Introduction to the C programming language

Lecture 3

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

November 21, 2008

Outline

1. Visibility, scope and lifetime
2. Structures
3. Casting
4. More on input/output: files
    - Exercises

Definitions

- **Global variables** are variables defined outside of any function
- **Local variables** are defined inside a function
- **The visibility** (or scope) of a variable is the set of statements that can “see” the variable
    - remember that a variable (or any other object) must be declared before it can be used
- **The lifetime** of a variable is the time during which the variable exists in memory

Examples

```
#include <stdio.h>

int pn[100];

int is_prime(int x)
{
    int i,j;
    ...
}

int temp;

int main()
{
    int res;
    char s[10];
    ...
}
```

pn is a global variable
scope: all program
lifetime: duration of the program

x is a parameter
scope: body of function is_prime
lifetime: during function execution

i,j are local variables
scope: body of function is_prime
lifetime: during function execution

temp is a global variable
scope: all objects defined after temp
lifetime: duration of the program

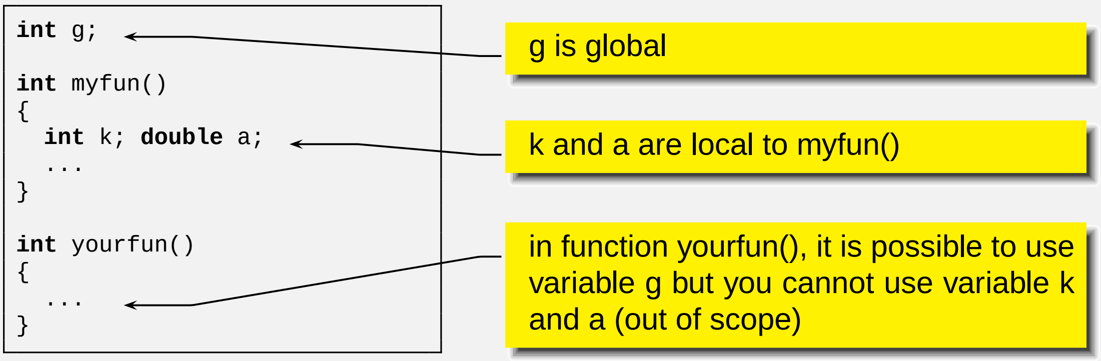
res and s[] are local variables
scope: body of function main
lifetime: duration of the program

Global scope

- A **global variable** is declared outside all functions
  - This variable is created before the program starts executing, and it exists until the program terminates
  - Hence, it's **lifetime** is the program duration
- The **scope** depends on the point in which it is declared
  - All variables and functions defined after the declaration can use it
  - Hence, it's scope depends on the position

Local variables

- Local variables are defined inside functions



```
int g;

int myfun()
{
  int k; double a;
  ...
}

int yourfun()
{
  ...
}
```

g is global

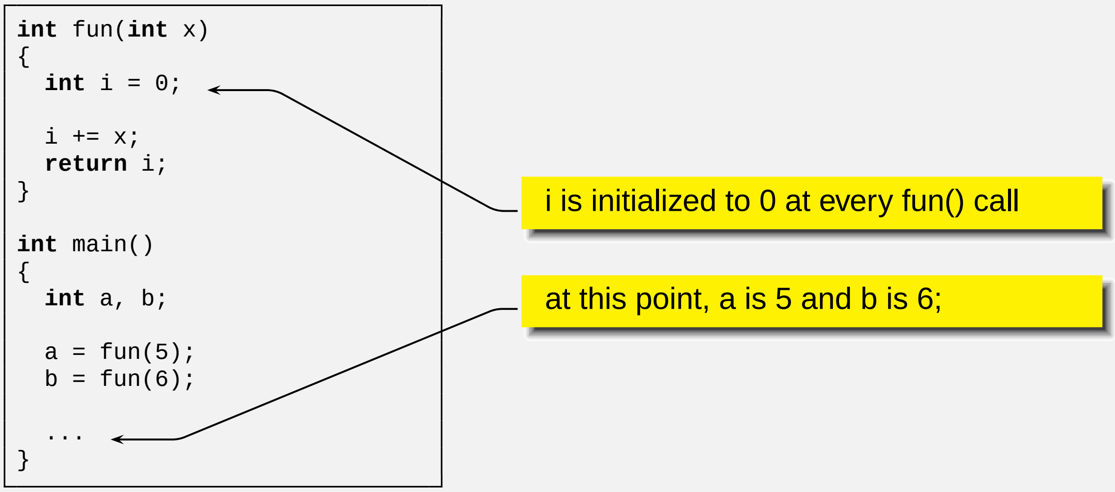
k and a are local to myfun()

in function yourfun(), it is possible to use variable g but you cannot use variable k and a (out of scope)

- `k` and `a` cannot be used in `yourfun()` because their scope is limited to function `myfun()`.

Local variable lifetime

- Local variable are *created* only when the function is invoked;
- They are *destroyed* when the function terminates
  - Their lifetime corresponds to the function execution
- Since they are created at every function call, they hold only temporary values useful for calculations
- their value is not kept between two calls!



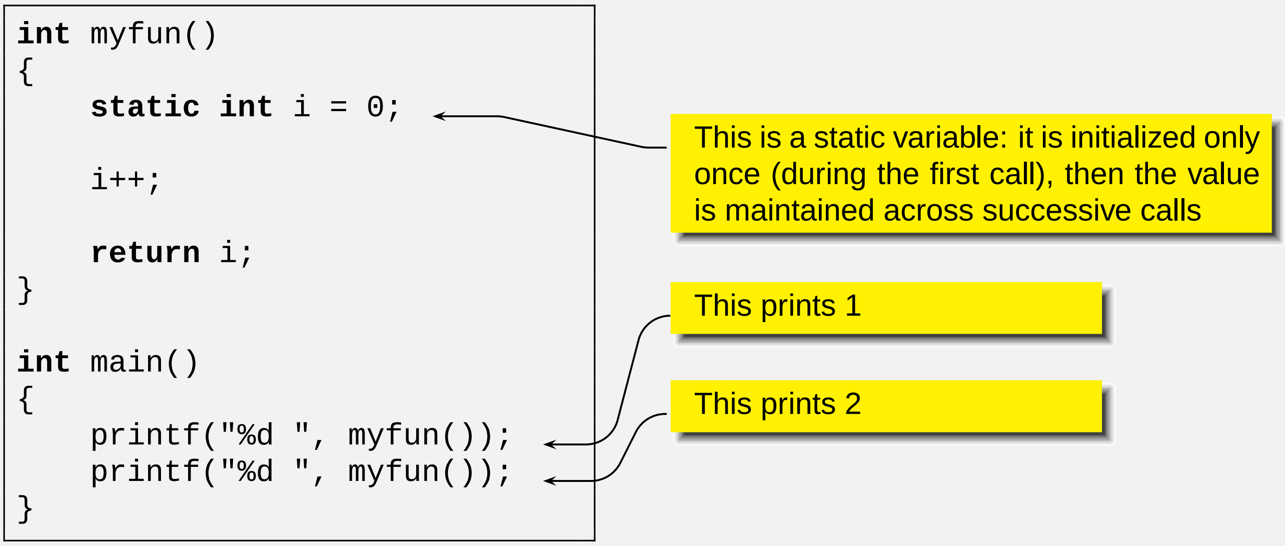
```
int fun(int x)
{
  int i = 0;

  i += x;
  return i;
}

int main()
{
  int a, b;

  a = fun(5);
  b = fun(6);

  ...
}
```

i is initialized to 0 at every fun() call

at this point, a is 5 and b is 6;

Modifying lifetime

- To modify the lifetime of a local variable, use the **`static`** keyword



```
int myfun()
{
    static int i = 0;

    i++;

    return i;
}

int main()
{
    printf("%d ", myfun());
    printf("%d ", myfun());
}
```

This is a static variable: it is initialized only once (during the first call), then the value is maintained across successive calls

This prints 1

This prints 2

Hiding

- It is possible to define two variables with the same name in two different scopes
- The compiler knows which variable to use depending on the scope
- It is also possible to hide a variable

```
int fun1()
{
  int i;
  ...
}
int fun2()
{
  int i;
  ...
  i++;
}
```

increments the local variable of fun2()

```
int i;
int fun1()
{
  int i;
  i++;
}
int fun2()
{
  i++;
}
```

Increments the local variable of fun1()

Increments the global variable

Structure definition

- In many cases we need to aggregate variables of different types that are related to the same concept
- each variable in the structure is called *a field*
- the structure is sometimes called *record*
- Example

```
struct student {
    char name[20];
    char surname[30];
    int age;
    int marks[20];
    char address[100];
    char country[100];
};

struct student s1;
```

```
struct position {
  double x;
  double y;
  double z;
};

struct position p1, p2, p3;
```

Accessing data

- To access a field of a structure, use the *dot notation*

```
struct student s1;
...
printf("Name: %s\n", s1.name);
printf("Age : %d\n", s1.age);
```

```
#include <math.h>

struct position p1;
...
p1.x = 10 * cos(0.74);
p1.y = 10 * sin(0.74);
```

Array of structures

- It is possible to declare array of structures as follows:

```
struct student  my_students[20];
int i;

my_student[0].name = "...";
my_student[0].age = "...";
...

for (i=0; i<20; i++) {
    printf("Student %d\n", i);
    printf("Name: %s\n", my_student[i].name);
    printf("Age: %d\n", my_student[i].age);
...
}
```

Other operations with structures

- When calling functions, structures are passed by value
  - that is, if you modify the parameter, you modify only the copy, and the original value is not modified
- Initialization: you can use curly braces to initialize a structure

```
struct point {
    double x;
    double y;
};

struct point x = {0.5, -7.1};
```

- You can use normal assignment between structures of the same type
  - the result is a field-by-field copy

```
struct point {
    double x;
    double y;
};

struct point x = {4.1, 5.0};

struct point y;

y = x;
```

Converting variables between types

- Sometimes we need to convert a variable between different types
- Example:

```
int a = 5;
double x;

x = a;

x = a / 2;

a = x * 2;
```

Here we have an implicit conversion from int to double; the compiler does not complain

Here we have an implicit conversion from int to double. However, the conversion is performed on the result of the division; therefore the result is 2 and not 2.5 as one might expect!

Here we have a conversion from double to int. With this conversion, we might lose in precision, hence the compiler issues a warning

Explicit casting

- It is possible to make casting explicit as follows

```
int a;
double x;

x = ((double) a) / 2;

a = (int)(x * 2);
```

Here the conversion is not explicit. First, a is converted to double; then, the division is performed (a fractional one); then the result (a double) is assigned to x.

Here the compiler does not issue any warning, because the programmer has made it explicit that he/she wants to do this operation.

A brief overview

- In the next slides we will present a quick overview of some functions to manipulate file
- These are useful to solve some exercises
- We will come back to these functions at some point

Files

- A file is a sequence of bytes, usually stored on mass-storage devices
    - We can read and/or write bytes from/to files sequentially (as in magnetic tapes)
- File can contais sequences of bytes (binary) or sequence of characters (text files)
    - There is really no difference: a character is nothing more than a byte
    - It's the *interpretation* that counts

File operations

- Before operating on a file, we must *open* it
- then we can operate on it
- finally we have to *close the file* when we have done
- in a C program, an open file is identified by a variable of type FILE *
    - The * denotes a pointer: we will see next lecture what a pointer is

Opening a file

- To open a file, call `fopen`

```
FILE *fopen(char *filename, char *mode);
```

- `filename` and `mode` are strings
  - `filename` is the name of the file (may include the path, relative or absolute)
  - `mode` is the opening mode
    - `"r"` for reading or `"w"` for writing or `"a"` for writing in append mode
- Example: open a file in reading mode

```
FILE *myfile;

myfile = fopen("textfile.txt", "r");
...

fclose(myfile);
```

Reading and writing

- At this stage, we will consider only text files
- You can use `fprintf()` and `fscan()`, similar to the functions yu have already seen

files/input.c

```
#include <stdio.h>

FILE *myfile;

int main()
{
    int a, b, c;
    char str[100];

    myfile = fopen("textfile.txt", "r");

    fscanf(myfile, "%d %d", &a, &b);
    fscanf(myfile, "%s", str);
    fscanf(myfile, "%d", &c);

    printf("what I have read:\n");
    printf("a = %d     b = %d      c = %d\n", a, b, c);
    printf("str = %s\n", str);
}
```

fprintf and fgets

files/output.c

```
#include <stdio.h>

FILE *myfile1;
FILE *myfile2;

int main()
{
    int i, nlines = 0;
    char str[255];

    myfile1 = fopen("textfile.txt", "r");
    myfile2 = fopen("copyfile.txt", "w");
    fgets(str, 255, myfile1);

    while (!feof(myfile1) {
        fprintf(myfile2, "%s", str);
        nlines++;
        fgets   (str, 255, myfile1);
    }
    printf("file has been copied!\n");
    printf("%d lines read\n", nlines);
}
```

Exercises with files

- Write a program that reads a file line by line and prints every line reversed
    - **Hint:** Write a function that reverts a string
- Write a function that reads a file and counts the number of words
    - **Hint:** two words are separated by spaces, commas “,”, full stop “.” , semicolon “;”, colon “:”, question mark “?”, exclamation mark “!”, dash “-”, brackets. see `http://en.wikipedia.org/wiki/Punctuation`
    - this is called tokenize