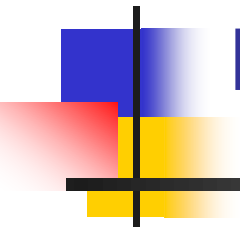
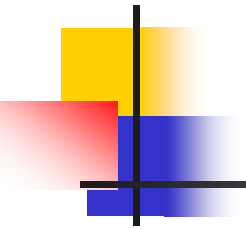


Basics of Software Engineering

J.W. Choi

2024

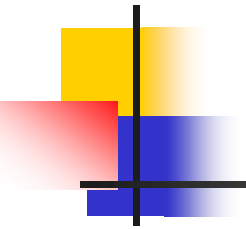
Basics of Software Engineering



Basics of Software Engineering

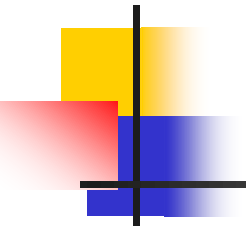
- Documentation
- Code Inspection
- Testing
- Coding Guidelines
- Defensive Coding
- Code Review

Documentation
(repeated from an earlier lecture)



Documenting a Program

- Block comments
  - function, module, subsystem, system
- Inline comments

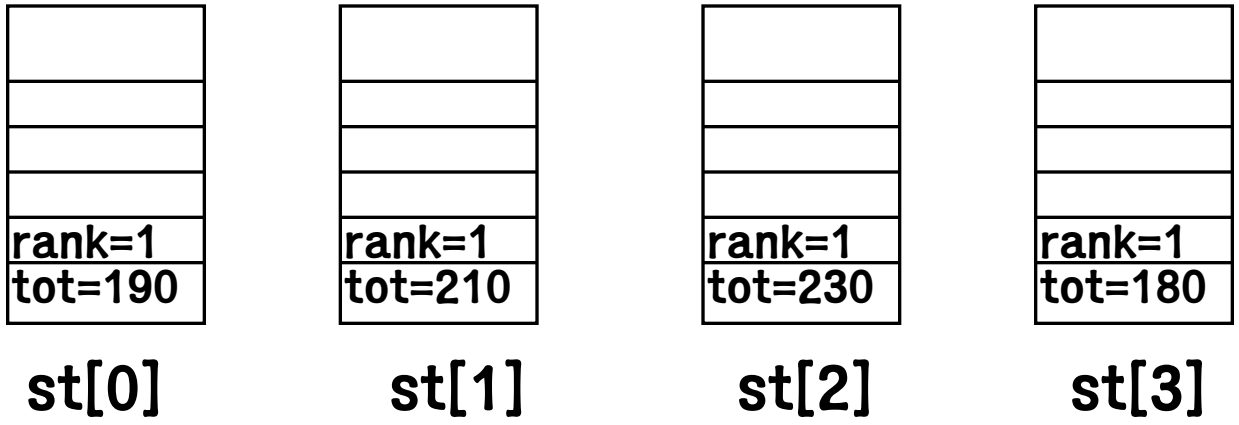


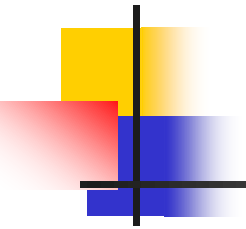
Block Comments

- Purpose of the block
- Input and output data
- Side effects
- Key logic used
- Other information helpful to reader
- Author, date, status

Example: rank function

```
void rank (struct student *pst, int n) {
    int i, j;
    /* rank is pre-set to "1" */
    for (i=0; i<n; i=i+1)
        for (j=0; j<n; j=j+1) /* see if pre-set rank will change*/
            if (pst[j].total > pst[i].total)
                pst[i].rank = pst[i].rank +1;
}
```





Function Comments (1/3)

purpose:

This function calculates the rank of each student based on his total score.

input:

pst: pointer to a struct array, and
n: the number of structs

each struct has total score.

output:

the calculated rank is stored in each struct in the struct array

Function Comments (2/3)

key logic:

the rank in each struct is first set to “1”.

(1) for the first struct,

scan “total” in the next n-1 structs

lower the “rank” each time a higher “total” is found in other structs

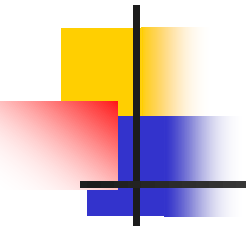
(2) for the second struct, scan “total” in the next n-2 structs

lower the “rank” each time a higher “total” is found in other structs

(3) continue for each of the remaining structs

Function Comments (3/3)

author:
    Hong Gildong
date:
    October 29, 2010
status:
    first written



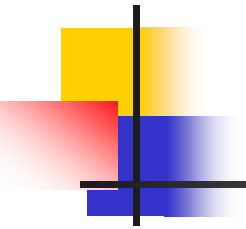
Inline Comments

- Simple explanation
  - for a block of code
  - for a key logic in a block of code

Inline Comments

```c
// Test function 1: Push and pop elements
void testPushPop() {
    Stack stack;
    initStack(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Popped: %d\n", pop(&stack)); // Should pop 30
    printf("Popped: %d\n", pop(&stack)); // Should pop 20
    printf("Popped: %d\n", pop(&stack)); // Should pop 10
    printf("Popped: %d\n", pop(&stack)); // Should indicate the stack is empty
}

// Test function 2: Check if the stack is empty
void testEmpty() {
    Stack stack;
    initStack(&stack);

    printf("Is stack empty? %d\n", empty(&stack)); // Should return 1 (true)

    push(&stack, 100);
    printf("Is stack empty? %d\n", empty(&stack)); // Should return 0 (false)

    pop(&stack);
    printf("Is stack empty? %d\n", empty(&stack)); // Should return 1 (true)
}
```

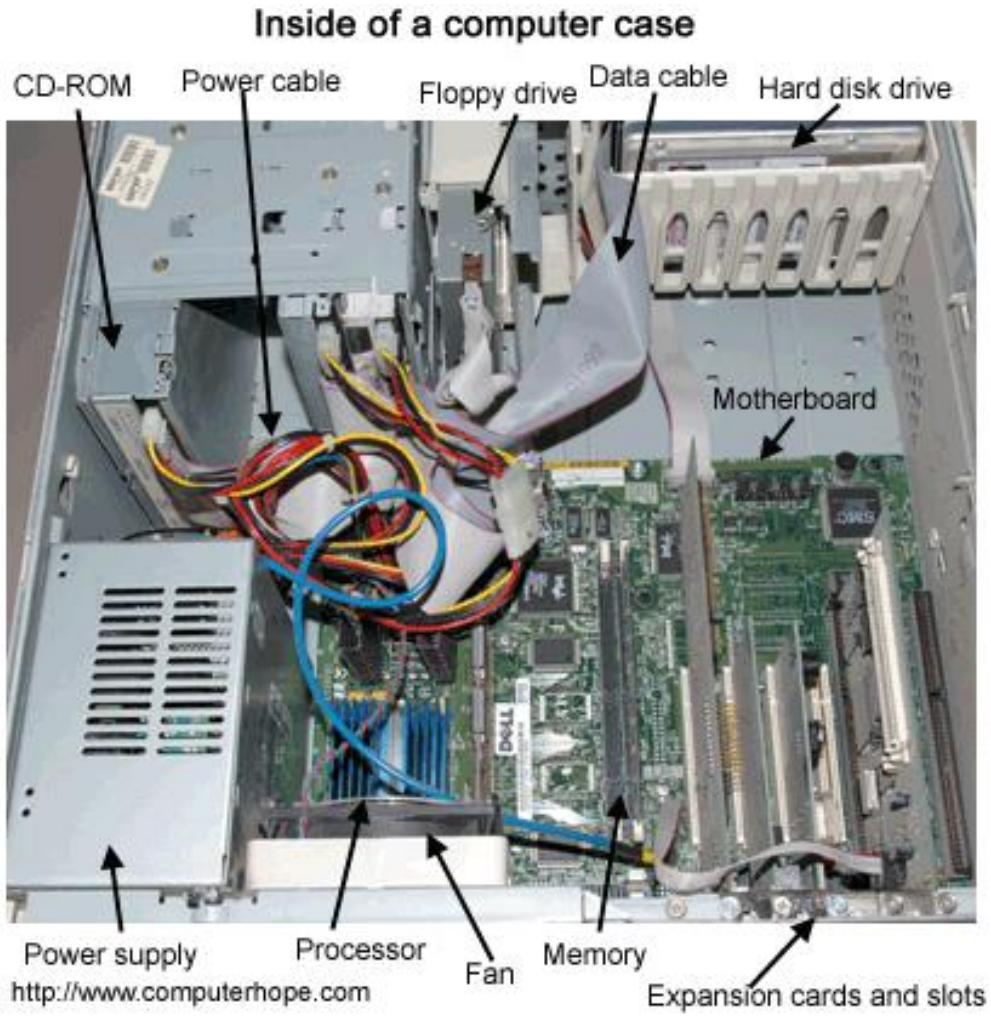
Code Inspection



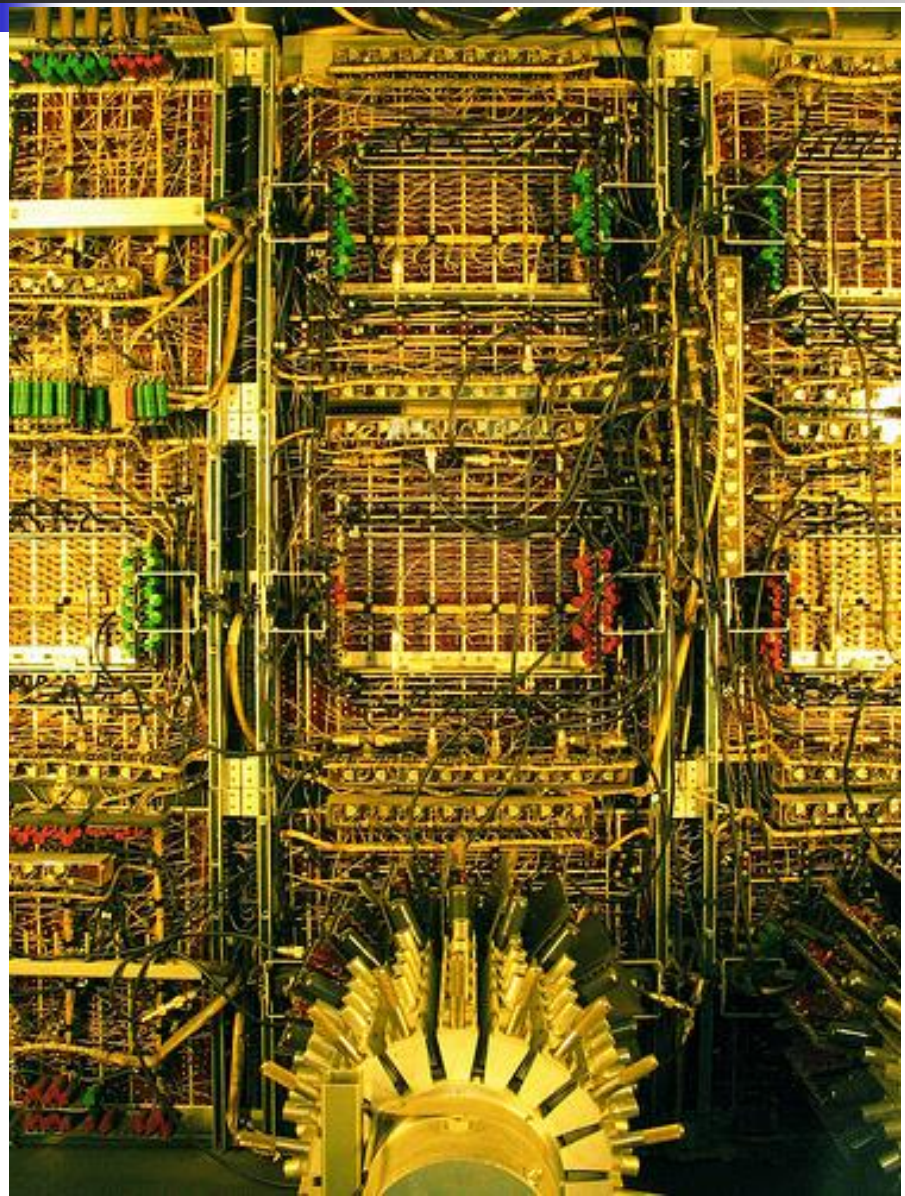
3 Uses of Code Inspection

- To detect errors and violation of guidelines before running a program
  - Code self-inspection
- To detect errors and violation of guidelines in other's program
  - Code review
- To identify errors when the program does not run
  - debugging

Inside of a Computer



“Bug” “Debug”

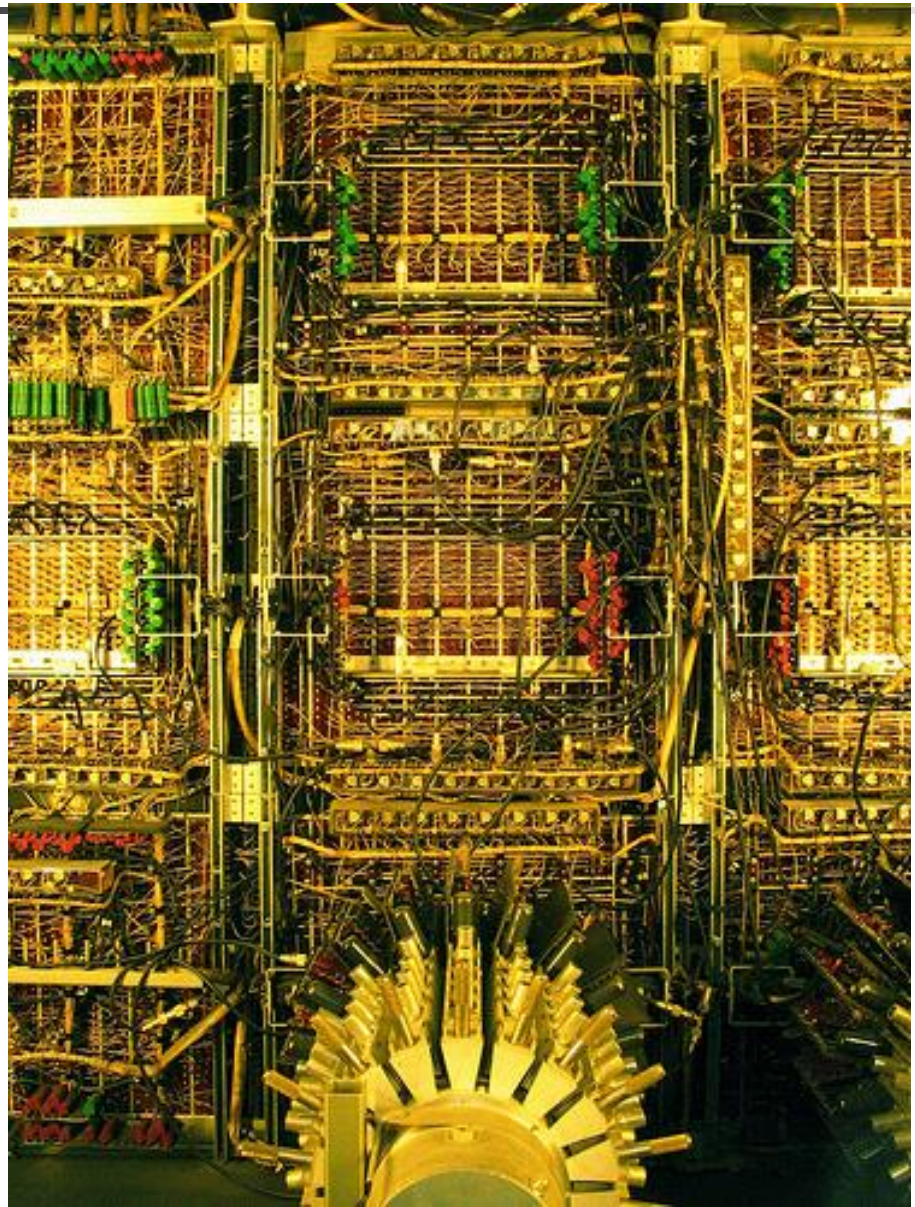


inside of

Univac 1

World’s First
Commercial Computer

“Bug” “Debug”



World’s First Computer Bug

Primary Errors to Look For (1/3)

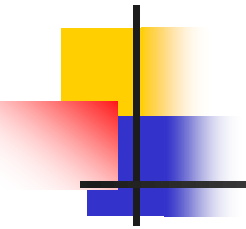
- loop termination and iteration conditions
  - wrong exit, wrong increment
  - infinite loop
  - dumb loop
- if-else conditions
  - compound conditions
  - long if-else chains
- switch conditions
  - switch value
  - break
  - default

Primary Errors to Look For (2/3)

- array and string
  - index overflow & underflow
  - dimension ordering for multi-dimensional array
  - end_of_line escape sequence for string
- function calls
  - return type and return values
  - number and data type of arguments and parameters
  - scoping rules for global variables and local variables

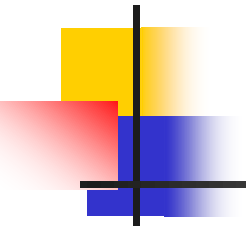
Primary Errors to Look For (3/3)

- verify values stored in memory (draw memory diagrams)
  - the values for variables
  - the addresses for pointer variables
  - function call argument values
  - function call return values



Secondary Errors to Look For

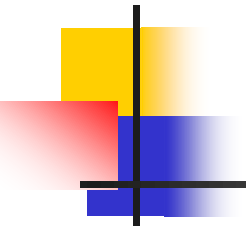
- #include standard library header files
- scanf (fscanf) conversion format specifiers
  - unneeded ,
  - missing &
- struct type definition not being global
- unneeded ;
- missing ;



Code Inspection and Debugging

- Manually compute the values of
  - each variable
  - each expression
  - each function call argument
  - each function call parameter
  - each function return value
- (If appropriate) minimize the problem size
  - loop iteration
  - array size
- Insert temporary printf, scanf statements
  - to print the values nearby (before or after) the problem being experienced.

Testing



Controlling the Number of Test Cases

- Integer Values
  - some small positive & negative numbers
  - some very large positive & negative numbers
  - zero
- String Values
  - some short strings
  - some long strings
  - some strings with some invalid symbols embedded
- Composite Input Values
  - some values for each component
- Data Sizes
  - small, medium, large

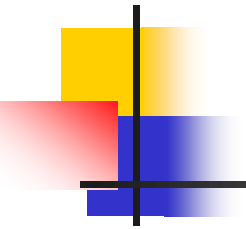
Positive Test, Negative Test

- Positive Test
  - Give valid inputs, and see if correct results are calculated.
- Negative Test
  - Give invalid inputs, and see if the errors are caught.
- Boundary Test
  - empty string, 1-too-many element in an array,…

Positive Test: Example

```
int   compute_age (int x, int y)
{
    return (x-y);
}
```

*** test "compute_age" with
correct integer values for x
(2008, larger than y)
correct integer values for y
(1980, smaller than x)



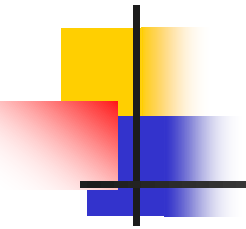
Positive Test

- Try large, mid, small numbers
- Try long, mid, short strings
- Try combinations of values
- Try 3 different orderings of data
  - ascending, descending, mixed

Negative Test: Example

```
int  age_compute (int x, int y)
{
   return (x-y);
}
```

*** test "age_compute" with
incorrect integer values for x
(1990, smaller than y, 0,...)
incorrect integer values for y
(2008, larger than x, -2000,...)



Negative Test

- Array underflow & overflow
- Wrong combinations of values
- Wrong data types
- 0 divide
- Invalid input argument in a function call
- Invalid return from a function call

Defensive Coding

- Catch errors, and avoid crashes or hanging
- Examples
  - default case in a switch statement
  - check for file open failure
  - check for malloc failure

Defensive Coding

- Array
  - check for overflow
  - check for underflow
- Function
  - function definition
    - check input values overflow
    - check input values underflow
    - check for zero divide
  - function calling
    - check for error return code
    - check for incorrect result

Example

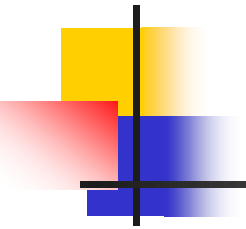
```
int  age_compute (int x, int y)
{
    if ((x-y)<0)
        return -1;
    return (x-y);
}
```

The Millennium Bug

char year[2];

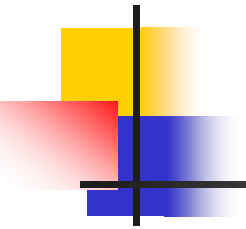
(19)00 -- 99

Formal Software Testing



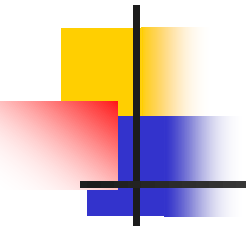
Software Testing

- Unit testing
- Integration testing
- System testing
- Regression testing
- Deployment testing
- Acceptance testing



Unit Testing

- Test each function separately
- Create test cases
  - positive test cases
  - negative test cases
  - boundary test cases
- Use scanf/printf, or write to a file
- Use a test harness/test script

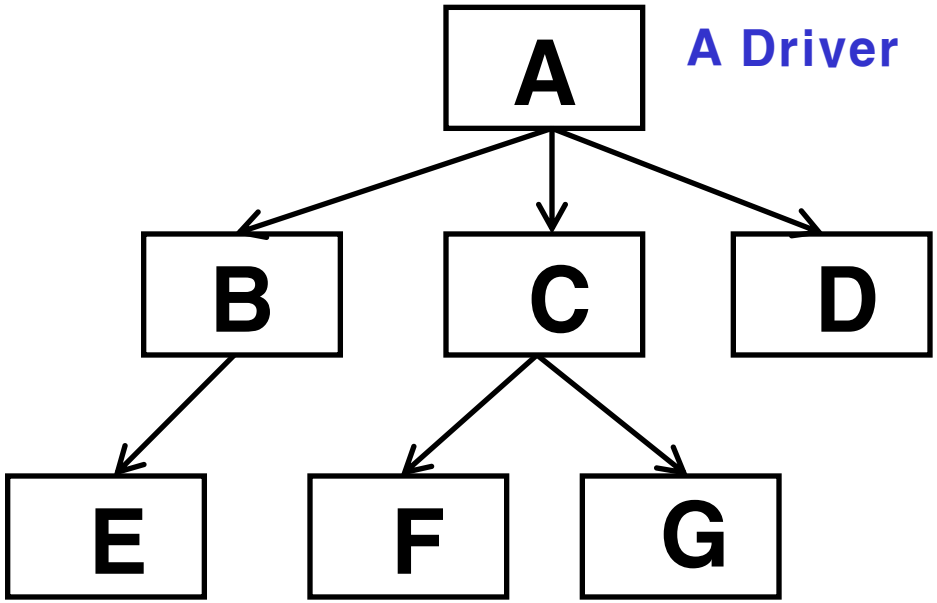


Integration Testing

- Test each module
  - Module Is a collection of functions.
  - module/component hierarchy
- Integration testing methods
  - bottom-up testing
  - top-down testing
  - hybrid testing
- Use a test harness/test script or scanf/printf
- Desirable to save test cases
  - for inclusion in system test suite

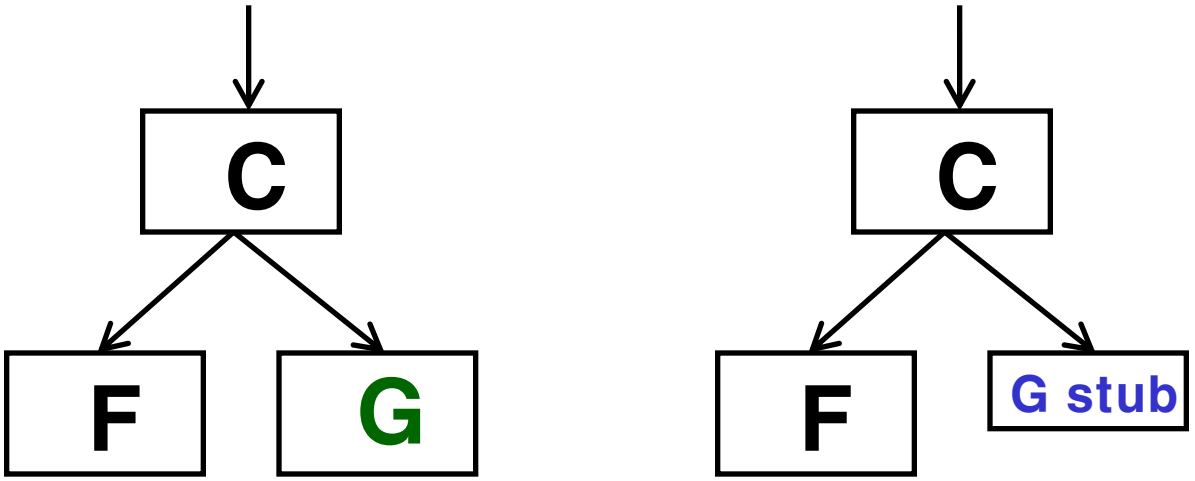
Bottom-Up Testing

- Start with leaf level modules, and
- Move up the module hierarchy.



Top-Down Testing

- Start at a higher level module, and
- Move down the module hierarchy.
- Use "stub function" or comment out "function call", if level module is not ready.



Exercise: Create Positive and Negative Test Cases for the Following Program.

```
int binary_search (int list[], int size, int key) {

   int index=-1, found=0, left=0, right=size-1, mid;

   while (left <= right) && !found) {
       mid = (int) ((left + right) / 2);
       if (key == list[mid]) {
           found = 1;
           index = mid;
           }
       else if (key > list[mid])
           left = mid +1;
       else
           right = mid -1;
        }
   return (index);
}
```

binary search

searching a key
by repeatedly
bisecting a sorted list

ex: search for 15

2 5 6 9 12 15 20
2 5 6 9 12 15 20
9 12 15 20
12 15 20

Solution

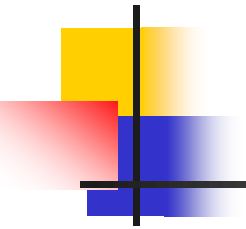
positive test cases

list	size	key
[2 5 6 9]	4	6
[2 5 6 9]	4	2
[2 5 6 9]	4	1
[2 5 12]	3	5
[2 5 12]	3	12
[2 5 12]	3	20
[2]	1	2
[2]	1	5
[ ]	0	7

negative test cases

list	size	key
[5 2 9 6]	5	6
[2 5 6 9]	3	9
[2 5 6 9]	6	2
[2 5 12]	0	5
[2]	3	12
[ ]	3	7

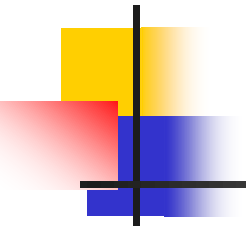
Coding Guidelines



Coding Guideline

- Make use of symbolic constants for constants subject to change.
- Avoid creating similar functions by copy and paste.
- Eliminate dead code.

- Free up resources when they are not needed
  - free after malloc
  - file close after file open



Eliminate Dead Code

- Variables and symbolic constants that are never used
- Statements that are never executed
  - impossible case in a switch statement
  - impossible else statement
- Functions that are never called

Exercise: Find Dead Code

```
int maxage=-1;
int maxagecnt;
int age[100], maxagegrp[100], minagegrp[100];
int i;

maxagecnt = 0;
for (i=0; i<100; i=i+1) {
   if (i > 100)
      break;
   if (age[i] > maxage) {
      maxage = age[i];              /* new max age found  */
      maxagecnt = 0;
      maxagegrp[maxagecnt] = i;  /* save the age index  */
   }
   else if (age[i] == maxage) {
      maxagecnt = maxagecnt + 1; /* tie max age found  */
      maxagegrp[maxagecnt] = i;  /* save the age index  */
   }
```

Scaling Up

Problem Sizes (Review)

- "Nothing"
  - examples in introductory C programming textbooks
  - exercises in "problem solving using C" course
- Tiny
  - exercises in "problem solving using C" course
  - team programming exercises in "problem solving using C" course
- Very Small
  - video rental management system, product defect detection system
- Small
  - VI editor, C compiler
- Medium-Size
  - Web browser, anti-virus software, C compiler, accounting software
- Large
  - word processor, email system, Web portal system, Internet search engine, enterprise application software, distributed online game
- Very Large
  - enterprise database management system, VoIP system
  - mobile phone software platform
- Huge
  - Windows Vista operating system

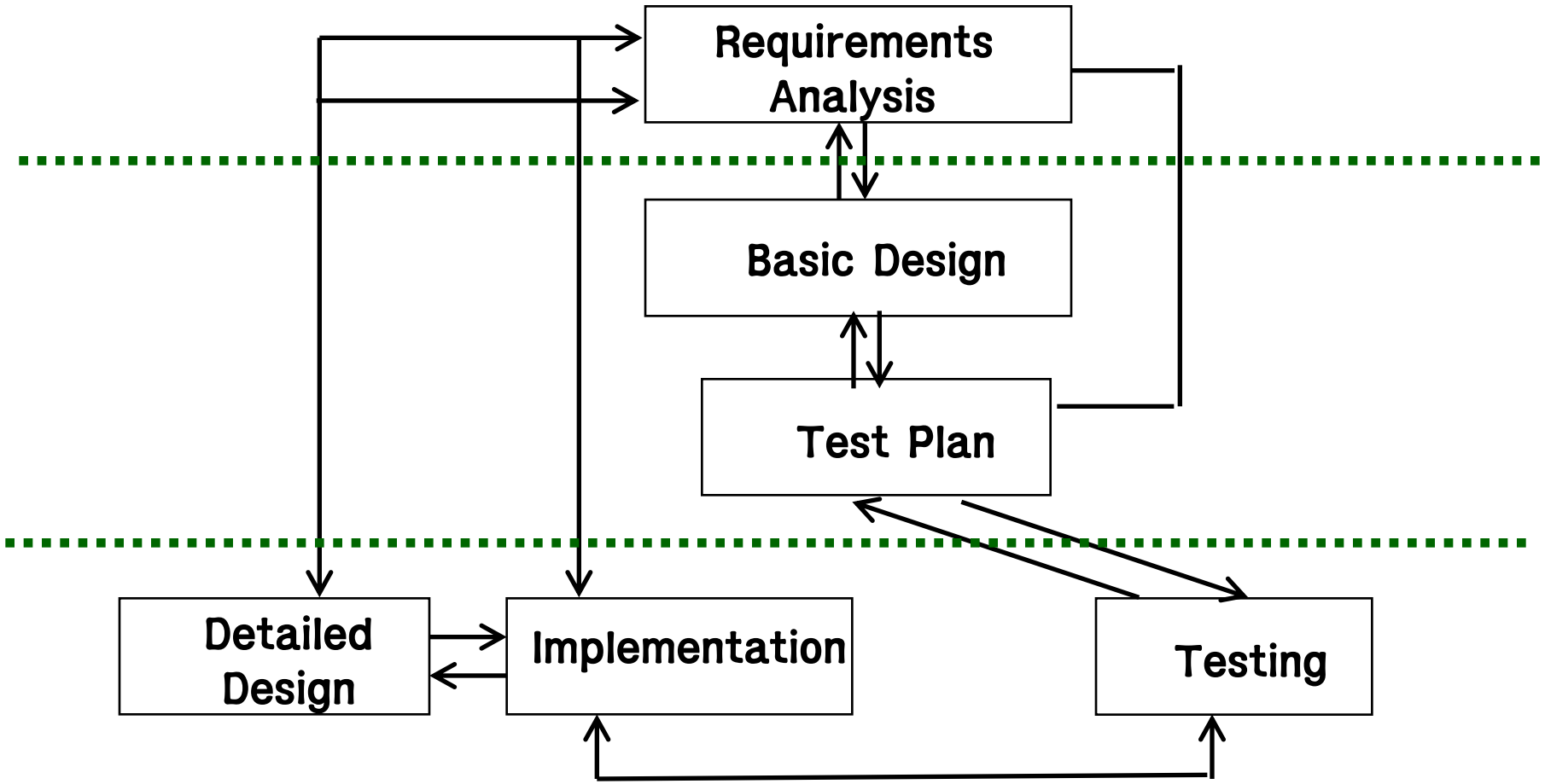
As Problem Size Increases

- Problem complexity
- Architecture issues
  - distributed computers
  - parallel computers
  - many concurrent users
  - performance, scalability, reliability, security, availability, extensibility, portability, modifiability
- Manpower requirements
- Business issues
- Legal issues

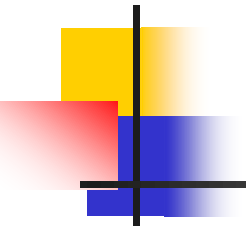
Software Lifecycle Process

- Planning
- Development
  - upstream
    - requirements analysis (requirements spec, review)
    - architecture (basic) design (architecture design spec, review)
    - test plan (test plan, review)
    - detailed design (detailed design spec, review)
  - downstream
    - implementation and development testing (source code documentation, code review, test cases, review)
    - system testing (test cases, review)
    - acceptance testing and release (release notes, user manuals, review)
- Maintenance
  - defect database
- Termination

Simple Software Development Process



Development by a Team

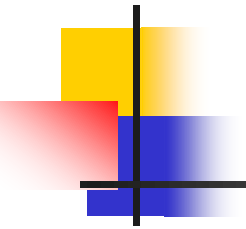


Four Principles for the Process

- Plan
- Document
- Review
- Iterate

Divide and Conquer

- Organize the problem and software into multiple levels.
  - system -> subsystems -> modules -> functions
- Apply the problem solving approach at each level.
- Mix top-down and bottom-up approaches.



Function Level

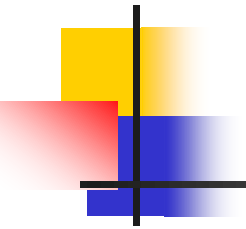
- step 1: understand the problem
- step 2: outline a solution

- step 3: form a program structure
- step 4: write a program outline (pseudo code)
- step 5: write the program

- step 6: compile the program
- step 7: inspect the program
- step 8: test the program

- iterate all steps above, as necessary, for optimization and correctness.

- step 9: document the program



Module Level

- step 1: understand the problem
- step 2: outline a solution

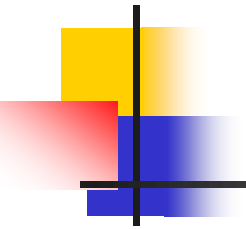
- step 3: form a module structure
- step 4: write a module outline (pseudo code)

- step 5: integrate functions

- step 7: inspect the module
- step 8: test the module

- iterate all steps above, as necessary, for optimization and correctness.

- step 9: document the module



Subsystem Level

- step 1: understand the problem
- step 2: outline a solution

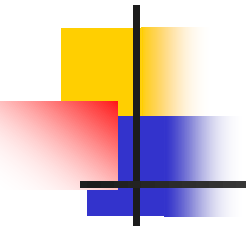
- step 3: form a subsystem structure
- step 4: write a subsystem outline (pseudo code)

- step 5: integrate modules

- step 7: inspect the subsystem
- step 8: test the subsystem

- iterate all steps above, as necessary, for optimization and correctness.

- step 9: document the subsystem



System Level

- step 1: understand the problem
- step 2: outline a solution

- step 3: form a system structure
- step 4: write a system outline (pseudo code)

- step 5: integrate subsystems

- step 7: inspect the system
- step 8: test the system

- iterate all steps above, as necessary, for optimization and correctness.

- step 9: document the system

End of Class