# Tips for Reducing Problems with Your Programs

(or Identifying Problems if it's too Late !!)

There are a number of common problems you may face while trying to compile and run your programs. It often does not matter which environment you are working in (e.g. Windows or Linux).

We have put together this document to help you to avoid or, if it is too late, identify problems students often run into. Please note that this document is not intended to address every problem you may face, but it should help with the most common ones. Please also note that there is some information which may not apply to your programs (e.g. pointers are not discussed in CISP 301).

## Types of Programming Problems

Before we take a look at reducing or finding problems, lets look at the different types of problems you may have to deal with. A short description of each is given below along with at least one example.

| | |
|---|---|
| Syntax/Compiler Errors | An error which results when a rule of the language is not followed. Examples include forgetting a semicolon, missing one of a pair of quotes, and not having a matched pair of braces. When a compiler identifies an error of this type, it will display an error message and not create an executable file. |
| Syntax/Compiler Warnings | Sometimes a problem the compiler identifies is not severe enough to prevent the executable file from being generated. When this happens, the compiler generates a warning message. It is important to recognize that the executable created may not run successfully.<br><br>An example of a problem which would generate a warning message is: **`int number = 4.5;`.** A warning would be generated because a floating-point value can not be assigned to an integer. |
| Linker Error | Sometimes the name of a function will not be spelled correctly which results in an executable file not being created. For example, if a function is named **readdata**, but the function call uses **ReadData**, a linker error (LNK2019) would be generated (recall that C++ is case sensitive). |
| Run-Time Error | An error discovered at the time the program is running which causes execution to end immediately. Attempting to divide by zero is one problem which will cause a run-time error. |
| Logic Error | A program compiles and runs successfully, but its output is incorrect. For example, a program should add two numbers and instead it multiplies them. Errors of this type are not identified by the compiler because the code is following all the rules of the language. |

## Suggestions for Reducing Programming Problems

As you are writing your programs, there are choices you can make which could reduce the time it takes to get your program to compile and run successfully. The suggestions below are grouped together by topic.

### Variables

- Choose meaningful names for variables, functions, constants, etc.

  Suppose three variables are declared in a program and named **amount1**, **amount2**, and **amount3**. It could be hard to remember that **amount1** should store the price of an item, **amount2** the amount it is on sale for, and **amount3** the amount the customer has to pay including taxes. If forgotten, the variables could be used incorrectly (e.g. the total stored in **amount1**), and then the program has a logic error to find. Better names would be **price**, **discount**, and **total**.

  It is also not really a good idea to give variables the same name except for a difference in the case of one or more letters. For example, having two variables named **amount** and **Amount** could still be a source of confusion.

- Keeps names used within a function unique.

  Suppose both a variable and a function are named **sum**. If **sum** is called in the same function in which the variable is declared, there will be a problem compiling.

  Another version of this type of problem is naming a variable declared in a function with the same name as a parameter. Remember, parameters do not need to be declared inside a function.

  CISP 400 Note: It is also not a good idea to name a variable declared inside a member function the same name as a data member.

- Use variables for a single purpose.

  If a variable named **discount** was created to store the discount rate for an item (e.g. 20%), it should later not be used to store the amount the customer saves on an item being purchased (e.g. 25 for a 125 dollar item). It could be hard to find a problem (another logic error!) in this program because each time the variable is used there are two different types of values it be storing.

- Pause to think about initializing variables.

  When declaring a variable ask yourself the following question: Does this variable need to already have a value the first time it is used in the program? If the answer is yes, give it a value when it is declared.

  For example, if the first time a variable is used will be in a **cin** statement to store a value entered by the user, the variable does not need to be initialized. If, however, the first time it is used one will be added to it (e.g. **`i++`**), it should be initialized.

Control Structures (e.g. loops and if statements)

- Comment closing braces.

  If your program has a lot of braces in the same function, it could get hard to identify which braces match up together. Putting comments on closing braces could help. For example:

| **Without Comments** | **With Comments** |
|---|---|
| `while(        )` | `while(        )` |
| `{` | `{` |
| `  if(        )` | `  if(        )` |
| `  {` | `  {` |
| `    for(    ;    ;    )` | `    for(    ;    ;    )` |
| `    {` | `    {` |
| | |
| `    }` | `    }  // end for` |
| `  }` | `  }  // end if` |
| `}` | `}  // end while` |

- Step through a test which uses **&&** and **||** on paper before running the program.

  It is easy to inadvertently use **&&** instead of **||**, or vice versa, when writing a test. For example, the following test would not work: **`number < 0 && number > 10`**. Before running the program, think through the test using values which should result in a true or false result. If you do not get the results you expect, think the test through again and also double-check your new version.

- Pause to think about using the assignment operator (=).

  A common problem is to use = instead of == in tests in **if** statements and loops. When using = in statements other than assignment statements, ask yourself if == should really be used instead.

Functions

- Stub Programming

  Suppose a program needs to be written which has five functions in addition to **main**. One approach to writing this program is to write all the code for the entire program at one time and then compile it. For most students this will result in many problems to identify and fix which could be anywhere in the program.

  A second approach, stub programming, involves writing the program one function at a time. Once a function is written, the program should be tested until the function seems error free. Then the next function can be written and tested. This process should continue until the program is completed. The benefit of this approach is that, at any point in time, there should be fewer problems to deal with and they should be located in the newly written function making them easier to find.

- When writing a function, ask the following questions: Does this function need any data from the calling function to do its job? Does this function have any data that the calling function needs?

  Two potential sources of problems when writing a function are making sure the function has all the data it needs to perform the task it was written to do and making sure it sends the data it needs to back to the calling function. Pausing to ask these two questions, and writing the function accordingly, could help prevent these problems.

- Only use pointers when passing data to functions if needed. (This does not apply to CISP 301.)

  Suppose a function named **work** needs to use the value of a variable named **number**. There are two ways the function call can be written (with the function's parameter written accordingly):

  | | | | |
  |---|---|---|---|
  | Function Calls: | `work(number);` | OR | `work(&number);` |
  | Function Header: | void work(int number) | OR | void work(int * number) |

  In the first case, **work** can use **number**'s value. In the second case **work** can use **number**'s value, but **work** can also change its value. If the function does not need to change **number**, the second way of writing the function call results in a potential new source of problems. Suppose, **number** was given an incorrect value somewhere in the program. If the pointer was not used, **work** could be immediately ruled out as a potential source of the problem.

  Note: Sometimes pointers are used for efficiency reasons (e.g. structs).

Additional Suggestions

- Make sure to pair /* with */.

  It is really easy to forget to end a comment or use /* at the end of a comment instead of */. Double check your comments to make sure both parts are included.

- Pause to think about division.

  Remember that dividing two integers can give you a different answer than dividing numbers with other data types. For example, 5/2 = 2, but 5.0/2 = 2.5. Make sure the division you get is the division you want.

- Pause to think about array subscript values.

  It is a programmer's responsibility to make sure the subscript values used for an array are valid. Make sure to pause and think about whether the subscripts being used make sense for the array.

- The last character in a C-style string has to be a null character (**\0**). (This does not apply to CISP 301.)

  ```
  char string[5]

  string [0] = 'c';
  string [1] = 'a';
  string [2] = 'r';
  ```

  In the example to the right, the intention is to have the variable **string** store the word **car**. A null character is not, however, assigned to **string[3**], so **string** contains letters **c**, **a**, and **r** not the word **car**. If the null character is missing, none of the string library functions (e.g. **strcat**), will work as expected.

  A null character is placed at the end of a string automatically in many instances. Some examples include: initialization with a string constant (e.g. **char word[10] = "house"**), when data is read using **cin** , and when **strcat** or **strcpy** are used.

## Suggestions for Identifying Problems

No matter how hard programmers try, there are usually some problems which make their way into a program. Suggestions for figuring out what is wrong are given below.

Finding Problems (In General)

- Look at the line above.

  If the compiler tells you there is something wrong with a particular line of code, and you can not find anything wrong with that line, look at the line above the one the compiler does not like.

- Fix the first compiler error.

  One syntax problem in your code can often lead to the compiler generating multiple error messages. If the compiler displays a multitude of error messages, consider fixing just the first error indicated, then recompile.

- Add print statements at key points in your program.

  Sometimes a program will have a run-time error and there will be no way to tell from the feedback on the screen how far the program got before it failed. So the first problem is to figure out which line of code is causing the program to fail.

  A common way to handle this situation is to add print statements at important points in a program. For example, suppose the program calls three functions, **read**, **calculate**, and **print**. Print statements could be added as shown.

```
main()
{
  read();
  cout << "Finished read" << endl;

  calculate();
  cout << "Finished calculate" << endl;

  print();
  cout << "Finished print" << endl;
}
```

  When the program runs, if only **Finished read** prints, then the program fails in **calculate**. If both **Finished read** and **Finished calculate** prints, then the problem is in **print**.

  Make sure there is an **endl** at the end of your **cout** statements (as shown in the example above). If newlines are placed at the beginning of the text, the text may not be displayed on the screen even though the error follows the print statement.

Specific Problems

- A number printed to the screen is very large.

  A common source of this problem is that a variable is being printed which does not have a value. Take a look at how the variable is used and where it needs to be given a value before it is printed. Be sure to consider whether the variable should be initialized.

- The code in a loop never executes.

  This is especially common with a **while** loop. Make sure the loop's test will be true the first time it is checked. Be sure to consider whether the variable used in the test needs to be initialized.

Linux Environment Problems (This does not apply to CISP 301.)

- Segmentation Faults

  Segmentation faults are due to an invalid use of memory or an attempt to use an address which is not valid. Common causes include attempting to use a pointer which does not yet point to anything and using an invalid array subscript. This is not intended to be a complete list.

  To solve this problem, first the offending line of code needs to be discovered. There is not usually enough feedback on the screen to identify where the program fails, so adding print statements to the program is usually the best way to go (see the print statements bullet above).

  Once you find the line of code causing the problem, check to see if the problem falls into one of the common causes listed.

- Unable to save on the Linux server

  Every student with an account on the Linux server has a limited amount of space to use. If you reach a point where you are unable to save anything to the server, you will need to delete some files to free up some space. If you type **`ls -l`** (lowercase L) at the Linux prompt, you will be shown, among other things, the size of the files in the current directory. Look for any unusually large files which can be deleted. Also, remove your old .o files and old copies of programs that don't run.

- Wiping out a source code file when trying to compile.

  If the name of a source code file is used after the **–o** option instead of the name of the executable, the source code file could be wiped out. For example, suppose a program named **work.cpp** has previously been compiled and then new changes are made to it. An executable file, named **work**, is already in the current directory at the time the new version of the program is compiled. If the following statement is used to compile the program, the **work.cpp** file will be deleted: **g++ -Wall -o work.cpp work**.

  Please keep in mind that files on the server are not backed up. Consider creating a subdirectory in which to store a copy of your files.

- Avoid using special characters in file names.

  Use only letters (preferably lowercase), numbers, underscore characters and periods in file names. If other characters are used, you may have problems using the file (e.g. compiling, renaming or copying).