Data Structures in C++

Chapter 3

Tim Budd

Oregon State University Corvallis, Oregon USA

Outline – Chapter 3

Analytical Tools

- Introduction to Algorithms
- Recipes as Algorithms
- Analyzing Computer Algorithms
- Specification of Input
- Describing Result
- Proving Termination

Algorithms

An *algorithm* is a set of instructions used to solve a specific problem.

Synonyms

- *process*
- *method*
- *technique*
- *procedure*
- *routine*
- *recipe*

Properties of Algorithms

In order to be useful, an algorithm must have the following properties.

- **accurate specification of the input**.

  Under what conditions is the algorithm expected to give the correct answers.

- **definiteness of each instruction**

  Each instruction must be clear and unambiguous.

- **correctness**.

  The algorithm must correctly solve a problem.

- **termination**

  The algorithm must eventually halt for all legal input values.

- **description of the result or effect**

It must be clear what the algorithm is intended to do.

Recipes as Algorithms

Probably the most common form of algorithm is the recipe.

Lemon Soufflé

1 envelope unflavored gelatin

$\frac{1}{4}$ Cup cold water

6 Egg yolks

1 Cup sugar

$\frac{2}{3}$ Cup Lemon juice

1 Tablespoon grated Lemon rind

4 Egg whites

$1\frac{1}{2}$ Cup heavy cream

1. Soften the gelatin in water. Beat egg yokes and sugar until thick and light. Stir in lemon juice and cook over low heat, beating steadily with a wisk until thickened and hot but not boiling (10-15 minutes).

2. Pour mixture into large bowl, mix in gelatin until dissolved then add lemon rind. Stir occasionally until cool.

3. Beat egg whites until stiff but not dry. Fold into lemon mixture, then whip the cream and fold in. Pour into a two-quart soufflé dish (or large bowl) and refrigerate at least 12 hours.

Evaluation of Recipe as Algorithm

- **input**.

  Ingredients are listed.

- **result**.

  Indicated by the name.

- **correctness**.

  The proof of the pudding is in the eating.

- **termination**.

  What if it never boils? or never becomes light?

- **definiteness**.

  What is “light”? How long should it be whipped?

Computer Algorithms

How do computer algorithms differ from instructions for human beings with regards to

- description of input
- specification of result
- time to execute (some here, more in Chapter 4)
- correctness (more in Chapter 5)
- instruction precision

Specification of Input

Two most widely used techniques

- Type declarations for argument values
- comments associated with procedure heading

```
int min (int a, int b)
    // return smaller of two integer arguments
{
    int smallest;

    if (a < b)
        smallest = a;
    else
        smallest = b;
    return smallest;
}
```

Conditions that are not types

Occasionally input conditions are not types, but can nevertheless be checked at run-time, using an `assert` statement.

```
char digitChar (unsigned int val)
    // return the character equivalent for
    // the integer argument value
    // which must be between zero and nine
{
        // make sure value is in proper range
    assert (val < 10);

    switch (val) {
        case 0: return '0';
        case 1: return '1';
            ...
        case 9: return '9';
        }
}
```

Simpler Version

By noting some characteristics of ASCII and the fact that characters are just small integers, we can simplify to the following:

```
char digitChar (unsigned int val)
    // return the character equivalent for
    // the integer argument value
    // which must be between zero and nine
{
    assert (val < 10);
    return '0' + val;
}
```

Some input conditions cannot be checked

Some input conditions cannot be checked, even with assert.

```
double power (double base, unsigned int n)
    // return the value yielded by
    // raising the double precision base to
    // the integer exponent
    // assumes floating point
    // overflow does not occur
{
        // initial result to base raised to zero
    double result = 1.0;

        // raise base to new powers
    for (unsigned int i = 0; i < n; i++) {
        result *= base;
        }
    return result;
}
```

Describing the Result

Similarly, the result of executing an algorithm is often provided

- by the return type of a function
- by comments

```
int min (int a, int b)
    // return the smaller of two arguments
{
    ...
}
```

Side Effects

Sometimes algorithms are not executed for their result, but for a *side effect.* Common side effects:

- Printing a result
- Setting or modifying a value in a variable (global variable, instance variable)
- Copying information to a file
- Many others

These should always be documented in a comment, since they are not immediately obvious.

Printing Integers

Problem - print a positive integer value on the output stream.

General problem solving techniques

- Reduce task to a sequence of steps (we saw this in the Lemon Soufflé recipe), or
- Reduce a task by considering various cases

Can write a recursive procedure to handle this.

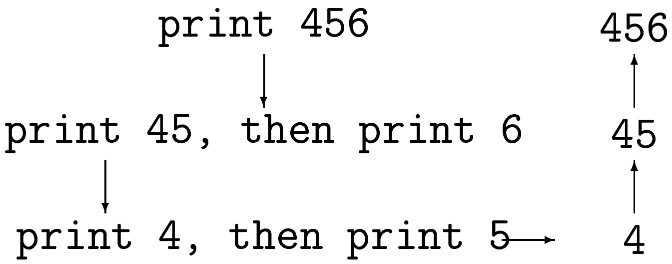
Printing Integer Code

```
void printUnsigned (unsigned int val)
    // print the character representation of
    // the unsigned argument value
{
    if (val < 10)
        printChar (digitChar(val));
    else {
            // print high order part
        printUnsigned (val / 10);
            // print last character
        printChar (digitChar(val % 10));
        }
}
```

It is relatively easy to see the input conditions, the fact that the algorithm is definite. What remains is showing termination, correctness, and execution time.

General Picture

The following illustrates the steps involved in printing the integer value 456.



Allocating space for Parameters

When pass by value is used as a parameter passing mechanism, each new recursive call gets a *different* copy of each parameter value and each local variables.

The sequence of function returns is the inverse of the sequence of function calls. Each invocation uses its own values.

Instruction Precision

Note difference in level of precision required for instructions to be given to a human being and instructions to be given to a computer:

"go to the store and buy me something to make lunch"

For a human, emphasis on the objective (the what), for a computer, emphasis on the process (the how). Must be specified in great detail.

Time to Execute

We will divide the discussion of analyzing execution time into two parts:

- Proving termination at all
- Characterizing running time (more in next chapter).

Proving Termination

In order to prove termination, find a property or value that possesses the following three characteristics:

- Can be placed in one-to-one correspondence with the integers.
- Is non-negative.
- Decreases as the algorithm executes.

Note we need all three characteristics.

Loops are usually easy

```
double power (double base, unsigned int n)
    // return the value yielded by
    // raising the double precision base to
    // the integer exponent
    // assumes floating point
    // overflow does not occur
{
        // initial result to base raised to zero
    double result = 1.0;

        // raise base to new powers
    for (unsigned int i = 0; i < n; i++) {
        result *= base;
        }

    return result;
}
```

We can consider the quantity $n - i$, which decreases as i increases.

The value need not be in the algorithm

What value can we use to prove the termination of the following:

```
unsigned int gcd (unsigned int n, unsigned int m)
    // compute the greatest common divisor
    // of two positive integer values
{
    assert (n > 0 && m > 0);

    while (m != n) {
        if (n > m)
            n = n - m;
        else
            m = m - n;
        }
    return n;
}
```

Proving Termination of Recursive Algorithms

If each step of the recursive algorithm can be proved to terminate, the proving termination of the entire procedure involves bounding the number of recursive calls.

In the case of the integer printing algorithm, we note that the magnitude of the number is reduced on each step. Note that the number of recursive calls is equal to the number of digits of the original value.

Space Utilization

Sometimes two or more algorithms differ in the amount of space (that is, memory) they use. For example, some sorting algorithms copy values into a new data structure, then copy them back. Others work "in place" and thus use far less memory.

Sometimes improvements in speed can come at a cost of more space. This is termed a time/space tradeoff. We will see examples of this later.

Recursive Algorithms

A procedure that invokes itself is termed recursive.

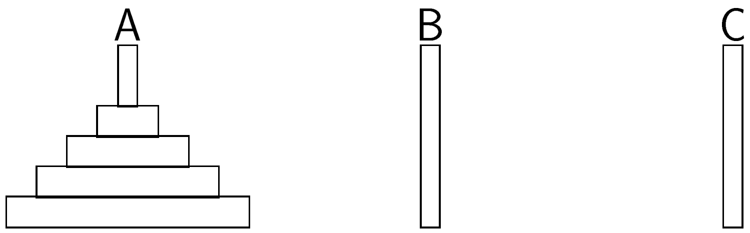
Recursive algorithms must always have

- A base case, that handles at least one condition without recursion
- An inductive case, that "reduces" a problem to another problem of the *same form* but *smaller* inputs.

Difficulty is finding out how to do the reduction.

Sometimes it helps to imagine you have solved the problem a certain distance, and you are in the middle of execution.

Towers of Hanoi puzzle



Only two possible first steps, but which one is the correct one to use?

Approaches to Problem Solving we have Seen

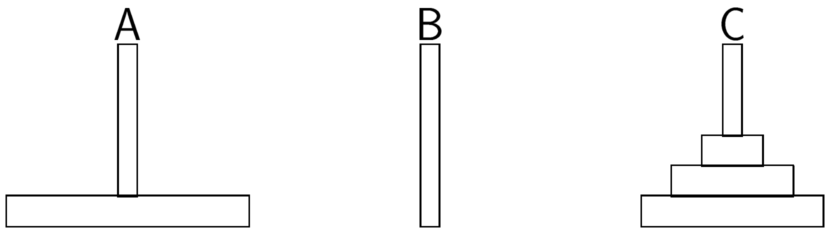
We have seen the following approaches to problem solving:

- Determine the first step, then go on. (Stepwise refinement. Produces a series of *sequential* steps.)
- Break down into cases. Handle each separately (Produces a *conditional* statement).

Neither seem applicable here. How about starting in the middle, and working both forwards and backwards.

Imagine You have Solved Problem a Bit Already

Imagine you are in the middle of execution:



Provides the Key

This picture gives us the key to figuring out how to take the task and “reduce” it to a similar problem with “smaller” size. Let the “size” be the number of disks to move, and let the argument values represent source pole, target pole, and temporary pole.

To move N disks from a to b using c:

- Move $N - 1$ disks from a to c using b
- Move 1 disk from a to b
- Move $N - 1$ disks from c to b using a

First cut at Solution

```
void Hanoi (int n, char a, char b, char c)
   // move n disks from a to b, using c
{
      // first move all but last disk to tower c
   Hanoi (n-1, a, c, b);
      // then move one disk from a to b
   cout << "move disk from tower " << a <<
" to " << b << endl;
      // then move all disks from c back to b
   Hanoi (n-1, c, b, a);
}
```

Opps. Never terminates. Need a base case.

Solution with Base Case Included

```
void Hanoi (int n, char a, char b, char c)
   // move n disks from a to b, using c
{
   if (n == 1) {
         // can move smallest disk directly
      cout << "move disk from tower " << a
<< " to " << b << endl;
      }
   else {
         // first move all but last disk to
tower c
      Hanoi (n-1, a, c, b);
         // then move one disk from a to b
      cout << "move disk from tower " << a
<< " to " << b << endl;
         // then move all disks from c back to
b
      Hanoi (n-1, c, b, a);
      }
}
```