Error Messages and Debugging in C++

Errors are a fact of life in computer programming. They are the greatest source of frustration when developing programs, but also provide some great challenges while trying to determining their cause. The *debugging* (error finding) process is rather daunting for the first-time programmer; this handout is designed to help you with the process.

Types of Error Messages

There are two or three types of error messages, depending upon how you count:

Compilation (or syntax) errors - these occur during the translation (compilation) process and indicate that a C++ statement is not written correctly/syntactically, is missing, or is in the wrong order. The compiler usually provides useful information, such as the line number where the error occurred, to help you find the error. Less severe errors are called warnings - they do not stop the compiler from processing the program, but most likely they indicate some sort of problem that should not be ignored.

Execution errors - these errors occur when the computer tries to execute (perform the steps specified by) a program that has compiled correctly. They are caused by illegal operations (such as trying to divide by zero), and are more difficult to track down than compilation errors. The line number where the error occurred is NOT listed, so the programmer must utilize the "clues" provided by the program before it stopped executing to determine the cause and location of the error. Techniques and programming aids will be covered later in the class to help track down such errors.

Logic errors - The program compiles and executes without problems; however, the program does not compute the correct answers. In this case, the computer does not detect any problems and therefore does not produce any error messages (there is no error as far as the computer is concerned); the burden is placed entirely on the programmer to determine what is wrong. The best way to handle logic errors is to avoid them in the first place (easier said than done!).

Since compilation errors are the most common (although fortunately not the hardest to solve!), most of the information below applies to this type of error.

Anatomy of an Error Message

On *UNIX*, the `g++` compiler produces compilation error messages that look similar to:

```
testerr.c: In function ‘int main ()’:
testerr.c:10: ‘k’ undeclared (first use this function)
```

The message includes **(1)** the file, and the function within that file, where the error was detected, **(2)** the line number where the error was detected, **(3)** the error explanation, In the above example, an error occurred while compiling the function `main` in the file `testerr.c`, it was detected at line 10, and it was caused because the variable `k` is undeclared.

Other systems and compilers produce messages with similar information, although the format of the error message may be different.

Some Guidelines For Dealing with Errors

- **ATTITUDE is everything!** The correct attitude is: You WILL make errors, so don't be surprised or frustrated when it happens! Debugging is just part of the programming process, so accept the challenge!

- **Don't be alarmed at the number of errors you get** – it usually has nothing to do with the quality of your program or your ability as a programmer (although it may have something to do with your typing accuracy!). Sometimes, a single error made at the beginning of your program will produce several more errors further down in your program. **Start by trying to eliminate the first few errors and the obvious ones, then recompile** – many of the subsequent errors will disappear also!

- Often an error that is *made* in one line of a program will not be *detected* until later. Thus **the line number that is listed in the error message is NOT always the place where the error actually was made**. Also look at the lines immediately above the error line, and at lines involving the variables used in the error line.

- **Sometimes the compiler will try to interpret an offending line in a different way than you intended**, and thus the error message makes no sense to you. This especially occurs when you accidently use *keywords* (like `void` or `int`) for variable names, or when parentheses and brackets ({, }) are misplaced or unmatched. If you don't understand a message, look for these things. (A list of keywords is included in Appendix B of your book.)

- **Don't recompile without changing anything**, assuming that the "computer made a mistake." This simply does not happen! There is a definite reason why the compiler detected an error - it is your challenge to find out why.

- Similarly, **if you get an error that you can't understand and can't fix, don't just start *"changing stuff"*** – this technique usually doesn't help, wastes your time, and usually leads you further from the correct answer than a more systematic approach. If you get stuck on an error, ask someone (instructor, someone else in the lab, or a fellow student). Sometimes, just another "set of eyes" can be very helpful in finding errors.

- Remember that **debugging skills are developed with practice** – YOU WILL GET BETTER!! Good luck, and avoid getting frustrated.

Common Error Messages that Occur in Beginning Programs

`g++: test.c: No such file or directory`

You misspelled the name of the file you wanted to compile, or left off the `.c` extension when you typed the file in. The name of your file must be of the form *filename*`.c` and the compile command must be `g++` *filename*`.c` (with capitals and small letters exactly as shown!)

`parse error before '...'`

This is the general error message used when the compiler finds some syntactic problem with your program. Typical causes are missing or misplaced punctuation (commas, parens, brackets, etc.), improper use of operators (like `+`, `-`, `*`, `&&`, etc.), forgetting to end the last statement with a `;`, and misspelled keywords or variable names. The error was detected before the character shown, so look in front of the character. **NOTE:** The actual error may actually be on a previous line!!!

'*var*'`undeclared`

The variable name listed was not *declared* (with an `int` or `float` statement), or was misspelled either here or in the declaration.

`redeclaration of 'int` *var*`'`

The variable *var* has been *declared* more than once (in an `int` or `float` statement). Sometimes this happens if you try to use the same variable name for two different purposes.

`"cin" undeclared`
`"cout" undeclared`

These errors occur when you forget (or misspell) the `#include <iostream.h>` statement at the beginning of your program. Note that there is **NO** space between the `#` and `include`!

`no match for 'istream & << int &'`
`no match for 'istream & >> ostream &(&) (ostream &)'`
`no match for '_IO_ostream_withassign & >> int &'`

These are cryptic ones! These errors occur when the wrong input/output operator is used in a `cin` or `cout` statement (Remember: use >> for input (`cin`'s), << for output (`cout`'s)).

`missing terminating " character`

This error occurs when the closing quotes on a string are left off, as in:
`cout << "This string is not closed << endl;`

`non-lvalue in assignment`

The left hand side (called the *lvalue* in compiler lingo) of an assignment statement must be a single variable (as in `a = b + 4`), not a constant (as in `3 = a`) or an expression (as in `a + 2 = b`). While in algebra the equals sign means *"is equal to,"* in C++ it means *"is replaced by,"* and the order cannot be interchanged.

`parse error at end of input`
`parse error before '}'`

These errors usually occur because braces are not matched up somewhere. While the error will usually specify the last line of the file, the actual omitted brace is probably somewhere else in your program. These are often difficult errors to find!

NOTE: `vi` can help you locate what it thinks is the matching paren/bracket/brace: In command mode, put the cursor on one brace and type %. The cursor will then move to what it thinks is the matching brace. If the indicated brace is not supposed to be the matching one, or the cursor doesn't move, indicating no matching brace, then something is wrong!

Common Error Messages with Branching/Looping Statements and Program Blocks

`parse error before 'else'`

This is the message you get when the `else` clause of an `if` statement is misplaced. This may be caused by either not enclosing multiple statements to be controlled by the `if` statement in braces, or by having extraneous statements between the `if` and `else` clauses.

`case label 'x' not within a switch statement`
`default label not within a switch statement`

These messages occur if you forget to place braces around all the `case`s in a `switch-case` statement, or the braces are mismatched in some way.

`switch quantity not an integer`

In a `switch case` statement, this error occurs when the switch value (in parentheses after the `switch` keyword) is a float. A `switch-case` statement can only be used with *integral* types (i.e., `int`'s, `char`'s, and `enum` types).

Common Error Messages when Using Functions

"*func*" `undeclared`

This error occurs if no prototype is defined for the function *func* before it is used.

`undefined reference to '`*func*`'`

This error occurs if a function that is called from a program is not defined anywhere, i.e., the code for the function does not exist. It may also occur when the function prototype and the function definition don't match.

`warning: no return statement in function returning non-void`

A function was declared to return some type (besides `void`), but no `return` statement to return that type was found. Note that this is a warning, which means that the compiler will continue processing; however, the message also indicates a likely problem.

`too few arguments to function '`*func()*`'`
`too many arguments to function '`*func()*`'`

There is a mismatch between the number of arguments in the function prototype, function call, and/or function definition. There must be a one-to-one correspondence between arguments, in number and in type, in all three places.

`warning: return to 'int' from 'float'`
`warning: argument to 'int' from 'float'`

The function was declared to return one type, but the value in the return statement actually is returning another type. While these are just warnings, they probably indicate a problem.

Common Error Messages when Using Arrays

`variable-size type declared outside of any function`

The size of an array must be either a number or a value specified in a `const` statement.

`invalid types 'int[int]' for array subscript`

This cryptic message usually is caused when trying to use a variable as an array without declaring it as an array first, i.e., `a[3] = 5` where `a` has not been declared as an array. It also occurs if a non-integral value for a subscript of an array is used.

`Segmentation violation - core dumped`
`Memory fault - core dumped`

These messages almost always mean that an array subscript is out of bounds (for example, accessing `a[11]` of an array declared as `int a[10]`). Also remember, the largest subscript of an array is one less than the number of elements declared in the array, i.e., the statement `int a[10]` specifies an array with elements (`a[0], a[1], ..., a[9]`). These errors are difficult to find, because the statement that caused the error isn't identified. An illegal pointer reference (i.e., a pointer that doesn't point to anything) can also cause this problem.

Some "Non-Error" Errors - Also called "C-Gotchas"

A couple of "errors," that do not show up as errors, can cause serious problems (logic errors) in a program:

```
if(a <= b);
while(a < b);
for(i = 0; i < n; i++);
```

Note the *extra* semicolon at the end of these statements!! This semicolon causes each of the statements to be terminated, so the subsequent statement(s) are NOT being controlled as desired. In the case of the `if` and `for`, the statement does nothing; for the `while`, an infinite loop may result! Be careful NOT to type the extra semicolon!

```
 if(a > 4)
      r = 4 + b;
      s = a + 3;
      t = b + a;
 w = c + 10;
```

The indentation suggests that this code will define the values `r`, `s`, and `t` on the condition that `a` is greater than 4. However, notice that there are no braces around these statements. Therefore, the only statement controlled by the `if` statement is the first one; `s` and `t` will get defined unconditionally! Remember that the compiler ignores indentation; statements are grouped with braces.

```
if(a = 4) ...
```

While it appears that this statement determines if `a` equals 4, in fact the single equals sign indicates ASSIGNMENT, NOT EQUALITY! The proper way to check if `a` equals 4 is: `if(a == 4) ...`

```
if(0 < a < 10) ...          // Is a between 0 and 10?
if(a == 4 || 5) ...         // If a equals either 4 or 5 ...
```

Both of these statements contain malformed logical expressions, which, while being legal syntax (i.e., no error messages) are not going to do what is expected. All the logical operators (except ! (NOT)) are *binary* operators, i.e., they require an operand on each side. Thus, the proper way to write the above statements is:

```
if(a > 0 && a < 10) ...
if(a == 4 || a == 5) ...
```