Function

DoCSE

Kathmandu University

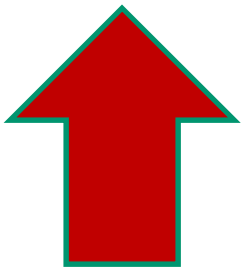
Introduction

- A C program is generally formed by a set of functions. These functions subsequently consist of many programming statements.
- By using functions, a large task can be broken down into smaller ones.
- Functions are important because of their reusability. That is, users can develop an application program based upon what others have done. They do not have to start from scratch.

Function is a set of collective commands.

Buy milk

- Walk to a MART
- Walk in
- Search for a bottle of milk
- Take out a wallet
- Take out money
- Give money to the cashier
- Walk out of the store



Function

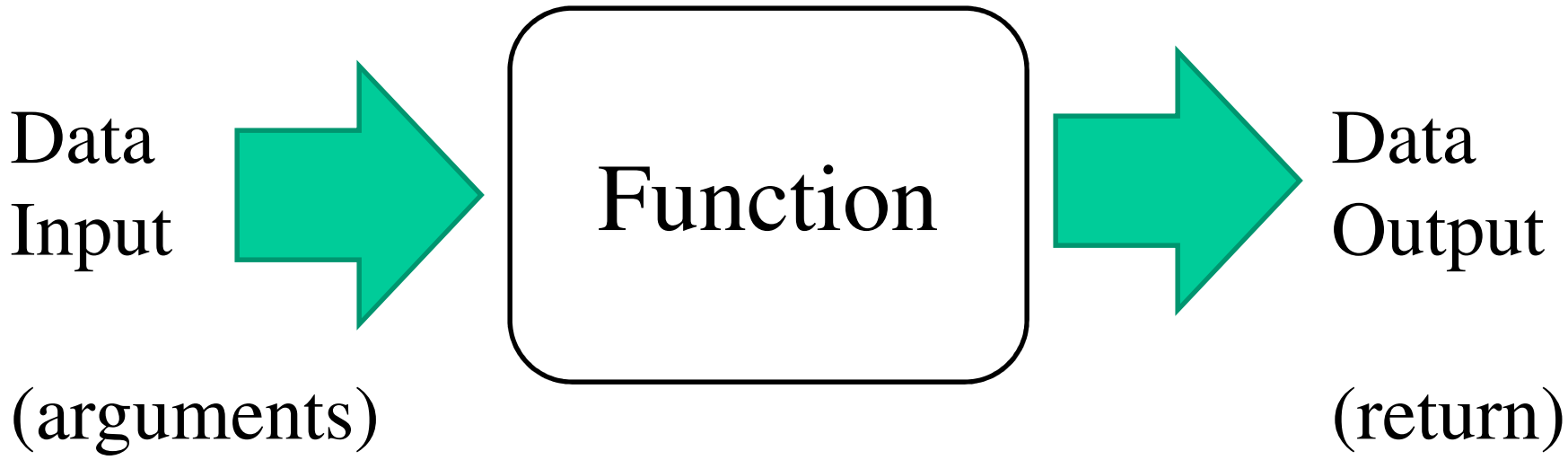
How to use function

```
BuyMilk( )
{
        :
}


main( )
{
        :
        BuyMilk( );
        :
}
```

Function Definition

Function call

In/Out Data of a Function



Note: Functions can have several arguments but only one return.

What are Functions?

- A function can be thought of as a mini-program, where you can pass- in information, execute a group of statements and return an optional value.
- A function is ***CALLED*** (or ***INVOKED***) when you need to branch off from the program flow and execute the group of statements within that function. Once the statements in the function are executed, program flow resumes from the place where you called your function.
- You've already encountered a few functions: main, printf and scanf.
- The main function is special, in the way that it's called automatically when the program starts.
- In C, and other programming languages, you can create your own functions. (user – defined function)

Function Features

Functions have 5 main features:

1. The ***DATA TYPE*** is the data type of the ***RETURN VALUE*** of the function.
2. The ***NAME*** is required as an identifier to the function, so that the computer knows which function is called. Naming of functions follows the same set of rules as the naming of variables.
3. Functions can take ***ARGUMENTS*** - a function might need extra information for it to work. Arguments are optional.
4. The function ***BODY*** is surrounded by curly brackets and contains the statements of the function.
5. The ***RETURN VALUE*** is the value that is passed back to the main program. Functions exit whenever you return a value. If the function does not return a value, we put the void keyword before the function name.

Why “Functions”?

- Allows a large program to be broken down into smaller, manageable, self contained parts
- Allows for *modularization* of a complex task
- *Functions* are the only way in which modularization can be achieved in C
- Avoids the need for repeated programming of the same instructions needed by various programs or parts of a program
- Decomposition of a large program through functions enhances *logical clarity*
- Programmers may build their own set of functions in a customized library
- Writing programs using functions allows the separation of machine dependent portions from others, enhancing *portability* of programs

The Function Body

- The function body is a set of statements enclosed between { }.
- These set of statements define the task carried out by the function.
- This function body, like any other compound statement, can contain any type of statements
  - assignment statement
  - compound statements
  - function calls
  - anything that is a valid statement
- The body must contain at least one return statement.
- The return statement helps in passing the single value result back to the calling program.

The *return* Statement

- The general form of the return statement is

```
return (expression);
```

- This contains the keyword return followed by an optional expression.
- The value of the expression is returned to the calling part of the program.
- Since an expression can result in only one value, a function can also return only one value.
- If the expression is omitted, the return statement does not transfer any value to the calling program, but just transfers control to the calling program.
- Such functions are defined as void.

```
void functionName()
 {
      statements;
      return;
 }
```

The *return* Statement (Contd…)

```
void functionName(int x, int y, int z)
 {
    if (x > y)
    {
    statements;
    return;
    else if (x > z)
    statements;
    return;
 }
```

- The above code fragment illustrates the use of return in void functions.
- The return statement causes the calling program to start execution from the statement immediately following the function call.
- A function body can contain one or more return statements.

Example

```
void display(void)
{
printf("Hello World");
return;
}

void main( )
{
display( );
}
```

```
void display(void);
main( )
{
display( );
}

void display(void)
{
printf("Hello World");
return;
}
```

Example 2

```
int add(int a, int b)
{
int z;
z=a+b;
return z;
}

void main( )
{
int x,y,k;
printf("Enter two values");
scanf("%d %d",&x,&y);
k=add(x,y);
printf("Sum=%d",k);
}
```

```
int add(int a, int b);

void main( )
{
int x,y,k;
printf("Enter two values");
scanf("%d %d",&a,&b);
k=add(a,b);
printf("Sum=%d",k);
}

int add(int a, int b)
{
int z;
z=a+b;
return z;
}
```

```
#include<stdio.h>

float SUM(int n);    /* function
     prototype or function declaration */

main( )                    // main function
{
int n;
float y;
printf("Number of input\n");
scanf("%d",&n);
y=SUM(n);     // function call
printf("Sum of %d is %f",n,y);
}
```

```
float SUM(int n) // function definition
{
int i;
float a, sum=0;
for(i=1;i<=n;++i)
              {
              scanf("%f",&a);
              sum=sum+a;
              }
return sum;
}
```

```
#include<stdio.h>

int factorial(int );

main( )
{
int n,r,combination;
printf("Value of n and r:");
scanf("%d%d",&n,&r);
combination=factorial(n)/(factor
    ial(n-r)*factorial(r));
printf("Combination:%d",comb
    ination);
}
```

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

Function Prototypes

- A function prototype is a declaration of a function that declares the return type and types of its parameters. For example,

```
double force(double t);
double accel(double t, double mu, double m);
```

- A function prototype contains no function definition.
- A function prototype can be declared at any lexical level, before and after its function definition.
- If the function is called before it is defined, **int** is assumed to be the return type of the function. Therefore, a function prototype is required if the return type of the function is not int and the function is called before its definition is processed, or if the function definition is located in a different file or in a library.

Function Definition:An Example

```
int max(int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
```

- A function called max is defined that returns an int.
- The function body is fairly simple as it just checks which of the variables in greater and returns that value

Invoking & Using Functions

- **A function can be invoked by specifying the name of the function followed by the list of arguments separated by commas.**
- **This is also called a *call to the function* or a *function call.***
- **If a function does not require any arguments, just write the name of the function with ( ) as in filler( );**
- **The function call can happen in a simple expression or part of a complex expression.**
- **The arguments in the function call are called as *actual arguments*.**
- **The arguments used in the function definition are called as *formal arguments.***
- **There is a one-to-one match of the list and type of the formal arguments with the actual arguments.**
- **The value of each actual parameter will be passed as input to the called function through the corresponding formal parameter.**
- **A function not returning anything can appear as a standalone statement such as printf(...);**

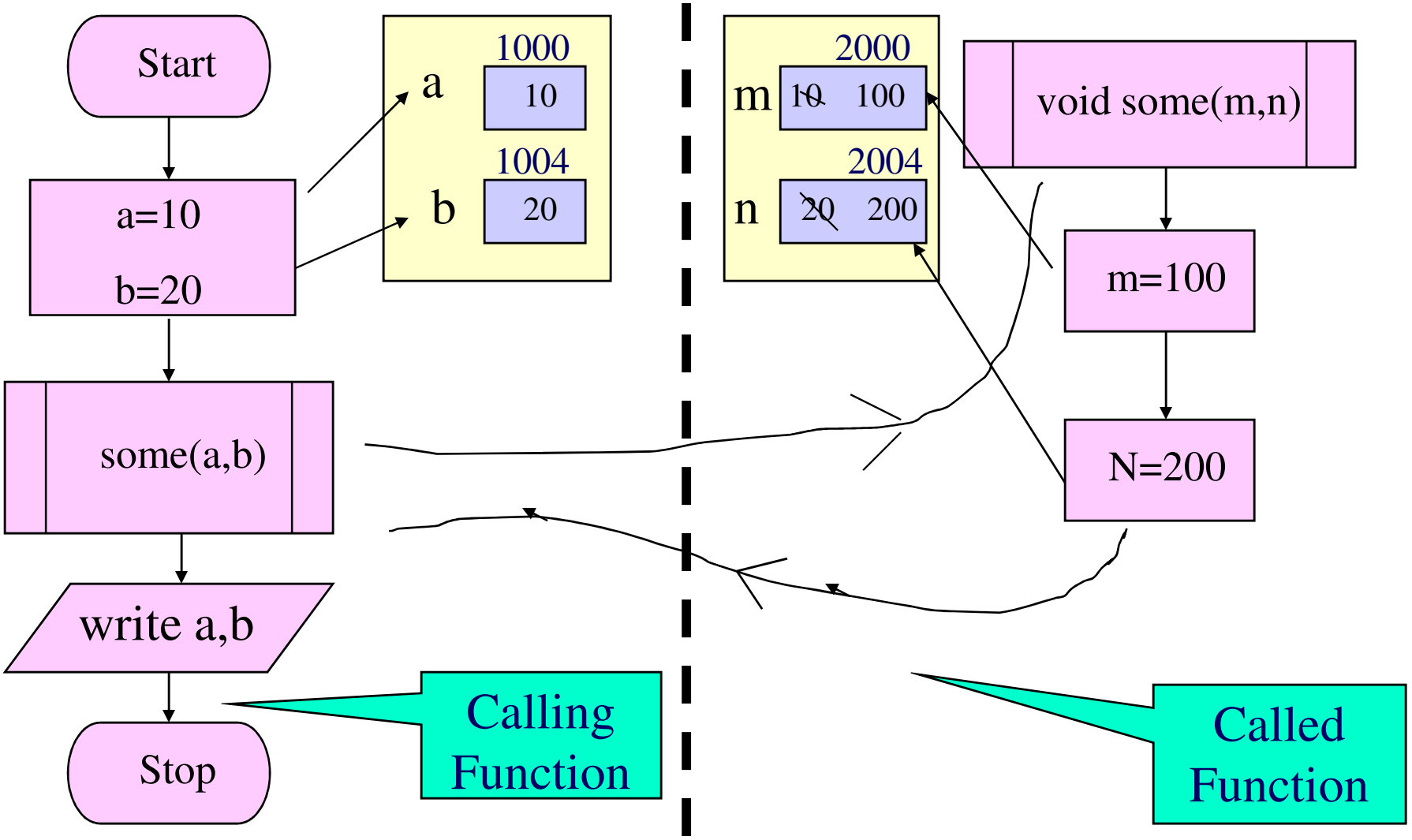
Call by Value

```
void some(int m,int n)
{
        m=100;
        n=200;
}
void main()
{
        int a,b;
        a=10;
        b=20;
        some(a,b);
        printf("a = %d and b = %d",a,b);

}
```

Called function

Calling function

Call by Value:Sequence of steps



Call by Value: Sequence of steps

1) Execution starts from the calling function.
2) After executing the statements **`a=10,b=20`** the values of 10 and 20 will be copied into memory locations()(Assume 1000 and 1004).
3) The function **`some()`** is invoked with two parameters **`a`** and **`b`**.
4) The control is now transferred to the called function **`some().`**
5) Memory is allocated for the ***formal parameters*** m and n(Assume 2000 and 2004).
6) When the statements **`m=100,n=200`** are executed, the earlier values of 10 and 20 are replaced with 100 and 200.
7) Control is transferred from the called function to the calling function to the point from where it has been invoked earlier.
8) ***The copy of actual parameters(formal parameters) is changed and actual parameters are un-affected.***
9) Execution is stopped.

Write a program to check whether a factorial of a given user input is primer number or not using function.

```
int fact(int);
int prime(int);
main( )
{
int value,z,s;
scanf("%d",&value);
z = fact(value);
s = prime(z);
if(s = = 2)
printf("%d is a prime numer",z);
else
printf("%d  is a not a prime numer",z);
}
```

```
int fact(int k)
{
?
}

int prime(int t)
{
?
}
```

Function Definitions

- A function can be defined in the form of

```
return_type function_name(argument declaration)
{
   statements
}
```

Example:

```
int addition(int a, int b)
   {
     int s;
     s = a + b;
     return s;
   }
main()
{
     int sum;
     sum = addition(3, 4);
     printf("sum = %d\n", sum);
}
```

The function definition has to appear before the real usage.
Otherwise, the compiler will not know the meaning of the function.

(Try swap the function definition to come after main().)

- The return type can be any valid type specifier.
- The **return** statement can be used to return a value from the called function to the calling function as in return expression;

  If necessary, the expression will be converted to the return type of the function. However, if the expression cannot be converted to the return type of the function according to the built-in data type conversion rules implicitly, it is a syntax error.

Example:

```
int func(void)
{
   double d;
   d = 4.6;
   return d; // OK: C type conversion, return 4
}
```

- If the return type is not **void**, a return statement is necessary at the end of the function. Otherwise, the default zero will be used as the return value.
- A calling function can freely ignore the return value.
- main() is also a function.

Example:

```
int func(int i){
   return i+1; // the same as 'return (i+1);'
}

int main() {
   int j;
   j = func(4);
   func(5);      // ignore the return value
   return 0;
}
```

- If the return type is **void**, the return statement is optional. However, no expression should follow `return`; otherwise, it is a syntax error.

Example:

```
void func(int i)
{
   if(i == 3)
   {
      printf("i is equal to 3 \n");
      return i;    // ERROR: return int
   }
   else if(i > 3)
   {
      printf("i is not equal to 3 \n");
      return;     // OK
   }
   i = -1;
     // return not necessary since return type is void
}

  int main(){
    func(2);
      return 0;
  }
```

- The data type of an actual argument of the calling function can be different from that of the formal argument of the called function as long as they are compatible. The value of an actual argument will be converted to the data type of its formal definition according to the built-in data conversion rules implicitly at the function interface stage.

Example:

```
int func1(int i)
 {  // argument type is int
    return 2*i;
}

int main(){
  func1(5);          // OK
  func1(5.0);        // OK, 5.0 converted to 5
  return 0;
}
```

Example Program 1:

Use a function to calculate the external force.

```
/* File: accelfun.c */
#include <stdio.h>
#define M_G   9.81

double force(double t) {
    double p;

    p = 4*(t-3)+20;
    return p;
}

int main() {
    double a, mu, m, p, t;

    mu = 0.2;
    m = 5.0;
    t = 2.0;
    p = force(t);
    a = (p-mu*m*M_G)/m;
    printf("Acceleration a = %f (m/s^2)\n", a);
    return 0;
}
```

Output:

```
Acceleration a = 1.238000 (m/s^2)
```

Example Program 2:

Use functions with multiple arguments.

```
/* File: accelfunm.c */
#include <stdio.h>
#define M_G   9.81

double force(double t) {
    double p;

    p = 4*(t-3)+20;
    return p;
}
double accel(double t, double mu, double m) {
    double a, p;

    p = force(t);
    a = (p-mu*m*M_G)/m;
    return a;
}
int main() {
    double a, mu, m, t;

    mu = 0.2;
    m = 5;
    t = 2;
    a = accel(t, mu, m);
    printf("Acceleration a = %f\n (m/s^2)", a);
    return 0;
}
```

Example Program 3:

Use function prototypes.

```
/* File: accelprot.c */
#include <stdio.h>
#define M_G   9.81

double force(double t);
double accel(double t, double mu, double m);

int main() {
    double a, mu, m, t;

    mu = 0.2;
    m = 5.0;
    t = 2.0;
    a = accel(t, mu, m);
    printf("Acceleration a = %f (m/s^2)\n", a);
    return 0;
}
double force(double t) {
    double p;

    p = 4*(t-3)+20;
    return p;
}
double accel(double t, double mu, double m) {
    double a, p;

    p = force(t);
    a = (p-mu*m*M_G)/m;
    return a;
}
```

Recursive Functions

- Functions may be used recursively. This means that a function can call itself directly.
- When a function calls itself recursively, each function call will have a new set of local variables.
- Recursive functions usually contain conditional statements, such as `if-else`, to allow exit of the function and return control to the calling function.
- A function may also call itself indirectly.
- Condition:
  - **Function must be terminated with a condition**
  - **Function must be written recursively**

What is Recursion?

- Recursion is the name given for expressing anything in terms of itself. A function which contains a call to itself or a call to another function, which eventually causes the first function to be called is known as ***RECURSIVE FUNCTION.***
- Each time a function is called, a new set of local variables and formal parameters are allocated on the ***stack*** and execution starts from the beginning of the function using these new value.
- The call to itself is repeated till a base condition is reached. Once a base condition or a terminal condition is reached, the function returns a result to the previous copy of the function.
- A sequence of returns ensure that the solution to the original problem is obtained.

Example:

Calculating a factorial 5! using a function with a for-loop.

The factorial n! is defined as n*(n-1)!

```
/* File: factorloop.c */
#include <stdio.h>

unsigned int factorial(int n);
int main() {
    int i;

    for(i=0; i<=5; i++)
       printf("%d! = %d\n", i, factorial(i));
    return 0;
}
unsigned int factorial(int n) {
    unsigned int i, f;

    for(i=1, f=1; i<=n; i++) {
       f *= i;
    }
    return f;
}
```

Output:

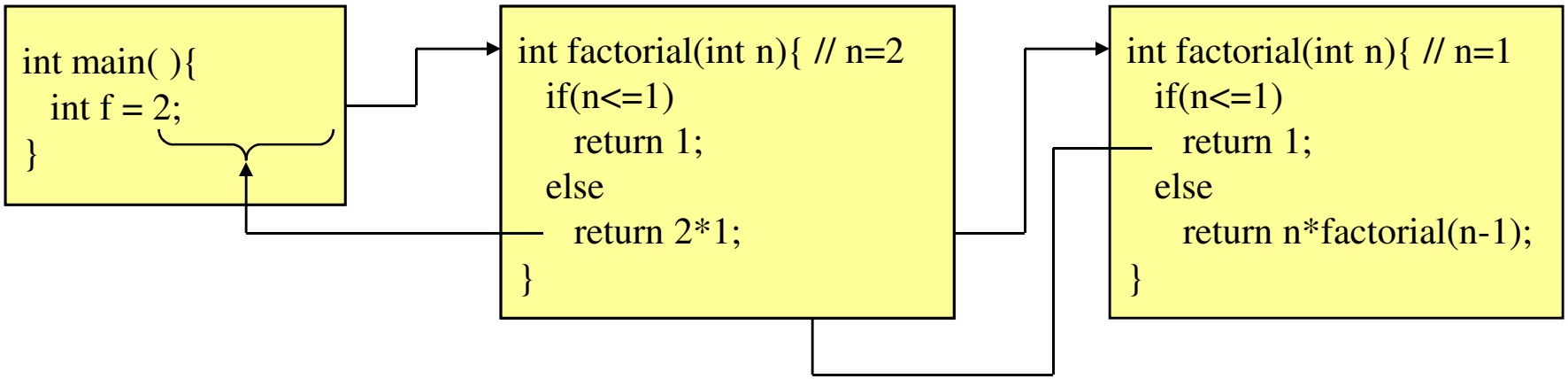
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Example: Calculating a factorial 5! using a recursive function.

```
/* File: factorrecursive.c */
#include <stdio.h>

unsigned int factorial(int n);

int main() {
    int i;

    for(i=0; i<=5; i++)
       printf("%d! = %d\n", i, factorial(i));

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

Output:

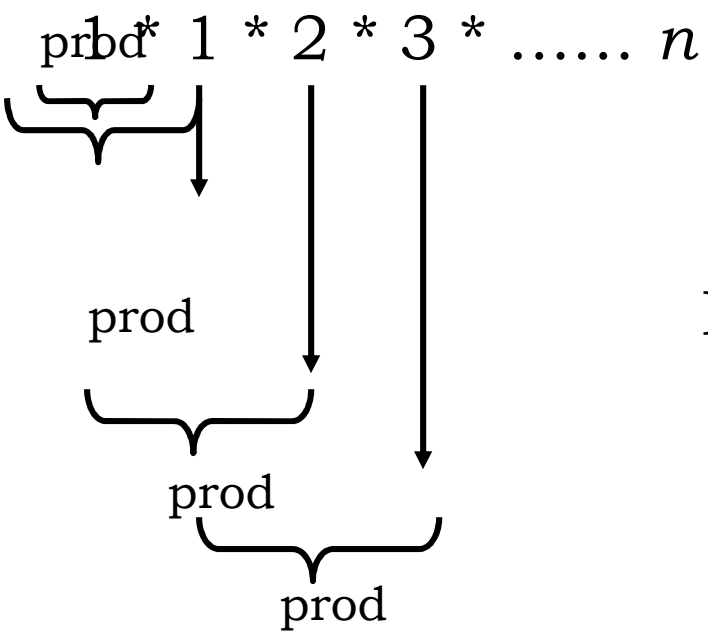
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```


int main( ){
int f = 2;
}
int factorial(int n){ // n=2
if(n<=1)
return 1;
else
return 2*1;
}
int factorial(int n){ // n=1
if(n<=1)
return 1;
else
return n*factorial(n-1);
}

Case Study: Factorial of *n*

Design Process

The series can be multiplied pictorially as shown below:



Recursive Technique

$$\text{Fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n-1) & \text{otherwise} \end{cases}$$

Program to find the factorial of *n*

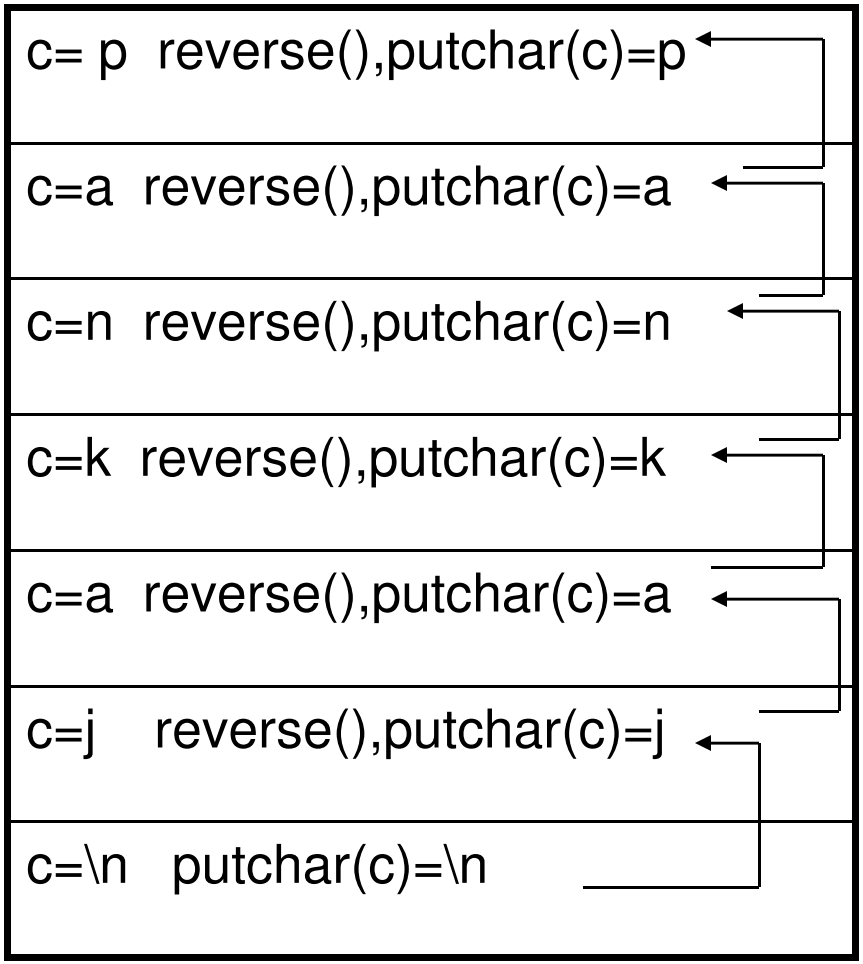
```
int fact(int n)
{
   if (n==0)        return 1;    /* 0! = 1 */
   return n*fact(n-1);           /* n!=n*(n-1)! */
}
void main()
{
   int n;
   printf("Enter value for n\n");
   scanf("%d",&n);
   printf("The factorial of %d = %d\n",n,fact(n));
}
```

```
#include<stdio.h>
#define EOLN '\n'

void reverse(void);

main()
{
printf("Please enter a line of text below\n");
reverse();
}

void reverse(void)
{
char c;
if((c=getchar())!=EOLN)
reverse();
putchar(c);
}
```

Contd…

Input: pankaj

Output:jaknap



Advantages and Disadvantages of Recursion

Advantages

- Clearer and simple versions of algorithms can be created.
- Recursive definition can be easily transferred into a recursive function.

Disadvantages

- Separate copy of the function variables are created for each call and definitely consumes lot of memory.
- Most of the time is spent in pushing and popping the necessary items from the stack and so consumes more time to compute the result.
- They often execute slowly when compared to their iterative counterpart because of the overhead of the repeated function calls.

Iteration v/s Recursion

Iteration

- Uses repetition structures such as for,while loops.
- It is counter controlled and the body of the loop terminates when the termination condition is failed.
- They execute much faster and occupy less memory and can be designed easily.

Recursion

- Uses selection structures such as if-else,switch statements.
- It is terminated with a base condition.The terminal condition is gradually reached by invocation of the same function repeatedly.
- It is expensive in terms of processor time and memory usage.

Storage Class

Pankaj Raj Dawadi
Lecturer, DoCSE
Kathmandu University

Storage Class Specifiers

- C has four kinds of storage classes
  - Automatic
  - Register
  - Static
  - External
- Storage class is used to denote to the compiler where ***memory*** is to be allocated for variables
- Storage class tells, what will be the ***initial value*** of the variable, if the initial value is not specifically assigned.(i.e the default initial value).
- Storage class informs the compiler the ***scope*** of the variable.Scope refers to which functions the value of the variable would be available.
- Storage class informs the compiler the ***life*** of the variable. Life refers to how long would the variable exist.

Automatic Storage Class

- All variables declared within a function are called *local* variables, by default belong to the *automatic* storage class.

Storage	Memory
Default Initial Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life	Till the control remains within the block in which it is defined

Example for Automatic Storage Class

```
main()
{
        auto int i,j;
        printf("%d \n %d \n",i,j);
}
```

Output

1211

876

Example for Automatic Storage Class

```
main()
{
        auto int j=1;
        {
                auto int j=2;
                {
                        auto int j=3;
                        printf("%d\n",j);
                }
                printf("%d\n",j);
        }
        printf("%d\n",j);
}
```

Output:

3

2

1

Register Storage Class

- *Register* storage class allocates memory in CPU's high speed registers.

Storage	CPU Registers
Default Initial Value	Garbage Value
Scope	Local to the block in which the variable is defined
Life	Till the control remains within the block in which it is defined

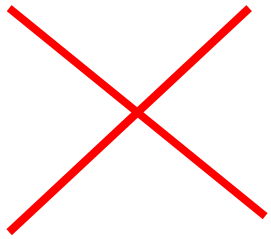
Example for Register Storage Class

```
main()
{
        register int i;
        for(i=1;i<=10;i++)
                printf("%d\n",i);
}
```

Note:

The register storage class cannot be used for all types of variables.
Example:

```
register float q;
register double s;
register long r;
```



Static Storage Class

- *Static* storage class informs the compiler that the value stored in the variables are available between function calls

Storage	Memory
Default Initial Value	Zero
Scope	Local to the block in which the variable is defined
Life	Value of the variable persists for different function calls

Example for Static Storage Class

```
main()
{
        function();
        function();
        function();
}
function()
{
        static int i=1;
        printf("%d\n",i);
        i=i+1;
}
```

Output

1

2

3

```
main()
{
        function();
        function();
        function();
}
function()
{
        auto int i=1;
        printf("%d\n",i);
        i=i+1;
}
```

Output

1

1

1

Example:2

```
int fibo(int count)
{
static int f1=1,f2=1;
int f;
f=(count<3)?1:f1+f2;
f2=f1;
f1=f;
return f;
}
```

```
main( )
{
int count,n;
printf("How many numbers:");
scanf("%d",&n);
for(count=1;count<=n;count++)
printf("count=%d f=%d",count,
fibo(count));
}
```

External Storage Class

- Variables declared outside functions have ***external*** storage class

Storage	Memory
Default Initial Value	Zero
Scope	Global
Life	As long as the program's execution doesn't come to an end

Example for External Storage Class

```
int i;

main()

{

     printf("i=",i
);

     increment();

     increment();

     decrement();

     decrement();

}
```

```
increment()

{

i=i+1;

printf("On incrementingi =
%d\n",i);

}

decrement()

{

i=i-1;

printf("On incrementing
i=%d\n",i);

}
```

Output:
i=0
On incrementing i=1
On incrementing i=2
On decrementing i=1
On decrementing i=0

Summary

- The different decision structure are if statement, if – else statement, multiple choice else if statement.
- The while loop keeps repeating an action until an associated test returns false.
- The do while loops is similar, but the test occurs after the loop body is executed.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times.
- Storage class is used to denote to the compiler where ***memory*** is to be allocated for variables
- C has four kinds of storage classes
  - Automatic
  - Register
  - Static
  - External

Thank You!