
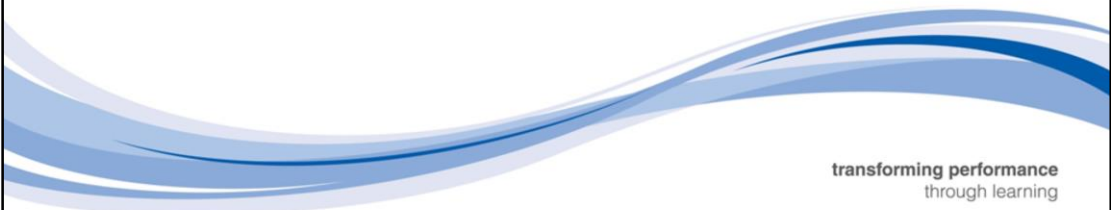


QACADV\_v1.0



# Appendix - Common C Pitfalls

## Advanced C



transforming performance  
through learning

## Common C Pitfalls

- **Lexical problems**
- **Syntax problems**
- **Runtime errors**
- **Out-by-one errors**
- **Summary**



---

The objective of this chapter is to describe some of the pitfalls found in the language so that they can be identified more easily in your code and therefore eradicated.

Some of the problems are commonplace, but we have included a few, more obscure ones, which can be extremely difficult to isolate. C is a very concise language and many of the problems found in everyday programming arise from the construction of overly complex expressions and test conditions. Runtime errors such as uninitialised pointers also account for a large number of mistakes in C programs, along with buffer overflow and out-by-one loop conditions.

A number of hints and tips are given throughout the chapter to help you to avoid these pitfalls in your programs.

## Lexical Problems: = and ==

- **= is an assignment; == is an equality comparison**

```
if ( x = y )
    printf("Same\n") ;

if ( x = y )
    printf("Same\n") ;

if ( ( x = y ) != 0 )
    printf("Same\n") ;

if ( ( 2 == y ) != 0 )
    printf("Same\n") ;
```

- **If x and y are the same value, print a message**
- **Assign y to x, and print a message if the value is non-zero**
- **As above, but clearer**
- **Useful tip: prevents the use of =**

Confusing the operators = and == is perhaps the most common lexical error found in C programs! The problem is made worse by the fact that many compilers do not give any hint or warning that there is anything wrong, although some compilers such as Microsoft do actually give a warning such as "assignment within conditional expression" when the highest warning level is used.

There are no hard and fast rules that can be applied to avoid this problem in all situations, although the last example above shows a helpful technique if an expression is being compared against a constant value; by placing the constant on the left-hand side of the expression, any mistaken use of = instead of == will now be trapped by the compiler:

```
if (2 == x) /* OK - is x equal to 2? */
...
if (2 = x)
...

```

Compiler traps this error - cannot assign to the constant value 2. For example, message is: "left operand must be lvalue".

## Lexical Problems: Token Size

- **Tokens are as large as possible as seen from left to right:**

`n-->0`

Means `n-- > 0` not `n - > 0`

Decrement and compare, not a structure access

`*p/*q`

Means `*p /* q` not `*p /* q`

Starts a comment!

`a---b`

Means `(a--) - (b)` not `(a) - (--b)`

`a+++++b;`

**How will this expression be compiled?**

The rules for establishing tokens are relatively straightforward as far as the compiler is concerned. There are six varieties of token:

- identifiers        - variable names, etc.
- keywords         - int, char, if, etc.
- constants        - integer constants, char constants), float constants
- string literals   - "Hello World", etc.
- operators        - any of the 45 operators in C, such as ++ and ==
- other separators - semicolons, colons, parentheses, etc.

Tokens may be delimited by white space, such as blanks, tabs and new lines, or by the start of a different token:

```
int a, b, c;
```

```
a=b*c+d*e++;
```

No white spaces! Compiler tokenises as `a = b * c + d * e++;`

One of the first tasks the compiler has to do is to split the program into a sequence of tokens. It adopts a fairly simple-minded approach to this task, by "gobbling up" the longest sequence of characters possible that constitute a recognised token. This rule is applied without regard to context.

Parentheses may be used to force the compiler to tokenise an expression in a particular way. Normally, parentheses are used to dictate precedence, but they may also be inserted into any expression to clarify the code.

## Lexical Problems: Logical vs. Bit-wise

- **Boolean operators generate true (1) or false (0)**
  - &&     and
  - ||       or
  - !        not
- **Bitwise operators work on corresponding bits**
  - &        and
  - |        or
  - ~        not

```
int x, y ;
char *sa, *sb, *sc, *sd ;

x = strcmp (sa, sb);
y = strcmp (sc, sd);

if ( x & y )
{
    printf(" a and b are
different\n");
    printf(" c and d are
different\n");
}
```

Return non-zero  
for different  
strings

The logical operators &&, || and ! allow complex boolean expressions to be combined in a single condition, and always give a result of 1 or 0.

Unfortunately, the bit-wise operators &, | and ~ are easily mistaken for these logical operators.

Whilst most people are aware of the differences between the logical and bit-wise operators, it can be irritatingly difficult to detect an error such as using & instead of &&. The compiler is unable to detect such programming errors, since all the operators in question expect integer operands.

Some software houses adopt an in-house standard which defines preprocessor symbols such as BITOR and OR to help to distinguish between bit-wise and logical operations. For example:

```
if ( (x == 1 OR x == 2) AND y > 0)
```

Does OR mean || or | ? Does AND mean && or & ?

Further problems can arise because of the different precedence of the operators && and |; the operator && has a higher precedence than |, so the following conditional statement will not work as expected! Parentheses would be required to achieve the desired effect:

```
int day, season;
```

```
if (day == SAT | | day == SUN && season == SUMMER) /* Original code */
if (day == SAT | | day == SUN && season == SUMMER) /* Intended logic */
if (day == SAT | | day == SUN && season == SUMMER) /* Actual logic! */
```

## Lexical Problems: Unexpected Octal

- Any integer constant beginning with the digit 0 is in base 8 (octal)
- K&R1 C permitted octal 8 and 9 digits!

```
struct Date
{
    int    day, month, year ;
};
struct Date BirthDay = { 08, 06, 63 } ;
struct Date AprilFool = { 01, 04, 93 } ;
```

It is worthwhile remembering that any integral constant starting with a zero is interpreted as an octal pattern. Similarly, any integral constant starting with the sequence 0x or 0X is treated as a hexadecimal value. There is no way of expressing an integral constant in binary; hexadecimal is usually used instead.

The printf function allows integers to be printed in decimal, octal or hexadecimal:

```
#define ESCAPE 27
printf ("decimal:%d\n", ESCAPE); /* Prints: decimal: 27 */
printf ("octal:  %o\n", ESCAPE); /* Prints: octal:   33 */
printf ("hex:    %x\n", ESCAPE); /* Prints: hex:    1b */
printf ("HEX:    %X\n", ESCAPE); /* Prints: HEX:    1B */
```

printf() also has a # format modifier which may be used to indicate whether the number being printed represents a decimal, octal or hexadecimal value. The number is displayed with a leading 0 or 0x to mimic the way the number would be written in a C program:

```
#define ESCAPE 27
printf ("decimal:%#d\n", ESCAPE); /* Prints: decimal: 27 */
printf ("octal:  %#o\n", ESCAPE); /* Prints: octal:  033 */
printf ("hex:    %#x\n", ESCAPE); /* Prints: hex:    0x1b */
printf ("HEX:    %#X\n", ESCAPE); /* Prints: HEX:    0X1B */
```

## Spot the Bug (1)

```
/*
 * Calculate the amount of income tax to be paid on earned amount.
 * Above a threshold of £25000, the tax rate increases.
 */

double tax_to_pay (double amount_earned)
{
    double tax_rate;

    if (amount_earned < 25,000)
        tax_rate = 0.25;
    else
        tax_rate = 0.40;

    return amount_earned * tax_rate;
}
```

This is the first of a number of pages on which a bug has been included in the code fragment. If you do not want to see the answer, do not look at the following paragraphs!

The problem shown above is quite subtle and hard to detect, and stems from the way we write numeric values greater than 1000 in the UK. Some European programmers on the continent would never make this mistake.

The problem arises because of the comma in the number 25,000. The comma is interpreted as the comma operator, which has the lowest precedence of all operators in C. The comma operator is often referred to as the *sequence operator*, since it causes expressions to be evaluated from left to right in sequence, with the result of the previous expression being thrown away.

Thus, the `if`-test will be parsed as follows by the compiler:

```
if ( (amount_earned < 25), 000)
```

The result of the first test (`amount_earned < 25`) will be evaluated first, but the result will be discarded immediately. The outcome of the whole `if`-test will be governed by the second expression `000` which follows the comma. Clearly, this expression always gives a result of 0, so the result of whole the test will always be false, implying that everyone will be taxed at 40%!

Note: the comma operator cannot be ambiguous in function calls, since the comma is taken as an argument separator. If a comma operator is required when specifying a function argument, the expression must be enclosed in parentheses.

## Pitfalls: Operator Precedence

**Beware of operator precedence: it's not always what you might expect!**

```
#define FLAG 0x02      /* 0000 0010 */
int x;
```

```
if ( x & FLAG ) ...
```

← Implicit test against zero

```
if ( x & FLAG != 0 ) ...
```

← Explicit test, but != has higher precedence

```
if ( (x & FLAG) != 0 ) ...
```

← Explicit test with parentheses

```
printf ("%d", x * 8 + 1);
```

← Normal multiplication (\* has precedence)

```
printf ("%d", x << 3 + 1);
```

← Erroneously replace with shift operator

```
printf ("%d", (x << 3 ) + 1);
```

← Parentheses are required

The examples shown above illustrate the problems encountered when mixing bit manipulation with relational and arithmetic operators. It is typical of the C language that it allows errors to be introduced when a programmer is trying to write more explicit code, such as introducing explicit tests against zero.

In the first block of code, the programmer falls foul of the fact that the boolean operators `==` and `!=` have a higher precedence than the bit-manipulation operators `&`, `|`, `^`, `~`, `<<` and `>>`. Explicit parentheses are required to overcome the problem.

In the second block of code, the shift-left operator has been used as a quicker alternative to multiplying a number by a power of two. The problem is that the shift operator has a lower precedence than the `+` operator. Tricks like these used to result in more efficient code on older C compilers, since a shift instruction in assembly language is faster than a multiply instruction. These days however, with the advent of modern optimising compilers, the compiler is capable of choosing for itself between `mult` and `shift` operations, so the programmer may as well use the multiplication operator for greater clarity.

In summary, the bit-wise operators have notoriously non-intuitive precedence rules. To be sure of avoiding problems, always use parentheses when using any of the bit-wise operators.



## Order of Evaluation in Expressions

- **The order in which expressions are evaluated is not specified in the standard**
  - Do not write code which depends upon the order of evaluation

```
int i = 2;
int sqr;
sqr = ++i * i;
```

X

```
void copy ( int d[ ], int s[ ], int n )
{
    int i = 0;
    while ( i < n)
        d[ i ] = s[ i ++ ];
}
```

X

One of the strengths of the C language is the compiler's ability to exploit implementation features whenever possible. One of these is its ability to choose the order of evaluation of expressions in a binary operation. Although a compiler is unlikely to change its mind from one day to the next, we should not assume that the evaluation order is the same for different compilers.

Typical problems arise when the operands of a binary operation are dependent upon each other, such as when the increment/decrement operators are used. Some compilers evaluate the left-hand expression first (`++i`), whereas other compilers evaluate the right-hand expression first (`i`). This will naturally lead to problems when using different compilers!

```
sqr = ++i * i; /* Result is 3 * 3 if LHS expression evaluated first */
           /* Result is 3 * 2 if RHS expression evaluated first */
```

A similar problem can arise when a variable is passed by reference. If the function modifies the variable, it should not be used again in the rest of the expression. In addition, when there are calls to functions that modify a global variable, the global variable should not be used again in the same expression. Consider the following example.:

```
int gCount = 2;

int IncCount (void)          /* Increment global counter variable */
{
    return ++gCount;
}

int sqr;

sqr = IncCount() * gCount;    /* 3 * 3 if LHS exp. evaluated first */
                               /* 3 * 2 if RHS exp. evaluated first */
```

It must be remembered that there are three binary operators that *are* evaluated in strict order (left to right). These are the logical AND (`&&`), logical OR (`||`) and the comma operators. You can also add the ternary operator to this list.

## Semantic Pitfalls

```
if ( (today == sat) || (today == sun) );
    printf ("Weekend");
```



```
for (j = 0; j < 10; arr[j++] = 0)
    /* Null Statement */ ;
```



```
for (j = 0; j < 10; arr[j++] = 0)
    /* Null Statement */ {}
```



```
#define NOTHING
for (j = 0; j < 10; arr[j++] = 0)
    NOTHING ;
```



The examples shown above illustrate the common problem of a semicolon appearing at the end of a conditional expression. The effect of placing a semicolon at the end of a conditional expression is to create a null statement that is executed if the test evaluates to TRUE. C compilers do not flag this as an error, since there are some situations where this is actually the desired effect; the classic example of this is the `strcpy()` function presented in Kernighan and Ritchies' book:

```
char *strcpy(char *d, const char *s)
{
    char *ret = d;
    while ((*d++ = *s++) != '\0');
        /* Null statement on each cycle */
    return ret;
}
```

Such cases are very much the exception, however, and the presence of a semicolon at the end of a conditional expression is more often than not a programming error. If a null statement is intended, the examples shown in the slide above should help to make these intentions plain to anyone reading the code.

## Spot the Bug (2)

```
#include <ctype.h>

int digits, vowels, wspace, others;

void classify (int c)
{
    switch ( tolower(c) )
    {
        case '0': case '1': /* Count all digits
        case '2': case '3':    0 through to 9 */
        case '4': case '5':
        case '6': case '7':
        case '8': case '9': ++digits; break;

        case 'a': case 'e':
        case 'i': case 'o':
        case 'u': ++vowels; break;

        case ' ': case '\t': ++wspace; break;

        default: ++others; break;
    }
}
```

Classify characters as digits, vowels, white space, or other character

Count digits

Count vowels

Count spaces

Count others

The code fragment shown above contains two deliberate errors. The first and most obvious error is the format of the comment appearing within the switch statement; the comment is not terminated at the end of the line, but continues on to the end of the next line, thereby eliminating the values '2' and '3' from the case branches. This problem can be overcome by the adoption of a simple inhouse standard such as the following:

Comments must be terminated at the end of a line and may not span more than one line. Where more extensive comments are required, the comment must be terminated at the end of each line and re-opened on the next line as follows:

```
for (i = NUM-1; i >= 0; i--)
    /* Loop through array of salaries */
{
    /* in reverse order, printing each */
    printf("%.2f\n", sal[i]);
    /* value on a separate line, to */
}
/* two decimal places. */
```

The second problem in the slide is far more subtle. Although goto labels are rarely used in structured programs, the C compiler allows a goto label to be placed almost anywhere in a program, including inside a switch statement! In the switch statement shown above, the intention was clearly to define a default branch to detect characters not falling into any of the other categories. However, default has been accidentally mistyped as default. Rather than complaining about a syntax error, the compiler takes default as a goto label. The switch statement is left without a genuine default branch, so any characters that do not match the preceding case branches will be ignored.



## Spot the Bug (3)

```
#include <stdio.h>

/*
 * Return 1 if the next character in "infile" is a newline character,
 * otherwise return 0.
 * If an EOF is reached, report an error and return EOF.
 */
int isnewline (FILE *infile)
{
    int c;

    if ( (c = getc(infile) ) != EOF)
        if (c == '\n')
            return 1;
    else
    {
        printf ("EOF reached\n");
        return EOF;
    }
    return 0;
}
```

Perhaps the biggest problem with the example function is the number of return statements. A return statement is like a goto, since it causes an unconditional jump from one point in a program to another.

Wherever possible, only have a single return statement in a function. This makes the function easier to understand and maintain;. The logic of the function is also incorrect, because the else clause corresponds to the second if-test rather than the first if-test (as was intended). This is quaintly referred to as the "dangling-else" problem - an else clause will always correspond to the most recent if-statement still available. In order to achieve the desired logic, an extra set of braces are necessary to group the else clause with the first if-test, as shown in the example below on the left. The example on the right uses a single return statement and is perhaps the preferred implementation for the function.

```
int isnewline (FILE *infile)
{
    int c;

    if ( (c = getc(infile) ) != EOF)
    {
        if (c == '\n')
            return 1;
    }
    else
    {
        printf ("EOF reached\n");
        return EOF;
    }

    return 0;
}
```

```
int isnewline (FILE *infile)
{
    int c, res;

    if ( (c = getc(infile) ) == EOF)
    {
        printf ("EOF reached\n");
        res = EOF;
    }
    else if (c == '\n')
    {
        res = 1;
    }
    else
    {
        res = 0;
    }
    return res;
}
```

### Spot the Bugs (4)

```
#include <stdio.h>
/*
 * Write a file, given by "fname", to the standard output device.
 */
void showfile (const char *fname)
{
    int c;
    FILE *fp = fopen (fname, "r");

    if ( fp == NULL)
        fprintf (stderr, "Failed to open %s\n", fname);
        return;

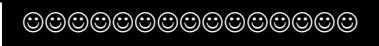
    while (c = getc (fp) != EOF)
        putchar (c);
    fclose (fp);
}
```

There are two problems in the above code fragment. The first problem is associated with the if-test; in the absence of explicit braces, the compiler assumes that the extent of the if-body is a single statement. Therefore, the return statement is executed regardless of the result of the test. Braces are required around the fprintf and return statements so that they are grouped within the if-body.

The second problem is one of precedence again and occurs within the while loop. The effect of this code is to call the getc macro first, which yields an integer value. Because the != operator has a higher precedence than the = operator, the integer is checked against EOF. The truth value 1 or 0 will then be assigned to the int variable c - a disaster! The value assigned to c will be 1, until EOF is reached. When this value is displayed as a character, a "smiley face" is displayed in the character set:



Input:

Output: 

The only easy solution is to use parentheses if you are unsure of the precedence. An experienced eye will detect problems like these relatively quickly, but in a large program it is all too easy to overlook simple mistakes like these!

## Runtime Pitfalls

- **A very large class of errors!**
- **Arithmetic overflows**
  - Overflows
  - Divide by zero
- **Runtime library errors**
  - If a routine returns an error indication, check it
- **Problems with pointers can be particularly difficult to trace**
- **Boundary errors**
  - Loop limits
  - No data
  - Too much data supplied
  - OBO (out-by-one ) errors

---

Runtime errors are an unfortunate fact of life that often come back to haunt us long after we have passed a program off as being tested and bug-free.

The main difficulty with runtime errors is that the compiler is often unable to detect any problem. Syntactically, a program may obey all the rules demanded by the compiler, yet still not work as expected.

In such situations, the programmer needs all the help available. A source-code debugger can prove invaluable for hunting down a well-concealed bug in a program that contains a runtime error. There are some situations, however, where source code debuggers are not available, such as in a real-time process control system. In such cases, memory dumps and tracing facilities can be helpful.

As in medicine, so in software, the phrase "prevention is better than cure" is very true. Where possible, runtime errors should be anticipated and detected during the development and testing phase, rather than being reported in the field. In order to make this a reality, intensive module testing is strongly recommended.

In order to be useful, testing must be thorough and exhaustive. Here are some suggestions:

- Test each path through a function, and check it by using invalid data from the user.
- Test function inputs and outputs in usual and exceptional circumstances.
- Check and guard against numeric overflow and underflow.
- Check the boundary conditions for loops - guard against "off-by-one" errors.
- Initialise pointers - even initialising them to NULL is better than nothing at all, since this will at least result in a "hard crash", rather than silently overwriting someone else's data.
- Check for buffer overflow - remember to allow for null terminators in strings!
- Check the return value from functions where applicable, such as `fopen()` and `malloc()`.
- Perform peer code reviews - yes, it is boring but it does help to trap mistakes!

In the next few pages we present some of the most common runtime errors in C. See how many of them you have made already!

## Spot the Bug (5)

```
#define LEN 80

char *getline (FILE *fp)
{
    int i, c;
    char buf [LEN];

    for (i = 0; i < LEN; i++)
    {
        if ( (c = getc(fp)) == EOF || c == '\n')
            break;

        buf[i] = (char) c;
    }
    if (c == EOF && i == 0)
        return NULL;

    buf [i] = '\0';
    return (&buf[0]);
}
```

Get next line from file and return a pointer to it. Return NULL on end of file.

This code contains two nasty bugs that might lie dormant for a long time, silently causing other data in the program to be corrupted.

The code within the function looks perfectly robust, and it is as long as you forget about the outside world. The buf array is an automatic variable, local to the function, and as such will be destroyed when the function terminates. Therefore, the line of text read from the file will be lost as soon as the function returns. The return statement returns the address of the local buffer to the calling function; if that function is quick enough, the buffer may still be lying around on the stack, and the data can be retrieved. However, if the calling function invokes some other functions in the meantime, these functions will overwrite the buffer with their own local variables on the stack, and the contents of the buffer will be lost. This is one of the most frustrating types of bugs to detect - a code fragment that works sometimes and fails at others.

To avoid this problem, never return the address of a local automatic variable, since when the function terminates, the variable will be destroyed and its memory will be re-used by subsequent functions.

There are two possible solutions to this problem. Firstly, the function interface could be changed so that the function takes a buffer as an argument from the calling function. The buffer is now defined in the calling function and will not be destroyed when getline() terminates. The second solution is to declare the buffer as a static variable in getline(). static variables are not destroyed when a function terminates, but persist across function calls and retain their previous contents.

The code fragment shown above contains a second deliberate mistake concerning the length of the buf character array. The array should be one character longer to allow for LEN real characters plus a null terminator at the end of the line. In order to clarify the situation, character arrays are often written as follows:

```
char buf [LEN + 1]; /* LEN real characters, plus the null terminator */
```



## Spot the Bug (6)

- What will the following program do?

```
int main (void)
{
    int i ;
    int a[6] ;

    for (i = 0 ; i <= 6 ; i++)
    {
        printf ("Initialising element %d\n", i) ;
        a [i] = 0 ;
    }

    return 0;
}
```

The function shown above has an invalid loop construct which iterates once too often and assigns zero to `a[6]`. This is of course a mistake. However, the symptoms of such a simple mistake can be quite alarming, depending on how the compiler lays out the local variables `i` and `a` on the stack. For example, if the function is compiled using the Microsoft compiler, then `a[6]` actually coincides with `i`. When `a[6]` is reset to zero, this has the effect of setting `i` back to zero on the last iteration, and the loop starts over again!

How can such a problem be avoided? The best advice is to perform stringent unit-testing techniques on loop constructs to check for boundary conditions. The `assert()` macro can be used at the end of the function to ensure that the variables have their expected values.

## Spot the Bug (7)

- **What is the problem with this simple function?**
- **How can the problem be rectified?**

```
double CalcMeanSpeed (double Distance, double Time )
{
    return Distance / Time ;
}
```

This program fragment looks very simple and problem-free, but consider what happens if Time is zero. In this case, the result of the division is undefined. Some compilers will cause a runtime error to occur, whilst others produce a value of -1 from a divide by zero! The only solution is to check for zero in the function explicitly, perhaps by using a construct such as the following:

```
return (Time ? Distance/Time : 0);
```

Another solution is to use the `assert()` macro, defined in `assert.h`. Although both of these solutions work, this sort of situation will typically arise many times in any application which performs arithmetic. An implementation strategy is required, such as ensuring that a non-zero value is assigned to Time in the first place. Checking that a variable is initialised correctly is a one-off task, and is usually easier than having to check that the variable is valid each time it is used.

A final word of caution: when Time is extremely small, there are potential overflow problems when the value is used as the denominator in division.

## Summary

- **A number of practical hints and tips will help you to avoid programming errors such as those seen in this chapter**
- **Know the C language, including any local non-Standard extensions**
- **Make your intentions plain - do not sacrifice readability for a small improvement in performance**
- **Program defensively and employ a careful incremental test strategy**
- **Team programming standards and code beautifiers can be helpful**

---

The best strategy for avoiding problems is to *know the language and its limits*. Be aware of the standard and local extensions and to be able to distinguish between the two.

*Do not talk yourself into seeing what is not there.* Psychologists call it "cognitive dissonance". We often see what we expect to see. Also, do not believe comments; the compiler doesn't even see them.

*Make your intentions plain.* Use extra parentheses, extra white space and temporary variables. In most cases these will not affect the quality of the final code, but will enhance the readability and maintenance.

*Check all boundary conditions and implementation limits.* Your program should handle array bounds, extreme data values, zero string lengths and zero loop iterations. It should be checked to work for changes in internal size and representation. Your program should work when "under stress", e.g. on Monday mornings and Friday evenings!

*Program defensively.* Always assume the worst; it will happen someday! The day it will choose to happen will be the worst possible day to put it right!

Lastly, and perhaps most importantly, *test, test and test again*. Do not use yourself - have arrangements with colleagues and teammates to check and test each other's modules. This should be made part of a project team's standards. You should not leave the testing to the users!