Unit 4

Functions and Libraries

Alberto Casagrande
Email: acasagrande@units.it

a.a. 2019/2020

Computing the Factorial

```
...
/* Factorial of 10 */
int fact=1;
for (int i=1; i<=10; i=i+1)
  fact=i*fact;

printf("The factorial of 10 is %d\n", fact);

/* Factorial of 7 */
fact=1;
for (int i=1; i<=7; i=i+1)
  fact=i*fact;

printf("The factorial of 7 is %d\n", fact);
...
```

Issues in Previous Code

- Complexity in the code (How to use it for a different number?)
- Explosion in the number of code lines
- Hard to bug fixing

Using Modules

To solve the issues

1. split the original problem into sub-problems
2. implement software modules to solve them
3. combine all the results

Using Modules

To solve the issues

1. split the original problem into sub-problems
2. implement software modules to solve them
3. combine all the results

Using Modules

To solve the issues

1. split the original problem into sub-problems
2. implement software modules to solve them
3. combine all the results

Using Modules

To solve the issues

1. split the original problem into sub-problems
2. implement software modules to solve them
3. combine all the results

Such modules should be:

- **Usable** little and clear code to use them
- **Re-usable** write once, use many times
- **Isolated** the results of their execution depend exclusively on their code

Functions

Are blocks of instructions equipped with:

- a name
- some input parameters (potentially 0)
- an output

They are meant to implement mathematical functions.

Syntax

```
...

/* function signature */
<output type> <function name>(
        <parameter type> <formal parameter>,
         ...)

/* function definition */
{
   ...
   return <return value>; /* can be an
                              expression */
   ...
}
```

An Example

```
...

unsigned int fact(unsigned int n)
{
  unsigned int result=1;

  for (int i=1; i<=n; i=i+1)
    result=result*i;

  return result;
}

...
```

The Void Type

When a function perform a task (e.g., printing) and does not return a value, the output type is void and the return instruction can be avoided.

```
void print_even_or_odd(int n)
{
  if (2*(n/2)==n) {
    printf("even\n");

    return;
  }

  println("odd\n");
}
```

Thus, . . .

```
int main(int argc, char *argv[])
{
   ...
}
```

is the definition of a function named `main`.

This is the *main* function of your program.

What are the parameters and the output values?

Thus, ... (Cont'd)

POSIX programs must return an error code: 0 means success.

The first parameter of the `main` function is the number of parameters $+1$ in the execution command.

E.g., during the execution of

```
al@foo:~>./a.out a 12 3
```

`argc` has value 4.

Function Calls

Functions can be called by other functions by using the syntax:

```
<function name>(<actual parameter>, ...)
```

Function Calls

Functions can be called by other functions by using the syntax:

```
<function name>(<actual parameter>, ...)
```

The result of a call is a value having the function output type.

It can be used inside an expression.

Examples of Calls

```
int main(int argc, char *argv[])
{
  unsigned int k=3;

  while (k<500000) {
    printf("The␣factorial␣of␣%d",
           k);
    k=fact(k);
    printf("␣is␣" + k);
  }
}
```

Back to the Original Example

```
...
/* Factorial of 10 */
int fact=1;
for (int i=1; i<=10; i=i+1)
  fact=i*fact;

printf("The factorial of 10 is %d\n", fact);

/* Factorial of 7 */
fact=1;
for (int i=1; i<=7; i=i+1)
  fact=i*fact;

printf("The factorial of 7 is %d\n", fact);
...
```

Back to the Original Example (Cont'd)

```
unsigned int fact(unsigned int x)
{
    unsigned int result=1;

    for (int i=1; i<=x; i++)
      result=i*result;

    return result;
}
```

Back to the Original Example (Cont'd 2)

```
int  main(int argc, char *argv[])
{
    printf("The factorial of 10", fact(10));
    printf("The factorial of 7", fact(7));

    return 0;
}
```

Signatures vs Definitions

Function calls can occur only after signatures

Signatures can be specified without defining functions (end them with ";").

```
int test();

int  main(int argc, char *argv[]) {
    return test();
}

int test() {
    return 0;
}
```

What About Re-usability?

Can we avoid to re-write the same code for any new program?

What About Re-usability?

Can we avoid to re-write the same code for any new program?

Libraries are sets of functions that can be linked to programs

You can both implements your own libraries and use already developed ones.

Static vs Dynamic Libraries

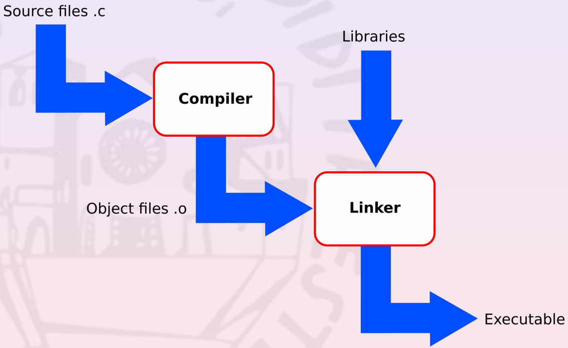
There are two kinds of libraries

Static library: their binary code is embeded into the program code (in GNU/Linux lib<name>.a)

Dynamic library: their binary code is loaded at runtime from a file which is shared (in GNU/Linux lib<name>.so)

Linkers

Are software to link different object files produced by a compiler.



GCC invokes the GNU linker, `ld`, by default.

How to Build a Dynamic Library with GCC

1. write the functions in a set of files e.g., `first_lib.c`
2. collect the signatures in one header file e.g., `first_lib.h`
3. build the dynamic library by using the options:

`-fPIC`	let code be position independent
`-shared`	produce an object that can be linked

```
al@foo:~/GP> gcc -fPIC -shared first_lib.c
                -o libflib.so
```

Including Header Files and Other Amenities

To call a library function we first need to declare its signature.

So, either we re-write its signature in any new program or...

Including Header Files and Other Amenities

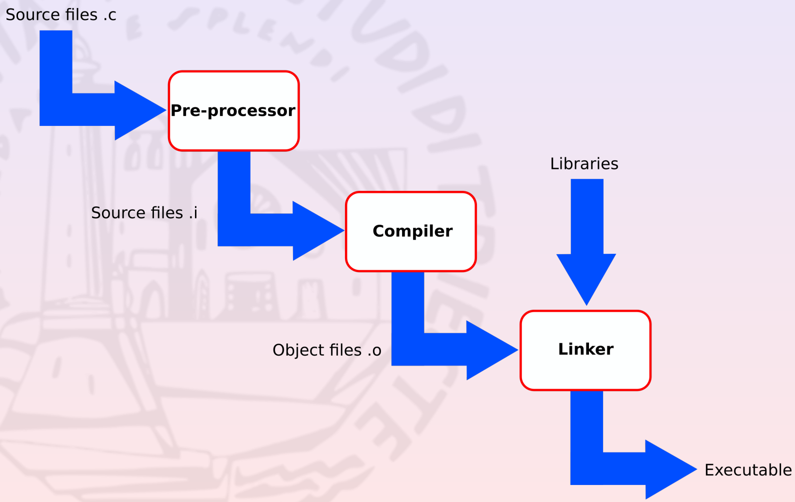
To call a library function we first need to declare its signature.

So, either we re-write its signature in any new program or...

we need a way to include its header file in the program.

This can be done by the pre-processor.

Including Header Files and Other Amenities



Including Header Files and Other Amenities

The pre-processor can:

- include files
- define and undefine macros
- evaluate macros

Every pre-processor directive begins with #.

Pre-processor Directive Examples

```c
#include <stdio.h>

#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))

#ifdef MIN

#undef min

#endif
```

How to Link a Dynamic Library with GCC

1. include the library header file
2. build your program by using the options:

   `-L<lib_path>` if the library is not in the standard libraries path

   `-Wl,-rpath=<lib_path>` if you have planned not to move the library to a standard library path

   `-l<name>` link the library `lib<name>.so`

```
gcc -L. -Wl,-rpath=. program.c -lflib
```

Coming soon...

- arrays
- pointers
- pointer arithmetic
- strings