CS 102
Introduction to Programming Using C++

Chapter 5

Functions-Part 2

Homework

- Written homework
- R5.1, 2, 4, 5, 7, 8, 9, 12, 17, 20

- Programs
- p. 237, choose one of P5.3, 4, 5
- Also, choose one of P5.27-30

Changing a Parameter in a Function

- C++ is call by value
- So, when a variable is passed to a function
  - The data is copied
  - Only the copy is given to the function
- This means that a function cannot change its arguments
- What if you want to change an argument?

An Example:
What Is Printed Here?

```
int main ()
{
    int i = 100;
    change (i);
    cout << i;
}
```

```
void change (int i)
{
    i = 10;
}
```

Using Call-by-Reference

- You cannot change a function’s arguments
- If you want to change an argument, you use a reference parameter
- A reference parameter is the address (in memory) of a variable
- To cause a function to use a reference parameter, you put & before the parameter

Changing the Value of a Parameter Using Call-By-Reference

```
int main ()
{
    int i = 100;
    change (i);
    cout << i;
}
```

```
void change (int& i)
{
    i = 10;
}
```

Using References vs. Returning Values

- For the most part, you should use return values instead of references
- One reason is that you cannot call the change() function with a constant value
  - This will give compile-time error

Rewriting the Code to Return a Value

```
int main ()
{
    int i = 100;
    int new_i = change (i);
    cout <<new_i;
}
```

```
void change (int& i)
{
        return 10;
}
```

Constant References

```
string duplicate_string (string str)
{
   return str + " " + str;
}
```

- Here, if str is long, copying it would take a long time
  - Remember, C++ is call-by-value
- We can change the function header to

  string duplicate_string (const string& str)
- Now, you get the efficiency of a reference, but the performance of a variable

Recursion

- Sometimes it’s convenient to create a function that calls itself
- Why would you want to do this?
  - It might be easier to program it
- A good example is the math function x!
  - Here, factorial (x) [C++ notation] denotes x! [math notation]
- 8! means 8x7x6x5x4x3x2x1
- 9! means 9x8x7x6x5x4x3x2x1
- 10! means 10x9x8x7x6x5x4x3x2x1

Re-examining the Factorial Function

- Wait! The end of 10! is 9!
  - And the end of 9! is 8!
    - It looks like 9! is just 9 x 8!
  - We notice that 10! is just 10 x 9! too
  - And we keep going
- So, we can say n! = n x (n-1)!
- In C++

```
factorial (n) = n * factorial (n-1);
```

Programming Factorials

- The code is (mostly)

```
int factorial (int n)
{
    return n * factorial (n-1);
}
```

- It’s that easy?
- Yes!

Another Example: Fibonacci Numbers

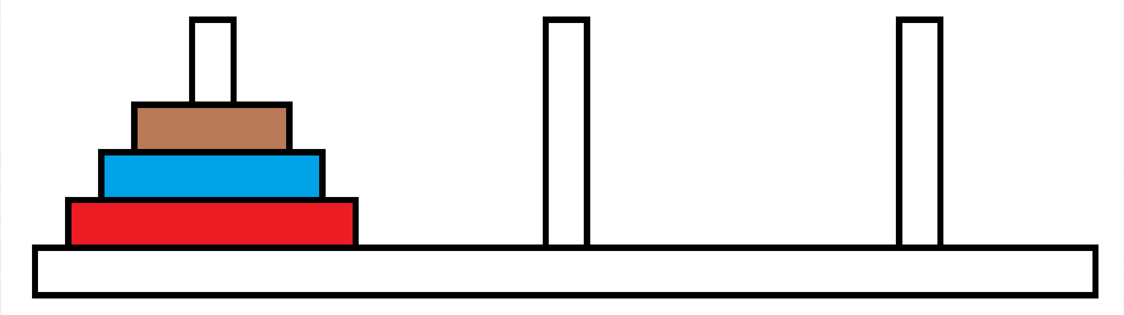
- The Fibonacci Numbers are a list (sequence) of numbers
  - The first two Fibonacci numbers are both 1
  - After that, to get the next Fibonacci number, add the two before it
- For example, the first seven numbers in the list are
  1, 1, 2, 3, 5, 8, 13
- This is recursive!
- The code is (mostly)

```
int fibonacci (int n)
{
   return fibonacci (n-1) + fibonacci (n-2);
}
```

- Try to write that code without recursion

A Third Example: The Towers of Hanoi

- This problem is not as mathematical as the first two
- Here is a picture of the towers



- There are many interactive websites, but here is a cute one
- https://www.mindgames.com/game/Tower+of+Hanoi

Moving the Disks

- There are several disks on each tower
- In this case, there are three disks
  - I have colored them red, blue, and brown
- Your job is to move them to the other end
- This seems simple enough

The Rules

- There are only two rules
  1. You can only move one disk at a time
  2. You cannot put a disk on top of a smaller disk
- This is much harder than it looks!

The Solution

- Suppose there are 10 disks
- Then, to move all ten disks to the other end
  - Move the top 9 disks to the middle post
  - Move the remaining disk to the end post
  - Move the 9 disks from the middle post to the end post
    - This puts those disks on top of the disk you just moved
- Wow! We have moved the disks

The Solution?

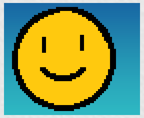
- Wait a minute…
- You can only move one disk at a time
- How did you move that stack of nine disks?
  - You moved them twice, even!
- You used recursion!

The Code

- So the pseudocode is (mostly)

```
void move_disks (int n, int start_pole,
                 int spare_pole, int end_pole)
{
    move_disks (n-1, start_pole, end_pole, spre_pole);
    Move top disk from start_pole to end_pole
    move_disks (n-1, spare_pole, end_pole, start_pole);
}
```

Recursion in General

- Recursion means
  - You turn the problem into a smaller version of the same problem
  - Then you call the same function to complete the solution
- In recursion, a function calls itself
  - Of course, when it calls itself, it has to present a smaller problem
  - factorial (n) = factorial (n)   is not useful 

A Problem with Recursion

- Let's go back to factorials
- Let's calculate 3!
- Using the code, 3! = 3 x 2!

  =3 x (2 x 1!)

  =3 x (2 x (1 x 0!))

  =3 x (2 x (1 x (0 x (-1)!)))
- Does this ever stop? No!

A Stopping Point

- For recursive code, we always need to include a stopping condition
  - This is called the base case
- The actual code for factorials with the base case is

```
int factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial (n-1);
}
```

The Fibonacci Sequence with Base Cases

```
int fibonacci (int n)
{
    if  (n == 0)
        return 1;
    else if  (n == 1)
        return 1;
    else
        return fibonacci (n-1) + fibonacci (n-2);
}
```

The Tower of Hanoi with the Base Cases-Part 1

```
void move_disks (int n, int start_pole,
                        int spare_pole, int end_pole)
{
  if (n == 1)
     Move top disk from start_pole to end_pole
```

The Tower of Hanoi with the Base Cases-Part 2

```
else
 {
     move_disks (n-1, start_pole, end_pole, spre_pole);
    Move top disk from start_pole to end_pole
    move_disks (n-1, spare_pole, end_pole, start_pole);
 }
}
```

Designing a Recursive Program

- To use recursion, you must be trying to solve a recursive problem
  - That means you have to recognize that solving the problem involves solving a smaller version of the exact same problem
- You also need to find a stopping condition when coding

A Disadvantage of Recursion

- Recursion is much slower than solving the problem directly
- However, programming the direct solution might be a lot harder
- This is a trade-off to consider when using recursion
- The usual decision is to use recursion if it's appropriate
- For the factorial function, it's easy to code directly and more efficient
  - We always use the direct method

Factorial vs. Factorial

A recursive version

```
int factorial (int n)
{
  if (n == 1)
    return 1;
  else
    return n * factorial (n-1);
}
```

A non-recursive version

```
int factorial (int n)
{
  int product = 1;
  for (int i=1;  i<=n;  i++)
      product = product * i;
  return product;
}
```

The Compilation Process

- This is how the compilation process works (in words)
- The C++ preprocessor does several things, including substituting #include files
  - This requires that the preprocessor be able to find the #include files
  - The C++ preprocessor also handles compiler directives
- The output from the C++ preprocessor is sent to the C++ compiler

The Compilation Process, Part 2

- The C++ compiler translates “pure” C++ into machine code
  - This code is called object code
  - The object code has “holes” that allow for references to external routines

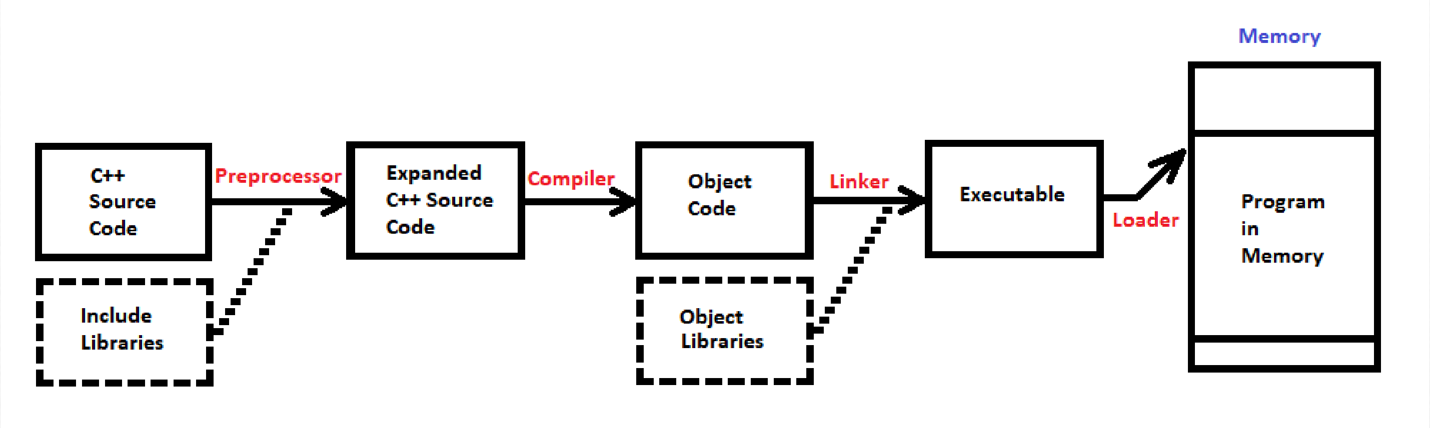
The Compilation Process, Part 3

- The linker then converts the object code into a program that can be run
  - We call this an executable
- This is a self-contained program that can run on its own

The Compilation Process, Part 4

- The loader loads the program into memory and then starts the program running
  - We say it transfers control to the program
- While the program is running the operating system “watches” it

A Picture of the Process



Questions?

- Are there any questions?