Storage allocation

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 8, 01 February 2024

Variables, functions and storage

- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses

Variables, functions and storage

- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses
- Functions represent blocks of (reusable) code
  - 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

Variables, functions and storage

- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses
- Functions represent blocks of (reusable) code
  - 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
- Scope and lifetime of variables

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);
}
```

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);
}
```

hides

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

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 – Region of text in which a declaration is visible

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below

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

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for 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 – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for 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 – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for 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 – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for 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 variables

- Recall that static variables are associated with a class as a whole
- Do not require instantiation of objects

static variables

- Recall that static variables are associated with a class as a whole
- Do not require instantiation of objects

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

static variables

- Recall that static variables are associated with a class as a whole
- Do not require instantiation of objects

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

- The `static` variable `howManyAs` counts the number of instances of `A` created

static variables

- Recall that static variables are associated with a class as a whole
- Do not require instantiation of objects

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

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

`static` variables

- Recall that static variables are associated with a class as a whole
- Do not require instantiation of objects

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

- The `static` variable `howManyAs` counts the number of instances of `A` created
- Lifetime of `howManyAs` spans the execution of the entire program
- Scope of `howManyAs` 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

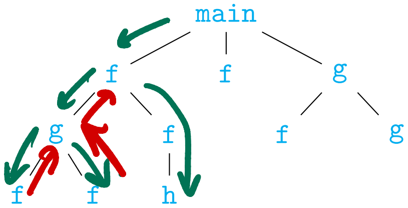
- For local variables and function parameters, we need to store one copy for each function invocation (or activation)
- Activation record — collection of all data related to a function invocation
- Includes space for local variables, parameters, intermediate results, and some pointers
- Also called a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

Call graph

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

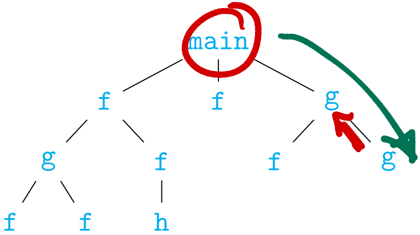
Call graph

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



Call graph

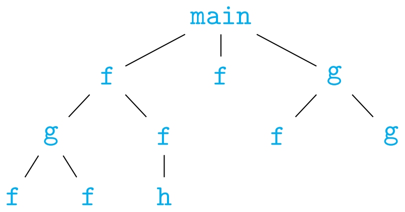
- A call graph helps us visualize the function calls during a program execution



- The set of active function calls at any point of time lies on the path from the root to the right most leaf

Call graph

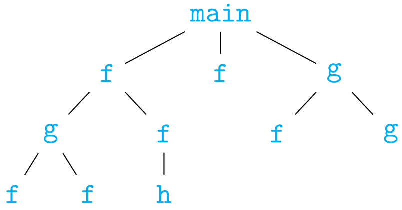
- A call graph helps us visualize the function calls during a program execution



- The set of active function calls at any point of time lies on the path from the root to the right most leaf
- If `f` calls `g`, then `g` is completed before `f`

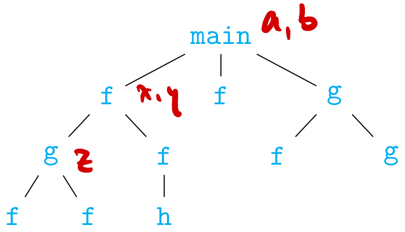
Call graph

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



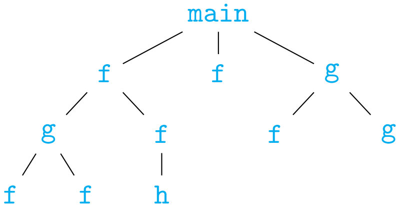
- The set of active function calls at any point of time lies on the path from the root to the right most leaf
- If `f` calls `g`, then `g` is completed before `f`
- Store the activation records on a stack

Activation records on stack



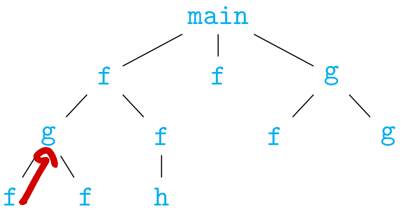
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`

Activation records on stack



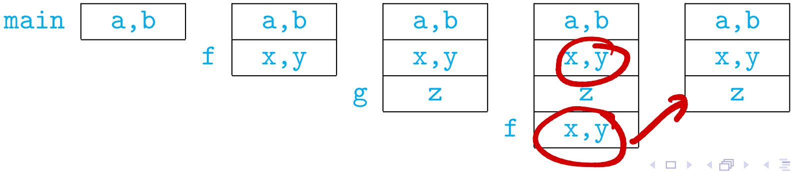
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- Place activation records on a stack — grows and shrinks as a program executes

Activation records on stack

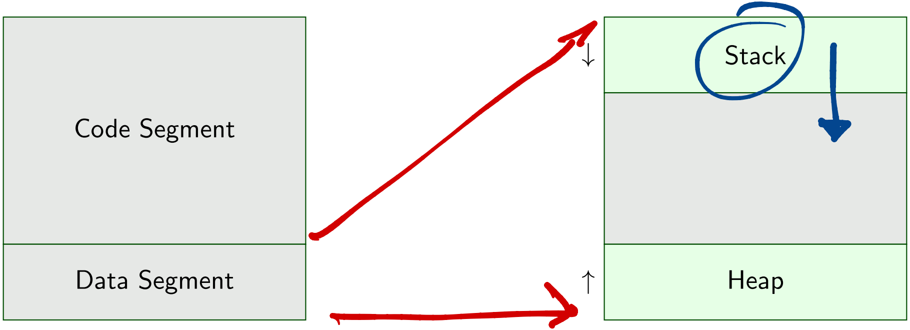


- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- Place activation records on a stack — grows and shrinks as a program executes

- The stack evolves as follows:



General layout of a program in memory



Activation record

- Contains information pertaining to a function invocation
  - Added to the top of the stack at the start of the function invocation
  - Removed from the stack at the end of the function invocation

Activation record

- Contains information pertaining to a function invocation
  - Added to the top of the stack at the start of the function invocation
  - Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function

Activation record

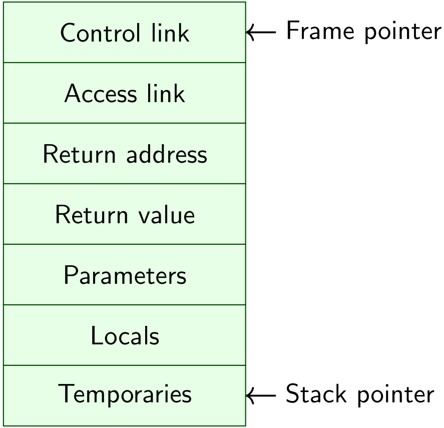
- Contains information pertaining to a function invocation
  - Added to the top of the stack at the start of the function invocation
  - Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers — Control link, access link, return address

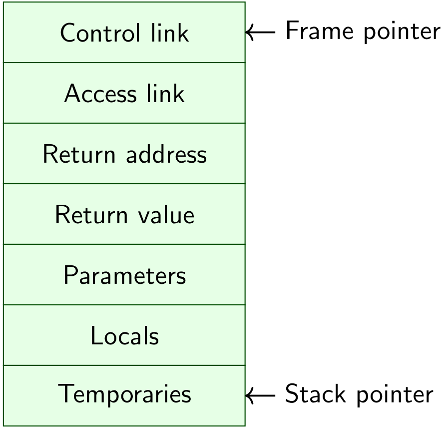
Activation record

- Contains information pertaining to a function invocation
  - Added to the top of the stack at the start of the function invocation
  - Removed from the stack at the end of the function invocation
- Stores parameters, local variables, temporary variables used in running the function
- Various pointers — Control link, access link, return address
- System-wide pointers
  - Program counter — address of the next instruction to execute
  - Stack pointer — points to the top of the system stack
  - Frame pointer — points to the start of the topmost frame on stack
  - Data in topmost frame accessed via offsets from the frame pointer or stack pointer — offsets can computed at compile time

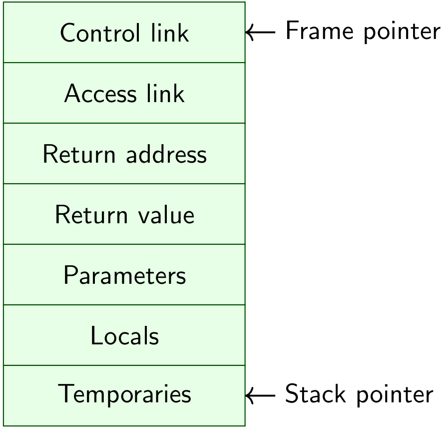
relative addressing

Activation record . . .



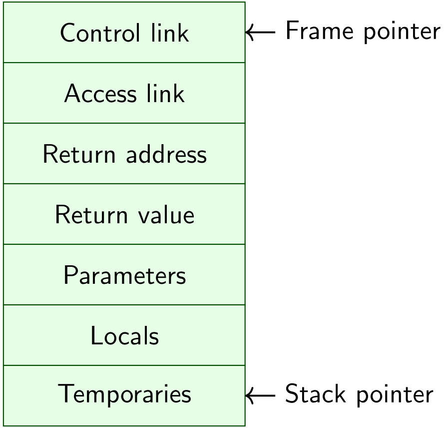


- Control link points to activation record of caller



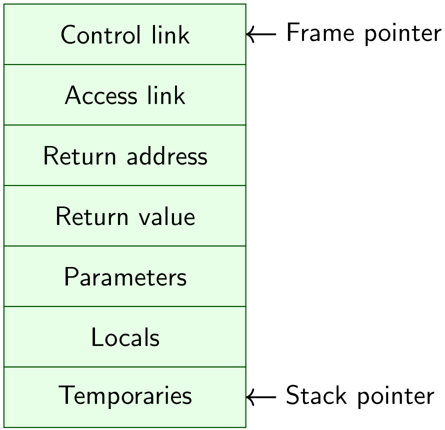
- Control link points to activation record of caller
- Access link is for non-local variable access

Activation record . . .



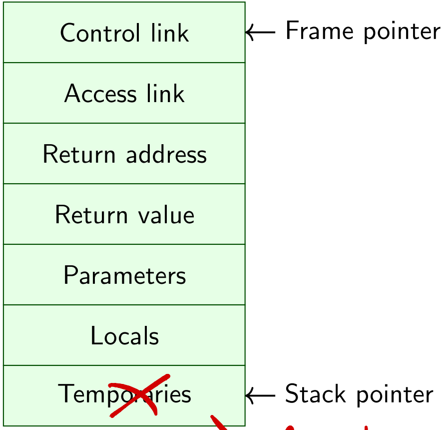
- Control link points to activation record of caller
- Access link is for non-local variable access
- Return address is the address of first instruction to execute after the function call returns

Activation record . . .



- Control link points to activation record of caller
- Access link is for non-local variable access
- Return address is the address of first instruction to execute after the function call returns
- Return value stores the return value, which should be picked up by the caller

Activation record ...



Compiler creates to avoid recomputing

- Control link points to activation record of caller
- Access link is for non-local variable access
- Return address is the address of first instruction to execute after the function call returns
- Return value stores the return value, which should be picked up by the caller
- Temporaries are locations to store intermediate values in

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n <= 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n <= 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local

Access links

```
func f {
    int x = 0;
    int fib(int n) {
        if n <= 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

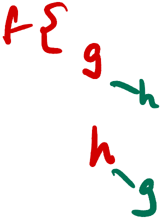
- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` can be accessed by following the control link

Access links

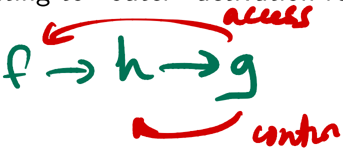
```
func f {
    int x = 0;
    int fib(int n) {
        if n <= 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```

- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` ~~can~~ could be accessed by following the control link
- But `fib(3)` is called by `fib(4)`, so control link cannot be used to access `x`

```
func f {
    int x = 0;
    int fib(int n) {
        if n <= 1 then return n;
        else {
            x += 1;
            return fib(n-1) + fib(n-2);
        }
    }
    print(fib(4));
}
```



- Count the number of additions in `fib(4)`
- `x` is non-local
- `fib(4)` is called by `f`, so `x` can be accessed by following the control link
- But `fib(3)` is called by `fib(4)`, so control link cannot be used to access `x`
- Need a new kind of link — access link pointing to “outer” activation record



Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x; ...
    }
    public int f(int n) {
        int arr[n]; ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100); ...
}
```

- Functions can handle complex data types – arrays / classes, ...

Dynamic allocation

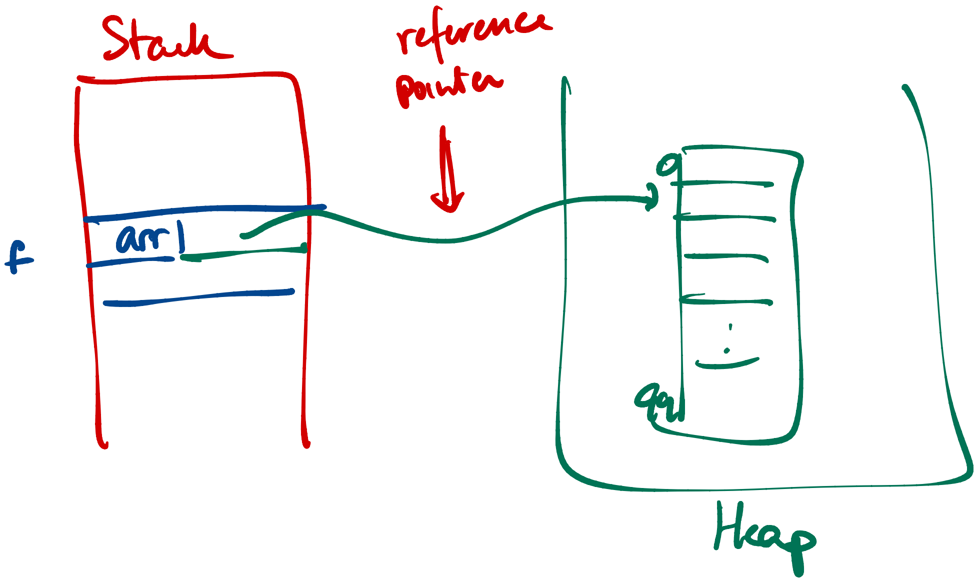
```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x; ...
    }
    public int f(int n) {
        int arr[n]; ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100); ...
}
```

- Functions can handle complex data types – arrays / classes, ...
- Dynamic data structures like linked lists / graphs
  - No pre-specified bound on the number of elements

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x; ...
    }
    public int f(int n) {
        int arr[n]; ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100); ...
}
```

- Functions can handle complex data types – arrays / classes, . . .
- Dynamic data structures like linked lists / graphs
  - No pre-specified bound on the number of elements
- The activation record for `main` will store a pointer (or reference) to the object `aObj` stored on the heap!


Stack
reference
pointer
f
arr1
0
99
Heap

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x; ...
    }
    public int f(int n) {
        int arr[n]; ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100); ...
}
```

- Functions can handle complex data types – arrays / classes, ...
- Dynamic data structures like linked lists / graphs
  - No pre-specified bound on the number of elements
- The activation record for `main` will store a pointer (or reference) to the object `aObj` stored on the heap!
- `aObj` itself has pointers to the class definition

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x; ...
    }
    public int f(int n) {
        int arr[n]; ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100); ...
}
```

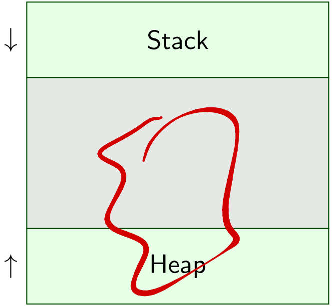
- Functions can handle complex data types – arrays / classes, ...
- Dynamic data structures like linked lists / graphs
  - No pre-specified bound on the number of elements
- The activation record for `main` will store a pointer (or reference) to the object `aObj` stored on the heap!
- `aObj` itself has pointers to the class definition
- The AR for `f` has a pointer to an array stored on heap

Heap

- Heap — just a chunk of memory
  - Unstructured
  - Nothing to do with the heap data structure used to implement priority queues!

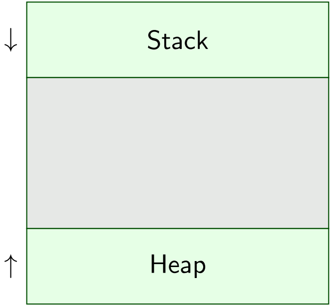
Heap

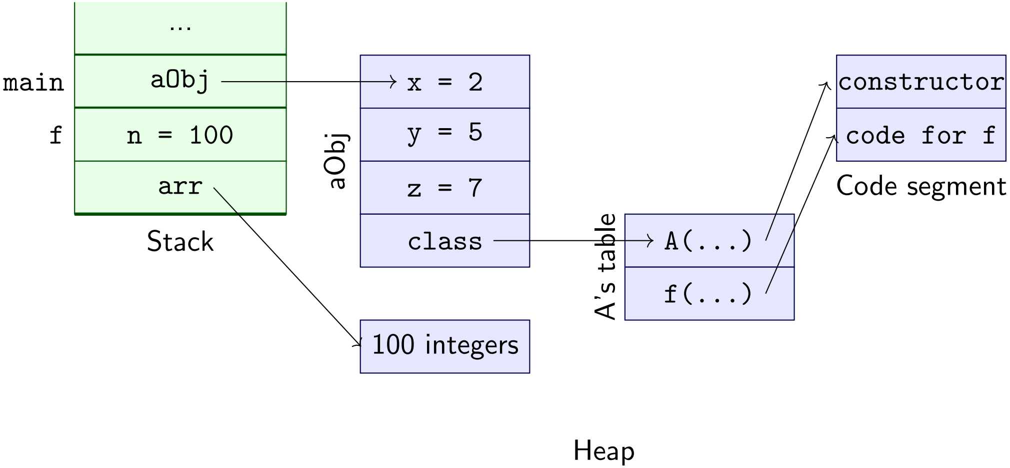
- Heap — just a chunk of memory
  - Unstructured
  - Nothing to do with the heap data structure used to implement priority queues!
- Typically depicted as "growing upward" (and the stack grows downward)



Heap

- Heap — just a chunk of memory
  - Unstructured
  - Nothing to do with the heap data structure used to implement priority queues!
- Typically depicted as "growing upward" (and the stack grows downward)
- Consist of chunks of allocated and unallocated memory




...
main
aObj
f
n = 100
arr
Stack
aObj
x = 2
y = 5
z = 7
class
100 integers
A's table
A(...)
f(...)
constructor
code for f
Code segment
Heap

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

- Runtime polymorphism has a simple implementation

Overriding, inheritance etc.

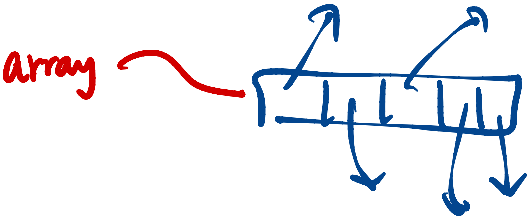
- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

- Runtime polymorphism has a simple implementation
- Consider an array of `Shape`, each element being an instance of a subclass

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

- Runtime polymorphism has a simple implementation
- Consider an array of `Shape`, each element being an instance of a subclass
- Elements of the array are pointers to objects



Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

- Runtime polymorphism has a simple implementation
- Consider an array of `Shape`, each element being an instance of a subclass
- Elements of the array are pointers to objects
- The object data has a pointer to the precise subclass it is an instance of!

Overriding, inheritance etc.

- Table for each class has a pointer to table for superclass
- Overloaded function
  - Find its pointer in the table for the class
  - Otherwise look at parent's table
- Might need to follow a chain of pointers to determine the code to run on a method call

- Runtime polymorphism has a simple implementation
- Consider an array of `Shape`, each element being an instance of a subclass
- Elements of the array are pointers to objects
- The object data has a pointer to the precise subclass it is an instance of!
- Calling `perimeter` on each element of the array runs the code pointed to by the appropriate subclass table

Heaps and memory management

- As functions are called, they allocate data on the heap

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be inaccessible from stack!

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be inaccessible from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be inaccessible from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap
- Allocated data might no longer have a reference from the stack (direct or indirect)

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be inaccessible from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap
- Allocated data might no longer have a reference from the stack (direct or indirect)
- This is called garbage – waste of memory

Explicit memory management

- Older languages expect programmer to manage memory
- `malloc / free` in C, `new / delete` in C++

Memory leak

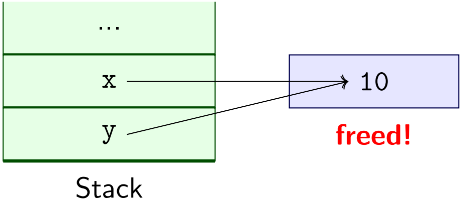
Explicit memory management

- Older languages expect programmer to manage memory
- `malloc / free` in C, `new / delete` in C++
- `free / delete` tells the system to take back ownership of memory locations from the program – deallocation

Explicit memory management

- Older languages expect programmer to manage memory
- `malloc / free` in C, `new / delete` in C++
- `free / delete` tells the system to take back ownership of memory locations from the program – deallocation
- Can cause the problem of dangling pointers – pointers to deallocated variables

```
int *x = malloc(sizeof(int));
*x = 10;
y = x;
free(x);
```



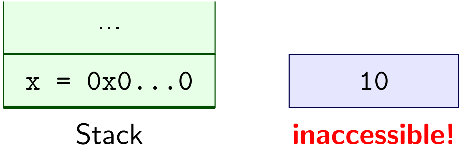
Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results
- Huge security risk!
- Garbage is not so serious, but wastes resources!
- Can happen even with explicit deallocation

Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results
- Huge security risk!
- Garbage is not so serious, but wastes resources!
- Can happen even with explicit deallocation

```
int *x = malloc(sizeof(int));
*x = 10;
x = NULL;
```



x = y for references
- copies address