Programming in C

Based on the Original Slides from
Politehnica International- Computer Engineering
Lecture Slides
Outline

• Functions
  – Defining a Function
  – Arguments and Local Variables
    • Automatic Local Variables
  – Returning Function Results
  – Declaring a Function Prototype
  – Array Parameters
  – Global Variables
  – Automatic and Static Variables
  – Recursive Functions
What is a function

• A function in C: is a self-contained unit of program code designed to accomplish a particular task.
• The concept has some equivalent in all high-level programming languages: functions, subroutines, and procedures
• The use of a function: a "black box"
  – defined in terms of the information that goes in (its input) and the value or action it produces (its output).
  – what goes on inside the black box is not your concern, unless you are the one who has to write the function.
  – Think on how you used functions printf, scanf, getchar!
• What kind of “output” comes out from a function black box?
  – Some functions find a value for a program to use. Example: getchar() returns to the program the next character from the standard input buffer.
  – Some functions cause an action to take place. Example: printf() causes data to be printed on the screen
  – In general, a function can both produce actions and provide values.
Defining a function

```c
#include <stdio.h>

void printMessage (void)
{
    printf ("Programming is fun.\n");
}

int main (void)
{
    printMessage ();
    printMessage ();
    return 0;
}
```

**Function Definition**
- occurs ONE time for all
- outside other functions

**Function calls (invocations)**
- occurs ANY (0-N) times
- statement inside (other) functions body
Transfer of control flow

When a function call is executed, program execution is transferred directly to the indicated function. After the called routine is finished (as signaled by the closing brace) the program *returns* to the calling routine, where program execution continues at the point where the function call was executed.
Function definitions

General form of function definition:

\[
\text{return-type function-name}(\text{argument declarations})
\{
    \text{declarations and statements}
\}
\]

```c
void printMessage (void)
{
    printf ("Programming is fun.
");
}
```

```c
void printMessage (void)
{
    printf ("Programming is fun.
");
    return;
}
```
Function prototype

- The first line of the function definition
- Contains everything that others need to know about the function
- void calculateTriangularNumber (int n)
- arguments (parameters): a kind of input for the function blackbox

**In the function definition: formal arguments (formal parameters)**
- Formal parameter: a name that is used inside the function body to refer to its argument

**In the function call: actual arguments (actual parameters)**
- The actual arguments are values are assigned to the corresponding formal parameters.
- The actual argument can be a constant, a variable, or an even more elaborate expression.
- The actual argument is evaluated, and its value is **copied** to the corresponding formal parameter for the function.
  - Because the called function works with data copied from the calling function, the original data in the calling function is protected from whatever manipulations the called function applies to the copies.
// Function to calculate the nth triangular number
#include <stdio.h>
void calculateTriangularNumber ( int n )
{
    int i, triangularNumber = 0;  // formal argument
    for ( i = 1; i <= n; ++i )
        triangularNumber += i;
    printf ("Triangular number %i is %i\n", n, triangularNumber);
}
int main (void)
{
    calculateTriangularNumber (10);
    calculateTriangularNumber (20);
    calculateTriangularNumber (50);
    return 0;
}
Arguments and local variables

- Variables defined inside a function: *automatic local variables*
  - they are automatically “created” each time the function is called
  - their values are local to the function:
    - The value of a local variable can only be accessed by the function in which the variable is defined
    - Its value cannot be accessed by any other function.
    - If an initial value is given to a variable inside a function, that initial value is assigned to the variable *each* time the function is called.

- Formal parameters: behave like local variables, private to the function.

- **Lifetime**: Period of time when memory location is allocated
- **Scope**: Region of program text where declaration is visible
- Scope: local variables and formal parameters => only in the body of the function
  - Local variable i in function `calculateTriangularNumber` is different from a variable i defined in another function (including main)
  - Formal parameter n in function `calculateTriangularNumber` is different from a variable n defined in another function
Automatic local variables

```cpp
void calculateTriangularNumber {
    int n = 10; // Local variable declared in calculateTriangularNumber
    int i = 0;
    int triangularNb = 0;
    for (i = 1; i <= n; i++) {
        triangularNb += i;
    }
    // return triangularNb;
}

int main {
    int n = 20; // Local variable declared in main
    int i = 0;
    int triangularNb = 0;
    for (i = 1; i <= n; i++) {
        triangularNb += i;
    }
    // return triangularNb;
}
```
Example: arguments

```c
#include <stdio.h>
void gcd (int u, int v)
{
    int temp;
    printf ("The gcd of %i and %i is ", u, v);
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    printf ("%i\n", u);
}
int main (void)
{
    gcd (150, 35);
gcd (1026, 405);
gcd (83, 240);
    return 0;
}
```

each declared parameter must be preceded by its type. That is, unlike the case with regular declarations, you can't use a list of variables of the same type, as (int u,v)
Example: arguments are passed by copying values!

```c
#include <stdio.h>

void gcd (int u, int v)
{
    int temp;
    printf ("The gcd of %i and %i is ", u, v);
    while ( v != 0 ) {
        temp = u % v;
        u = v;
        v = temp;
    }
    printf ("%i\n", u);
}

int main (void)
{
    int x=10, y=15;
    gcd (x, y);
    printf("x=%i y=%i \n", x, y);
    return 0;
}
```

The formal parameters `u` and `v` are assigned new values in the function.

The actual parameters `x` and `y` are not changed!
Example: arguments are passed by copying values!

```c
#include <stdio.h>
void multiplyBy2 (float x)
{
    printf("parameter at start: %.2f, at %p \n", x, &x);
    x *= 2;
    printf("parameter at end: %.2f, at %p \n", x, &x);
}
int main (void)
{
    float y = 7;
    printf ("y before call: %.2f, at %p \n", y, &y);
    multiplyBy2 (y);
    printf ("y after call: %.2f, at %p \n", y, &y);
    return 0;
}
```
Example: scope of local variables

```c
#include <stdio.h>
void f1 (float x)  {
    int n=6;
    printf("%f \n", x+n);
}
int f2(void) {
    float n=10;
    printf("%f \n",n);
}
int main (void)
{
    int n=5;
    f1(3);
    f2();
    return 0;
}
```
Returning function results

- A function in C can optionally return a single value
  
  ```
  return expression;
  ```

- The value of `expression` is returned to the calling function. If the type of `expression` does not agree with the return type declared in the function declaration, its value is automatically converted to the declared type before it is returned.

- A simpler format for declaring the return statement is as follows:
  
  ```
  return;
  ```

- Execution of the simple return statement causes program execution to be immediately returned to the calling function. This format can only be used to return from a function that does not return a value.

- If execution proceeds to the end of a function and a return statement is not encountered, it returns as if a return statement of this form had been executed. Therefore, in such a case, no value is returned.

- If the declaration of the type returned by a function is omitted, the C compiler assumes that the function returns an int!
/* Function to find the greatest common divisor of two nonnegative integer values and to return the result */
#include <stdio.h>
int gcd (int u, int v)
{
    int temp;
    while (v != 0)
    {
        temp = u % v;
        u = v;
        v = temp;
    }
    return u;
}

int main (void)
{
    int result;
    result = gcd (150, 35);
    printf("The gcd of 150 and 35 is %i\n", result);
    result = gcd (1026, 405);
    printf("The gcd of 1026 and 405 is %i\n", result);
    printf("The gcd of 83 and 240 is %i\n", gcd (83, 240));
    return 0;
}
Function declaration

• a function prototype—a declaration that states the return type, the number of arguments, and the types of those arguments.
• Useful mechanism when the called function is defined after the calling function
• The prototype of the called function is everything the compiler needs in order to be able to compile the calling function
• In order to produce the executable program, of course that also the whole definition of the function body is needed, but this occurs later, in the process of linking
Example: function declaration

```c
#include <stdio.h>

void printMessage (void) ;

int main (void)
{
    printMessage () ;
    printMessage () ;
    return 0 ;
}

void printMessage (void)
{
    printf ("Programming is fun.\n") ;
}
```

Function declaration (prototype)  
(has to be before the calling function)

Function calls

Function definition  
(can be after the calling function)
#include now explained

- `#include <stdio.h>`
- `#include <filename>` is a preprocessor directive
- Preprocessor: a first step in the C compilation process
- Preprocessor statements are identified by the pound sign `#` that must be the first nonspace character of a line
- The `#include` directive will insert in place the contents of the specified file
- These files usually have names that end with `.h` (header files)
- Header files usually contain declarations and definitions that are used by several programs
- `<stdio.h>` contains the declarations for the standard input output functions `printf`, `scanf`, `getchar`, etc. This is why any program that uses these functions has to include `<stdio.h>`
Examples: function declarations

In a function declaration you have to specify the argument type inside the parentheses, and not its name. You can optionally specify a “dummy” name for formal parameters after the type if you want.

```c
int gcd (int u, int v);
Or
int gcd (int, int);

void calculateTriangularNumber (int n);
Or
void calculateTriangularNumber (int);
```
Passing arrays as parameters

- A whole array can be one parameter in a function
- **In the function declaration, you can then omit the specification of the number of elements contained in the formal parameter array.**
  - The C compiler actually ignores this part of the declaration anyway; all the compiler is concerned with is the fact that an array is expected as an argument to the function and not how many elements are in it.
- Example: a function that returns the minimum value from an array given as parameter
  - int minimum (int values[10]);
    - We must modify the function definition if a different array size is needed!
  - int minimum (int values[]);
    - Syntactically OK, but how will the function know the actual size of the array ?!
  - int minimum (int values[], int numberOfElements);
Computing the minimum/maximum

```c
#include <limits.h>

int minValue, i;
minValue = INT_MIN;
for ( i = 0; i < 10; ++i )
    if ( values[i] < minValue )
        minValue = values[i];
```

```c
int values[10];
int minValue, i;
minValue = values[0];
for ( i = 1; i < 10; ++i )
    if ( values[i] < minValue )
        minValue = values[i];
```
Example: Passing arrays as parameters

// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[10]) {
    int minValue, i;
    minValue = values[0];
    for ( i = 1; i < 10; ++i )
        if ( values[i] < minValue )
            minValue = values[i];
    return minValue;
}

int main (void) {
    int scores[10], i, minScore;
    printf ("Enter 10 scores\n");
    for ( i = 0; i < 10; ++i )
        scanf ("%i", &scores[i]);
    minScore = minimum (scores);
    printf ("\nMinimum score is %i\n", minScore);
    return 0;
}
Example: No size specified for formal parameter array

// Function to find the minimum value in an array
#include <stdio.h>
int minimum (int values[], int numberOfElements)
{
    int minValue, i;
    minValue = values[0];
    for ( i = 1; i < numberOfElements; ++i )
        if ( values[i] < minValue )
            minValue = values[i];
    return minValue;
}
int main (void)
{
    int array1[5] = { 157, -28, -37, 26, 10 };
    int array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
    printf ("array1 minimum: %i\n", minimum (array1, 5));
    printf ("array2 minimum: %i\n", minimum (array2, 7));
    return 0;
}
Array parameters are passed by reference!

- If a function changes the value of an array element, that change is made to the original array that was passed to the function. This change remains in effect even after the function has completed execution and has returned to the calling routine.
- Parameters of non-array type: passed by copying values
- Parameters of array type: passed by reference
  - the entire contents of the array is \textit{not} copied into the formal parameter array.
  - the function gets passed information describing \textit{where} in the computer’s memory the original array is located.
  - Any changes made to the formal parameter array by the function are actually made to the original array passed to the function, and not to a copy of the array.
Example: Array parameters are passed by reference!

```c
#include <stdio.h>
void multiplyBy2 (float array[], int n)
{
    int i;
    for ( i = 0; i < n; ++i )
        array[i] *= 2;
}
int main (void)
{
    float floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };   
    int i;
    multiplyBy2 (floatVals, 4);
    for ( i = 0; i < 4; ++i )
        printf ("%.2f ", floatVals[i]);
    printf ("\n");
    return 0;
}
```
// Program to sort an array of integers
// into ascending order

#include <stdio.h>
void sort (int a[], int n) {
    int i, j, temp;
    for ( i = 0; i < n - 1; ++i )
        for ( j = i + 1; j < n; ++j )
            if ( a[i] > a[j] ) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
}
void sort (int a[], int n);

int main (void)
{
    int i;
    int array[16] = { 34, -5, 6, 0, 12, 100, 56, 22, 44, -3, -9, 12, 17, 22, 6, 11 };

    printf ("The array before the sort:\n");
    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);
    sort (array, 16);
    printf ("\n\nThe array after the sort:\n");
    for ( i = 0; i < 16; ++i )
        printf ("%i ", array[i]);
    printf ("\n\n");
    return 0;
}
Multidimensional arrays and functions

- When declaring a single-dimensional array as a formal parameter inside a function, the actual dimension of the array is not needed; simply use a pair of empty brackets to inform the C compiler that the parameter is, in fact, an array.
- This does not totally apply in the case of multidimensional arrays. For a two-dimensional array, the number of rows in the array can be omitted, but the declaration must contain the number of columns in the array.

- Valid examples:
  ```c
  function(int array_values[100][50]);
  function(int array_values[][50]);
  ```

- Invalid examples:
  ```c
  function(int array_values[100][]);
  function(int array_values[][[]]);
  function(int array_values[][[]][]);
  ```
Example: multidimensional array as function parameter

The number of columns must be specified!
No generic matrix display function possible!

```c
void displayMatrix (int matrix[3][5]) {
    int row, column;
    for (row = 0; row < 3; ++row) {
        for (column = 0; column < 5; ++column) {
            printf ("%5i", matrix[row][column]);
            printf ("\n");
        }
    }
}
```
Example: multidimensional variable length array as function parameter

A generic matrix display function is possible with the variable length array feature. The rows and columns must be listed as arguments before the matrix itself.

```c
void displayMatrix (int nRows, int nCols,
                   int matrix[nRows][nCols])
{
    int row, column;
    for ( row = 0; row < nRows; ++row) {
        for ( column = 0; column < nCols; ++column )
            printf ("%5i", matrix[row][column]);
        printf ("\n");
    }
}
```
Global variables

• A global variable declaration is made *outside* of any function.
• It does not belong to any particular function. *Any* function in the program can then access the value of that variable and can change its value.
• The primary use of global variables is in programs in which many functions must access the value of the same variable. Rather than having to pass the value of the variable to each individual function as an argument, the function can explicitly reference the variable instead.
• There is a **drawback** with this approach: Because the function explicitly references a particular global variable, the generality of the function is somewhat reduced!
• Global variables do have default initial values: zero
Example: global variables

```c
#include <stdio.h>

int x;

void f1 (void) {
    x++;
}

void f2 (void) {
    x++;
}

int main(void) {
    x=7;
    f1();
    f1();
    f2();
    printf("x=%i \n",x);
}
```
Automatic and static variables

• Automatic local variables:
  – an automatic variable disappears after the function where it is defined completes execution, the value of that variable disappears along with it.
  – the value an automatic variable has when a function finishes execution is guaranteed not to exist the next time the function is called.
  – The value of the expression is calculated and assigned to the automatic local variable each time the function is called.

• Static local variables:
  – If you place the word static in front of a variable declaration
  – “something that has no movement”
  – a static local variable—it does not come and go as the function is called and returns. This implies that the value a static variable has upon leaving a function is the same value that variable will have the next time the function is called.
  – Static variables also differ with respect to their initialization. A static, local variable is initialized only once at the start of overall program execution—and not each time that the function is called. Furthermore, the initial value specified for a static variable must be a simple constant or constant expression. Static variables also have default initial values of zero, unlike automatic variables, which have no default initial value.
Example: Automatic and static variables

// Program to illustrate static and automatic variables
#include <stdio.h>
void auto_static (void)
{
    int autoVar = 1;
    static int staticVar = 1;
    printf ("automatic = %i, static = %i\n", autoVar, staticVar);
    ++autoVar;
    ++staticVar;
}
int main (void)
{
    int i;
    for ( i = 0; i < 5; ++i )
        auto_static ();
    return 0;
}
Recursive functions

- C permits a function to call itself. This process is named recursion.
- Useful when the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem.
- Example: factorial: recursive definition:
  - \( n! = n \times (n-1)! \)
  - factorial(n) = factorial(n-1)
Example: recursive function

// Recursive function to calculate the factorial of n
unsigned long int factorial (unsigned int n)
{
    unsigned long int result;
    if ( n == 0 )
        result = 1;
    else
        result = n * factorial (n - 1);
    return result;
}

factorial(3) = 3 * factorial(2); = 6
  ↓  ↑
factorial(2) = 2 * factorial(1); = 2
       ↓  ↑
factorial(1) = 1 * factorial(0); = 1
            ↓  ↑
factorial(0) = 1
Recursive function calls

- Each time any function is called in C—be it recursive or not—the function gets its own set of local variables and formal parameters with which to work!
- These local automatic variables are stored in a memory area called stack
- Each new function call pushes a new activation record on the stack
- This activation record contains its set of automatic local variables
- When a function call returns, its activation record is removed from the top of the stack

The local variable result and the formal parameter n that exist when the factorial function is called to calculate the factorial of 3 are distinct from the variable result and the parameter n when the function is called to calculate the factorial of 2.
Example: recursive function calls

```c
void up_and_down(int n)
{
    printf("Start call %d: n location %p\n", n, &n);
    if (n < 4)
        up_and_down(n+1);
    printf("End call %d: n location %p\n", n, &n);
}

... 

up_and_down(0);
```
Recursion pros and cons

- Tricky with recursion: programmer must make sure to get the recursion to an end at some time!
  - a function that calls itself tends to do so indefinitely unless the programming includes a conditional test to terminate recursion.
- Recursion often can be used where loops can be used. Sometimes the iterative solution is more obvious; sometimes the recursive solution is more obvious.
- Recursive solutions tend to be more elegant and less efficient than iterative solutions.
Functions - Summary

• We distinguish between **Function definition** and **Function call**

• Function definition general format:

```plaintext
returnType name ( type1 param1, type2 param2, ... )
{
    variableDeclarations
    programStatement
    programStatement
    ...
    return expression;
}
```

• The function called *name* is defined, which returns a value of type *returnType* and has formal parameters *param1, param2,*... . *param1* is declared to be of type *type1, param2* is declared to be of type *type2, etc.*
Function Definitions - Summary

• Local variables are typically declared at the beginning of the function, but that's not required. They can be declared anywhere, in which case their access is limited to statements appearing after their declaration in the function.

• If the function does not return a value, returnType is specified as void.

• If just void is specified inside the parentheses, the function takes no arguments.

• Declarations for single-dimensional array arguments do not have to specify the number of elements in the array. For multidimensional arrays, the size of each dimension except the first must be specified.
Function Calls - Summary

- `name (arg1, arg2, ...)`
- The function called `name` is called and the values `arg1, arg2, ...` are passed as arguments to the function. If the function takes no arguments, just the open and closed parentheses are needed.
- If you are calling a function that is defined after the call, or in another file, you should include a *prototype declaration* for the function, which has the following general format:
  - `returnType name (type1, type2, ...);`
- This tells the compiler the function’s return type, the number of arguments it takes, and the type of each argument.
- A function whose return type is declared as void causes the compiler to flag any calls to that function that try to make use of a returned value.
- All arguments to a function are passed by value; therefore, their values cannot be changed by the function. Exception from this rule are arrays passed as parameters.