Defensive Programming

Comp-206 : Introduction to Software Systems
Lecture 18

Alexandre Denault
Computer Science
McGill University
Fall 2006

Write a C function that tell me if a series of number are ordered.

Don't ask me questions!
Don't talk to your neighbor!

Defensive Programming

- The biggest danger to your application is user input.
  - It's uncontrolled, unexpected and unpredictable.
- The input sent to your application could be malicious.
- Or it could just be something you never expected.
- Debugging takes a lot of time.
- Defensive Programming is a technique where you assume the worst from all input.
- Also known as Proactive Debugging.
- Let's look at Alex's three rule of Defensive Programming.

First Rule

- The first rule of defensive programming is :

Never Assume Anything!

- A lot of problems in applications can be attributed to unexpected input.
- Another common source of error is the programmer assuming something about a programming language.

Input Validation

- As previously mentioned, data from the user cannot be trusted.
- As such, all input must be validated.
- For each input:
  - Define the set of all legal input values.
  - When receiving input, validate against this set.
  - Determine the behavior when input is incorrect:
    - Terminate
    - Retry
    - Warning

Validation Example

- Lets assume input expect an monetary value.
  - Is the amount numeric?
  - Is the amount large enough or small enough?
  - Is it positive?
  - What decimal symbol was used?
  - How many decimal point does it have? (ex: 20.2555$)
  - Is it only composed of number? (ex: 10+25 is considered numeric by some systems)

Testing Strategy

- Just testing that it works is not good enough.
- You need to test the error cases, to see that your application reacts accordingly.
- Then you need to test for the illogical
  - Strange ASCII character test
  - Rolling head test
- Ask other people to test your application
  - First start with the CS testers
  - The asks non-CS people

Order of Precedence

- The order of precedence it the set order that statements are resolved.
- However, when debugging, it's not always easy to see errors in the order of precedence.

```
if (InVar = getc(input) != EOF)
```

- When in doubt, add the proper parenthesis.

Size of Variables

- Some primitive data types have different values depending on the operating system and the hardware platform.
  - For example, integers have been 8,16,32 and 64 bits.
- Assuming the size of a data type can be disastrous when working on different platform.
- In C, the size of data types are defined in limits.h.
- In addition, C has the *sizeof* operator which will calculate the size of a variable.
- You need to be especially careful on integer operation.

  short x = 10 000 * 10
  - Will x overflow?

limit.h

```
/* Minimum and maximum values a `signed char' can hold.  */
#  define SCHAR_MIN     (-128)
#  define SCHAR_MAX     127

/* Maximum value an `unsigned char' can hold.  (Minimum is 0.)  */
#  define UCHAR_MAX     255
define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#  define USHRT_MAX     65535

/* Minimum and maximum values a `signed int' can hold.  */
#  define INT_MIN       (-INT_MAX - 1)
#  define INT_MAX       2147483647

/* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
#  define UINT_MAX      4294967295U

...
```

Second Rule

- The second rule of defensive programming is to use Standards.
- Proper coding standards address weaknesses in the language standard and/or compiler design.
- They also defines a format or “style” used for writing code.
- Every software development team should have an agreed-upon and formally documented coding standard.

Programming Standards

Coding Standard

- Coding standards make code more coherent and easier to read.
  - Thus reduce the likelihood of bugs.
- They cover a wide range of topics.
  - Variable naming, indentation, position of brackets, content of header files, function declaration, etc.
- Many different coding standards for every different programming language are available on the web.
  - One of the most popular, used for variable names, it the Hungarian Notation.
  - For programming in C, the Indian Hill C Style and Coding Standards seems popular.
- When working on an existing project, find out if a coding standard is used. If not, impose one.

Hungarian notation

- The Hungarian Notation is a language independent standard for naming variable.
- Variable name starts with one or more lower-case letters which are mnemonics for the type or purpose of that variable.
  - ulAccountNum : variable is an unsigned long intege
  - szName : variable is a zero-terminated string
  - bBusy : boolean
  - cApples : count of items
  - iSize : integer (systems) or index (application)

Magic Numbers

- Never use constant values in your code.
  - Makes the code difficult to understand.
  - Makes the code difficult to maintain.

```
int friction = (4.3563 / 5.463) * x;
```

- Use constant variable instead.

```
const int PI = 3.14159265
int surface = PI * r * r;
```

Pi

3.1415926535 8979323846 2643383279
5028841971 6939937510 5820974944
5923078164 0628620899 8628034825
3421170679 8214808651 3282306647
0938446095 5058223172 5359408128
4811174502 8410270193 8521105559
6446229489 5493038196 4428810975
6659334461 2847564823 3786783165
2712019091 4564856692 3460348610
4543266482 1339360726 0249141273
7245870066 0631558817 4881520920
9628292540 9171536436 7892590360
0113305305 4882046652 1384146951 ...

Indentation

```
if (strcmp(tree->value, value) > 0) {
if (tree->left != NULL) {
addToBinaryTree(tree->left, value);
} else {
tree->left = createBTNode(value);
}
} else {
if (tree->right != NULL) {
addToBinaryTree(tree->right, value);
} else {
tree->right = createBTNode(value);
}
}
```

Proper indentation

- Proper indentation is key to making your code readable.

```
if (strcmp(tree->value, value) > 0) {
    if (tree->left != NULL) {
        addToBinaryTree(tree->left, value);
    } else {
        tree->left = createBTNode(value);
    }
} else {
    if (tree->right != NULL) {
        addToBinaryTree(tree->right, value);
    } else {
        tree->right = createBTNode(value);
    }
}
```

Third Rule

- The third rule of Defensive Programming is to keep your code as simple as possible.

  Complexity breeds bugs
- Software should only contain the features it needs.
- Proper planning is key to keeping you application simple.
  - Before coding, you should write down the major ideas of what you are trying to do (module names, files, etc.)

What makes software complex?

Contract

- Functions should be seen as a contract.
- Given input, the execute a specific task.
- They should not do anything other than that specific task.
- If they cannot execute that task, they should have some kind of indicator so that the callee can detect the error.
  - Throw an exception (doesn't work in C)
  - Set a global error value
  - Returns an invalid value
    - NULL?
    - False?
    - Negative number?

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

-- www.refactoring.com

- By itself, refactoring is not a bug-fixing technique.
- However, refactoring is a good technique to battle feature creep:
  - Features are often added during development.
  - These features are more often the source of problems.
  - Refactoring fights this by forcing the programmer to reevaluate the structure of his/her program.
- Refactoring can help you keep you application simple.

Third-party libraries

- Code reuse is not just a smart-choice, it's a safe choice.
- Odds are that the library has proven itself and is much more stable than anything you could build short-term.
- Although code reuse is highly recommended, many questions must be addressed before using someone else's code:
  - Do this do exactly what I need?
  - How much will I need to change my design?
  - How stable is it? What reputation does it have?
  - How old is the code?
  - Who built it?
  - Are people still using it? Can I get help?
  - How much documentation is there?

Summary

- 1st : Never Assume Anything
- 2nd : Use Coding Standard
- 3rd : Keep it simple