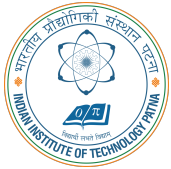
CS1101: Foundations of Programming

Functions



Dept. of Computer Science & Engineering

Indian Institute of Technology Patna

Function

- A function is a self-contained program segment that carries out some specific, well-defined task.
- Example
  - A function to add two numbers
  - A function to find binary representation of a decimal number
- A function will carry out its intended task whenever it is called or invoked
  - Can be called multiple times
- Every C program consists of one or more functions
- One of these functions must be called `main`
- Execution of the program always begins by carrying out the instructions in `main`
- Functions call other functions as instructions

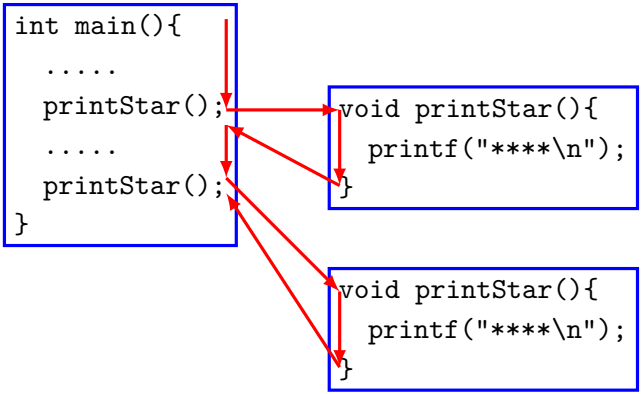
Function control flow

Code

```
void printStar(){
  printf("****\n");
}
```

```
int main(){
  .....
  printStar();
  .....
  printStar();
}
```

Execution



Functions

- Calling function (caller) may pass information to the called function (callee) as parameters/arguments
  - For example, the numbers to add
- The callee may return a single value to the caller
  - Some functions may not return anything
- If function X calls function Y
  - X : calling / caller function
  - Y : called / callee function

Why functions?

- Allows one to develop a program in a modular fashion
  - Code becomes readable
  - Easy to debug and maintain
- Write your own function and avoid writing the same code multiple times
- Use existing functions as building blocks for new programs
- Abstraction: hide internal details (library functions)

Example

```
int main(){
  float cent,fahr;
  scanf("%f",&cent);
  fahr=cent2fahr(cent);
  printf("%f",fahr);
  return 0;
}
```

```
float cent2fahr(float data){
  float result;
  result=data*9/5+32;
  return result;
}
```

Another example

```
int factorial(int m){
  int i, temp=1;
  for(i=1; i<=m; i++)
    temp = temp*i;
  return(temp);
}

int main(){
  int n;
  for(n=1; n<=10; n++)
    printf("%d! = %d\n",n,factorial(n));
  return 0;
}
```

```
Output:
1! = 1
2! = 2
3! = 6
...
```

Defining a function

- A function definition has two parts
  - The first line, called header
  - The body of the function

```
return-type function-name( parameter-list ){
  declarations and statements
}
```

- The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses
  - Each argument has an associated type declaration
  - The arguments are called formal arguments or formal parameters
- The body of the function is actually a block of statement that defines the action to be taken by the function

Accessing/Calling a function

- A function can be accessed by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas
  - There can be no argument also, an empty parentheses need to be provided
  - Function call can be part of expressions, assignments, etc.
- Arguments appearing on a function call are *actual arguments*
- Arguments appearing on definition function are *formal arguments*
- Arguments are matched based on the position
- Formal and actual arguments are expected to match. Mismatches are auto-typecasted if possible

Example: Accessing/Calling a function

```
char LTU(char in){
  char c;
  c = (in >= 'a' && in <= 'z') ? ('A' + in -'a') : in ;
  return c;
}

int main(){
  char c, upper;
  scanf("%c",&c);
  upper = LTU(c);
  printf("%c\n",upper);
  return 0;
}
```

Function prototypes

- A function is usually defined before it is called
  - `main()` is usually the last function
  - Compiler can identify function definitions in a single pass
- One can write the function definition after `main` but needs to be declared before `main`
  - Compiler will know the set of function call
  - Function prototypes are used for this purpose
- Function prototypes are usually written before main
- Prototype must mention the return type and the types of the arguments.
  - `int max(int a, int b);`
  - `int LUT(char);`
  - Parameter names are ignored by compiler

Example

```
#include <stdio.h>
int cmp(int, int);

int main(){
  int x, y, out;
  scanf("%d%d",&x, &y);
  out = cmp(x, y);
  printf("%d\n",out);
}

int cmp(int a, int b){
  return (a > b) ? a : b;
}
```

Function prototype

Function call

Return type

Function definition

Return value

Return value

- A function can return a value
  - Using `return` statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```
float fahr, cent;
scanf("%f",&cent);
fahr=cent2fahr(cent);
printf("%f",fahr);
```

- A function may not return any value to caller. The return type of such function is `void`

Function not returning any value

```
void div3(int n){
  if((n%3)==0)
    printf("%d is divisible by 3",n);
  else
    printf("%d is not divisible by 3",n);
  return; /* optional return */
}
```

`return` statement

- In a value-returning function (result type is not void), return does two distinct things
  - Specify the value returned by the execution of the function
  - Terminate that execution of the callee and transfer control back to the caller
- A function can only return one value
  - The value can be any expression matching the return type
  - But it might contain more than one return statement.
- In a `void` function
  - `return` is optional at the end of the function body.
  - `return` may also be used to terminate execution of the function explicitly.
  - No return value should appear following `return`

Example: print grade

```
void printGrade(int n){
  if((n >= 90){
    printf("AA\n"); return;
  }
  if((n >= 80){
    printf("AB\n"); return;
  }
  if((n >= 70){
    printf("BB\n"); return;
  }
  printf("FAILED\n");
}
```

Execution flow

```
int main(){
   int num, j=1;
   long int val;
   scanf("%d",&num);
   while(j < num){
      val = fact(j);
      printf("%d! = %ld\n",j, val);
      j++;
   }
   return 0;
}
```

```
long int fact(int n){
   int j=1;
   long int val=1;
   for(; j <= n; j++){
      val *= j;
   }
   return val;
}
```

Execution flow

```
int main(){
   int num, j=1;
   long int val;
   scanf("%d",&num);
   while(j < num){
      val = fact(j);
      printf("%d! = %ld\n",j, val);
      j++;
   }
   return 0;
}
```

```
long int fact(int n){
   int j=1;
   long int val=1;
   printf("BEFORE\n");
   for(; j <= n; j++){
      val *= j;
   }
   printf("AFTER\n");
   return val;
}
```

Nested function

- A function cannot be defined inside another function
- All function definitions must be separate
- A function can call any other functions
- Nested function calls are allowed. For example, A calls B, B calls C, C calls D, etc.
- A function can call itself directly or indirectly
  - Function A calls A
  - A calls B, B calls C, C calls A
  - Known as recursive call or recursion

Nested call

```
#include <stdio.h>
int npr(int, int);
int fact(int);

int main(){
   int n, r, val;
   scanf("%d%d", &n, &r);
   val = npr(n, r);
   printf("val = %d\n", val);
}
```

```
int fact(int n){
   int j=1;
   int val=1;
   for(; j <= n; j++){
      val *= j;
   }
   return val;
}

int npr(int n, int r){
   return (fact(n)/fact(r));
}
```

Nested call (recursion)

```
#include <stdio.h>
int npr(int, int);
int fact(int);

int main(){
   int n, r, val;
   scanf("%d%d", &n, &r);
   val = npr(n, r);
   printf("val = %d\n", val);
}
```

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

int npr(int n, int r){
   return (fact(n)/fact(r));
}
```

Local variables

- A function can have its own local variables
- The local variables can be accessed from the function body only in which it is defined
  - Such variables cease to exist once it returns from the function
  - Each execution of the function will use different set of local variables
- Parameters are also local
- Example:

```
float perimeter(float radius){
   float val, dia;
   dia = 2 * radius;
   val = (22.0)/7 * dia;
   return val;
}
```

Scope of a variable

- Scope - part of the program in which a variable is accessible
- Scope of a variable is the block `{ }` in which it is defined
- Local variable - scope is usually within the function
  - Two different functions can have a variable with the same name, but they are non-identical
- Global variables - declared outside the all functions including `main`
  - Scope of such variable is entire program but it can be hidden in a block if a local variable with same name is defined
  - Try to avoid global variables

Example: Global scope

```
#include <stdio.h>
int x;

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   x = 2;
   printf("MYFUNC: x = %d\n", x);
}
```

Example: Global scope

```
#include <stdio.h>
int x;

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   x = 2;
   printf("MYFUNC: x = %d\n", x);
}
```

Output:

```
MYFUNC: x = 2
MAIN: x = 2
```

Example: Global scope

```
#include <stdio.h>
int x;

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   x = 2;
   printf("MYFUNC: x = %d\n", x);
}
```

Output:

```
MYFUNC: x = 2
MAIN: x = 2
```

Scope of x is the whole program.

Example: Global & local scope

```
#include <stdio.h>
int x; /* global x */

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   int x = 2; /* local x */
   printf("MYFUNC: x = %d\n", x);
}
```

Example: Global & local scope

```
#include <stdio.h>
int x; /* global x */

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   int x = 2; /* local x */
   printf("MYFUNC: x = %d\n", x);
}
```

Output:

```
MYFUNC: x = 2
MAIN: x = 1
```

Example: Global & local scope

```
#include <stdio.h>
int x; /* global x */

int main(){
   x = 1;
   myfunc();
   printf("MAIN: x = %d\n", x);
   return 0;
}

void myfunc(){
   int x = 2; /* local x */
   printf("MYFUNC: x = %d\n", x);
}
```

Output:

```
MYFUNC: x = 2
MAIN: x = 1
```

There are two `x` variables - one is global and the other is local to `myfunc`.

Passing parameters to functions

- When an argument is passed, the value of the actual argument is copied into the function
- The value of the formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change
- Known as passing by value
- Called function receive a copy of the actual arguments. Such copy will be destroyed when the execution returns from the called function

Call by reference

- Passes the address of the original argument to a called function
- Execution of the function may affect the original argument in the calling function
- Not directly supported in C, but supported in C++
- In C, you can pass copies of address to get the desired effect

Example: Passing parameters to functions

```
#include <stdio.h>
void func(int a);

int main(){
   int x = 2;
   printf("M1: x = %d\n", x);
   func(x);
   printf("M2: x = %d\n", x);
   return 0;
}

void func(int x){
   printf("F1: x = %d\n", x);
   x *= x;
   printf("F2: x = %d\n", x);
}
```

Example: Passing parameters to functions

```
#include <stdio.h>
void func(int a);

int main(){
   int x = 2;
   printf("M1: x = %d\n", x);
   func(x);
   printf("M2: x = %d\n", x);
   return 0;
}

void func(int x){
   printf("F1: x = %d\n", x);
   x *= x;
   printf("F2: x = %d\n", x);
}
```

Output:

```
M1: x = 2
F1: x = 2
F2: x = 4
M2: x = 2
```

Example: Passing parameters to functions

```
#include <stdio.h>
void func(int, int);

int main(){
   int x = 2, y = 3;
   printf("M1: x=%d, y=%d\n", x, y);
   func(x, y);
   printf("M2: x=%d, y=%d\n", x, y);
   return 0;
}

void func(int x, int y){
   printf("F1: x=%d, y=%d\n", x, y);
   tmp=x; x=y; y=tmp;
   printf("F2: x=%d, y=%d\n", x, y);
}
```

Example: Passing parameters to functions

```
#include <stdio.h>
void func(int, int);

int main(){
   int x = 2, y = 3;
   printf("M1: x=%d, y=%d\n", x, y);
   func(x, y);
   printf("M2: x=%d, y=%d\n", x, y);
   return 0;
}

void func(int x, int y){
   printf("F1: x=%d, y=%d\n", x, y);
   tmp=x; x=y; y=tmp;
   printf("F2: x=%d, y=%d\n", x, y);
}
```

Output:

```
M1: x=2, y=3
F1: x=2, y=3
F2: x=3, y=2
M2: x=2, y=3
```

Passing arrays to a function

- Sometime, we need to pass an array to a function
- The array name is used to pass the entire array (not exactly though)
- Function prototype and definition with an array: no need to specify size

```
void func(int []);
void func(int x[]){
   …
}
```

- Function call with an array: only the name needs to be specified

```
int x[10];
func(x);
```

Example: Passing array to function

```
#include <stdio.h>
int sum(int, int[]);

int main(){
   int n, x[100], i, out;
   scanf("%d",&n);
   for(i=0; i<n; i++){
     scanf("%d",&x[i]);
   }
   out = sum(n, x);
   printf("Sum: %d",out);
   return 0;
}
```

```
int sum(int a, int b[]){
   int j=0, s=0;
   for(; j<a; j++){
     s = s + b[j];
   }
   return s;
}
```

When a function receives an array as input, it does not know the size. The programmer needs to keep track of the size.

What happens when an array is passed?

- The values of the array are NOT passed to the function
- The array name, which is passed in function, considered as the address of the first element of the array
- The formal argument is actually the pointer to the first element of the array
- When an element of the array is accessed, the address is calculated as `base-address + index × size-of-single-element`
- Changes made in any element inside the function will be visible to calling function

Parameter passing

- In C parameters are passed using call-by-value
- Passing the starting address when array is sent as arguments simulates call-by-reference
- If a function changes the elements of array which is passed as arguments, such changes will be reflected to the original array
- If an element of array is passed, then such modification will not be visible in the calling function

Example

```
int main(){
   int x[]={1, 2, 3, 4, 5};
   int y[]={1, 4, 9, 16, 25};
   printf("M1: x[1]:%d y[1]:%d\n", x[1], y[1]);
   update(x, y[1]);
   printf("M2: x[1]:%d y[1]:%d\n", x[1], y[1]);
   return 0;
}

void update(int a[], int b){
   a[1] = 6;
   b = 36;
}
```

Example

```
int main(){
   int x[]={1, 2, 3, 4, 5};
   int y[]={1, 4, 9, 16, 25};
   printf("M1: x[1]:%d y[1]:%d\n", x[1], y[1]);
   update(x, y[1]);
   printf("M2: x[1]:%d y[1]:%d\n", x[1], y[1]);
   return 0;
}

void update(int a[], int b){
   a[1] = 6;
   b = 36;
}
```

Output:

```
M1:  x[1]:2 y[1]:4
M2:  x[1]:6 y[1]:4
```

Example

```
#include <stdio.h>
void cube(int, int[]);

int main(){
   int n, x[100], i, out;
   scanf("%d",&n);
   for(i=0; i<n; i++){
     scanf("%d",&x[i]);
   }
   cube(n, x);
   for(i=0; i<n; i++){
     printf("%d ",x[i]);
   }
   return 0;
}
```

```
void cube(int size, int a[]){
   int j=0
   for(; j<size; j++){
     a[j] = a[j] * a[j] * a[j];
   }
}
```

Header files

- Contains function declarations / prototype for library functions
- There are many headers files such as `<stdlib.h>`, `<ctype.h>`, etc.
- Such files need to be loaded with `#include` directive
- Function prototype are usually specified in header file and actual function definition is available in library
- One can write custom header files, eg. `myheader.h` (say)
- It can be loaded by `#include "myheader.h"`

C preprocessor

- Statement starts with # are preprocessor directives and handled by the preprocessor
- It can be a separate program or the compiler
- Preprocessing is done before the compilation
- The preprocessor usually substitute the text
- For example, `#include ...` is replaced by the contents of the specified file

#define: Macro

- **Preprocessor directive. #define string1 string2**
- **It replaces `string1` by `string2` wherever it occurs**

```
#define PI 3.14
int main(){
   float r, p;
   p = 2 * PI * r;
}
```

```
int main(){
   float r, p;
   p = 2 * 3.14 * r;
}
```

`#define`: with arguments

- `#define` **statements can be used with arguments also**

```
#define sqr(x) x*x
r = sqr(3) + sqr(a) // r = 3*3 + a*a
r = sqr(a+b) // r = a+b*a+b
```

- **One should be careful in using this**

```
sqr(x) (x)*(x)
r = sqr(a+b); // r = (a+b)*(a+b)
r = c / sqr(a+b); // r = c / (a+b)*(a+b)
```

- **Macros are not functions. They are literally substituted without evaluation**

Practice problems

- **Write a menu driven program to perform the following operations -**
  (a) read in a number and print the sum of digits,
  (b) read in a number and check for primality
  (c) read in n numbers in an array and determine mean, mode, median, standard deviation
  (d) read in n numbers in an array, check another given number exists in the array or not
  (e) read an integer convert it in binary
- **Write a menu driven program for following**
  (a) read in set of roll numbers (int) of students and marks in 6 subjects for each students
  (b) print the whole record
  (c) print total marks for each student
  (d) print the roll number who has the highest total
  (e) print the roll number who scored maximum marks in subject j $(1 \leq j \leq 6)$
  (f) find standard deviation of marks for each subject