Linux System Programming: Introduction

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

System programming...

This lectures will focus on system programming, which is the practice of writing system software.

System programming...

This lectures will focus on system programming, which is the practice of writing system software.

System software lives at a low level, interfacing directly with the kernel and core system libraries.

System programming...

This lectures will focus on system programming, which is the practice of writing system software.

System software lives at a low level, interfacing directly with the kernel and core system libraries.

Examples of system software are...

... your text editor, your compiler and your debugger, your core utilities and system daemons are all system software;

... but also the network server, the web server, and the database.

System programming...

This lectures will focus on system programming, which is the practice of writing system software.

System software lives at a low level, interfacing directly with the kernel and core system libraries.

Examples of system software are...

... your text editor, your compiler and your debugger, your core utilities and system daemons are all system software;

... but also the network server, the web server, and the database.

All these components primarily, if not exclusively, interact with the kernel and the C library.

System programming...

Traditionally, all Unix programming was system-level programming.

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

We can compare and contrast system programming with application programming

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

We can compare and contrast system programming with application programming

System programming: programmer must have an acute awareness of the hardware and the operating system on which they work.

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

We can compare and contrast system programming with application programming

System programming: programmer must have an acute awareness of the hardware and the operating system on which they work.

Application programming: programmer also use high-level libraries that abstract away the details of the hardware and operating system.

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

We can compare and contrast system programming with application programming

System programming: programmer must have an acute awareness of the hardware and the operating system on which they work.

Efficiency!

Application programming: programmer also use high-level libraries that abstract away the details of the hardware and operating system.

System programming...

Traditionally, all Unix programming was system-level programming.

Unix systems historically did not include many higher-level abstractions

We can compare and contrast system programming with application programming

System programming: programmer must have an acute awareness of the hardware and the operating system on which they work.

Efficiency!

Application programming: programmer also use high-level libraries that abstract away the details of the hardware and operating system.

Portability!

In this lecture we will focus on Linux System Programming!

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

System calls (often shortened to *syscalls*) are function invocations made from user space into the kernel (the core internals of the system) in order to request some service or resource from the operating system.
Examples of system calls are:

- read()
- write()
- get_thread_area()
- set_tid_address()

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*.

In addition to standard C library, the *GNU C library* provides wrappers for *system calls*, *threading support*, and *basic application facilities*.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

The C library (*libc*) is at the heart of Unix applications.

On modern Linux systems, the C library is provided by *GNU libc*, abbreviated *glibc*.

In addition to standard C library, the *GNU C library* provides wrappers for *system calls*, *threading support*, and *basic application facilities*.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

In Linux, the standard C compiler is provided by the GNU Compiler Collection (*gcc*).

- *gcc* supports many languages (C, C++, Java, Ada...);
- *gcc* is also the name of the *C compiler*.

Cornerstones of Linux System Programming...

There are three cornerstones of system programming in Linux:

1. system calls,
2. the C library,
3. and the C compiler.

In Linux, the standard C compiler is provided by the GNU Compiler Collection (*gcc*).

- *gcc* supports many languages (C, C++, Java, Ada...);
- *gcc* is also the name of the *C compiler*.

Portability...

At the system level, there are two separate sets of definitions and descriptions that impact portability:

- Application Programming Interface (API)
- Application Binary Interface (ABI)

Portability: API vs ABI...

At the system level, there are two separate sets of definitions and descriptions that impact portability:

- Application Programming Interface (API)
- Application Binary Interface (ABI)

Portability: API vs ABI...

At the system level, there are two separate sets of definitions and descriptions that impact portability:

- Application Programming Interface (API)
- Application Binary Interface (ABI)

Application Programming Interface (API)

An API defines the interfaces by which one piece of software communicates with another at the source level

It provides abstraction by providing a standard set of interfaces —usually functions— that one piece of software can invoke from another piece of software.

API guarantees *source compatibility*: the user of the API will successfully compile against the implementation of the API.

Portability: API vs ABI...

At the system level, there are two separate sets of definitions and descriptions that impact portability:

- Application Programming Interface (API)
- Application Binary Interface (ABI)

Application Programming Interface (API)

An API defines the interfaces by which one piece of software communicates with another at the source level

It provides abstraction by providing a standard set of interfaces —usually functions— that one piece of software can invoke from another piece of software.

API guarantees *source compatibility*: the user of the API will successfully compile against the implementation of the API.

Portability: API vs ABI...

At the system level, there are two separate sets of definitions and descriptions that impact portability:

- Application Programming Interface (API)
- Application Binary Interface (ABI)

Application Binary Interface (ABI)

ABI defines the binary interface between two or more pieces of software on a particular architecture

It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries.

ABI ensures *binary compatibility*: a piece of object code will function on any system with the same ABI, without requiring recompilation.

C Language Standards...

K&R C Dennis Ritchie and Brian Kernighan's, *The C Programming Language* (Prentice Hall), acted as the informal *C* specification;

ANSI C *American National Standards Institute* (ANSI) developed an official version of C (1989);

ISO CXX *International Organization for Standardization* (ISO) updates the *C* standards:

- *ISO C90*, based on *ANSI C*;
- *ISO C95*, an updated of *ISO C90*;
- *ISO C99*, introduced many new features, including inline functions, new data types, variable-length arrays...;
- *ISO C11*, latest version of the standard equipped with a formalized memory model, enabling the portable use of threads across platforms.

Language C: Introduction

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

C keywords...

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
inline	int	long	register
restrict	return	short	signed
sizeof	static	struct	switch
typedef	union	unsigned	void
volatile	while		

C keywords...

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
inline	int	long	register
restrict	return	short	signed
sizeof	static	struct	switch
typedef	union	unsigned	void
volatile	while		

You have already seen many of these keywords in Java...

C keywords...

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
inline	int	long	register
restrict	return	short	signed
sizeof	static	struct	switch
typedef	union	unsigned	void
volatile	while		

You have already seen many of these keywords in Java...

...the meaning is almost the same in C!

Your first *C* program...

```c
/**
This is a comment that, like in Java...
...goes on multiple lines!
**/

//Statement to include a set of definitions!
#include <stdio.h>

#define MESSAGE "Hello World!\n"

int main(void) {
   printf(MESSAGE);
}
```

Compiling your program...

Compiling a C program is a multi-stage process. At an overview level, the process can be split into four separate stages:

1. preprocessing;
2. compilation;
3. assembly;
4. linking.

Compiling your program: Preprocessing...

The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the preprocessor as preprocessor commands.

Compiling your program: Preprocessing...

The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the preprocessor as preprocessor commands.

To print the result of the preprocessing stage, pass the –E option to *gcc*:

```
gcc -E hello_world.c
```

Compiling your program: Compilation...

In this stage, the preprocessed code is translated to assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

Compiling your program: Compilation...

In this stage, the preprocessed code is translated to assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

To save the result of the compilation stage, pass the –S option to *gcc*:

```
gcc -S hello_world.c
```

Compiling your program: Assembly...

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or object code. The output consists of actual instructions to be run by the target processor.

Compiling your program: Assembly...

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or object code. The output consists of actual instructions to be run by the target processor.

To save the result of the compilation stage, pass the `-c` option to *gcc*:

```
gcc -c hello_world.c
```

Compiling your program: Assembly...

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or object code. The output consists of actual instructions to be run by the target processor.

To save the result of the compilation stage, pass the `-c` option to *gcc*:

```
gcc -c hello_world.c
```

Running the above command will create a file named `hello_world.o`, containing the object code of the program. The contents of this file is in a binary format and can be inspected using `hexdump` or `od`:

```
hexdump hello_world.o
od -c hello_world.o
```

Compiling your program: Linking...

The object code generated in the assembly stage is composed of machine instructions that the processor understands.

Compiling your program: Linking...

The object code generated in the assembly stage is composed of machine instructions that the processor understands.

However, some pieces of the program are out of order or missing.

Compiling your program: Linking...

The object code generated in the assembly stage is composed of machine instructions that the processor understands.

However, some pieces of the program are out of order or missing.

The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions used by the program.

Compiling your program: Linking...

The object code generated in the assembly stage is composed of machine instructions that the processor understands.

However, some pieces of the program are out of order or missing.

The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions used by the program.

The result of this stage is the final executable program:

```
gcc -o hello_world hello_world.c
```

Without the `-o hello_world` option, file `a.out` is generated.

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

1. can only have alphanumeric characters (a−z , A−Z , 0−9) and underscore (_);

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

1. can only have alphanumeric characters (a−z , A−Z , 0−9) and underscore (_);
2. the first character of an identifier can only contain alphabet (a−z , A−Z) or underscore (_);

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

1. can only have alphanumeric characters (a−z , A−Z , 0−9) and underscore (_);
2. the first character of an identifier can only contain alphabet (a−z , A−Z) or underscore (_);
3. are case sensitive...

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

1. can only have alphanumeric characters (a−z , A−Z , 0−9) and underscore (_);
2. the first character of an identifier can only contain alphabet (a−z , A−Z) or underscore (_);
3. are case sensitive...

   ... for example name and Name are two different identifiers in *C*;

Identifiers...

In *C* language identifiers are the names given to variables, constants, functions and user-define data.

These identifier are defined against a set of rules:

1. can only have alphanumeric characters (a−z , A−Z , 0−9) and underscore (_);
2. the first character of an identifier can only contain alphabet (a−z , A−Z) or underscore (_);
3. are case sensitive...
   ... for example name and Name are two different identifiers in *C*;
4. keywords are not allowed to be used as Identifiers.

Types...

C Data Types	
void	comprises an empty set of values; it is an incomplete object type that cannot be completed.
char	usually 8-bits (1 byte)
short int	at least 16-bits
int	usually the natural *word* size for a machine or OS (e.g., 16, 32, 64 bits)
long int	at least 32-bits
float	usually 32-bits
double	usually 64-bits
long double	usually at least 64-bits

Types: Ranges...

```c
#include <stdio.h>
#include <limits.h> /* integer specifications */
#include <float.h> /* floating-point specifications */

/* Look at range limits of certain types */
int main (void)
{
    printf("Integer range:\t%d\t%d\n", INT_MIN, INT_MAX);
    printf("Long range:\t%ld\t%ld\n", LONG_MIN, LONG_MAX);
    printf("Float range:\t%e\t%e\n", FLT_MIN, FLT_MAX);
    printf("Double range:\t%e\t%e\n", DBL_MIN, DBL_MAX);
    printf("Long double range:\t%e\t%e\n", LDBL_MIN, LDBL_MAX
    );
    printf("Float-Double epsilon:\t%e\t%e\n", FLT_EPSILON,
    DBL_EPSILON);
}
```

Types: size in bytes...

The instruction sizeof can be used to obtain the *size*, in byte, of a *C* type.

Types: size in bytes...

The instruction sizeof can be used to obtain the *size*, in byte, of a *C* type.

```
##include <stdio.h>

int main (void)
 /* Print the size of various types in "number-of-chars" */
{
   printf ("void\tchar\tshort\tint\tlong\tfloat\tdouble\n");
   printf ("%3d\t%3d\t%3d\t%3d\t%3d\t%3d\t%3d\n",\
            sizeof(void), sizeof(char), sizeof(short),\
            sizeof(int),sizeof(long), sizeof(float),\
            sizeof(double));
}
```

Types...

Type modifiers...

- signed and unsigned, can be applied to integer types (char, int, long)...

Types...

Type modifiers...

- signed and unsigned, can be applied to integer types (char, int, long)...
  ... an unsigned type is always non-negative;

Types...

Type modifiers...

- signed and unsigned, can be applied to integer types (char, int, long)...
  - ... an unsigned type is always non-negative;
  - ... integer types are signed by default.

Types...

Type modifiers...

- signed and unsigned, can be applied to integer types (char, int, long)...
  - ... an unsigned type is always non-negative;
  - ... integer types are signed by default.
- const, identifies a *variable* that cannot be changed

```
const int DoesNotChange = 5;
DoesNotChange = 6; /* Error: will not compile */
```

Types...

Type modifiers...

- signed and unsigned, can be applied to integer types (char, int, long)...
  - ... an unsigned type is always non-negative;
  - ... integer types are signed by default.
- const, identifies a *variable* that cannot be changed

```
const int DoesNotChange = 5;
DoesNotChange = 6; /* Error: will not compile */
```

- volatile , refers to variables whose value may change in a manner beyond the normal control of the program (in multithreading).

Literals. . .

Integers can be a decimal, octal, or hexadecimal constant

- prefix `0x` (or `0X`) indicates hexadecimal;
- prefix `0` indicates octal;
- nothing for decima

Literals...

Integers can be a decimal, octal, or hexadecimal constant

- prefix `0x` (or `0X`) indicates hexadecimal;
- prefix `0` indicates octal;
- nothing for decima

A suffix that is a combination of `u` (or `U`) and `l` (or `L`), for unsigned and long, respectively.

Literals...

Integers can be a decimal, octal, or hexadecimal constant

- prefix `0x` (or `0X`) indicates hexadecimal;
- prefix `0` indicates octal;
- nothing for decima

A suffix that is a combination of `u` (or `U`) and `l` (or `L`), for unsigned and long, respectively.

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Decimal form: the decimal point, the exponent, or both must be included;

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Decimal form: the decimal point, the exponent, or both must be included;

Exponential form: integer part, the fractional part, or both must be include, the signed exponent is introduced by `e` or `E`.

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Decimal form: the decimal point, the exponent, or both must be included;

Exponential form: integer part, the fractional part, or both must be include, the signed exponent is introduced by `e` or `E`.

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Decimal form: the decimal point, the exponent, or both must be included;

Exponential form: integer part, the fractional part, or both must be include, the signed exponent is introduced by `e` or `E`.

Suffix `f` is used for literal of type float , while `d` is used for type double.

Literals...

A floating-point literal can be represented either in decimal form or exponential form. It has:

- an integer part;
- a decimal point;
- a fractional part;
- and an exponent part.

Decimal form: the decimal point, the exponent, or both must be included;

Exponential form: integer part, the fractional part, or both must be include, the signed exponent is introduced by `e` or `E`.

Suffix `f` is used for literal of type float, while `d` is used for type double.

```
3.14159          /* Legal */
314159E-5L       /* Legal */
510E             /* Illegal: incomplete exponent */
210f             /* Illegal: no decimal or exponent */
.e55             /* Illegal: missing integer or fraction
```

Symbolic Constants...

Symbolic constants represent constant values, from the set of constant types, by a symbolic name:

```
#define BLOCK_SIZE 100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define HELLO "Hello World\n"
#define EXP 2.7183
```

Symbolic Constants...

Symbolic constants represent constant values, from the set of constant types, by a symbolic name:

```
#define BLOCK_SIZE 100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define HELLO "Hello World\n"
#define EXP 2.7183
```

NB: #define, like #include, is a preprocessor command!

Symbolic Constants...

Symbolic constants represent constant values, from the set of constant types, by a symbolic name:

```
#define BLOCK_SIZE 100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define HELLO "Hello World\n"
#define EXP 2.7183
```

NB: #define, like #include, is a preprocessor command!

Another form of symbolic constant is an enumeration, which is a list of constant integer values:

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

Symbolic Constants...

Symbolic constants represent constant values, from the set of constant types, by a symbolic name:

```
#define BLOCK_SIZE 100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define HELLO "Hello World\n"
#define EXP 2.7183
```

NB: #define, like #include, is a preprocessor command!

Another form of symbolic constant is an enumeration, which is a list of constant integer values:

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

Symbolic constants and enumerations are by convention given uppercase names.

Declarations. . .

All variables must be declared before they are used.

Declarations...

All variables must be declared before they are used.

The general form of a declaration6 is

```
<qualifier> <type> <identifier1> = <value1>, <identifier2> =
    <value2>, ... ;
```

where the assignment to an initial value is optional.

Declarations...

All variables must be declared before they are used.

The general form of a declaration6 is

```
<qualifier> <type> <identifier1> = <value1>, <identifier2> =
    <value2>, ... ;
```

where the assignment to an initial value is optional.

```
int lower, upper, step; /* 3 uninitialised ints */
char tab = '\t'; /* a char initialised with '\t' */
char buf[10]; /* an uninitialised array of chars */
int m = 2+3+4; /* constant expression: 9 */
int n = m + 5; /* initialised with 9+5 = 14 */
float limit = 9.34f;
const double PI = 3.1416;
```

Arithmetic operators...

Operator name	Syntax
Basic assignment	a = b
Addition	a + b
Subtraction	a − b
Unary plus (integer promotion)	+a
Unary minus (additive inverse)	−a
Multiplication	a * b
Division	a / b
Modulo (integer remainder)	a % b
Increment Prefix	++a
Increment Postfix	a++
Decrement Postfix	a++
Decrement Postfix	a−−

Relational and logical operators...

Operator name	Syntax
Equal to	a == b
Not equal to	a != b
Greater than	a > b
Less than	a < b
Greater than or equal to	a >= b
Less than or equal to	a <= b

Operator name	Syntax
Logical negation (NOT)	!a
Logical AND	a && b
Logical OR	a || b

Bitwise operators...

Operator name	Syntax
Bitwise NOT	~a
Bitwise AND	a & b
Bitwise OR	a | b
Bitwise XOR	a ^ b
Bitwise left shift	a << b
Bitwise right shift	a >> b

Compound assignment operators...

Operator name	Syntax	Meaning
Addition assignment	a += b	a = a + b
Subtraction assignment	a -= b	a = a - b
Multiplication assignment	a *= b	a = a * b
Division assignment	a /= b	a = a / b
Modulo assignment	a %= b	a = a % b
Bitwise AND assignment	a &= b	a and_eq b
Bitwise OR assignment	a |= b	a or_eq b
Bitwise XOR assignment	a ^= b	a xor_eq b
Bitwise left shift assignment	a <<= b	a = a << b
Bitwise right shift assignment	a >>= b	a = a >> b

Branching and iterations...

We use statement to indicate a single *C* instruction or a block (a sequence of statements grouped by { and }).

- If-Then-Else:

```
if (expression) statement
else statement
```

Branching and iterations...

We use statement to indicate a single *C* instruction or a block (a sequence of statements grouped by { and }).

- If-Then-Else:

```
if (expression) statement
else statement
```

- Conditional expression: (expression1 ? expresion2 : expression3)

Branching and iterations...

We use statement to indicate a single *C* instruction or a block (a sequence of statements grouped by { and }).

- If-Then-Else:

```
if (expression) statement
else statement
```

- Conditional expression: (expression1 ? expresion2 : expression3)
- Switch:

```
switch (expression) {
  case cont-int-expr: statment
  case cont-int-expr: statment
  ...
  default: statements
}
```

Branching and iterations...

- While Loops:

```
while (expression)
  statement

do
  statement
while (expression);
```

Branching and iterations...

- While Loops:

```
while (expression)
  statement

do
  statement
while (expression);
```

- For Loops:

```
for( expr1 ; expr2 ; expr3 )
  statement
```

Functions...

In *C* functions represent the basic building blocks of computations.

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;
3. types of parameters.

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;
3. types of parameters.

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;
3. types of parameters.

Function Definitions provide the actual body of the declared functions.

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;
3. types of parameters.

Function Definitions provide the actual body of the declared functions.

Each function must be declared before its definition!

Functions...

In *C* functions represent the basic building blocks of computations.

Function Declarations provide info (used by the compiler) about the prototype of a function:

1. function's name;
2. return type;
3. types of parameters.

Function Definitions provide the actual body of the declared functions.

Each function must be declared before its definition!

Typically declarations are placed in `myfile.h` file, this is included in the corresponding `myfile.c` where function definitions are placed.

Functions: An example...

```c
#include <stdio.h>

int factorial( int );

int factorial( int n ) {
  if (n>0) {
    return n*factorial(n-1);
  } else {
    return 1;
  }
}

int main(void) {
  printf("!%d=%d\n",5,factorial(5));
}
```

Handling errors...

To handle errors *C* standard library provides the macro `assert`.

This macro can be used to che if a give condition is satisfied or not.

Handling errors...

To handle errors *C* standard library provides the macro assert.

This macro can be used to che if a give condition is satisfied or not.

```
int factorial( int n ) {
  assert(n>=0); //<-- NEW!
  if (n>0) {
    return n*factorial(n-1);
  } else {
    return 1;
  }
}
```

Handling errors...

To handle errors *C* standard library provides the macro `assert`. This macro can be used to che if a give condition is satisfied or not.

Handling errors...

To handle errors *C* standard library provides the macro `assert`. This macro can be used to che if a give condition is satisfied or not.

```
int factorial( int n ) {
  assert(n>=0); //<-- NEW!
  if (n>0) {
    return n*factorial(n-1);
  } else {
    return 1;
  }
}
```

Handling errors...

To handle errors *C* standard library provides the macro assert. This macro can be used to che if a give condition is satisfied or not.

```
int factorial( int n ) {
  assert(n>=0); //<-- NEW!
  if (n>0) {
    return n*factorial(n-1);
  } else {
    return 1;
  }
}
```

To use the macro, #include <assert.h> must be used at the beginning of your program.

C Standard Library...

The standard library has a large number of functions (about 145) which provide many commonly- used routines and operations.

- Mathematical functions: sqrt, pow, sin, cos, tan.
- Manipulating characters. isdigit, isalpha, isspace, toupper, tolower.
- Manipulating strings. strlen, strcpy, strcmp, strcat, strstr, strtok.
- Formatted input and output. printf, scanf, sprintf, sscanf.
- File input and output. fopen, fclose, fgets, getchar, fseek.
- Error handling. assert, exit.
- Time and date functions. clock, time, difftime.
- Sort and search. qsort, bsearch.
- Low-level memory operations. memcpy, memset.