Variables, Functions and Allocation Strategies

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 6
24 January 2023

Modularity

- The principles of **abstraction** and **modularity** pervade computer science

Modularity

- The principles of **abstraction** and **modularity** pervade computer science
- Present at the lowest levels of computation

Modularity

- The principles of **abstraction** and **modularity** pervade computer science
- Present at the lowest levels of computation
- Modularity in the **von Neumann architecture**: CPU + addressable memory

Modularity

- The principles of **abstraction** and **modularity** pervade computer science
- Present at the lowest levels of computation
- Modularity in the **von Neumann architecture**: CPU + addressable memory
- Computation performed by CPU, inputs in expected places (**registers**)

Modularity

- The principles of **abstraction** and **modularity** pervade computer science
- Present at the lowest levels of computation
- Modularity in the **von Neumann architecture**: CPU + addressable memory
- Computation performed by CPU, inputs in expected places (**registers**)
- Data resides in memory, transferred to registers and back as needed

Modularity

- The principles of **abstraction** and **modularity** pervade computer science
- Present at the lowest levels of computation
- Modularity in the **von Neumann architecture**: CPU + addressable memory
- Computation performed by CPU, inputs in expected places (**registers**)
- Data resides in memory, transferred to registers and back as needed
- `x = y+z` ⇝

```
LOAD y
LOAD z
ADD
STORE x
```

Abstraction

- Basic unit of abstraction: **variables** and **function names**

Abstraction

- Basic unit of abstraction: **variables** and **function names**
- Complex expressions and **high-level control statements**

Abstraction

- Basic unit of abstraction: **variables** and **function names**
- Complex expressions and **high-level control statements**
- Variables represent data residing in a memory location

Abstraction

- Basic unit of abstraction: **variables** and **function names**
- Complex expressions and **high-level control statements**
- Variables represent data residing in a memory location
- We need a map from variables and names to addresses

Abstraction

- Basic unit of abstraction: **variables** and **function names**
- Complex expressions and **high-level control statements**
- Variables represent data residing in a memory location
- We need a map from variables and names to addresses
- Main job of a **compiler**

Static allocation

```
var x = 0;
var y = 1;
while (x < 100) {
    x = y;
    y = x+y;
}
print x;
```

Code segment

```
0000   OPCONST    1   '0'
0002   OPGLOBAL   0   'x'
0004   OPCONST    3   '1'
0006   OPGLOBAL   2   'y'
0010   OPCONST    5   '100'
              …
0038   OPRET
```

Data segment

```
0039   0×00…00        'x'
0040   0×00…00
0041   0×00…00        'y'
0042   0×00…01
0044   0×00…64
```

Indirect addressing using offsets from 0039

Suitable when all variables are **global** – no functions or blocks

Functions

- Makes programs **modular**

Functions

- Makes programs **modular**
- Complexities introduced by **recursion**

Functions

- Makes programs **modular**
- Complexities introduced by **recursion**
- Many versions of the same **local variable** active at the same time

Functions

- Makes programs **modular**
- Complexities introduced by **recursion**
- Many versions of the same **local variable** active at the same time
- Need a way to keep track of all copies of a local x

Functions

- Makes programs **modular**
- Complexities introduced by **recursion**
- Many versions of the same **local variable** active at the same time
- Need a way to keep track of all copies of a local x
- Figure out which copy of x is referred to at any point of the execution

Functions

- Makes programs **modular**
- Complexities introduced by **recursion**
- Many versions of the same **local variable** active at the same time
- Need a way to keep track of all copies of a local x
- Figure out which copy of x is referred to at any point of the execution
- Need to understand the **scope** and **lifetime** of a variable declaration

Scope

- Consider the following program block

```
{
    int x = 2;
    int y = 4;
    {
        int y = 3;
        x = x+2; y = x+y;
        print(x,y);
    }
        x = x+2; y = x+y;
        print(x,y);
}
```

```
Outer y is hidden.
Updated y value is not propagated outside
4, 7

Outer y value and updated x value
6, 10
```

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration
- Consider the example below

```
{ int x =  ... ;
    { int y =  ... ;
        { int x =  ... ;
            ...
        }
    }
}
```

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration
- Consider the example below

```
{ int x =  ... ;
    { int y =  ... ;
        { int x =  ... ;
              ...
        }
    }
}
```

- Scope of outer x is the two outer blocks

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration
- Consider the example below

```
{ int x =  ... ;
    { int y =  ... ;
        { int x =  ... ;
              ...
        }
    }
}
```

- Scope of outer x is the two outer blocks
- Scope of the inner x is the innermost block

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration
- Consider the example below

```
{ int x =  ... ;
    { int y =  ... ;
        { int x =  ... ;
              ...
        }
    }
}
```

- Scope of outer x is the two outer blocks
- Scope of the inner x is the innermost block
- Lifetime of inner x is the time during which innermost block is active

Scope and Lifetime

- **Scope** – A region of text in which a declaration is visible
- **Lifetime** – The duration, during a run of a program, in which a location is allocated as the result of a specific declaration
- Consider the example below

```
{ int x =  ... ;
    { int y =  ... ;
        { int x =  ... ;
              ...
        }
    }
}
```

- Scope of outer x is the two outer blocks
- Scope of the inner x is the innermost block
- Lifetime of inner x is the time during which innermost block is active
- Lifetime of outer x is the time during which outermost block is active (includes the lifetime of inner x)

static and lifetimes

- We use **static variables** to store class attributes rather than object attributes

```
class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        aCount += 1;
        this.id = id;
    }
}
```

static and lifetimes

- We use **static variables** to store class attributes rather than object attributes

```
class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        aCount += 1;
        this.id = id;
    }
}
```

- The **static** variable counts the number of instances of A that are created

`static` and lifetimes

- We use **static variables** to store class attributes rather than object attributes

```
class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        aCount += 1;
        this.id = id;
    }
}
```

- The **`static`** variable counts the number of instances of `A` that are created
- Lifetime spans the execution of the entire program

static and lifetimes

- We use **static variables** to store class attributes rather than object attributes

```
class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        aCount += 1;
        this.id = id;
    }
}
```

- The **static** variable counts the number of instances of A that are created
- Lifetime spans the execution of the entire program
- Scope is limited to the class A

Activation Record

- For local variables and function parameters, we need to store one copy for each **function invocation** (or **activation**)

Activation Record

- For local variables and function parameters, we need to store one copy for each **function invocation** (or **activation**)
- **Activation record** – collection of all data related to a function invocation

Activation Record

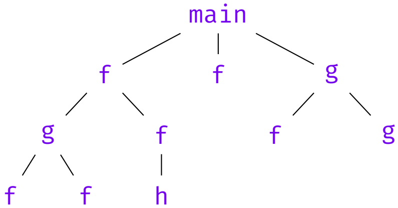
- For local variables and function parameters, we need to store one copy for each **function invocation** (or **activation**)
- **Activation record** – collection of all data related to a function invocation
- Includes space for local variables, parameters, intermediate results, and some pointers

Activation Record

- For local variables and function parameters, we need to store one copy for each **function invocation** (or **activation**)
- **Activation record** – collection of all data related to a function invocation
- Includes space for local variables, parameters, intermediate results, and some pointers
- Also called a **stack frame** – the reason will be clear later

Call graph

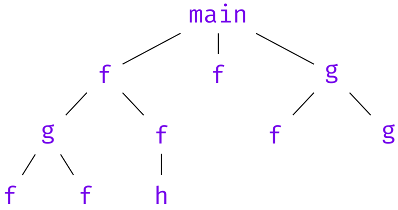
A **call graph** helps us visualize the function calls during a program execution



- The set of **active function calls** at any point of time lies is a path from root to some node in the graph

Call graph

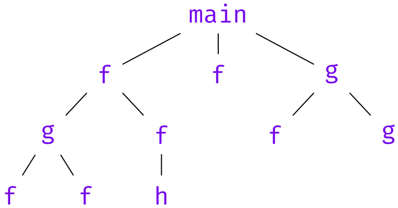
A **call graph** helps us visualize the function calls during a program execution



- The set of **active function calls** at any point of time lies is a path from root to some node in the graph
- If `f` calls `g`, then `g` is completed before `f`

Call graph

A **call graph** helps us visualize the function calls during a program execution



- The set of **active function calls** at any point of time lies is a path from root to some node in the graph
- If f calls g, then g is completed before f
- Store the activation records on a **stack**

Activation records on stack

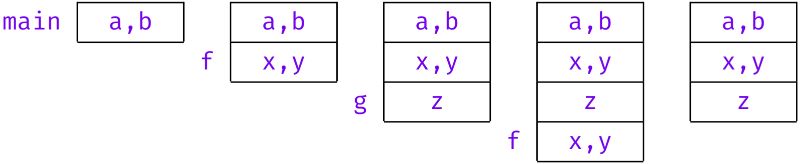
- We can place the activation records on a stack, which grows and shrinks as a program executes

Activation records on stack

- We can place the activation records on a stack, which grows and shrinks as a program executes
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`

Activation records on stack

- We can place the activation records on a stack, which grows and shrinks as a program executes
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- The stack evolves as follows:



main a,b

a,b
f x,y

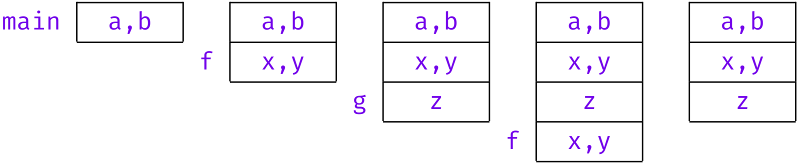
a,b
x,y
g z

a,b
x,y
z
f x,y

a,b
x,y
z

Activation records on stack

- We can place the activation records on a stack, which grows and shrinks as a program executes
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- The stack evolves as follows:



main a,b

a,b
f x,y

a,b
x,y
g z

a,b
x,y
z
f x,y

a,b
x,y
z

- More on activation records in the next lecture!