JACOBS
UNIVERSITY

"*Better Prevent Than Cure*" - Defensive Programming

Credits:
Fresh Sources Inc.

Instructor: Peter Baumann

email: p.baumann@jacobs-university.de

tel: -3178

office: room 88, Research 1

Cannot find REALITY.SYS.
Universe halted.

Big Code – Lines of Code

Average iPhone app	= 50.000 lines
Hubble Space Telescope	= 2 million lines
Windows 3.1 (1992)	= 2.5 million lines
Control software for US military drone	= 3.5 million lines
Windows NT 3.1 (1993)	= 4.5 million lines
HD DVD Player Xbox	= 4.5 million lines
World of Warcraft Server	= 5.5 million lines
Google Chrome	= 6.5 million lines
Windows NT 4 (1996)	= 11 million lines
MySQL	= 12 million lines
Boing 787 Flight Software	= 14 million lines
F35 Fighter jet	= 23 million lines
Microsoft Office 2013	= 44 million lines
Large Hadron Collider	= 50 million lines
Facebook	= 61 million lines
US Army Future Combat System	= 63 million lines
MacOS X 4.1 Tiger	= 85 million lines
Average high-end car	= 100 million lines
1.3+ million iPhone apps, 1.3+ million Android apps	= 170billion lines

source: http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

Big Data in Industry

- Industry 4.0: integration of production & IT
  - Optimising value chain & life cycle
- Automobiles
  - Networked with co-traffic, traffic lights, ... → PB a day
  - BMW iDrive:
    - *Onboard 40+ sensors, 30+ antennas*
    - *Gbit Ethernet, up to 30 Gb/s*
    - *5G → BMW Cloud*
- Airplanes
  - A380: 1b LoC
  - Per engine: 1 TB / 3 min
    - *LHR → JFK = 640 TB*

[Kristen Nicole]

[Airbus]

Software Crisis

- difficult to write useful & efficient computer programs in the required time
- Reason: rapid increases in computer power, complexity of problems that could be tackled
- Consequences:
  - Projects running over-budget, over-time
  - Software inefficient, of low quality, not meeting requirements
  - Projects unmanageable, code difficult to maintain
  - Software was never delivered

Software Extinction Events

- 1950s: assembler code not manageable
  - symbolic PLs: COBOL, FORTRAN
- 1960s: 100,000s LoC not manageable
  - structured programming [Djikstra et al]:
    - *Bad stmts forbidden; blocks to enter at top & leave at bottom*
  - disentangled code → easier to read + test + maintain; measurable!
- 1980s: multi-millions LoC not manageable
  - object orientation, UML
- 2000s: proliferating Web services not manageable
  - service-oriented architecture: functional building-blocks accessible over standard Internet

Spaghetti Code

foo.h

```
#define BAR(x,y) (x)=2*(y)
#define FOO(x)   BAR(index,x)
```

foo.c

```
#include "foo.h"
int index = 42;

int f()
{       int i;
        for ( i=0; i<10; i++ )
        {       FOO(i);
                weirdStuff(index,i);
        }
}
```

Image: Wikipedia – check it out!

Now some "purist" renames `i` to `index` ...

Software Crisis: Response

- Structured programming
  - Functions, blocks...all is better than `goto`!
  - Avoid spaghetti code
- Object-oriented programming
- Defensive programming
  - Better check twice
    – in particular across interfaces!
  - Runtime checks, safer PLs
- Correctness proofs
- Systematic testing

Image: Wikipedia
– check it out!

Defensive Programming

- Prevention is better than cure, therefore:
- Defensive Programming intends "to ensure the continuing function of a piece of software in spite of unforeseeable usage of said software"
  - [http://en.wikipedia.org/wiki/Defensive_programming]
- Good design yields better product
  - Defending against errors avoids lengthy debugging sessions
- Good design should be *evident* in code
  - Code is executable; comments aren't
  - Key design checkpoints should be checked by your code

Defensive Programming: Example

```c
int risky_programming(char *input){
  char str[1000+1];
  // ...
  strcpy(str, input);
  // ...
}
```

```c
int secure_programming(char *input){
  char str[1000];
  // ...
  strncpy(str, input, sizeof(str));
  str[sizeof(str) - 1] = '\0';
  // ...
}
```

[http://en.wikipedia.org/wiki/Defensive_programming]

JACOBS UNIVERSITY

Invariants

- Conditions that do not vary
  - "Design mileposts" in your code
- Loop invariants
  - True at beginning of each loop iteration (and after termination if all went well)
- Class invariants
  - True before and after each method call
- Method invariants
  - Pre- and post conditions
  - Part of "Design-by-contract"
- …plus plain old invariants

Loop Invariant Example

- Program for computing the factorial of (integer) n:

Credit:
Alden Wright, U of Montana

```
unsigned int factorial( unsigned int n )
{
        unsigned int i = 1, fact = 1;
        while (i != n)
        {
                i++;
                fact *= i;
        }
        return fact;
}
```

Unsafe
– in practice, better use
while (i < n)

- Precondition: n >= 1
- Postcondition: fact == n!

Loop Invariant Example (contd.)

- The loop invariant can be:
  - fact = i!
- Initialization:
  - Before first iteration: i=1, fact=1 => fact=i!
- Maintenance:
  - Let i , fact denote values on previous iteration
  - Assume fact =i‘!, prove fact=i!
  - Proof:

    i = i +1 and fact = fact *i // after loop body
    fact = i !
    fact *i = i ! * i // multiplying both sides by i
    fact = (i-1)! * i
    fact = i!
- Termination:
  - When loop terminates, i = n
  - This plus the loop invariant implies postcondition.
- Precondition necessary!

```
uint factorial( uint n )
{    uint i = 1, uint fact = 1;
     while (i != n)
          i++, fact *= i;
     return fact;
}
```

JACOBS UNIVERSITY

Class Invariants

- All constructors should place their object in a valid state
- All methods should leave their object in a valid state
  - pre-condition and post-condition together should guarantee this
  - Better than just blind coding and testing!
- Example: **`Rational`** class:
  - **`denominator > 0`**
  - **`gcd(num,den) == 1`**

JACOBS UNIVERSITY

Method Invariants

- "Design by Contract"
  - Introduced by a Frenchman working in Switzerland living in California
- Methods are *contracts* with the user
- Users must meet pre-conditions of the method
  - Index in a certain range, for example
- Method guarantees post-conditions

Design by Contract: Example

- Users must meet method's pre-conditions:
  - "s is a string with length between 0 and SMAX-1"
  - "n is an integer between 0 and NMAX"
- drawback:
  frequent "*still all ok?*" checks
  - But simple sequence, no deep "if" nesting

```
int myFunc( char *s, int n )
{
    int result = RC_OK;

    if (s = = NULL)
        result = RC_INPUT_ERROR;
    else if (strlen(s) >= SMAX)
            result = RC_INPUT_ERROR;
    else if (n < 0 || n > NMAX)
        result = RC_INPUT_ERROR;

    if (result = = RC_OK)
    {
        do_whatever_is_to_be_done;
    }

    return result;
}
```

Enforcing Invariants – aka "Error Handling"

JACOBS UNIVERSITY

- Several techniques available, best usage depends…
- assertions = *force-terminate program*
  - For programmer errors that don't depend on end user, non-public member functions
- exceptions = *break flow of control (aka goto)*
  - For pre-conditions on **public** member functions
- return codes = *data-oriented, keep flow of control*
  - Post-conditions are usually a method's output

Assertions

- `assert()` macro
  - around since old C days
- if argument is false:
  - prints expression, file, and line number
  - then calls `abort()`
- Handling:
  - Enabled by default
  - Can turn off with `NDEBUG`:
  - `#define NDEBUG`
    `#include <cassert>`

```
void MyVector::push_back( int x )
{
   if (nextSlot == capacity)
        grow();
   assert( nextSlot < capacity );
   data[ nextSlot++ ] = x;
}
```

- Brute force method
- Never ever use it in a server !!!
  - (*would you like it in your editor?*)

Exceptions

- Interrupt regular flow of control, ripple up calling hierarchy
  - Until matching try/catch embrace
  - Otherwise abort program
- Exceptions are classes!
  - `throw()` instantiates exception object
  - can have parameters
  - catch sensitive per exception type
- Can have multiple `catch()`
  - `catch(...)` sensitive to any exception type

```
try
{
        s = myFunc();
}
catch (Error &e)
{
        // error log, file emergency close, ...
}
```

```
char *myFunc() throw (Error)
{
    char *myPtr = malloc( size );
    if (myPtr == NULL)
        throw new Error(ERR_BAD_ALLOC);
    return myPtr;
}
```

Return Codes

- Methods have a return parameter
  - For otherwise void result, it carries only success information
  - If method has regular result: reserve otherwise unused value
    - *NULL for strings, -1 for int, …*
- It's an interface property -- document clearly!
  - …and check in caller code!
- Strongly recommended: single-return functions
  - use a local result variable!

```
int myFunc( string s, int n )
{
    int result = RC_OK;
    if (s = = NULL)
        result = RC_INPUT_ERROR;
    else if (strlen(s) >= SMAX)
            result = RC_INPUT_ERROR;
    else if (n < 0 || n > NMAX)
        result = RC_INPUT_ERROR;
    if (result = = RC_OK)
    {
        do_whatever_is_to_be_done;
    }
    return result;
}
```

Excursion: Another Real-Life Example

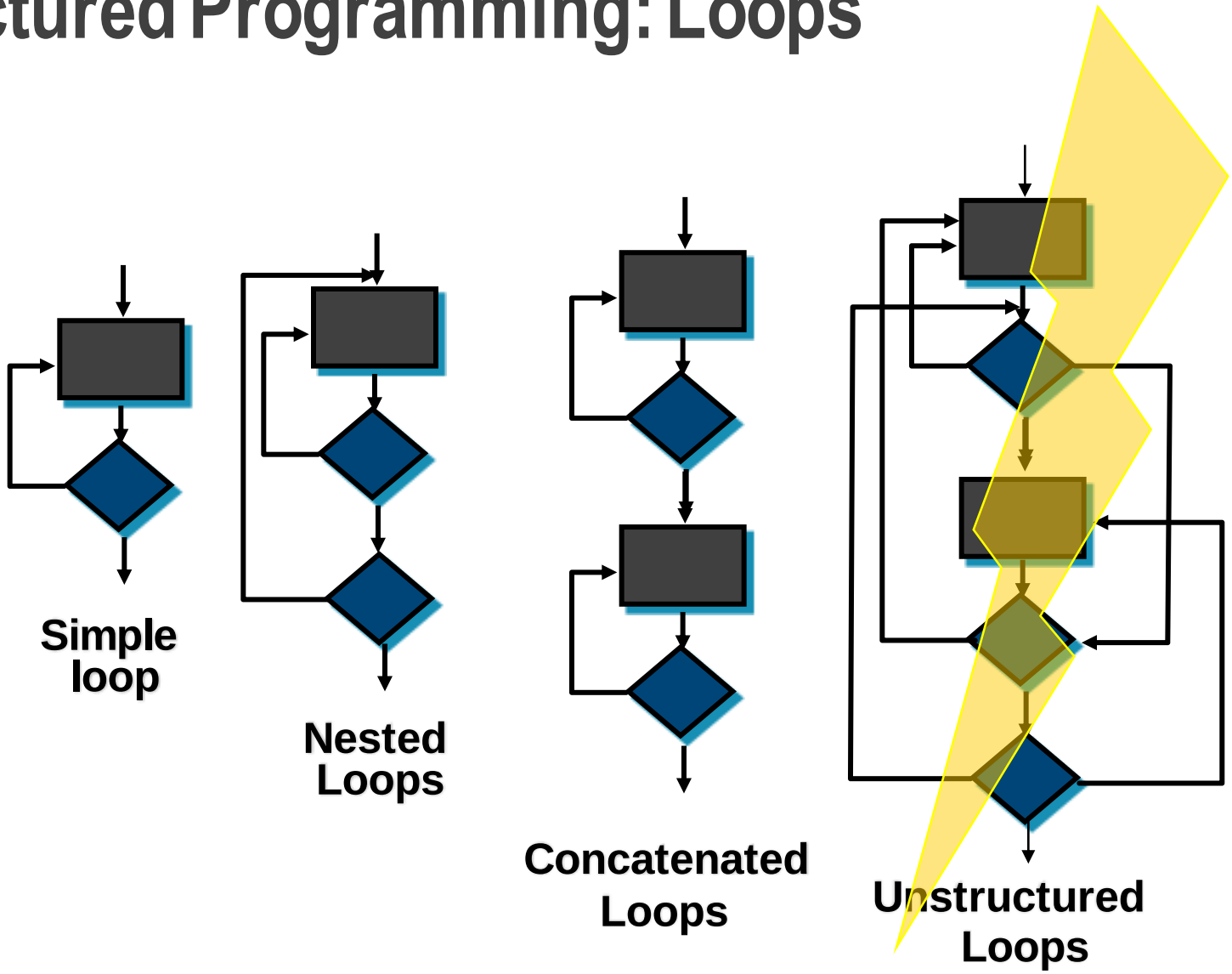
```
for ( count = 0, *templateList = myClass_New ( templateCount, char *);
      *templateList
      && count < templateCount
      && ( ( *templateList)[count] = aux_Duplicate (templates[count] ) );
      count++ );
```

- documenting this takes longer than writing a clear version of the code.
- no error handling at all!
- *How to do better?*

Structured Programming

- Structured programming
  = component-level design technique [Djikstra et al, early 1960s]
  which uses only small set of programming constructs
- Principle: building blocks to enter at top & leave at bottom
  - Good: sequence(";"); condition; repetition
  - Bad: (computed) goto; break; continue; ...
- Advantage: less complex code → easier to read + test + maintain
  - Measurable quality: small complexity (e.g., cyclometric)
  - *...but no dogma:* if it leads to excessive complexity, violating can be ok

Structured Programming: Loops



Apple 'goto fail' Bug [more]

```
static OSStatus SSLVerifySignedServerKeyExchange (
   SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
   uint8 t *signature, UInt16 signatureLen )
{
   OSStatus err;
   . . .
   if (( err = SSLHashSHA1. update(&hashCtx , &serverRandom )) != 0)
      goto fail;
   if (( err = SSLHashSHA1. update(&hashCtx , &signedParams )) != 0)
      goto fail;
      goto fail;
   if (( err = SSLHashSHA1. final(&hashCtx , &hashOut )) != 0)
   goto fail;
   . . .
fail:
   SSLFreeBuffer(&signedHashes );
   SSLFreeBuffer(&hashCtx );
   return err;
}
```

- 2012 – 2014: Apple iOS SSL/TLS library falsely accepted faulty certificates
- Impersonation, man-in-the-middle attacks

use beautifiers!!!

JACOBS UNIVERSITY

Excursion: Expressing Control Flow

- Real-life example!
- Nesting-bad.cc: original code
  - how easy to follow & change?
- Nesting-good.cc: modified code
  - less lines, less columns, less nesting, less getting lost

Code Guides

- Code guide
  = set of rules to which programmers must (should) adhere
  - Within company or project
- Twofold purpose:
  - Have uniform style
    = less surprises = better learning curve for newbies
  - Codify best practice
    = what is acknowledged to be advantageous
- *Varying, individual, maybe not all convincing...yet: stick with it!*
- Let's see an example code guide…

Core Coding Rules

- Reflect before typing!
  - why are you doing what you are doing?
  - what is the best approach?
- Be pedantic
  - As far as ever possible, make it foolproof
  - No monkey tricks
  - Document!
- Design cost-aware
  - is it worth the effort?
  - Is it maintainable?

Tool Support: What Language?

- "Certain programming languages, including C/C++, enable bugs because of how the language was designed" – NIAG SG254
  - memory unsafety fixes: Microsoft 70%, Apple 66%, Android 90%, Chromium 70%
- Safety coding rules constraining allowable language constructs for:
  - Worst case memory & stack usage & analysis; Data coupling & control coupling analysis; Heap fragmentation; Code coverage & test coverage analysis; Object code analysis; memory & thread safety; Portability
- Overcoming C/C++: Ada, Rust, ...
  - Correctness: "if it compiles, then it works"
  - Strong typing semantics as well as the "ownership" and "borrowing" concepts
  - Null pointer safety, thread safety
  - High-level, zero-cost abstractions and language features resulting in clear & concise code

Summary

- Defensive Programming
  = practises to avoid bugs upfront
- Helpful:
  think in terms of assertions / contracts / pre- and postconditions / ...
  - Document and check preconditions for all public interfaces
  - Document postconditions (results, exceptions, ...) *and keep that promises*
- *How to write unmaintainable code:*
  http://mindprod.com/jgloss/unmain.html
- Not addressed here: security
  - Signed config files & executables