Designing Programming Languages for Reliability

Harry Porter
Computer Science Department
Portland State University

October 16, 2001

Abstract

This paper contains several comments and thoughts on designing programming languages so that programs tend to be more reliable. It is organized as a list of suggestions for anyone who is about to design a new language.

Introduction

The Ariane-5 was a reusable space vehicle designed and manufactured in Europe. On its maiden flight, the vehicle was destroyed in the launch phase, due to a software error. An unhandled floating-point overflow was the problem. The software was written in Ada, the language which, several years ago, the U.S. Defense Department decreed was to used for all military systems.

Software errors have killed innocent people. In an early infamous example, a computer controlled the X-ray dosage in a medical machine used to treat cancer patients. A programming error caused the machine to give lethal doses during radiation treatment, instead of the much smaller doses actually prescribed. Unfortunately, the lethality of the doses was not immediately detected and several patients were affected.

On a smaller scale, home computers are crashing all the time. Perhaps you too have been annoyed by an unexpected, unexplained failure of software that you felt ought to function correctly.

Software reliability is clearly important and much research has been done on how to increase reliability. Nevertheless, software quality remains lower than most people find acceptable. What is the problem?

First, we must realize that reliability is costly and not every program requires the same level of reliability. The problem isn't that we can't produce programs with the desired level of reliability but that we don't put an appropriate emphasis on it. This is fundamentally an economic issue, not a technical issue.

Second, most programmers enjoy programming and the goal of making programming more enjoyable is often directly opposed to the approaches for making programs more reliable.

Third, some of the theoretical work on program correctness is too abstract. It may be too difficult for many programmers to understand and too costly for organizations to use. It may be overkill. For many applications, we want a small, incremental increase in reliability for a small, incremental increase in programmer effort.

My feeling is that the programming language itself has a huge impact on reliability. This document contains a number of thoughts on how we can design languages to encourage correct programming.

I'll show several examples in C and C++. I don't mean to pick on C and C++ in particular. After all, it was clearly stated in the design of C and C++ that tradeoffs were always made in the favor of efficiency and that the languages were inherently dangerous and difficult and meant only for experienced programmers, who were instructed "buyer beware." I'll use C and C++ since the problem of bugs is worse in those languages than other, more modern languages and since they are so widely known and used today.

Overall Principle: Readability

We must place more emphasis on making programs readable, versus making them easy to write. Generally, readable programs are longer and require more typing. Most programming languages tend to emphasize conciseness, but conciseness is not good for reliability.

Which is easier to read?

```
================
class C : D {
  f1: int;
  ...
}
================
class C
  superClass
    D
  fields
    f1: Integer;
  ...
endClass
================
```

Typing in the program is only part of the programming process. After the program is written, the programmer will spend time looking at it, both while new code is being written and while the program is being debugged.

It important that programs are easy to read, and that the language design emphasizes readability over writability. An early experience with COBOL gave "readability" a bad name. COBOL was designed in part to encourage readability over conciseness. COBOL had many problems and was not, in the end, a successful language. There was a spill-over effect, giving "readability" a bad name by association.

As programs get larger and more complex, readability becomes ever more important. A general theme in the suggestions made below is that the readability of the language is more important than how easy it is to write.

Overall Principle: Reliability v. Execution Speed

Much of the computer industry places a huge emphasis on execution efficiency. Without doubt, we face a tradeoff between reliability and speed. Programmers have a tendency to make irrational decisions in favor speed over reliability and this is one major cause of the software reliability problem. We must pull the false God of "execution efficiency" off its pedestal.

How many programmers will admit to spending time optimizing a program that will be run only a few times? I have spent hours tweaking a program to run (say) twice a fast, when the CPU time I save over the entire lifetime of the program will be measured in microseconds.

Many of the ideas I'll discuss below will make programs run slower, but who cares? I literally ask "who cares and why?" Some programs do need to run really fast, but it is important to make sure that speed is important enough to sacrifice correctness.

The old adage says, "Get the program working first, then worry about optimization only after execution speed proves to be a problem." This approach is full of wisdom. In addition, we might also say, "Use a programming language designed for reliability first, and rewrite the program in another language when it becomes a problem."

Most of the reliability features I discuss have a "constant-time" execution cost, which is often completely insignificant when we consider (1) today's processor speeds, (2) the speedups from using better algorithms, and (3) the costs of program failures.

Programming is a passionate activity and we can't hope to defeat the speed-lust mentality quickly. But we must recognize that the irrational emphasis placed on speed is directly opposed to reliability of computers. The tradeoff may be between an application that works slowly and an application that crashes occasionally.

One way to approach this tradeoff is to put as much of the execution burden onto the compiler and runtime environment as possible. This allows the programmer to make a single decision about the tradeoff between execution speed and reliability at the time he chooses a programming language to use. This decision can be a calm, rational, one-time decision, not a decision made in the irrational heat of a programming frenzy, where decisions effecting reliability are usually made.

Relationship Between Reliability and Portability

Programs should be portable from one machine to another or from one operating system to another. It is usually the subtle technicalities of a programming language's semantics that make porting programs difficult. It is also in these subtleties that bugs arise.

Portability and reliability are closely related. Designing a language to encourage correctness and reliability will also increase the portability of the programs, and vice versa.

Simpler Syntax

Program syntax is a major source of confusion and bugs. Programs should be easy to read and, to help that, the syntax of a programming language should be simple. We now have the compiler technology to automatically parse very complex language grammars, but what is important is that humans should be able to quickly grasp and understand programs. Complex syntax is a problem.

I feel that programming languages should have a grammar that is LL(1). In a nutshell, LL(1) means that you don't need to look ahead to know what sort of syntactic construct you are dealing

with. Grammars that are LL(1) are intrinsically easy to read since they can be parsed reading left to right, one token at a time.

```
// LL(1) example:
if ...
```

Here you know by reading one token exactly where you are. You know what will come next (a conditional expression).

C and C++ are not even close to LL(1), as the next example shows.

```
// Non-LL(1) example of C++
x * * * ...
```

What comes next? Here are some possibilities:

```
=========================
typedef ... x;
x * * * foo (...) {...}
=========================
int x;
int * * p;
x * * * p > 0 ? ... : ...
=========================
```

The point is that it is better to have a syntactically simple language. C and C++ are grammatically complex, and this has several negative consequences. (1) Beginning programmers have difficulty learning the language and equate learning the syntax with learning to program. (2) Compilers have difficulty producing good error messages. (3) Experienced programmers have to devote a little extra mental effort to parsing.

The last point is relevant to reliability. By "parsing," I mean the subconscious effort that a programmer makes when looking over his own code and typing in new code. A proficient programmer doesn't spend much time "parsing" and we don't really realize that we spend any, but we do. By making the syntax more complex than necessary, a programming language throws a million micro-distractions at the programmer. These take their toll. Anything we can do to make it easier to program will increase program reliability.

There are many ways to measure grammar complexity, but the easiest and simplest is to simply count tokens. By this measure, grammars that require more tokens are estimated to be more complex.

As an example, compare the same program fragment coded in "C" and in a language I designed, called "Portlandish."

```
=========================
while (1) {
  ...
  if (p == NULL) break;
  ...
}
=========================
loop
  ...
```

```
  until p == NULL;
  ...
endLoop
=========================
```

The first example contains 14 tokens, while the second contains 7 tokens. Of course the second example will be slightly unfamiliar, but I hope you can appreciate the point about syntactic simplicity.

I also think that there is a small mental burden imposed by punctuation tokens in comparison to keyword tokens. As children, we learn to read words quickly and easily. Punctuation is smaller, harder to see.

```
=========================
if ((p != NULL) && (p->next != NULL)) {
  ...
} else {
  ...
}
=========================
if (p != NULL) and (p->next != NULL) then
  ...
else
  ...
endIf
=========================
```

No Name Hiding

Consider a C++ program with a global variable called "count" and a class containing a field with the same name. Suppose that in some section of code there is a reference to "count." Which variable is meant? We have developed the ideas of scope, visibility, and name hiding to answer this question. As another example, consider a function called "foo" and a method in a class, also called "foo." In a line of code, we see "foo" invoked; which "foo" is intended?

I suggest that name hiding is a bad idea, since it may lead to confusion. Even if the programmer does not make any mistakes regarding name visibility, he will be a little distracted. The microsecond spent thinking "Isn't this foo the method...? Yeah," may be a microsecond where he would have thought "What if we are at EOF... will this still work?" Perhaps that was the one and only moment when that critical possibility would have been thought of. Instead, he thought "Which foo was it?" and never noticed a bug related to EOF.

As a general rule, every language feature that confuses beginning programmers has a negative impact on program reliability. Such features require more thought than simpler alternatives. For the beginning programmer they are simply a burden; for the experienced programmer, they constitute thousands of micro-distractions that go unnoticed, but continually take away mental resources that would otherwise be spent asking questions like, "What if i is zero here?" or "Am I really sure that this will work at EOF?"

Getting rid of name hiding means that the programmer will have to think of more names to reduce the ambiguity. I feel that allowing local variables in a function or method to hide other entities is okay, but that global variables, function names, class names, class fields, class methods, and so on, should all have distinct names.

Fewer Base Types

C and C++ have several integer data types, including "short int," "int," "long int," and "char." There are also several floating-point types, including "float," and "double." Actually, these types may or may not be distinct. This is an implementation dependency and C/C++ takes a very complex position about the differences between these types.

This is ridiculous. Certainly we need a character type, an integer type, and a floating-point type, but there are very few programs that need more. For the floating-point type, "double" is sufficient for most applications. I think it is better to take a larger format, which will slow programs down a little, but which will result in greater accuracy for all programs. Some programs fail due to accuracy and are fixed by changing the data type from (say) "float" to "double." Just start with a large data type and forget about the smaller, (error-prone but efficient) types altogether.

As for the integer data type, most all computers are geared for efficient manipulation of 32-bit integers, so just leave out 16-bit, and 64-bit data types. Also, signed arithmetic is sufficient for almost all programs, so leave out unsigned integers. The only reason for unsigned data types seems to be to allow access to the unsigned operations on the processor, but we should remember that the processor architecture should be designed to support high-level language features. Unsigned operations only survived in successive generations of processor architectures because C allowed them to be used.

Eliminate Machine Dependencies

C and C++ have been designed to allow efficient implementations on a wide variety of machines, but this has been achieved at the cost of making the language definitions "machine dependent." One example is the number of bits used for the "int" data type.

The result is that it is possible to make efficient implementations of C/C++ on different machines. But it is also that case that the same program may run differently on different machines. It may even run differently on different compilers on the same machine. This is unacceptable since it has reliability and portability consequences.

It is important that the language take a stand on the base data types. For example, the language should say that "Integer" means signed, 32-bit arithmetic. If this means that the compiler implementation on some machines is difficult, it also means that programs written in the language will always port to those machines, with no problems.

The language definition should be absolute and leave no flexibility to the compiler writer.

Reduce Similarity of Feautres

In C/C++ there is a well-understood problem concerning the confusion between = and ==.

```
================================================
if (i = 0) ...      // What was typed
if (i == 0) ...     // What was intended
================================================
```

One solution is to use a different symbol for assignment

```
================================================
i = 0;              // C, C++, Java
```

```
i := 0;             // Many other languages
=============================================
```

Another solution is to have a more rigorous typing, requiring conditional expressions to have type "Boolean."

The general rule in language design is that different features should look different to reduce the similarity of features. This is a good principle, but the real work comes in the many decisions about which symbols are selected for which operators and which names are selected for which functions.

Here are some other similar features whose confusion is dangerous.

```
=============================================
&&          &
||          |
!           ~
==          =
getc()      getchar()
printf()    fprintf()
malloc()    calloc()
=============================================
```

Perhaps the best we can say is that more emphasis must be placed on selecting good names for things.

Another principle is that instead of providing several similar functions, the language should just provide one. It should provide a single, all inclusive function, rather than several small functions with similar functionality.

As an example, we should keep "calloc" and get rid of "malloc." Calloc is a little harder to use (more arguments and slower execution), but the benefit of throwing "malloc" out is that there will be no confusion from using "malloc" and expecting "calloc" functionality (a nasty little bug you may have experienced at some time).

Eliminate "Undefined" Behavior

In the name of greater efficiency, C/C++ does not initialize local variables. They get their values from whatever happens to have been in memory. Occasionally, this results in bugs that are really hard to track down, since changing the program even a little often changes the runtime behavior and failure mode. And sometimes the program will not fail even though there is a bug.

The cost of initializing variables to zero, or having the compiler force the programmer to initialize all variables explicitly is relatively small. If we are to increase program reliability in practice, we must be willing to pay a performance cost from time to time.

More generally, the language should not permit any "undefined" behavior. The language documentation should make it clear how each and every program will behave. For starters, it means disallowing the program to pick up random values from memory. Ideally, all pointers will be typed and it should be difficult or impossible to accidentally pick up a random value from memory.

All operations should be unambiguously defined for all input combinations. As an example, the C/C++ operations for right-shift is incompletely specified (Is it logical or arithmetic?); the

behavior is "machine dependent." Another example is the modulo operator %, which handles negative arguments differently on different machines. These operations are both fundamentally important operations, yet exceedingly dangerous and difficult to use correctly!

Greater Runtime Checking

An example runtime check is to make sure that array references are within bounds (i.e., "array overflow"). While there are some theoretical approaches to avoid such runtime checking, as a practical matter, the compiler will need to insert a runtime check every place an array element is read or written. There will be a small execution speed penalty associated with this checking.

For correct programs, additional runtime checking is pure overhead. The real value comes from programs with errors. With runtime checking, the error will always be caught. Without the checking, the program may fetch a random memory value or write to an address outside of the array and the program may, if lucky, continue to function and produce the correct behavior.

The benefit of the runtime checking is that it speeds debugging. If the array limits are exceeded, it will be caught the first time and attention will be immediately focussed on the offending instruction. This makes debugging faster and easier.

Any technique that makes debugging easier will improve program reliability for two reasons. First, it may make the difference between finding a bug and not finding it. Perhaps an array overflows once during testing, but with no ill consequences. With checking, the bug will be flagged. Without checking, the bug may not get noticed.

Second, techniques that make debugging easier will free up resources and programmer time for additional debugging. Making it quicker to find the array indexing problem may allow a few additional tests to be run, catching other (non-array) bugs.

Another important runtime check is testing pointers for NULL. C/C++ specifies that no object may be stored at the NULL address, so the program is guaranteed to crash if a NULL pointer is dereferenced. But a specific runtime check will immediately flag the offending instruction with a message like, "Attempt to dereference a NULL pointer on line xxx." Such a message is more helpful than something like "Segmentation fault, core dumped," since the core dump itself has to be analyzed just to determine what went wrong.

Simpler Primitive Operations

The next recommendation is to limit the basic, built-in operations of the language to simpler operations. One example in C would be "printf", which is a very complex function. Of course, "printf" is not technically a part of C, but a library function, but effectively it is a part of the language.

The idea is to replace one complex operation with a number of simpler operations and require the programmer to put the pieces together. For example, "printf" would be replaced with a number of operations to convert various integers and floats into strings, operations to combine strings, and operations to write strings to the output.

The benefit would be that complex interactions among the components of a complex operation (like "printf") would be eliminated. Since doing a single "printf" would now require several instructions, these could be debugged sequentially and errors could be isolated and reported more accurately.

Another example of complex operations are the assignment operators, like +=, *=, <<=, and so on. The programmer can easily substitute other operations for them:

```
x += 1;          // Dangerous
x = x + 1;       // Better
```

The second instruction will be understood by all C/C++ programmers, while the first statement will not be understood by a few programmers. Some programmers simply don't use the assignment operators and their use is always a little extra mental effort. Fewer keystrokes does not always mean simpler. Also, beginning programmers may not yet be familiar with the += operator. And who needs %= or <<= after all?

Another issue is the post- and pre-increment operators. My feeling and opinion is that "increment" is so widely used as to be indispensable, but that the distinction between pre- and post-increment is dangerously confusing. I might suggest eliminating pre-increment altogether and possibly eliminating post-increment, to leave just a stand alone increment operator.

```
a[++i] = 100;    // Eliminate pre-increment
a[i++] = 100;    // Possibly eliminate this too???
i++;             // Keep this
```

The same comments apply to decrement.

Less Implicit Computation

All computation should be explicit: the compiler should not insert complex code sequences without anything in the source flagging the insertion.

One example is the implicit casting inserted by the C/C++ compilers.

```
int i;
double d;
...
i = d;         // implicit cast inserted here
```

It is a little effort to show the cast explicitly, but it flags the operation being performed. Casting can be done using the function call syntax:

```
i = double_to_int (d);
```

In this example, there are two things that can go wrong. First, the number may be rounded, and this rounding could be unexpected or wrong. Second, the value may exceed the capacity of an integer and overflow could occur, resulting in a value that is not even close.

These two conditions can be discussed clearly in the documentation associated with the "double_to_int" function and the programmer will know exactly where to look. It is harder to find documentation for implicitly inserted computation.

Greater Compiler Analysis

Over the years, researchers have developed a number of compiler algorithms to analyze source programs and these algorithms should be used to the fullest.

One example is that the compiler can check for variables that are declared but not used. A variable that is never used clutters up the program and may cause confusion with other variables. It is probably the result of some code that was changed or eliminated.

```
    int myVariable = 1;
    ...
    int myVar;
    ...
    x = myVar;           // Error: myVar not initialized
correctly
```

Another thing the compiler can check for is variables that are used before being given initial values. I feel that the language should be specific about variable initialization (either forcing the declaration to have an initial value or explicitly stating that the variables are initialized to zero), but if this is not the case (as in C/C++), the compiler ought to look for cases where the variable is used before being assigned. A completely accurate analysis is not always possible for every program. The compiler can determine whether a variable is (1) definitely set before being used, (2) definitely used before being set, or (3) unable to determine since it is dependent on execution order. In the last case which fairly rare, the programmer should be forced to throw in an extra initialization just to be sure.

Compilers can also check that routines have correct return statements. If a function must return a value, the compiler should verify that every path through the function ends with a return statement with a value. The C/C++ compiler will let the programmer leave out a return statement and the program will quietly pick up a random value when the function returns.

Fuller Specification and Emphasis on Error Handling

Many operations have error conditions or have some special behavior for extreme values. For example, dividing by zero is an error that can arise for the divide operation. Most arithmetic operators (like + and *) can result in overflow if the operands are too large.

The language should place a greater emphasis on what happens in these error conditions.

First, every operation should be fully specified. What happens when zero-divide or arithmetic overflow occurs? It must be clear from the definition of the language. When the language says something like, "if such-and-such condition holds, then the result is undefined or implementation dependent," problems in reliability and portability inevitably occur.

Perhaps the operation is handled one way on machine A and this results in correct behavior of the program, while on machine B, the response to the error causes incorrect behavior.

Second, the language documentation and programming culture should place a greater emphasis on the handling of errors. Whenever we learn about a new function, we are first concerned with what it does under normal circumstances. What must change is the attitude that we can stop learning about the function before we have discovered its full, complete specification. Programmers must develop the habit of not using operations and functions without thinking about how they behave in bizarre, limiting cases. Documentation should never discuss an operation without discussing its error handling.

Third, the designers of functions and languages should think very carefully about how their functions behave when errors arise. For example, when the program tries to divide by zero, the language designer might choose to say either (1) "The result is zero" or (2) "The error will be caught at runtime and the program will abort." Which behavior is the better design? In the case

of zero-divide, I would suggest that aborting the program is the better behavior, but my point is that the designer must think long and hard to make sure that the response to bizarre cases is the most rational and logical.

Often programmers don't think clearly about the error cases, so exactly how the language behaves becomes critical. I would argue that aborting the program on zero-divide is the better approach since it may arise from a programmer failing to consider the zero-divide case and therefore may indicate a program bug. It is better to catch and report bugs that to keep on going with no indication.

Capability for Handling Errors

Errors will arise and every language needs a mechanism for handling them. The throw-catch mechanism, which I first encountered in a LISP system, seems like a necessary feature for any language designed for reliability.

The idea is that extreme, bizarre cases, which are expected to arise from time to time in the normal execution flow are best handled out-of-line. The throw-catch mechanism takes care of this naturally. For example, the programmer can deal with a "zero-divide" error, if it arises, in a way that makes sense to the application without cluttering up the main body of code.

A second application of error handling mechanisms such as throw-catch involves a different paradigm of use. The idea is that the error is not expected to occur at all, but if it does in spite of all precautions, the program must still be able to complete its task.

As an example of the second paradigm of error handling, consider a program that performs a division operation. Assume the programmer has analyzed the program and feels certain that zero-divide can never occur. However the application is so critical that it cannot be allowed to fail. Perhaps the program is an on-board avionics program whose failure would result in loss of life.

In such an approach, a backup algorithm designed solely for redundancy will be provided. Often a separate programming team will code and debug the exact same programming task completely independently. If a "zero-divide" bug manages to creep into the primary program, in conjunction with a failure in the analysis of why zero-divide can't possibly occur in the primary program and with a failure in debugging to catch the error, then if the "zero-divide" condition ever arises during the execution of the program, the backup program would be invoked to take over the task using the separately-coded algorithm. Hopefully, the backup code could be used to fly the plane.

Graceful Degradation

Biological systems often fail gradually, not suddenly, slowly degrading over time.

The benefits of gradual degradation are first, that the system can keep on going rather than stopping suddenly, and second, that a program that is gradually failing can be watched and monitored as it fails. This may allow a backup system to be invoked when the performance falls below a critical threshold. It may also allow debugging to commence while the problem is occurring.

Two places where graceful degradation can be applied are (1) too many memory allocations, and (2) too many threads being created.

As a program runs, it allocates memory. At some point, the memory resources are exhausted and the program must fail. What I propose is that the system forces the program to begin to fail

before the point of hard failure is reached. For example, the runtime system might begin to impose small waits when available memory falls below some threshold. If the program continues to ask for more memory, the waits might be increased and warnings might be issued. The idea is that the user of the program will notice the problem before the program actually fails, giving him time to take appropriate action.

Sometimes programs will create several threads-or-control (or processes) as they execute. Occasionally, a program with a bug will create lots of threads and bring the entire system down. Typically, with lots of threads, the CPU becomes evermore scarce, until all processing grinds to a halt. It is better if the scheduling algorithm has a priority scheme where threads that are creating threads are themselves slowed down more and more, causing them to fail, rather than causing the entire system to fail.

Emphasis on Style and Programming Conventions

The language documentation tells you what "white space" is, but not how to indent, space, and format your program. Yet formatting and making your programs neat and attractive is a large part of making them readable. When I look at students' programs, there is an obvious correlation between neatness and correctness. Sloppy programs have more bugs.

I feel that more emphasis should be placed on making programs neat and readable. In my own programming of C/C++, I have adopted a set of personal stylistic conventions. These conventions make it easier to program since I don't have to make a myriad of small micro-decisions.

Consider the following lines of code, which differ only by spacing.

```
i = foo (x, y, z);
i=foo(x,y,z);
```

This is not the place to argue which is better, although I prefer the first, since (1) it tends to spread things out a little, and (2) it follows the spacing conventions of natural language.

The point is that while typing this line in, five micro-decisions about spacing had to be made. By adopting a set of stylistic standards, this mental effort will be freed up for more important thoughts, perhaps affecting correctness of the code.

Once the program is typed and is being read or reviewed, it becomes much easier to read when a single style pervades. The eyes can scan the code faster if it is formatted and indented the same way consistently.

Other conventions may help to eliminate bugs in a different way. One rule I use is to never use assignment = any place except at the statement level, to prevent confusion with ==.

```
if (i = 0) ...      // Avoid this use of =
```

Whenever my eye happens to fall on a statement like the above, it stands out as an error without even thinking about the semantics. Here, the point is that certain stylistic conventions can be used to avoid constructs that tend to be buggy or to write them in a way that reduces the chances of bug.

Another example is the failure to properly bracket "if" statements. Suppose we start with an "if" statement with a single statement in the "then" statement group.

```
z = foo (max);
if (x < y)
  max = y;
z = bar (max);
```

Now suppose we want to add a single statement:

```
z = foo (max);
if (x < y)
  max = y;
  maxChanged = 1;     // This stmt added to the THEN stmts
z = bar (max);
```

The intent was that the "maxChanged" statement should be included in the "then" statement group, but was the programmer forgot to add braces, as C/C++ requires. It was even indented properly, causing it to appear to be correct.

```
z = foo (max);
if (x < y) {
  max = y;
  maxChanged = 1;      // This was intended
}
z = bar (max);
```

One programming convention I follow is to always use braces, even though there is only one statement. This convention reduces this sort of error since (1) the braces would most likely have already been there before the addition, so we would have to make two errors to cause a bug: forgetting them in the first place, and not noticing when we made the addition; and (2) once braces are used everywhere, their absence begins to stand out and when indenting fails to match the braces, it becomes more noticeable.

This is an example where the use of a programming convention may reduce errors. The style and programming conventions should be taught in parallel with the language itself. I feel that each language should have a "recommended style" and this is something that should be designed in parallel with the language.

Debugging Support

Better support for debugging reduces program bugs, since it makes it easier and faster to catch bugs. The easier and faster it is, the more bugs will be caught. Each program will be subjected to a given amount of debugging effort. I would estimate that I will typically write code for (say) 15 minutes and then spend about the same amount of time debugging it. A more critical software project might allocate 3 months of verification for a piece of code that was written over two weeks. In any case, the point is that there is a finite amount of resources that will be spent debugging so any increase in the productivity of debugging will have direct consequences on reliability.

Here are some debugging features that are most useful:

Ability to "single-step" the program. The programmer can stop the program during execution and can look at data values. Occasionally a bug will be localized to a few lines, but still be unapparent. It is sometimes useful to be able to execute each instruction in turn to see exactly when and how the problem is manifested.

Debugger incorporated into the run-time environment. Bugs may happen at anytime, although they are more frequent immediately after the program is written. The debugger should be incorporated directly into the run-time system so that it is available and active for every run of the program. If the debugger is a separate tool, it is one more step in the process to invoke it, and this slows debugging down.

A more important justification occurs with especially complex programs, where the conditions bringing out the bug are complex and difficult to create. Sometimes it requires an enormous effort to make the bug repeatable so that traditional debugging can begin, only to find the bug immediately once you know what line number is involved. By making the debugger always active, whenever any runtime error occurs, the debugging can begin immediately.

One complaint might be that users should be isolated from the details of the source and run-time environment, but this is not a valid reason for leaving the debugger out of the shipped product. First, when a program fails, the user is simply not isolated from the implementation; you can insulate the user from debugging by not shipping programs with bugs. But practically, when bugs do occur to users, sometimes the user may be willing to attempt to fix the program or perhaps work with the programmer over the telephone to identify the state and "capture" the bug while it is manifest. Second, the issue of shipping or releasing source code is not related. It is easily possible to include the debugger with every runnable version of the program without also including the source code. Having the debugger loaded and running may mean that you get a clean error message and source line number, even though the source is not available.

Which source lines have been executed. The debugger should track which source lines have been executed and which have never been executed. The idea is that to be thorough, the programmer should have executed each and every line of code at least once. While this hardly constitutes complete or thorough testing, it is often a good target that is not always reached.

This support need not be completely thorough or accurate. For example, when I bring up the source code in the debugger, I would like to be able to flip a switch that would cause all source code that has never been executed to be highlighted. Even if it only highlights the lines that have not been executed since the most recent compile, that would be a tremendous help.

In lieu of this, I use the following technique. When coding, I will often write and type in several routines at once and then switch to debugging. At the beginning of each routine I place a print statement to indicate that the routine has not been debugged.

```
void foo (int x) {
  int a, b, c;
  printf ("*****  Untested  *****\n");
  ...
}
```

These lines serve two purposes. First, they serve as a reminder to me of which routines I have debugged. This is important since a nasty bug will occasionally distract me for a long time and I may have to leave and resume debugging much later. If I don't mark routines, I may fail to debug a routine that needs it. Second, when I am executing code, I will try to focus on a few routines, but execution may stray into other, untested routines. When this occurs, I like to know I am executing code I have not had a chance to test yet.

With complex, nested "if" statements, it is often important to keep track of which cases have been tested. In such situations, I will put print statements in each case to help me track which ones I have looked at.

Marking routines as "tested." The debugger should also provide support for tracking the debugging process at the routine level, as well as the statement level. The idea that each routine is either "tested" or "not yet tested" is crude but useful and it would be nice if the debugger had a simple flag associated with each routine in the program. Each flag would have two states. The programmer would run the program a few times until he convinced himself that the routine had been sufficiently exercised. Then he would flip the flag from "untested" to "tested."

A more complex debugger would provide more support. Each routine could be marked with information about its debugging history: (1) who performed the debugging, (2) date of debugging, (3) comments about the debugging, (4) level of debugging, from "lightly tested" to "thoroughly tested" to "exhaustively tested" to "proved correct." Perhaps the debugger would take into account re-compilations, or even differentiating between re-compilations that directly altered the source code of the routine in question.

Limited I/O Capabilities

Communication between a program and its environment is fraught with danger. I/O is a major source of portability problems, since different environments have different I/O characteristics. The tradeoff seems to be between (1) standardized, simple but limited I/O and (2) full-function but environment-dependent I/O.

Unix has been so successful in part because of the simple "stream of bytes" I/O model (i.e., "stdin" and "stdout"). A large number of programs use only this sort of I/O and these programs have continued to run over the years in a wide variety of environments. Programs once designed to be used by humans are invoked from other "shell" programs or strung together with pipes; they still run with no change.

As computer interfaces have become more complex, the Application Program Interfaces (API's) have become correspondingly complex. Today, applications are generally tailor-made for a single specific environment. An application for Unix will not run on Windows or the Mac.

So there are two problems here. One is portability and the other is the complexity of the I/O interface. I can't propose anything specific, except to say that the interface between a program and its environment is a major problem area. Such an interface should be designed to be simple and portable. If reliability is a concern, then the interface may need to be simplified. Perhaps we should remember that the good, old command-line interface was reliable and simpler to program for.

Support for Re-use

Of course one route to reliability is to use pre-existing code that has been thoroughly tested, instead of writing new code.

The problems of code re-use are: (1) it is difficult to locate code that does the desired task; (2) it is difficult to use code in a different context than it was intended for; (3) the code to be re-used may not do exactly what you want and modifying it reduces the usefulness of re-use; and (4) it is difficult to understand and trust code that you did not write yourself.

Re-use occurs at several levels. (1) Programmers re-use algorithms that were invented or discovered years ago. This re-use is common but requires a lot of coding and debugging effort. (2) Programmers re-use code that was written by someone else. An example is the use of library functions. (3) Programmers use code that they wrote themselves in another context. Often, you can cut-and-paste from code you wrote a while ago, modifying it to fit the new context. (4)

Companies re-use code by buying the source and also by hiring the programmers who bring knowledge about the code in their heads. (Perhaps this is "coder re-use" and not "code re-use.")

The thing which most encourages code re-use is good documentation. In order to re-use code (either source or algorithm), you must know about the original work. You must also understand how it works, understand its details, understand its limitations, and have confidence in its correctness. Probably the best form of documentation is when the code or algorithm is discussed in a textbook, and the textbook is widely read.

Re-use is particularly important for container classes (e.g., "sets," "lists," "tables," etc.) These common data structures and their associated algorithms are well studied, widely used, and needed in almost every program, so it makes sense that they should be made available for easy re-use in the programming language. Parameterized types and classes (also called "template classes" or "generic types") are critical to support typing in the presence of such re-use.

This is not the place to discuss the intricacies of parameterized types and classes, but good language design in this area is critical to support this common form of code re-use. As an example, look back to the Smalltalk language. In part, Smalltalk was so successful because there was a tremendous amount of code re-use, particularly with the container classes, but also with the classes related to the user interface.

Reduction of Ambiguity

The programming language should allow only one way to say things, if possible. In other words, ambiguity should be kept to a minimum. This is a difficult goal to achieve, but the benefit is that two pieces of code that do the same thing will tend to look the same.

One area where improvement can be made is in the rules for operator precedence and grouping in C and C++. These rules are fairly complex since they try to match the precedence rules from mathematics. While this may seem to increase program readability, I think that it can also have a negative impact, since it requires a little extra effort to recall and apply these rules.

Here is an example:

```
x = (MyType *) p -> q;
```

Does the cast apply to "p" or to "p->q"? I find myself putting in extra parentheses rather than checking the book to see exactly what the rule is. Perhaps you are familiar with the C/C++ rule here, but the point is still valid.

In other languages (like Lisp and Smalltalk) the operator precedence rules are very simple, but they do not match standard mathematical convention. When first learning the language, these rules cause a little confusion, but after becoming familiar with the rules, using these rules becomes second nature. After using both styles extensively, it is my belief that simple precedence rules are better. Parentheses get added only when necessary and understanding complex expressions becomes easier.

A larger point is that the syntax for C/C++ is highly ambiguous, but this is a separate issue.

Another issue is with function calls with more than one argument. The problem is that confusion as to which argument is which can lead to program bugs.

```
computeInterest (x, n, 8);
```

Sometimes, the typing system can catch mistakes, but when the arguments happen to have the same type, the compiler cannot catch the bug.

One successful innovation is the idea of "keyword" parameters, as in Smalltalk. The syntax is a little confusing if you are unfamiliar with Smalltalk, but its superiority quickly becomes apparent after you use it a while.

Consider the following Smalltalk expression, in which a single method is invoked. The full name of the method is "computeInterestOn:numberOfPeriods:withRate:". Granted the method name is a long and contains colons, but it quickly becomes clear which argument is which.

```
computeInterestOn: x numberOfPeriods: n withRate: 8
```

Adding keyword message syntax to C/C++ is a big jump because of its strangeness to some people, but the idea of somehow helping the programmer to keep straight the different arguments to a routine is probably good if it can be done without decreasing the overall readability of programs.

Support for Modularity

A large program consists of several pieces which are created at different times, possibly by different people. These pieces may be compiled and debugged separately. They may also be written, modified, and documented separately.

It is important that the pieces fit together correctly and it would be nice if the compiler provided support for this process. In particular, the compiler should keep track of which pieces depend on which pieces and make sure that full checking occurs across all the pieces. For example, if a change is made to one piece that requires recompilation of other pieces, then the compiler ought to force the compilation of whatever is required. It ought to be impossible to run the program without building it in the proper way.

In Unix, there is often a dependence on "makefiles" to perform the necessary recompilations. A small mistake in the makefile can easily go unnoticed. For example, failing to include a dependence link may result in the failure to recompile a program component when necessary. This can lead to a very subtle problem; the program may build correctly most of the time, but every once in a while, it may fail to re-compile a component when necessary. The result is that the executable will not match the source.

Going one step further, the compiler system ought to provide support for the debugging process. Each program component ought to be flagged as either "debugged" or not. This was discussed above.

Incorporation of Documentation into Software

Large programs typically include documentation in addition to code. Often this documentation is kept outside the program source, for example, as a "Word" document. It is crucial that documentation be kept in synch with the program, and when they get out of synch, this is a source of confusion and bugs. The documentation must be kept up-to-date; when the program is altered, all copies of the documentation must also be modified, and vice-versa.

There may be several versions of the source code and several versions of the documentation, making things even more complex and error-prone.

The simple solution is to link to documentation directly to the source. Creating a new version of one ought to create a new copy of the other. Each copy of the program should have a copy of the documentation and each copy of the documentation should refer to exactly one version of the program. Whenever one version of the source is changed, the corresponding version of the documentation should also be altered and updated. Encouraging this sort of "documentation maintenance" is important and can be facilitated if there is a strong link between each copy of the source and the corresponding copy of the documentation.

The simplest and most practical way to link source and documentation is to keep both of them in the same file. In other words, the "user-related" documentation should be included directly into the source code.

One approach is to enclose the user documentation in "program comments." To read the documentation, the user must read the source code. This is appropriate, reasonable, and efficient for some applications, but is impractical for others.

Another approach is to include a feature in the program that allows users to see the documentation when the program is running. (Perhaps the "-h" flag causes a print-out of the user documentation or perhaps clicking a "help" button pops up a window containing the documentation.) Including the documentation this way is convenient for users since it will be available when they are using the program, which is usually when they want it.

If the user documentation is included directly in the source file, the documentation never becomes separated from the source and it also becomes a little easier to modify the documentation. Whenever you are editing the source, you are also automatically editing the documentation. It is right there.

Whenever the source file is copied, a new version is effectively created. This new copy may be changed independently of the original file so the new copy becomes subject to a separate future history of modifications. If the user documentation is included directly in the source file, then it is also copied, effectively creating a new version of the documentation which will then follow the new source version.

Keep Source as a Formatted File

Generally a program exists in at least two pieces: the source file and the executable file. Often there are many pieces, since a large application will consist of a number of source files.

The same problems that apply to documentation and source also apply to source file and executable file: they should be kept in synch and should go together. The source of some bugs is that the source file and the executable file do not match.

One solution is quite radical and requires a re-thinking of the concept of the "file." The idea is that the operating system should support a thing called a "program file," which is not exactly like a "text file" or like an "executable file." The "program file" is a little like both text and executable. The idea is that a program file can be executed as well as edited. It combines both the source and the executable parts into one whole.

The programmer would use some sort of a text editor to edit the program file. Then the programmer would use a compiler to process the program file, updating its executable part. Finally, the programmer or user could execute a program file. In this scheme, the source and executable are always together. If one is copied, then both are copied.

It might also be desirable to include some sort of automatic version tracking into the program file. Whenever it is modified, perhaps the programmer's name and a timestamp are added to the program file. This becomes a simple form of automatic version control and tracking.

The "program file" might also contain more than just the "source" and "executable" portions. Perhaps the "user documentation" would be a third component of the file. This operating-system-of-the-future would then allow the user to see any program's documentation by accessing this component of the program file.

Concluding Remarks

In this document, I have discussed a number of ideas for increasing the reliability of programming languages. We want new languages to be designed in such a way that programs written in them tend to be more reliable.

These ideas range from common-sense suggestions that every language designer would agree on, to more complex or questionable ideas. I have focused on "C" and "C++", using examples from these languages, but many of the ideas apply to the design of any new programming language.

I'll conclude by reiterating how important program correctness is. Programming culture seems to be shifting and many novice programmers seem to put less emphasis on program correctness than it deserves. Meanwhile, there is a large body of research attempting to "solve" the problem or program quality once and for all, or at least come up with generalized formulae for reliability.

While fundamental research in reliability, correctness, and software engineering is unquestionably important, many gains can be made by applying the simple, pragmatic ideas I've discussed here. I believe that an appreciation for programming as engineering (as opposed to artistic creativity) is a route that warrants more attention. Following a few good practices can yield substantial gains in programming quality, reducing software costs and increasing reliability dramatically.