NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 14: The procedure abstraction

2020-02-25

Michael Engel

Overview

- Procedures and encapsulation
  - Structured programming
  - The procedure abstraction
  - Activation records

Giving programs a structure

- So far, we have considered sequences of instructions
- Early programs were often unstructured
  - Only global variables
  - Repetition of code
  - Common source of many programming errors
- **Idea:** Introduce structure and hierarchy into programs [1]
  - split program into procedures
  - scopes for names of variables, functions, etc.

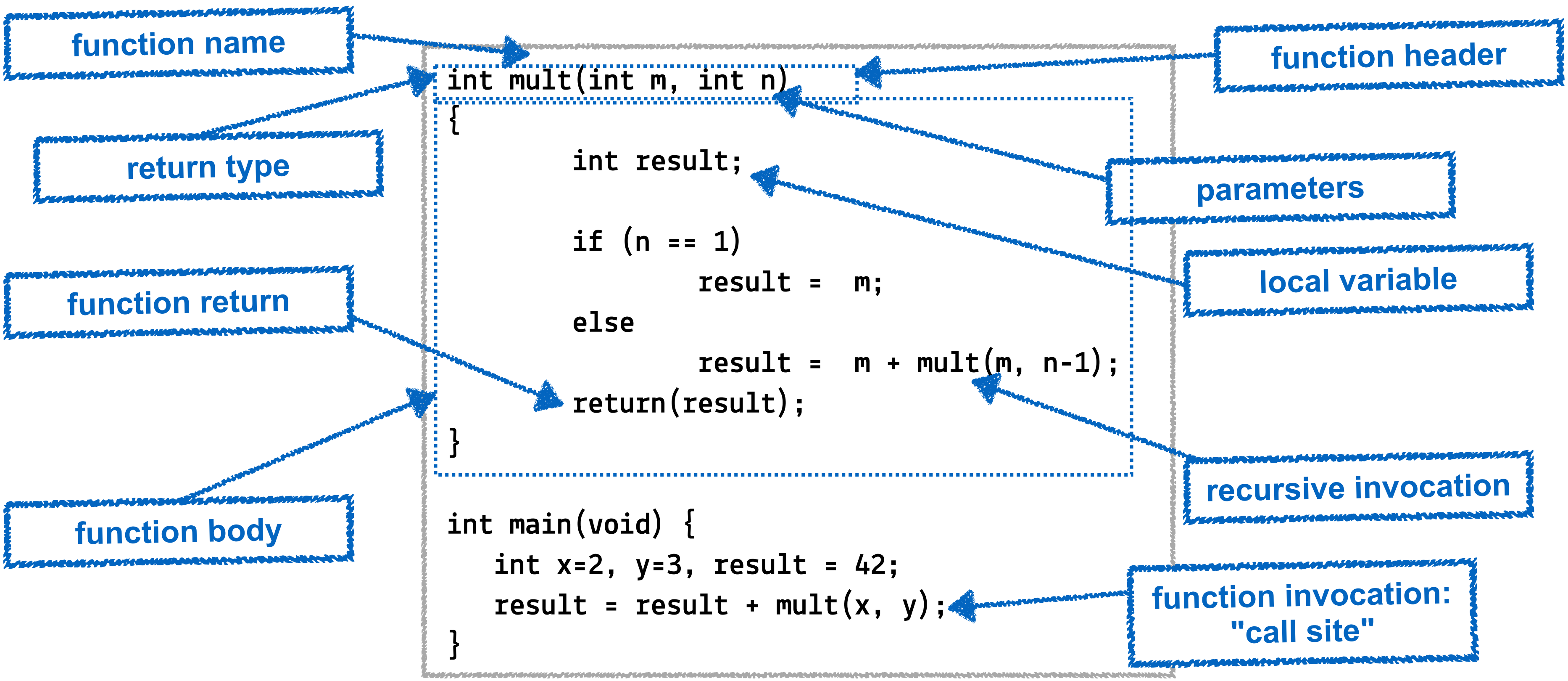
```
int main(void) {
    int x = 0;
    char *a, *b;
    while (*a++) x++;
    while (*b++) x++;
}
```

```
int strlen(char *s) {
  int len = 0;
  while (*s++) len++;
  return len;
}

int main(void) {
  int x;
  char *a="Hello", *b="World";
  x = strlen(a)+strlen(b);
}
```

The anatomy of a procedure

Example: C functions

- Some languages distinguish between functions and procedures
- Functions return a value, procedures don't



Concepts of procedures

Procedures are a ***programming abstraction*** that makes the development of large software systems practical and possible by

- **Information hiding**
  - The structure and content of data objects used inside a procedure is hidden from the rest of the program
- Distinct and separable **name spaces**
  - Data objects used inside a procedure do not interfere with identically named objects of other procedures or on global scope
- Uniform **interfaces**
  - Procedures provide a pattern to model the access to data

- There is usually almost no hardware support for implementing procedures
  - The compiler has to provide efficient implementations

Information hiding

- **Information hiding**
  - The structure and content of data objects used inside a procedure is hidden from the rest of the program

- In our example:
  - Type and name of local variable **`result`** is not known outside of function **`mult`**
  - **`main`** or other functions cannot access the value of **`result`** inside of **`mult`**

```
int mult(int m, int n)
{
        int result;

        if (n == 1)
                result =  m;
        else
                result =  m + mult(m, n-1);
        return(result);
}

int main(void) {
   int x=2, y=3, result = 42;
   result = result + mult(x, y);
}
```

Name spaces

- Distinct and separable **name spaces**
  - Data objects used inside a procedure do not interfere with identically named objects of other procedures or on global scope

- In our example:
  - There are variables named `result` declared ***both*** in function `mult` and `main`
  - Code inside of function `mult` cannot "see" `main`'s variable `result` → `result` in `main` retains its value across the call to `mult`
  - The compiler has to implement this "***lexical scoping***"

```
int mult(int m, int n)
{
        int result;

        if (n == 1)
                result =  m;
        else
                result =  m + mult(m, n-1);
        return(result);
}

int main(void) {
   int x=2, y=3, result = 42;
   result = result + mult(x, y);
}
```

Name spaces

- Recursion and **name spaces**
  - Programming languages that allow recursion (such as C) have to ensure that **every separate invocation** of a function has its own copy of local variables

- In our example:
  - Function **`mult`** calls itself recursively
  - All recursive invocations have to have their own copy of **`result`**
  - Again, the compiler has to ensure this

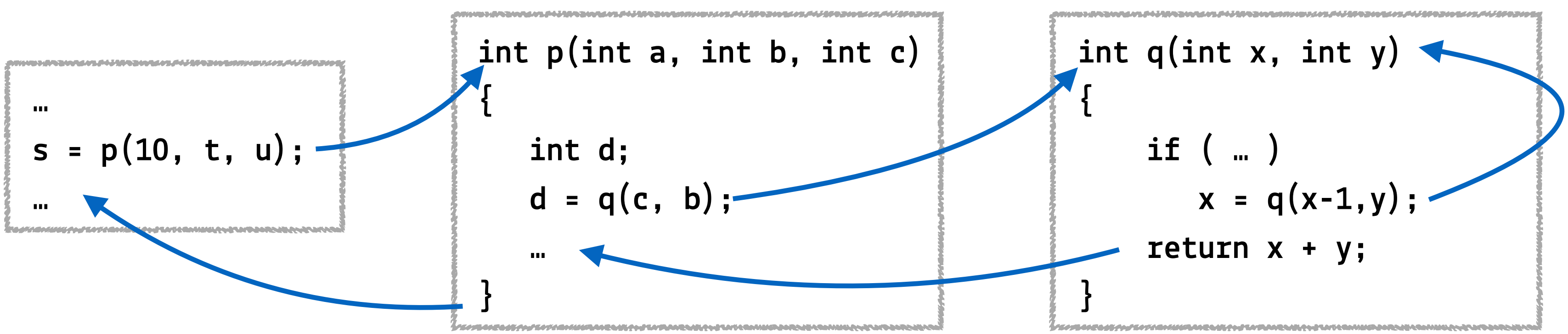
```
int mult(int m, int n)
{
        int result;

        if (n == 1)
                result =  m;
        else
                result =  m + mult(m, n-1);
        return(result);
}

int main(void) {
   int x=2, y=3, result = 42;
   result = result + mult(x, y);
}
```

Procedures and control flow

Procedures have well-defined control-flow

- Invoked at a call site, with some set of actual parameters
- Control returns to call site, immediately after invocation
  - A function can have multiple call sites
    ⇒ we need to remember where to return to!

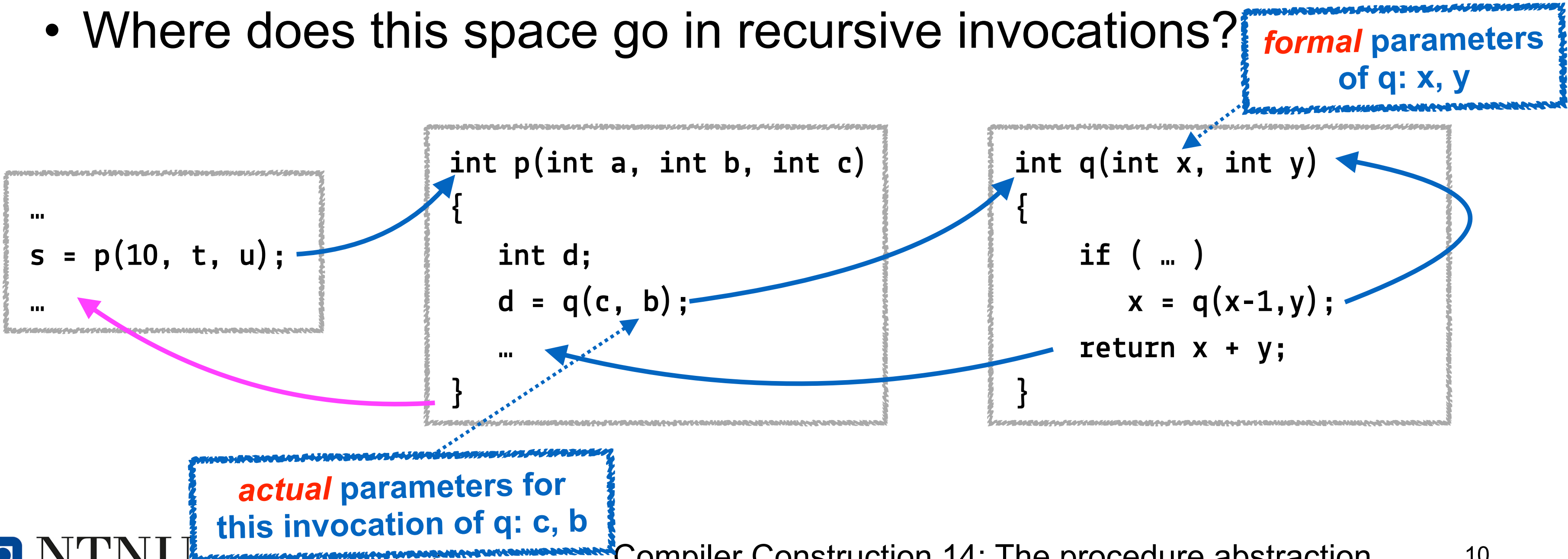
- Most languages allow recursion



Procedures and control flow

Implementing procedures with this behavior

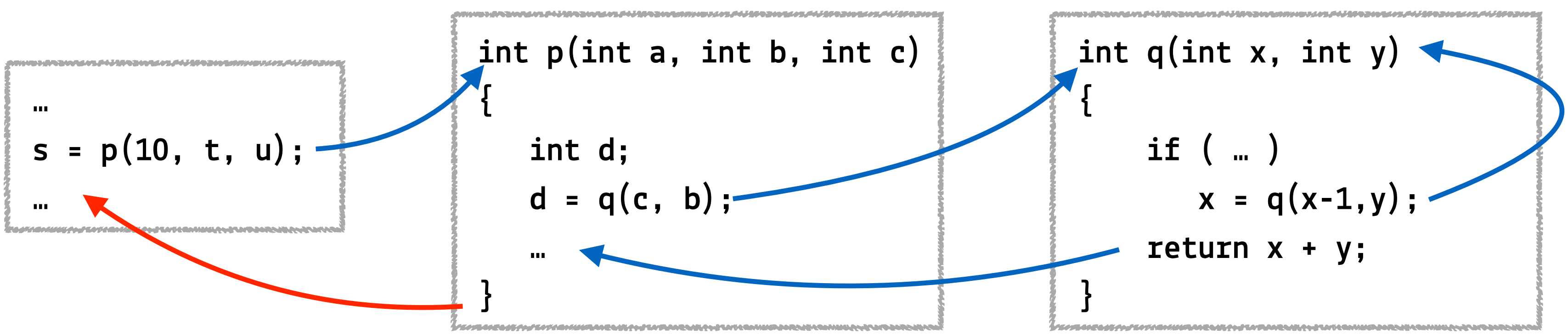
- Requires code to save and restore a "***return address***"
- Must map ***actual*** parameters to ***formal*** parameters (**c**→**x**, **b**→**y**)
- Must create storage for local variables (and maybe parameters)
- **p** needs space for variable **d** (and maybe also **a**, **b**, & **c**)
- Where does this space go in recursive invocations?



Procedures as control abstraction

Implementing procedures with this behavior

- Must preserve **p**'s state while **q** executes
- recursion causes the real problem here
- Strategy: Create unique location for each procedure activation
- Common to use a ***stack of memory blocks*** to hold local storage and return addresses



Compilers and procedures

Which tasks does a compiler perform to *implement* procedures?

- **Task at compile time**
  - Determine memory locations for each variable
  - Map each variable to its lexically correct scope
  - Ensure the mapping of actual to formal parameters
  - Generate code for function

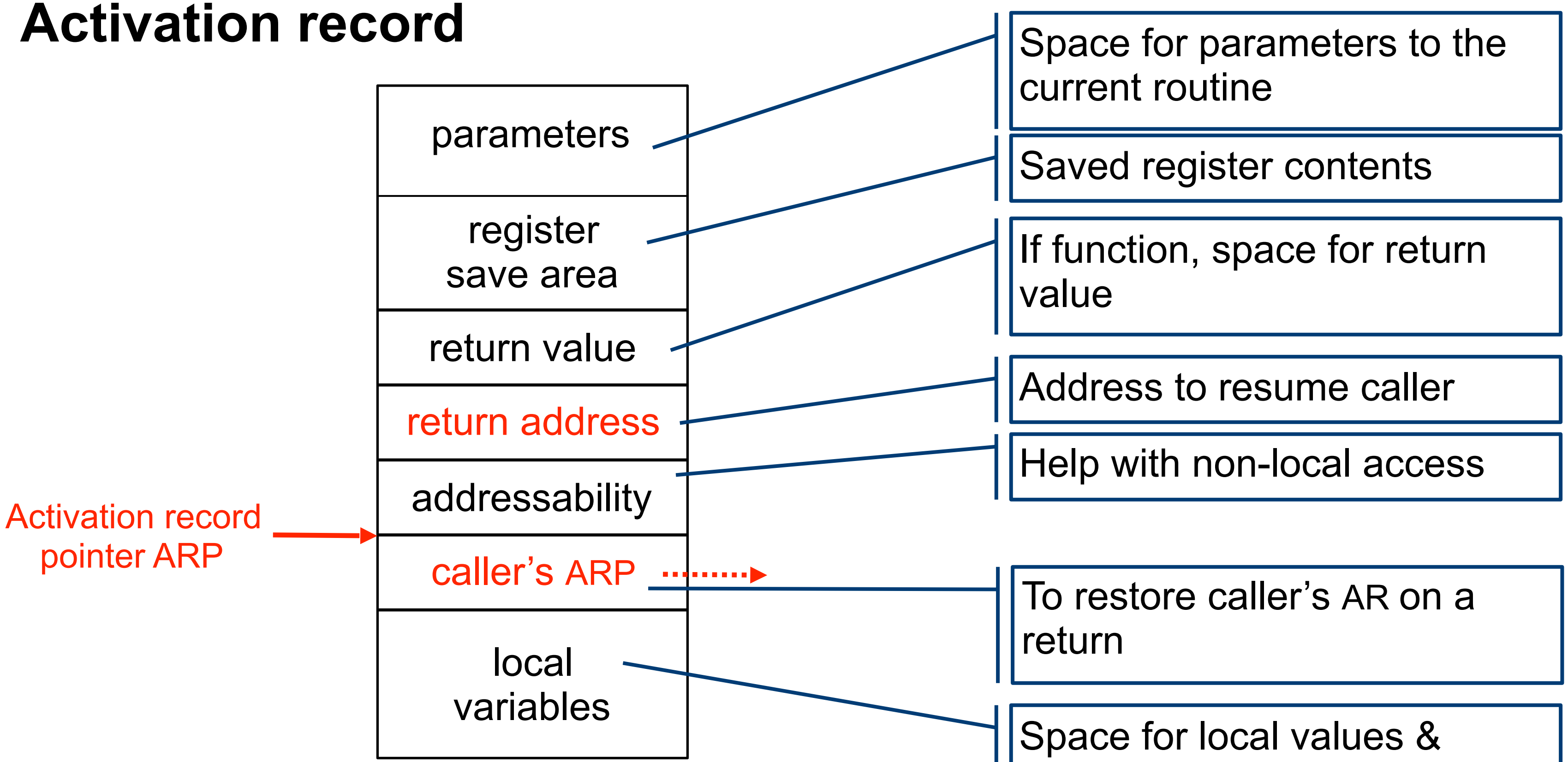
What happens when we *call* a procedure?

- **…at runtime** (code for this has been ***generated at compile time***)
  - Create space for storage of procedure-related data
  - Store the return address
  - Copy parameters into appropriate memory locations
  - Change control flow to procedure

Activation records

Where to store parameters, return address, local variables?

Activation record



One AR for each invocation of a procedure

Activation record details

How does the compiler find the variables?

- They are at known offsets from the AR pointer ARP
- This offset can be used in a special “load indexed" operation
- Level on stack specifies an ARP, offset is the constant

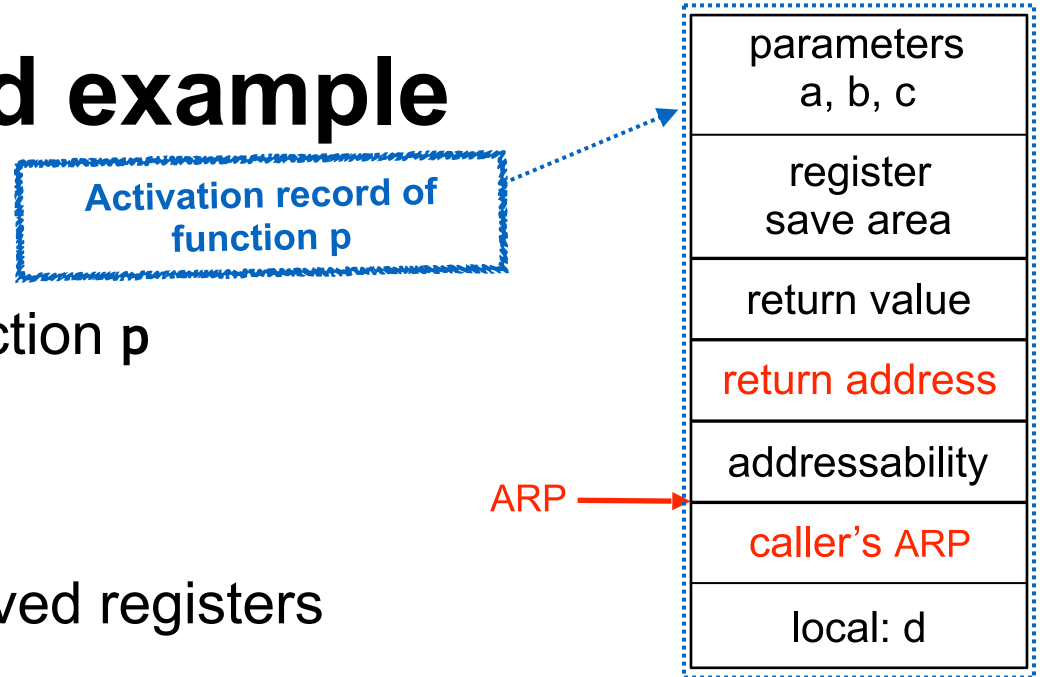
Variable-length data

- If AR can be extended, put it above local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

Initializing local variables

- Compiler must generate explicit code to store the values
- Among the procedure’s first actions

Activation record example



Execution has arrived at function **p**

- Local AR for **p** contains
  - Parameters **a, b, c**
  - Return address + saved registers
  - Space for return value
  - ARP of function that called **p**

```
int p(int a, int b, int c)
{
    int d;
    d = q(c, b);
    …
}
```

```
int q(int x, int y)
{
    if ( … )
        x = q(x-1,y);
    return x + y;
}
```

Activation record example

Activation record of function p

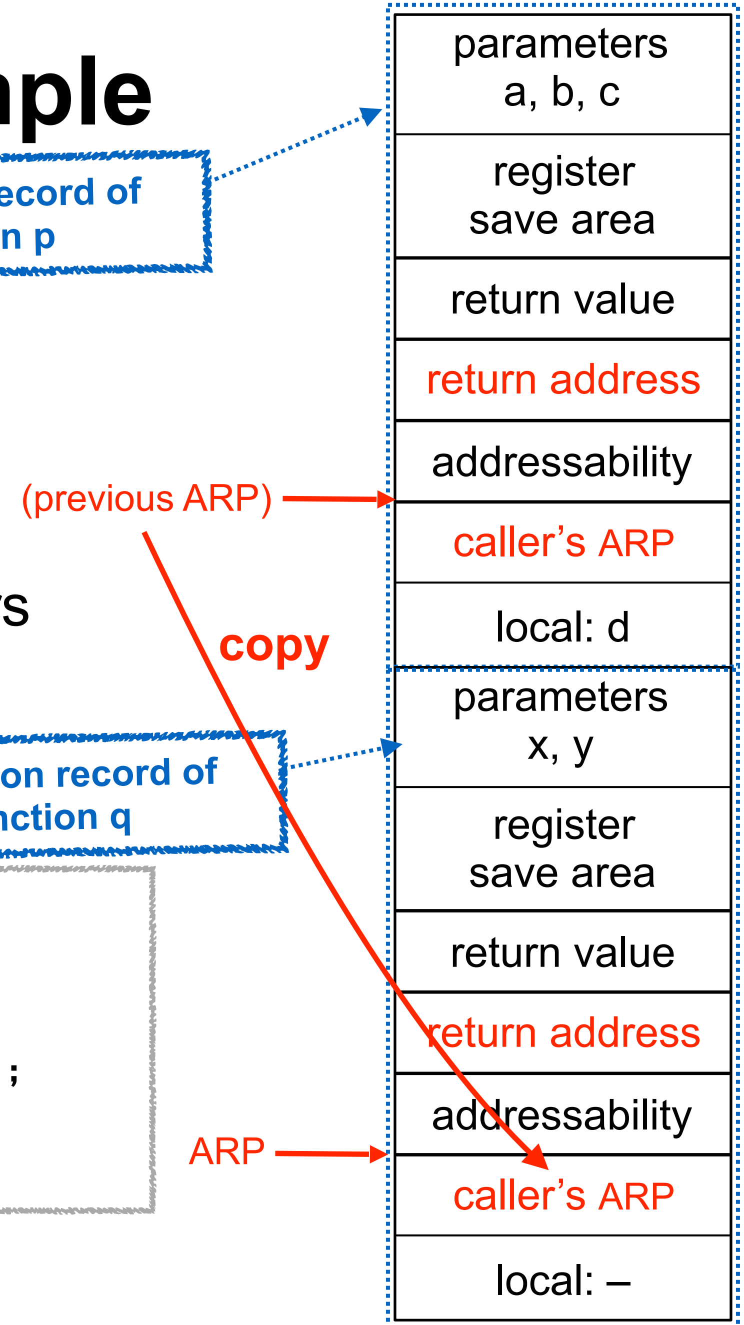
Execution has proceeded to **q**

- Local AR for **q** (below the one for **p**)
  - Parameters **x, y**
  - Return address + saved registers
  - Space for return value
  - ARP of **p**

Activation record of function q

```
int p(int a, int b, int c)
{
    int d;
    d = q(c, b);
    …
}
```

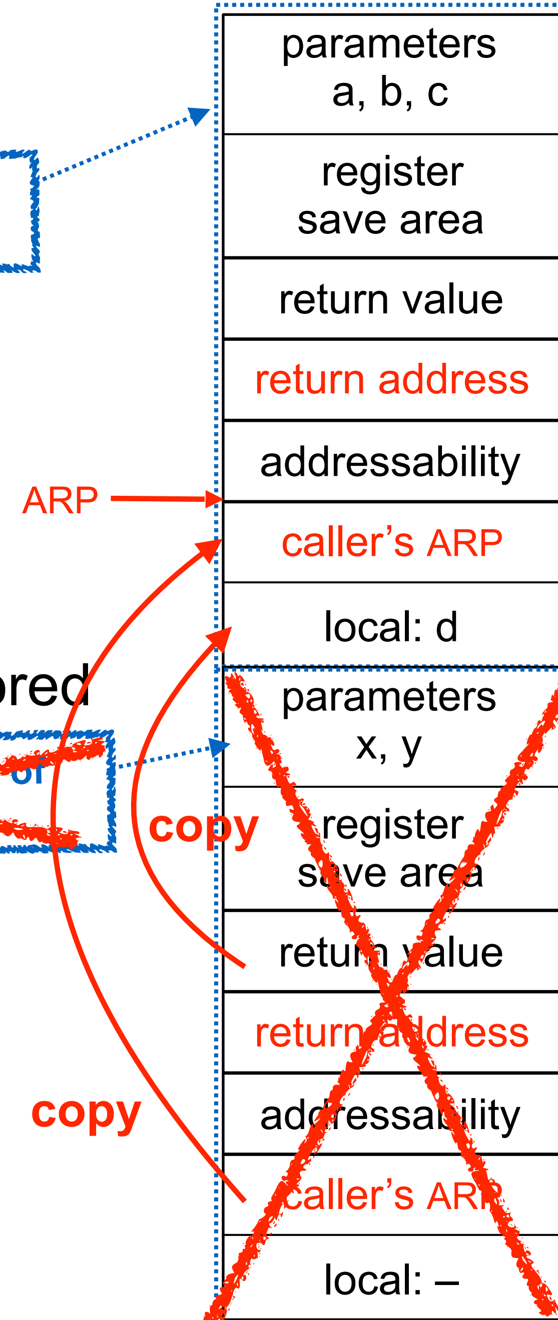
```
int q(int x, int y)
{
    if ( … )
        x = q(x-1,y);
    return x + y;
}
```



Activation record example

Execution has returned from **q**

- Return address used to return from **q**
- Return value (**x+y**) copied into **d** from **q** 's AR
- AR of **q** is invalidated, previous ARP restored
  - **q**'s AR stays in memory




```
int p(int a, int b, int c)
{
    int d;
    d = q(c, b);
    …
}
```

```
int q(int x, int y)
{
    if ( … )
        x = q(x-1,y);
    return x + y;
}
```

ARs and recursion?

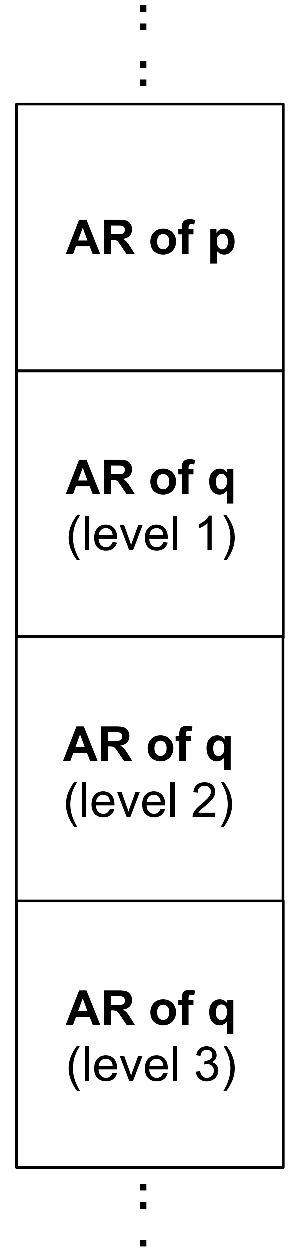
What happens when **q** recursively calls itself?

- The same as with every other function call
- Additional activation record for **q** is created on the stack
- and so on for each new level of recursion

- Too many recursion levels → ***stack overflow***

```
int p(int a, int b, int c)
{
   int d;
   d = q(c, b);
   …
}
```

```
int q(int x, int y)
{
   if ( … )
      x = q(x-1,y);
   return x + y;
}
```



What’s next?

- Intro to x86-64 assembly language
- Procedures in real life on x86-64

References

[1] Dijkstra, Edsger W. (March 1968). "Letters to the editor: Go to statement considered harmful". Communications of the ACM. 11 (3): 147–148. doi:10.1145/362929.362947