# Foundations of Programming Languages

Variables

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

22. Oktober 2014

## Variables and Bindings

Variables have the following bindings:

**Name**

Identifier

**Scope**

Identifier visibility

**Type**

What kinds of things can be stored?

**Value**

What is currently stored?

**Lifetime**

When and how allocated?
When and how deallocated?

**Address**

Where in memory is it stored?
How can it be accessed?

**Access Rights**

Who has permissions to do what with it?

## Access Rights

Languages permit restrictions to operations on variables

### Access Rights

```
{
  const int x = 1;
  ...
  x = 2;
}
```

*Disallowed*: **const** removes write permissions from x

### Visibility

```
{
  {
    int x = 1;
  }
  ...
  x = 2;
}
```

*Error*: x not visible in assignment

- Forms of access rights:
  read, write, call, instantiate, get-address-off, ...

**Access Rights $\neq$ Visibility**

## Storage and Lifetime

- Each variable is encoded in memory
  $\Rightarrow$ must be *allocated*, *de-allocated*

$$\begin{array}{l}\quad\quad\quad\;\textbf{allocation}\\ \text{lifetime}\left\{\begin{array}{l}\\ \text{variable use}\\ \\ \end{array}\right.\\ \quad\quad\quad\;\textbf{deallocation}\end{array}$$

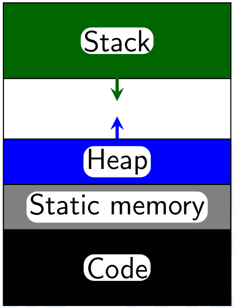
$\boxed{\textbf{Variable lifetime: period between allocation, deallocation}}$

## Static Variables

- **Location**: Static memory
- **Allocation**: Compile-time
- **Deallocation**: Never
- **Lifetime**: Entire run-time
- **Address**: Relative to $gp
- **Example**:

C

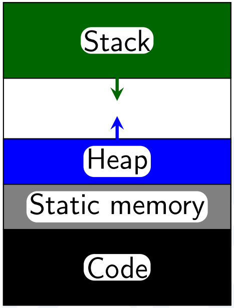
```c
int next() {
    static int count = 0;
    count = count + 1;
    return count;
}
```



**Global variables are often implemented as static variables**

## Stack-Dynamic Variables

- **Location**: Stack
- **Allocation**: Enter scope
- **Deallocation**: Leave scope
- **Lifetime**: Execution of block
- **Address**: Relative to `$fp` or `$sp`
- **Examples**:
  - Local variables
  - Parameters
  - Temporary variables

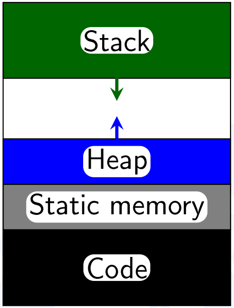


## Heap-Dynamic Variables

- **Location**: Heap memory
- **Allocation**: Explicit or implicit
- **Deallocation**: Explicit or garbage collector
- **Lifetime**: Custom
- **Address**: Anywhere on the heap
- **Example**:

C++

```cpp
string* s = new string();
...
delete s;
```



## Explicit Heap-Dynamic Variables

```
// Java
String s = new String("foo");
String s2 = s;
```

- Heap-dynamic variable has no name
- Variables s, s2 both *reference* or *point to* anonymous variable
  ⇒ s, s2 are *reference variables*

| **Lang.** | **allocate** | **dealloc** |
|---|---|---|
| C | `malloc` | `free` |
| C++ | **new** | **delete** |
| Java | **new** | (implicit) |
| C# | **new** | (implicit) |

**Heap variables are anonymous**

## Implicit Heap-Dynamic Variables

### Python

```python
def f(x):
    return [1, x, 2]
```

- Return value to `f` allocated on heap *implicitly*
- Deallocation implicit: Python uses automatic heap memory management

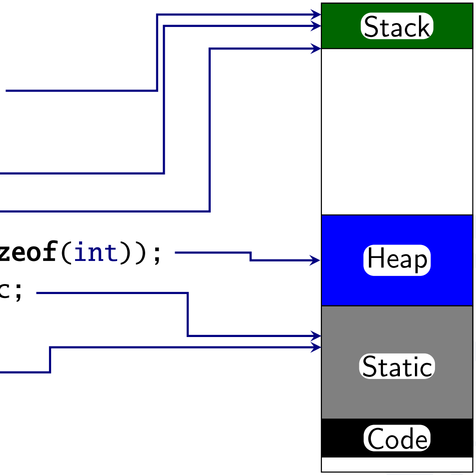
**Return value is again a nameless variable**

## References and Pointers

- *Reference variables*:
  Variables that point to either:
  - some other variable
  - special ‘nothing’ marker (`null`, `nil`, `None`, `NULL`, . . . )
- *Pointer variables*:
  Variables that contain an *arbitrary* memory address
  - May point anywhere in memory
  - Dangerous when used incorrectly (hard-to-find bugs)
  - Vital to systems programming
  - Only in very few languages: Assembly, C, C++, Modula-3, . . .

## Example in C

```
void f(int x)
{
  int y;
  int *a =
    malloc(sizeof(int));
  static int c;
}
int x;
```



## Summary

- Variables have up to 7 bindings:
  - *name* and *scope*: who can refer to them where?
  - *type* and *value*: what can they store, what do they store?
  - *lifetime*: when allocated, when deallocated?
  - *address*: what register+offset tells me how and where to read/write?
  - *access rights*: who may do what to the variable?
- Three storage strategies:
  - *Static*: fixed-size block
  - *Stack-dynamic*: dynamic FILO memory
  - *Heap-dynamic*: dynamic free-form memory
  - Beware: some programs use multiple stacks/heaps/static segments