System and Languages for Informatics

– 4 –

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
    1. Getting started with C Progamming
    2. Variables, Data-types, Operators and Control Flow
    3. Functions and Libraries
    4. Arrays and Pointers
    5. User defined datatype and data structure
    6. Input and Output
- Basic system programming in Linux (10h)

Overview

Motivation

- **Divide and conquer** - Break up(complex) problem into simpler sub-problems, each performing a special task.
- **Readability** - details are hidden from main program.
- **Simplicity** - Tasks can be *called* iteratively or recursively within loop.
- **Efficiency** - However functions are only useful, if transfer of state (i.e. communication) between the functions is *minimized*.
- For example, `printf("Hello World");`

Function Syntax

- Function is set of statements that together perform a task.
- Syntax

```
Output TYPE <function name>(Input TYPE
Parameter, Input TYPE Parameter, ...  )
```

Input and *output* are not mandatory.

- **Arguments** of the function go between the parantheses ()
  - There may be **no** arguments. Older C compilers require the keyword `void`.
  - There may be **multiple** arguments (type + parameter), separated by comma.

Function Syntax (2)

- **Return type** of the function goes to the left side of the expression
  - there may be **no** return type, handled by keyword `void`.
  - there may be **maximum one** return type.
  - Multiple return types? Several workarounds available.
- C standard library provides several built-in functions (`gets()`, `printf()`, `sqrt()`, etc.)

Example

- Writing pseudocode initially might help.
- Let's design a program that checks whether the number is prime.

Get num from command line
loop from i = 2 to num-1
  if modulo(a,i) gives zero, the number is prime
end
print result

Prime factor decomposition

```
int check_prime(int a)    /* Function prototype */
{                         /* { Function body } */
  int c;

  for (c=2; c<a; c++)
  {
    if (a%c == 0)
    return 0;
  }
  return 1;
}
```

Calling a function

- by its name with parameters as inputs
- output is set "=" to return value
- for example, `is_prime = check_prime(41);`
- Call:
  - **direct** if function in the same file.
  - **indirect** by telling the compiler its location, if the function is in another file.

the `main` function

```
int main() {
  /* do some stuff */
  return 0; /+ exit with success */
}
```

- Every C program has at least one function, namely `main()`.
- Initial function of C program.
- Several functions possible but only one `main()`.
- All functions are called from here.
- return type is `int`
  - `return 0` - successful termination
  - a non-zero return value indicates a failure or unexpected termination
  - Macros `EXIT_SUCCESS` and `EXIT_FAILURE` from `stdlib.h` can be used.

Example `myprogram.c` (cont'd)

```
#include<stdio.h>
int check_prime(int a)  {
  int c;
  for (c=2; c<a; c++)  {
    if (a%c == 0)
    return 0;
  }
  return 1;
}
int main() {
   int num, result;
   printf("Enter any number: ");
   scanf("%d", &num);
   result = check_prime(num);
   if (result == 1)  printf("%d is prime.\n", num);
   else printf("%d is not prime.\n", num);
   return 0;
}
```

Example (cont'd)

- Calling our program with `gcc -Wall -o myprogram myprogram.c`, we obtain

```
$ ./myprogram
$ Enter any number:  41
41 is prime.
```

Returning multiple values

- Our function `check_prime` returns one integer.
- Functions, working with primitive data types, return up to one value.
- Solution 1: Pointers in C
  `void myfunction(int *a, char *b)` [see next lesson]
- Solution 2: Array in C
  `int * myfunction(int *a)` [see next lesson]
- Solution 3: Use structure `struct()` [see following lesson]
- Solution 4: Use **global** variables.....

Scope rules

- The *scope* of a variable/function is the part of the program within which they are visible.
- **Global visibility** for identifies defined above all functions.
- Visible by all subsequent functions in the **same** source file, only.

```
#include<stdio.h>
int global_variable;

int main(void){
...  /* the global variable is visible here. */
return 0;
}
```

Scope rules (2)

Block Visibility: identifier is declared within a block, and limited to the block itself

```
int a;
scanf("%d",&a);
if (a>10) {
   int b = 10;
}
printf("%d",b);
```

What does the compiler say? Error!

Scope rules (3)

- Sometimes the source code for a program is contained in more than one text file.
- To make a global variable visible to the other source files, declare it **extern** there.

```c
/* main.c */

int main() {
extern int myvar;
myvar = 10;
print_myvalue();
return 0;
```

```c
/* extern.c */
#include <stdio.h>
int myvar;
void print_myvalue()
    {
    printf("myvar = %d\n", myvar);
    }
```

Scope rules (4)

- Compile with `gcc main.c extern.c`.

```
$ myvar = 10
```

- Another important class specifier is `static`. These variables **remain** their values even after they are out of their scope.

Example

```
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Shell

```
$ 1 2
```

Example (revisited)

```
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Shell

```
$ 1 1
```

Masking

Suppose an identifier is declared outside a block and redeclared inside the block. Then that inside the block **masks** the external.

```
#include<stdio.h>
int global_variable;

int main() {
double global_variable;
...
}
```

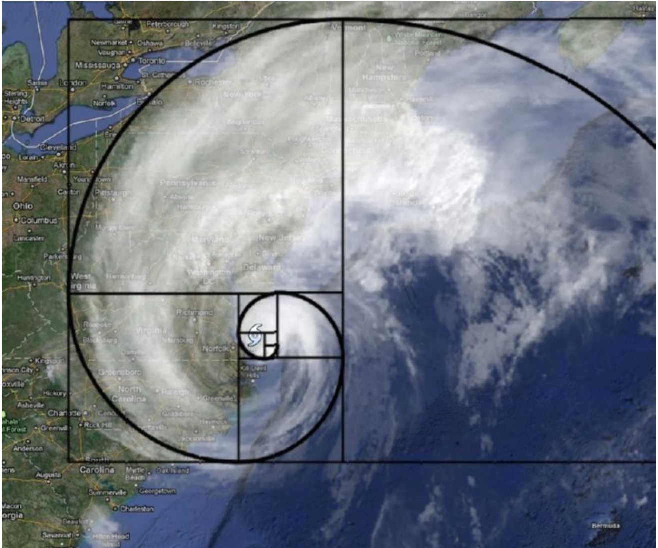
Recursive functions

A function that calls (in its block) itself.
For example, the Fibonacci sequence can be computed as

$$F_n = F_{n-1} + F_{n-2}; \quad n \geq 2; \quad F_0 = 0, F_1 = 1.$$

```
int fibonacci(int i)
{
   int res;
   if (i == 0)
       res = 0;
   else if (i == 1)
       res = 1 ;
   else
      res = fibonacci(i-1) + fibonacci(i-2);
   return res;
}
```

Recursive functions



Our example revisited

Let us return to our example from above:

```
#include<stdio.h>
int fun()     /* function specification */
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());  /* function call */
  printf("%d ", fun());
  return 0;
}
```

C programs do not need to be monolithic!

Function prototypes

Each function has to be declared before being used. The following conventions are typically used:

1. Declare all functions (but the `main`);
2. Define `main`;
3. Define all other functions.

In this way, each function is declared before being used:

Example:

```
int max(int,int);
int mcm(int,int);
int main(){ …}
int max(int a, int b){ … }
int mcm(int a, int b){…}
```

Function prototypes

- Alternative implementation with function prototype, making the compile aware, but without actual implementation.

```
#include<stdio.h>
int fun(); /* function prototype  */

int main() {
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}

int fun()
{
  int count = 0;
  count++;
  return count;
}
```

Too many lines of code

- Linux Code is written in ca. 12 Mio. lines of code.
- We need some mechanism to divide our code.
- **Modular** programming is essential.
  - Interface in header file (saved with extension **.h**)
  - Implementation in auxiliary source **.c**/object files **.o**

Implementation our function as `fun.c`

Implement `fun()` in `myfun.c`:

```
int fun()
{
  int count = 0;
  count++;
  return count;
}
```

Other functions can be embedded subsequently or in other auxiliary files.

The Interface `fun.h`

communicates all global variables and functions to other source files in form of **header files.**

- Function prototypes
- Struct, enum and custom type definitions
- Global variable declaration using the `extern` keyword
- Header guards
  - ensure that the contents of the header file will not be copied more than once in several files in your project (causing compilation errors).

Note

Header files should never contain any executable code.

Header files

- **Convention for header guards**: use two leading underscores with all letters in the name of the header file converted to uppercase and periods to underscores.

```
#ifndef __FUN_H
#define __FUN_H

/* declarations come here */
int fun();

#endif /* __FUN_H */
```

The modular program

```
/* main.c */
#include<stdio.h>
#include "fun.h"

int main() {
  printf("%d ",fun());
  printf("%d ",fun());
  return 0;
}
```

- Compile source with `gcc -Wall -o myprogram main.c fun.c`.

```
$ ./myprogram
1 1
```

By the way, ...

Each standard library has a *header file*, containing:

- definition of constants;
- definition of types;
- declaration of all library functions.

Esempi:

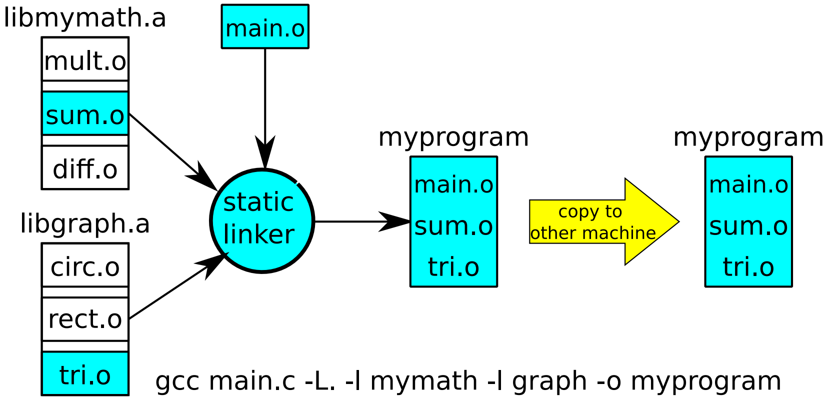
`<stdio.h>`	input/output
`<stdlib.h>`	memory, random, generic utils
`<string.h>`	strings
`<limits.h>`	limits of the system for integers
`<float.h>`	limits of the systems for float
`<math.h>`	math functions
...	

Libs can also be created by the programmer, such as ”mylib.h”.

Statically Linked Library (1)

- Set of routines, external functions and variables that are resolved at compile-time and copied into a target application by a compiler/linker. Resulting static library is a **stand-alone executable**.
- All the functions within the library are organized and indexed with a symbol and address, kind of TOC.
- Archive extension ***.a** (Linux) and ***.lib** (Windows).
- The Linker makes copy of all used library functions to the main executable file.
- Typical library functions are `printf()`,`scanf()`,`sqrt()`,etc.
- We may create a static library on our own.

Statically Linked Library (2)



Statically Linked Library (3)

Advantages

- Pre-compiled libraries **increase build speed and reduce dev times** in large projects.
- App can be sure that all libraries are present and up-to-date, avoiding dependency problems.
- Only **part** of the library, containing requested functions, are loaded (For dynamic libraries, the entire must be loaded.)
- App in a single executable file, simplifying distribution and installation.

Statically Linked Library (4)

Disadvantages

- Generally, **trust** that 3rd party library optimizes runtime and memory without security vulnerabilities.
- Deep third-party dependencies can slip under the radar.
- Specifically, **size** of executable becomes large, as all the library code is stored within the same executable rather than in separate files.

MyStaticLibrary (1)

- Suppose, we want to re-use a function, computing the sum of two integer numbers, throughout the same project.

```c
/* mult.c */
int mult(int a, int b)
{
    return (a * b);
}

/* sum.c */
int sum(int a, int b)
{
    return (a + b);
}
```

```c
/* mymath.h */
#ifndef __MYMATH_H
#define __MYMATH_H

int mult(int,int);
int sum(int,int);

#endif
```

MyStaticLibrary (2)

- Create object file by stopping GNU compiler at compiler stage with `-c` option (no executable):

```
$ gcc -Wall -c mult.c sum.c
```

- Make static library by archiving object file with the `-c` (**r**eplace pre-existing object files in the library with the same name and **c**reate archive without warning).
- Convention for linux is to use a filename starting with **lib-**.

```
$ ar -rc libmymath.a mult.o sum.o
```

MyStaticLibrary (3)

- To verify the symbol table of our library,

```
$ nm libmymath.a
mult.o:
0000000000000000 T mult
sum.o:
0000000000000000 T sum
```

- ”virtual address: 0”, ”text symbol, global”, name is ”sum”.
- Common symbol names used in the object file

b/B	uninitialized data, local/global
d/D	initialized data, local/global
L	Global thread-local symbol
t	Static thread-local symbol
U	Undefined symbol

Use MyStaticLibrary (1)

- We have created object files, zipped them in an library and indexed it. We want to use it in the main file.

```c
#include<stdio.h>
#include "mymath.h"

int main(void)
{
    int result;
    result = sum(5, 8);
    printf("result = %d \n",result);
    return (0);
}
```

Use MyStaticLibrary (2)

```
$ gcc -Wall main.c -L. -l mymath -o myprogram
```

Specifically,

-L	directory of library
.	current directory
-l	library file to be linked
mymath	library file without prefix
-o	name of executable

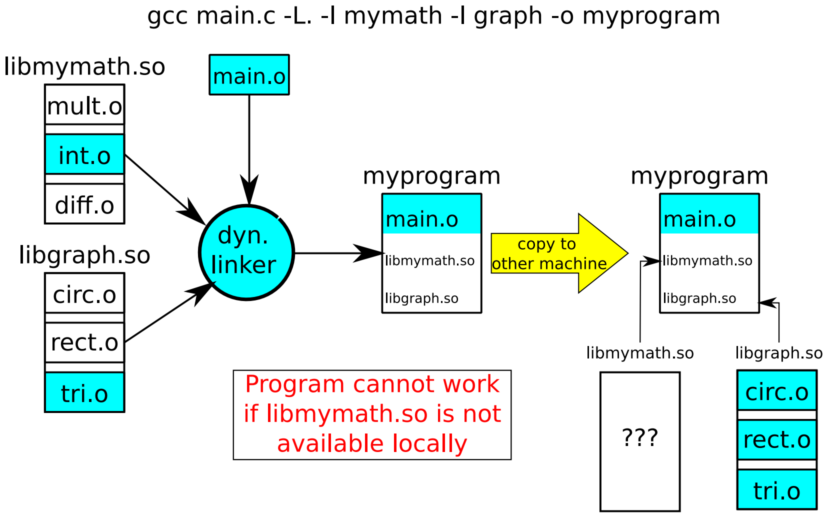
Shell

```
$ ./myprogram
result = 13
```

Shared (dynamic) Library (1)

- Also dynamic linking collects and combines multiple object files, to create a single executable, but ...
- Linking is performed in real-time as programs are executed (Remember that static libraries are put into an executable file already at compile time)
- Dynamic libraries are loaded into (separate) memory by the starting programs.
- Once loaded, library code can be used by any number of programs.

Shared (dynamic) Library (2)



Advantages

- Low memory footprint, as only **one** copy of the shared library is kept in memory.
- Libraries can be updated independent of the executable files.
- All running applications can use the same library without the need for each to have it's own copy.

Disadvantages

- Shared library attacks easily possible if not handled with care. For example, a malicious library can be linked according to

CAUTION

```
$LD_LIBRARY_PATH=/some-fake-dir/:$LD_LIBRARY_PATH
```

- Compatible is an issue. The new library version assumes compatibility with programs built for the previous one.
- Execution speed lower at run time, as the library is **only linked** to the executable file.

MySharedLibrary (1)

Let us return to our example.

```
/* mult.c */
int mult(int a, int b)
{
    return (a * b);
}

/* sum.c */
int sum(int a, int b)
{
    return (a + b);
}
```

```
/* mymath.h */
#ifndef __MYMATH_H
#define __MYMATH_H

int mult(int,int);
int sum(int,int);

#endif
```

MySharedLibrary (2)

1. Compiling into Position Independent Code (is not dependent on being located at a specific address in order to work.)

```
$ gcc -c -Wall -fpic mult.c sum.c
```

2. Create a shared library from object file

```
$ gcc -shared -o libmymath.so mult.o sum.o
```

3. Link program with our shared library

```
$ gcc -Wall main.c -L. -lmymath -o myprogram
```

Use MyShared Library (3)

Now let us run our program.

```
$ ./myprogram
```

Error

```
./myprogram: error while loading shared
libraries: libmymath.so: cannot open shared
object file: No such file or directory
```

4. Expose library at runtime
   - Prepend our working directory to the path.
   - Export the changes,

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
$ ./myprogram
result = 13
```

Quiz

```
extern int prod(int x, int y, int z)
{
    return (x * y * z);
}
```

What does the `extern` keyword do in above code?

1. It makes the function visible to the whole program.
2. It does nothing. All functions have external linkage by default.
3. The function is declared somewhere else (”externally”).
4. The scope of the function `prod` limited to its object file i.e, it is visible only in its object file.