LESSON SET

1 Introduction to Programming and the Translation Process

PURPOSE

1. To become familiar with the login process and the C++ environment used in the lab
2. To understand the basics of program design and algorithm development
3. To learn, recognize and correct the three types of computer errors:

   syntax errors
   run time errors
   logic errors
4. To learn the basics of an editor and compiler and be able to compile and run existing programs
5. To enter code and run a simple program from scratch

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete Labs 1.1 through 1.4 in sequence. Your instructor will give further instructions as to grading and completion of the lab.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	2	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	6	
Lesson 1A				
Lab 1.1 Opening, Compiling and Running Your First Program	Pre-lab reading	20 min. (Including overview of local system)	7	
Lab 1.2 Compiling a Program with a Syntax Error	Familiarity with the environment Finished Lab 1.1	15 min.	7	

continues

Lab 1.3 Running a Program with a Run Time Error	Understanding of the three types of errors	15 min.	8
Lesson 1B			
Lab 1.4 Working with Logic Errors	Understanding of logic errors	15 min.	9
Lab 1.5 Writing Your First Program	Finished Labs 1.1 through 1.4	30 min.	11

PRE-LAB READING ASSIGNMENT

Computer Systems

A **computer system** consists of all the components (hardware and software) used to execute the desires of the computer user. **Hardware** is the electronic physical components that can retrieve, process and store data. It is generally broken down into five basic components:

Central Processing Unit (C.P.U.)	This is the unit where programs are executed. It consists of the **control unit**, which oversees the overall operation of program execution and the **A.L.U.** (Arithmetic/Logic Unit), which performs the mathematical and comparison operations.
Main Memory	The area where programs and data are stored for use by the CPU
Secondary Storage	The area where programs and data are filed (stored) for use at a later time
Input Devices	The devices used to get programs and data into the computer (e.g., a keyboard)
Output Devices	The devices used to get programs and data from the computer (e.g., a printer)

Software consists of a sequence of instructions given to perform some pre-defined task. These labs concentrate on the software portion of a computer system.

Introduction to Programming

A **computer program** is a series of instructions written in some computer language that performs a particular task. Many times beginning students concentrate solely on the language code; however, quality software is accomplished only after careful design that identifies the needs, data, process and anticipated outcomes. For this reason it is critical that students learn good design techniques before attempting to produce a quality program. Design is guided by an **algorithm**, which is a plan of attacking some problem. An algorithm is used for many aspects of tasks, whether a recipe for a cake, a guide to study for an exam or the specifications of a rocket engine.

Problem example: Develop an algorithm to find the average of five test grades.

An algorithm usually begins with a general broad statement of the problem.

$\boxed{\text{Find the average of Five Test Grades}}$

From here we can further refine the statement by listing commands that will accomplish our goal.

$\boxed{\text{Read in the Grades}}$ $\boxed{\text{Find the Average}}$ $\boxed{\text{Write out the Average}}$

Each box (called a node) may or may not be refined further depending on its clarity to the user. For example: **Find the Average** may be as refined as an experienced programmer needs to accomplish the task of finding an average; however, students learning how to compute averages for the first time may need more refinement about how to accomplish the goal. This refinement process continues until we have a listing of steps understandable to the user to accomplish the task. For example, **Find the Average** may be refined into the following two nodes.

$\boxed{\text{Total=sum of 5 grades}}$ $\boxed{\text{Average=Total/5}}$

Starting from left to right, a node that has no refinement becomes part of the algorithm. The actual algorithm (steps in solving the above program) is listed in bold.

```
Find the Average of Five Test Grades
    Read in the Grades

Find the Average
    Total = sum of 5 grades
    Average = Total / 5
    Write Out the Average
```

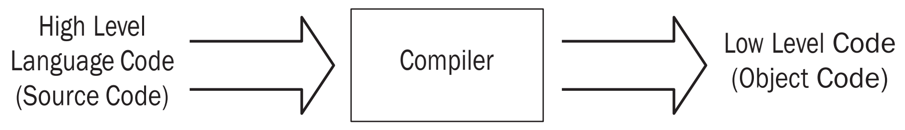
From this algorithm, a program can be written in C++.

Translation Process

Computers are strange in that they only understand a sequence of 1s and 0s. The following looks like nonsense to us but, in fact, is how the computer reads and executes everything that it does:

1001000111101010111001000111000100

Because computers only use two numbers (1 and 0), this is called **binary** code. can imagine how complicated programming would be if we had to learn this very complex language. That, in fact, was how programming was done many years ago; however, today we are fortunate to have what are called **high level languages** such as C++. These languages are geared more for human understanding and thus make the task of programming much easier. However, since the computer only understands low level binary code (often called machine code), there must be a translation process to convert these high level languages to machine code. This is often done by a **compiler**, which is a software package that translates high level languages into machine code. Without it we could not run our programs. The figure below illustrates the role of the compiler.



The compiler translates source code into object code. The type of code is often reflected in the extension name of the file where it is located.

Example: We will write source (high level language) code in C++ and all our file names will end with `.cpp`, such as:

```
firstprogram.cpp   secondprogram.cpp
```

When those programs are compiled, a new file (object file) will be created that ends with `.obj`, such as:

```
firstprogram.obj   secondprogram.obj
```

The compiler also catches grammatical errors called **syntax errors** in the source code. Just like English, all computer languages have their own set of grammar rules that have to be obeyed. If we turned in a paper with a proper name (like John) not capitalized, we would be called to task by our teacher, and probably made to correct the mistake. The compiler does the same thing. If we have something that violates the grammatical rules of the language, the compiler will give us error messages. These have to be corrected and a grammar error free program must be submitted to the compiler before it translates the source code into machine language. In C++, for example, instructions end with a semicolon. The following would indicate a syntax error:

```
cout << "Hi there" << endl
```

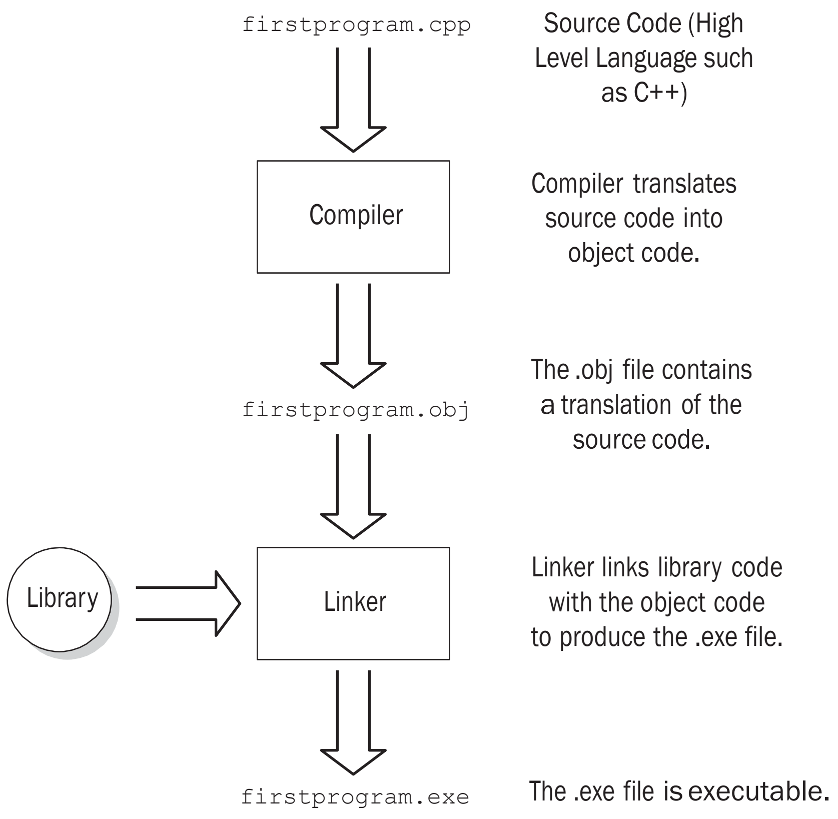
Since there is no semicolon at the end, the compiler would indicate an error, which must be corrected as follows:

```
cout << "Hi there" << endl;
```

After the compile process is completed, the computer must do one more thing before we have a copy of the machine code that is ready to be executed. Most programs are not entirely complete in and of themselves. They need other modules previously written that perform certain operations such as data input and output. Our programs need these attachments in order to run. This is the function of the **linking process**. Suppose you are writing a term paper on whales and would like a few library articles attached to your report. You would go to the library, get a copy of the articles (assuming it would be legal to do so), and attach them to your paper before turning it in. The **linker** does this to your program. It goes to a "software library" of programs and attaches the appropriate code to your program. This produces what is called the executable code, generated in a file that often ends with `.exe`.

Example: `firstprogram.exe   secondprogram.exe`

The following figure summarizes the translation process:


firstprogram.cpp
Source Code (High Level Language such as C++)
Compiler
Compiler translates source code into object code.
firstprogram.obj
The .obj file contains a translation of the source code.
Library
Linker
Linker links library code with the object code to produce the .exe file.
firstprogram.exe
The .exe file is executable.

Once we have the executable code, the program is ready to be run. Hopefully it will run correctly and everything will be fine; however that is not always the case. During "run time", we may encounter a second kind of error called a **run time error**. This error occurs when we ask the computer to do something it cannot do. Look at the following sentence:

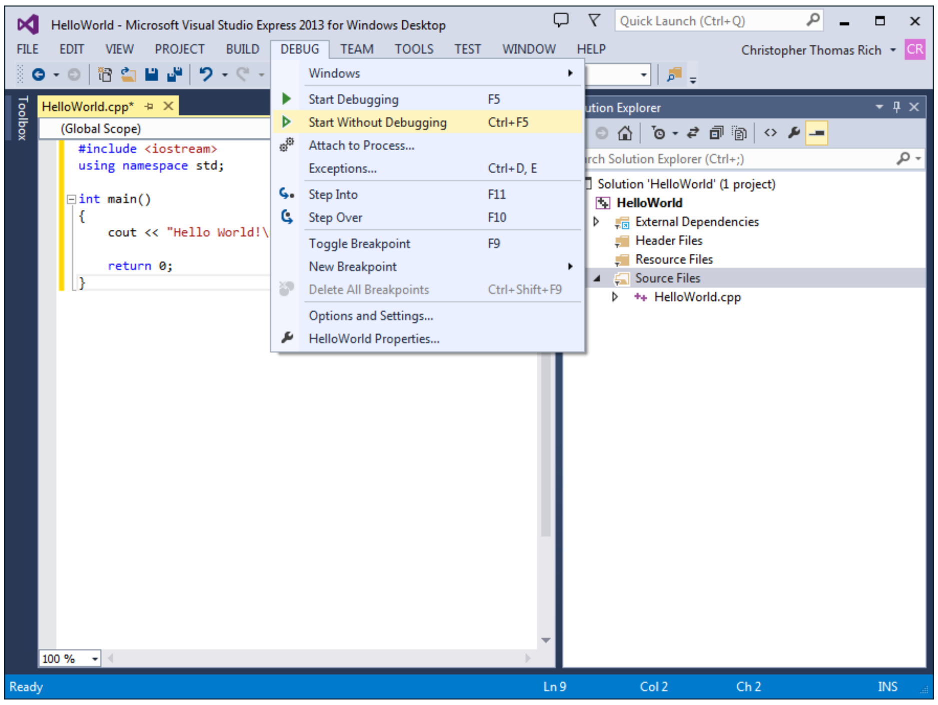
You are required to swim from Naples, Italy to New York in five minutes.

Although this statement is grammatically correct, it is asking someone to do the impossible. Just as we cannot break the laws of nature, the computer cannot violate the laws of mathematics and other binding restrictions. Asking the computer to divide by 0 is an example of a run time error. We get executable code; however, when the program tries to execute the command to divide by 0, the program will stop with a run time error. Run time errors, particularly in C++, are usually more challenging to find than syntax errors.

Once we run our program and get neither syntax nor run time errors, are we free to rejoice? Not exactly. Unfortunately, it is now that we may encounter the worst type of error: the dreaded **Logic error**. Whenever we ask the computer to do something, but mean for it to do something else, we have a logic error. Just as there needs to be a "meeting of the minds" between two people for meaningful communication to take place, there must be precise and clear instructions that generate our intentions to the computer. The computer only does what we ask it to do. It does not read our minds or our intentions! If we ask a group of people to cut down the tree when we really meant for them to trim the bush, we have a communication problem. They will do what we ask, but what we asked and what we wanted are two different things. The same is true for the computer. Asking it to multiply by 3 when we want something doubled is an example of a logic error. Logic errors are the most difficult to find and correct because there are no error messages to help us locate the problem. A great deal of programming time is spent on solving logic errors.

Integrated Environments

An integrated development environment (IDE) is a software package that bundles an editor (used to write programs), a compiler (that translates programs) and a run time component into one system. For example, the figure below shows a screen from the Microsoft Visual C++ integrated environment.



```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!\

    return 0;
}
```

Other systems may have these components separate which makes the process of running a program a little more difficult. You should also be aware of which Operating System you are using. An **Operating System** is the most important software on your computer. It is the “grand master” of programs that interfaces the computer with your requests. Your instructor will explain your particular system and C++ environment so that you will be able to develop, compile and run C++ programs on it.

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. Compilers detect ________________errors.
2. Usually the most difficult errors to correct are the ________________errors, since they are not detected in the compilation process.
3. Attaching other pre-written routines to your program is done by the ________________process.
4. ________________code is the machine code consisting of ones and zeroes that is read by the computer.
5. Dividing by zero is an example of a ________________error.