innovate achieve lead

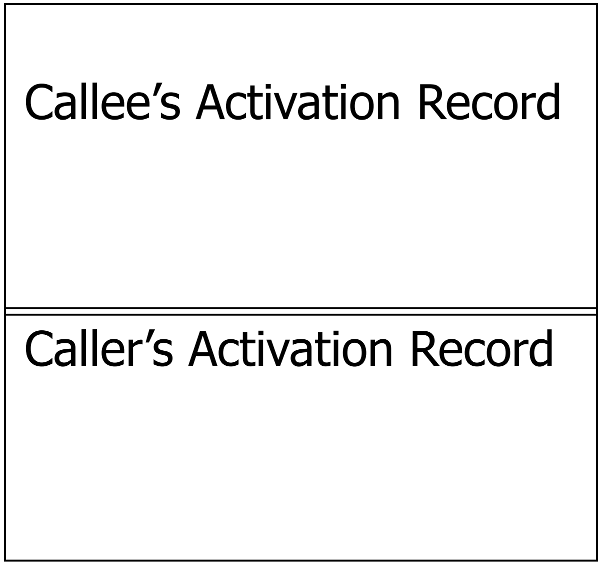
BITS Pilani

Lecture 17
Procedure Activations

Communication of values between the caller and the callee

- Formal Parameters (part of callee's activation record)
- Actual parameters (part of caller's activation record)
- Return value (part of callee's activation record)

- **Above values are placed in the beginning of the callee's Activation Records**



Activation record structure

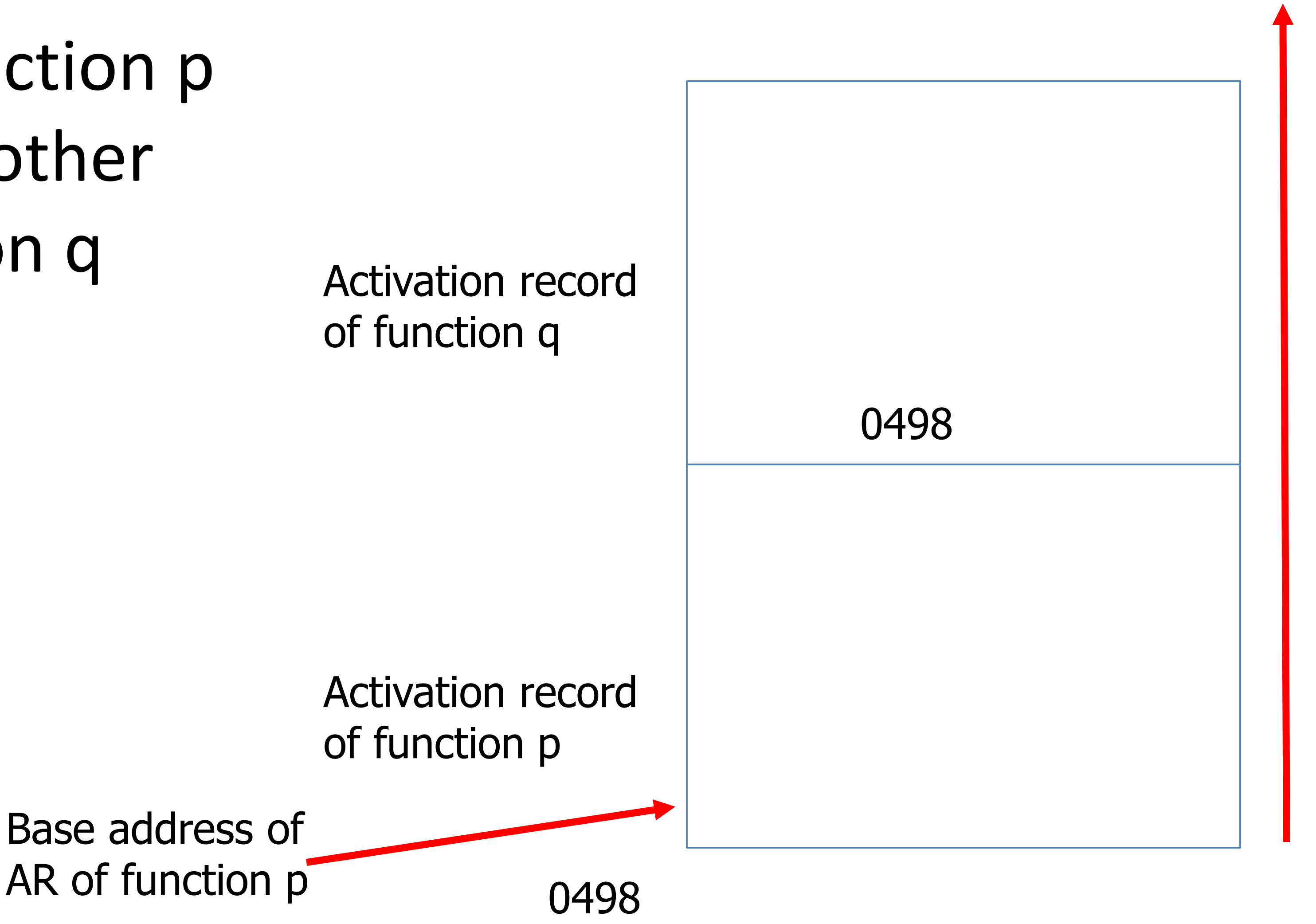
- Values communicated between caller and callee- beginning of the callee’s activation record
- Fixed length items (control link, access link, m/c status fields)- middle of the activation record
- Local variables (fixed length, variable length)- placed at the end
- Temporaries (values are finally mapped in registers)- not known before code generation

Control and access links

- Control link: This is the link used at run time to maintain address of the calling function’s activation record.
- Access link: This is used to implement the nested scoped languages following static scope rules.

Control link

- Let function p call another function q



Activation record of function q

0498

Activation record of function p

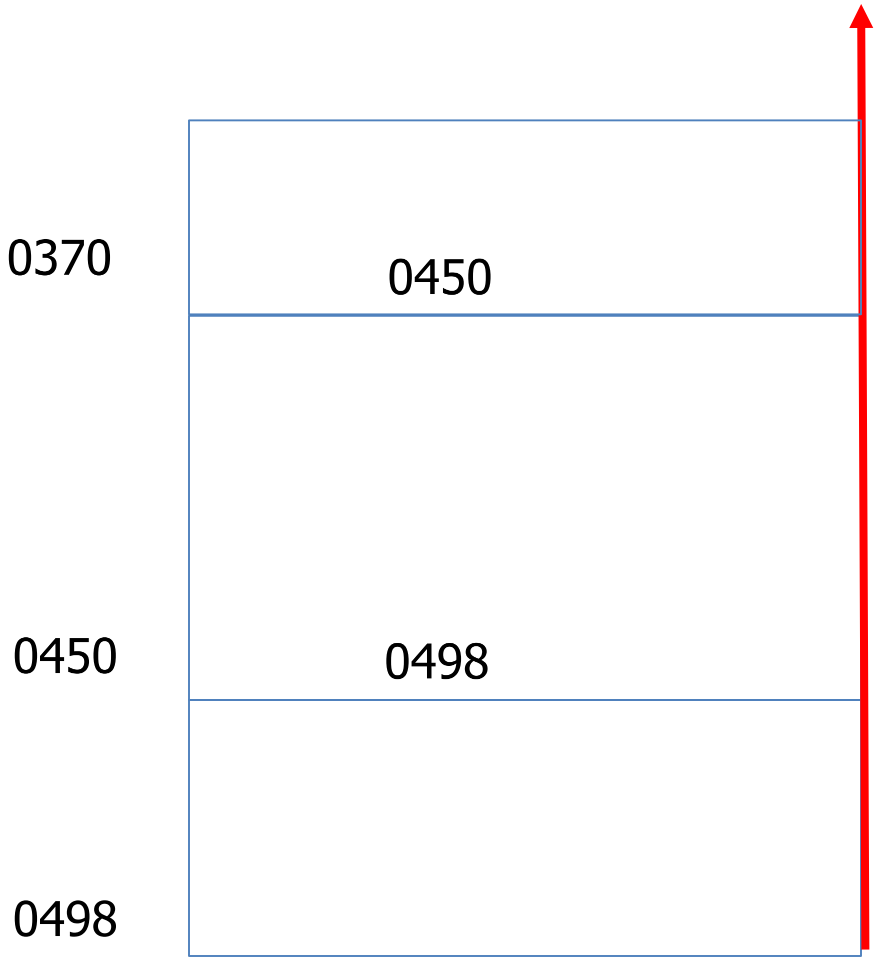
Base address of AR of function p

0498

Growth of call stack

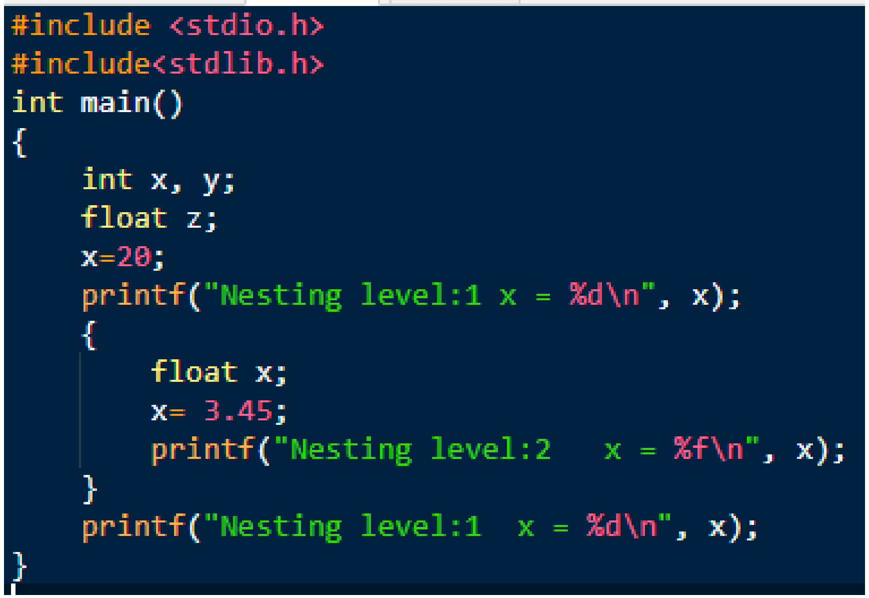
- function_1() calls function_2()
- function_2() calls function_3()

- Then there are three activation records active at run time on the call stack.
- The activation record of function_3() is at the top of the stack.
- The activation record of function_1() is at the bottom of the stack.

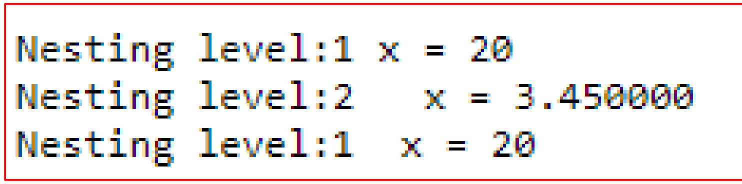


Scope of a variable

- Scope rules of a language determine which declaration of a name x applies to an occurrence of x in a program.
- Example:



```c
#include <stdio.h>
#include<stdlib.h>
int main()
{
    int x, y;
    float z;
    x=20;
    printf("Nesting level:1 x = %d\n", x);
    {
        float x;
        x= 3.45;
        printf("Nesting level:2    x = %f\n", x);
    }
    printf("Nesting level:1  x = %d\n", x);
}
```



```
Nesting level:1 x = 20
Nesting level:2    x = 3.450000
Nesting level:1  x = 20
```

Nested procedure definitions

```
Function_definition <parameter_list>
{
        local variables x, y, z
        ..........................
          function_definition_f2 <parameter_list>
           {
                local variables u, v, w
                use of u, v, x in an expression
          }
}
```

Languages that support nested procedures

- Algol 60
- Algol 68
- Pascal
- Ada
- Javascript
- Python
- Ruby
- Lua

etc.

Scope rules

- Scope rules determine how a particular occurrence of a name is associated with a variable.
- This association provides base for **non local** accesses of the variables.
- The scope can be one block of code text, or a procedure/function definition or the code segment of another procedure/function (parent) having nested definition of another procedure/function (child)

Example: nested scope

```
#include<stdio.h>
main()
{
        int p = 7;
        int q = 4;
        int r = 15;
        {
                int p = 6;
                int r = 12;
                p = p + q + r;
                printf("L1: %d %d %d ", p,q,r);
                {
                        int r =21;
                        int q = -2;
                        p = p + q + r;
                        printf("   L2: %d %d %d ", p,q,r);
                }
                {
                        int q = 11;
                        int p = 19;
                        p = p + q + r;
                        printf("   L3: %d %d %d ", p,q,r);
                }
        }
        p = p + q + r;
        printf("   L4: %d %d %d ", p, q, r);
}
```

Local and non-local variables

- Local variable: A variable is local in a function definition or block, if it is declared there.
- Non-local variable: These variables are visible in the program block or function definition, but are not declared there.
  - ❑ A variable occurrence as an argument in an expression or as an actual parameter looks for the declaration of that variable that applies to it based on the scope rules
  - ❑ Example

```
int p = 6;
int r = 12;
p = p + q + r;
printf("L1: %d %d %d ", p,q,r);
{
        int r =21;
        int q = -2;
        p = p + q + r;
        printf("   L2: %d %d %d ", p,q,r);
}
```

Variables r and q are local to the block

The variable p is non-local

The expression uses p, q and r on the RHS

Scope rules for binding names to non-local variables

- Static scope
- Dynamic scope

Static scope

- The scope of the variable is statically computed. It is compile time computable
- Which declaration applies to a name occurrence of a variable is known only by looking at the source code.
- The code segment holding the non-local variable's declaration is known as Static Parent.

Dynamic Scope

- This is based on the calling sequence of the functions.
- The association of the named occurrence (in a piece of code of the function say C1) of a variable to its declaration (type definition) does not depend on the position in the code text. It depends on the text segment of the function that called C1.

Example code

```
f1(){
        int x, y, z;         //D1
        x=10; y=15; z= 7;
        x=x*2+y;             //U1
        f2(){
                int  w;    //D2
                w= z + x;   //U2
        }
        f3(){
                int z, u;  //D3
                z=17;
                u = z + y - x;  //U3
                f2();
        }
        call f3();
}
```

Consider the variable occurrence of z at line U2

Which definition of z applies here? (to get the data from the associated location)

Think of offset of the location bound to z

Static scoping: D1 applies for z at U2

Dynamic scoping: D3 applies for z at U2

Example code

```
f1(){
        int x, y, z;          //D1
        x=10; y=15; z= 7;
        x=x*2+y;              //U1
        f2(){
                int  w;    //D2
                w= z + x;   //U2
        }
        f3(){
                int z, u;  //D3
                z=17;
                u = z + y - x;  //U3
                f2();
        }
        call f3();
}
```

Static parent of f2() is f1()

Static parent of f3() is f1()

Dynamic parent of f2() is f3()

Dynamic parent of f3() is f1()

Dynamic ancestor of f2() is f1()

Static scoping: D1 applies for x at U2

Dynamic scoping: D1 applies for x at U2