CPNM Lecture 10 - Functions

Mridul Sankar Barik

Jadavpur University

2022-23

Functions - Introduction

- Functions are series of statements grouped together and given a name
- C functions vs. Mathematical functions : In C, a function doesn't necessarily have arguments, nor does it necessarily compute a value
- Functions are building blocks of C programs
- Each function is essentially a small program with its own declarations and statements
- Functions introduces modularity $\Rightarrow$ more understandable and modifiable
- Helps to avoid duplicating code that is used more than once
- Functions are reusable

Function Definition I

- Function definition

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

- Declarations and statements: function body (block)
  - Variables can be declared inside blocks (can be nested)
  - Functions can not be defined inside other functions
- Returning control
  - If nothing returned
    - return;
    - or, until reaches right brace
  - If something returned
    - return expression;
- A function can only return a single result

Function Definition II

- Example:

```
int myAdd(int x, int y){
    int z;
    z = x + y;
    return z;
}

void myPrint(){
    printf("Hello World\n");
}

float myFn(float x){
    float res = 0;
    res = x*x*x - x - 1;
    return(res);
}
```

Function Definition III

- Use `pow` function in `math.h` (compile with `-lm` option to link math library)

```
float myFn(float x){
    float res = 0;
    res = pow(x, 3) - x - 1;
    return(res);
}
```

Function Call I

- **Formal parameter**: identifiers (variable names) used in function definition
  - Formal parameters are local variables within the function
  - Formal parameters are assigned values from the corresponding actual parameters when the function is called
- **Actual parameter**: expressions with which a function is called

Function Call II

- Example:

```
int myAdd(int x, int y){
    int z;
    z = x + y;
    return z;
}

int main(void){
    int a=5, b=3, c;

    c = myAdd(a, b);
    c = myAdd(a+b, b);
    c = myAdd(2+3, 6*b);
}
```

- `x` and `y` are formal parameters used in the definition of function `myAdd`

Function Call III

- In the function call `c = myAdd(a, b);` `a` and `b` are actual parameters
  - Value of actual parameter `a` is copied as value of formal parameter `x`
  - Value of actual parameter `b` is copied as value of formal parameter `y`
- An actual parameter can be any expression (even function calls) provided it evaluates to the `type` of the corresponding formal parameter
- Example:

```
if(myAdd(-2, myAdd(1, 1)))
    printf("Hello");
else
    printf("World");
```

Function Prototype I

- A **function prototype** specifies the function's name and type signature (number of parameters, type of each parameter, and return type)
- Compiler uses this information to ensure conformity of subsequent function definition and function calls
- Example: following prototype declarations are equivalent

```
int myAdd(int x, int y);
int myAdd(int, int);
int myAdd(int p, int q);
```

- Needed only if a function is defined after it is called

Passing Arrays as Arguments I

- Example:

```
int arrAdd(int a[], int n){
  int i, sum = 0;
  for(i=0;i<n;i++)
    sum = sum + a[i];
  return(sum);
}
main(){
    int ar[]={1, 2, 3, 4};
    printf("%d\n", arrAdd(ar, 4));
}
```

- As strings are terminated by '\0', we need not specify number of character as an argument
- Example:

Passing Arrays as Arguments II

```
int myStrLen(char str[]){
  int i = 0;
  while(str[i]!='\0')
    i++;
  return(i);
}
main(){
    char s[]="Hello World";
    printf("%d\n", myStrLen(s));
}
```

Recursion I

- When a function calls itself
  - Direct Recursion

```
void A(void){
    ... A(); ...
}
```

  - Indirect Recursion

```
void A(void){
    ... B(); ...
}
void B(void){
    ... A(); ...
}
```

Recursion II

- To solve a problem recursively
  - Redefine the problem in terms of a smaller subproblem of the same type as the original problem
  - Also find a stopping criteria (base case) when there will no further call to itself
  - Example:
    - Factorial of n is factorial of $n-1$ multiplied by n
      `F(n) = n * F(n-1)`
      Base case: when `n==1`
    - To print reverse of a string print the last character of the string and then print the reverse of the remaining string
      `PrintRev(str) = Print(Last(str)) + PrintRev(exceptLast(str))`
      Base case: when `str` contains a single character
    - Sum of digits of a integer can be obtained by adding the last digit to the sum of digits of the remaining
      `SoD(n) = n%10 + SoD(n/10)`
      Base case: when `n` is a single digit number

Types of Recursion

- Tail Recursion: A call is tail-recursive if nothing has to be done after the call returns

```
void tail(int n){
    if(n == 1)
        return;
    else
        printf("%d\n", n);
    tail(n-1);
}
```

- Head Recursion: A call is head-recursive when the first statement of the function is the recursive call.

```
void head(int n){
    if(n == 0)
        return;
    else
        head(n-1);
    printf("%d\n", n);
}
```