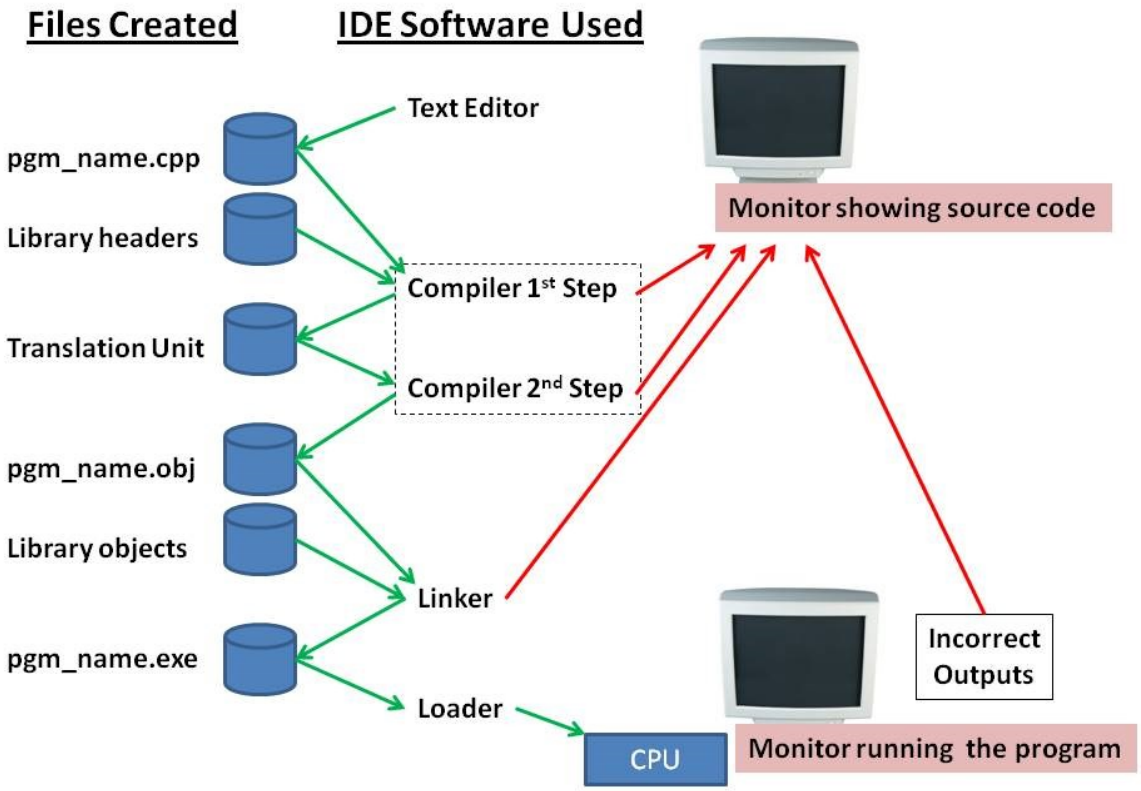
Compiler Design

Natural Language Processing (NLP)

High-level language programs are usually written (coded) as ASCII text into a source code file. A unique file extension (Examples: .asm .c .cpp .java .js .py) is used to identify it as a source code file. As you might guess for our examples – Assembly, "C", "C++", Java, JavaScript, and Python, however, they are just ASCII text files (other text files usually use the extension of .txt). The source code produced by the programmer must be converted to an executable machine code file specifically for the computer's CPU (usually an Intel or Intel-compatible CPU within today's world of computers). There are several steps in getting a program from its source code stage to running the program on your computer.

Historically, we had to use several software programs (a text editor, a compiler, a linker, and operating system commands) to make the conversion and run our program. However, today all those software programs with their associated tasks have been **integrated** into one program. However, this one program is really many software items that create an **environment** used by programmers to **develop** software. Thus the name: Integrated Development Environment or IDE.

Programs written in a high-level language are either directly executed by some kind of interpreter or converted into machine code by a compiler (and assembler and linker) for the CPU to execute. JavaScript, Perl, Python, and Ruby are examples of interpreted programming languages. C, C++, C#, Java, and Swift are examples of compiled programming languages.[2] The following figure shows the progression of activity in an IDE as a programmer enters the source code and then directs the IDE to compile and run the program.



Integrated Development Environment or IDE

Upon starting the IDE software the programmer usually indicates the file he or she wants to open for editing as source code. As they make changes they might either do a "save as" or "save". When they have finished entering the source code, they usually direct the IDE to "compile & run" the program. The IDE does the following steps:

1. If there are any unsaved changes to the source code file it has the **test editor** save the changes.
2. The **compiler** opens the source code file and does its **first step** which is executing the **pre-processor** compiler directives and other steps needed to get the file ready for the second step. The #include will insert header files into the code at this point. If it encounters an error, it stops the process and returns the user to the source code file within the text editor with an error message. If no problems encountered it saves the source code to a temporary file called a translation unit.
3. The **compiler** opens the translation unit file and does its **second step** which is **converting** the programming language code to machine instructions for the CPU, a data area, and a list of items to be resolved by the linker. Any problems encountered (usually a syntax or violation of the programming language rules) stops the process and returns the user to the source code file within the text editor with an error message. If

no problems encountered it saves the machine instructions, data area, and linker resolution list as an object file.

4. The **linker** opens the program object file and links it with the library object files as needed. Unless all linker items are resolved, the process stops and returns the user to the source code file within the text editor with an error message. If no problems encountered it saves the linked objects as an executable file.
5. The IDE directs the operating system's program called the **loader** to load the executable file into the computer's memory and have the Central Processing Unit (CPU) start processing the instructions. As the user interacts with the program, entering test data, he or she might discover that the outputs are not correct. These types of errors are called logic errors and would require the user to return to the source code to change the algorithm.

Resolving Errors

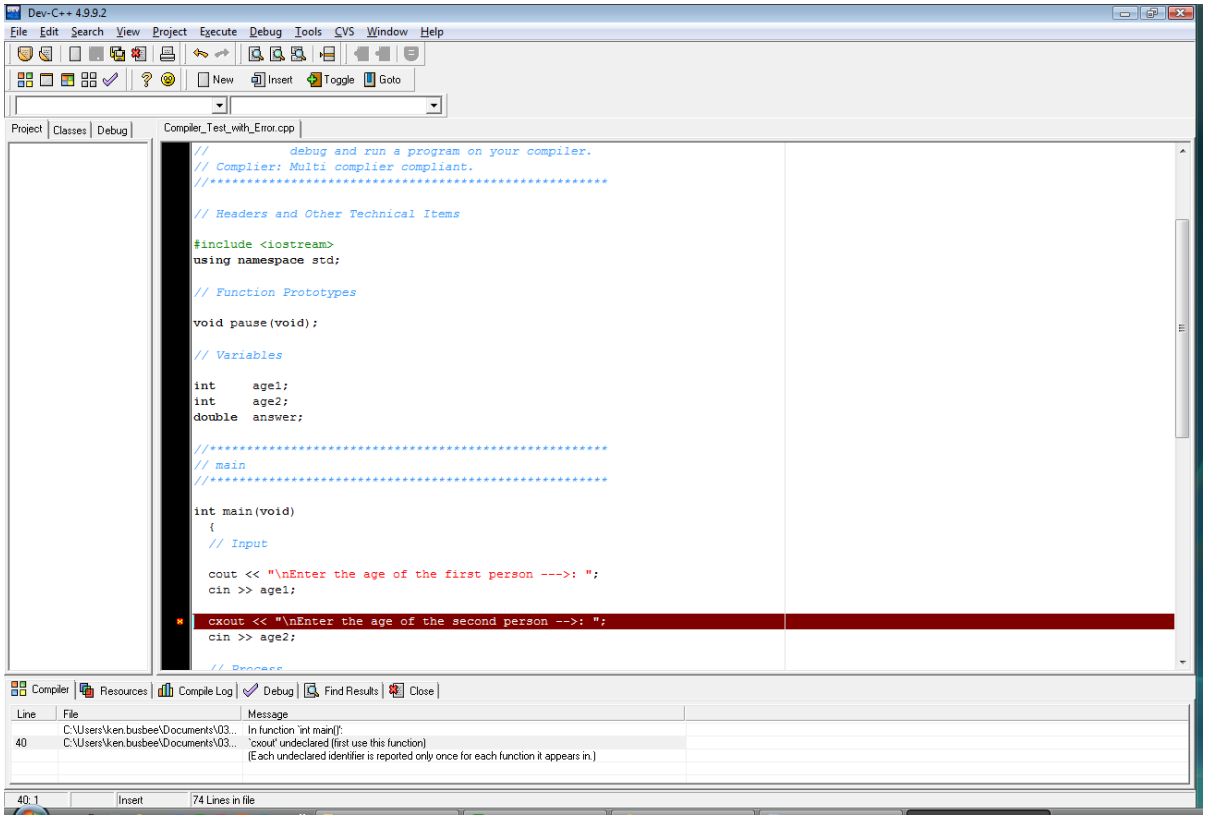
Despite our best efforts at becoming perfect programmers, we will create errors. Solving these errors is known as **debugging** your program. The three types of errors in the order that they occur are:

1. Compiler
2. Linker
3. Logic

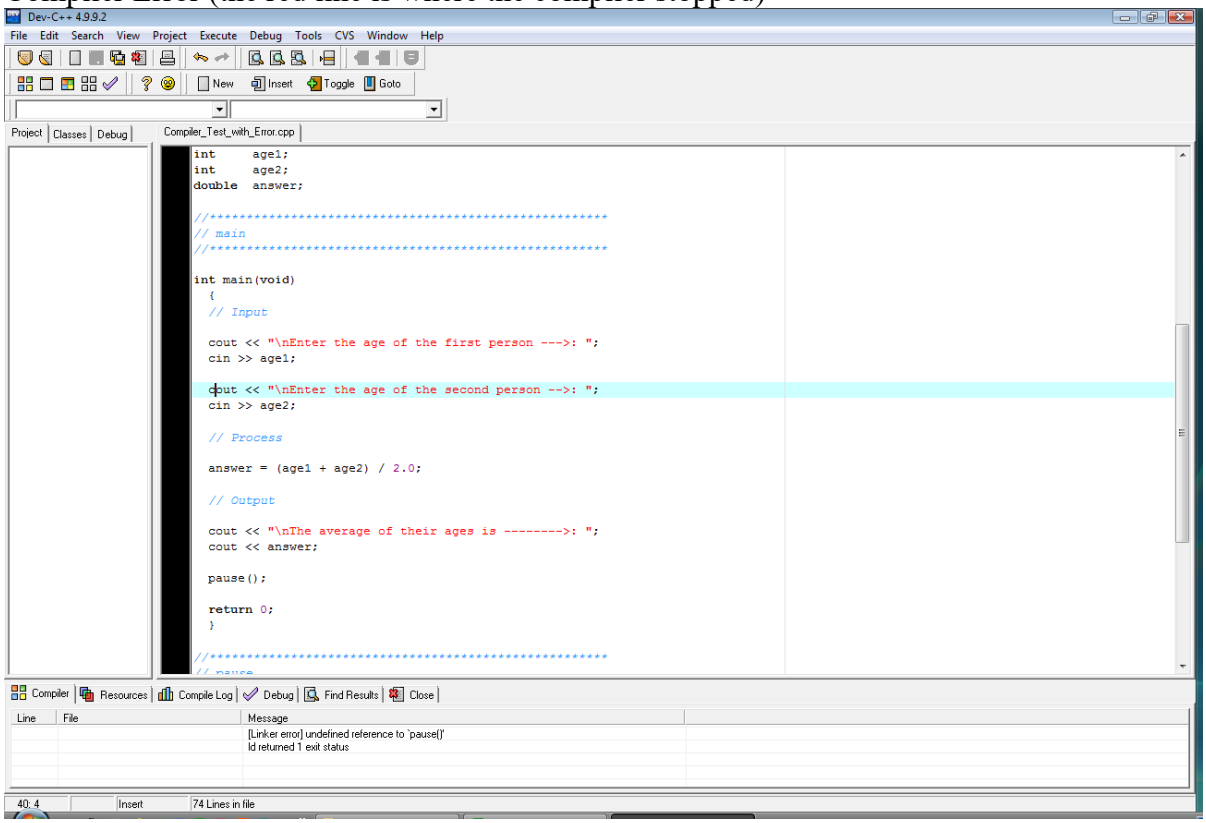
There are two types of compiler errors; pre-processor (1st step) and conversion (2nd step). A review of Figure 1 above shows the four arrows returning to the source code so that the programmer can correct the mistake.

During the conversion (2nd step) the compiler might give a **warning** message which in some cases may not be a problem to worry about. For example: Data type demotion may be exactly what you want your program to do, but most compilers give a warning message. Warnings don't stop the compiling process but as their name implies, they should be reviewed.

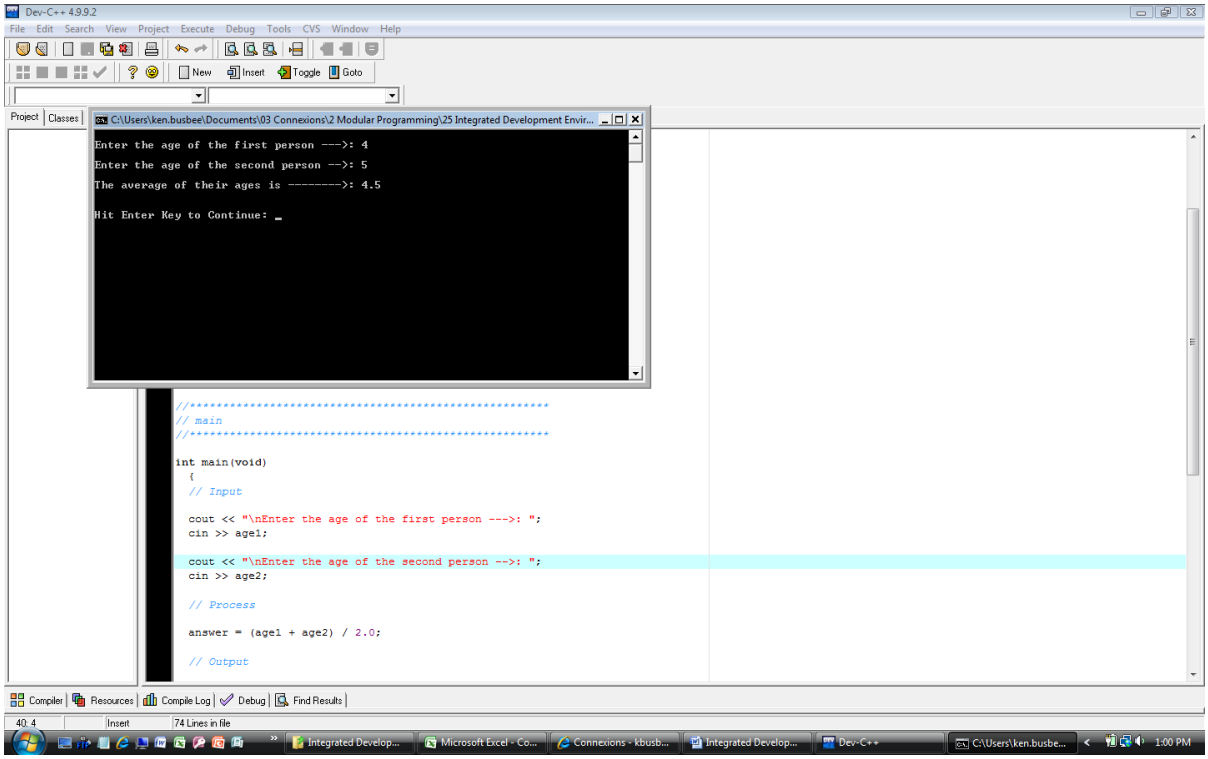
The next three figures show IDE monitor interaction for the **Bloodshed Dev-C++ 5 compiler/IDE**.



Compiler Error (the red line is where the compiler stopped)



Linker Error (no red line with an error message describing a linking problem)



Logic Error (from the output within the “Black Box” area)

Key Terms

compiler

Converts source code to object code.

debugging

The process of removing errors from a program. 1) compiler 2) linker 3) logic

linker

Connects or links object files into an executable file.

loader

Part of the operating system that loads executable files into memory and directs the CPU to start running the program.

pre-processor

The first step the compiler does in converting source code to object code.

text editor

A software program for creating and editing ASCII text files.

warning

A compiler alert that there might be a problem.

References

1. **https://press.rebus.community/programmingfundamentals/chapter/integrated-development-environment/**
2. cnx.org: Programming Fundamentals – A Modular Structured Approach using C++
3. Wikipedia: Integrated development environment
4. Wikipedia: Interpreter (computing)