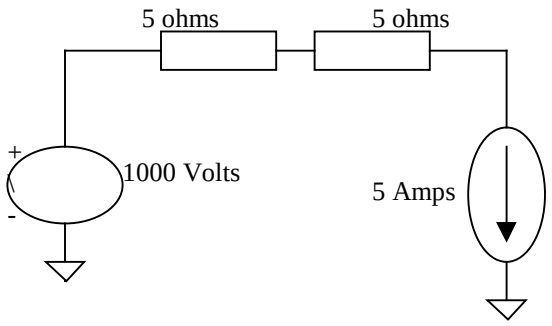


Figure 3: Example circuit solved with classes on Figure 2.

```
void main()
{
Component *pCmp,
        *pB;

// Create source and set topology pointers for source
pCmp = new VoltageSource( 1000. );
Component::setStart( pCmp );
pCmp->setBack( NULL );

pB = pCmp;

// Create first resistor and set topology pointers
pCmp = new Resistor( 5.0 );
pCmp->setBack( pB );
pB->setForward( pCmp );
pB = pCmp;

// Create second resistor and set topology pointers
pCmp = new Resistor( 5.0 );
pCmp->setBack( pB );
pB->setForward( pCmp );
pB = pCmp;

// Create load and set topology pointers
pCmp = new Load( 50. );
Component::setEnd( pCmp );
```

```
pCmp->setBack( pB );
pB->setForward( pCmp );
pCmp->setForward( NULL );

// Calculate component currents with reverse trace
pCmp = Component::getEnd();
while( pCmp ){
        pCmp->i();
        pCmp = pCmp->b();
}

// Calculate node voltages with forward trace
pCmp = Component::getStart();
while( pCmp ){
        pCmp->v();
        pCmp->print();
        pCmp = pCmp->f();
}

// Clean up memory
pCmp = Component::getStart();
while( pCmp ) {
  Component *pF = pCmp->f();
  delete pCmp;
  pCmp = pF;
}

} /* End Main */
```

The Component class member functions illustrated in the above code are as follows:

void setStart(Component*) sets a static class member variable to point to the first component in the circuit

void setEnd(Component*) sets a static class member variable to point to the last component in the circuit

void setBack(Component*) sets the backward pointer for the doubly linked list

void setForward(Component*) sets the forward pointer for the linked list

Component* getEnd() returns the pointer to the last component in the circuit

Component* getStart() returns the pointer to the first component in the circuit

Component* b() returns the pointer to the backward component in the doubly linked list

Component* f() returns the pointer to the forward component in the doubly linked list

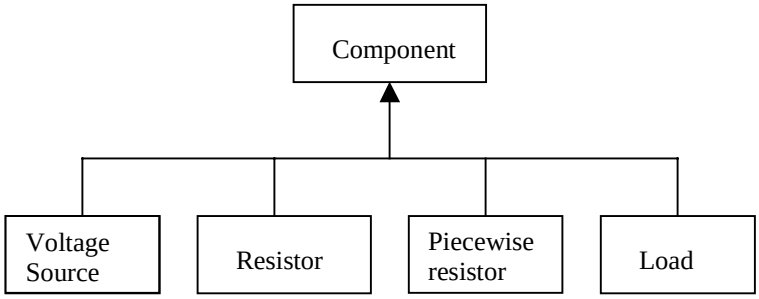
virtual void print() may be overridden by each derived component class to print results pertaining to the component

virtual double v() must be overridden by each derived component class to calculate the voltage for the component

virtual double i() must be overridden by each derived component class to calculate the current for the component.

The above code illustrates the use of base class pointers to manipulate derived class objects. Note that a new type of component may be defined and the code for solving the circuit, represented by the two while loops associated with the reverse and forward trace, does not need to be changed. This is illustrated in the next section.

Figure 4: Class Inheritance Diagram for Project



VII. Extending the DC circuit model components

Once the students understood the circuit modeling classes presented in the previous section, they were given a project to create a class to model a piecewise linear resistor. The resistor could have any number of linear segments (i.e., breakpoints), and hence the students had to use dynamic memory management to model the curve in the voltage-current graph of the device. Figure 4 shows the dc circuit modeling class inheritance diagram with the PieceWiseResistor class added.

Without modifying any of the previously existing classes or the algorithm for solving the circuit, the students could solve circuits like that shown in Figure 5. This exercise illustrates software extensibility with inheritance and polymorphism.

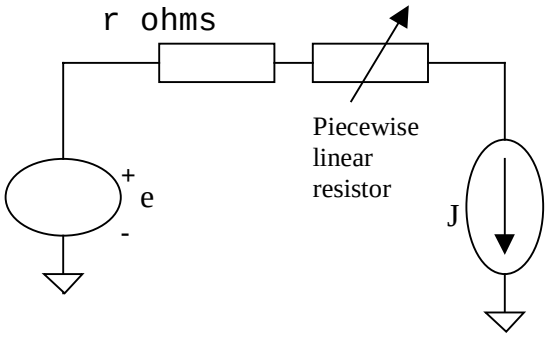


Figure 5: Example circuit for student project

VIII. Conclusions

The course was very successful and popular among the students. Off-campus students were the most pleased with the structure and development of the course. The use of easily available windows based tools [3] and a textbook [4] targeting engineering students were also very helpful. The authors are currently investigating the applications of the proposed approach to other programming oriented courses at Virginia Tech.

Bibliography

1. P. H. Winston, **On To C++**, Addison-Wesley Publishing Company, 1994.

2. S. Maguire, **Writing Solid Code**, Microsoft Press, 1993.

3. Microsoft Visual C++ Version 4.

4. D. M. Capper, **C++ for Scientists, Engineers and Mathematicians**, Springer-Verlag, 1996.

ROBERT BROADWATER
Robert Broadwater is a Professor at Virginia Tech where he teaches courses in computer-aided engineering design and object-oriented analysis. His research is primarily in the area of computer-aided engineering applications for electrical distribution system analysis, design, and operations. He received his B.S., M.S., and Ph.D. degrees in Electrical Engineering from Virginia Tech.

YOSEF GAVRIEL
Yosef Gavriel (a.k.a. J.C. Yosef Gavriel Ben Abraham Tirat-Gefen DeSouza-Batista) is an Assistant-Professor Tenure-Track of Computer Engineering at Virginia Tech. He has been involved in software engineering for scientific and engineering applications for over 13 years in both academia and industry. Dr. Yosef Gavriel has taught courses on C++, VHDL and Digital Design. He is a Senior Member of IEEE, HKN, Sigma-Xi, ACM and ASEE.