Basics of C

Lecture 4

Summer 2018
Innopolis University
Eugene Zouev

http://c-faq.com/decl/spiral.anderson.html
[This was posted to comp.lang.c by its author, David Anderson, on **1994-05-06**.]

The ”Clockwise/Spiral Rule”

By David Anderson

- How to “bootstrap” C declarations ☺

Previous classes:

- C memory model
- Typical program structure
- Declarations & types
- Pointers, arrays
- Global/local & dynamic objects
- Static/auto & global/local objects

Today:

- Structures & unions
- Memory model: execution stack
- Scopes & blocks

Structures

ISO C Standard, 6.7.2.1, §6

…A structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence

Type tag

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

Member declarations

- **How to declare structures?**
- **How to use structures?**

Structures: how to declare

Two ways (as usual for *C*):

- **Static**
- **Dynamic**

```
struct S
{
   int a, b;
   double c;
   int* p;
};
```

```
struct S s1;
```

s1 is the object of type **struct** S created **in the stack** and accessible from within a local scope or in the global scope.

```
struct S* ps =
     (struct S*)malloc(sizeof(struct S));
```

ps points to the dynamic object of type **struct** S created **in the heap** and accessible via pointer.

Structures: how to use

Two ways of accessing structure elements:

- Via name
- Via pointer

Via name: dot notation

struct-name . member-name

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

```
struct S s1;
...
s1.a = 777;
s1.b = (int)s1.p;
...
```

Structures: how to use

Two ways of accessing structure elements:

- **Via name**
- **Via pointer**

Via pointer: **arrow notation**

pointer-to-struct -> member-name

```
struct S* ps =
     (struct S*)malloc(sizeof(struct S));
...
ps->a = 777;
ps->b = (int)ps->p;
...
```

Structures: details

Design points:

- Notation p->m seems to be unclear: actually pointer p doesn't point to the m member – it points to the structure as a whole! ☺
- Why have two kinds of notation **for the same notion**?

Mixing of access kinds:

```
struct S
{
  int a, b;
  double c;
  int* p;
};
```

```
struct S s1;
...
s1.a = 777;
...
struct S* p = &s1;
...
p->a = 777;
```

s1 designates (refers to) **the same object** as p points to.

Nested Structures

```c
struct Person
{
  char* name;
  struct { int unique_num, salary; } personal_info;
  int* extra_info;
};
```

```c
struct Person john;
...
john.name = "John";
john.personal_info.unique_num = 12345678;
...
struct Person* p = &john;
...
p->personal_info.salary += 100;
```

Structures: Initialization

```
struct SheetOfPaper
{
  int height;
  int width;
};
...
struct SheetOfPaper letter;
letter.height = 279;
letter.width = 216;
...
struct SheetOfPaper A4 =
          { .height = 210, .width = 297 };
```

Structures: Alignment

```
struct S
{
   char* m1;
   short m2[3];
   long m3;
};
```

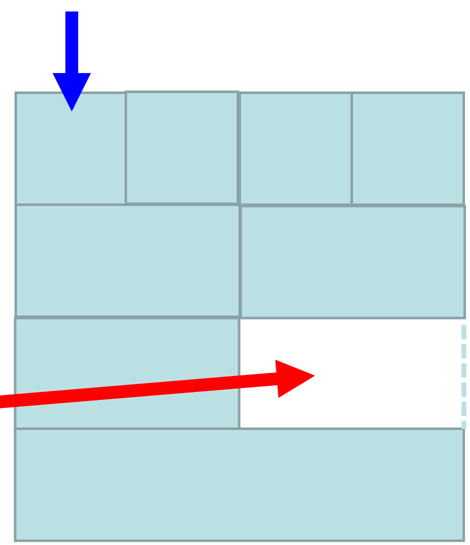
```
...
struct S s;
...
```

What about the following equation:

```
sizeof(s) == ?
    sizeof(s.m1) + sizeof(s.m2) + sizeof(m3);
```

S internal layout:

Addressable bytes



m1 is pointer to char: **4 bytes**

m[0], m[1] are shorts; **2 bytes** each

m[2] is short: **2 bytes**

m3 is long: **4 bytes**

This memory is not used!

Structures: bit-fields

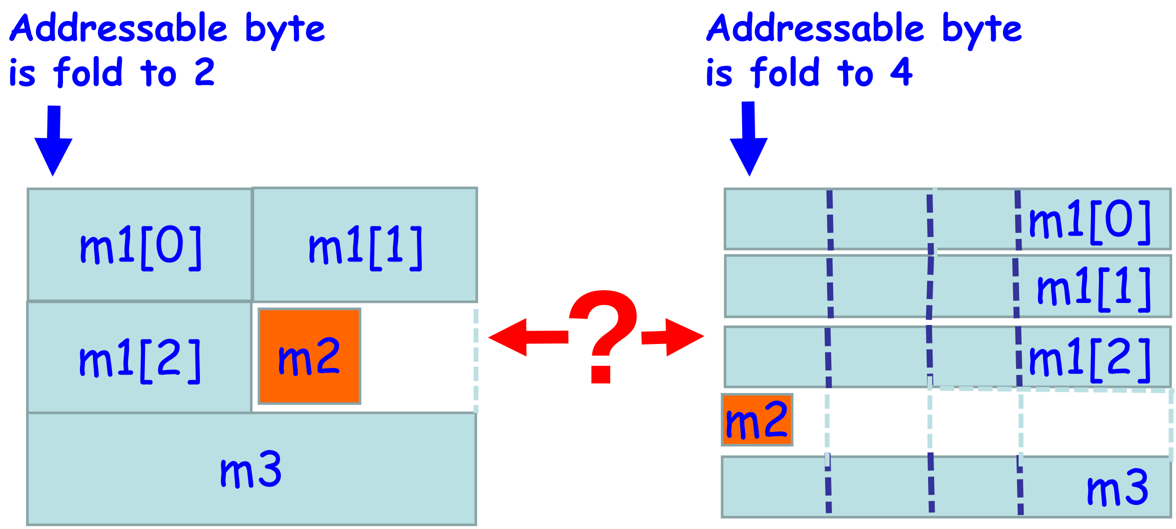
ISO C Standard, 6.7.2.1, §9-11

…A member may be declared to consist of a **specified number of bits** (including a sign bit, if any). Such a member is called a *bit-field.*

A bit-field is interpreted as having **a signed or unsigned integer type** consisting of the specified number of bits

An implementation may allocate **any addressable storage unit** large enough to hold a bit-field.

```
struct S
{
  short m1[3];
  int m2:5;
  long m3;
};
```



Which layout is used?
- Implementation-defined!

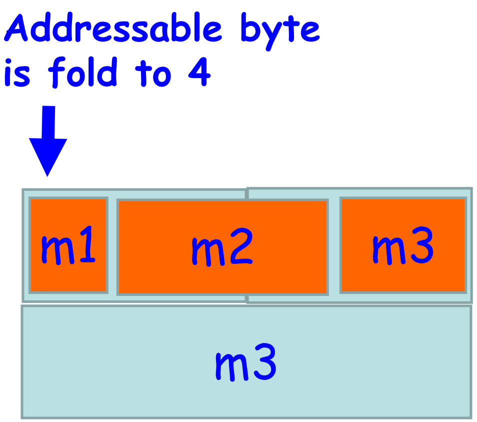
Structures: bit-fields

ISO C Standard, 6.7.2.1, §11

…If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed **into adjacent bits of the same unit**.

```
struct MyLayout
{
  unsigned int m1:2;
  unsigned int m2:10;
  unsigned int   :4;
  long m3;
};
```

Unnamed bit-field

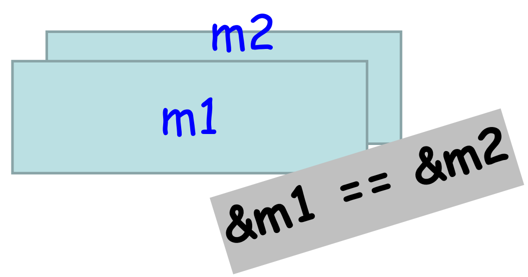


Structures: bit-fields

ISO C Standard, 6.7.2.1, §6

…**Union** is a type consisting of a sequence of members whose storage **overlap**.

```
union U
{
  unsigned int m1;
  int*         m2;
};
```

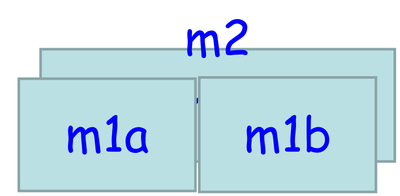


Both m1 and m2 are in the same memory.

sizeof(U) == max(**sizeof**(m1), **sizeof**(m2))

```
int x;
union U u;
...
u.m2 = &x;
...
unsigned y = u.m1;
```

```
union U1
{
  int m1a, m1b;
  int* m2;
};
```



Memory Model: Stack

- In C as well as in most modern languages the execution is centered around the **execution stack**.
- All algorithms are organized into functions (sometimes called procedures, methods, routines etc.)
- The order of execution of functions is LIFO, i.e. the last function called is the first to terminate (this behavior is obtained using the stack)

Typical operations on stack:
push, pop, empty

Memory Model: Stack

Stack & Activation Record 1

Call chain

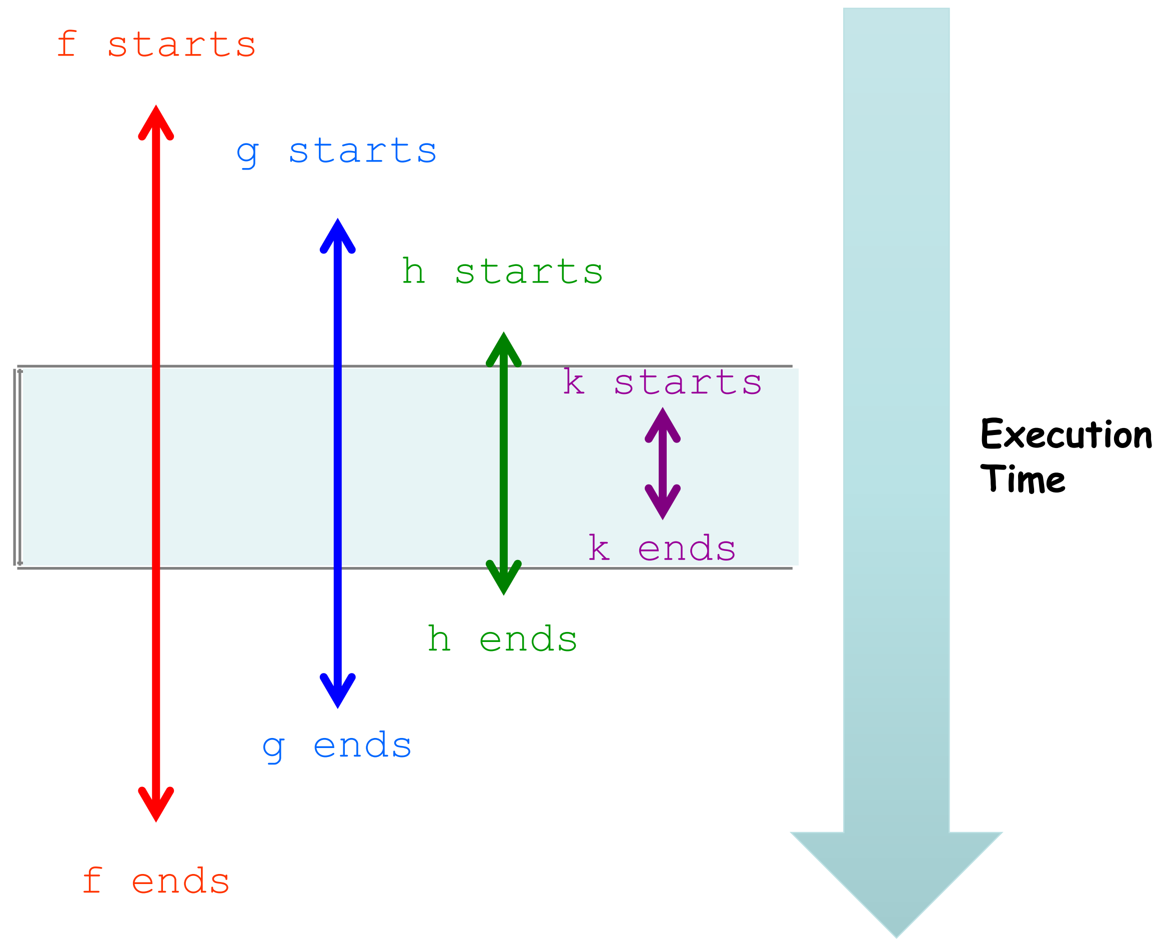
Return chain

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```

Stack & Activation Record 2

Code

```
void f() {
    int i = 3;
    g();
}
void g() {
    int i = 4;
    i = i + h();
}
int h() {
    return 1 + k(5);
}
int k(int z) {
    return z+1;
}
```



Stack & Activation Record 3

- Each time a function is called, **all the information specifically needed for the function** execution are put on the stack
- That information is collectively called the **activation record (AR)** of the function call
- This allows recursion, since for each call there will be a separate activation record on the stack
- When the call is completed (the function "returns") the corresponding AR is destroyed ("popped out" of the stack)
- Activation records are organized from bottom to top in memory diagram

Stack & Activation Record 4

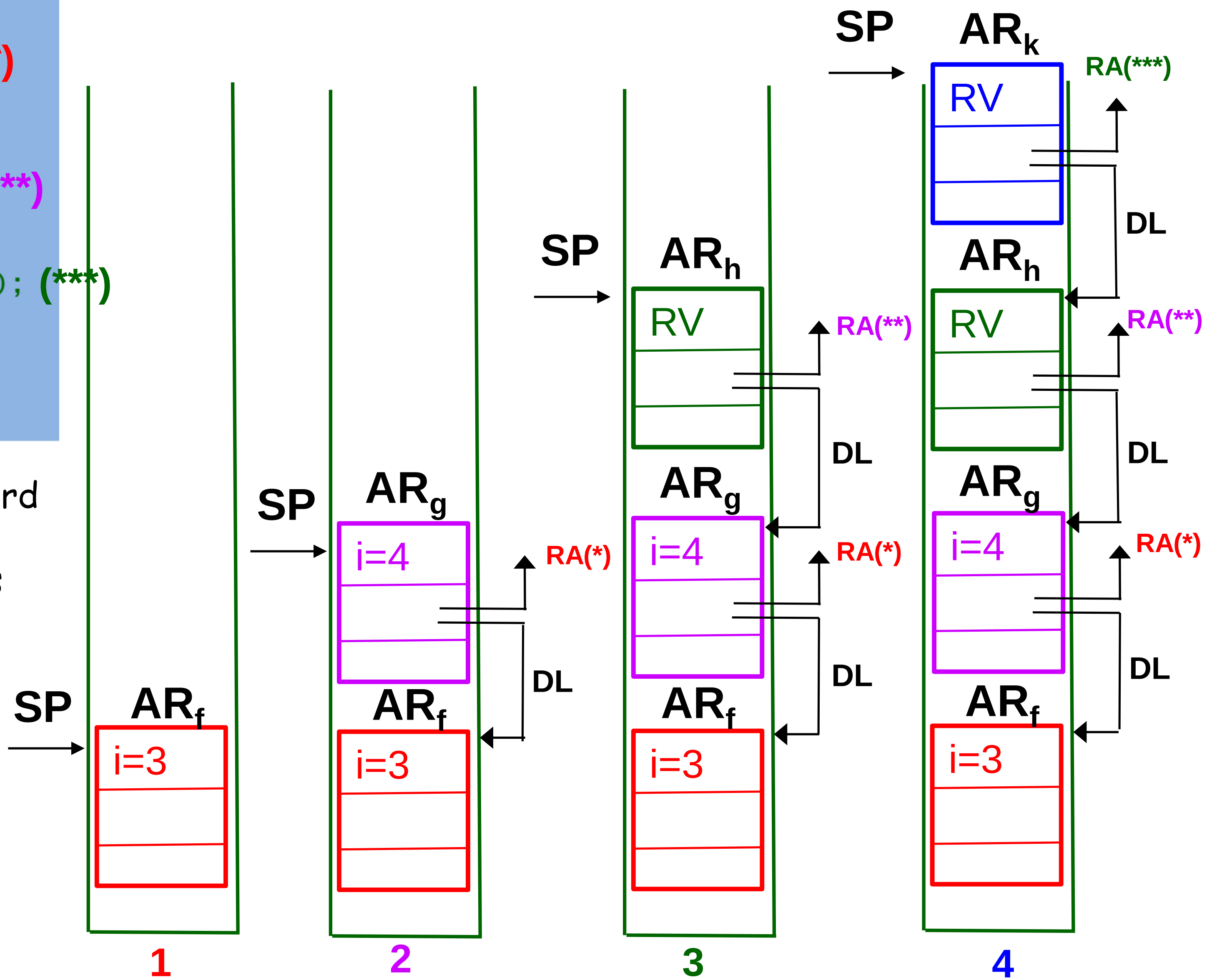
The information stored in the AR (also known as Stack Frame) for one call are the following:

- Information to restart the execution at the end of the call, i.e. after the function "returns"; these usually are:
  - **Return address**
  - **Pointer to the Stack portion devoted to the calling function**
  - **Return value** (if any)
- Information needed to perform the computation (usually the **actual arguments** passed to the function in the call – if any)
- **Local variables** (if any)

Stack & Activation Record 5

1
```
void f() {
      int i = 3;
      g();              (*)
}
```
2
```
void g() {
      int i = 4;
      i = i + h();    (**)
}
```
3
```
int h() {
      return 1 + k(5); (***)
}
```
4
```
int k(int z) {
      return z+1;
}
```

AR – Activation Record
SP – Stack Pointer
RA – Return Address
RV – Return Value
DL – Dynamic Link



Scope of a Variable

- The **scope** of a variable is a portion of the (source) code in which that variable is visible
  - the scope is where in the code we can refer to the variable declared
- Scoping rules (of some language) define the scope of a variable
- Scoping rules may vary from language to language and also among different declaration types in the same language
  - i.e. scoping rules for variable declarations may be different from those for function declarations

Blocks

- In most structured high-level languages the notion of **block** is central to scope identification
- A block is a portion of code enclosed between two special symbols, which mark the beginning and the end of the block.
  - In C (in Java, C++ etc.) blocks are marked by curly braces:

    { *this is a block* **}**

  - In some other languages blocks are marked by **begin** and **end** keywords or in some other manner (e.g. implicitly).
- Usually, blocks can be **nested**; but some language-dependent limitations are possible.

Scopes & Blocks

- Variable is visible
  - In the block it is defined
    - Starting from the line of definition
  - In all inner bocks unless a variable of the same name is declared within
- Global variables (if exist in the language)
  - Defined outside the scope of any block
- Hiding a variable
  - *A* homonymous variable declared within a block makes a variable of the same name declared outside <u>invisible</u>

Scopes & Blocks

- Scope is a **rule** determining existence and visibility of variables.
- Block is a compound language **construct** where variables (and other program entities) are declared.
- Declared entities are valid only within their scope, e.g. a variable exists only in its scope. The system is unaware of these entities in other parts of the code.

Scopes & Blocks: an Example

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

Function body is the block

The scope of i starts from its declaration until the end of the block except inner scope where local i is declared

The loop body is the block. j and k are declared in the block that is the scope for them

The scope of inner i is this block. The local i <u>hides</u> the i from the outer block

The scope of inner j is this block. The local j <u>hides</u> the j from the outer block

Function body is the block. The scope for z and i is the body. g's i <u>is not related</u> to f's i.

Scope Activation Record 1

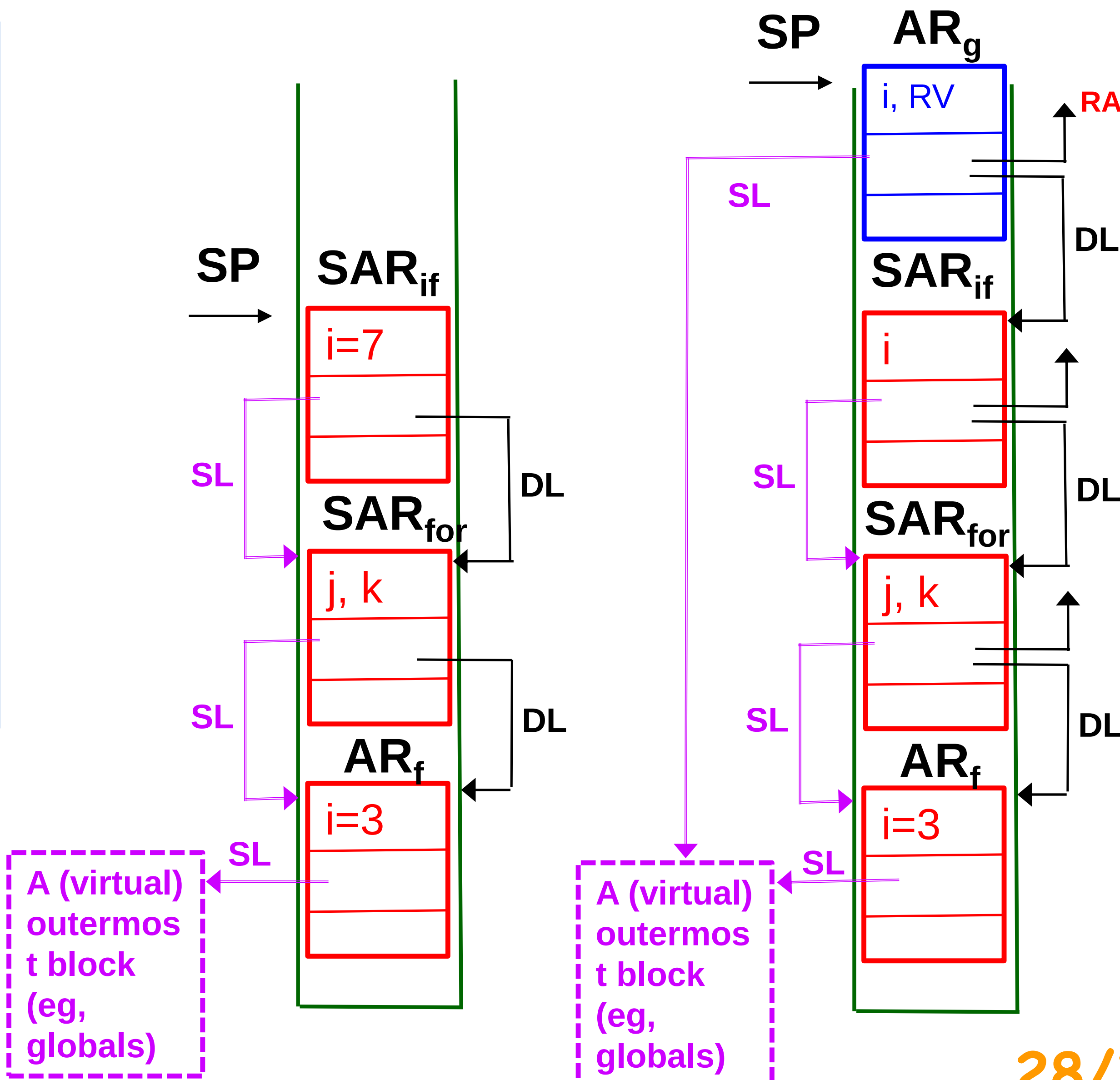
- In order to support blocks & scopes the extension to AR based model is used: Scope Activation Record (SAR)
- Every time a new block is encountered in the program flow, the SAR corresponding to this block is put on the stack.
- The SAR of a block is analogous to the AR of a function call, and it holds all the necessary information supporting visibility rules.

To remind:
the difference between **functions** and (usual) **blocks** is that blocks can be **nested** whereas functions cannot (in C)

Scope Activation Record 2

```
void f()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( condition )
        {
            int i = 7;
            ...i+k...g(k)...
        }
        else
        {
            int j = g(k+i);
            ...
        }
    }
}
int g(int z) {
    int i = z+1;
    ...
    return i*i;
}
```

In these points, we are interested of how to access to **i** variables and **k** variable



Scope Activation Record 3

- SARs contain, at least, two different kinds of information:
  - **local variables** (local to the block itself)
  - **the Static Link** (SL - also known as SAR link)
- In languages like C (Java, C++) variables defined in a block are always local to that block.
- The **SAR link** is a pointer to the SAR of the immediate enclosing block and it is used to **access local variables of outer blocks** (in a recursive fashion) from the current block, thus enforcing scoping rules at implementation level
  - Each time a variable is used in a scope, but there is no definition of such variable in such scope, the system uses the SL to reach out for the next enclosing scope to find that variable.
  - If it is not there, the SL is used to reach the next enclosing scope, and so on, recursively; until reaching the global scope.

Exercise 1

Consider this (actually, very stupid ☺) program:

```
void main()
{
    int i = 3;
    for ( int j=0; j<20; j++ )
    {
        int k;
        if ( j == 10 )
        {
            int i = i+k;
            g(k);
        }
        else
        {
            int j = g(k+i);
        }
    }
}
```

```
int g(int z) {
    int i = z+1;
    f();
    return i*i;
}
void f()
{
    int p = 0;
    while ( p <=10 )
    {
        p++;
        g(p); (*)
    }
}
```

Suppose the execution starts from the `main` function.

1. Draw the stack diagram for the case when the control flow comes to the point (*).
2. Find the ***logical*** problem with the program.

Exercise 2

There are no nested functions in C. However, some other modern languages (eg, **Scala**) do have nested functions.

Suppose there is nesting for functions in the language. Draw the stack configuration for the following program at the point marked by **(*)**.

```
void main()
{
    long factorial(long n)
    {
        if ( n <= 1 ) {
            return 1;
        else
            return n*factorial(n-1);
    }
    long testFactorial(long n)
    {
        for ( int j=1; j<n; j++ )
        {
            long i = factorial(j);
            if ( i > 100 )
            {
                signal(i);     (*)
                break;
            }
        }
    }
    testFactorial(100);
}
void signal(long i)
{
}
```