Abstract

In the Computer Science and Engineering Department at Michigan State University (CSE/MSU), we use timed programming exams in our introductory programming courses to assess both individual student programming skills and course instruction. Administration and design of these exams presented challenging problems. In this paper, we describe these problems and how we solved them in our programming exam system. Additionally, we describe the exams themselves and the particular outcomes under assessment. These courses at CSE/MSU use C++ for programming; however, the issues and methods discussed apply to any programming language.

Introduction

Working on programming projects is perhaps the most common method for students to learn the skills necessary for programming. The use of individual programming projects in teaching is grounded in modern pedagogical theories, such as problem-based and active learning.\(^1,2\) Programming projects may be graded to help in assessing student progress in learning and effectiveness of instruction, and also to motivate students to carry out the projects and to provide them constructive feedback. However, using programming projects in assessment is problematic. Some students spend an unusual amount of time on programming projects or receive too much help in doing the work. Moreover, inappropriate copying of code developed by others is also common.

Written exams often provide the primary means for assessment in large introductory programming courses. Unfortunately, it is difficult to determine how well questions on written exams correlate with programming skills. Exams in large introductory programming courses are often multiple choice or short-answer. Such questions typically test knowledge of specific aspects of programming features, rather than ability to devise a solution and realize the solution in code. Moreover, feedback from students indicates that they feel their performance on such exams is not a good measure of their programming skills. They find multiple choice questions to be “tricky” and complain of difficulty expressing themselves in short answers. In fact, communication skills may be more prominent factors in determining how well students perform on written exams than are programming skills. For these reasons, CSE/MSU has started using controlled programming examinations in the introductory programming courses for the purpose of assessing programming skills of individual students and adequacy of instruction in programming. We use programming exams to augment more traditional assessment techniques, including individual and small group programming projects and written examinations. There are two ancillary benefits of using programming exams in assessment. First, feedback from our
students shows that they accept the programming exams as an appropriate assessment means. Second, we estimate that the human effort in grading a lab exam is approximately half of that needed to grade a program project done for homework.

Two programming exams were successfully piloted in CS2 during fall semester of 2002. As a result, during spring semester of 2003, we added two new programming exams to CS2 and also piloted two programming exams in CS1. Students in our CS1 and CS2 courses sign up for a scheduled two-hour closed lab session each week where they get supervised exercise in C++ from a teaching assistant (TA). During spring semester of 2003, we used two of the labs in CS1 and four of them in CS2 for programming exams; the results of these exams are reported below.

Computing Support for Assessment

We are concerned with the interaction of two pedagogical issues: (1) the need for assessment of both individual student achievement and course instruction, and (2) the use of current computing resources, including extensive network connectivity and programming development environments, in such assessment. Modern computing resources provide considerable support for teaching and learning. Interestingly, these same resources create obstacles when assessing individual student achievement. This paper describes some of these obstacles and the problems that they present for administration and design of programming exams, and explains how we addressed them.

Courseware is now commonly used for managing courses and student work, and is commercially available. Some systems provide the capability of generating exams for individual students on the fly: these often randomly select a subset of questions from a large set of available questions. John English describes a custom system that provides both objective test questions and programming problems that require program development and compilation. Two large courses at MSU, CSE 101 and CSE 131, use custom systems to deliver problems that are made unique by changing parameters in a base problem. The CSE 101 course, which currently enrolls about 2000 students per semester, uses “bridge tasks” to assess student progress. Bridge tasks are directly related to desired outcomes; they are graded pass/fail and students cannot proceed to the next assessment until they have passed the current one. Web-based testing systems depend on problem uniqueness and authentication to ensure that the work being assessed was performed by a particular individual. Usually, authentication depends on the student being present in a particular lab at a particular time.

We use similar techniques in implementing our programming exam system. Our system generates exams by selecting one or more problems from a partitioned collection of programming problems. Programming problems within a partition are designed to test similar programming skills. We allow that several students will receive the same exam, but we take strong measures to ensure that each student works her/his exam individually without access to other persons, the web, email, personal files, or other (non-electronic) media. Our system requires authentication at the time and place of a prescheduled lab session. (Those interested in acquiring our software and scripts can contact author Paul Albee at albee@cps.cmich.edu.)
Outcomes to be assessed

We believe that the following outcomes are important for computer science students and programs. These outcomes are only those that we assess via our programming exams and are not intended to be exhaustive. Two relate to our overall computer science program and are addressed by other courses in addition to our first two programming courses.
1. A graduate can solve problems and design computer solutions.
2. A graduate can develop programs on both Windows and UNIX platforms.

Some particular outcomes for CS1, our first programming course, include:
3. Student can use imperative control constructs in problem-solving.
4. Student can define and use functions with both value parameters and reference parameters.
5. Student can use a vector in solving a problem, including operations on vectors, such as initialization, I/O, and passing a vector to a function.
6. Student can use streams for I/O, including file streams and operations that manipulate streams.

Some particular outcomes for CS2, our second course, include:
7. Student can use an array in solving a problem, including operations on an array, such as initialization, I/O, and passing an array to a function.
8. Student can do object-centered design to develop a C++ class.
9. Student can use recursion in problem-solving and can create recursive functions and recursive data structures.
10. Student can define and use linked data structures (with pointers).

The outcomes (7) and (8) for CS2 are also supported by work in CS1.

CS1 moves slowly in the first three weeks of the semester because many students have no prior programming experience. The first programming exam in CS1, which is administered in week 5, can therefore assess only outcome (3). We attempted to assess outcomes (4)—(6) in the second programming exam, which is administered in week 10. However, as discussed in a subsequent section, our results indicate that the second programming exam was too ambitious. Each programming exam in CS2 was designed to assess one of the outcomes (7)—(10).

Problems Encountered

Administration and design of the programming exams presented challenging problems. First, because lab sections in CS1 and CS2 run from Tuesday morning to Friday afternoon, we had to ensure that students enrolled in the lab sessions that meet later in the week could not learn specifics about their exams that those enrolled in the earlier lab sessions could not also learn. Second, we had to prevent students from communicating with each other during an exam and from accessing files, email, or the Internet, while at the same time providing them with an editor, compiler, and other support tools. Third, because CS1 and CS2 acquaint students with different platforms, we had to create exam environments for both Windows and UNIX. This section explains how we dealt with the first problem. The exam environments were designed to deal with the second problem. Details of the exam environments are given in subsequent sections.
For each of the CS1 programming exams, we post the entire collection of possible problems on the course website a week in advance of the exam. The collection of programming problems is partitioned into “problem sets,” and students are informed that they will be asked to solve one problem from each of the problem sets. Our programming exam environment for CS1 provides students with a hyperlinked on-line summary of the syntax and semantics of C++, including example code using the programming features needed to solve the programming problems; we post this summary to the course website at the same time that we post the problem sets.

Programming assignments in CS1 are normally due at midnight on Mondays; but no assignment is due in weeks prior to a programming exam. During the week preceding a programming exam, we encourage students to familiarize themselves with the on-line summary of C++ and the posted problem sets; to design solutions to all the problems in the problem sets; to practice programming a representative subset of the problems; and to participate in a monitored electronic discussion of the problem specifications and of alternative solutions to the posted problems. By these means, we provide all students equal opportunity to learn the kinds of problems that they will find on their exam, clarify the problem specifications, and obtain answers to programming questions.

We use the MSU WebTalk conference tool for on-line discussion of the programming exam problems. WebTalk supports threaded conversations on multiple topics. We configure WebTalk with a topic for each programming exam and, under these topics, with conversations for each programming set and a conversation for general concerns about the exam. A students posts questions to the appropriate conversations, and may also post responses to questions that others have posted. A TA is responsible to monitor and respond to postings at regular intervals in the afternoon and early evening hours throughout the semester. (We also use WebTalk for general course and programming questions.) During weeks prior to a programming exam, the instructors also regularly monitor and respond to postings on the programming exam topics.

In CS2, we do not publish a problem set in advance, but we have a large set of roughly equivalent programming problems. Although each student solves only one problem, all students are expected to be able to solve any of the problems so we do not split hairs over relative difficulty. The programming exam is given in one source file, which the student develops into the submitted solution. Instructions for taking an exam and instructions common to all programming problems are contained in an exam “template.” The exam generator inserts comment statements into the template describing specific programming problems. In addition to instructions and problem requirements, the comments contain sample data with required output and perhaps a function or function prototype. There are always some executable statements so that the student has examples of syntax.

Unix System Approach

Our UNIX lab exam system is a menu application written in a high-level scripting language that uses standard UNIX utilities to create a controlled environment for taking an exam. We create special lab exam user accounts so that students coming to their closed lab session find the same physical environment but very different system and personal resources. In particular, they do not have access to their own files, email, or the internet. We provide the password for an exam
We configure the FVWM\textsuperscript{16} window manager to eliminate shell access for these accounts. One terminal running a special menu script appears automatically on the screen at login to a lab exam account. We configure FVWM so that the only functionality it provides is the ability to produce more of these menus, if necessary, and to logout. Configuration of the accounts is straightforward, primarily disabling existing functionality in two configuration files (.fvwm and .xinitrc).

The menu application is written in Perl and consists of approximately 1000 lines of code. The graphical portion is rendered and displayed using Perl's curses module\textsuperscript{17}. The menu allows students to generate an exam, edit their exam code, compile their exam code, run their executable program, and hand in their exam code. No command line is available. Students cannot use email or access the Internet.

The menu application constructs lab exams. It authenticates the student's identity by requesting their UNIX username and password. Using built-in Perl functions, it checks the password entered by the student against the password stored by the UNIX system. If the passwords match, the system selects an exam problem from an available pool of problems and creates a lab exam with student information, such as name and student number. The system also ensures that students in adjacent seats do not receive the same problem.

The menu application uses pico\textsuperscript{18} as the text editor. Pico was chosen because it is easy to use and does not provide shell access. The application invokes pico with the "-o" option, which limits the ability of pico to read and write files. Specifically, only files within the lab exam account may be read or written.

Files that the student generates using pico within the lab exam system are left in the exam account’s directory. Later, teaching assistants log into these accounts remotely via SSH\textsuperscript{19} and use a shell for grading. A system administrator resets the lab exam accounts before every lab exam, clearing files created in earlier lab exams. Students may print their program file during an exam and, in fact, submit a printed version to the TA for grading and for backup to the electronic record. The TA monitors use of the printer and all student activity.

Windows System Approach

The Windows exam system utilizes a client-server approach for administering exams. The server is a secure web server running Apache\textsuperscript{10} and PHP\textsuperscript{11}. The web server communicates with a password-protected MySQL\textsuperscript{12} database server that stores and serves all questions, and stores student responses. All incoming and outgoing network traffic of the machines in the examination room is mediated through a Squid\textsuperscript{13} proxy server configured as a router. The exam environment provides four levels of access to the examination system: (1) exam question administration, (2) course-level exam administration, (3) section-level exam administration, and (4) student exam access.
The first level, exam question administration, generates both the questions for an exam and the corresponding section-question matrix. All exams are generated at this level and are made available to the other access levels. Instructors use course-level exam administration access to grant time extensions to students and to examine students’ submissions. Instructors have access to all sections of a course. Teaching assistants use section-level exam administration access to monitor exam progress and to grade exams. A teaching assistant has access to only his/her assigned sections’ exams.

Students use the fourth level of access to take an exam. A student logs into the exam system on an exam machine and is issued a set of programming problems, generated by the exam system with respect to criteria described below. A student uses the development environment installed on his/her exam machine to program solutions to the problems. We provide Microsoft Visual Studio .NET, the development environment that students also use in their normal lab sessions. The implementation file for a student’s problem solutions is then uploaded to the secure web server, and stored in the exam solution database.

Administering an exam for a given course requires several databases to be in place. The first set of databases identifies the students in the course by section, the teaching assistants for the course by section, and the faculty for the course by section. It also includes an override database to facilitate permitting a student to take an exam at some alternate time. These databases are generated automatically from electronic course rosters. The second set of databases describes the physical layout of the examination room. By taking into account the relative locations of exam machines, we ensure that students in adjacent seats do not receive the same questions. The third set of databases details the questions for the exam. A section-question matrix is used to choose a subset of the question pool for each section. This approach allows us to present different exams to students over the course of a week. The fourth set of databases stores the students’ submissions. Students are told to submit solutions early and often. By default, the TA grades the latest submission that compiles. However, the student can request grading of an earlier submission in case a last minute edit breaks an earlier partial solution.

During exam sessions, the machines in the examination room are configured to use the proxy server as their router, allowing us to prevent access to web resources. Using the proxy server also allows us to prevent electronic communication between students during the exam.

We encountered two problems during deployment of this exam system related to slow exam machines and hard time limits imposed by the examination system. Occasionally, a machine slowed down so much that it became unusable for taking the exam. We hypothesized that the slowdown was due, in part, to an automatic update of the machine’s operating system, or some other automatic network access. We therefore instructed each TA to reboot all exam machines at the end of the exam period, prior to the start of the next exam. With 20 minutes between exam periods, this task was easily accomplished and seemed to fix the first problem. The second problem was one of timing. The examination system stops accepting submissions at the end of the examination period; therefore, students must submit their work by the end of the period. However, differences in the system times on the examination machines and web server made it difficult to reliably enforce this requirement and the abrupt termination distressed students. The remedy for the timing issue is to have the teaching assistants announce when work...
must be submitted at the start of the exam and warn students of the time as the deadline
approaches, to extend the submission deadline to a few minutes past the official end of the
examination session, and to ensure that system times are synchronized between the web server
and the exam machines.

Programming Exam Results

For the programming exams in CS1, we defined successful mastery for a student as a grade of
60% or more, which the student could obtain by correctly coding the first two of three
progressively more difficult programming problems. Tables 1 and 2 summarize our results for
outcomes assessed by the programming exams. The columns in Table 1 give, respectively, the
section number, the number of students in a section, the number of students scoring 60% or
higher on the first programming exam, and the number of students scoring 60% or higher on the
second programming exam. Table 2 summarizes the percentages of students that demonstrated
mastery on the two programming exams (first row) and the average grades on the exams (second
row). Note the difference between average grade on an exam and the percentage success rate.
The average grade for the first programming exam was 69%; however, only 64% of the students
were deemed to have mastered elementary control constructs. We take these numbers as
measures of class learning, discussing these results further in the next section.

Table 1. Programming Exam Results for CS 1 by Lab Section

<table>
<thead>
<tr>
<th>CS1 Sec #</th>
<th>N</th>
<th>P1:control</th>
<th>P2:func+Vectors+Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>9</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>
Table 2. Summary assessment of learning outcomes: percentage of students scoring 60% or better.

<table>
<thead>
<tr>
<th>CS1 Summary</th>
<th>P1:control</th>
<th>P2:func+vectors+streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>% successful</td>
<td>64%</td>
<td>20%</td>
</tr>
<tr>
<td>avg. grade</td>
<td>69%</td>
<td>36%</td>
</tr>
</tbody>
</table>

For the programming exams in CS2, we defined successful mastery for a student as a grade of 75% or better and success for learning by the entire class as 75% of the students being successful. Tables 3 and 4 summarize results for outcomes assessed by the programming exams in CS2. These tables provide information similar to the previous tables, but use a 75% cut-off as an indication of mastery. As above, the average student grade can be contrasted with the success rate. The average grade for programming with pointers was 75%; however, only 67% of the students were deemed to have mastered pointers. Similarly, for programming using recursion, the average grade was 85%—a good number of students scored 100%—but only 76% of them were deemed to have mastered recursion.

Table 3. Programming Exam Results for CS 2 by Lab Section

<table>
<thead>
<tr>
<th>CS2 Sec #</th>
<th>N</th>
<th>P1:arrays</th>
<th>P2:class</th>
<th>P3:pointers</th>
<th>P4:recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>11</td>
<td>10</td>
<td>8</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>13</td>
<td>14</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 4. Summary assessment of learning outcomes: percentage of students performing 75% or better.

<table>
<thead>
<tr>
<th>CS2 Summary</th>
<th>P1:arrays</th>
<th>P2:class</th>
<th>P3:pointers</th>
<th>P4:recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>% successful</td>
<td>94%</td>
<td>93%</td>
<td>67%</td>
<td>76%</td>
</tr>
<tr>
<td>avg. grade</td>
<td>91%</td>
<td>92%</td>
<td>75%</td>
<td>85%</td>
</tr>
</tbody>
</table>

Discussion of Results

The results of mastery-level learning by the entire class on the CS1 programming exams were disappointing, with 64% demonstrating that they could use elementary control constructs and only 20% demonstrating that they knew how to use functions, vectors, and streams. We believe that several factors may help explain these results. The first relates to the wide range in the backgrounds and abilities of students in the course. CS1 and CS2 are entry courses into the CSE major as well as terminal courses for other majors. In a typical semester, between 25-30% of the students who start CS1 will quit—some officially drop the course and some realize too late that
they are not adequately prepared for the course and just stop participating. In contrast, a student
who enrolls in CS2 must have successfully completed CS1 or an equivalent course at another
institution. Thus, the second programming course enrolls a more homogenous group of students
and experiences a lower attrition rate. This situation in CS1 can certainly explain a 64% success
rate and an average grade of 69% on the first programming exam. Some of our colleagues at
other institutions report similar observations.

The large gap between expectations and performance on the second programming exam,
however, cannot be so easily dismissed and must be addressed by improved instruction on the
topics covered. It is well known that many students have difficulty with function parameters and
streams when they first encounter them. While most students are familiar with the concept of
vectors, understanding vectors in C++ requires some understanding of templates, another
unfamiliar and confusing concept for many students. In retrospect, we feel that we were too
ambitious in trying to assess so many advanced outcomes in one programming exam and so soon
after students were first introduced to them. In the future, we plan to use shorter programming
exams, each designed to assess a single outcome, and to administer more exams. Moreover, we
may replace some of the more advanced outcomes for assessment in programming exams with
more elementary, but equally important outcomes.

In CS2 (Spring 2003), we were pleased that students did well on arrays (94% success) and
classes (93% success) and conclude that our students are learning those topics well. Some
weakness was shown on the use of pointers/links (67% success rate) and on recursion (76%),
although by our definition the class successfully learned recursion. Pointers and recursion are
known to be difficult for some students. Lack of success is correlated: most of the unsuccessful
programming exam grades are due to students who have more than one low grade. These results
support the hypothesis that students reaching a goal level of both abstraction and programming
skills will succeed. We will experiment with increased practical work on the use of pointers to
see if we can increase success for that outcome. Although we have not previously discussed the
relation of CS1 and CS2 in this paper, we know that success in programming tasks in CS2 is in
part due to the learning from CS1. We conclude this, not only from the success on tasks in CS1,
but also from an analysis of test items from the first written exam in CS2, which show that
students understand the abstraction of object-centered programming and, in general, can
correctly write member functions for simple object classes. We note that the topics for P1:arrays
and P2:class were begun in CS1, so success on these is attributed to both courses.

Concluding Discussion

Hard work in setup has resulted in a system that meets our requirements. We believe the
assessment system is valid and acceptably convenient and safe. Since the environment supports
development of C++ programs, it is not bulletproof. It does assess student skills and students
accept that it serves that purpose, according to their written course evaluations. In addition, we
are assessing the effectiveness of our courses in teaching important programming skills. We are
indebted to our TAs for providing careful qualitative assessments of student work.

There is a tradeoff in the number of skills assessed and constraints on the number of exam
periods available. The CSE 101 course \(^6\) may assess 3 to 7 skills in a single bridge task and they
carefully grade each. It is, of course, important that students understand the objectives of each programming exam. Publishing problem sets in advance helps with fairness when exams are administered over a week, but does not entirely eliminate the advantages for students enrolled in lab session that meet later in the week. CS1 problem sets were published a full week in advance of an exam. The timing of the activity on WebTalk relating to the programming exams indicates that this strategy was sound—the low volume of questions in the first half of a study week suggests that few, if any, students started studying the problems sets more than a week in advance of their scheduled lab session.

Overall student performance has shown that learning has met expectations in most cases for CS2 and has shown where more attention is needed. There are other benefits. Whereas most students increased their confidence in their programming ability, some realized that they were relying too heavily on outside help to complete programming projects. Also, since programming exams use problems that are shorter and more clearly defined than programming projects, grading time is reduced relative to programming projects.

References


4 http://www.iticse2002.dk/conference/sessionmat/assessment/2.pdf. Slide presentation by John English on his use of system administered exams that include programming.

5 www.cse.msu.edu/~cse101.


7 www.cse.msu.edu/~cse131

8 www.cse.msu.edu/~cse101. In particular, pages on the philosophy and mechanisms of bridge tasks (BTs).

9 http://www.msu.edu/user/pmhunt/. WebTalk Central.


12 http://www.mysql.com/. “MySQL AB.” MySQL.


14 http://www.fvwm.org/ documentation for FVWM
GEORGE STOCKMAN

Research interests include computer vision and graphics. Teaching interests are in computer vision, artificial intelligence, data structures and programming. He received a BS and an MAT in Math-Ed from East Stroudsburg State and Harvard, respectively, and the PhD in Computer Science from Maryland in 1977. He has been at MSU/CSE since 1982 and has taught CS2 about 25 times.

LAURA DILLON

Research interests center on concurrent software systems, programming languages, and software engineering. She received the Ph.D. degree in CS from the University of Massachusetts in 1984 and joined MSU/CSE in 1997, where she is currently Acting Chair. Active in IEEE and ACM since 1980, she is a member of the editorial board of ACM Trans. on Soft. Eng.& Method. and of the Executive Committee of the ACM Special Interest Group in Soft. Eng.

PAUL ALBEE

A PhD candidate at Michigan State University, and Instructor at Central Michigan University, his research interests include volumetric data analysis, computer based testing, and computer assisted education. He has a BS from Tri-State University in computer science, and an MS from Bowling Green State University in computer science.

JONATHON OLESZKIEWICZ

Research interests include parallel computing systems and operating systems. He has been both a Teaching Assistant and Research Assistant at MSU/CSE and will receive his MS in May, 2004. He was the 2002 Outstanding Graduating Senior at Grand Valley State University.

Appendix 1: Sample CS1 Programming Set

//*************************************************************************
//
// Name: quotient
//
// Purpose: Find the quotient of two integers by repeated subtraction
//
// Receives:
// The dividend (non-negative integer)
// The divisor (positive integer)
//
// Return:
// The quotient
//
// For example: quotient(9, 2) returns 4, which is found by counting
// the number of times that 2 objects can be subtracted from a set
// that starts out with 9 objects in it
//
// Number of subtractions
//                        -----------------------
//      9 - 2 = 7             1
//      7 - 2 = 5             2
//      5 - 2 = 3             3
//      3 - 2 = 1             4
//
// Note: You are NOT allowed to use any functions from the math library
// for this problem.
//
//****************************************************************************
int quotient(int dividend, int divisor);
//****************************************************************************

//**************************************************************************
// Name: remainder
//
// Purpose: Find the remainder of one integer divided by another by
// repeated subtraction
//
// Receives:
// The dividend (non-negative integer)
// The divisor (positive integer)
//
// Returns:
// The remainder
//
// Example: remainder(17, 5) returns 2, which is found by repeatedly
// subtracting 5 objects from a set that starts out with 17 objects in
// it, and returning the number of things remaining in the set when it
// has fewer than 5 objects remaining in it.
//
//    17 - 5 = 12
//    12 - 5 = 7
//    7 - 5 = 2
//
// Note: You are NOT allowed to use any functions from the math library
// for this problem.
//
//****************************************************************************
int remainder(int dividend, int divisor);
//****************************************************************************

//**************************************************************************
// Name: gcd
//
// Purpose: Find the greatest common divisor of two positive integers by
// brute force
//
// Receives:
// Two positive integers N and M
//
// Returns:
// The greatest common divisor of N and M
int gcd(int n1, int n2);

int lcm(int n1, int n2);

Appendix 2: Sample CS2 Programming Problem

SKIM THE ENTIRE QUESTION BEFORE GETTING IMMERSED IN PROGRAMMING
In this programming exam problem, demonstrate your skills in working with linked structures.
Do not create a class -- just use procedural programming with the main and other required functions.
DO NOT USE cin, just use cout to demonstrate the operations.
Follow the instructions in the comment statements. They will
// give the steps required to develop and test your source code.
//
// Below is an example of a queue of int implemented via a
// linked chain of nodes with front pointer F and rear pointer R.
//
// F *---> 52 *---> 79 *---> 35 *---> 44 /
//     ^
//     |
//     * R
//
// (1) Complete the definition of the node structure.

struct Node      // structure of list node
{
    
};

// (3) Complete the definition of the function to print out all information in the
// linked queue whose first node is *Front. For each node, the pointer to the
// node is also printed out with the integer and link to the next node.
// Test this function by printing the data structure shown above.

void checkQueue ( Node* Front )
{
}

// (4) Complete the definition of the function to insert a new data element into the queue.
// The function should return true if the operation is completed and false if it cannot be completed.

// (5) Complete the definition of the function to reset a queue Q by returning all of its nodes to
// heap storage and setting F and R to NULL.

void reset ( Node* &F, Node* &R)
{
}

int main ()
{
    cout << "\n*** Lab Exam 3: queue1 ***\n";

    // (2) Write declaration and assignment statements in the main program to build the data
    // structure shown above containing the integers 52, 79, 35, 44.

    // (3) Demonstrate the checkQueue function.

    // (4) Write and demonstrate a function to insert a new integer into queue Q. Show how
    // it works by inserting 37 and then 54 into the example queue above and calling
    // checkQueue. Also show that it works when inserting an integer into an empty queue,
    // where both F and R are NULL.

    // (5) Demonstrate the reset function here.

    return 0;
}   // end main

Proceedings of the 2004 American Society for Engineering Education Annual Conference & Exposition
Copyright © 2004, American Society for Engineering Education

Page 9.358.14