COMP0204: Introduction to Programming for Robotics and AI

Random numbers, Functions and Scope

Course lead: Dr Sophia Bano

MEng Robotics and AI

UCL Computer Science

Recap (previous week)

- Information representation – understanding how computer sees the data
- Design and development – understanding how to develop a program
- Control flows – if-else, for, while, switch

Control Instructions

- Used to determine flow of program
  a. Sequence control – instructions run in sequence
  b. Decision control – if- else conditional statements
  c. Case control – switch - do separate things given a case
  d. Loop control – for, while loops – to do a task repeatedly

if Conditional Statement

Exercise

Write a C program that takes a year as input and determines whether it is a leap year or not. A leap year is defined as follows:

- If the year is evenly divisible by 4, it is a leap year.
- However, if the year is evenly divisible by 100, it is not a leap year.
- But, if the year is evenly divisible by 400, it is still a leap year.

if Conditional Statement

Solution

```c
#include <stdio.h>
/*Ex 2_1:
    if else practice
    Write a program to check if a user input year is leap year or not.
*/

int main() {
    int year;

    // Input: Get the year from the user
    printf("Enter a year: ");
    scanf("%d", &year);

    // Check if it's a leap year
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }

    return 0;
}
```

```c
#include <stdio.h>
/*Ex 2_1:
    if else practice
    Write a program to check if a user input year is leap year or not.
*/

#include <stdio.h>

int main() {
    int year;

    // Input: Get the year from the user
    printf("Enter a year: ");
    scanf("%d", &year);

    // Check if it's a leap year
    if (year % 4 == 0) {
        if (year % 100 == 0) {
            if (year % 400 == 0) {
                printf("%d is a leap year.\n", year);
            } else {
                printf("%d is not a leap year.\n", year);
            }
        } else {
            printf("%d is a leap year.\n", year);
        }
    } else {
        printf("%d is not a leap year.\n", year);
    }

    return 0;
}
```

switch Conditional Statement

Exercise

Write a C program that takes a numerical grade as input (0-100) and calculates the corresponding letter grade based on the following grading scale:

- A: 90-100
- B: 80-89
- C: 70-79
- D: 60-69
- F: 0-59

Use a **switch** statement to determine and display the letter grade for the input grade.

switch Conditional Statement

Solution

```c
#include <stdio.h>

int main() {
    int numGrade;
    char letGrade;

    // Input: Get the numerical grade from the user
    printf("Enter the numerical grade (0-100): ");
    scanf("%d", &numGrade);

    // Determine the letter grade using a switch statement
    switch (numGrade / 10) {
        case 10:
        case 9:
            letGrade = 'A';
            break;
        case 8:
            letGrade = 'B';
            break;
        case 7:
            letGrade = 'C';
            break;
        case 6:
            letGrade = 'D';
            break;
        default:
            letGrade = 'F';
            break;
    }

    // Output: Display the calculated letter grade
    printf("The letter grade is: %c\n", letGrade);

    return 0;
}
```

Loops - Practice Example

Exercise

Write a C program that takes a positive integer num as input, display the sum of all even numbers from 1 to num.

Use for loop to implement

Use while loop to implement

Use do while loop to implement

Loops - Practice Example

for loop

```c
#include <stdio.h>

int main() {
    int n;
    int sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1; // Exit the program with an error code
    }

    for (int i = 2; i <= n; i += 2) {
        sum += i;
    }

    printf("The sum of even numbers from 1 to %d is %d.\n", n, sum);

    return 0;
}
```

while loop

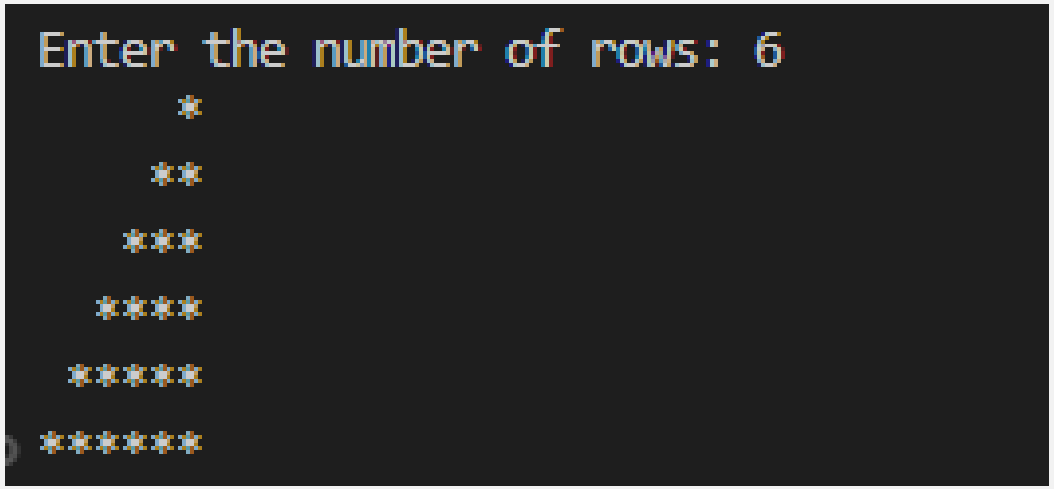
```c
#include <stdio.h>

int main() {
    int n;
    int sum = 0;
    int i = 2; // Start with the first even number

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1; // Exit the program with an error code
    }

    while (i <= n) {
        sum += i;
        i += 2; // Move to the next even number
    }

    printf("The sum of even numbers from 1 to %d is %d.\n", n, sum);

    return 0;
}
```

do while loop

```c
#include <stdio.h>

int main() {
    int n;
    int sum = 0;
    int i = 2; // Start with the first even number

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 1; // Exit the program with an error code
    }

    do {
        sum += i;
        i += 2; // Move to the next even number
    } while (i <= n);

    printf("The sum of even numbers from 1 to %d is %d.\n", n, sum);

    return 0;
}
```

Right-sided tringle of stars

- Take the number of rows as input from the user. Write a C program using nested for loops that print a right-sided right-angle triangle of stars where the number of rows are input from the user. The output should look like the following:

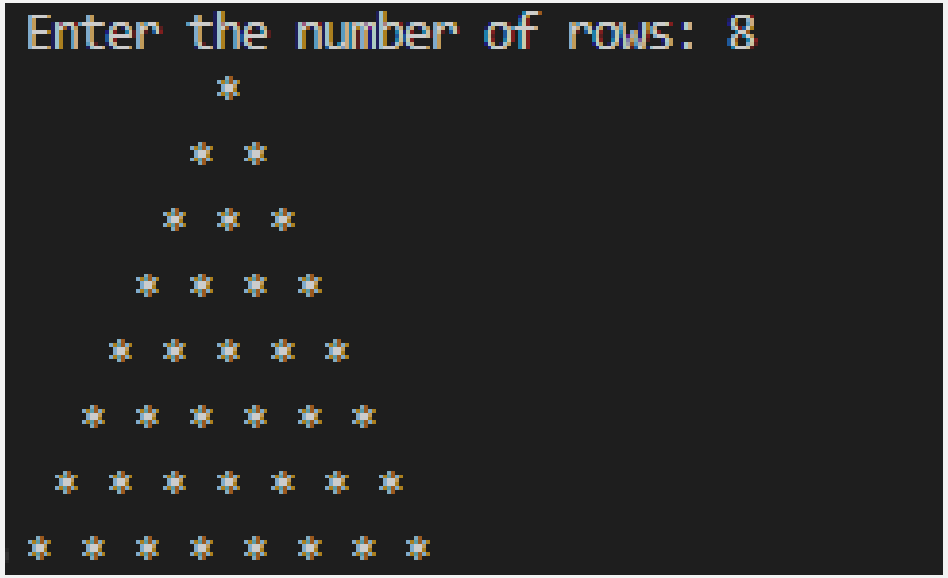


Right-sided tringle of stars (solution)

```c
#include <stdio.h>

int main() {
    int rows; // Number of rows for the pattern

    printf("Enter the number of rows: ");
    scanf("%d", &rows);

    // Outer loop for rows
    for (int i = 1; i <= rows; i++)
        {
        // Inner loop to print spaces
        for (int j = 1; j <= rows - i; j++)
        {
            printf(" ");
        }

        // Inner loop to print stars
        for (int k = 1; k <= i; k++)
        {
            printf("*");
        }

        // Move to the next line
        printf("\n");
    }

    return 0;
}
```

Star Pyramid

- Take the number of rows as input from the user. Write a C program using nested for loops that print a pyramid of stars where the number of rows are input from the user. The output should look like the following:



Star Pyramid

- Solution

```c
#include <stdio.h>

int main() {
    int rows; // Number of rows for the pattern

    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    for (int i = 1; i <= rows; i++)
        {
        // Print spaces before the stars in each row
        for (int j = 1; j <= rows - i; j++)
        {
            printf(" ");
        }

        // Print stars in each row
        for (int k = 1; k <= i; k++)
        {
            printf("* ");
        }

        // Move to the next line after each row
        printf("\n");
    }

    return 0;
}
```

This week

- Random numbers
- Functions
- Variable scope
- Recursion
- Arrays and String

Random Numbers

Random Numbers

- C language has the ability to generate the random numbers
  - Random numbers are useful in game designing, mathematical simulations, experimental evaluations etc.
- In C language we can use the **rand** function which generates a random integer between 0 and a defined maximum value (usually 32767)
- The description of the **rand** function is:

```
int rand(void)
```

- This means that it does not have any input (void) and it **returns** (or outputs) an integer value

Using the rand function

- The rand function could be used as follows:

```
int randomNumber = 0;
randomNumber = rand();
```

- How to restrict the range of the generated random number?
  - Use the modulo operator (%) – gives the reminder

- To generate a random number between 0 and 99

```
randomNumber = rand() % 100;
```

Are the numbers really random?

- In C there is no such think as truly random number generation
- C language actually generate pseudo-random numbers
  - They use an iterative equation to generate each new random number.
  - The first random number generated is called the seed of the equation.
  - The next random numbers are generated on the basis of the value of seed.
  - If the seed is changed we can generate different random numbers
  - If this seed is not changed then each time when we run the program the SAME sequence of random numbers will be generated!!

Example

```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int rand_num = 0;

    for ( int i = 0; i < 5; i++ )
    {
        rand_num =rand()%50;
        printf("%d\n",rand_num);
    }

    return 0;
}
```

- Observe same output every time this is run

Changing the Seed

- The C function **srand** is used to specify the seed to be used for the random number generation equation
- If you do not use this function in your program then a fixed value is used as the seed
- The decription for the **srand** function is

  void srand(int seed)

- The function has one input (the seed number) and has no output

Changing the Seed

- If you want the sequence of random numbers generated by rand() to be different each time a program runs then a different seed must be used each time that the program runs
- You could ask the user of the program to enter a value for the seed at the very start of the program!
- There is an easier way – use the internal time on the computer's clock
- Clearly, this will be different each time the program is run!!

Using the System's Clock

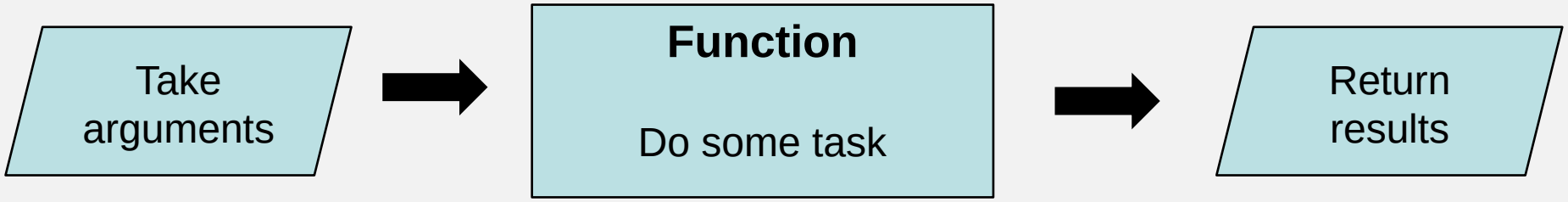
- To do this, we the **time(NULL)** function and passes its value to **srand**.
- **time(NULL)** returns the current time in seconds since the Unix epoch (January 1, 1970).

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int main()
{
    int rand_num = 0;

    // Seed the random number generator
    srand(time(NULL));

    for ( int i = 0; i < 5; i++ )
    {
        rand_num =rand()%50;
        printf("%d\n",rand_num);
    }

    return 0;
}
```

Functions

Functions

- Block of code that performs a particular task



- It can be used multiple times
- Increase code reusability

Functions in C

- All the programs that we have studied to date contain only one **function**
  - The **main** function

- For more complex programs, it may be useful to write other functions

- A **function** can be viewed as a “sub-program” which
  - Does a specific task
  - May use input values (variables)
  - May produce a SINGLE output value
  - May use its own **local** variables internally within the function

- Improve code organization
- Enhance code readability

Function types

- Library functions
  - Inbuilt in C – scanf(), printf()

- User-defined functions
  - Declared and defined by coder/programmer

Writing functions in C

- There are three different aspects to writing and using a C function
  - Defining the function (**prototype** – also known as function header)
  - Implementing the function (**Implementation** – function body)
  - Using the function (**Function calling**)

```c
#include <stdio.h>

int add(int a, int b); // Function prototype

int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3); // Calling the 'add' function
    printf("Result: %d\n", result);
    return 0;
}
```

Function Prototype

- The function prototype is a single line summary of the function
  - Tells the number of inputs and output
  - Tells the type of inputs and output (int, float, double, char).
- It is normally located at the top of your program listing
  - Usually above the main function and below the #include files definition
- A function prototype has the form:

```
output Parameter functionName(input parameter, input parameter, …);
```

Example:

```
int add(int a, int b);
```

Function Return Type

- A function can only return either a single variable OR nothing at all
  - Multiple values are return using pointers and can be specify in the input parameters.

- The return type indicates the type of variable (if any) that the function outputs (returns)

- This will be one of the variable type keywords (e.g. int, float, char) OR the word **void** (indicating the function does not return anything)

Example:

```
return a + b;
```

Passing arguments

- Functions can take value (parameters) and give some value (return value)

```
void printHello();

void printTable(int x);

int add(int a, int b);

double factorial(int value);
```

Function Name

- A function name can be anything but it is a good idea that the name used tells you what the function does
  - For example, if the function is used for deleting any student data (use name deleteStudentData)

- We can declare more than one functions with the same name:
  - But there input or output parameters should be different.
  - For example
    - void deleteStudentData (void); // delete all the students data
    - void deleteStudentData (int studentID); // delete the student data with specific ID.

Function Inputs

- Most functions will use input values
  - It is possible to write a function that has no inputs (e.g. rand function)
  - Example void deleteStudentData (void);
- Inputs must be listed (separated by commas if there is more than one input value) within the brackets () after the function name
  - Example i. void deleteStudentData (int studentID)
    ii. int add(int a, int b)
- Each entry in this list of arguments must identify the input variable type and define a name by which that input can be referred to
- When there are no inputs to a function the keyword **void** appears instead.

Example

- Let's write a function to calculate the factorial of an integer
  - Input: A **single integer** (let's refer to it by the name **value**)
  - Function Name: **factorial**
  - Return value: A **double** (precision real number)

- Why are we returning a double?

- The prototype of this function is:

```
double factorial(int value);
```

Function Implementation

- ALL functions are typically located either before the **main** function OR after the **main** function
- A **function implementation** is very similar to the main function itself
  - Starts with the function description (effectively the same as the prototype)
    - Without the ';'
  - Bounded by a pair of curly brackets { }
  - (**Local**) variables can be defined within (at the start of) the function
  - If the function has an output, a new C command (**return**) must be present in the function

Function Implementation

- The C lines in the function can refer to the variables that are either
  i. defined **locally** in that function or
  ii. reference the “input” variable names
  iii. ? (defined **globally**)
- You cannot refer to a variable that is defined in any other function directly
- The return command is used as:

  return(variableName);
- This return command can:
  - Appear only in a function if it is defined as having an output value in its prototype.
  - Causes the function to end and output the value in the single returned variable.
  - Appear (if required) several times in a function

Example

```
unsigned long factorial(int value)
{
    long result = 1.0;
    int count;

    for (count = 2; count <= value; count++)
    {
        result = result*count;
    }
    return result;
}
```

Calling the Function

- If the function returns a value, we can store the result in a suitable variable type using the assignment (=) operator
- Examples of function calls would be:

```
variableName = functionOne();
fact = factorial(num);
result = add(5, 3);
```

- A function can be called from the main function or any other function
- When a function is completed the program returns to the next line after where the function was called from

```c
#include <stdio.h>

unsigned long factorial(int value);

// Function to calculate the factorial of a positive integer
unsigned long factorial(int value)
{
    long result = 1.0;
    int count;

    for (count = 2; count <= value; count++)
    {
        result = result*count;
    }
    return result;
}


int main() {
    int num;
    unsigned long fact;
    // Prompt the user to enter a positive integer
    printf("Enter a positive integer: ");
    scanf("%d", &num);

    fact = factorial(num);
    printf("Factorial of %d is %lu\n", num, fact);

    fact = factorial(10);
    printf("Factorial of %d is %lu\n", 10, fact);

    return 0;
}
```

Key properties

- C program execution always starts from the main function
- A function gets called directly or indirectly from the main function
- There can be multiple functions in a program
- Changes to parameters in function don’t change the values in calling functions

Function – use library functions

Exercise

Write a C program that uses library functions to calculate the square of a number given by user.

```c
#include<stdio.h>
#include<math.h>

int main(){
    int n = 4;

    printf("%f", pow(2, n));
    return 0;
}
```

Read about math.h – HERE and in the recommended text book

Function – practice exercise

Exercise

Write a simple calculator program that performs basic arithmetic operations (addition, subtraction, multiplication, and division) using custom functions.

Recursion in C

Recursion

- When a function calls itself to solve a smaller instance of the same problem.
- Breaks down a complex problem into simpler subproblems until they can be solved directly.
- Simplifies the implementation of certain algorithms and aids in solving complex problems efficiently.
- Depends on the concept of a base case, which is the simplest form of the problem that doesn't need further recursion.

Recursion – factorial example

- Loop and function

```c
unsigned long factorial(int value)
{
    long result = 1.0;
    int count;

    for (count = 2; count <= value; count++)
    {
        result = result*count;
    }
    return result;
}
```

- Recursion

```c
// Recursive function to calculate factorial
unsigned long factorial(int n) {
    if (n == 0 ||  n == 1) {
        return 1; // Base case: factorial of 0 and 1 is 1
    } else {
        return n * factorial(n - 1); // Recursive case: n! = n * (n-1)!
    }
}
```

Recursion function - basics

- Recursive functions must have a termination condition to prevent infinite recursion.
- The base case represents the simplest form of the problem that doesn't require further recursion.
  - Base case is the condition which stops recursion.
- The recursive case involves breaking down the problem into smaller subproblems and calling the function recursively.
- Anything that can be done with iteration, can be done with recursion and vice versa
- Recursion can sometimes give the most simple solution.

Recursion – practice exercise

- Write a C program using recursion function that take a natural number *n* as input, and calculate the sum of all natural numbers upto *n*.

Hint: *n* → *1 + 2 + 3 + … (n-1) + n*

Function – practice exercise (variable scope)

Exercise

Write a function in C that calculate adds UK VAT to a given amount.

Function – practice exercise (variable scope)

```c
/* write a function in C that calculate adds UK VAT to a given amount.
*/

#include <stdio.h>

float addVAT(float amount) {
    float VAT = 0.2;
    float total = amount + (amount * VAT);
    return total;
}

int main() {
    float amount = 100;

    printf("The total amount is: %.2f\n", addVAT(amount));
    printf("The total amount without VAT is: %.2f\n", amount);
    return 0;
}
```

```
The total amount is: 120.00
The total amount without VAT is: 100.00
```

Variable Scope

Variable Scope

- **Scope** refers to the region of a program where a variable or identifier is valid and can be used.
- It defines where in the code a variable can be accessed or manipulated.
- Primary types of scope:
  1. **Local scope** - Variables declared within a function or a block of code.
  2. **Global scope** - Variables declared outside of any function or block.
  3. **File scope** - Variables declared as static at the global level.
  4. **Function Parameters –** Variables declared in the parameter list of a function

Local scope

- Variables declared within a function or a block of code have local scope.
- They are accessible only within that specific function or block.
- Once the function or block's execution is complete, the local variables cease to exist.
- This scoping is often referred to as function scope or block scope.

```
void myFunction() {
    int x = 8; // x has local scope within myFunction
}
```

Global Scope

- Variables declared outside of any function or block have global scope.
- They are accessible from any part of the program, including functions and blocks.
- Global variables persist throughout the program's execution and are generally defined at the top of the program.

```
int globalVar = 15; // globalVar has global scope

void someFunction() {
    // globalVar can be accessed here
}
```

File Scope (Static Variables)

- **static** keyword has two meanings, depending on where the static variable is declared
  - Outside a function, static variables/functions only visible within that file, not globally
  - Inside a function, static variables:
    - are still local to that function
    - are initialized only during program initialization
    - do not get reinitialized with each function call

```
static int somePersistentVar = 0;
```

- **static** variables are allocated memory when the program starts and retain their values between function calls and across different scopes within a single source file.

Static variables – example

```c
#include <stdio.h>

void increment() {
    static int counter = 0; // Static variable with local scope
    counter++;
    printf("Counter: %d\n", counter);
}

int main() {
    increment(); // Counter: 1
    increment(); // Counter: 2
    increment(); // Counter: 3
    return 0;
}
```

- Output

```
Counter: 1
Counter: 2
Counter: 3
```

Function Parameters

- Parameters declared in the parameter list of a function definition also have local scope within that function.
- They act like local variables but are initialized with the values passed as arguments when the function is called.

```
void someFunction(int param1, float param2) {
    // param1 and param2 have local scope within someFunction
}
```

Benefits of Understanding Scope

- Avoiding naming conflicts and ambiguity in variable usage.
- Organizing code effectively by limiting variable visibility.
- Efficient memory management by controlling variable lifetimes.

Scope – practice exercise

Exercise

- Write a C program to demonstrate the scope of variables. Define a global variable ‘globalvar’ and a local variable ‘localvar’ in the main function.
- Create a user-defined function ‘update_variables’ that attempts to access, modify both the global and local variables, and print them.
- In the main function, print the values of both variables before and after calling the ‘update_variables’ function to observe how the scope affects the accessibility and modification of these variables.