Lecture 2:

Functions and Variable Scopes

Content:

I. Motivation

II. Modular Design and Coding

III. Functions in C

IV. Scope of Variable

V. Passing Arguments into Functions

Please check the note sections

I. Motivation

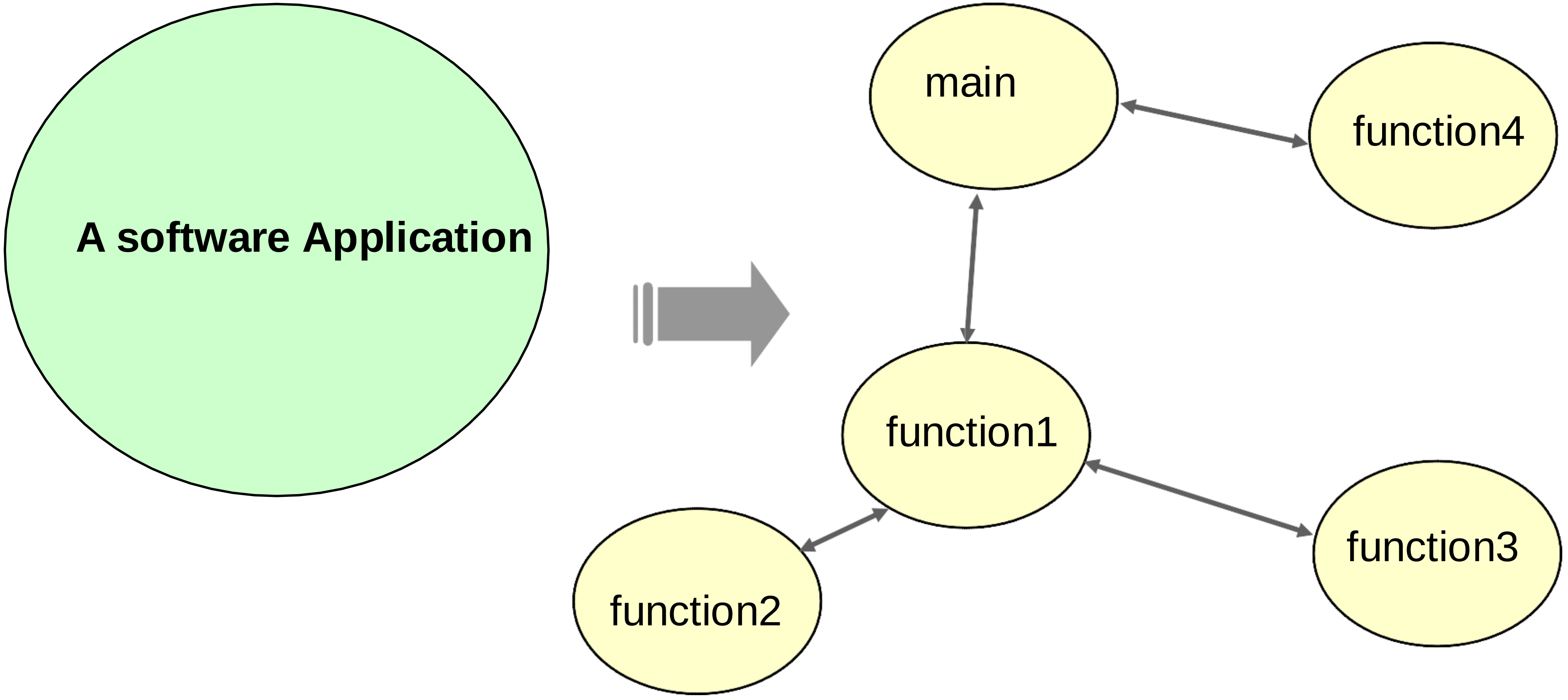
- Coding a **complex algorithm** may result in a **long** `main` **function** inflated to an unmanageable complexity. How to get around this problem?

- How **several software engineers <u>can work simultaneously</u> on the same project** when all C code is in the `main` ?

- If the **same set of statements** has to be used in a p**rogram several times**, **how to avoid code duplication**?

Solution: Modular Design

- Major task of **software design** is how to **split system's data and its processing into a set of <u>interacting functions</u>**



II. Modular Design and Coding

- C is a **structured modular programming language**
  - **Allows to solve large problems by dividing into smaller ones**: modules, called functions
  - A **function** is a **block of code that performs a specific task**
    - It has a **name** and it is **reusable** i.e. it can be executed from as many different parts in a C program as required.
    - e.g. Program to process students' grades - can use several functions (calculate grade, display record, get record, input marks etc)
- **Functions** can call other functions; even themselves (recursion)

Advantages of Use of Function

- **Functions** are useful to
  - **decompose a complex program into manageable parts**
  - **reduce duplication of code** (it can be executed from as many different parts in a program as required), **enable code re-use, and ease future amendments and maintenance of the code**
- Functions allow **code reusability:**
  - **Functions** can be created for problems that **generally need frequently used solutions**
  - A common example we’ve seen so far is using **system functions: printf()** and **scanf().** We have constantly been calling the function for reuse.

Modular Design: Top Down Design

- **Top Down Design breaks the specification down into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used.**
  - Break down program into small and manageable components, starting from the top. In C, this is your **main()** function. From **main()** your other smaller components will be used

Example

- **Task:** program a virtual ATM
  - What functions do we need?
    - ATM needs to display balance, withdraw money, deposit/transfer money, etc.
    - Then consider breaking those individual components further to make the implementation more manageable.

- Take the *withdraw* function, break it down further:
  - Get current balance
  - Compare balance to amount requested
  - If OK
    - Update balance
    - Distribute money
    - Print a message
  - If not OK
    - Decline and print message

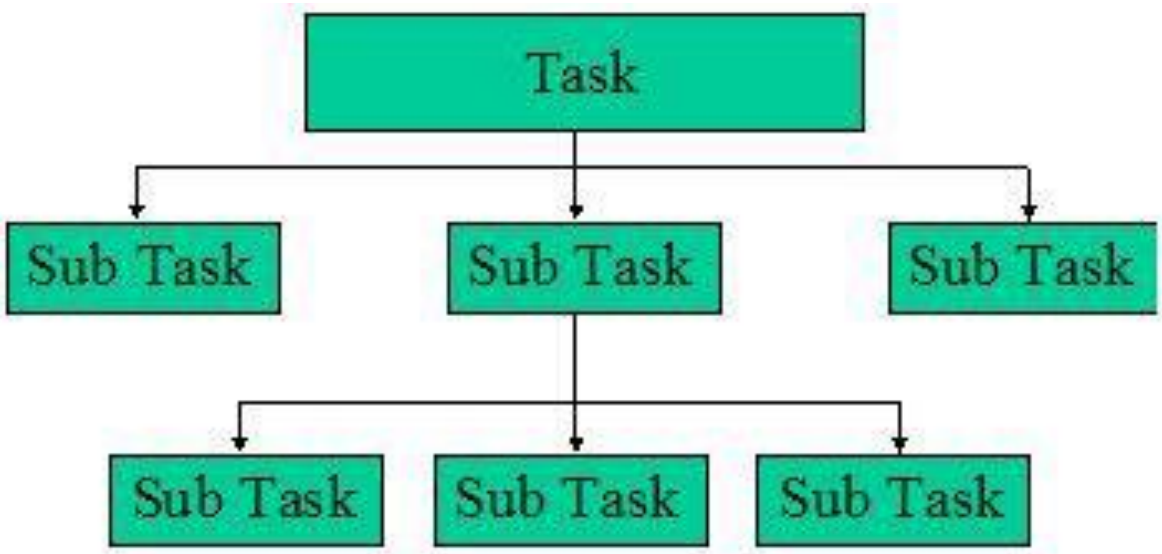
…Continued

- Do we need to go further? Take “Distribute Money” part/function
  - Verify the ATM has enough money to dispense
  - Initiate (virtual) mechanical process
  - Update bank record
  - Update ATM current money
  - Contact bank if money is low
  - ….

- If used properly, Top Down Programming is a useful tool **for making problems easier to understand and implement**
  - Look at main functionality → break it down into smaller problems

Structure Chart

- Structure charts are drawn **top-down**
- Each rectangle in the chart represents a function, its name to be used as the function name in your program
- **Function chart** describes **only relationship between functions**
- Function chart **does not tell when and how many times each function is called**



Top Down Design

- ❖ **Advantages:**
  - ❑ Typically smoother and shorter process planned properly and the programmer has a good understanding of the task
  - ❑ Decisions can be made and implemented much quicker - useful when time is a factor!
  - ❑ Develop and test most important function first (main())
  - ❑ Easy to see progress
  - ❑ Testing and debugging is easier

Modular Design: Bottom Up Approach

- An alternative approach: starts with low level system design → work way up!
  - Example: A calculator:
    - Look at the problem objectively - e.g. what are the mathematical functions I would need to design a calculator?
    - Design the modules/functions like addition/subtraction, then decide how it will be called and then design main function around it
    - main() function will hold code together as a sequence

III. Function Definition in C

```
type function_name (type1 par1, type2 par2,.. )
{
      statements;
}
```

```
int product(int x, int y)
{
  int result;
  result = x * y;
  return result;
}
```

- Function "header"
  - **type:** return data type
    - – void if no value is returned
  - ***function_name:*** *the* ***name*** of the *function*
    - – choose a descriptive name
  - Parameters: par1, par2,..of data types type1, type2, etc…
    - – void or () if no other data is passed into the function
  - Statements
    - – variable declaration
    - – operations
    - – return value (if any)
      - Parentheses around the expression to return is <u>optional</u>

...Continued

- *In C,*
  - Functions can return up to one value to the calling program through a **return** statement.
  - Functions of type **void** do not return a value
    - end the function with **return;**
  - Functions can however have multiple return statements, written anywhere in the function body.
    - The first return executed **exits** the function code execution

Note: A C function can also return a struct value, more in the coming weeks.

Function Prototype and Execution

- A function prototype is written before main(). It tells the compiler that the function definition will come later and gives the function a file scope
- With a **Function call,** program execution(control flow) passes to the function code
- Function return statement terminates the execution of the current function
  - program control flow **passes back** to the point immediately after the function call

```c
#include <stdio.h>
/* function prototype */
int product(int x, int y);

int main(){
  int var1 = 3, var2 = 5;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
  return 0;
}
/* function definition */
int product(int x, int y)
 {
   int result;
   result = x * y;
   return result;
 }
```

Function definition

Function Prototype vs. Function Definition

- **Function Prototype:**
  - outside and before main()
  - only needs to define variable data type for input and output, **the variable names are OPTIONAL**
  - has a semicolon at its end
- **Function definition:**
  - **header of the function** uses the same name as prototype and contains **actual** variables to be used in the function implementation
  - must have { }
  - no semicolon after the header
  - **Can** use return value

IV. Scope of Variable

- Does it make a difference ***where*** in a program a variable is declared?
  - YES! --> concept of SCOPE
- **Scope** refers to the region in the program where the variable can be **referenced.**
- Five types of scope:
  - Program (global scope)
  - File
  - Function prototype
  - Function
  - Block (“between the { } scope”)

Program Scope

- **Program (global) scope**
  - if declared *outside* of all functions
  - Visible to ***all*** functions from point of declaration
  - Visible to functions in other source files (SHARED)
  - It is kept throughout the life of your program
  - Can be easily modified by any function and can lead to errors

```c
#include <stdio.h>
int a = 10;
int product(int x, int y);

int main()
{
  int var1 = 3, var2 = 5;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
}
/* function definition */
int product(int x, int y)
 {
   int result;
   result = x * y + a;
   return result;
 }
```

Example

```
#include <stdion.h>
void scaleNumber(int ); // function prototype
int luckyNum; // global variable

int main(){
int factor=10;
printf("enter a num");
scanf("%d", &luckyNum); // update variable with input
scaleNumber(factor);
printf("\n luckyNum= %d", luckyNum);
return 0;
}
void scaleNumber(int scale) // function declaration
{
luckyNum*=scale;
}
```

File Scope

- **File scope**
  - Keyword **static**
    - – Makes a variable "visible" only within this source file
  - Use file scope to avoid **naming conflict** if **multiple source files** are used

```
#include <stdio.h>
static int a = 10;
int product(int x, int y);

int main()
{
  int var1 = 3.0, var2 = 5.0;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
}
/* function definition */
int product(int x, int y)
 {
   int result;
   result = x * y;
   return result;
 }
```

Function Prototype Scope

- **Function prototype scope**
  - Identifiers x and y are *not* visible outside the prototype
  - Thus, names in the prototype do not have to match names in the function definition
    - MUST **match types,** however!

```c
#include <stdio.h>
int product(int x, int y);

int main()
{
  int a = 10;
  int var1 = 3.0, var2 = 5.0;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
}
/* function definition */
int product(int A, int B)
 {
   int result;
   result = A * B;
   return result;
 }
```

Function & Block Scopes

❖ **Function** scope

- Active from the beginning to the end of a **function execution ONLY**
- **Example:** each time the function *product* is **run,** C allocates memory space for the integer variable *result* with its variable declaration.
- Data stored in the variable is **lost** when the *product* function **is terminated**

```c
#include <stdio.h>
int product(int x, int y);

int main()
{
  int a = 10;
  int var1 = 3.0, var2 = 5.0;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
}
/* function definition */
int product(int x, int y)
 {
   int result;
   int a= -33;
    result = x * y+a;
   return result;
 }
```

Example

- Because the variable *num1* is inside the *main()* code, the same variable *name* can be used in other functions **without any conflict.** The two variables when allocated in the memory occupy **different memory locations**

```
#include<stdio.h>
int getNextNum(); // function prototype

int main(){
int num1;
printf("\n Enter num1 to the main");
scanf ("%d", &num1);
printf("num1 entered %d and num1 returned %d", num1, getNextNum());
}

int getNextNum() { // function definition
int num1;
printf("\n Enter num1 to the called function");
scanf ("%d", &num1);
return num1;
 }
```

Block Scope

- **Block** (local) scope
  - A block is a series of statements enclosed in braces { }
  - The variable scope is active from the point of declaration to the end of the block (**}**)
  - Nested blocks can *both* declare the same variable name and not interfere

Example

- What kind of scope do the variables have?
  - i
  - j
  - m
  - k

```
#include <stdio.h>
int i;
static float m;
int k=10;
int main()
{
  int j;
  for(j=0; j<5; ++j)
  {
    printf("\n j= %d", j);
  }

  {
    int k=7;
    printf("\n k= %d", k);
  }

}
```

V. Passing Arguments into Functions

- ❖ In Programming, caller arguments are passed to the called functions by **value or reference**
  - ❑ **Pass by value: the values of the arguments are passed to the function**
    - – Function arguments can be *expressions* which get evaluated. Their resulting values are then passed to the called function
  - ❑ Pass by reference: **the addresses** of the arguments, i.e., references to the caller's variables are passed to the function's parameters so **it can change** their memory content.

☞ **Pass by Value:** the function **does NOT** change the values of the passed-in arguments

☞ **Pass by Reference: the function can** change the values of the passed-in arguments

C language

- **C language always uses "Pass by value".** However**, we can simulate/implement "Pass by reference"** through the use of Pointers.
  - A **Pointer** is a variable that stores a memory address
  - To get multiple values returned, we often use **pointers** in the arguments of the function and its return value
  - More in the next lectures…

Example 1: C Pass By Value

❖ Example: in the function call:

- The **values** of the **caller's** arguments are passed to the corresponding parameters of the **calle**d function
- The values of *var1* and *var2* are passed/copied to the function parameters: A and B
  - var1 and var2 are NOT changed

```
#include <stdio.h>
int product(int x, int y);

int main()
{
  int a = 10;
  int var1 = 3, var2 = 5;
  int ans;
  ans = product(var1, var2);
  printf("var1 = %d\n"
         "var2 = %d\n",var1,var2);
  printf("var1*var2 = %d\n", ans);
}
/* function definition */
int product(int A, int B)
 {
   int result;
   result = A * B;
   return result;
 }
```

Example 2: C Pass By Value

```
#include<stdio.h>

void PassByValue(int); // prototype

int main(){

int x = 0;

printf("Enter value for x");

scanf("%d", &x);   // enter, assume to be 50

PassByValue(x); // call function and pass x

printf("\n x value after the function call is %d", x);  // print 50

}

void PassByValue(int x) {

x += 5;              // add 5 to passed in x

printf("\n the value of x from the called function is %d", x);  // print 55

}
```