PROGRAMMING IN C: INTRODUCTION

COMP1206 - PROGRAMMING II

Enrico Marchioni
e.marchioni@soton.ac.uk
Building 32 - Room 4019

This is a (very!) brief introduction to some of the basic features of C.

You will find many similarities with Java.

We will start with an example and analyze it to introduce:

- The main structure of a C program;
- Data types;
- Input and output;
- Conditional statements;
- Loops.

To conclude, we will look at another simple program and learn how arrays work.

In the next few days we will talk more extensively about:

- Pointers and memory allocation;
- Functions;
- Structures.

Most of the material you find in these (and the other) slides is taken from (or inspired by):

- Stephen G. Kochan. *Programming in C*. 4th Edition, Developer's Library, Addison Wesley, 2015.
- Dan Gookin. *C. All-in-One Desk Reference for Dummies*. Wiley, 2004.

Those are good basic references, but I encourage you to look for more material/tutorials online!

INTRODUCTION

- Where does the C language come from?
- At the end of the ’60s, researcher at Bells Labs (AT&T) developed a new programming language called B and a new operating system called UNIX.
- UNIX was originally written in assembly code and also had an interpreter for B.
- B did not have data types, nor structures.
- In 1971-1973, Dennis Ritchie developed C by expanding and modifying B with the addition of data types, structures, etc.
- Eventually, UNIX was rewritten in C.
- In 1978, “ The C Programming Language” by B. Kernigham and D. Richie became the standard reference for C.

- The growth in popularity of UNIX and the success of the IBM PC made C more popular as well.
- Several companies started developing their own C compilers.
- In the early 80's, the American National Standards Institution (ANSI) standardized C (published in 1990) and later on the International Standard Organization (ISO) adopted the standard.
- The most recent standard is C11.
- C has influenced many other languages such as C#, C++, Java, JavaScript, Perl, PHP, Python, etc.

- Popular programs written in C:
  - Kernels: OSX, Linux, Windows, iOS, Android.
  - Databases: Oracle Database, MySQL, MS SQL Server, PostgreSQL.
  - Videogames: Doom, Quake, Quake II, Quake III, Return to Castle Wolfenstein.
  - Embedded systems.
- C is still used today and will be for quite some time. Why?
- Its portability: there exist C compilers for most existing architectures.
- Compilers, libraries and interpreters for other programming languages are often implemented in C (Python, Perl, and PHP).
- C allows for memory management and optimization.
- C is very efficient: it has small runtime and small memory footprint compared to several other languages.

- Before we start... how do we write a program in C and how do we compile it and run it?
- You can simply use a text editor, write your code and name the file anything you want with the extension "`.c`". For example:

  ```
  myfile.c
  ```

- To compile your file under UNIX, you can use a compiler like GNU C, and type

  ```
  gcc myfile.c
  ```

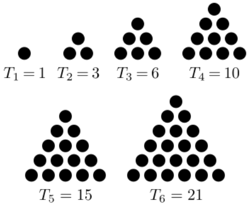
- If the file is compiled successfully, the output in UNIX is an executable named, by default,

  `a.out` (or `myfile.exe` under Windows)

- You can now run your program and see if it works as expected.
- Alternatively, you can write, compile and run C code with some popular IDEs such as Xcode, Microsoft Visual Studio, Netbeans, Code::Blocks, etc., depending on your OS and your preferences.

EXAMPLE

- We introduce C by looking at a simple program and analyzing some of its basic features.
- After so much Java, any basic C program should look very familiar to you.
- Let's examine a program that computes any triangular number up to 100.
- A triangular number counts objects arranged in an equilateral triangle.
- The *n*-th triangular number is the number of dots in the triangular arrangement with *n* dots on a side.
- The *n*-th triangular number is the sum of the natural numbers from 1 to *n*.



```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```
What triangular number do you want?
23
Triangular number 23 is 276
Program ended with exit code: 0
```

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```
What triangular number do you want?
111
111 is out of range
Program ended with exit code: 1
```

MAIN

```
#include <stdio.h>

int main (void)
{

    return 0;
}
```

- This is what the main body of a C program looks like.
- `stdio.h` is a header file where several important I/O routines are defined (i.e. `printf`, `scanf`, etc.)
- `#include <stdio.h>`

  tells the compiler to insert text from the header into the source code.
- `int main (void)`

  tells the system where to begin execution of the program and to return an integer as an output.
- `main` is a special kind of function, and, outside of the `main` routine, there can be several other functions (more on that later!).

```
#include <stdio.h>

int main (void)
{

    return 0;
}
```

- All program statements included between the braces { } are taken as part of the `main` routine by the system.
- All program statements in C must be terminated by a semicolon (;).

  ```
  return 0;
  triangularNumber = 0;
  scanf ("%i", &number);
  ```

- `return 0;`

  tells the system to finish the execution of `main` and return value `0`.
- `0` is used by convention to indicate that the program completed successfully.
- You can use other integer values with `return`. This is useful when you want the compiler to identify certain errors:

  ```
  return 1;
  ```

DATA TYPES

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```c
int n, number, triangularNumber;
```

- This statement declares the variables

  `n, number, triangularNumber`

  as integers.
- Variable declaration follows the format

  ```c
  type variablename;
  ```
- Variables must be declared before use.
- Variables must be initialised!

  ```c
  triangularNumber = 0;
  ```
- Data types in C are very similar to Java.

Type	**Meaning**
`int`	Integer value; that is, a value that contains no decimal point; guaranteed to contain at least 16 bits of precision.
`short int`	Integer value of reduced precision; takes half as much memory as an `int` on some machines; guaranteed to contain at least 16 bits of precision.
`long int`	Integer value of extended precision; guaranteed to contain at least 32 bits of precision.
`long long int`	Integer value of extraextended precision; guaranteed to contain at least 64 bits of precision.
`unsigned int`	Positive integer value; can store positive values up to twice as large as an `int`; guaranteed to contain at least 16 bits of precision.
`float`	Floating-point value; that is, a value that can contain decimal places; guaranteed to contain at least six digits of precision.
`double`	Extended accuracy floating-point value; guaranteed to contain at least 10 digits of precision.
`long double`	Extraextended accuracy floating-point value; guaranteed to contain at least 10 digits of precision.

`char`	Single character value; on some systems, sign extension might occur when used in an expression.
`unsigned char`	Same as `char`, except ensures that sign extension does not occur as a result of integral promotion.
`signed char`	Same as `char`, except ensures that sign extension does occur as a result of integral promotion.
`_Bool`	Boolean type; large enough to store the values `0` or `1`.
`float _Complex`	Complex number.
`double _Complex`	Extended accuracy complex number.
`long double _Complex`	Extraextended accuracy complex number.
`void`	No type; used to ensure that a function that does not return a value is not used as if it does return one, or to explicitly "discard" the results of an expression. Also used as a generic pointer type (`void *`).

```
Data Type                   Size in bits:

Int                              32 bits.
Short int                        16 bits.
Long int                         64 bits.
Long long int                    64 bits.
Unsigned long int                64 bits.
Float                            32 bits.
Double                           64 bits.
Long double                     128 bits.
Char                              8 bits.
Unsigned char                     8 bits.
Signed char                       8 bits.
_Bool                             8 bits.
Float _Complex                   64 bits.
Double _Complex                 128 bits.
Long Double _Complex            256 bits.
```

printf AND scanf

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```
printf("What triangular number do you want?\n");

printf("Triangular number %i is %i\n",number,triangularNumber);
```

- The `printf` routine is a function that displays its arguments.
- It has the form

  ```
  printf ("Text ... %i ... %i ... \n", value1, value2);
  ```

- The first argument is always the character string to be displayed.
- `%i` is a placeholder.
- `%i` tells the routine to display the corresponding argument, which has to be of the type specified by `i`.
- Arguments in `printf` must be in the order they need to appear.

  ```
  printf("Triangular number %i is %i\textbackslash n", number, triangularNumber)
  ```

```
scanf("%i", &number);
```

- scanf is a routine that enables you to type values into the program.
- It has the form

  ```
  scanf("%i", &value);
  ```

- The first argument is the format string that tells the system what types of values are to be read in from the terminal.
- %i specifies the type of value.
- The second argument &value specifies where the value entered by the user is to be stored. This is related to pointers (more on that later!)

Type	printf chars
`char`	`%c`
`_Bool`	`%i`, `%u`
`short int`	`%hi`, `%hx`, `%ho`
`unsigned short int`	`%hu`, `%hx`, `%ho`
`int`	`%i`, `%x`, `%o`
`unsigned int`	`%u`, `%x`, `%o`
`long int`	`%li`, `%lx`, `%lo`
`unsigned long int`	`%lu`, `%lx`, `%lo`
`long long int`	`%lli`, `%llx`, `&llo`
`unsigned long long int`	`%llu`, `%llx`, `%llo`
`float`	`%f`, `%e`, `%g, %a`
`double`	`%f`, `%e`, `%g, %a`
`long double`	`%Lf`, `$Le`, `%Lg`

IF...ELSE...

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```
if (number <= 100)
{

}
else
{

}
```

- Conditional statements have the form

```
    if ( expression )
        program statement 1
    else
        program statement 2
```

- The system check if `expression` is true. If it is, then `program statement 1` is executed.
- If `expression` is not true, `program statement 2` is executed.

```
if (number <= 100)
{
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
                triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,triangularNumber);
}
else
{
        printf("%i is out of range\n", number);
        return 1;
}
```

- If the expression

  `number <= 100`

  is true, the program initializes `triangularNumber` and executes the loop.
- If the expression is not true, it prints that the entered number is out of range and ends its run with

  `return 1;`

- Notice that here we use `1` to highlight this outcome.

- What happens if we write something like this?

```
if (number = 100)

else
```

- The program compiles without any issues.
- This is the output running the program on input 23:

```
What triangular number do you want?
23
Triangular number 100 is 5050
Program ended with exit code: 0
```

- This is the output running the program on input 111:

```
What triangular number do you want?
111
Triangular number 100 is 5050
Program ended with exit code: 0
```

- This is the output running the program on input 100:

```
What triangular number do you want?
100
Triangular number 100 is 5050
Program ended with exit code: 0
```

- What happens if we write something like this?

```
if (number = 0)

else
```

- The program compiles without any issues.
- This is the output running the program on input 23:

```
What triangular number do you want?
23
0 is out of range
Program ended with exit code: 1
```

- This is the output running the program on input 111:

```
What triangular number do you want?
111
0 is out of range
Program ended with exit code: 1
```

- This is the output running the program on input 100:

```
What triangular number do you want?
100
0 is out of range
Program ended with exit code: 1
```

- If the conditions specified in the if statement are satisfied, then the next statements are executed.
- In C, satisfied means nonzero.
- If you write

  ```
  if (number = 100)
  ```

  `number = 100` just means something different from zero, so C sees the condition as true and whatever follows is executed.
- If you write

  ```
  if (number = 0)
  ```

  `number = 0` just means something equals zero, so C sees the condition as false and jumps to the statements following `else`.
- Even with these mistakes your C program will compile just fine, but, of course, it will not behave like you expected!

FOR LOOP

```c
#include <stdio.h>

int main (void)
{
    int n, number, triangularNumber;

    printf ("What triangular number do you want?\n");
    scanf ("%i", &number);

    if (number <= 100)
    {
        triangularNumber = 0;

        for(n=1; n<=number; ++n)
            triangularNumber += n;

        printf ("Triangular number %i is %i\n", number,
                triangularNumber);
    }
    else
    {
        printf("%i is out of range\n", number);
        return 1;
    }

    return 0;
}
```

```
for(n=1; n<=number; ++n)
        triangularNumber += n;
```

- The general format of the `for` statement is:

  ```
  for ( init_expression; loop_condition; loop_expression )
          program statement
  ```

- `init_expression` sets the initial values before the loop begins.
- `loop_condition` sets the conditions for the loop to continue.
- `loop_expression` contains an expression that changes the value of the index variable after the body of the loop is executed.

```
for(n=1; n<=number; ++n)
        triangularNumber += n;
```

1. User assigned `number` a value.
2. `triangularNumber` has value `0`.
3. `n` is initialised and assigned `1`
4. if (`n <= number`), `triangularNumber`:

   ```
   triangularNumber += n;
   ```

   which is the same as

   ```
   triangularNumber = triangular number + n;
   ```

5. The value of `n` is increased by 1 with `++n`
6. Loop starts again until...
7. ...loop ends and program prints:

   `Triangular number` **number** `is` **triangularNumber**

ARRAYS

```
// Generate Fibonacci numbers using variable length arrays

#include <stdio.h>

int main (void)
{
        int i, numFibs;

        printf ("How many Fibonacci numbers do you want (between 1 and 75)? \n");
        scanf ("%i", &numFibs);

        if (numFibs < 1 || numFibs > 75) {
        printf ("Bad number, sorry!\n");
        return 1;
        }

        unsigned long long int Fibonacci[numFibs];

        Fibonacci[0] = 0;       // by definition
        Fibonacci[1] = 1;       // ditto

        for(i=2; i<numFibs; ++i)
                Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];


        for(i=0; i<numFibs; ++i)
                printf ("%llu ", Fibonacci[i]);

        printf ("\n");

        return 0;
}
```

This is the output of running two instances of the above program:

```
How many Fibonacci numbers do you want (between 1 and 75)?
25
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
    6765 10946 17711 28657 46368
Program ended with exit code: 0
```

```
How many Fibonacci numbers do you want (between 1 and 75)?
83
Bad number, sorry!
Program ended with exit code: 1
```

- An array is a list of elements.
- Arrays are declared with the format

  ```
  int (float, ...) arrayname[10];
  ```

  where `10` is the number of elements in the array.
- To initialize an array, you can initialize individual elements

  ```
  int array[5];

  array[0] = 2;
  array[2] = 15;
  ```

- Alternatively you can initialize all elements at once:

  ```
  int array[5] = { 2, 8, 41, 15, 7 };
  ```

- In an n-element array, first element is indexed by zero, last element indexed by $n-1$. References are made by using subscripts from 0 to $n-1$.

- You can initialize some elements

  ```
  int array[5] = { 2, 8, 41 };
  ```

  (only he first 3 elements are specified)
- or you can initialize specific elements

  ```
  int array[5] = { [3] = 15, [1] = 8, [0] = 2 };
  ```

- Values of uninitialized elements are not defined!
- C allows you to define an array without specifying the number of elements:

  ```
  char word[ ] = { "H", 'e' , 'l', 'l', 'o', '!' };
  ```

- The size of the array is determined automatically based on the number of initialization elements.
- This requires to initialize every element in the array when it is defined. If this is not to be the case, you must explicitly dimension the array.

```c
#include <stdio.h>

int main()
{
    int x, array[5];

    for (x=0; x<5; x++)
    {
        array[x] = x;
    }

    printf("The value is %i\n", array[13]);

    return 0;
}
```

```
The value is 32767
Program ended with exit code: 0
```

- C does not check array bounds!
- The above program prints the value of `array[13]`, which is out of the bounds of the array we declared.
- The program complies just fine though, and we get whatever value was at the memory location of `array[13]`.

- Multidimensional arrays are declared similar to linear arrays:

```
int M[4][5];
```

- 4 refers to the number of rows, 5 to the number of columns.
- Multidimensional arrays can be initialized in different ways:

```
int M[4][5] = {
                              {10, 5,-3,17,82},
                              { 9, 0, 0, 8,-7},
                              {32,20, 1, 0,14},
                              { 0, 0, 8, 7, 6}
                   };
```

or

```
int M[4][5]={10,5,-3,17,82,9,0,0,8,-7,32,
                   20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

- Similar to the linear case, you can initialize just specific elements:

```
int matrix[4][3] = { [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

- It is possible to declare arrays of variables size.

  ```c
  int array[n];
  ```

  where *n* is not a set number.

- We define a program to generate Fibonacci numbers, i.e. numbers from the sequence

  $$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

  where every number after the first two is the sum of the two preceding ones.

- The program will ask how many Fibonacci numbers you want to display between 1 and 75.

- It will compute and print all those numbers by using arrays of variable size

```c
// Generate Fibonacci numbers using variable length arrays

#include <stdio.h>

int main (void)
{
        int i, numFibs;

        printf ("How many Fibonacci numbers do you want (between 1 and 75)? ");
        scanf ("%i", &numFibs);

        if (numFibs < 1 || numFibs > 75) {
        printf ("Bad number, sorry!\n");
        return 1;
        }

        unsigned long long int Fibonacci[numFibs];

        Fibonacci[0] = 0;       // by definition
        Fibonacci[1] = 1;       // ditto

        for(i=2; i<numFibs; ++i)
                Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];


        for(i=0; i<numFibs; ++i)
                printf ("%llu ", Fibonacci[i]);

        printf ("\n");

        return 0;
}
```