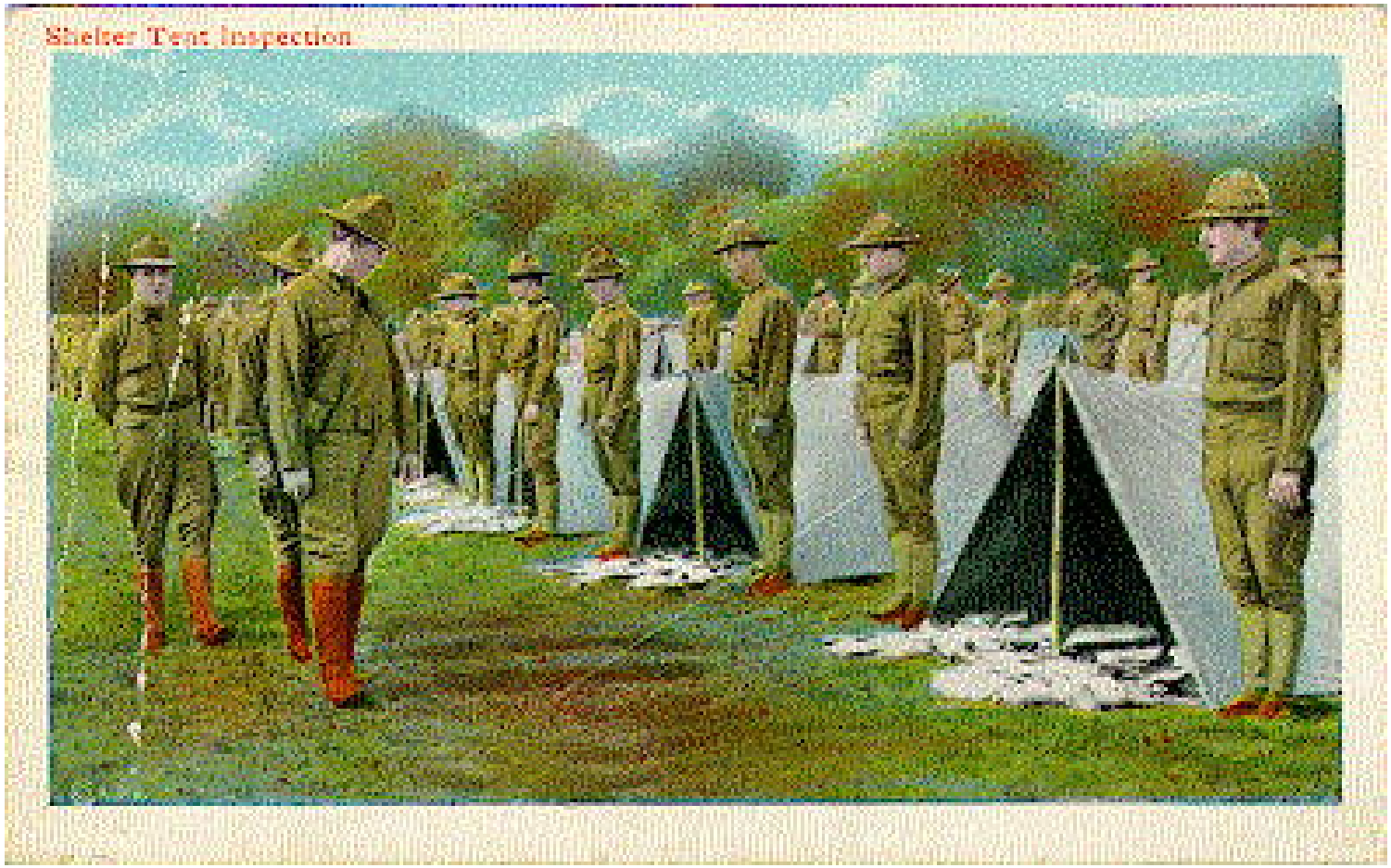
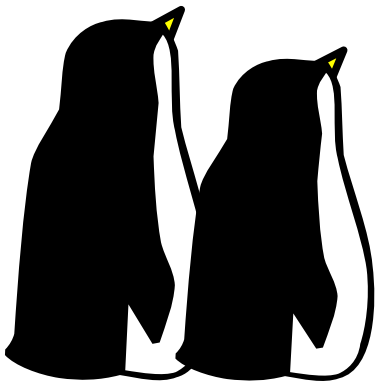
Inspections





On the surface

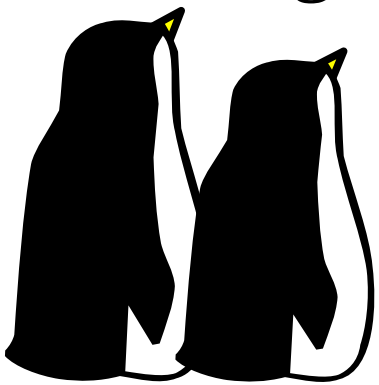
Nice puppy!

I love the

pony tail!

What a cute

little girl!

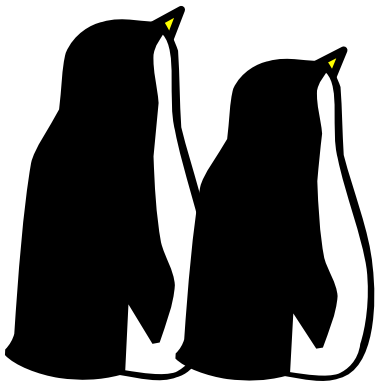


But an inspection shows

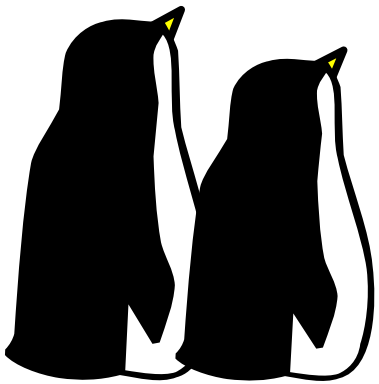
I'm adult not a baby!

I'm wearing a Top Knot, not a pony tail!

and



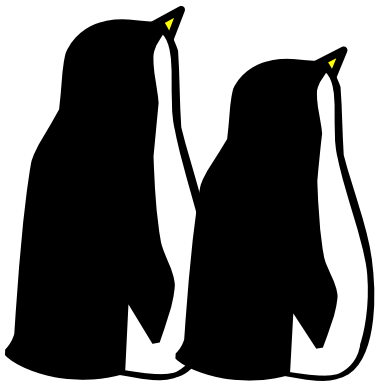
I'm not a girl!



Why Inspect?

- It is the single best way of reducing errors.

- It produces not only better code but better *programmers*.



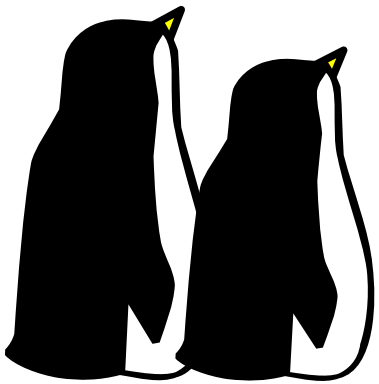
Types of Inspections

1) Checklists

2) Master Coder Review

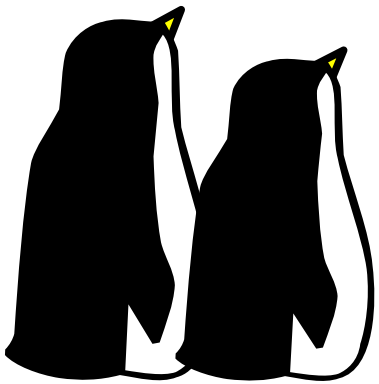
3) Walkthroughs

4) Data Analysis



Checklists

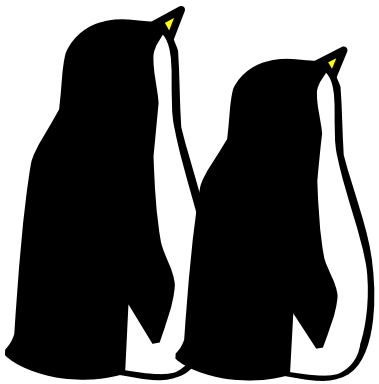
- Quick
- Simple
- Large return on the time invested
- Can lead to surprisingly good results



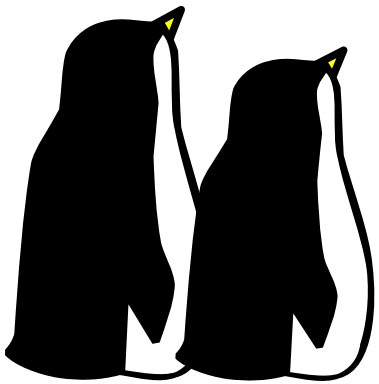
Designing the Rules

Before we can create a checklist we need something to check.

- Style Sheets
- Programming Rules

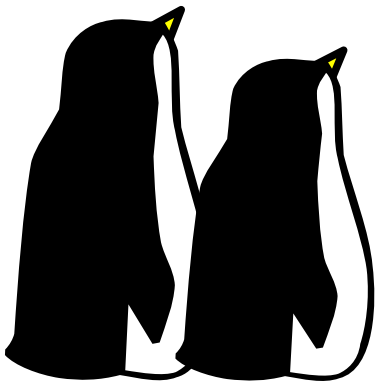


Style Rules

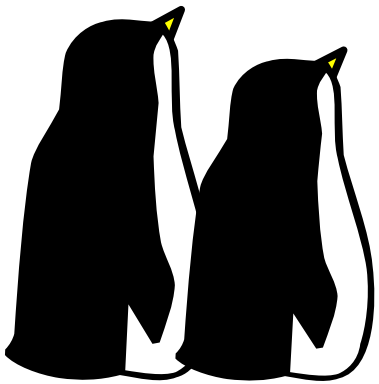


Designing the Rules

- **Simple**
- Explained
- Unambiguous. Binary rules (it's right or it's wrong -- no debate)
- Instantly verifiable



Simple




UNDERGROUND
Travel Information 0171-222-1234
Travelcheck 0171-222-1200
© London Regional Transport
Key to Lines
Bakerloo
Central
Circle
District
East London
Hammersmith & City
Jubilee
Metropolitan
Northern
Piccadilly
Victoria
Waterloo & City
Docklands Light Railway
British Rail
Peak hours only
Restricted service
Peak hours and Sunday mornings
Peak hours only
Under construction
Peak hours only
Peak hours only
Under construction
Restricted service
Interchange stations
Connections with British Rail
Connections with British Rail within walking distance
Airport interchange
Closed Sundays
Closed Saturdays and Sundays
Mornington Crescent closed for rebuilding
For opening times see poster journey planners
Certain stations are closed during public holidays
Diary 1A 4 95

Bad Password Picking Rule

•**CORPORATE DIRECTIVE NUMBER 88-570471**

•

•In order to increase the security of all company computing facilities, and to avoid the possibility of unauthorized use of these facilities, new rules are being put into effect concerning the selection of passwords. All users of computing facilities are instructed to change their passwords to conform to these rules immediately.

•

•**RULES FOR THE SELECTION OF PASSWORDS:**

•

•1. A password must be at least six characters long, and must not contain two occurrences of a character in a row, or a sequence of two or more characters from the alphabet in forward or reverse order.

•

•Example: HGQQXP is an invalid password. GFEDCB is an invalid password.

•

•2. A password may not contain two or more letters in the same position as any previous password. Example: If a previous password was GKPWTZ, then NRPWHS would be invalid because PW occurs in the same position in both passwords.

•

•3. A password may not contain the name of a month or an abbreviation for a month. Example: MARCHBC is an invalid password. VWMARBC is an invalid password.

•

•4. A password may not contain the numeric representation of a month. Therefore, a password containing any number except zero is invalid. Example: WKBH3LG is invalid because it contains the numeric representation for the month of March.

•

•5. A password may not contain any words from any language. Thus, a password may not contain the letters A, or I, or sequences such as AT, ME, or TO because these are all words.

•

•6. A password may not contain sequences of two or more characters which are adjacent to each other on a keyboard in a horizontal, vertical, or diagonal direction. Example: QWERTY is an invalid password. GHNLWT is an invalid password because G and H are horizontally adjacent to each other. HUKWVM is an invalid password because H and U are diagonally adjacent to each other.

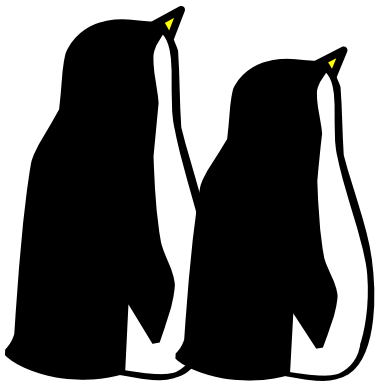
•

•7. A password may not contain the name of a person, place, or thing. Example: JOHNBOY is an invalid password.

•

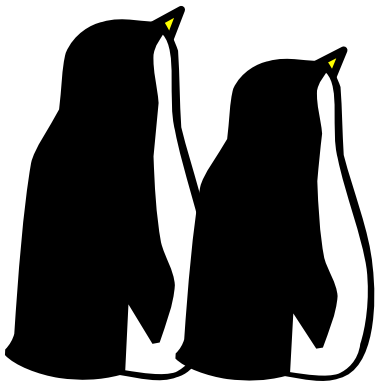
•Because of the complexity of the password selection rules, there is actually only one password which passes all the tests. To make the selection of this password simpler for the user, it will be distributed to all supervisors. All users are instructed to obtain this password from his or her supervisor and begin using it immediately.

•



Rules must be Simple

- Chapter 3 of "C Elements of Style" contains over 20 pages of rules on how to name a variable. No one can remember all of them. (`http://www.oualline.com`.)

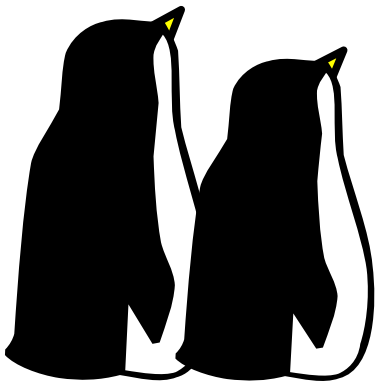


- vs.
- Variable names are all lower case, whole words (or industry standard abbreviations) separated by underscore.
-
- Example: `count_of_events`.

Designing the Rules

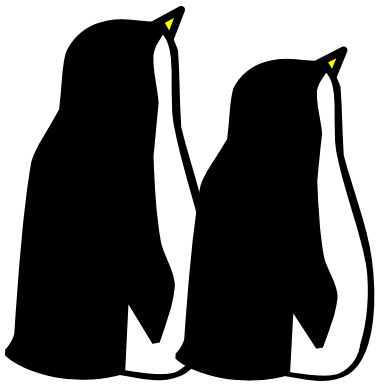
- Simple
- **Explained**

  (People follow the rules better if they know there was a good reason behind the creation of the rule.)
- Unambiguous. Binary. (it's right or it's wrong -- no debate)
- Instantly verifiable



Explained

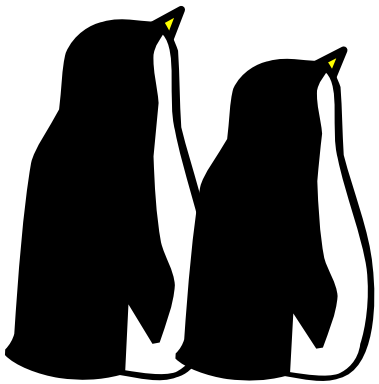
- or not....



Rule Design

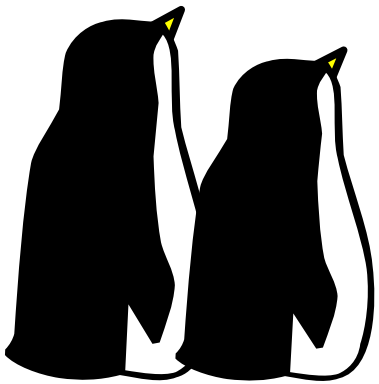
- The reason behind the rule must be documented.

  (People follow the rules better if they know there was a good reason behind the creation of the rule.)



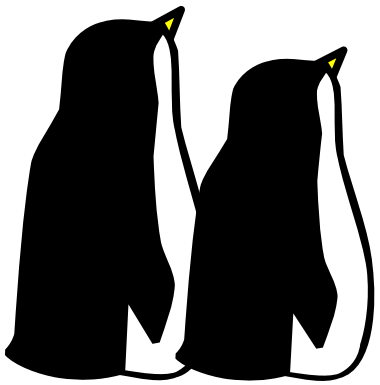
Example

v9) Abbreviations are not to be used in variable names except for `_ptr` and industry standard acronyms.



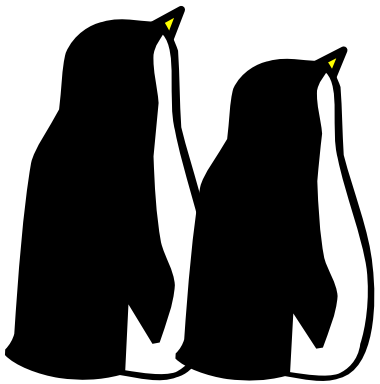
Why?

- How many ways can you spell `ground_point`?
-
- (A "ground point" is a basic unit of measurement used throughout software used to analyze aerial photographs.)



Ground Point

- gp
- gnd_pt
- ground_point
- GroundPoint
- .....

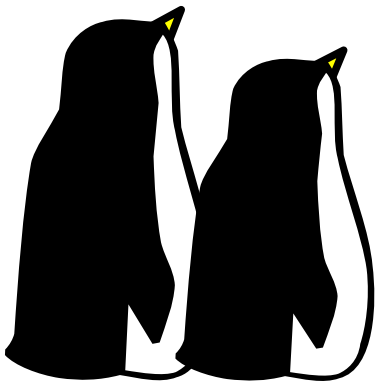


Reason for the rule:

- Error reduction

  There are many different abbreviations for "ground" and "point". Especially when you consider that many programmers abbreviate by throwing out random letters.

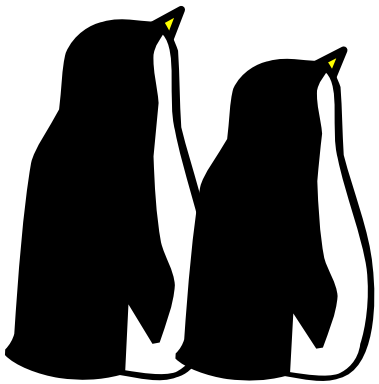
  There is only one full spelling of "ground".



Reasons for the Rule

- Make things easy programmers who's primary language is not English.

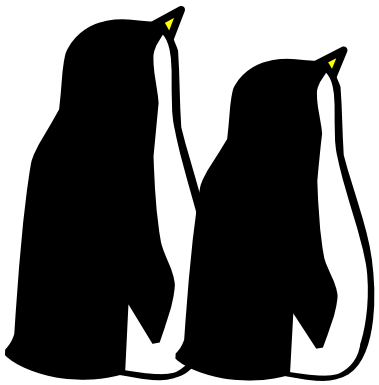
  You can find "ground" in a Finnish-English dictionary. Try looking up "gndpnt".



Feedback: Errors => Rules

When an error occurs:

1) Not only fix the error but,
2) Determine the cause of the error
3) See if you can devise a rule to prevent a recurrence of the error.



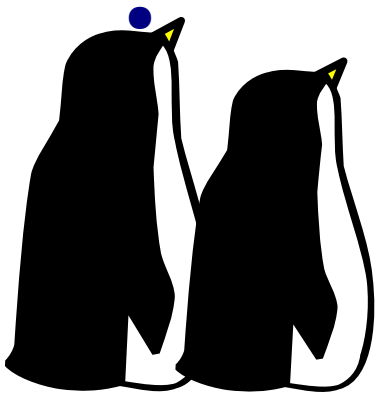
Example

- The error:

```
struct id_type id_data;

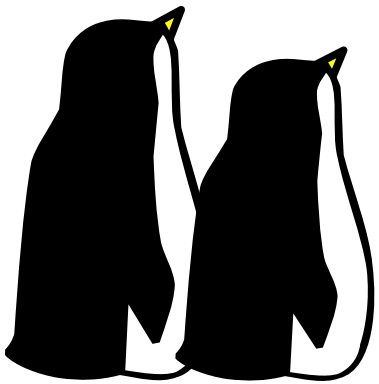
memset(&id_data, '\0',
        sizeof(struct  name_id_type));
```

- This causes random memory to be zeroed.



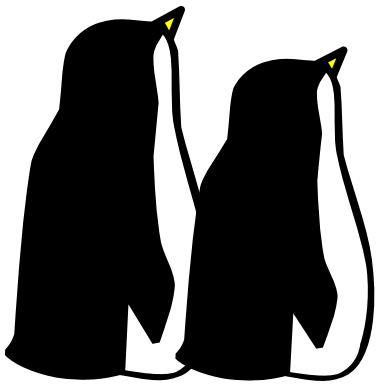
The Rule

- Whenever possible, the third argument to `memset` must be `sizeof(first[0])`.
-
- Example:
- `memset(f_ptr, '\0', sizeof(f_ptr[0]));`
- `memset(&st, '\0', sizeof(st));`

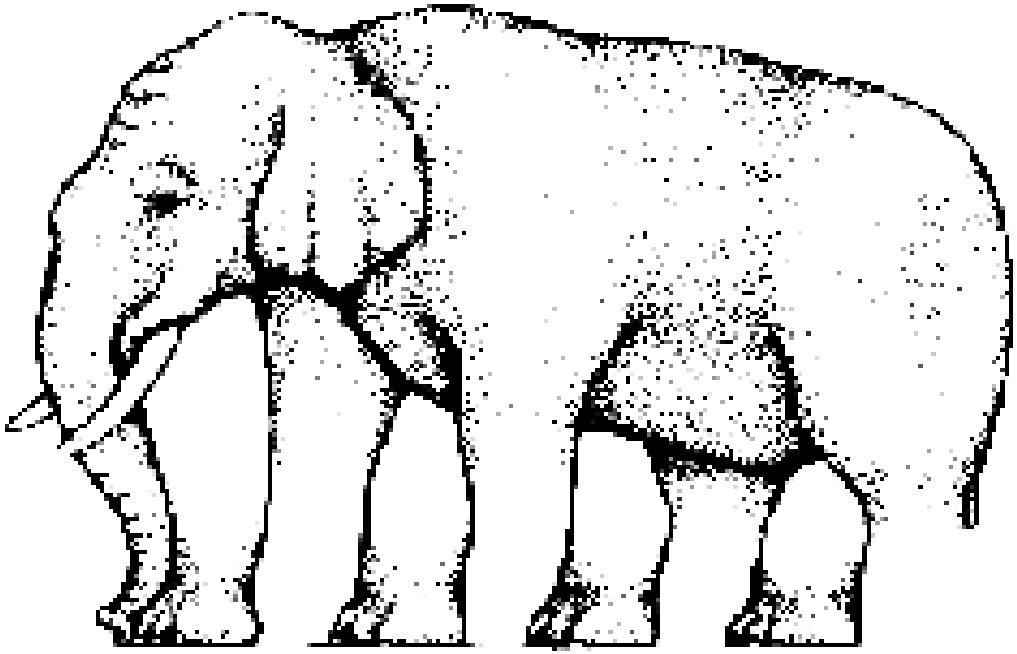


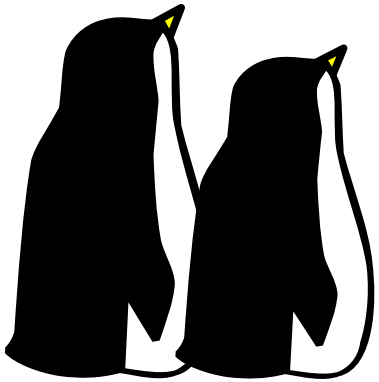
Designing the Rules

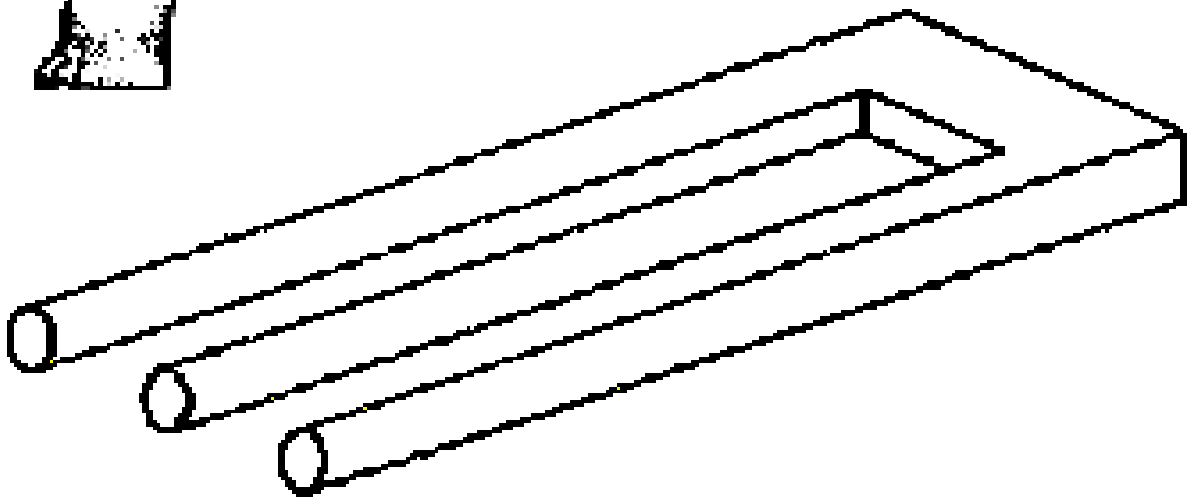
- Simple
- Explained
- **Unambiguous. Binary rules (it's right or it's wrong -- no debate)**
- Instantly verifiable



Unambiguous



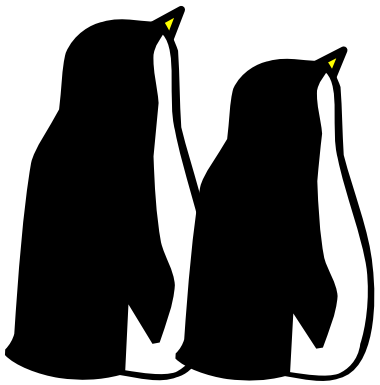




Ambiguous

Bad Rule

- Heading comments must clearly describe the function that follows.

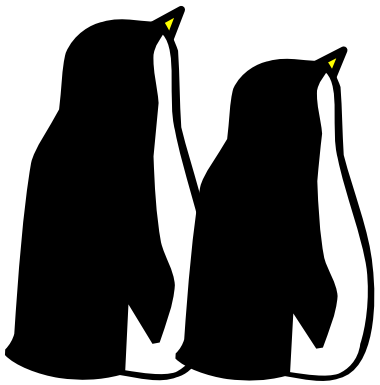


Unambiguous

Good Rule

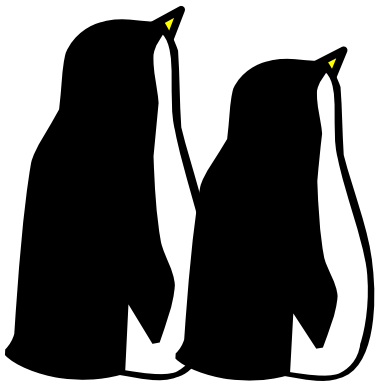
D1 All variable declarations must end in a comment describing the variable

```
int count; // Count of items the
           // data array (Right)
int svm;   (Wrong)
```

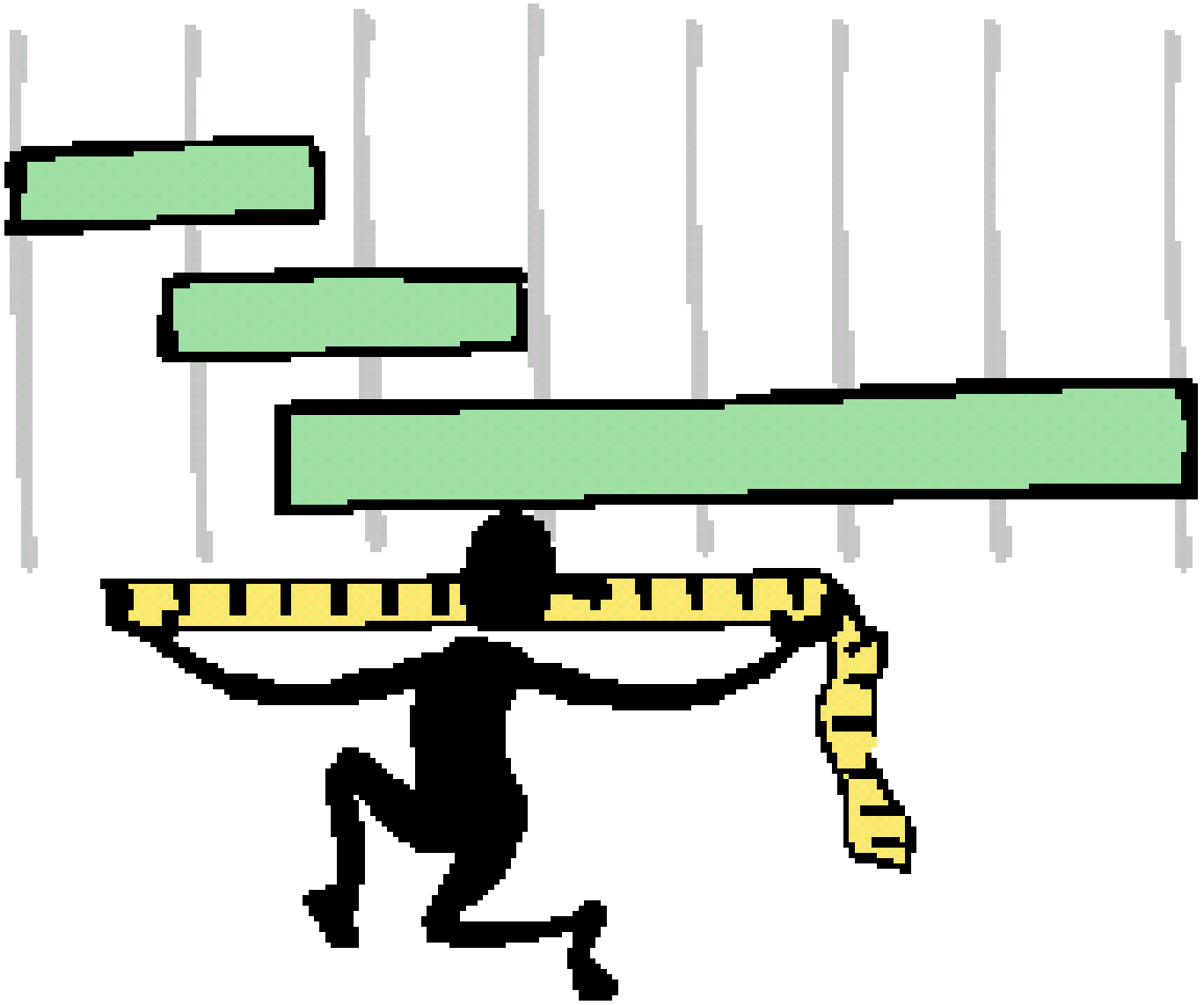


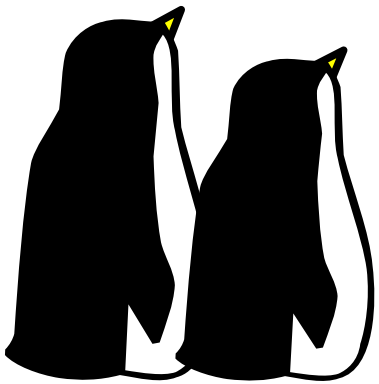
Designing the Rules

- Simple
- Explained
- Unambiguous. Binary rules (it's right or it's wrong -- no debate)
- **Instantly verifiable**



Instant Verification

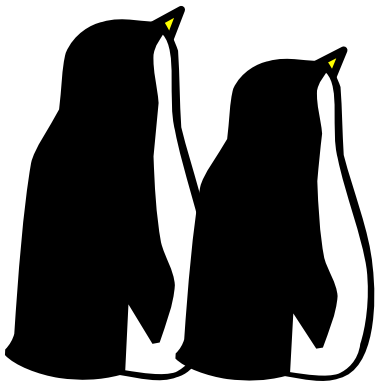




Hard to verify

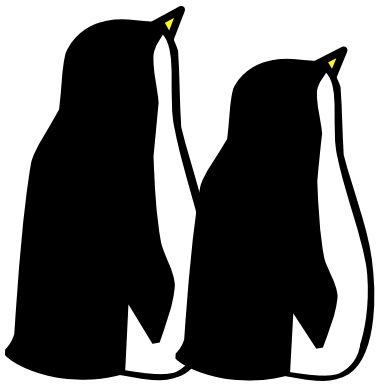
Bad Rule

- All loops must exit or be marked as infinite loops.

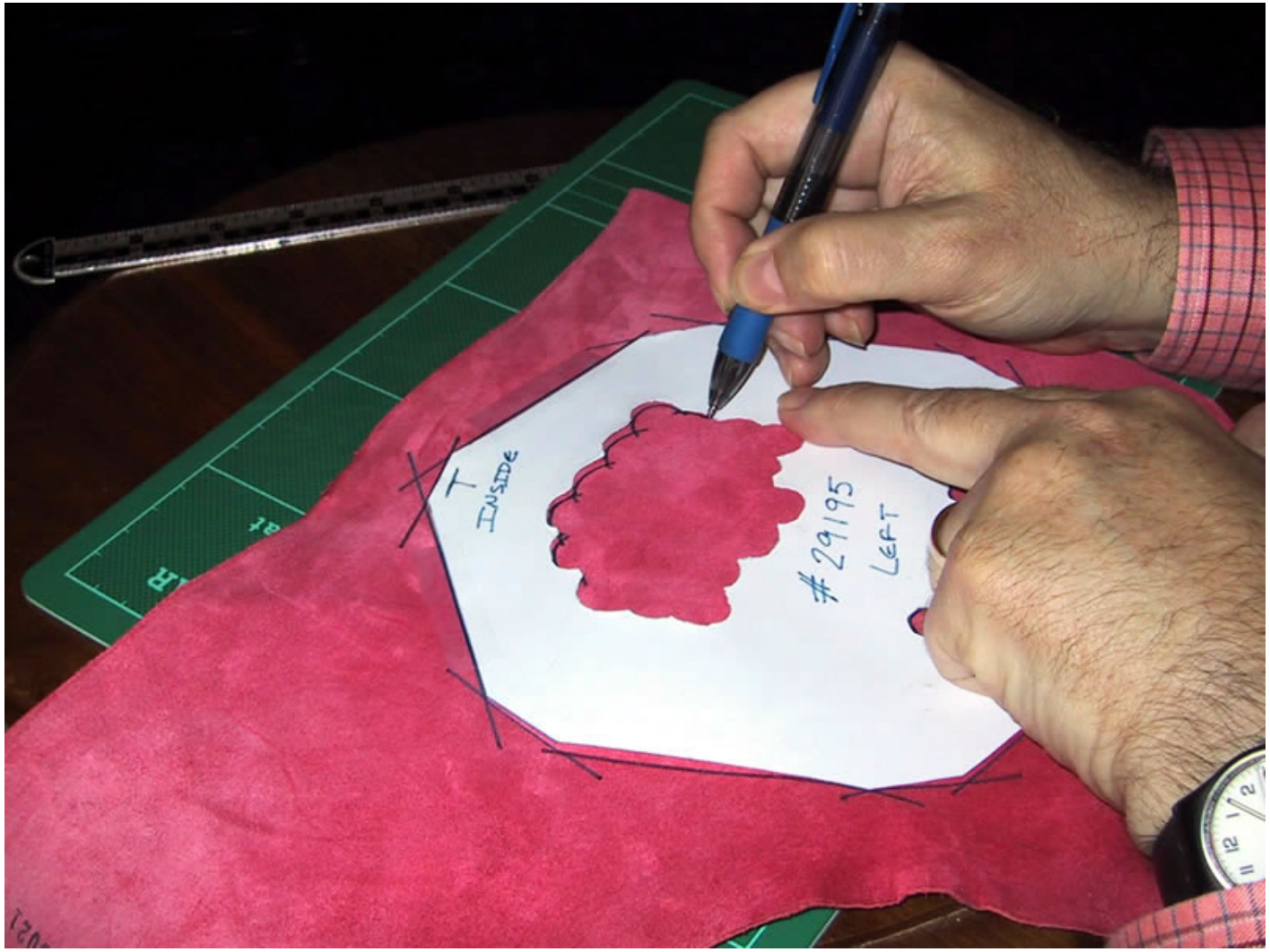


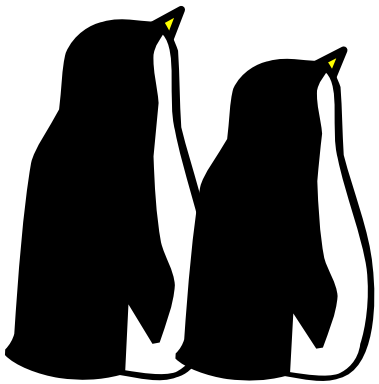
Easy to Verify

- All case statements must end in:
- ```
  break;
  ```
- or
- ```
  // Fall through
  ```



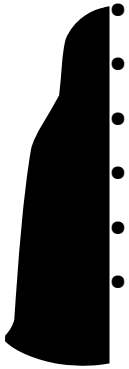
Templates



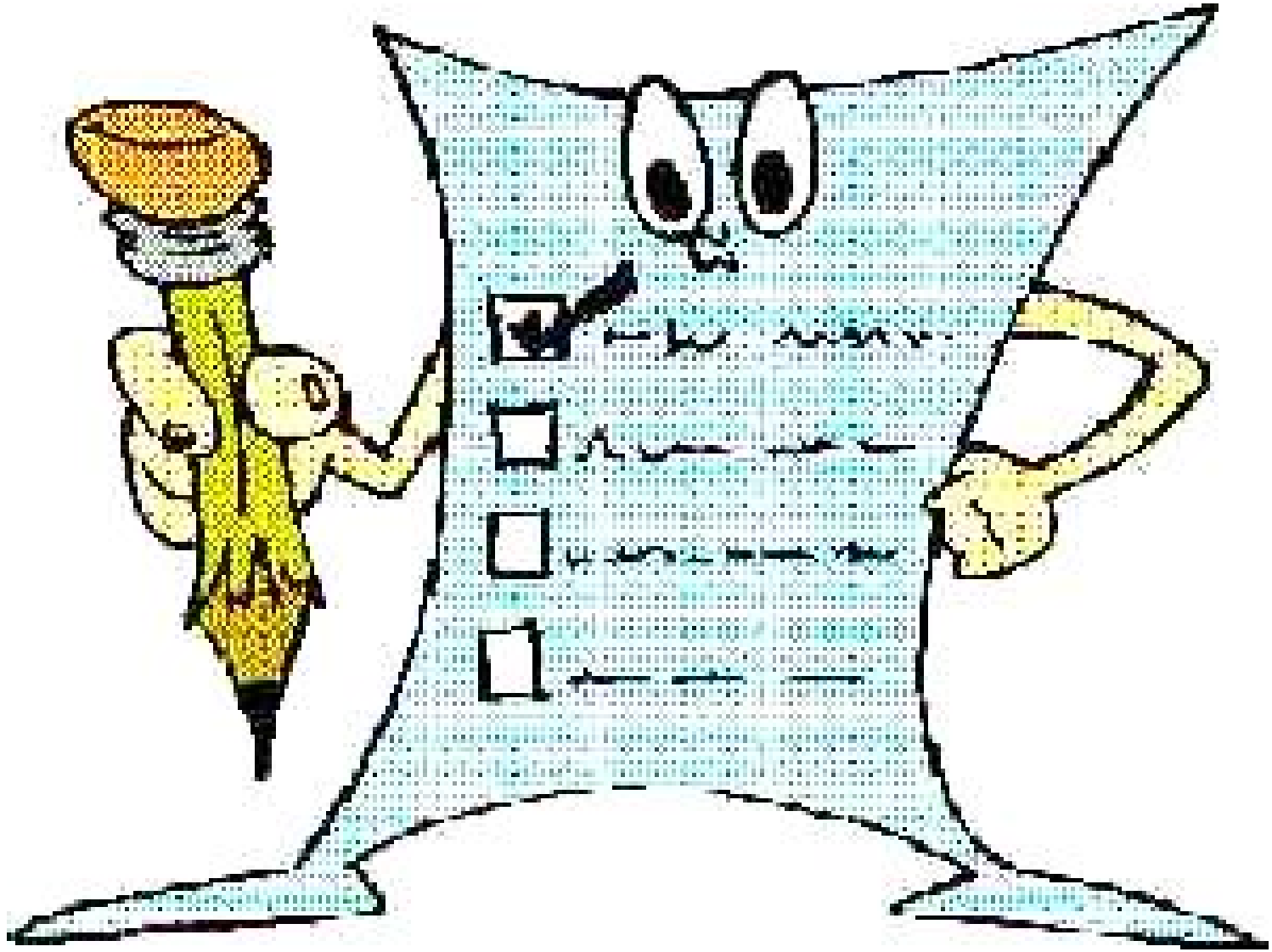


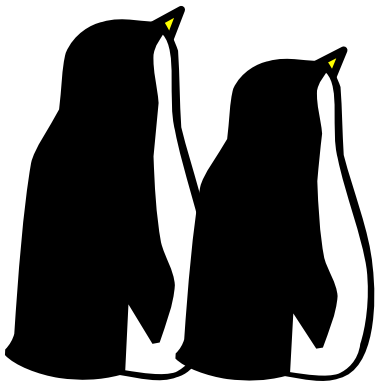
Templates Help Keep Things Standard

```
/*****************************************************
 * funct_name -- One line description of the function   *
 *                                                      *
 * Multi-line description of what the function does.    *
 *                                                      *
 * Memory usage:                                        *
 *   Indicate if the function allocates memory          *
 *   and if so, how the memory is to be disposed of.    *
 *                                                      *
 *   If the function disposes of memory, indicate       *
 *   who you expect to have allocated it.               *
 *                                                      *
 * Returns:                                             *
 *     What does the function return.                   *
 *****************************************************/
extern int16 funct_name(
    const int32 param1,// Description of the first parameter
    const int32 param2 // Description of the second parameter
);
```



Checklist

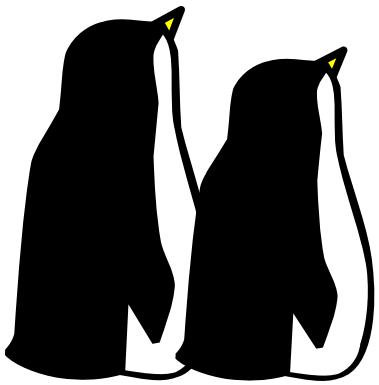




Array Protection

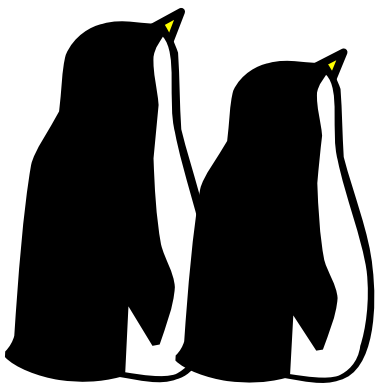
- A1) All array accesses must protected by an assert statement. (Exceptions are allowed only in those cases where it's absolutely impossible to determine the array bounds.)
- Reason: Array protection

```
assert((i >= 0) &&
    (i < sizeof(array)/
        sizeof(array[0])));
x = array[i];
```



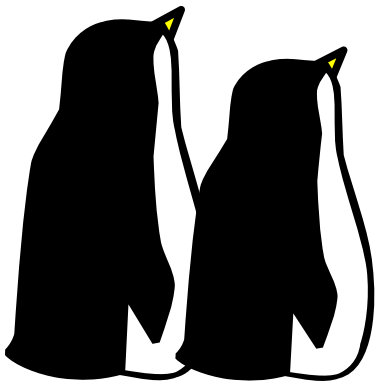
Avoiding Evil Procedures

- A2) Use of the following functions prohibited:
  - `a. strcpy`
  - `b. strcat`
  - `c. gets`
  - `d. sprintf`
  - Reason: These functions can easily cause memory memory problems.



Avoiding Evil Procedures

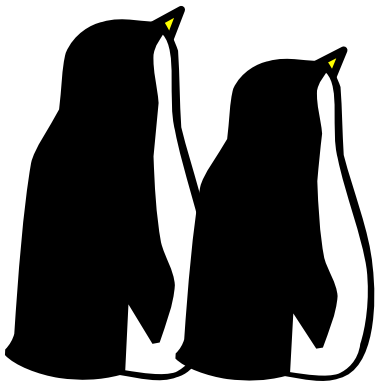
Don't Use	Use
`strcpy`	`strncpy`
`strcat`	`strncat`
`gets`	`fgets`
`sprintf`	`snprintf`



File Related Rules

- F1 File uses standard comment structure for the heading. (See programming template.)
-

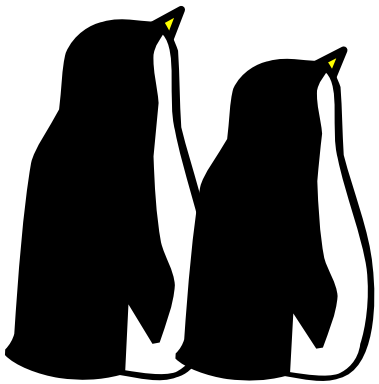
•Reason: It doesn't matter which standard we use, as long as everyone uses the same standard.



File Related Checklist

- F2 Files should no be longer than 3,000 lines and only unusual files use be longer than 6,000 lines long.
-

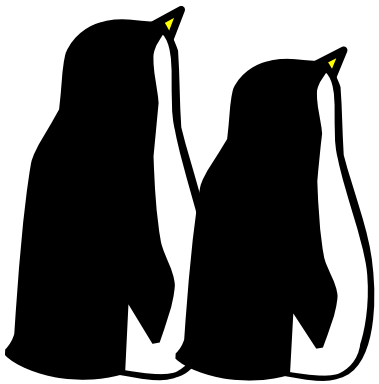
  - Reason: A person should be able to understand a complete file. Too long and it's not possible to understand. Also files longer than 3,000 lines are more difficult to edit and print.



File Related Rule

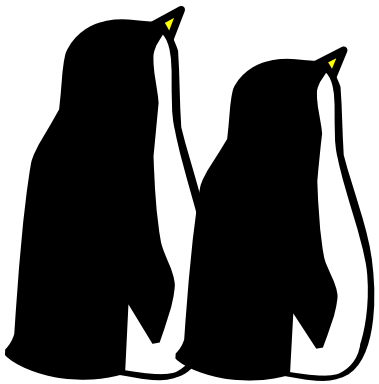
-
- F3 The file compiles without warnings with full compiler warnings turned on.
-

  •Reason: Let the compiler check as much as possible so that error don't make it into the source.



Procedure Related Rules

-
- P1 All procedures have a standard comment heading.
-
  - Reason: Everyone needs to do things the same way.
-



Procedure Example

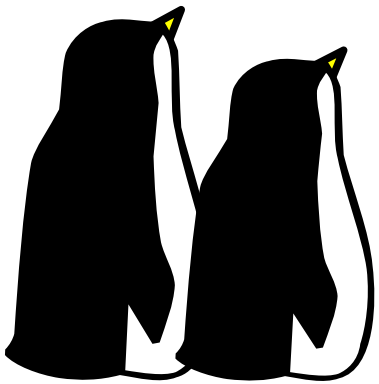
```
/*************************************************
 * close_in_file                                 *
 *     Close the input file.                     *
 *                                               *
 * Returns                                       *
 *     0 -- Success                              *
 *    NZ -- Error code (see errno.h)             *
 *
*************************************************/
static int close_in_file(
    /* The file to initialize */
    struct in_file_struct *the_file
)
{
```

Procedure Related Rules

-
- P2 All private functions are declared **static**
-

  •Reason: Scope of a all functions should be as limited as possible. If a function is **static** then we know the scope is limited to the file in which it is declared.

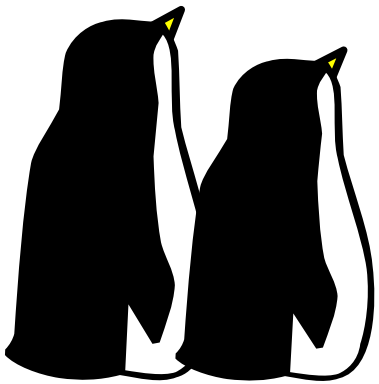
-



Procedure Related Rules

-
- P3 All public functions have a prototype in the header file.
-

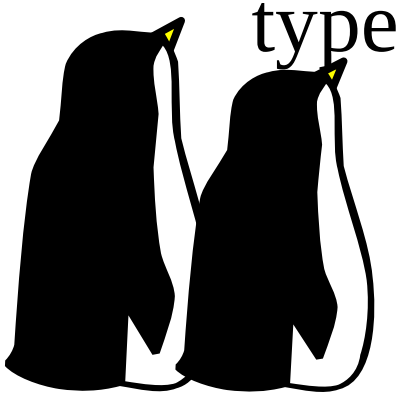
  •Reason: If we put the prototypes in a standard place we know where to find them.



Procedure Related Rules

-
- P4 The return type of a function does not default
-

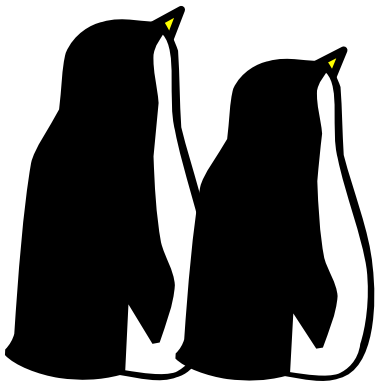
  •Reasons: If the programmer lets the return type default we can't tell the difference between a function that should return an **int** and a function where the programmer forgot to put in a return type.



Procedure Rules

-
- P5 Function and variable definitions in header files must be declared **`extern`.**
-

  •Reason: Be consistent Treat external data and procedures the same. Use **extern**.

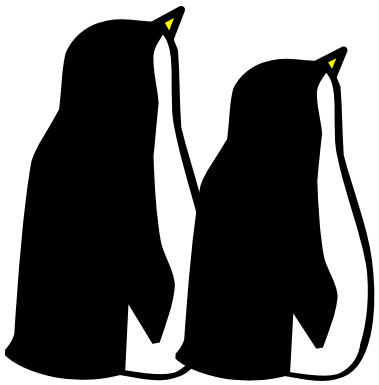


Procedure Rules

- P6 Modules must include all the header files which contain external definitions of functions and variables defined in the file. (In other words if *foo.h* contains

```
extern int foo_size;
```

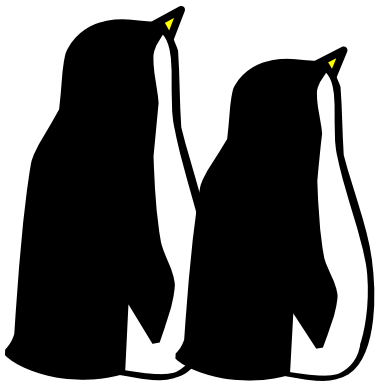
and *foo.c* defines `foo_size` then *foo.c* must include *foo.h*.



- Reason: Gives us a standard place where external declarations are put.

Indentation

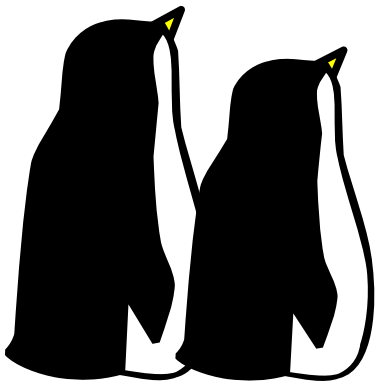
- I1 Indentation is 4 spaces. Tab stops are every 8 spaces.
-
  - Reason: Studies at Rice University have shown that 4 spaces makes the code the most readable.
  - The default tab stop on most terminals is 8.



Indentation

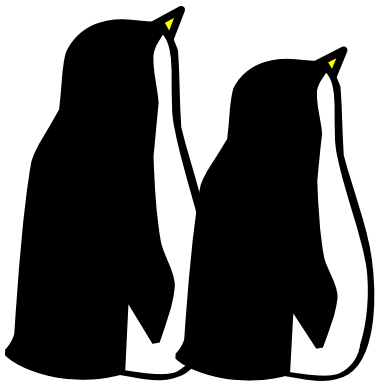
-
- I2 Curly braces are on separate lines indented the same as the statement that precedes them.
-

•Reason: No good reason for this placement, but we need some standard.



Indentation

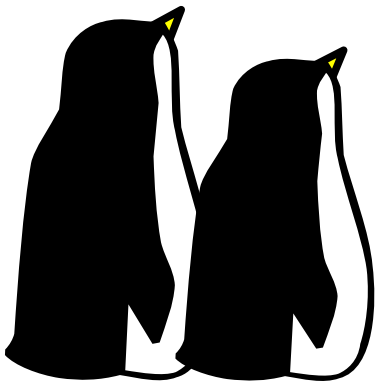
- I3 Statements inside curly braces are indented one level.
-
  - Reason: There are many variations of indentation. We needed to pick one.



Variables

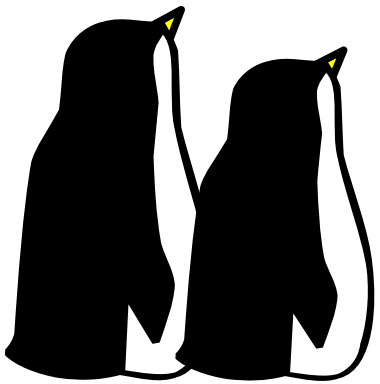
-
- V1 Variables are lower case words separated by underscores (`this_is_a_var`)
-

  •Reason: It doesn't matter which naming convention you use as long as it's simple and everyone uses the same one.



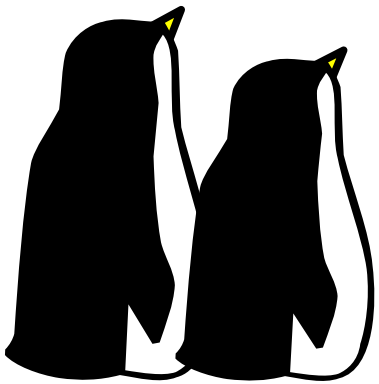
Bad Naming Conventions

- What are the rules for X Motif constants and variables?
- ```
  XmNtextHighlightCallback
  xmTextWidgetClass
  WidgetClass
  ```
- What's the naming convention for the Microsoft Windows API?
-


  ```

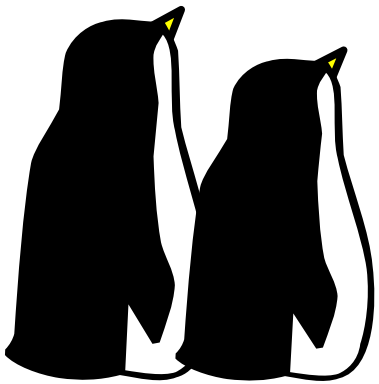
Microsoft Notation Evil

- Microsoft or Hungarian Notation uses a type prefix in front of each variable It's bad because:
- It obscures the variable name.
- It gives the programmer type information of little interest to him.
- The type information is featured more prominently than other information about the variable, like what it does.



Microsoft Notation Evil

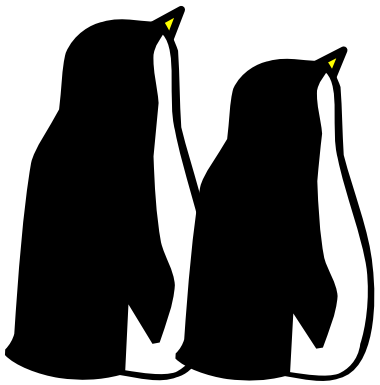
- Microsoft Notation changes depending on:
  - Which API you are using
  - Which version of the API you are using
  - The phase of the moon



Variables

-
- V2 Constants are upper case words separated by underscores. (`THIS_IS_A_CONST`)
-

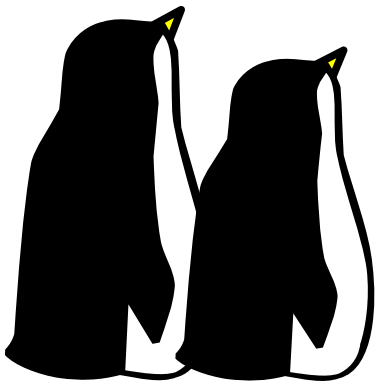
  •Reason: We need some simple convention for everyone. This one works.



Variables

-
- V3 No int declarations. (Use `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32` instead.)
-

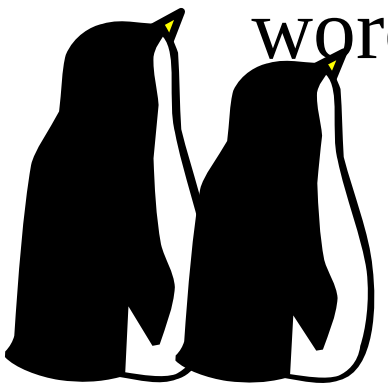
  •Reason: The size of an **int** can vary. If we use these types, we know how big our integers are.
-



Variables

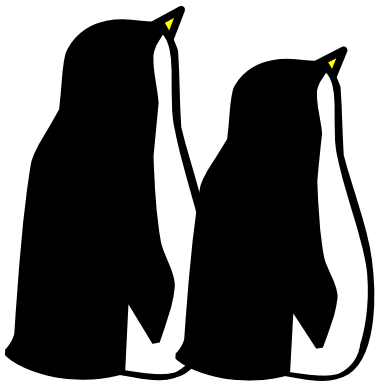
-
- V4 One declaration per line. (Exception: Highly coupled variables, i.e. `width` and `height`.)
-

•Reason: By declaring one variable per line and commenting them, we produce a dictionary gives us simple definitions for all the special words (variables) we use in our program.



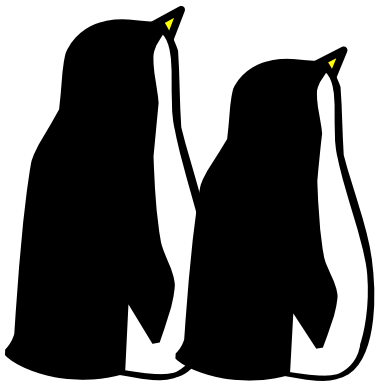
Variables

- V5 All declarations have a comment after them explaining the variable.
- •Reason: Produces a mini-dictionary explaining your words.



Variables

-
- V6 Units are declared when appropriate.
-
- Reason: Avoids unit confusion.



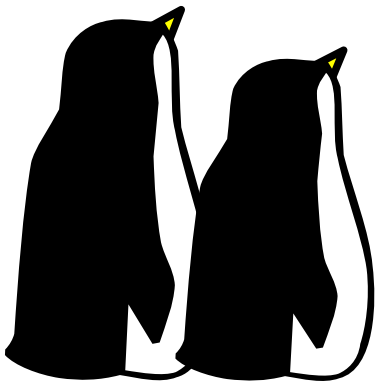
Variables

-

- V7 No hidden variables. That is, a variable defined in an inner block may not have the same name as variable in an outer block.

-

- Reason: Hidden variables cause name confusion.



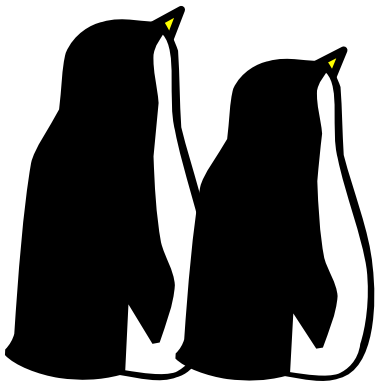
Variables

-

- V8 Never use "O" (Capital O) or "l" (lower case "l") for variable or constant names.

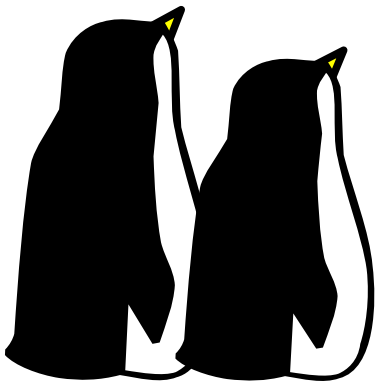
-

- Reason: Avoids confusion



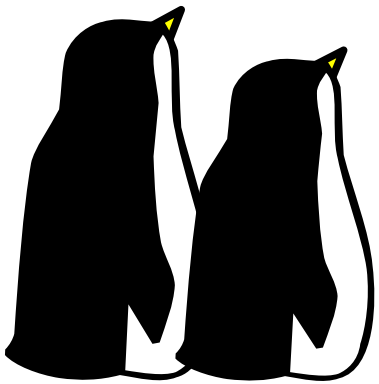
Avoiding Parking Tickets

- To avoid parking tickets, a fellow ordered a special personalized license plate. His three choices were:
  - 1) `OOOOOO` 2) `1I1I1I` 3) `OOOOOOOO`
  -
- He figured that it would be impossible for the police to copy down the license number.



Parking Tickets II

- Unfortunately no DVM clerk could read the license as well. He got a plate which read:
-
  - 000000

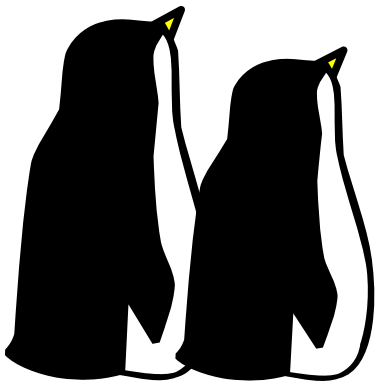


Statements

- S1 No side effects. The operators ++ and -- must be on lines by themselves.

- Reason: Avoids problems. Doing many simple things is simpler than doing one complex thing.

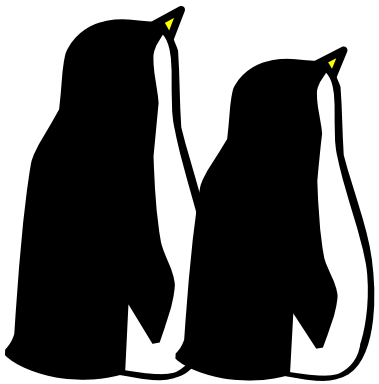
```
// Wrong -- ambiguous
i = array1[i++] + array2[i++];
```



```
// Right
i = array1[i] + array2[i+1];
i += 2;
```

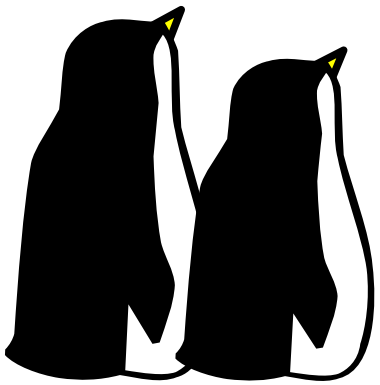
Statements

-
- S2 The assignment operator (=) is not used inside an **if** or other statements.
-
- Reasons: Avoids confusion and helps prevent errors.



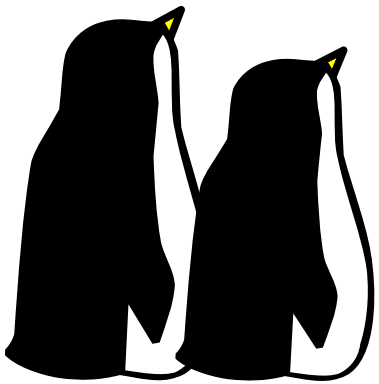
Switch Statements

- S3 All **switch** statements have a default case. (Even if it is `/* Do nothing */`.)
- •Reason: It forces the programmer to consider the default case and tells other programmers what the default case should do.



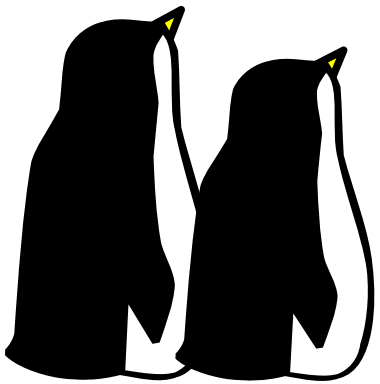
Statements

- S4 The last **case** of a **switch** should end with a **break**.
- •Reason: You may add to the **switch** later and forget to add the **break** to the previous case.



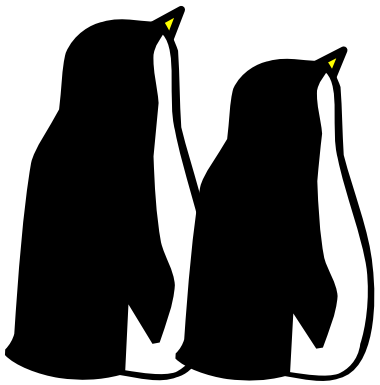
Statements

-
- S5 Empty statements must contain **continue** or `/* Do nothing */`
-
- Reason: The statement ";" makes a lousy empty statement because it's so hard to see.



Switch Example

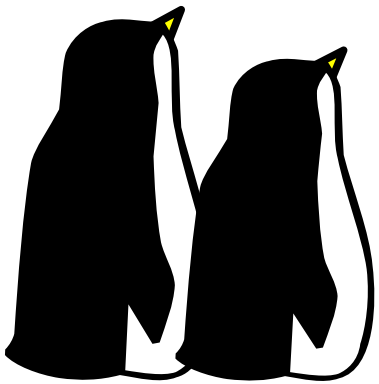
```
switch (cmd) {
    case 'A':
        do_power_down();
        // Fall through
    case 'B':
        do_reset();
        break;
    default:
        // Do nothing
        break;
}
```



Preprocessor

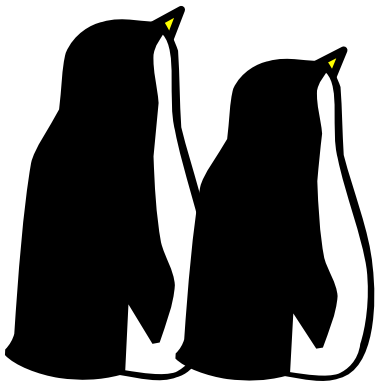
- R1 The backslashes (\) for a multi-line #define are in the same column.
- Reason: It makes it easy to make sure that you put a backslash at the end of each line:

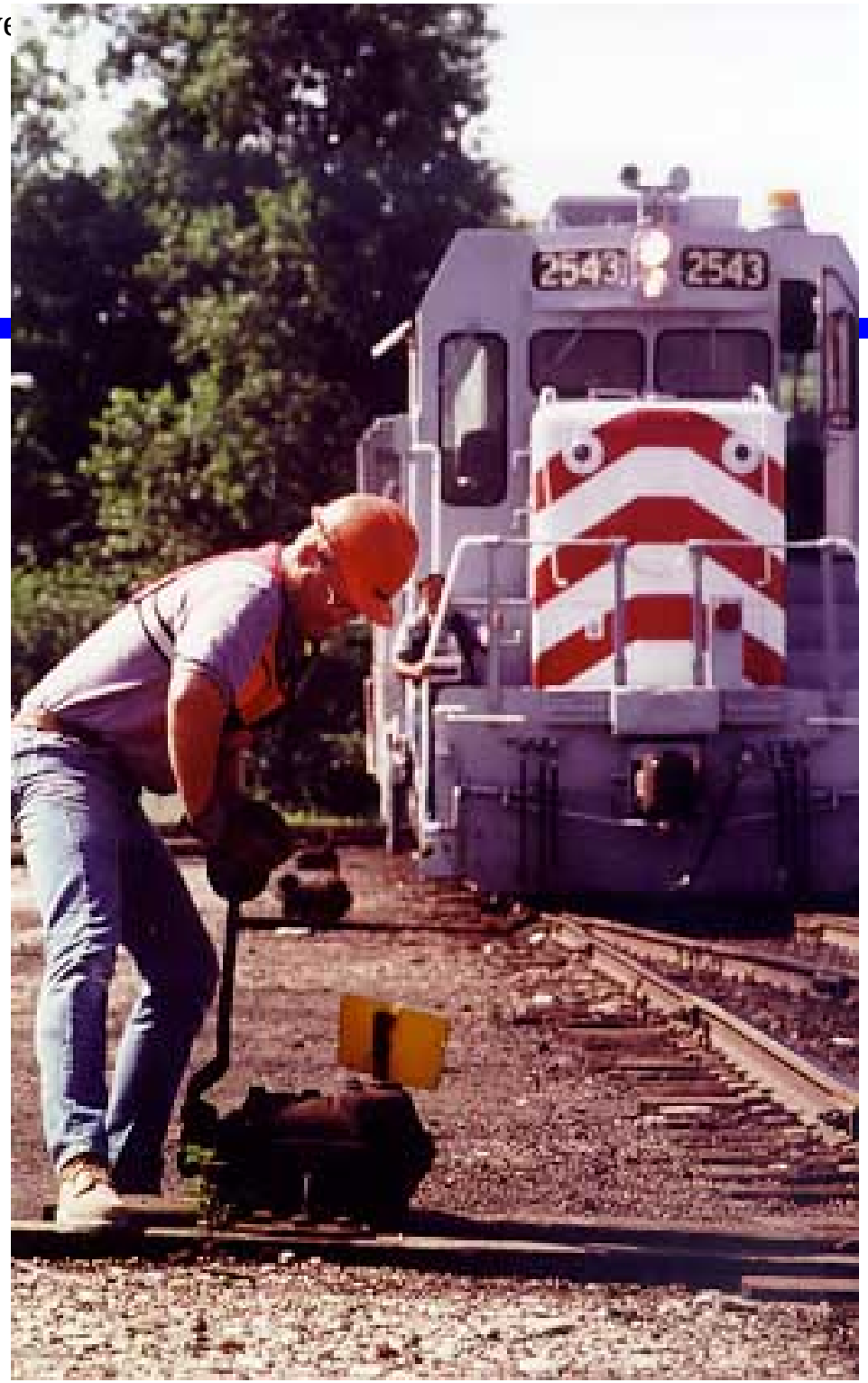
```
#define CHECK_CALL(funct)
    \
      status = funct();
    \
      if (status != 0) {
    \
```



Switches

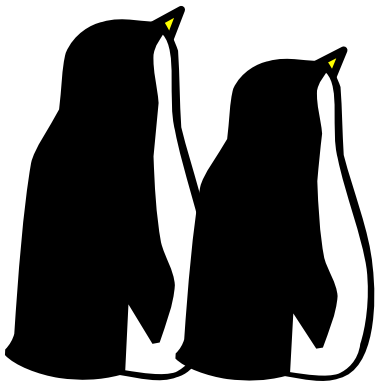
- Rule 25a: Breakmen must remain at least 25 feet away from a switch while a train passes over it.





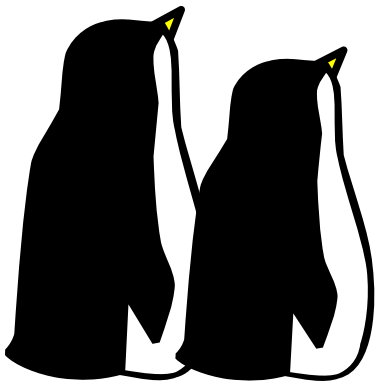
Switches

- Reason:
- 1) Breakmen have been know to throw a switch while a train in going over it. If they're 25 away this is less likely to happen.
- 2) Derailments frequently occur at switches. It's safer to be somewhere else when that happens.



Preprocessor

- R2 Use **typedef** instead of **#define**. to define new types.
- Reason: Less error prone.
  - `#define CHAR_PTR char *`
  - `CHAR_PTR foo, bar;`
  - What is the type of "bar"?



typedef vs. #define

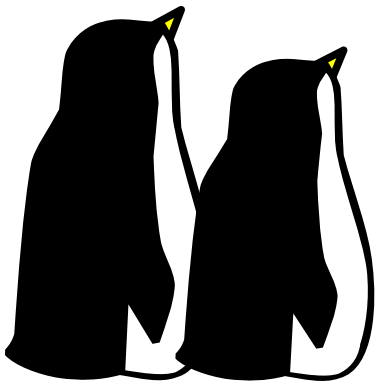
```
#define CHAR_PTR char *

CHAR_PTR foo, bar;
// Result
char *foo, bar;

--- bar of of type char

// Better
typedef char *CHAR_PTR

CHAR_PTR foo, bar
```

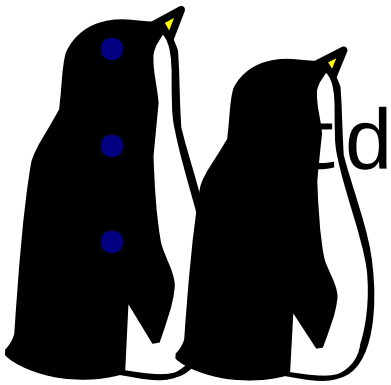


Preprocessor

- R3 Use **const** instead of **#define** to define constants.
- Reason: Less error prone

```
#define SIZE 23 + 34  // Wrong
const int SIZE = 23 + 34; // Right

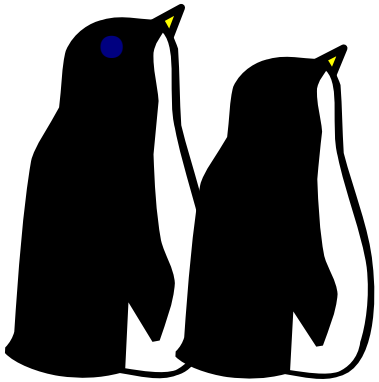
std::cout << SIZE * 2 << '\n';
```



Preprocessor

- R4 Put parenthesis around all parameters used in a parameterized macro definition.
- Reason: Safety again.

```
// Wrong
#define SQUARE(x) (x * x)
i = SQUARE(3 + 4);  // Trouble
```

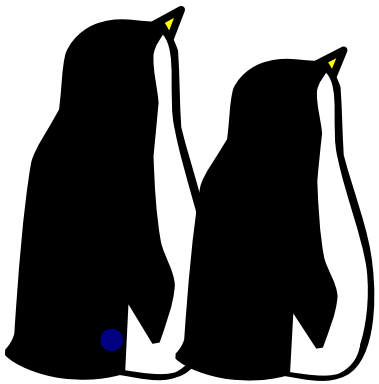


```
// This is better
// But still dangerous
#define SQUARE(x) ((x) * (x))
```

Preprocessor

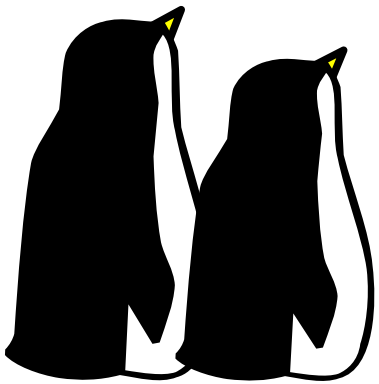
- R5 Use **inline** functions instead of parametrized macros.
- Better:

```
static inline int SQUARE(
    const int x
)
{
    return (x * x);
}
```

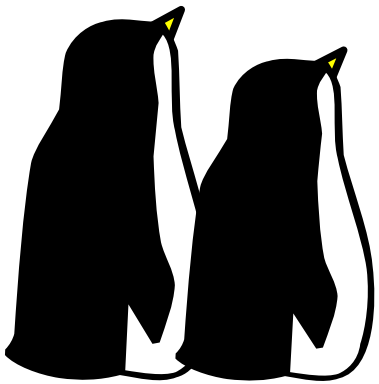


Preprocessor

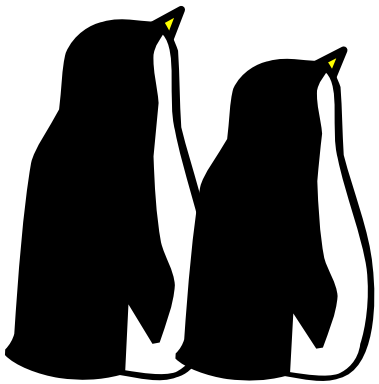
- R6 **#define** should ***never*** be used to redefine C or C++ keywords. Specifically, it they keyword **extern** should never be redefined.
  - Never, never, never!!!
  -
  - `#define extern /* */`
  - `#include "local_vars.h"`
  - `#undef extern`



This will get you shot

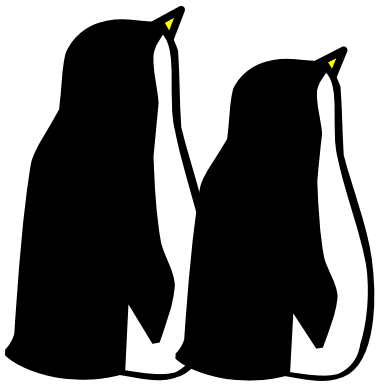


Memory



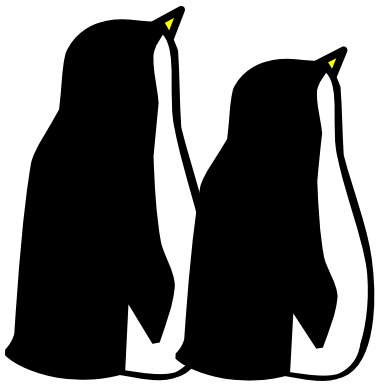
Memory

-
- M1 All **malloc** and related calls have their return value checked against **NULL.**
-
  - Reason: Never assume your program won't run out of memory. It may.
-



Memory

-
- M2 Every **malloc** has a corresponding **free**.
  Every **new** has a corresponding **delete**.
-
- Reason: Avoids memory leaks.
-



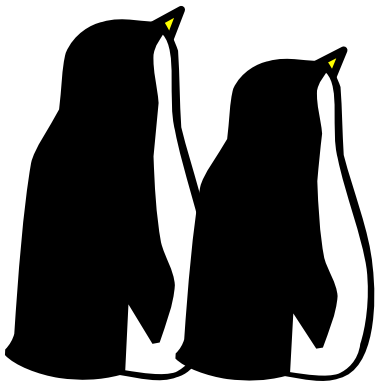
Memory

-

- M3 Every **malloc** has a comment explaining where the **free** is done. The same rule applies for **new** and **delete.**

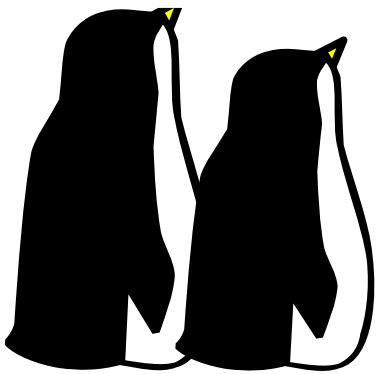
  - Reason: Make the programmer think about where memory is deleted and thus helps avoids memory leaks.

  - Also helps maintenance programmers know where memory to is supposed to be freed



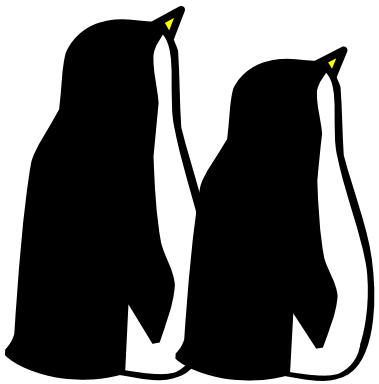
Memory

- M4 Byte streams should have their length checked to prevent overruns. (Example: use **strncpy** instead of **strcpy.**)
-
- Reason: Buffer overflows are a major cause of security problem and other bugs.
-



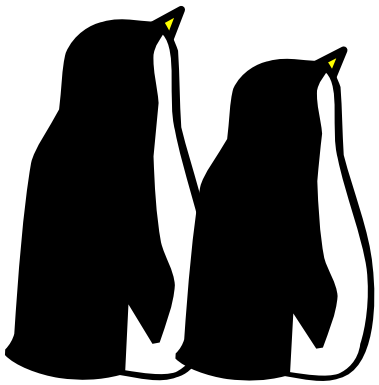
Memory

- M5 Every procedure that allocates memory which the caller must free, must document this fact in the function header.
-
- Reason: Avoids confusion. All allocations should be obvious.



Memory

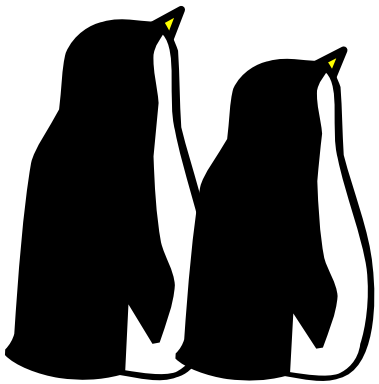
- M6 Every **memcpy** function code should be of the form:
- `memcpy(ptr, src, sizeof(ptr[0]));` whenever possible. Specifically, the third parameter should be sizeof the data pointed to by the first parameter.



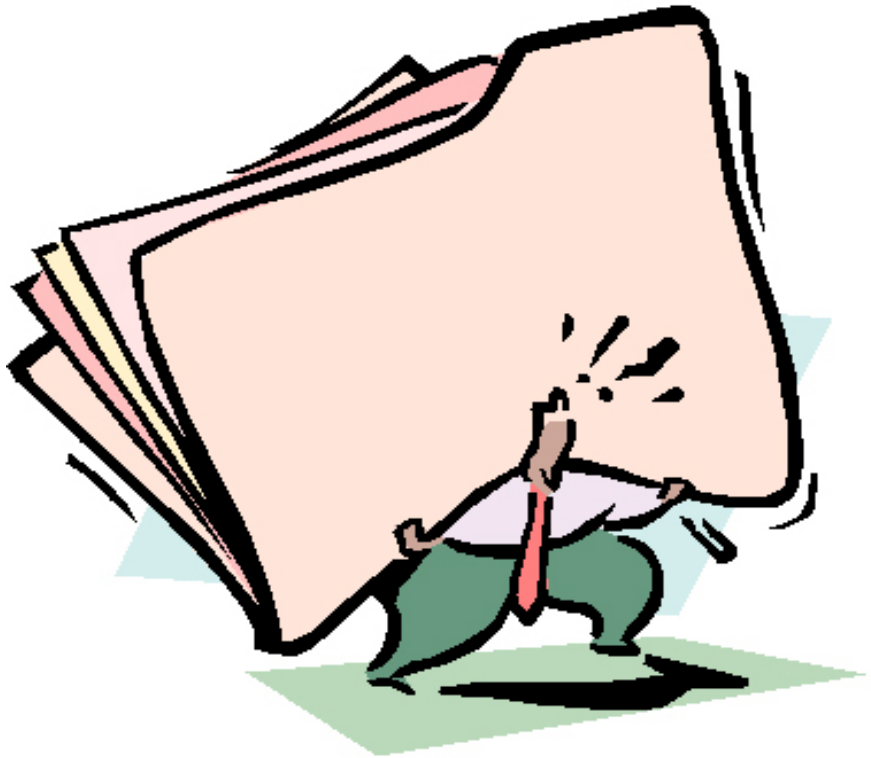
- Reason: Avoids memory overwrites.

Memory

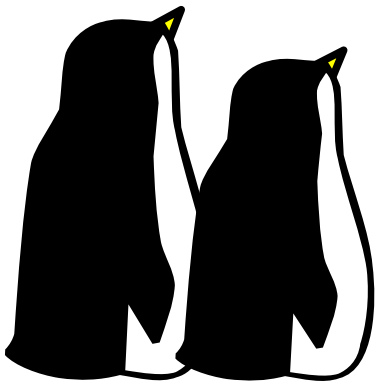
- M7 A similar rule should be followed for **memset**.
-
  - Reason: Avoids memory overwrites.



How to Perform an Inspection

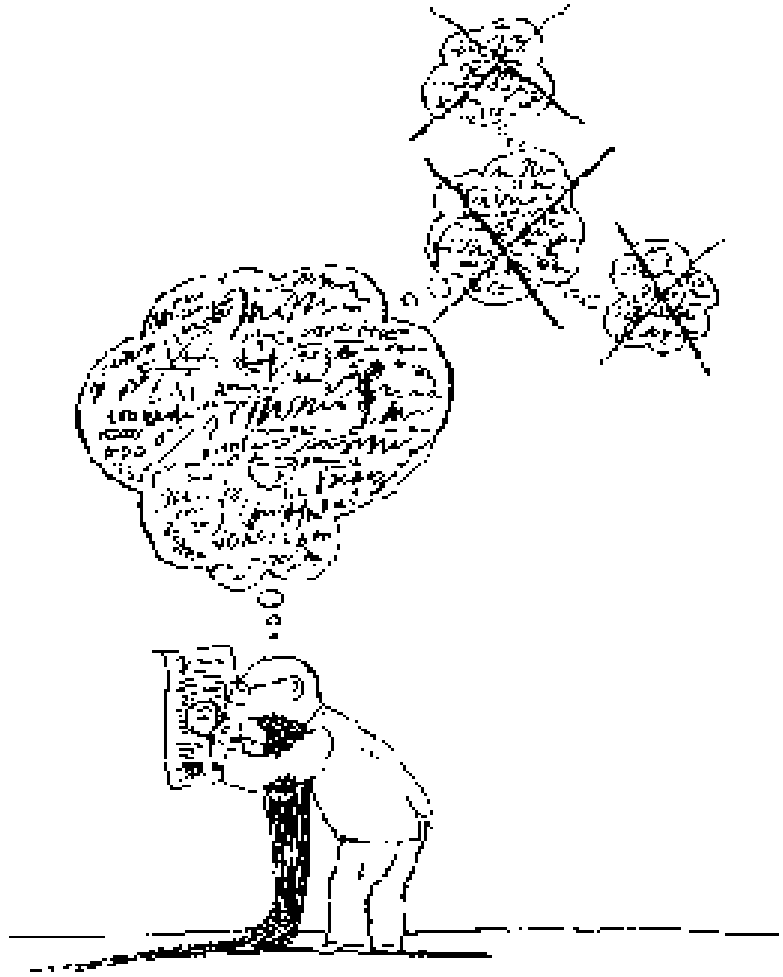


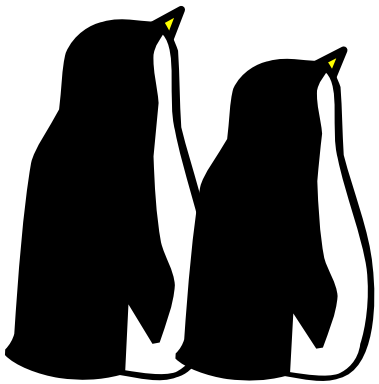
1) Give our code to one to five inspectors for inspectors.



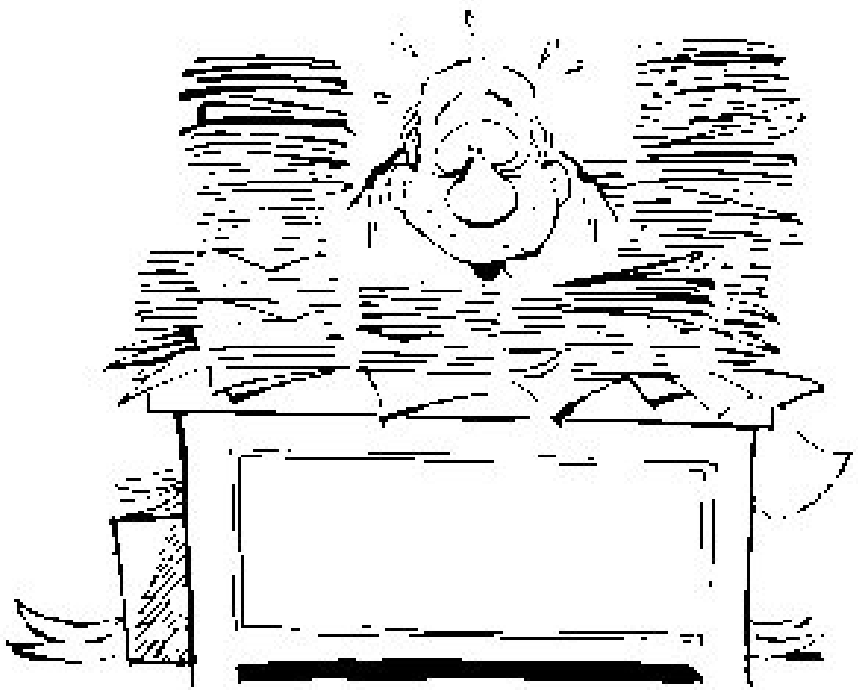
How to Perform an Inspection

2) The inspectors perform the inspection.



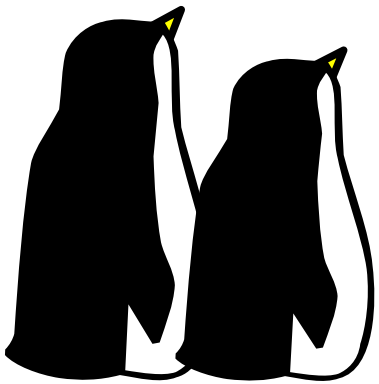


How to Perform an Inspection



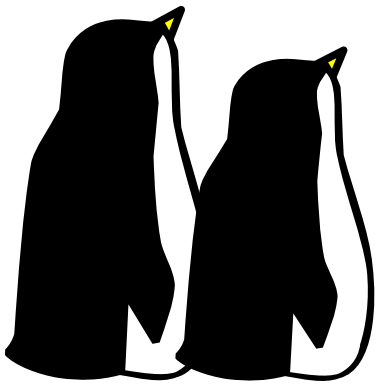
3) The inspectors turn the results in to the programmer.

4) Defects are fixed.



How to Perform an Inspection

5) Metrics are reported.



Inspection Results

Return types defaults

```
is_in(
    exclude_struct *exclude,/* Exclude information */
    const char file_name[]  /* Name of the file to look for */
)
{
    char *cur_file;     /* Current file character */
    char *cur_data;

    while (*cur_file++) {
        switch (*cur_data) {
            case '\n':
                if (*cur_file == '\0')
                    return (1); /* Found it */

                /* Didn't find it, try next variable */
                cur_file = (char *)file_name;
                cur_data++;     /* Skip past the newline */

            case '\0':
                return (0);     /* Didn't find it */
        }
```

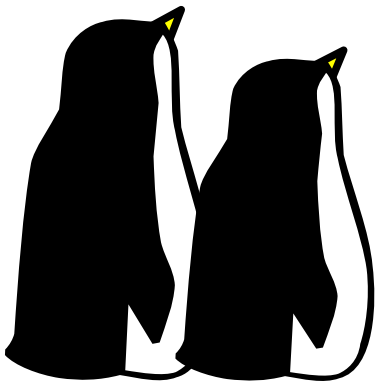
No comment

Case ends incorrectly

No default case

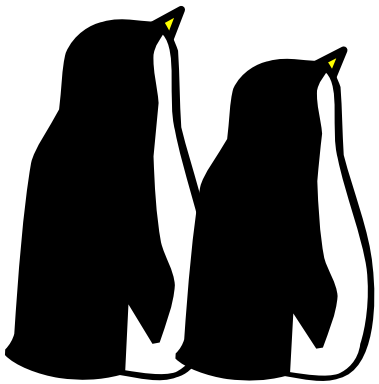
Metrics

Name of Inspector	
Program Inspected	
Start Time	
End Time	
Elapsed Time	
# lines inspected	
# defects found (checklist)	
# non-checklist defects found	

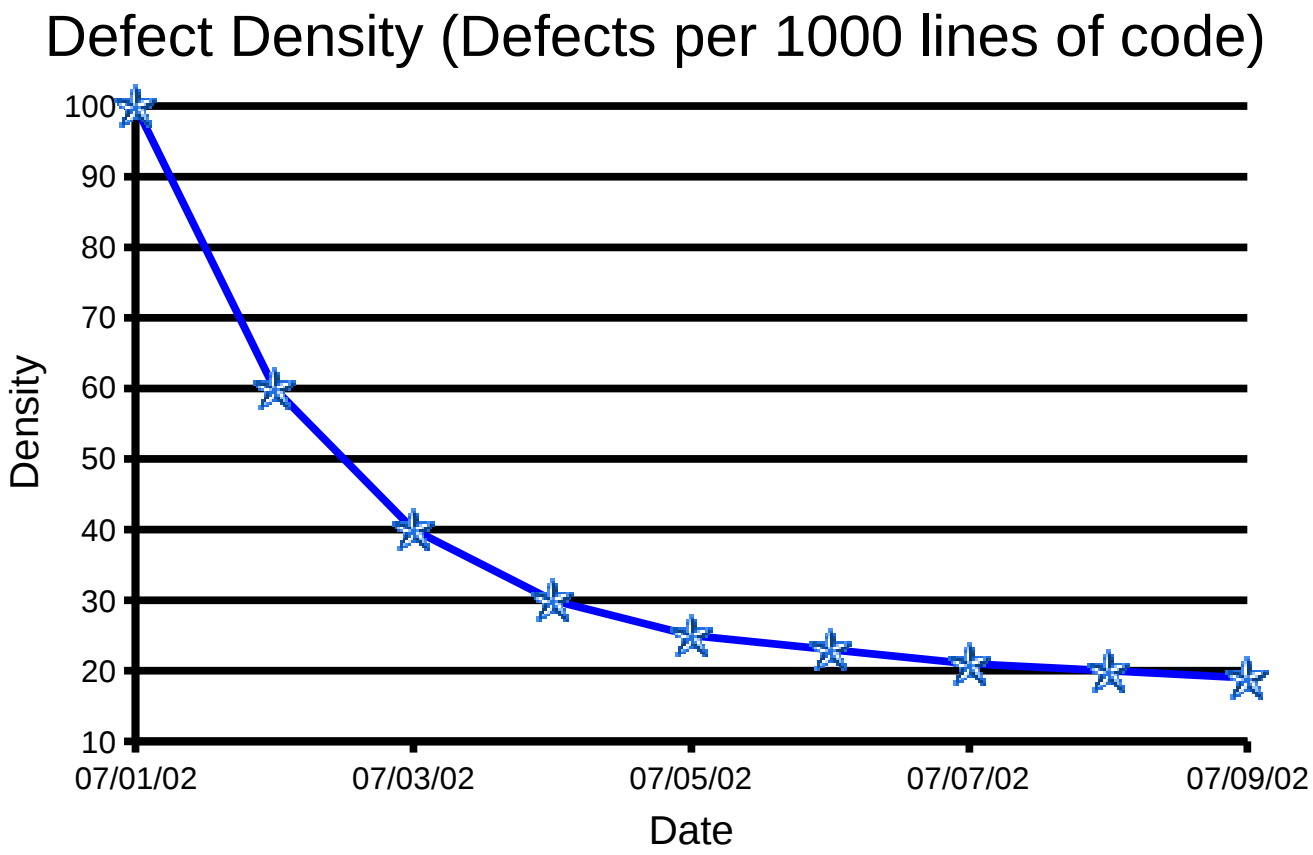


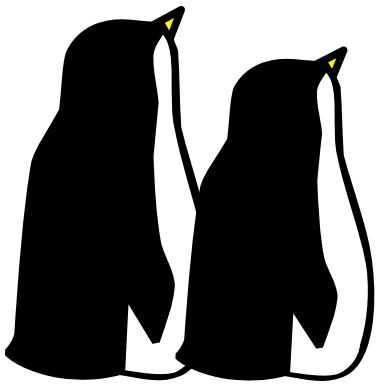
Metrics

Ins.	# lines	Rate	# defects	# defects	Defect density
Name	Ins.	(In /hour)	(C-List)	(Non-CL)	#/K In code
Total					

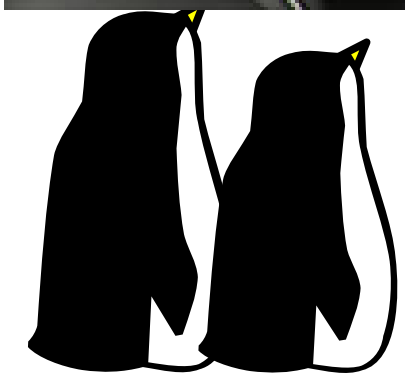


Results of Good Inspection Process





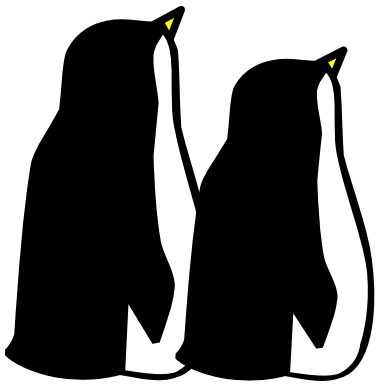
Master Code Review



- The programmer sits down with a master coder and explains his code.
-
- Master Coders know all about:
  - How to design
  - How to code
  - Clean and clear coding

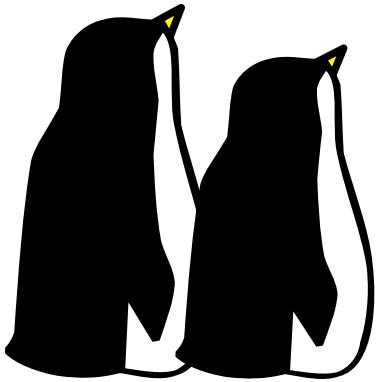
Master Code Review

- Master coders are also know a lot about what *not* to do.



Inspections: Walkthroughs

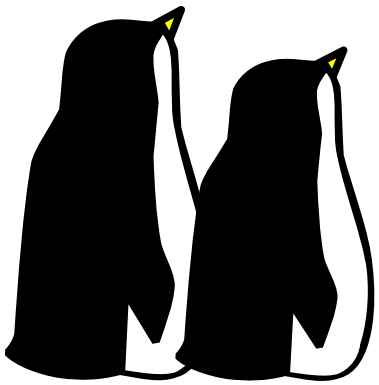
- Where the programmer leads a team of inspectors through his code.



Walkthrough Technique

- 1) Inspectors are selected. (At least one senior programmer should participate.)
- 2) The programmer provides the code to be inspected to the inspectors.
- 3) A meeting is held where the programmer goes through his code. (In any order he wants to.)

4) Defects are recorded by each inspector on their copy of the code.
5) Programmer fixes defects
6) Metric Report Produced



Rules of Thumb

- Inspection meetings should be about 1 hour long. Any shorter and you're not inspecting enough.
- Any longer and people get restless and want to leave.

Conflicts

- ***"Is it a defect or not?"***
- The programmer decides.
  - It's her code.
  - She must make it work.
  - Her review is based on this code.
  - If it doesn't work it's her fault.

Keep on Track

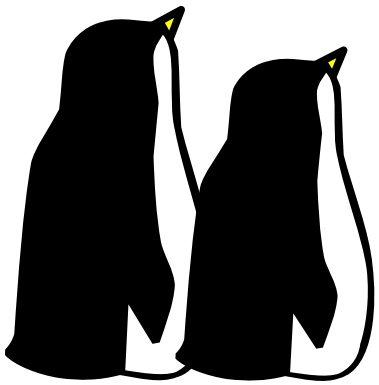
- **The purpose of a code review is to *review code*.**

It's not to:

- Discuss coding technique
- Talk about efficiency or algorithms.
- Discuss other people's code.
- Prove how smart you are.

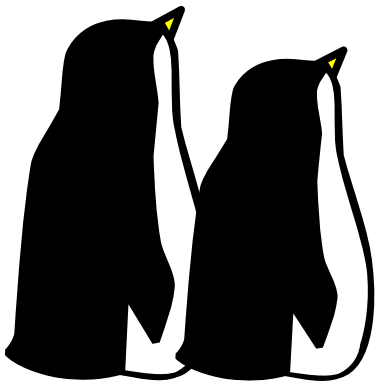
Metrics

- Don't forget the paperwork.
-
- Metrics are important.



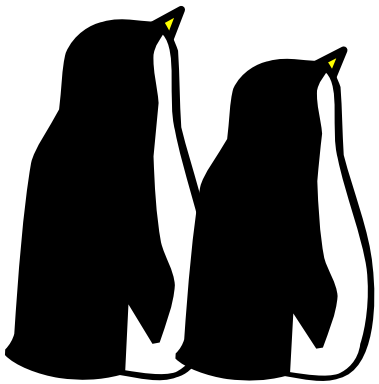
Data Flow Analysis

- Concentrates on the data, not the procedures
- Very good for finding memory leaks
- Useful for handling message systems such a protocol stacks
- Time consuming



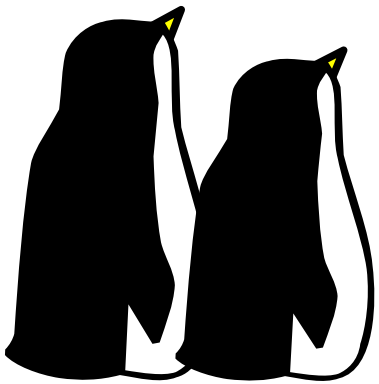
Data Flow Analysis Technique

- This inspection system is similar to a walkthrough except the programmer follows the data.
- Data is traced from where it is allocated to where it is freed.
- Messages are traced from where they are received to where they are disposed of.



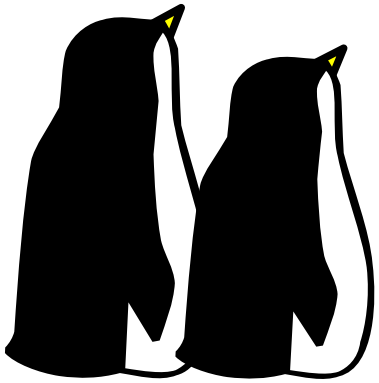
Uses for Data Flow Analysis

- Eliminating memory leaks in programs where memory is critical. (i.e. embedded systems.)
- Protocol Stacks or other message passing systems.



Selling the System

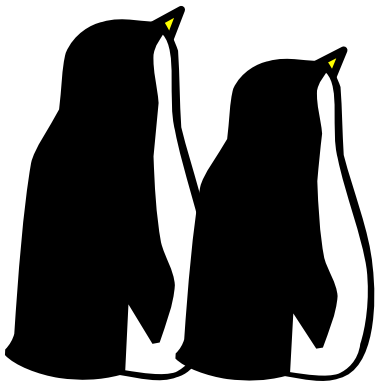
The one thing that sells something to management:



- *Money!*

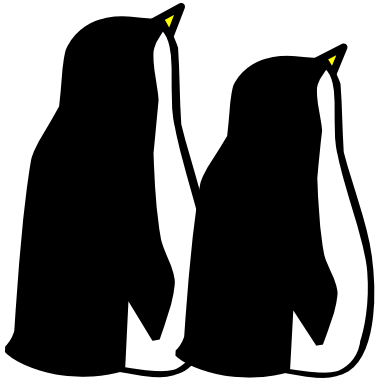
Inspections

- Reduces the number of defects in the software released to the testers.
  - Less time is needed for testing.
  - Less time is spent fixing bugs in the code.
  - Releases can be made faster.
-
-



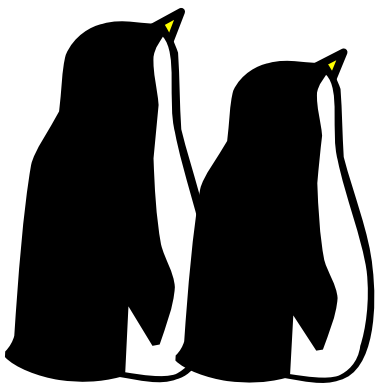
- *Money!*

Inspections

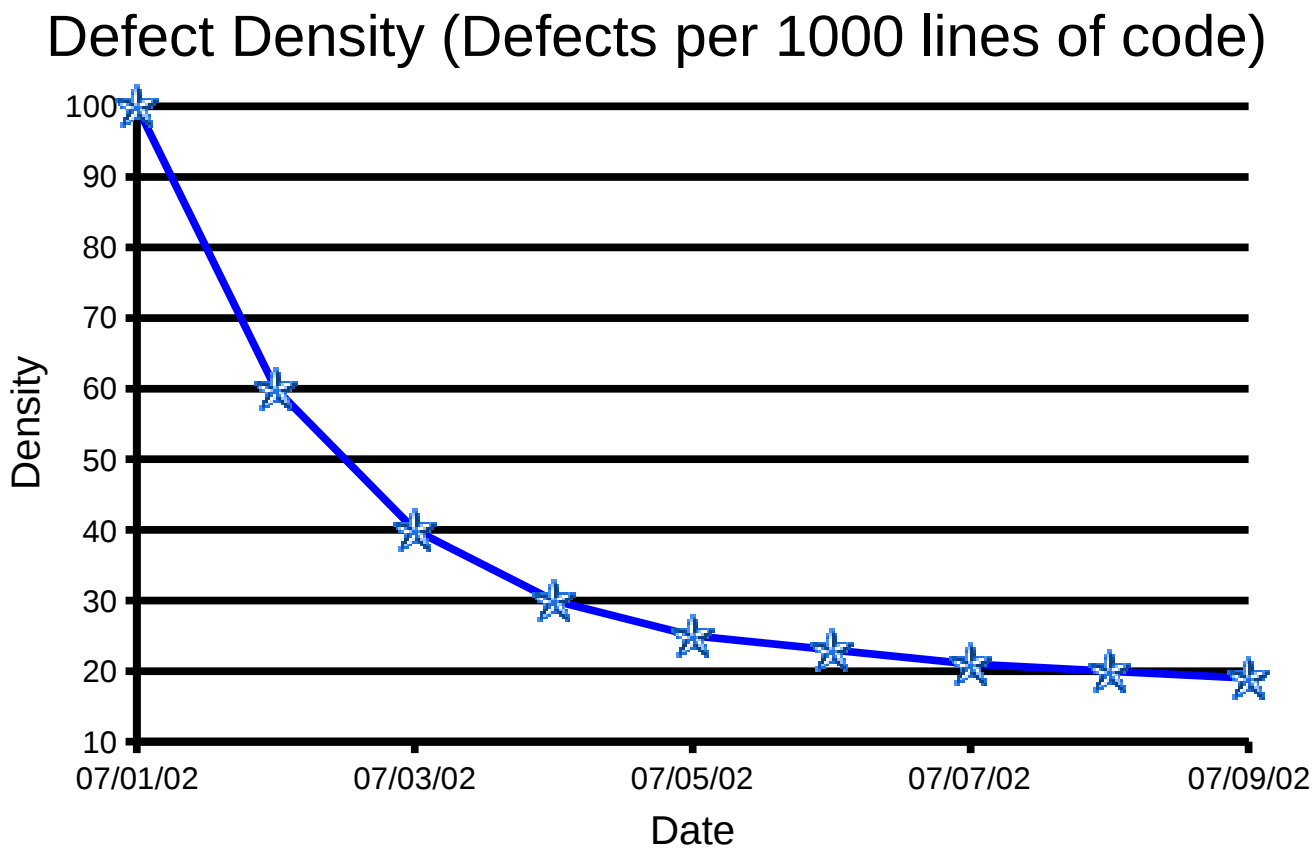
- Reduces the defects made in to code released to the user.
  - Customer service costs are reduced.
  - Customer satisfaction is improved.
  - Recalls and patches are reduced.
- 
- *Money!*

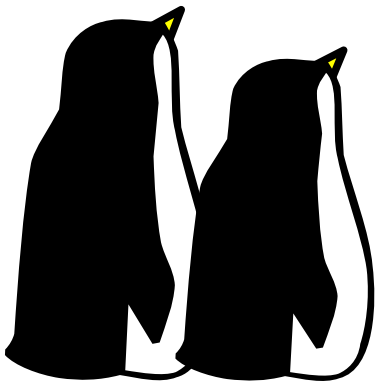
Inspections

- Make better programmers.
  - Errors are eliminated before they start.
  - Result is better code.
  - Result is higher productivity.
- *Money!*



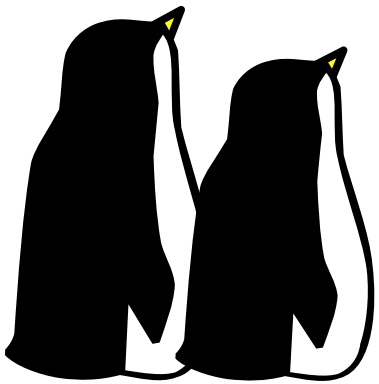
Remember the Metrics





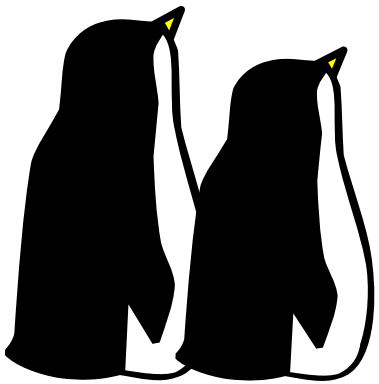
Overcoming Management Objections

- Management: We don't have enough time to do inspections. We have a schedule to keep.
- A1: Inspections *save* time.
- A2: Is our goal to produce software, or to produce *working* software. We need inspections to make sure our software works.



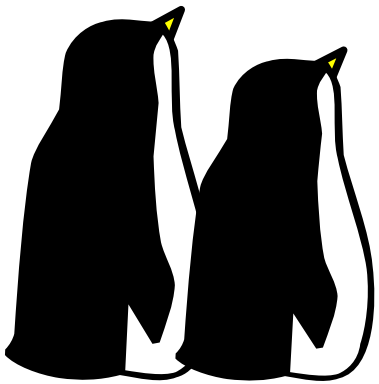
Overcoming Management Objections

- Management: We don't have enough resources to do inspections.
- A: It will take more resources to fix the problems caused by lack of inspections.
-

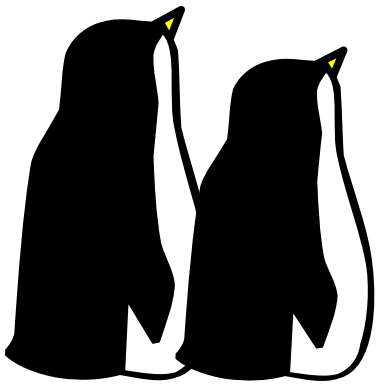


Overcoming Management Objections

- Management: We'll deploy inspection "real soon now".
- A: How many unnecessary errors are going to happen in that time?
-

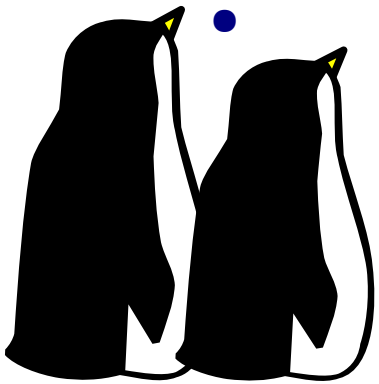


Why Inspection Systems Fail



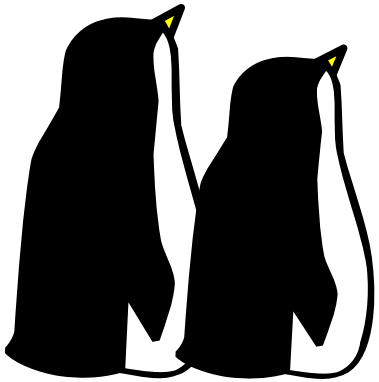
The Perfection Trap

- Trap: We must produce the perfect coding style guide and inspection rules before we can proceed.
- Result: Perfection is never achieved and the deployment of the inspection process endlessly waits for the design process to finish.



Approval Trap

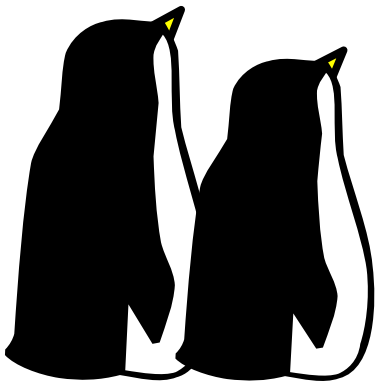
- Approval list:
  - Software lead
  - Assistant software lead
  - IT Manager
  - Project Lead
  - Team Lead
  - Department Head
  - Vice President in charge of software
  - Vice President in charge of forms
  - Third assistant Janitor
  - .....
-



Result: You never get all these people to agree

Onerous Process Trap

- Don't make the process so difficult and onerous that no one wants to do it.



Lack of Metrics

- Without metrics you can't tell how well the process is doing.
-