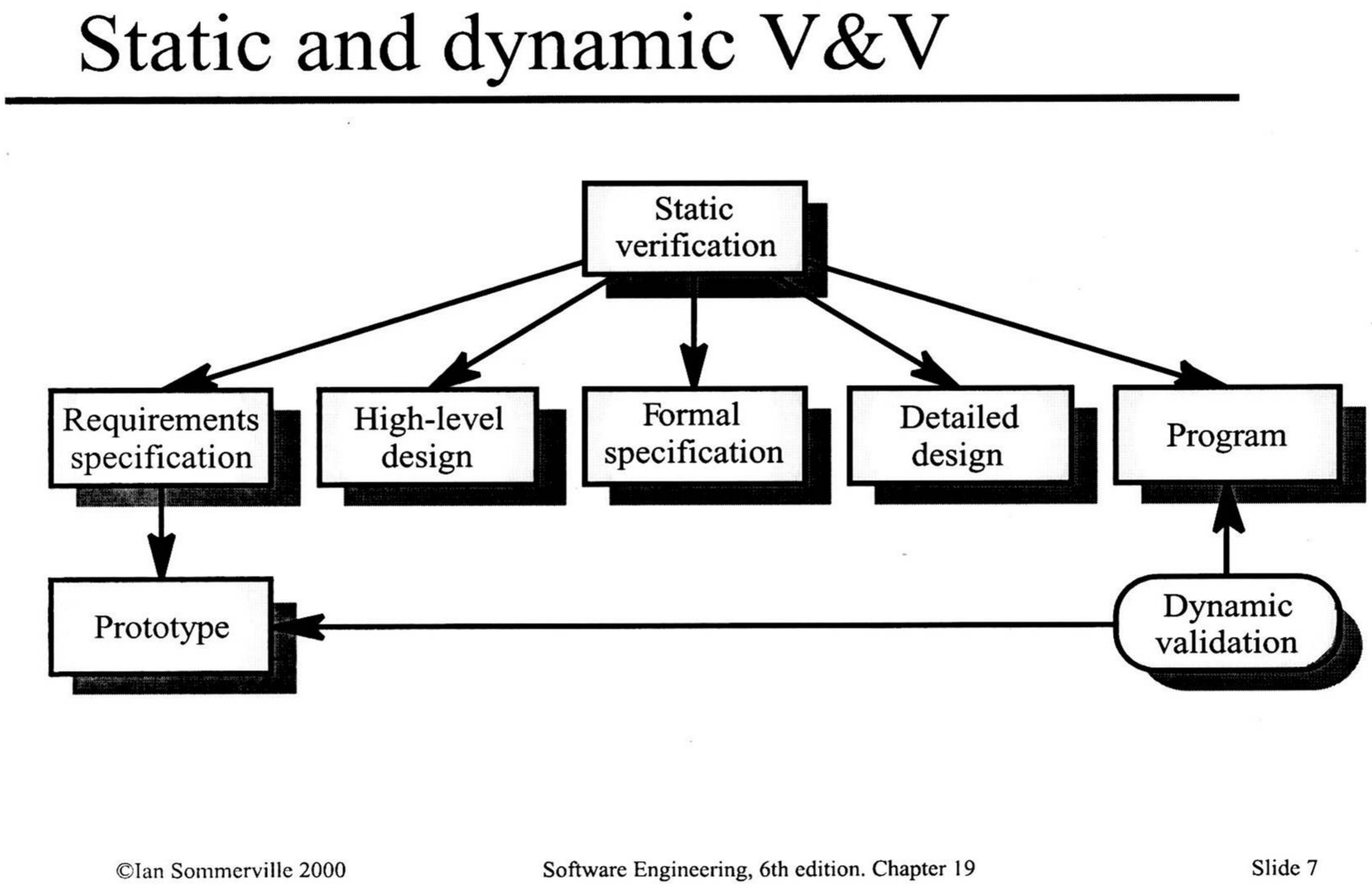
Chap2: Static Analysis : Verification, Validation.

- Approaches to analyse software code. Lab oriented discussion on Software Validation and Verification. Static Analysis in Testing phase. The errors corrected by Static Analysis. Review of the Synopsis report on Static Analysis. Using Splint. Lab Assignment on Splint. [2 hrs]

Verification, Validation & Testing Techs

- Verification and Validation
  - Assuring that a software system confirms its specs and meets a user's needs
  - Which phases does it last to?
- Is verification the same as validation?
  - "Are we building the right product?" /*what are we testing here? */
    - The S/W to confirm to its specifications – check the real product
  - "Are we building the product right?" /*what are we verifying here?*/
    - The software should do what the user really requires – verify design and code

Static and dynamic verification



Verification, Validation

Verification	Validation
1. Verification is a static practice of verifying documents, design, code and program.	1. Validation is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.
4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
5. **Verification** is to check whether the software conforms to specifications.	5. **Validation** is to check whether software meets the customer expectations and requirements.

Verification, Validation…

Verification	Validation
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. Validation is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after **verification**.
10. Are we building the product right?	10. Are we building the right product?
Walkthroughs, Inspection, Code Review, Formal Techniques	Testing, End users code execution etc.

Verification, Validation & Testing Techs (contd)

- What are the roles of verification and validation?
- Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

Objectives

- To review the basics of non-execution based testing.
- To introduce one of the formal software verification technique – static analysis and learn using the tools viz. splint and coverity scan.
- To describe the program inspection process and its role in V&V
- To describe the Cleanroom s/w development process
- To understand Testing and various types of testing

The V & V process

- Two techniques applied:
  - Informal Testing : by programmers themselves
  - Methodical Testing
    - Non-execution based testing
      - S/W inspections – concerns with static verification
      - May be supplement by tool-based (automatic ??) document & code analysis
    - Execution based testing – concerns dynamic verification
      - The system is executed with test data and its operational behaviour is observed

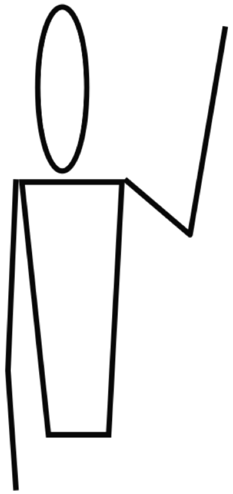
Testing characteristics

- Attributes of a good test
  - all the tests be traceable to customer requirements
  - tests should be planned earlier
  - progress from "in the small" to "the large"
- the Pareto principle applies to testing. What is that ?
- exhaustive testing is not possible
- should be conducted by some third player.

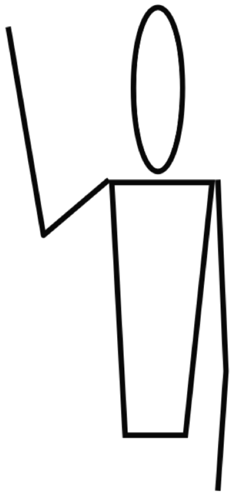
Testability (by James Bach)

- operability
  - *it operates cleanly*
- observability
  - *the results are easy to see*
- controllability
  - *processing can be controlled*
- decomposability
  - *testing can be targeted*
- simplicity
  - *no complex architecture and logic*
- stability
  - *few changes are requested during testing*

Who tests the software ?



developer



independent tester

developer	independent tester
Understands the system	Must learn about the system,
but, will test "gently"	but, will attempt to break it
and, is driven by "delivery"	and, is driven by quality

Non-Execution based testing

- Walkthroughs AND Inspections
- Inspections and testing are complementary and not opposing verification techniques
- Inspections check conformance with a specification but not conformance with the customer's real reqs.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Walkthroughs

- member – senior technical staff
- chaired normally by one from SQA.
- modes of operation :
  - document driven
  - participant driven

- no checklist
- no formalization of faults

Differentiate between Walkthroughs and Inspections

Software inspections

- goes beyond walkthroughs
- five specific steps :
  - overview of the document
  - preparation
  - actual inspection
  - rework
  - follow-up

Software inspections (contd)

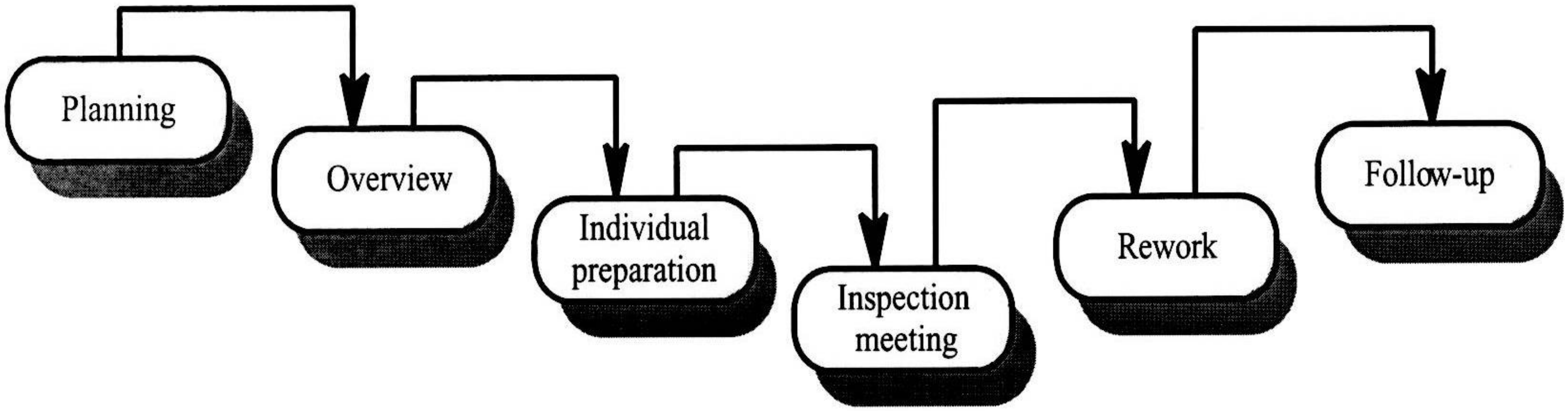
- Involve people examining the source representation with the aim of discovering anomalies and defects
- Specifically a person
  - from SQA — Author
  - from design team — Inspector
  - from coding team — Reader
  - from testing team — Moderator
  - Client Scribe — Chief Moderator
- Do not require execution of a system so may be used before implementation
- Very effective technique for discovering errors

- in a typical 6000-line business DP application, 93% faults found were during inspections

Inspection pre-conditions

- a precise specification must be available
- familiarity with the organization standards
- syntactically correct code must be available
- an error checklist should be prepared
- Management - aware of increase in costs and must not use inspections for staff appraisal

The inspection process



Inspection success

- Many different defects may be discovered in a single inspection.
- In testing, one defect, may mask another.
- What is the problem caused due to this ?
  - Cascaded defects - repeated executions
- reuses domain and programming knowledge so reviewers are likely to have seen the types of errors
- intended explicitly for defect DETECTION ONLY.

Inspection checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The 'weaker' the type checking, the larger the checklist
- Examples: Initialization, Constant naming, loop termination, array bounds, etc.

Inspection rate

- 500 statements/hour during overview
- 125 source statement/hour during individual preparation
- 90-125 statements/hour can be inspected
- Inspection is therefore an expensive process
- Inspecting 500 lines costs about 40 man/hours effort = £28008

Inspection success

- Defects which could be detected are
  - logical errors, violation of portability or maintainability issues
  - anomalies in the code
    - an uninitialised variable or non-compliance with standards
    - memory leaks
    - null-pointer assignments
    - are all input variables used?
    - are all output variables assigned a value before they are output?
    - for each conditional statement, is the condition correct?
    - is each loop certain to terminate?
    - are compound statements correctly bracketed?
- Would these be flagged by a compiler ?
- If 93% faults found were during inspections, is it worth to do it manually ? Can it be automated?

Inspection checks

Fault class	Inspection check
Data faults	Are all program variables initialised before their valuesare used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage mngt faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception mngt faults	Have all possible error conditions been taken into account?

Automated static analysis

- Static analyzers are software tools for source text processing
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team
- Very effective as an aid to inspections. A supplement to but not a replacement for inspections

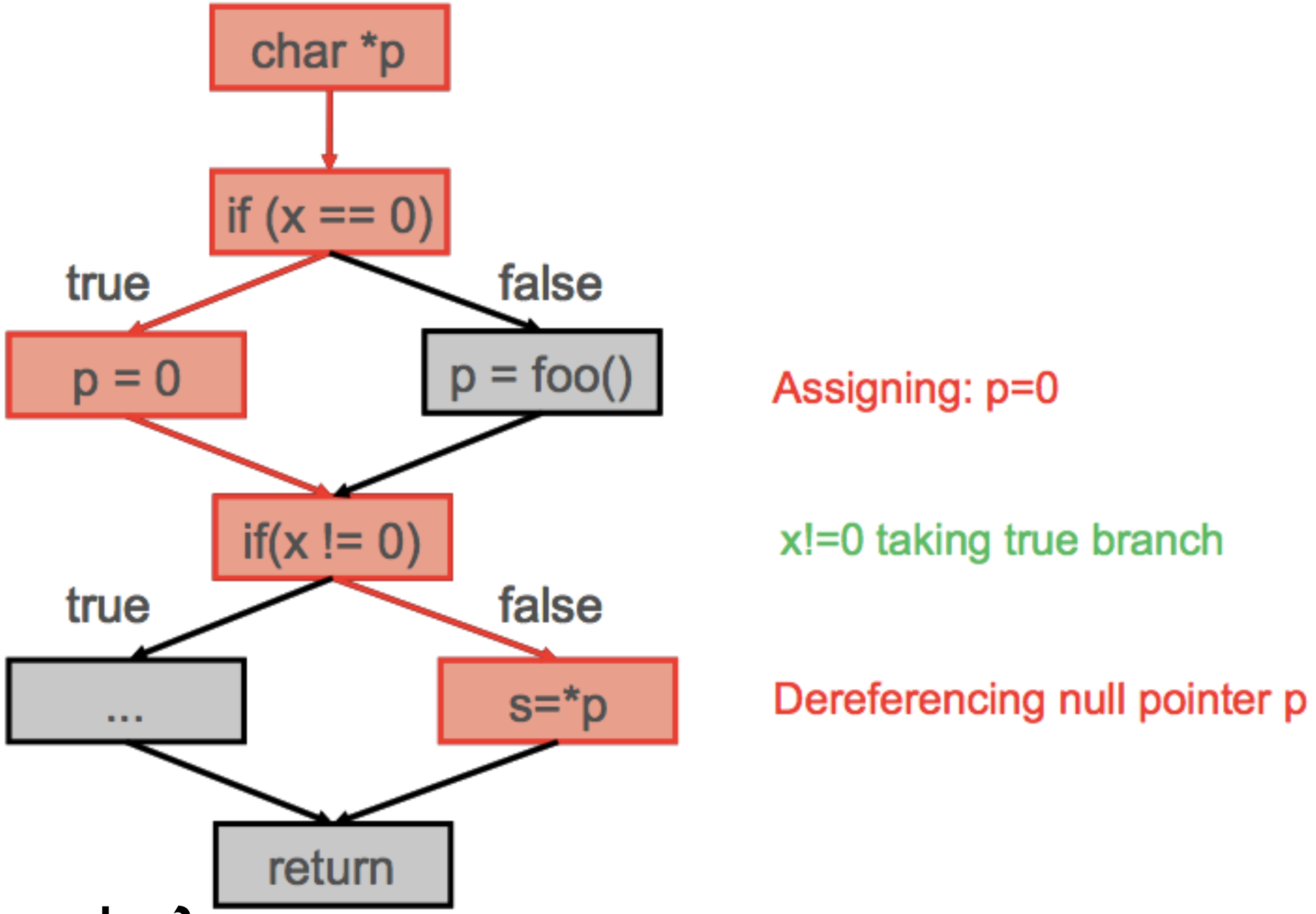
Static Analysis - an illustration

```
char *p;
if (x==0)
    p = 0;
else
    p = foo();
if (x !=0)
     s=*p;
else
     …;
return;
```

- What are the issues with this code ?
- Would gcc flag any errors ?

Static Analysis - an illustration

```
char *p;
if (x==0)
     p = 0;
else
     p = foo();
if (x !=0)
      s=*p;
else
      …;
return;
```



- What are the issues with this code ?
- Would gcc flag any errors ?

Illustration 3:Splint examples

```c
#include <stdio.h>
#include <stdlib.h>
#define  MAXTMP 80
void static foo(void)
{
    char *tmp;
    tmp = (char *) malloc(MAXTMP);
    *tmp = 'X’;
    free(tmp);
}
int main()
{
    foo();
    return 0;
}
```

First compile the code with gcc. Note the errors flagged, if any.

Now, run the code with splint as follows:
$ splint TestDeref.c

Note the errors flagged….compare with the output of gcc.

Illustration 3:Splint examples…

```
char firstChar1(char *s)
{
        return *s;
}
char firstChar2(char *s)
{
        if (s==NULL) return '\0';
    return *s;
}
```

```
char firstChar1(/*@null@*/ char *s)
{
        return *s;
// Test4NullPointers.c:3:10: Dereference of possibly null pointer s: *s. Since
function argument is delcared with "/*@null@*"; s can be NULL and we are
returning possible null value without declaring in the return type of function
declaration.
}
char firstChar2(/*@null@*/ char *s)
{
        if (s==NULL) return '\0';
// No error here since we are checking for possibility of s being NULL.
    return *s;
}
```

Illustration 3:Splint examples…

```
/* Include the files <stdio.h> <string.h> <stdlib.h>. <stddef.h>
void static updateEnv(char *str, size_t size) /*@requires maxSet(str) >= (size-1);@*/
{
char *tmp;
     tmp = getenv("HOME");
     if (tmp != NULL) {
             strncpy(str,tmp,size-1);
             str[size -1] = '\0';
  } }
int main() {
     char *str;
         size_t size;
     str = "Hello World";
     size = strlen(str);
     updateEnv(str, (size-1));
     printf("\nThe Environment variable copied\n");
     return 0;
}
```

/*compile this code with gcc - gives no errors. Then to check with splint - to prevent other trivial errors cropping up, compile splint with +bounds -usedef -exportlocal options i.e.
$splint +bounds -usedef -exportlocal Test2.c */

Another illustration…

```c
#include <stdlib.h>

int process(char*, char*, char*, int);

int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
        if(names != NULL) free(selection);
        if(namesbuf != NULL) free(namesbuf);
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

- What are the issues with this code ?

First defect: Memory leak

```
#include <stdlib.h>

int process(char*, char*, char*, int);

int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

CID 68629: Resource leak (RESOURCE_LEAK) [select defect]
!   names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
CID 68630: Double free (USE_AFTER_FREE) [select defect]
!       if(names != NULL) free(selection);
        if(namesbuf != NULL) free(namesbuf);
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

•

Second defect: Double free

Allocated “names”

Checking for allocation failures for all variables

Freeing “selection” instead of “names”

“names” leaked

```
int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

CID 68629: Resource leak (RESOURCE_LEAK)
Calling allocation function "malloc".
Assigning: "names" = storage returned from "malloc(size)".
    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

At conditional (1): "names == NULL" taking the false branch.
At conditional (2): "namesbuf == NULL" taking the false branch.
At conditional (3): "selection == NULL" taking the true branch.
    if(names == NULL || namesbuf == NULL || selection == NULL) {
CID 68630: Double free (USE_AFTER_FREE) [select defect]
At conditional (4): "names != NULL" taking the true branch.
        if(names != NULL) free(selection);
At conditional (5): "namesbuf != NULL" taking the true branch.
        if(namesbuf != NULL) free(namesbuf);
At conditional (6): "selection != NULL" taking the false branch.
        if(selection != NULL) free(selection);
Variable "names" going out of scope leaks the storage it points to.
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

C/C++ errors that typical tools can detect

Memory-corruptions • Out-of-bounds access • String length miscalculations • Copying to destination buffers too small • Overflowed pointer write • Negative array index write • Allocation size error	Memory-illegal access • Incorrect delete operator • Overflowed pointer read • Out-of-bounds read • Returning pointer to local variable • Negative array index read • Use/read pointer after free
Integer handling issues • Improper use of negative value • Unintended sign extension	Improper Use of APIs • Insecure chroot • Using invalid iterator printf() argument mismatch
Resource Leaks • Memory leaks • Resource leak in object • Incomplete delete • Microsoft COM BSTR memory leaks	Concurrency Issues • Deadlocks • Race conditions • Blocking call misuse

C/C++ errors that typical tools can detect

Uninitialized variables • Missing return statement • Uninitialized pointer/scalar/array read/write • Uninitialized data member in class or structure	Control flow issues • Logically dead code • Missing break in switch • Structurally dead code
Error handling issues • Unchecked return value • Uncaught exception • Invalid use of negative variables	Program hangs • Infinite loop • Double lock or missing unlock • Negative loop bound • Thread deadlock • sleep() while holding a lock
Insecure data handling • Integer overflow • Loop bound by untrusted source • Write/read array/pointer with untrusted value • Format string with untrusted source	Null pointer differences • Dereference after a null check • Dereference a null return value • Dereference before a null check

- These are detected by Coverity Scan from Synopsis.

C/C++ errors that typical tools can detect

Code Maintainability Issues\ • Multiple return statements • Unused pointer value	Performance inefficiencies • Big parameter passed by value • Large stack use
Security best practices violations • Possible buffer overflow • Copy into a fixed size buffer • Calling risky function • Use of insecure temporary file • Time of check different than time of use • User pointer dereference	

- These are detected by Coverity Scan from Synopsis.

C++/Java errors that typical tools can detect

Resource Leaks • Database connection leaks • Resource leaks • Socket & Stream leaks • Volatile not atomically updated	API usage errors • Using invalid iterator • Unmodifiable collection error • Use of freed resources
Concurrent data access violations • Values not atomically updated • Double checked locking • Data race condition	Performance inefficiencies • Use of inefficient method • String concatenation in loop • Unnecessary synchronization
Code maintainability issues • Calling a deprecated method • Explicit garbage collection • Static set in non-static method	Control flow issues • Return inside finally block • Missing break in switch pointer
Class hierarchy inconsistencies • Failure to call super.clone() or supler.finalize() • Missing call to super class • Virtual method in constructor	Null pointer dereferences • Dereference after null check • Dereference before null check • Dereference null return value

- These are detected by Coverity Scan from Synopsis.

C++/Java errors that typical tools can detect

Error handling issues • Unchecked return value	Program hangs • Thread deadlock

- These are detected by Coverity Scan from Synopsis.

Static analysis checks

Fault class	**Static analysis check**
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer Aritematic

Program Testing

- still predominant V&V technique
- can reveal the presence of errors NOT their absence
- When can a test be considered successful?
- the only validation technique for non-functional requirements
- should be used in conjunction with static verification to provide full V&V coverage