The Art of Design and Development

!Goofing up code

Flawed logic

- Most popular mistake and source of bugs and employment

```
#define isOctal(c) ((c) > '0' && (c) <= '8')
```

```
#define isOctal(c) ((c) > '0' && (c) <= '7')
```

Flawed logic

- Checking wrong return value
  - Oct-Nov 2017 of MacOS High Sierra security bug
  - A simplified picture:

```
int verify_password(hashValue *password_supplied , userData user, valData *retval)
```

Popular irritants

- Bad indentation

```
for(i++;i<100;f[i++]=`\0`);
*i = `\0`;
return (`\n`);
```

```
for (i++; i < 100; i++)
    f[i] = '\0';
*i = `\0`
return (`\n`);
```

- Can be lethal in indentation sensitive languages like Python

Popular irritants

- Writing complex to read conditional expressions

```
if (!(blk_id < actblocks) || !(blk_id >= unblks))
```

```
if (blk_id >= actblocks || blk_id < unblks)
```

Popular irritants

- Not using parenthesis when needed
  - Normally makes code easier to understand, just don’t over do it

```
if (x & SETMASK == BITS)

actually means if (x & (SETMASK == BITS))
```

```
if ( (x & SETMASK) == BITS)
...
```

Popular irritants

- Complex expressions
  - One source of this in recent times is copy pasted code from the net

```
r += rval * (re = (5*n > (x+math.log(m))) ? c[r]:c[r-1])
```

```
if (5*n > (x + math.log(m))
    re = c[r];
else
    re = c[r-1];

r += rval * re
```

Understand Language semantics

- Order of evaluation of expressions
    - Language specific aspects need to be understood

```
scanf("%d %d", &yr, total_profit[&yr])
```

```
scanf("%d", &yr)
scanf("%d", total_profit[&yr])
```

Understand Language semantics

- Writing Idiomatic code
    - Language specific aspects are best adhered to

```
j = 0;
while (j <= n-1)
    arr[j++] = BASE_VAL;

for (j = 0; j < n; )
    arr[j++] = BASE_VAL;
```

```
for (j = 0; j < n ; j++)
    arr[j] = BASE_VAL;
```

Understand Language semantics

- Writing Idiomatic code - another example
  - It may be shorter at times too

```
do {
    ch = getchar();
    putchar(ch);
} while (c != EOF);
```

```
while ((ch = getchar()) != EOF)
    putchar(ch);
```

Readability

- Reusing code
  - May be risky at the cost of readability & possibly little performance gain

```
switch(ch) {
     case `-`: sign = -1;
     case `+`: ch = getchar()
     case `.`: break;
     default: if (!isdigit(ch))
           return 0;
}
```

Readability

- Reusing code
  - May be risky at the cost of readability & possibly little performance gain

```
switch(ch) {
    case '-':
      sign = -1;
    //fall through here to execute + code also!
    case '+':
      c = getnextchar(ch);
      break;
    case '.':
      break;
    default:
      if (!isdigit(c))
           return 0;
      Break;
}
```

Readability

- Reusing code, not always necessary
  - Code clarity always better*

```
if (ch == '-') {
    signval = -1;
    ch = getnextchar();
} else if (ch == '+') {
    ch = getnextchar();
} else if (ch != '.' && !isdigit(ch)) {
    return INVALID
}
```

Readability

- Writing code which is clear to understand
  - Sometimes it may make no difference to a writer

```
func doSomething (i int) {
    ....
    //Some code here
        ....
    //Some more code here
    return u
}
```

```
func doSomething (i int) {
     ....
     //Some code here
     u = &User {
     Name: "Harry",
     Email : "harry@nobody.com"
     }
     ....
     //Some more code here
     return u
}
```

Readability

- Writing code which is clear to understand
  - But to a reader it can & does

```
func doSomething (i int) {
     ....
     //Some code here
     u = User {
     Name: "Harry",
     Email : "harry@nobody.com"
     }
     ....
     //Some more code here
     return &u
}
```

Readability

- Scoping
  - Shadowing gets tricky at times, gets missed even in reviews

```
var u *user //→ A GLOBAL
//And far down..somewhere
func doSomething (u string) {
     var u *user //Possibly unaware that there’s a global by this name
     //Some code here
     //....
     //And further down
     for i=0; i < MAX_USERS; i++ {
          var u *user
    //Aware that there’s a global and outer scope variable by this name?
    ...
     }
     ....
     //Some more code here
     return &u
}
```

Readability

- Keep Happy Path easy to read
  - The main (un-indented) path is usually the “Happy” path, align it left
  - Avoid hiding happy path logic nested inside braces
  - Error path and edge cases indented

```
func doSomething () {
     if !err {
          //Main code path here
     }
     //error handling here

}
```

```
func doSomething () {
    if err {
      //Handle error
    }
    //Main/Happy code path here

}
```

Readability

- Line of sight in code
  - Better code with fewer indented paths
  - Happy return statement is last line

```
if something.OK() {
     //do something more
     err := something.Do()
     if err == nil {
               startJob()
               log.Println("working...")
               doWork(something)
               log.Println("finished")
               return nil
     } else {
               return err
     }
} else {
     return errors.New("something not ok")
}
```

```
if !something.OK() {
     return errors.New("something not ok")
}

//do something more
err := something.Do()
if err != nil {
    return err
}
startJob()
log.Println("working...")
doWork(something)
log.Println("finished")
return nil
```

Readability

- Consider moving big conditional blocks to functions

```
func processSomething(int val) {
     switch(val) {
          case ADDED_GAIN:
               add_gain();
               break;
          case DEFERRED_GAIN:
               defer_gain();
               break;
          case INVEST_NOW:
               invest_now();
               break;
     }
}
```

Best avoided

- Every language has good parts
    - And `Not so good` parts of the language
- E.g. : Macros in C
    - Can sometimes cause more problems than they solve

```
#define isupper(c)  ((c) >= 'A' && (c) <= 'Z')
...

while(isupper(c = getchar()))
```

Best avoided

- Macros in C
  - Parenthesize macro arguments and body

```
1/square(x)

#define square(x) (x) * (x)
…
```

```
#define square ((x) * (x))
```

Other tips

- Better to define numbers as constants than macros
  - People looking into the code MUST clearly understand what’s being done

```
if (ch >= 97 && ch <= 122)
```

```
if (ch >= 'a' && ch <= 'z')
```

Commenting

- Avoid unnecessary comments
    - Sometimes these are in abundance, even when code is obvious

```
if (ch == '\n') /* Newline character */
    action = newline
else if (isdigit(ch)) /* number */
    action = digit
else if (ch = ':') /* colon */
    action = colon
```

Commenting

- Important not to comment everything
  - Your code should be easy to understand
- Best to comment API’s, Globals
  - Globals may be used by large number of functions/others
  - API’s will be used by others using your library/software
  - Internal functions, again can be commented on a need basis
- Misleading comments is a source of bugs
  - Comments not updated to reflect current thinking/updated business logic
  - For complex stuff, if required put up a version wise explanation or ref to a design note

Commenting

- To clarify parts of the programs
  - Not easily understood by the code

```
int strcmp (char *s1, char *s2)
/*string comparison routine returns -1 if s1 is */
/*above s2 in ascending order list, 0 if equal*/
{



}
```

```
int strcmp (char *s1, char *s2)
/*returns < 0 if s1 < s2, > 0 if s1 > s2, 0 if equal */
/*ANSI/ISO 9899-1990*/
{

}
```