CS101 Introduction to computing

Function, Scope Rules and Storage class

A. Sahu and S. V .Rao

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Functions
  - Modular Program
  - Inbuilt functions or library functions
  - User defined functions
- Parameter Passing
  - Pass by value, Pass by reference
- Storage Class for variable
- Scope Rules

Introduction

- Divide and conquer
  - Construct a program from smaller pieces or components
  - Each piece more manageable than the original program
- Functions
  - Modules in C
  - Programs written by combining user-defined functions with library functions
    - C standard library has a wide variety of functions
    - Makes programmer's job easier - avoid reinventing the wheel

Function calls

- Invoking functions
  - Provide function name and arguments (data)
  - Function performs operations or manipulations
  - Function returns results
- Boss asks worker to complete task
  - Worker gets information, does task, returns result
  - Information hiding: boss does not know details

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`
- Format for calling functions

  `FunctionName (argument);`
  - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

Math Library Functions

- double sin(double x) double cos(double x),
- double pow(double x, double n)
- long powl(long x, long n)
- float powf(float x, float n)
- double ceil(double x), double exp(double x)
- double acos(double x), double asin(double x)
- ………list continues…….

It is not good idea to remember all math functions .

Use man page

$man math.h

$man pow

$man sin

Std Library Functions and others

- double rand()
- void exit()
- int atoi()
- Long atol()

- ....list continues.......

It is not good idea to remember all stdlib functions .

Use man page

$man stdlib.h
$man rand
$man random

Functions

- Modularize a program
- All variables declared inside functions are local variables : Known only in function defined
- Parameters: Communicate info. between functions
- Function Benefits
  - Divide and conquer : Manageable program development
  - Software reusability : Use existing functions as building blocks for new programs and
  - Abstraction : hide internal details (library functions)
  - Avoids code repetition

Function Definitions

- Function definition format

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

- Function-name: any valid identifier
- Return-value-type: data type of the result (default **int**)

  –**void** - function returns nothing

- Parameter-list: comma separated list, declares parameters (default **int**)

Function Definitions

- Function definition format

```
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)
  - In C++, Function can not be defined inside another function : **But in c it is allowed**
- Returning control
  - If nothing returned : **return**;
    - or, until reaches right brace
  - If something returned : *return expression;*

Review of Structured Programming

- Structured programming is a problem solving strategy and a programming methodology that includes the following guidelines

- The program uses only the sequence, selection, and repetition control structures.

- The flow of control in the program should be as simple as possible.

- The **construction of a program embodies top-down design**.

Review of Top-Down Design

- Involves repeatedly **decomposing** a problem into smaller problems

- Eventually leads to a collection of
  - Small problems or tasks each of which can be easily coded

- The **function** construct in C is used
  - To write code for these small, simple problems.

Functions

- A C program is made up of one or more functions, one of which is main( ).

- Execution always begins with main( )
  - No matter where it is placed in the program.

- main( ) is located before all other functions.

- When program control encounters a function name, the function is **called** (**invoked**).

  1. Program control passes to the function.
  2. The function is executed.
  3. Control is passed back to the calling function.

Sample Function Call

```c
#include <stdio.h>
 int main ( ) {

  printf("Hello World!\n");
  return 0 ;
}
```

printf is the name of a **predefined function** in the stdio library

this statement is is known as a **function call**

this is a string we are **passing** as an **argument** (**parameter**) to the printf function

Functions (con't)

- We have used three predefined functions so far:

  - printf, scanf, pow, sqrt, abs, sin, cos

- Programmers can write their own functions.

- Typically, each module in a program's design hierarchy chart is implemented as a function.

Sample -Defined Function

```c
#include <stdio.h>
void PrintMessage(void);
int main(){
    PrintMessage();
     return 0 ;
}
void PrintMessage(void){
  printf("A MSG :\n\n");
  printf("Nice day!\n");
}
```

Function Prototype/ Declaration

Function Call

Function Header

Function Body or Definition

The Function Prototype

- Informs the compiler that there will be a function defined later that:

```
void PrintMessage(void);
```

| Returns this type | Has this name | Takes these type of arguments in order |

- Needed because the function call is made before the definition -- the compiler uses it to see if the call is made properly

The Function Call

- Passes program control to the function
- Must match the prototype in name, number of arguments, and types of arguments

```
void PrintMessage(void);
int main(){

    PrintMessage( );

    return 0 ;

}
```

No Arguments

Same Name
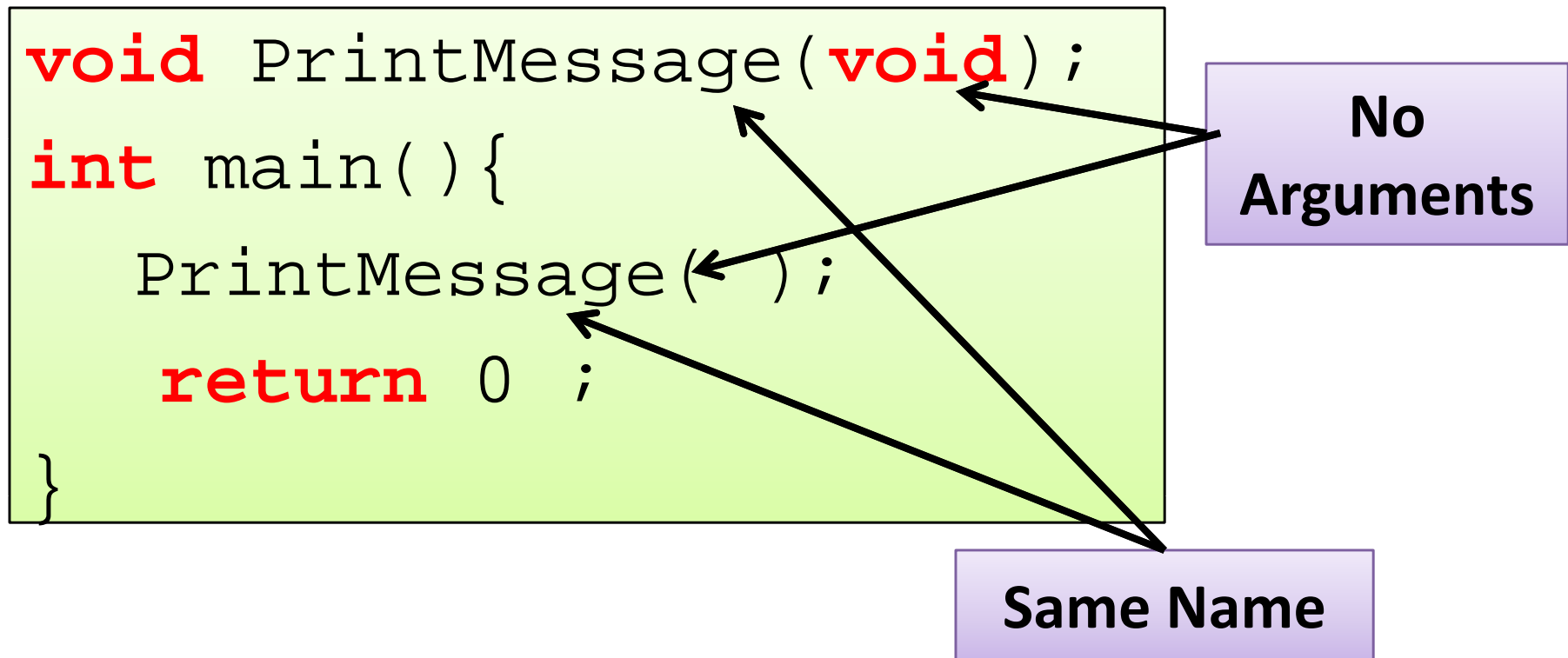
The Function Definition

- Control is passed to the function by function call
  - The statements within the function body will then be executed

```c
void PrintMessage(void){
  printf("A MSG :\n\n");
  printf("Nice day!\n");
}
```

- After statements in the function have completed
  - Control is passed back to the **calling function**
- In this case main( )
  - Note that the calling function does not have to be main( ) .

General Function Definition Syntax

type functionName (parameter$_1$, . . . , parameter$_n$) {
     variable declaration(s)
     statement(s)

}

- If there are no parameters
  - either `functionName()` OR `functionName(void)`
- There may be no variable declarations.
- If the **function type** (**return type**) is void, a return statement is not required
  - Permitted: `return; OR return();`

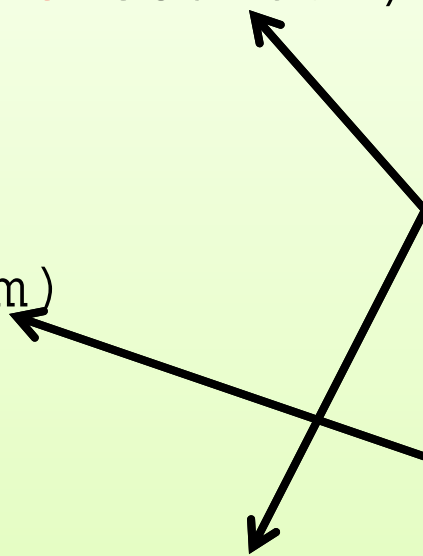# Input Parameters to Function

```c
void PrintMessage(int counter) ;

int main ( ){
    int 10;
    PrintMessage(num)
    return 0 ;
}


void PrintMessage(int counter) {
    int i ;
    for (i=0;i<counter; i++)
        printf ("Nice day!\n") ;
}
```

matches the one **formal parameter** of type int

one argument of type int

Functions Can Return Values : Example

```c
#include <stdio.h>
float AverageTwo(int num1,int num2);
int main(){
  float ave ;
  int value1 = 5, value2 = 8 ;
  ave=AverageTwo(value1, value2) ;
  printf("The average of %d & %d
      is %f\n",value1,value2,ave);
 return 0 ;
}
float AverageTwo (int num1, int num2) {
  return  (float)((num1+num2)/2.0) ;
}
```

Temp Convert Function in C

```c
double CtoF ( double paramCel){
    return paramCel*1.8+32.0;
}
```

- This function takes an input parameter
  - Called paramCel (temp in degree Celsius)
- Returns a value
  - that corresponds to the temp in degree Fahrenheit

How to use a function?

```c
#include <stdio.h>
double CtoF( double );
/*  Purpose: to convert temperature
* from Celsius to Fahrenheit *****/
int main() {
  double c, f;
  printf("Enter the degree (in Celsius): ");
  scanf("%lf", &c);
  f = CtoF(c);
  printf("Temperature (in Fahrenheit)
                          is %lf\n", f);
}
double CtoF ( double paramCel) {
  return paramCel * 1.8 + 32.0;
}
}
```

Terminology

- **Declaration**

```
    double CtoF(double);
```

- **Invocation (Call)**

```
    double CtoF(double);
```

- **Definition**

```
duuble CtoF( double paramCel){
    return paramCel*1.8 + 32.0;
  }
```

Modularity: Example

```c
#include <stdio.h>

double GetTemp();
double CelsToFahr(double);
void DispRes(double,double);

int main(){
   double TempC, TempF;

   TempC=GetTemp();
   TempF=CelsToFahr(TempC);
   DispRes(TempC,TempF);

   return 0;
}
```

```c
double CelsToFahr(double Tem){
   return (Tem * 1.8 + 32.0);
}
```

```c
double GetTemp (){
   double Temp;
   printf("Please enter temp in
        degrees Celsius:");
   scanf("%lf", &Temp);
   return Temp;
}
```

```c
void DispRes(double CTemp,
        double Ftemp){
   printf("Original: %5.2f
        C\n", CTemp);
   printf("Equivalent: %5.2f
        F\n", FTemp);
}
```

Abstractions

- We are hiding details on *how* something is done in the function implementation
    - Put in library ☺ ☺ : do you require to know code for printf ? **No**

```
#include <stdio.h>

int main(){
  double TempC, TempF;

  TempC=GetTemp();
  TempF=CelsToFahr(TempC);
  DispRes(TempC,TempF);

  return 0;
}
```

```
double CelsToFahr(double Tem){
  return (Tem * 1.8 + 32.0);
}
```

```
double GetTemp (){
  double Temp;
  printf("Please enter temp in
         degrees Celsius:");
  scanf("%lf", &Temp);
  return Temp;
}
```

```
void DispRes(double CTemp,
            double Ftemp){
  printf("Original: %5.2f
         C\n", CTemp);
  printf("Equivalent: %5.2f
         F\n", FTemp);
}
```

Parameter Passing

- **Actual parameters** are the parameters that appear in the function call

```
ave =AverageTwo(value1,value2) ;
```

- **Formal parameters** are the parameters that appear in the function header

```
float AverageTwo(int num1,int num2)
```

- Actual and formal parameters are matched by position.

- Each formal parameter receives the value of its corresponding actual parameter.

Parameter Passing (cont..)

- Corresponding actual and formal parameters

  - Do not have to have the same name, but they may.

  - Must be of the same data type, with some exceptions, Exception example

```
int fact(int N){ //Code for Fact;}
main(){
    float X=10.34;
    printf("fact of %f is %d",x, fact( X ));
}
```

auto conversion

Local Variables

- Functions only "see" (have access to) their own **local variables**. This includes main( )

- Formal parameters are declarations of local variables.
  - The values passed are assigned to those variables.

- Other local variables can be declared within the function body.

Parameter Passing and Local Variables

```c
int main(){
   float ave ;
   int v1=5, v2=8 ;
   ave=AvgOfTwo(v1, v2);
   printf ("The average
       is %f\n", ave);
   return 0 ;
}
```

```c
float AvgOfTwo(int n1,
               int n2){
   float average;
   average=(n1+n2)/2;
   return average;
}
```

Local copy of variables

5	8	6.5
v1	**v2**	**ave**

5	8	6.5
n1	**n2**	**average**

Same Name, Still Different Memory Locations

```c
int main(){
    float ave ;
    int n1=5, n2=8 ;
    ave=AvgOfTwo(n1, n2);
    printf ("The average
        is %f\n", ave);
    return 0 ;
}
```

```c
float AvgOfTwo(int n1,
               int n2){
    float average;
    average=(n1+n2)/2;
    return average;
}
```

Local copy of variables

5	8	6.5
n1	**n2**	**ave**

5	8	6.5
n1	**n2**	**average**

Changes to Local Variables Do NOT Change Other Variables with the Same Name

```c
int main(){
   int n1=5;
   AddOne(n1);
   printf ("In main
       n1 is %d\n",n1);
   return 0 ;
}
```

```c
void AddOne (int n1){
   n1=n1+1;
   printf ("In AddOneF
       n1 is %d\n",n1);
   return;
}
```

5

n1

6

n1

Local copy of variables

OUTPUT
In AddOneF n1 = 6
In main n1 = 5

Solution : use Pass by reference

```c
int main(){
   int n1=5;
   int *Pn1;
   Pn1=&n1;
   AddOne(Pn1);
   printf ("In main
      n1 is %d\n",n1);
   return 0 ;
}
```

```c
void AddOne(
        int *Pn1){
   *Pn1=*Pn1+1;
   printf ("In AddOneF
      n1 is %d\n",*Pn1);
   return;
}
```

5	&n1

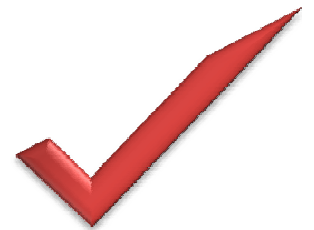**n1** **Pn1**

&n1

Pn1

Local copy of Ptr variables

OUTPUT

In AddOneF n1 = 6

In main n1 = 6

Changes to Local Variables Do NOT Change Other Variables with the Same Name

```
int main(){
   int n1=5, n2=10;
   swap(n1,n2);
   printf ("In main n1=
      %d n2=%d\n",n1,n2);
   return 0 ;
}
```

```
void swap(int n1,
      int n2){
int tmp;
tmp=n1; n1=n2;n2=tmp;
printf ("In main n1=
      %d n2=%d\n",n1,n2);
}
```

Local copy of variables

5	10
n1	n2

10	5	5
n1	n2	tmp

OUTPUT
In swap n1 = 10 n2 =5
In main n1 = 5 n2 =10

Use Pass by Address/Reference

```c
int main(){
  int n1=5, n2=10;
  swap(&n1,&n2);
  printf ("In main n1=
    %d n2=%d\n",n1,n2);
  return 0 ;
}
```

```c
void swap(int *Pn1,
    int *Pn2){
int tmp;
tmp=*Pn1;
*Pn1=*Pn2;*Pn2=tmp;
printf ("In main n1=
    %d n2=%d\n",n1,n2);
}
```

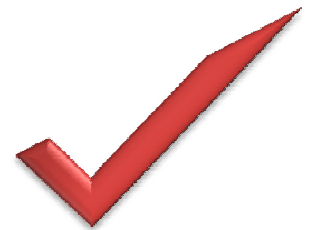**Local copy of variables**

5	10
n1	n2

&n1	&n2	5
Pn1	Pn2	tmp

OUTPUT

In swap n1 = 10 n2 =5

In main n1 = 10 n2 =5

Passing Array to Function

```
//(const float *age) (float *age) (float age[6]) same
```

```c
float average(float age[]){
    int i; float avg, sum = 0.0;
    for (i = 0; i < 6; ++i) {
        sum = sum +age[i];  age[i]=1;
    }
    avg = (sum / 6);  return avg;
}
```

```c
int main(){
  float avg, age[]={23.4,55,22.6,3,40.5,18};
  int i;
  avg = average(age);
  printf("Average age=%.2f\n", avg);
  for(i=0;i<6;++i) printf("%1.2f",age[i]);
  return 0 ;
}
```

Storage Classes

- **Storage class specifiers** : static, register, auto, extern
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known
- **Automatic storage**
  - Object created and destroyed within its block
  - `auto:` default for local variables ` auto double x, y;`
  - `register:` tries to put variable into high-speed registers
    - Can only be used for automatic variables

Automatic Storage

- Object created and destroyed within its block
- `auto`: **default for local variables**

```
auto double x, y; //same as double x, y
```

- **Conserving memory**
  - because automatic variables exist only when they are needed.
  - They are created when the function in which they are defined is entered
  - and they are destroyed when the function is exited
- **Principle of least privilege**
  - Allowing access to data only when it is absolutely needed.
  - Why have variables stored in memory and accessible when in fact they are not needed?

39

Register Storage

- The storage-class specifier **register** can be placed before an automatic variable declaration
  - To suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers.

```
register int counter;
```

  - If intensely used variables such as counters or totals can be maintained in hardware registers
- **Often, register declarations are unnecessary**
  - Today's optimizing compilers are capable of recognizing frequently used variables
  - Can decide to place them in registers without the need for a register declaration

Static storage Classes

- Variables exist for entire program execution

- Default value of zero

- `static`: local variables defined in functions.

  – ***Keep value after function ends***

  – Only known in their own function

- `extern:` default for ***global variables*** and functions

  – Known in any function

Tips for Storage Class

- **Defining a variable as global rather than local**
  - Allows unintended side effects to occur
  - When a function that does not need access to the variable accidentally or maliciously modifies it
- **In general, use of global variables** should be avoided : except in certain situations
- **Variables used only in a particular function**
  - Should be defined as local variables in that function
  - Rather than as external variables.

Scope Rules

- File scope
  - Identifier defined outside function, known in all functions
  - Used for *global variables, function definitions, function prototypes*
- Function scope
  - Can only be referenced inside a function body

43

Scope Rules

- Block scope
  - Identifier declared inside a block
    - Block scope begins at definition, ends at right brace
  - Used for *variables, function parameters (local variables of function*)
  - **Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block**
- Function prototype scope
  - Used for identifiers in *parameter list*

Scope Rule Example

```c
int A; //global
int main(){
 A=1;
 MyProc();
 printf("A=%d\n",A);
 return 0 ;
}
void myProc(){
  int A=2;
  while(A==2){
    int A=3;
    printf("A=%d\n",A);
    break;
  }
  printf("A=%d\n",A);
}
```

Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

Printout:

A = 3

A = 2

A = 1

Scope and Life : Static Vs Global

```c
int GA; //global
int main(){
  int i;
  GA=1;
  for(i=1;i<10;i++)
     MyProc();
  printf("GA=%d",GA);
  return 0 ;
}
void myProc(){
   static int SA=2;
   SA=SA+1;
}
```

Both SA and GA Variables exist for entire program execution

- **SA initialized once**
- **SA can be accessible from myProc only**

- **But GA accessible from any part of Program**

Scope Rule Example

```c
int FunA(){return 4;}; //global
int main(){

   {

    int FunA(){return 3;};
   pintf("FA=%d\n",FunA());
   }


   pintf("FA=%d\n",FunA());
   return 0 ;
}
```

Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

Printout:

FA = 3

FA = 4

Compile using gcc

This code will not compile using c++/g++ compiler