Computer-Based Problem Solving

- What is problem solving?
- Problem of problem solving
- Problem solving process
- Some problem solving techniques

What is Problem Solving?

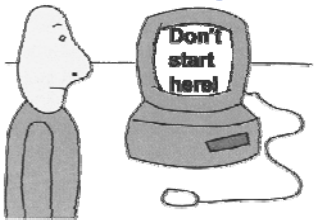
- The entire process of...
  - … taking the statement of a problem and...
  - … developing a computer program that solves the problem
- A widely recommended approach to this process requires you to systematically go through several phases,...
  - from analyzing, understanding and specifying the problem…
  - through designing a conceptual solution,…
  - to implementing and testing the solution with a programming language
    - Where software is involved → part of *software life cycle*

Problem of Problem Solving

- Two independent activities
  - Conceptual problem solving (*no need for computer*) – produces design of solution
  - Programming – implements design

Problem of Problem Solving

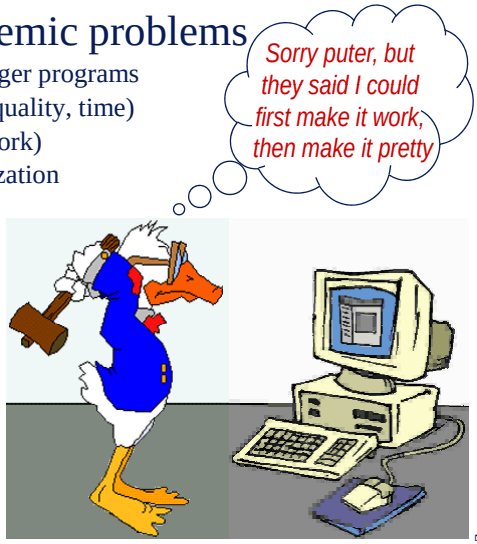
- Two independent activities
  - Conceptual problem solving (*no need for computer*) – produces design of solution
  - Programming – implements design



- Conceptual problem solving should *always* precede programming
  - *Good programs* generally follow easily *from good designs*
  - *Poor programs* generally result when problem solving *begins with typing in programs at the keyboard*
  - A lesson some only learn with experience, and some never seem to learn it at all

For Those Who Think It's All Bull...

- Real-world versus academic problems
  - Are more complex and require larger programs
  - Have tighter constraints (budget, quality, time)
  - Entail collaborative effort (teamwork)
  - Can't do without planning, organization and communication
- Haphazard approach...
  - May have gotten you through CS 1428/2308
  - May even get you through CS 3358
  - Won't serve you well in the long run
  - Won't cut it in the real world



…House Rules DO Exist After All

- You guessed it right
  - Your homework assignments/projects will be graded in part for good problem solving approach *explicitly* shown in your work
  - Good *programming style* counts
    - Be sure to do justice to programming *style guide* (on class homepage)
    - Bad programming style = loss of valuable points

Problem Solving Process

- Problem analysis
  - Thoroughly study and understand the problem to be solved
  - Precisely know the specific goals (results/outputs) to be realized
- Solution design – conceptual problem solving
  - Specification: *what* is the problem ← to solve the right problem
  - Design: *how* to solve the problem ← to solve the problem right
    - Data structures and algorithms
  - Analysis: *how* well is problem solved
- Solution implementation and testing
  - Program the solution and check correctness and efficiency

Abstraction: A Problem Solving Technique

- Abstraction – 2 key ideas (check dictionary)
  1. Process of forming a general concept from consideration of particular instances
  2. Process of ignoring those aspects of the subject not relevant to the current purpose in order to concentrate solely on those that are
- Abstraction – 2 key ideas more simply put
  1. Forming generalizations by studying specific instances
  2. Hiding irrelevant/unnecessary details

Abstraction: How Is It Important?

- Enables us to manage complexity in 2 major ways
  1. *Complexity in **data** through **data abstractions***
  2. *Complexity in **algorithms** through **procedural abstractions***

`

Abstraction: How Is It Important?

- Enables us to manage complexity in 2 major ways
  1. *Complexity in **data** through **data abstractions***
     - identifying/building *data types* based on generalizations formed by studying specific instances of data items (each data type serves to capture the common characteristics of all data items belonging to a generalization and each data item belonging to a generalization is then represented by an *instance of the type* representing the generalization)
     - a vital part of the object-oriented paradigm
     - bottom-up design strategy
  2. *Complexity in **algorithms** through **procedural abstractions***
     - thinking about something via its external behavior (interface) without concern for its internal workings (construction/implementation)
     - decomposing a complex problem into smaller, less-complex problems, and so on until a manageable level is reached
     - top-down design strategy

Procedural Abstraction in the Context of C++

- Functional abstraction is perhaps more appropriate
  - Since algorithms are implemented as functions in C++
- To use a function, we
  - need to know *what* the function does, NOT *how* it does it
- When designing a function, we
  - shouldn't have to worry about how other functions perform their jobs
  - but if our function need to interact with other functions, we do need to know something about what the other functions do
- The trick is to know *just enough* and *no more*
  - One technique is to specify functions using *preconditions* and *postconditions*

Preconditions and Postconditions

- An important topic: ***preconditions*** and ***postconditions***.
- They are a method of specifying what a function accomplishes.

Preconditions and Postconditions

Frequently a programmer must communicate precisely ***what*** a function accomplishes, without any indication of ***how*** the function does its work.

Can you think of a situation where this would occur ?

Example

- You are the head of a programming team and you want one of your programmers to write a function for part of a project.

What are Preconditions and Postconditions?

- One way to specify such requirements is with a pair of statements about the function.
- The ***precondition*** statement indicates what must be true before the function is called.
- The ***postcondition*** statement indicates what will be true when the function finishes its work.

Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{


 . . .


}
```

Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{
    . . .
}
```

- The precondition and postcondition appear as comments in your program.
- They are usually placed after the function's parameter list.

Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{
    . . .
}
```

- In this example, the precondition requires that

  x >= 0

  be true whenever the function is called.

Example

Which of these function calls meet the precondition ?

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```


Example

Which of these function calls meet the precondition ?

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

The second and third calls are fine, since the argument is greater than or equal to zero.

Example

Which of these function calls meet the precondition ?

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

But the first call violates the precondition, since the argument is less than zero.

Example

```
void write_sqrt( double x)
//  Precondition:  x  >=  0.
//  Postcondition:  The square root of x has
//  been written to the standard output.
{

 . . .


}
```

- The postcondition always indicates what work the function has accomplished. In this case, when the function returns the square root of **x** has been written.

Another Example

```
bool is_vowel( char letter )
//  Precondition:  letter is an uppercase or
//  lowercase letter (in the range 'A' ... 'Z' or 'a' ... 'z') .
//  Postcondition:  The value returned by the
//  function is true if letter is a vowel;
//  otherwise the value returned by the function is
//  false.
{

...

}
```


Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel( 'Z' );
is_vowel( '?' );
```

Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel( 'Z' );
is_vowel( '?' );
```

true

false

Nobody knows, because the precondition has been violated.

Another Example

What values will be returned by these function calls ?

```
is_vowel( 'A' );
is_vowel( 'Z' );
is_vowel( '?' );
```

Violating the precondition might even crash the computer.

Always make sure the precondition is valid . . .

- The programmer who calls the function is responsible for **ensuring that the precondition is valid** when the functi… called.

. . . so the postcondition becomes true at the function's end.

- The programmer who writes the function counts on the precondition being valid, and **ensures that the postcondition becomes true** at the function's end.

A Quiz

Suppose that you call a function, and you neglect to make sure that the precondition is valid.
Who is responsible if this inadvertently causes a 40-day flood or other disaster?

- You
- The programmer who wrote that torrential function
- Noah

A Quiz

Suppose that you call a function, and you neglect to make sure that the precondition is valid.
Who is responsible if this inadvertently causes a 40-day flood or other disaster?

- You

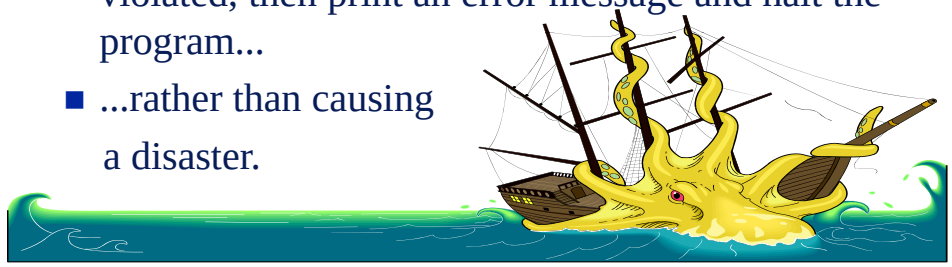
The programmer who calls a function is responsible for ensuring that the precondition is valid.

On the other hand, careful programmers also follow these rules:

- When you write a function, you should make every effort to detect when a precondition has been violated.

Is detection always possible?

- If you detect that a precondition has been violated, then print an error message and halt the program...
- ...rather than causing a disaster.



Example

```
void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
{
    assert(x >= 0);

    . . .

}
```

The assert function (described in Section 1.1) is useful for detecting violations of a precondition.

Advantages of Using Preconditions and Postconditions

- Succinctly describes the behavior of a function...
- ... without cluttering up your thinking with details of how the function works.
- At a later point, you may reimplement the function in a new way...
- ...but programs (which only depend on the precondition/postcondition) will still work with no changes.

Testing and Correcting Programs

- Program testing occurs when…
  - ...you run a program and observe its behavior
- Program testing is a broad and complex topic…
  - ...but it fundamentally involves the construction of a set of inputs designed to exercise a particular functionality of a program
  - …with the intention to discover faults

- Testing ***cannot be used to prove absence of faults***
- Testing ***can only be used to prove presence of faults***
- ***Why ??***

Choosing Test Data

- Good test input data needs to have two properties
  - Its correct output must be known
  - It should have high potential for exposing software faults
- What good is running a test case…
  - ...without knowing how to tell if it finds a fault?
  - ...knowing that it has little potential finding faults?

Testing Terminology

- Human beings (programmers) make ***errors***
- Errors manifest themselves in code as ***faults***
- A ***failure*** occurs when a program executes code that contains a fault
  - A failure is an event that occurs while the program is running
  - Code that never runs never fails

Boundary Value Testing

- Finding test inputs…
  - ...that are likely to expose software faults…
  - ...is a difficult proposition
- In commercial software development…
  - ...there is but limited time with which to test software
- One simplistic approach…
  - …to finding test inputs that might expose faults…
  - …is ***boundary value testing***

Boundary Value Testing

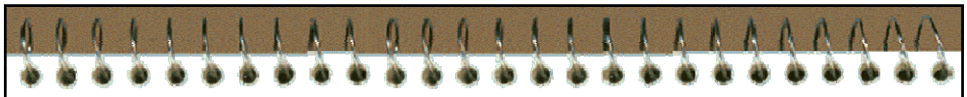
- A boundary value of a problem…
  - ...is an input that is one step away from a different type of behavior
- Consider the precondition `0 <= hour <= 23`
  - Four boundary values for **`hour`** are…
    - **-1** → one step away from being valid
    - **0** → one step away from being invalid
    - **23** → one step away from being invalid
    - **24** → one step away from being valid
  - If the program behaves differently for morning and afternoon hours, **11** and **12** are also boundary values

Fully Exercising Code

- Another testing strategy is to establish some level of code coverage, such as
  - All *paths* through the code must be followed
  - All *statements* must be executed (each at least once)
  - All *decisions* must be covered
  - All *combinations of predicate values* must be covered
    - `if ( (a > b) || (a > c) )`

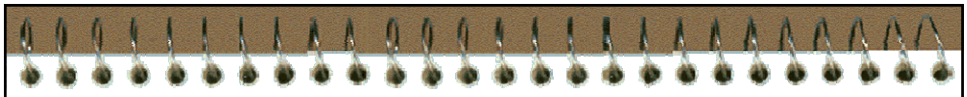
Code Coverage Tools

- Code coverage tools exist…
  - ...to help monitor the testing of software
- These tools can provide a listing…
  - ...that indicates which decisions haven't been tested
  - (or which result has not been tested)
- They also show…
  - ...which statements haven't been exercised by some test
- Some can optimize a test suite



Code Profilers

- **A code profiler…**
  - is a program for assessing the run-time efficiency of a program
- **It measures…**
  - how much execution time is spent within various functions
  - (or even on specific statements)
- **This can enable the programmer…**
  - to focus performance improvements on time consuming code
- **A typical profiler generates a listing indicating…**
  - how often each statement of a program was executed
- **As a side benefit, we can use such a listing…**
  - to help us spot parts of our program that were not tested



Debuggers

- Developing a test case...
  - ...that exposes a fault in the code…
  - ...is all well and good…
  - ...but the code is still faulty (and needs to be fixed)
- Debuggers are tools…
  - ...that allow you to observe the execution of the program in hopes of finding where things go astray
  - …that allow you to watch variable values and step through the program slowly

Correcting Code

- Determine exactly why a test case fails and limit changes to corrections of known errors
  - Avoid the temptation to start changing suspicious code on the hope that the change "might work better"
- Once a correction has been made, retest the code
  - The correction could have introduced a new fault
  - The correction may have made it possible for an additional, hidden fault to be encountered

Readings

- Chapter 1 of textbook
  - Section 1.1
  - Section 1.3