A CATALOG OF COMMON BUGS IN C++ PROGRAMMING

Venu Dasigi
Department of Computer Science
Southern Polytechnic State University
1100 S. Marietta Parkway
Marietta, GA 30060-2896
(770) 528-5559
vdasigi@spsu.edu
http://lovelace.spsu.edu/vdasigi/

Abstract

In this paper, we briefly discuss the pedagogical issues surrounding the teaching of techniques to diagnose and correct programming errors. Then we catalog several common bugs students grapple with during the course of their programming projects. We found it very valuable to document them so students can help themselves, as well as be helped by the instructor.

1. INTRODUCTION

This paper is an attempt to fill what the author has observed to be a frequently overlooked void in introductory and advanced programming courses. Such course include, for example, a first undergraduate programming course or a graduate transition course, or even any other course with a substantial programming component, before which students have not mastered simple error diagnostic techniques. By including a formal or informal treatment of material similar to what is contained in this paper in the first few programming courses, it is hoped that much frustration will be avoided on the part of both students and instructors at later stages in a computer science program.

In this paper, we use the term "debugging" to refer to detection and correction techniques for various types of program errors (or "bugs") by inspection or by reasoning about the programs. We specifically *exclude* the "debugger tool" sense of the word; thus, we are *not* talking here about teaching how to use a debugger. The author has often seen faculty members tell students to develop their own debugging skills. "There is no way I am going to teach debugging in my classes; I have more important things to do," is a common refrain. This position is sometimes the result of a real or perceived obligation to teach a wide variety of concepts in programming courses, which does not seem to allow the time for a treatment of debugging. However, it is the author's belief that debugging is an important skill for students to master. It is hoped that this collection of common bugs that can "creep" into C++ program development will be useful to students, as well as faculty. If there is not enough time to devote to such material in the classroom, faculty can make this or an augmented list available to students, and spend some time outside the classroom helping students apply the techniques. Alternatively, it could be incorporated into a lab section if the course has an associated lab. A number of common C++ bugs are covered in an excellent, yet relatively little-known, book [Spuler, 94]. The reader might find some of the bugs we point out, as observed in code written by our students, to be similar to the bugs listed in that book, but we believe our treatment is original and represents classroom experience.

In the following section, we identify a classification of common bugs that show up in programs written by students. Recognizing a bug as relevant to the compiler, the linker, or the logic of the program can be very helpful in understanding it. In Section 3, we list several bugs, relating each of them to one of the classes identified in Section 2. We explain each bug, and identify a fix if appropriate. Section 4 concludes the paper.

2. CLASSES OF BUGS

We use the word "bug" in the broad sense of any programmer error that can result in a failure during compilation, linking, execution, or any other phase that would prevent the program from eventually running successfully. Thus debugging refers to understanding "bugs" and either getting rid of them or simply avoiding them before they are likely to show up. A broad understanding of the different steps in getting the program to run successfully and the roles of the different components used in the process is helpful in debugging.

2.1. Compiler Errors and the Role of the Compiler

As a general philosophy, students should learn to appreciate that the compiler is their "friend," in spite of the fact that it frequently catches errors in their programs. Some of the compiler errors are the result of not having mastered some trivial syntactic details (e.g., Should it be a comma or a semicolon?), but other compiler errors are indicators of more serious conceptual or logic errors. A frequent *compiler error message* results from confusion between different, perhaps closely related, types, e.g., a pointer to an object and the object itself, or a node type and the type of the main / key field in the node. It should be impressed on the students that such mix-ups are akin to "mixing apples and oranges," and compilers are fussy about them because such mix-ups are surface manifestations of deeper problems. Extending this notion further, students should also be encouraged to pay attention to *compiler warning messages*, which are generally indications of significant oversights or other logic errors (e.g., uninitialized variables). If a compiler error is fatal, a compiler warning is to be viewed as likely poison!

In C++, the role of the compiler is to make sure the program is syntactically correct, and then generate the object code. Thus, every function call is matched against a prototype, but *for the purpose of the compiler, a complete* ***definition*** *of the function is*

***not** required*, although a definition is required for linking and successful execution. In this context, we make an important distinction between *object code* and *executable code*. The former is machine code, which may or may not be complete and executable. The latter is complete machine code with all necessary definitions, including system and user libraries, ready to be loaded and executed.

2.2. Linker Errors and the Role of the Linker/Loader

Once the program files are compiled individually, the linker links them together along with any system libraries so all externals are resolved, and loads the resulting executable. Externals might refer to system libraries and in the case of a multi-file program, externals might refer to symbols and functions defined in a different file. A file or a collection of files that can generate an executable is generally referred to as a *project* in many development environments. Sometimes errors can arise from functions that are "promised" through prototype *declarations* (which satisfy the compiler), but are not "delivered" through *definitions* in any of the files being linked together in the project. Some of these errors can be particularly subtle in C++. For the purpose of this paper, we assume that a multi-file project has at least one *header file* (e.g., `class.h` that declares a class, containing mostly the prototypes of member functions, but not all definitions), one *class library definition file* (e.g., `class.cpp` that defines all member functions), and one *driver / application file* (e.g., `classprj.cpp` that makes use of the class); we use these terms consistently and repeatedly in this paper. Generally speaking, linker errors arise from failed attempts at reconciling cross-references.

2.3. Errors related to Language Design Philosophy

This category of errors relates to a misunderstanding of the backward compatibility of C++ with C, and various problems related to pointers and dynamic memory management. Dynamic memory management issues fall into this category. Most programming languages either leave the complete responsibility of avoiding memory leaks and dangling references to the programmer (e.g., Pascal, C, and Ada) or perform some variation of automatic garbage collection (e.g., LISP, Java, and Ada). C++, however, while following the first method, attempts to lessen the burden on the programmer. It lets the programmer define code for the destructor in a class once, and automatically invokes the destructor when the lifetime of any object of that class ends. While, on the surface, this philosophy appears to help the programmer, in reality, it puts additional burden on the programmer when dynamically allocated structures are potentially shared between objects. However, once the issues are understood, solutions can be easily implemented.

2.4. Logic Errors

The logic errors that can be found in code written by C++ students can be of a wide variety, but we will try to focus on problems that are particularly frequent or those that are specific to C++. Examples include uninitialized pointers, certain types of memory reference errors, etc. The category of logic errors, while an important one, receives relatively little attention in this paper, because most good books do a good job of identifying them, e.g., [Main and Savitch, 97]. Another recent paper by the author is also relevant in this context [Dasigi, 96].

2.5. Meta Errors

These are the errors that can result from or during the process of fixing other errors. Some of them remain dormant until other errors are fixed: They have always been there, but just happen not to manifest themselves since they are somehow overshadowed. Other meta errors get introduced innocuously during the process of detecting or fixing some existing errors.

2.6. Other Errors

Some of these errors are related to the way multi-file projects need to be organized. Some relate to more than one kind of the errors mentioned previously, e.g., different issues related to templates. Still others relate to the choice of the type of project created in the different development environments.

3. KNOWING THE BUGS AND DEBUGGING THEM

In this section, we discuss several common bugs belonging to the above categories. Each error is identified by its category, indicated in parentheses by the corresponding subsection number from the preceding section (e.g., 2.1, 2.2, etc.)

3.1. "Innocently Undeclared" Functions (2.1)

Depending on the exact chronological sequence of events during the development of code, a "member function" of a class sometimes gets defined in the class definition file, without having been declared in the header file. At least two compiler errors arise out of this oversight. (i) A *use* of the function anywhere, especially in the driver file, gives rise to the "undeclared function" error; and (ii) the definition of the function in the class library definition file causes an error message to the effect that the function is not a member of the class in question. Introducing a prototype of the function into the class declaration in the header file should fix both problems.

3.2. Missing Default Constructor (2.1)

A missing default constructor by itself is not a major problem except that its members could go uninitialized. A potential, and rather insidious, error awaits the programmer who forgets (or simply does not define) the default constructor, but defines *at least one other* constructor. If no constructor has been defined, a default constructor is automatically generated. However, according to the C++ reference manual, "A default constructor will be generated for a class X only if no constructor has been declared for class X " [Ellis and Stroustrup, 90]. This would become a real problem, and results in a compiler error message if an array of objects of the class type in question were declared anywhere, because array components are constructed by generating calls to the default constructor. Since the default constructor is called for plain object declarations (that is, object declarations with no arguments), such declarations result in a compiler error, as well. The way to fix this error would be to define a default constructor; even one with an empty body would do.

3.3. Incomplete Forward References (2.1)

Consider the following scenario:

```
class Node {
     …
     friend class Stack;
     …
     };

class Stack {
     …
     Node * top;
     …
     };
```

The `Stack` class uses the `Node` class, but for efficiency, the `Node` class needs to grant friendship to the `Stack` class. Forgetting the keyword `class` in the `friend` declaration causes an "undeclared identifier" error, and is a common problem.

3.4. Mismatched Signatures between Member Declaration and Definition (2.2)

This is a particularly common error pattern. The problem arises, for example, when the header file contains a constant member function (also known as an observer function), with the `const` qualifier, and the definition in the library definition file mistakenly omits the qualifier. The effect is that when the linker tries to resolve a reference in the application program to the constant member function, it does not find the definition! This is a rather subtle error to find, since the error message does not come from the compiler. The compiler matches the declaration in the header to the call in the application program, but at the time of linking, the linker cannot find a definition that matches the original signature! This kind of situation calls for a very watchful eye, and once the source of the error is localized, it can be fixed by changing the definition or the declaration appropriately. Other variations of this kind of error include many other types of mismatched signatures *that cannot be distinguished just by an inspection of the call*.

3.5. Function Declared in the Header, but not Defined Later (2.2)

This is a slightly more serious version of the error just discussed. If the preceding error is a result of a promise unintentionally broken, this one is a case of a promise simply not kept! All calls to the declared function pass the compiler, but with no compiled definition to link the calls to, the linker gives an error message.

A particularly egregious version of this error occurs when a constructor or a destructor that was declared in the header file is not defined in the library definition file. The error messages resulting from this kind of situation are particularly hard to understand because the constructor and the destructor are generally not explicitly called. Also, the problem would not arise if, for instance, the destructor, the default constructor, or the copy constructor is simply not declared in the header in the first place, since in these cases, the compiler automatically supplies them!

3.6. Mutators Invoked in an Observer Context (2.2)

Methods that do not modify the object to which they are passed as messages are called observers (`const` member functions in C++), whereas those that do modify the target object are called mutators (non-`const` member functions). Since non-`const` member functions are intended to modify the target object, it is an error to call them on an object that is expected to remain constant (e.g., one that has been passed to a function by `const` reference). It is also an error to call a non-`const` member function without an argument within a `const` member function, since that would indicate a conflict of purpose. Fortunately, both kinds of errors are caught by most compilers. The errors can be avoided by avoiding such calls, which is always possible with a good design.

3.7. Variations of the "Shared Structure Problem" (2.3)

When a class contains at least one pointer data member (even if it is intended for a dynamically created array), potential exists for structure sharing. An object pointed at by the pointer data member can become shared when the object is copied into another (through a copy constructor) or is assigned to another (through the assignment operator). Structure sharing happens if a shallow copy (that is a bit-wise or member-wise copy) is made, which is what the compiler-supplied default implementations do for both the copy constructor and the assignment operator.

Structure sharing in itself is not a problem if managed well, e.g., by keeping track of the number of references in the shared structure itself. However, if not managed well, when the lifetime of one of the objects sharing a common structure ends, a destructor is automatically called by the compiler, and the dynamically created shared structure ends up getting deallocated. The result, of course, is a dangling reference in the other objects that had been sharing the deallocated structure! An attempt to access the deallocated structure through the dangling reference could raise an exception, which could lead to a program crash.

There is generally a friction between approaches that attempt to minimize generation of garbage and those that attempt to avoid dangling references. Since dangling references are fatal (while garbage generation may be thought of as "slow poison"), a simple solution to the above problem is to deliberately not define[1] the destructor, thereby allowing memory to "leak". From a pedagogical perspective, this approach has the merit of deferring discussion of dynamic storage management to a later point in the course. A better solution is to include at least some rudimentary elements of dynamic storage management. The solutions of deep copy semantics for or reference counting in the copy constructor and assignment operator may be introduced as deemed appropriate [Pohl, 97].

This problem can take on a slightly different flavor in the context of inheritance. At issue are the C++ conventions that the *copy constructors* and *assignment operators* of derived classes do not automatically call their base class versions, although the base class versions are automatically called for *default constructors* and *destructors*! Thus, *all* derived classes of a class with a non-default copy constructor / assignment operator pair should define their own versions of copy constructor and assignment operator. If the derived class has no additional pointer data members, all that the implementations of the derived class versions need to do is to invoke the base class versions of the functions. If the derived class has additional pointer data members, the derived class implementations should, *in addition,* also perform deep copying or reference counting.

3.8. Declaring a Pointer Does NOT Create an Object! (2.4)

This is a rather commonly misunderstood idea about pointers when a student is first exposed to pointers. While the misunderstanding is easy to correct, being aware of how common it is helps an instructor watch out for this common error pattern. The problem manifests when a student declares a pointer and soon starts dereferencing it!

3.9. Laziness can sometimes be good! (2.4)

[1] Of course, if this "solution" is chosen, care must be also taken to not *declare* the destructor (See Sections 3.4 and 3.5).

Lazy evaluation of Boolean expressions, which is a standard feature in C++, can be very helpful in avoiding certain kinds of memory reference errors. Walking off the end of an array or a linked list can be avoided by exploiting lazy evaluation. Consider,

```
while ((i < ARRAYSIZE) && (A[i] != target)) i++;
```

or, in the context of a linked list,

```
while ((p != NULL) && (p->data != target)) p=p->link;
```

In both cases, memory reference problems can be avoided, but students need understand the role of short-circuit evaluation in avoiding the problems. Without an adequate understanding, the student might switch the order of the conjuncts as the author's students sometimes did, which could cause runtime errors.

3.10. Error in the Use of "Debug Print Statements" (2.5)

It is the author's belief that as a "debugging aid," reasoning about program logic is much superior to symbolic debuggers. The author *has* used symbolic debuggers as a *teaching tool* (e.g., to show the program stack during recursive calls, or even to watch how values of different variables change as one steps through a program). However, it is the author's (admittedly arguable) belief that debuggers distract students from a deeper understanding of the program. It has been observed that if students are encouraged to reason about their program (e.g., by directly simulating debugger tools, e.g., trace, watch, step, etc.), they eventually detect the bug(s), and in the process, come away with a deeper appreciation of the program logic.

Occasionally, they resort to using the so-called "debug print statements." While use of these print statements amounts to setting up watches, students need to plan their placement carefully. For this purpose, the author frequently teaches students to use a compile time DEBUG flag that can be used to turn the debug print statements on or off.

The most common error the author observed in this context, also documented in [Spuler, 94], is forgetting to flush the print buffer[2] in the debug print statements. In such a case, if the program were to crash, it may well go past the debug print statement without actually sending the debug output into the output display (only because the buffer had not been flushed)! In such a situation, the student would erroneously believe that the crash happened *before* the debug print statement was reached!

3.11. The Need for Frequent Recompilation (2.5)

[2] This can be done using the `endl` manipulator, but some implementations of C++ flush the output buffer on the use of an end of line character `\n`.

The students would be best advised to recompile their programs frequently during the process of fixing compiler errors, especially after fixing important errors. This is because oftentimes multiple error messages are generated by different interpretations by the compiler of the same basic error. Therefore, fixing an error can suddenly get rid of several other error messages, saving time that would have been spent understanding all of them. This technique is particularly valuable if an error message turns out to be hard to understand, since it could go away after one of the related, more obvious, errors is fixed. Occasionally, however, fixing an error message that might have caused the compiler to abort further processing could result in several new error messages on recompilation. It should be explained to the student that the new error messages are not *caused* by recompilation, but by problems that are in the original code itself.

3.12. Problems with the "#include" Directive (2.6)

C++ supports nested inclusion through the "`#include`" directive. However, there are times when unintentionally the same file gets included multiple times. This can happen, for instance, when `file1.h` is included in `file2.`h, and both of them are included in `file3`. In this situation, the contents of `file1.h` are processed twice in sequence, and could lead to duplicate definition errors! The standard "`#ifndef` technique" involves surrounding the normal contents of each header file with compiler directives and addresses this issue as follows:

In `file1.h`:

```
#ifndef FILE1_H
#define FILE1_H
…
#endif
```

Here `FILE1_H` is a unique compiler constant used to keep track of whether the contents of `file1.h` have yet been processed in the compiler environment. The first time the inclusion is processed, the compiler constant `FILE1_H` would not have been known so far, so the contents do get processed. In doing so, the compiler defines `FILE1_H` and thus, prevents any further inclusions of the same file.

3.13. Defining and Using Function Templates (2.6)

In its most common form, a function template specifies a generic type parameter name that is used to define objects in its parameter list and / or body. It is a conceptual error to instantiate the function template in the context of an object of a type for which some of the operations used in the body of the template are undefined. This kind of error is generally caught by the compiler.

Another error that can arise with templates relates to the organization of a project into different files. A header file generally contains only function declarations, not complete function definitions. In this context, we might think of a function declaration as not resulting in machine code, whereas a function definition does. In this sense, a templated function "definition" is really only a declaration, since actual machine code cannot be generated until the template is actually instantiated! Thus, a variety of errors can result from not placing templates into header files and incorrectly placing them into library definition files.

Finally, students occasionally template the `main` function! Since there is no place to instantiate such a template, when the files in the project are about to be linked and loaded, an error results, indicating that the `main` function is missing! The fix, of course, is not to template the `main` function.

3.14. Creating the Right Kind of Project (2.6)

In most introductory courses, as long as GUIs or windows applications are not being created, a console application is a simple and adequate choice for the type of project to be created. This is a choice most C++ environments offer, and choosing any other kind of project results in a more complex environment than the student is ready for. This point needs to be emphasized early on, or else the students would not be able to continue the development of the project without understanding the creation of different resources and many built-in class libraries.

4. CONCLUSION

We believe we have outlined some of the most frequent errors that plague students in their early programming courses. Many good books discuss common syntax and logic errors, but a systematic treatment of common errors and how to understand and prevent them would be very valuable to students, and is likely to reduce the length of the lines at the instructor's office.

ACKNOWLEDGEMENTS

The author appreciates the comments and suggestions from one of the anonymous reviewers, and his colleagues Briana Morrison and Saun Shewanown.

REFERENCES

[Dasigi, 96] Dasigi, Venu: "C++ > C + OOP (A Personal View of Teaching Introductory Programming using C/C++)," *Journal of Small College Computing*, pp. 42-150, 1996.

[Ellis and Stroustrup, 90] Ellis, Margaret, and Stroustrup, Bjarne, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Main and Savitch, 97] Main, Michael, and Savitch, Walter: *Data Structures and Other Objects using C++*, Addison-Wesley, 1997.

[Pohl, 97] Pohl, Ira, *Object-Oriented Programming using C++*, Addison-Wesley, 1997.

[Spuler, 99] Spuler, David: *C++ and C Debugging, Testing, and Reliability - The prevention, detection, and correction of program errors,* Prentice-Hall, Sydney, 1994.