Solving Complex Problems

Review

- A subroutine is a set of instructions to perform a particular computation
  - used to solve subproblems of more complex problems
  - used for computations performed more than once in a single program or that are required in many different programs
  - you only need to write it once
- Subroutines have names, zero or more parameters, return types, and a body of statements that perform the actual computations

Syntax

- The return-type specifies the data type of the output
- The parameter-list is a comma-separated list of the input variables and their types
  - each list item has the form input-type variable-name
- A program refers to the subroutine by subroutine-name
- The body consists of the statements between { }

Add two integers

```c
int add(int x, int y)
{
    return x+y;
}
```

Minimum of two integers

```c
int minimum(int x, int y)
{
    if(x < y)
        return x;
    else
        return y;
}
```

Sum of positive integers

```c
int sum_integers(int n)
{
    int k;
    int sum=0;
    for(k=1; k<=n; k++)
    {
        sum=sum+k;
    }
    return sum;
}
```
Factorial

```c
int factorial(int n) {
    int k;
    int fn=1;
    for(k=2; k<=n; k++) {
        fn=fn*k;
    }
    return fn;
}
```

What about harder problems?

- Examples so far: add, minimum, sum, factorial
  - straightforward computation
  - small number of intermediate variables
  - single control structure
- How do we design algorithms for more complicated problems?
  - complex computation
  - many intermediate variables
  - multiple control structures
- Often difficult to visualize all the details of a complete solution to a problem

Top-down design

- Top-down design
  - original problem is divided into simpler independent subproblems
  - successive refinement: these subproblems may require further division into smaller subproblems
  - continue until all subproblems can be easily solved or you already have a method to solve them

Solving complex problems

- Divide complex problem into subproblems
- For each subproblem
  - analyze the problem
    - do we already have a subroutine to solve it?
    - can we easily design an algorithm to solve it?
    - can it be divided further?
  - solve the problem
- Combine solutions of all subproblems to solve the original problem

Minimum of three integers

Problem: Find the minimum of three integers

Minimum of three integers

- Analyze the problem
  - Inputs
    - \(x\) first integer
    - \(y\) second integer
    - \(z\) third integer
  - Output
    - \(\text{min}_{xyz}\) minimum of \(x, y,\) and \(z\)
  - How do we find the minimum?
    - we already have a subroutine to find the minimum of two integers ...
**Top-down design**

Compute minimum of $x$, $y$, and $z$

Compute minimum of $x$ and $y$

Compute minimum of solution to SUBPROBLEM1 and $z$

**Calling subroutines**

- A subroutine call statement has the following syntax:

  \[ \text{subroutine-name}( \text{parameters} ); \]

- Calling a subroutine executes the statements in the body of the called subroutine using the specified parameters
- \text{parameters} is a comma-separated list of input values
  - the input values must be of the same type as those in the subroutine’s parameter list
- The subroutine call statement may be part of an assignment statement or an return statement

**Minimum of three integers**

- We can call our \text{minimum} subroutine twice to compute the minimum of three integers

  ```c
  int min_xy, min_xyz;
  min_xy = minimum(x,y);
  min_xyz = minimum(min_xy,z);
  ```

**Minimum of three integers - subroutine**

```c
int min3(int x, int y, int z)
{
    int min_xy, min_xyz;
    min_xy = minimum(x,y);
    min_xyz = minimum(min_xy,z);
    return min_xyz;
}
```

**Combination**

**Problem: Compute $C(n,k)$**

**Definition: combination**

- A combination $C(n,k)$ is the number of ways to select a group of $k$ objects from a population of $n$ distinct objects

  \[ C(n,k) = \frac{n!}{k!(n-k)!} \]

  where $n!$ is the factorial function

  \[ n! = n*(n-1)*(n-2) \ldots 3*2*1 \]
Top-down design

\[
\text{Compute } C(n, k) = \frac{n!}{k!(n-k)!}
\]

combination - subroutine

We can call our factorial subroutine three times to compute \( C(n, k) \)

\[
\text{int combination(int n, int k) }
\]
\[
\{ 
\text{ return factorial(n) /} \\
\text{ (factorial(k)*factorial(n-k));}
\}
\]

Exercises

1. We have a subroutine `sum_integers` that computes the sum of the first \( n \) positive integers

   Write a subroutine named `sum_between` to compute the inclusive sum between two integers by making two calls to `sum_integers`

2. Write a subroutine named `min4` to compute the minimum of four numbers