A program is correct if it compiles without errors and when executed produces output that satisfies the specification for the program. Correctness is more important than efficiency (or anything else).

Levels of Correctness:

1. No syntax errors
2. No semantic errors
3. There exists some test data for which the program gives a correct answer
4. The program gives a correct answer for a reasonable set of data
5. The program gives a correct answer for difficult test data
6. The program gives a correct answer for test data of any kind
7. There exists some test data for which the program gives a correct answer
8. No errors occur
9. No syntax errors
10. No semantic errors

Software Debugging and Testing

Debugging is the process of finding errors in a program under development.

Testing is the process of determining if the errors in a program that is thought to be correct.

Testing attempts to establish that a program satisfies its specification.

Exhaustive testing is not possible for real programs due to combinatorial explosion. Amount of testing performed must be balanced against the cost of undiscovered errors.

Regression testing is used to compare a modified version of a program against a previous version.

Testing can establish the presence of errors but cannot guarantee their absence.

S. McConnell Chapter 26, 25

Supplementary reading

Reading Assignment

Supplementary notes

CSC 181F Lecture Notes
Testing Based on the Source Program

Branch testing - design test cases to cause each condition in if to evaluate.
Condition testing - design test cases to test all possible outcomes for each

Derive test cases from examination of the program.
The program is executed at least once
Basic Path testing - design test cases to guarantee that every path through

Testing Strategy

Try to test all parts of the program.
Test if response is probable errors (e.g., invalid input data). 

Sources for Test Cases

Requirements and Specification for the program
General knowledge of the application area
Program design and user documentation
Specific knowledge of the programming language and implementation
Specific knowledge of the program source code (White Box Testing)

Insight and imagination are essential in the design of good test cases.
Test case selection is one key factor in successful testing.
When can I do make the program look good?
and not
What is the absolutely worst thing I can do to the program?

Test cases should be:
Testing efficiency is measured in the number of errors discovered per hour of
Program under test with the least effort expended.
The goal in testing software is to find as many errors as possible in the
Any array containing exactly one or zero elements
Any array with zero elements of the same type
Any array with one element of the same type
Any array with an empty size, not equal to the size of the array
Any array containing ordered, reverse ordered, unordered data
Any array with one element, not in the above in the last
Any array with one element, not in the above in the middle

Test Data for Search

Testing Example - Quadratic Program

Easiest Quadratic

Testing Example - Quadratic Program

Testing - Example

int Search(int Ar[] , const int ArSize , const int val)

Specification

val is an element of the array Ar. Then Search

in Search( Ar[] , ArSize , val)

Program

Search an array for a given value

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example

Testing - Example
How to inspect Programs

1. Learn from your mistakes. If you consistently make one kind of error, inspect extra.
2. Check all function calls for the correct type and order of parameters.
3. Check all expressions to make sure the correct value is being computed.
4. Check that all subscript expressions will be valid.
5. Check all conditions in all statements.
6. Check that all subexpressions will always have a value before they are used.

Check that every variable will always have a value before it is used.

Checking for uninitialized variables.

Example:

```c
sum += A[K];
```

- sum += 4[A K]++
- `for (K = 0; K < ASIZE; K++)`
- Assume A is given a value here
- `++ A[K]` +=
- Check sum, `A ASIZE`!


Eliminating uninitialized variable errors by inspection is much more effective than tracing and debugging a running program.

Uninitialized Variable Errors

An uninitialized variable error occurs when the value of a variable is used in a program before it has been assigned a value.

- The program looks OK, uninitialized variable errors are hard to see.
- Symptom: Erratic - adding debugging code may change or obscure the error.
- Symptom: May not show up when one module is tested alone.
- Uninitialized variable errors are often very hard to find.
### The Integer Types

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Size Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned char</code></td>
<td>8 bits</td>
</tr>
<tr>
<td><code>signed char</code></td>
<td>8 bits</td>
</tr>
<tr>
<td><code>short</code></td>
<td>16 bits</td>
</tr>
<tr>
<td><code>short int</code></td>
<td>16 bits</td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td>16 bits</td>
</tr>
<tr>
<td><code>int</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>signed int</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>unsigned</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>long</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>long int</code></td>
<td>32 bits</td>
</tr>
<tr>
<td><code>unsigned long</code></td>
<td>32 bits</td>
</tr>
</tbody>
</table>

---

### Program Inspection Example

```c
/* Read in year and day, output month, day, year. */
const char * monthName[13] = /* monthName[0] unused */
    "", "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December";
short monthLength[13] = /* monthLength[0] unused */
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31;

int year, dayInYear, precedingDays, month = 0;

while (true) /* Infinite loop to read then output dates */
    while (true) /* Process one correct input */
        printf("Enter year and number of days in year
\n") ;
        if (scanf("%d%d", & year, & day) == 2) {
            if (1900 <= year && year <= 3000 && dayInYear >= 1) {
                if (year % 4 == 0) monthLength[2] = 29;
                if (dayInYear <= 366) break;
                else monthLength[2] = 28;
            } else if (dayInYear <= 365) break;

            precedingDays = 0;
            while (dayInYear > precedingDays + monthLength[month])
                precedingDays = precedingDays + monthLength[month];
            month++;
            printf("The Date is %s %d, %d
\n", monthName[month], dayInYear - precedingDays, year);
        }
```

---

### Reading Assignment

Sections 16.5, 7.2, 7.5
K.N. King
Enumerations are a mechanism that allows you to declare a set of related symbolic names in cases when you don't care about the internal representation.

Use enumerations to represent the state of variables that take on a small number of values. The symbolic names make the program easier to read. This same effect could be achieved using `#define` but enum is more elegant and makes the program easier to read.

Almost all enumeration declarations should have an enum tag (unless they appear in a typedef declaration).

### Examples:

```c
enum directions
    south, southWest, west, northWest, north, northEast, east, southEast;
enum StopLight
    green, flashingGreen, yellow, red;
typedef enum
    Clubs, Diamonds, Hearts, Spades
    Suit;
```

### HOW TO USE ENUMERATIONS

1. **Declaration**: Define the enumeration type with an `enum` keyword, followed by a list of symbolic names (the identifiers). Optionally specify an enumTag, which provides a name for the enumeration type.

2. **Declaration**: The compiler will assign an internal representation to each identifier in the list. If you specify values, the compiler will use those values for the identifiers.

3. **Conversion**: You can assign values to enum variables.

4. **Typedef**: You can declare a typedef for an enumeration to reuse the symbols in another part of your program.

5. **Implicit Conversion**: When using an enum in an expression, it is implicitly converted to an int.

6. **Explicit Type Conversion**: You can convert an enum value to another type using an explicit cast.

```
int I;
float X;
...
I = (int)X;
X = (float)I;
```

### Casting - Explicit Type Conversion

- **Example**: `int x = (int) 1.5;`  
- **Result**: `int 1` or `long 1` (unsigned)

### Enum Names

- **Examples**: `0123 1.234 1234.567 1234.567 FP1:

- **Example**: `long double 80/128 bits`
The bitwise operators can be used to extract information and to create a hole to put it into, so packing or unpacking can be done. The space saving is usually important.

Information is stored in the computer's registers, and there is a simpler alternative.

Bitwise operators should not be used if they make the program hard to understand and if there is a simpler alternative.

Bitwise operations can be used to extract information and to create a hole to put it into. There is usually a simpler alternative.

To use bitwise operators you need to understand how information is represented internally in the computer. See previous slide.

Bitwise operations are used to manipulate the pattern of bits in an expression, e.g., extracting or combining information.

WARNING: BE CAREFUL, don't confuse

A & B could be zero (false) even if A and B are non-zero (true).

A and &&, | and ||, ~ and !

Right shift this

Left shift this

Bitwise exclusive or

Bitwise and

A B | A & B | A B | A B

Bitwise Operators Definitions

Bitwise Operators

Bitwise Operator Examples

unsigned short A, B, C, D; /* 16-bit variables */

/* Value in Binary (base 2) */
A = 03567; /* 0000011101110111 */
B = 255; /* 0000000011111111 */
C = 0x35AF; /* 0011010110101111 */
D = ~C; /* 1100101001010000 */
D = B & C; /* 0000000010101111 */
D = ~B & C; /* 0011010100000000 */
D = A | C; /* 0011011111111111 */
D = A ^ C; /* 0011001010111000 */
D = B << 3; /* 0000011111111000 */
D = C >> 7; /* 0000000001101011 */

/* Unpacked Short A, B, C, D */

* Bitwise Operators Definitions

* Bitwise Operators

* Bitwise Operator Examples

* Bitwise Operators Definitions

* Bitwise Operators
Storage Classes

A storage class is associated with every declared object. This storage class determines the extent (lifetime) of the storage associated with the object. In some cases the storage class also affects the visibility of the object.

The storage classes in C

- auto: locally created storage is the default
- static: for variables indicates statically created permanent storage also restricts visibility to file of declaration
- extern: name is visible outside file of declaration
- register: hints to compiler to store variable in a hardware register

Examples

static double RandomSeed = 123456.789;
register int I, K;
extern long sharedData;

/* Extract D from x and store in tmp */
dtmp = (x & DMASK) >> DSHIFT;

/* Insert new D value (dtmp1) into x, trim to fit. */
x = (x & ~DMASK) | ((dtmp1 & FIVEBITS) << DSHIFT);

Data Packing Example

Assume that six 5-bit integers (values 0..31) are to be packed in a 32-bit unsigned variable. This example shows how to access one field (D) of the packed information.

#define DMASK (0x00007C00) /* 0..0111110000000000 */
#define FIVEBITS (0x1F) /* 0..011111 */
#define DSHIFT (10) /* # bits to the right of D */

unsigned x; /* Variable containing A, B, C, D, E, F */
short dtmp, dtmp1; /* Variable to hold D */
x = 0xCAE1DB75;
dtmp1 = 0x35;

/* Extract D from x and store in tmp */
dtmp = (x & ~DMASK) >> DSHIFT;
/* Insert new D value (dtmp1) into x, trim to fit. */
x = (x & ~DMASK) | ((dtmp1 & FIVEBITS) << DSHIFT);
The factorial function is a very simple example of a function that can be computed using recursion.

**Mathematical Definition**

\[
\begin{align*}
\text{if } n < 0 & \\
\text{then } N & = 0 \\
\text{else } N & = (1 - \gamma) \cdot N + 1
\end{align*}
\]

The factorial function is a very simple example of a function that can be computed using recursion.

**Simple Example - Factorial**

**Why Recursion?**

- Recursion divides a large problem into smaller, easier to solve pieces.
- Use of recursion often leads to simpler, more elegant algorithms.
- Recursive programs are easier to make correct.
- Recursive solutions are frequently simpler than non-recursive solutions.

**Keywords**

- Divide and Conquer
- Induction

**Reading Assignment**

K.N. King Section 9.6
E.Roberts, Programming Abstractions in C
E.Roberts, Thinking Recursively
S. McConnell Chapter 16
R.N. King
SHORT-TERM LOAN ENGINEERING LIBRARY

**Supporting Reading**

SHORT-TERM LOAN ENGINEERING LIBRARY

**Extremely Important Programming Technique**

Recursion

**Key Insights**

- Recursive solutions are frequently simpler than non-recursive solutions.
- Use of recursion often leads to simpler, more elegant algorithms.
- Recursive programs are easier to make correct.
- Recursive solutions are frequently simpler than non-recursive solutions.

**Why Recursion?**

- Recursion divides a large problem into smaller, easier to solve pieces.
- Use of recursion often leads to simpler, more elegant algorithms.
- Recursive programs are easier to make correct.
- Recursive solutions are frequently simpler than non-recursive solutions.

**Keywords**

- Divide and Conquer
- Induction

**Reading Assignment**

K.N. King Section 9.6
E.Roberts, Programming Abstractions in C
E.Roberts, Thinking Recursively
S. McConnell Chapter 16
R.N. King
SHORT-TERM LOAN ENGINEERING LIBRARY

**Supporting Reading**

SHORT-TERM LOAN ENGINEERING LIBRARY

**Extremely Important Programming Technique**

Recursion
The Towers of Hanoi Problem

In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles each acubit high and as thick as the body of a bee. On one of these needles are placed sixty-four disks of pure gold, the largest disk on the top needle and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and Night, unceasingly, the priests transfer the disks from one needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and there must be no smaller disk below it. When all the 64 disks have been transferred from one needle to one of the other needles, tower, temple and Brahmin alike will crumble into dust and with a thunderclap the world will vanish.

W.W.R. Ballas quoted by E. Roberts, Programming Abstractions in C, page 196

HOW TO Use Recursion

Analyzing the Problem

Identify simple cases

Identify way to divide problem

Simple cases

Rest of problems same/similar form as the problem

Select data structure to represent problem

Writeln "Generic Recursive Model"

Recursive functions & procedures

Handles simple cases directly

Uses decomposition and recursion on the rest

When recursive function is called

else

Similiar to mathematical induction

Base problem same as the problem

Identify simple cases

Analyze the problem

Trace of call Factorial(5)

Example - Factorial

Trace of call Factorial(s)
HOW TO Avoid Pitfalls in Recursive Solutions

Aresimplecasescheckedforfirst?
Arethesimplecasessolvedcorrectly?
Doestherecursivecompositionmaketheproblemsimpler?
Eachrecursionshouldmaketheprogress towardreachingoneofthesimplecases.
Willtherecursionalwaysterminate?
Doesthesimplificationprocessexaminetothesimplecases?
Havesomesimplecasesbeenoverlooked?
Dotherecursivecallseachrepresentsubproblemsthataretrulyidenticalinform?
Dothesolutionstotherecursivesubproblemsprovideacompletesolutiontotheoriginalproblem?

Example-Hanoi
/*SolutiontoTowersofHanoiPuzzle*/
voidhanoi(constcharsource, constchardest, constchargetemp, constintN)
/*MovetopN-1diskstheawaytotemporarystorage*/
if(N==1)
printf("Moveadiskfrom%c
to%d
",source,dest);
return;
else
/*Fixthestacksothatdisk1canbedropped*/
hanoi(source,temp,dest,N-1);
/*Movethebottomdisktotheoriginaldestination*/
printf("Moveadiskfrom%c
to%d
",source,dest);
/*MoveotherN-1diskstheoriginaldestination*/
hanoi(temp,dest,source,N-1);

HOW TO Find a Recursive Strategy

Anyproblemyouwanttosolveusingrecursionmustsatisfytheconditions:
1. There mustbeoneormoresimplecases.
2. Itmustbepossibletobreaktheproblemdownintosimplerparts.
3. Solutionofthesubproblemsmustsomewayhelp tosolvethelarger
   decompositionandrecombinationareoftenthehardestpartsofthestrategy.
   Onceyou'vedesignedarecurativesolution,youshouldvalidate thestrategyby
   workingthroughafewsimpleexamplesbyhand.

Example-TowersofHanoi
MoveNdiskfromonepegtoanotherbymovingonediskatatimesubjectto
theconstraintthatalargerdiskmayneverbedroppedonasmallerdisk
Analysis:
Simplecases:movetopN-1diskstheaway
Division:Movebottomdisktothefinaldestination
Composition:MoveremainingN-1diskstothedestinations
Example - Binary Search

Searchsortedtable(array) for key
Return table index if found, -1 otherwise.

Analysis:
Simple cases: empty table, table with one entry
Division: split table in half
Composition: result from half of table


Example - IsPalindrome

BoolIsPalindrome(const char string[])
/* Return TRUE if string is a palindrome, FALSE otherwise */
return IsPalindrome2(string, 0, strlen(string) - 1);

BoolIsPalindrome2(const char string[], const int first, const int last)
if (last - first <= 1) return true;
else return (string[first] == string[last]) && IsPalindrome2(string, first + 1, last - 1);

Analysis:
Return true if found. -1 otherwise.
Search sorted table (array) for key.

HOW TO Use Helper Functions

The solution to IsPalindrome uses a helper function IsPalindrome2 to do the real work.
Helper functions are appropriate when you need some extra parameters to carry additional information between levels of recursive calls. Think of using helper functions in cases where the function you need to write doesn't have all the parameters you need to compute its value efficiently. Try to use the minimum number of extra parameters needed to solve the problem. Usually one or two.

Example - Detecting Palindromes

A palindrome is a string that reads identically forward and backward. Examples: "level", "Madam I am Adam"

Design a recursive strategy to determine if a given string is a palindrome.

Analysis:
Delete a recursive strategy to determine if a given string is a palindrome.
Examples: "level", "Madam I am Adam"
Example - TreeSum

```c
int treeSum(const TreeNode* treePtr) {
    if (treePtr == NULL) return 0;
    if (treePtr->isLeaf) return treePtr->value;
    else return treeSum(treePtr->left) + treeSum(treePtr->right);
}
```

Analysis:
- Simple cases: null tree, leaf
- Decomposition: left branch & right branch
- Composition: sum of left and right branches

Trees using Arrays

```c
#define MAX_TREE 1000

typedef int TreeNodeType;
typedef int TreeLeafType;

TreeNodeType treeLeaf[MAX_TREE];
TreeNodeType treeLeaf[MAX_TREE];
TreeNodeType treeLeft[MAX_TREE];
TreeNodeType treeRight[MAX_TREE];
```

We'll see much better ways to do trees later in the term.

Example - Binary Search

```c
int binSearch(const int table[], const int low, const int high, const int key) {
    int mid = (low + high) / 2;
    if (high < low) return -1;
    if (table[mid] == key) return mid;
    else if (table[mid] > key) return binSearch(table, mid + 1, high, key);
    else return binSearch(table, low, mid - 1, key);
}
```