

# Best Practices in Programming

from B. Kernighan & R. Pike, "The Practice of Programming"

Giovanni Agosta

Piattaforme Software per la Rete – Modulo 2

# Outline

- 1 Principles
- 2 Programming Style
  - Naming Conventions
  - Expressions and Statement
  - Consistency
  - Macros and Comments
- 3 Designing Programs
  - Algorithms and Data Structures
  - Design Principles
- 4 Testing and Debugging
  - Testing

# Motivation

## Bad thing that happen with programs

- Overly complicated data structures
- Too much time spent on finding bugs that should have been obvious
- Excessive use of resources (time, memory)
- Lack of program portability
- Code so difficult to understand you have to rewrite it entirely

These are the results of **programming errors** as much as abnormal program termination or incorrect results!

# Principles

## Keep in sight the basic principles

**Simplicity** Keep programs short and manageable

**Clarity** Keep programs easy to understand for people and machines

**Generality** (Re-)Use and design adaptable solutions

**Automation** Avoid repetitive, error prone tasks by delegating to the machine

# The Techniques

## What do we need to learn, then?

- Programming Style
- Data Structure Construction
- Design and Implementation of Algorithms
- Isolation through Interfaces
- Testing and Debugging
- Programming for Portability
- Programming for Performance
- Tools for Automation

# Naming Conventions

Use descriptive names for globals, short names for locals

## An example of bad conventions

```
for (theElementIndex = 0 ;  
    theElementIndex < numberOfElements ;  
    theElementIndex++)  
    elementArray[theElementIndex] =  
        theElementIndex ;
```

## Should be rewritten as

```
for (i = 0 ; i < nelems ; i++)  
    elem[i] = i ;
```

# Naming Conventions

Be consistent

## Inconsistent and redundant use of names

```
struct __queue {  
    queueElem *queuehead;  
    queueElem *TailOfQueue;  
    int noOfItemsInQ;  
} queue;
```

## Should be restated as

```
struct __queue {  
    queueElem *head;  
    queueElem *tail;  
    int nitems;  
}
```

# Naming Conventions

Use accurate names

Use active names for functions

```
putchar( '\n' );
```

But ambiguity should be avoided

```
//incorrect  
if (checkoctal(c)) ...
```

```
//correct  
if (isoctal(c)) ...
```



# Expressions and Statement Indentation

## Example of bad indentation

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

## Reformatting and restructuring

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

# Expressions and Statement

Write expressions in natural form

## Avoid negations if possible

```
if (!(block_id < actblks) ||  
    !(block_id >= unblocks)) ...
```

## Restructuring to read naturally

```
if ((block_id >= actblks) ||  
    (block_id < unblocks)) ...
```

# Expressions and Statement

Avoid ambiguity by using parentheses

Works, but is hard to read

```
leap_year = y % 4 == 0 && y % 100 != 0 ||  
            y % 400 == 0;
```

Parenthesize to make easier to read

```
leap_year = ((y%4 == 0) && (y%100 != 0)) ||  
            (y%400 == 0);
```

Note that in many cases parentheses are **needed** to specify operator precedence!

# Expressions and Statement

Break up complex expressions

Works, but is totally unreadable

```
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

Restructure to make easier to read

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp
```

# Expressions and Statement

Write for clarity

What does this code do?

```
subkey = subkey >>  
        (bitoff - ((bitoff >> 3) << 3));
```

- Shift bitoff by 3 right, then left → zero the last three bits
- The subtraction gets the three removed bits as results
- The three last bits of bitoff are used to shift subkey

Restructure to make clear and concise

```
subkey >>= bitoff & 0x7;
```

# Expressions and Statement

Be careful with side effects!

The order of execution of side effects is undefined

```
str[i++] = str[i++] = ' ';
```

- Intent: store blank in both spaces
- Effect: depends on when *i* is updated!

Restructure to make unambiguous

```
str[i++] = ' ';  
str[i++] = ' ';
```

# Expressions and Statement

Be careful with side effects!

Argument evaluation happens before the call

```
scanf("%d_%d", &yr, &profit[yr]);
```

- Intent: read values from input and store profit for corresponding yr
- Effect: stores profit at previous value of yr

Correct version

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

# Consistency

Use idioms for consistency

Idioms are conventional ways to express concepts

- The language may offer multiple ways to express a concept (e.g., a loop)
- Certain forms are idiomatic, and should be used instead of less common ones

```
for (i=0; i<N; i++){ ... }
```

```
i=0; while(i<N) { ... ; i+=1; }
```



# Consistency

## Indentation

### Use consistent indentation

- Syntax-driven editing tools may help
- `indent` may also help
- Major software projects mandate their own style!

```
#include <stdio.h>
```

```
int main(int argc, char **argv){  
    /* we aren't checking for missing arg! */  
    char *hello_string=argv[1];  
    printf("%s\n", hello_string); /* print the input */  
    return 0; /* Success */  
}
```

# Consistency

Indentation: GNU style

```
#include <stdio.h>
```

```
int
```

```
main ( int argc , char **argv )
```

```
{  
    /* we aren't checking for missing arg! */  
    char *hello_string = argv [1];  
    printf ("%s\n" , hello_string ); /* print th  
    return 0; /* Success */  
}
```

# Consistency

Indentation: Linux style

```
#include <stdio.h>
```

```
int main( int argc , char **argv )
```

```
{  
    /* we aren't checking for missing arg! */  
    char *hello_string = argv [1];  
    printf( "%s\n" , hello_string ); /* print th  
    return 0; /* Success */  
}
```

# Consistency

Indentation: Kernighan & Ritchie style

```
#include <stdio.h>
```

```
int main( int argc , char **argv )
```

```
{  
    /* we aren't checking for missing arg! */  
    char *hello_string = argv [1];  
    printf( "%s\n" , hello_string ); /* print th  
    return 0; /* Success */  
}
```

# Consistency

Indentation: Berkeley style

```
#include <stdio.h>
```

```
int
```

```
main(int argc, char **argv)
```

```
{
```

```
    /*
```

```
     *we aren't checking for missing arg!
```

```
     */
```

```
    char *hello_string = argv[1];
```

```
    printf("%s\n", hello_string); /* print th
```

```
    return 0; /* Success */
```

```
}
```

# Function Macros

## Beware the semantics!

- Macros work by textual substitution
- Multiple instances of an argument may cause multiple evaluations
- This does not happen with functions
- C99 supports inline functions, use them instead!

# Comments

## Don't do the following:

- Belabor the obvious
- Comment bad code instead of rewriting it
- Contradict the code

## Do the following:

- Point out salient details or large-scale view
- Write code so that meaning is self-evident
- Update comments with the code
- Comment functions and global data

Use `doxygen` for documenting C/C++ code!

# Algorithms and Data Structures

## Choosing an algorithm

### Assess potential algorithms and data structures

- Small-data problems: choose **simple** solutions
- Large-data problems: choose **scalable** solutions

### Choose implementation means

- Use *language features* if possible, *libraries* otherwise
- If you have to design solutions, start from simple ones, then refine for performance



# Algorithms and Data Structures

## Choosing data structures

- A small set of data structures covers most problems
  - **Arrays** fast but no dynamic shrinking/growing
  - **Lists** dynamic shrinking/growing but slow
  - **Trees** combine both, but balancing is required
  - **Hash tables** combine both, but balancing is required
- Specialized data structures might be needed for particular applications

# Designing Programs

## Think first, code later

- Search for *standard* solutions to subproblems
- Choose (tentative) algorithms
- Design the corresponding data structures

## Prototype first, productize later

- Production-quality code takes 10x to 100x more time than prototypes
- Prototyping forces clarification of decisions
- Start simple, but evolve as needed

# Design Issues

## Issues in building components for larger programs

**Interfaces** provide uniform and convenient services

**Information hiding** provide straightforward access to components while hiding implementation details

**Resource management** manage dynamic memory allocation and shared information

**Error handling** detect and report errors

# Library Interfaces

## Hiding implementation details

- Hide details that are irrelevant to the user
- Example: C I/O library, `FILE_*` hides the implementation
- Use *opaque types* if possible
- Avoid global variables

# Library Interfaces

## Select small, orthogonal set of primitives

- Provide just one way to perform each operation
- Provide operations that do not overlap
- Modify the implementation rather than the interface
- If more convenient ways of doing things are desired, use higher level libraries (wrappers)

# Library Interfaces

## Don't reach behind the user's back

- Do not modify global data or input data (except for output parameters)
- E.g., consider `strtok`, which destroys the input string
- A better implementation could work on a copy

## Keep consistency

- Use the same semantics for parameters across the whole set of primitives
- Compare `stdio.h` with `string.h`
- Also, keep consistency with similar libraries and/or libraries used in the same project

# Resource Management

## Allocate and free resources in the same layer

- E.g., a library that allocates data should free it
- Choose a style and keep it
- Write reentrant code: avoid global variables, static local variables

# Error Handling

## Error detection at low level, handling at high level

- Detect error at as low level as possible
- Handle error at high level: let caller function decide on handling
- Library functions should fail gracefully (e.g., return NULL rather than abort)
- In C, use `errno.h` to distinguish between various types of error

## Use exceptions for exceptional behaviour

- C exception handling: `setjmp` and `longjmp`
- Very low level mechanism, use only for truly exceptional behaviour



# User Interfaces

Just because it's not Graphical, it doesn't mean it's not a UI

- Text-based programs also have interfaces
- The goal is to keep them simple
- Also, design the interface to allow both programs and humans to use them

Error reporting and input interfaces

- Provide meaningful error reports
- Identify error site (including program name), reason for failure, hints at how to correct
- Use domain-specific mini-languages for complex input
- If extensive interaction is needed, consider using a scripting front-end

# Overview

## Testing vs Debugging vs Correct by Construction

### What is debugging?

An attempt to find the error in a program that is known to be broken

### What is testing?

A systematic attempt to break a program you think is working

### Limits of testing

- You can only demonstrate the presence of bugs, not their absence
- However, *correct by construction* is unfeasible in most cases

# Overview

## Making Program Correct by Construction

### Generate code programmatically

- E.g., generate assembly programs from high level languages
- Use scripting languages or small ad-hoc languages for specialized tasks

# Testing

When to perform testing?

## Test code as you write it!

On small code fragments:

- Boundary condition testing: check for empty input, overfull input, exactly full input...
- Pre- and post-conditions: check that input and output values stay within the expected value ranges
- Use `assert.h` to check properties
- Defensive programming: handle logically impossible cases, detecting and reporting errors
- Use the error facilities provided by called functions!

# Testing

## Systematic Testing

### Test incrementally, but thoroughly

- Test incrementally, starting from small code units
- Test simple parts or functionalities first
- Once simple functionalities work, check more complex cases
- Know what to expect as the correct result! Use properties of the application domain as much as possible
- Verify conservation properties (check that data structures are not destroyed by mistake, and output is consistent with input)
- Compare independent implementations
- Measure test coverage: check that all code is actually tested (with tools such as Gcov)

# Testing

## Test Automation

### Basics

- *Regression testing*: check that a new version obtains the same results as the previous one
- Create tests that are fully self-contained
- Use scripting languages (bash, PERL, AWK, Python)
- Use system tools: `diff`, `sort`, `grep`, `wc`, `sum`

### Advanced Test Automation

- Large code projects provide specialized automation tools for testing: Litmus (Mozilla), Google Testing Framework (Google), xUnit, DejaGNU (GNU)