Functions (cont.)
Example: Prime Example with Functions

- Write a program that reads an integer \( n \) from the user input and computes the smallest prime number greater than \( n \).

```
Read n from user
n++
Is n prime?
Yes
print n
No
```

Example: Prime Example with Functions
Example: Prime Example with Functions

Read n from user

n++

Is n prime?

Yes

print n

No

output:
- 1 if a is prime
- 0 if a is not prime

Example: Prime Example with Functions

isPrime()

(a positive number)
Prime Example with Functions

```c
#include<stdio.h>

/* A function check if a number is prime or not */

int isPrime(int a)
{
    // Code here
}

int main()
{
    // Code here
    return 0;
}

output:
- 1 if a is prime
- 0 if a is not prime
```
Prime Example with Functions

```c
#include<stdio.h>

/* A function check if a number is prime or not */

int isPrime(int a)
{
    
}

int main()
{
    int n = 0, num = 0;
    printf("Enter the number: \n");
    scanf("%d",&n);

    for ( num = n+1; ; num++ )
        if(isPrime(num))
        {
            printf("The smallest prime number greater than %d is %d\n",n, num);
            break;
        }

    return 0;
}
```
Prime Example with Functions

```c
#include <stdio.h>

int isPrime(int a)
{
    int flag;
    flag = 1;
    for (int i = 2; i < a; i++)
    {
        if (a % i == 0)
        {
            return 0; // the number is not prime
        }
    }
    return 1; // the number is prime
}

int main()
{
    int n = 0, num =0;
    printf("Enter the number: \n");
    scanf("%d", &n);

    for (num = n+1; num++)
    {
        if(isPrime(num))
        {
            printf("The smallest prime number greater than %d is %d\n",n, num );
        }
    }

    return 0;
}
```

if `return` statement is called, the rest of code is ignored!
Decoupling Body from Declaration
#include <stdio.h>

float percentage (float a, float b) {
    float p = a/b*100;
    return p;
}

int main () {
    float x = 5, y = 20;
    float val = percentage(x, y);
    printf("%f to %f is %f \%\n", x, y, percentage(x, y));
    printf("7 to 35 is %f \%\n", percentage(7, 35));
}

Decoupling Body from Declaration

#include <stdio.h>

double percentage (double a, double b);  // function declaration or prototype

int main () {
    double x = 5, y = 20;
    double val = percentage(x, y);
}

double percentage (double a, double b) {
    double p = a/b*100;
    return p;
}

Do not forget ; here.
#include <stdio.h>

double percentage (double, double);

int main () {
    double x = 5, y = 20;
    double val = percentage(x, y);
}

double percentage (double a, double b) {
    double p = a/b*100;
    return p;
}

Don’t need names in declaration, but better practice to keep them for readability
Function

- C program does not execute the statements in a function until the function is called (lazy or maybe polite :-)).

- When it is called, the program can send information to the function in the form of one or more arguments although it is not a mandatory.

- When the function finished processing, program returns to the same location which called the function.

- Argument is a program data needed by the function to perform its task.
Function

When a function is called **execution begins** at the start of the function body and **terminates** (returns to the calling program) when a **return** statement is encountered or when execution reaches the closing braces (`{}`).
In C, the "main" function is treated the same as every function, it has a return type (and in some cases accepts inputs via parameters). The only difference is that the main function is "called" by the operating system when the user runs the program. Thus the main function is always the first code executed when a program starts.

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    // this is the body of the function (lots of code can go here)
    return 0;
}
```

```bash
c:\> ./mycode.exe
```
Recursion
Recursion

• In our world, there's an interesting pattern for many problems: a big problem contains a smaller problem with exactly the same structure
Recursion

1 + 2 + 3 + ... + n = n + [1 + 2 + 3 + .. + (n-1)]

\[\text{sum (1, n)} = n + \text{sum (1, n-1)}\]
\[= n + (n-1) + \text{sum (1, n-2)}\]
\[= n + (n-1) + (n-2) + \text{sum (1, n-3)}\]
\[= n + (n-1) + (n-2) + (n-3) + \text{sum (1, n-4)}\]
\[= ...\]
\[= n + (n-1) + (n-2) + (n-3) + (n-4) + ... + \text{sum (1, 1)}\]

\[\text{sum (1, 1)} = 1\]
Recursion

• A big problem contains a smaller problem with exactly the same structure

\[ a^n = a \times a \times a \times a \times a \times \ldots \times a \]

\[
\text{power}(a, n) = a \times \text{power}(a, n-1)
\]
\[
= a \times a \times \text{power}(a, n-2)
\]
\[
= a \times a \times a \times \text{power}(a, n-3)
\]
\[
= a \times a \times a \times a \times \text{power}(a, n-4)
\]
\[
= \ldots
\]
\[
= a \times a \times a \times a \times \ldots \times \text{power}(a, 1)
\]

\[
\text{power}(a, 1) = a
\]
Recursion

• A big problem contains a smaller problem with exactly the same structure

\[ 1 \times 2 \times 3 \times \ldots \times n = n \times [1 \times 2 \times 3 \times \ldots \times (n-1)] \]

\[
\text{fact}(n) = n \times \text{fact}(n-1) \\
= n \times (n-1) \times \text{fact}(n-2) \\
= n \times (n-1) \times (n-2) \times \text{fact}(n-3) \\
= n \times (n-1) \times (n-2) \times (n-3) \times \text{fact}(n-4) \\
= \ldots \\
= n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times \text{fact}(1) \\
\]

\[
\text{fact}(1) = 1
\]
Recursive Function:— a function that calls itself (directly or indirectly)

Each recursive call is made with a new, independent set of arguments (previous calls are suspended)

Allows very simple programs for very complex problems

Such a pattern makes it (surprisingly) easy for writing a computer program to do it
Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

Recursion is a technique that solves a problem by solving a smaller problem of the same type.

Some problems are too hard to solve without recursion.
Content of a Recursive Method

Base case(s):  \( \text{fact}(1) = 1 \)

- Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
- Every possible chain of recursive calls must eventually reach a base case.

Recursive calls:  \( \text{fact}(n) = n \times \text{fact}(n-1) \)

- Calls to the current method.
- Each recursive call should be defined so that it makes progress towards a base case.
Recursive Programming

- **Know** how to solve a problem immediately for “small” of bases cases

- **Know** to break up the problem into smaller chunks until you reach
Content of a Recursive Function

```c
return_type recursive_function (type1 parameter1, …) {
    1: if( base case holds)
        return the base result;
    2: find the parameters for subproblem(s);
    3: call the recursive_function with new parameters and get results;
    4: combine the results of subproblem(s) and return final result;
}
```
Factorial: Iterative

- For example, factorial:
  \[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]

```c
#include <stdio.h>

int fact(int n)
{
    int i, result = 1;
    for(i=1; i<=n; i++)
        result *= i;
    return result;
}

int main()
{
    printf("The fact of 5 is: %d\n", fact(5));
    return 0;
}
```
Recursion

• A big problem contains a smaller problem with exactly the same structure

\[ 1 \times 2 \times 3 \times \ldots \times n = n \times [1 \times 2 \times 3 \times \ldots \times (n-1)] \]

\[ \text{fact}(n) = n \times \text{fact}(n-1) \]
\[ = n \times (n-1) \times \text{fact}(n-2) \]
\[ = n \times (n-1) \times (n-2) \times \text{fact}(n-3) \]
\[ = n \times (n-1) \times (n-2) \times (n-3) \times \text{fact}(n-4) \]
\[ = \ldots \]
\[ = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times \text{fact}(1) \]

\[ \text{fact}(1) = 1 \]
Factorial: Recursive

- For example, factorial:

\[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]

\[ = n \times (n - 1)! \]

```c
#include <stdio.h>

int fact(int n)
{
    if (n <= 1)
        return 1;
    return n*fact(n-1);
}

int main()
{
    printf("The fact of 5 is: %d\n", fact(5));
    return 0;
}
```
Factorial: Recursive

– For example, factorial:

\[ n! = n \times (n - 1) \times (n - 2) \times \ldots \times 1 \]

\[ = n \times (n - 1)! \]

```c
#include <stdio.h>

int fact(int n)
{
    if (n <= 1)
        return 1;
    return n*fact(n-1);
}

int main()
{
    printf("The fact of 5 is: %d\n", fact(5));
    return 0;
}
```
Recursion

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main()
{
    int x = fact(4);
    x = 4 * fact(3);
    x = 4 * ( 3 * fact(2) );
    x = 4 * ( 3 * ( 2 * fact(1) ) );
    x = 4 * ( 3 * ( 2 * 1 ) );
}
```

**Termination Condition**
Recursion

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n * fact(n-1);
}

int main() {
    int x = fact(4);
}
```
Recursion

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    int x = fact(4);
}
```
Recursion

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main()
{
    int x = fact(4);
}
```
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main()
{
    int x = fact(4);
}

Recursion
Recursion

int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main()
{
    int x = fact(4);
}
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    int x = fact(4);
}
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    int x = fact(4);
}
int fact(int n) {
    if (n <= 1)
        return 1;
    else
        return n*fact(n-1);
}

int main() {
    int x = fact(4);
}

int fact(int n) {
    return n*fact(n-1);
}

int x = fact(4);
int x = 4*fact(3);
int x = 4*(3*fact(2));
int x = 4*(3*(2*fact(1)));
int x = 4*(3*(2*1*(fact(0))));
int x = 4*(3*(2*1*(0*(fact(-1)))));
int x = 4*(3*(2*1*(0*(-1*(fact(-2))))));
... ... ...

What if we remove the termination condition?
- the function will call itself infinite number of times.
Example: Fibonacci Numbers

- Write a recursive function that computes the Fibonacci sequence:

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

\[
f(n) = \begin{cases} 
  0 & n = 0 \\
  1 & n = 1 \\
  f(n-1) + f(n-2) & n > 1 
\end{cases}
\]
Example: Fibonacci Numbers

```c
#include <stdio.h>

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

int main()
{
    int n;
    printf("Please neter the number: \n");
    scanf("%d", &n);
    printf("The %dth numbers is: %d\n", n, fibonacci(n));
    return 0;
}
```
Example: Fibonacci Numbers

```c
#include <stdio.h>

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

int main() {  
    int n;  
    printf("Please neter the number: \n");  
    scanf("%d", &n);  
    printf("The %dth numbers is: %d\n", n, fibonacci(n));  
    return 0;  
}
```
Example: Fibonacci Numbers

The function is called 15 times!
Example: Fibonacci Numbers

[cse220:>gcc fibo.cc -o fibo.exe
[cse220:>./fibo.exe
Please enter the number:
10
The 10th numbers is: 55

Example

• Write a recursive function that computes $x^n$

$$x^n = x \times x^{n-1}$$
Exercise: Decimal to Binary

Write a recursive method `printBinary` that accepts an integer and prints that number's representation in binary (base 2).

Example: `printBinary(7)` prints 111
Example: `printBinary(12)` prints 1100
Example: `printBinary(42)` prints 101010

Write the method recursively and without using any loops.
Move stack of disks from one peg to another
Move one disk at a time
Larger disk may never be on top of smaller disk