

Extent and closure

- **Extent of a variable**

- We have talked about the following properties of a variable:
 - Address (lvalue)
 - Value (rvalue)
 - Name: String used to refer to the variable
 - Scope: Range of code in which the name is bound to the variable
- Extent: Lifetime of a variable
- Example (C):

```
int foo(void)
{
    int i;
    i = 42;
    bar();
    return i;
}
```

- When does the variable `i` come into being? When `foo()` starts to execute.
- When does it stop to exist? When `foo()` returns.
- The fact that the value of `i` is returned doesn't matter because return is by value: The value of `i` is copied to another variable (in the caller).
- The variable `i` is a **stack-dynamic** variable and has **dynamic extent**.
- Example:

```
int *foo(void)
{
    int *a;
    a = malloc(sizeof(int));
    if (!a) { fprintf(stderr, "Failed to allocate memory.
\n"); exit(1); }
    return a;
}
```

- The variable `a` is a pointer to another (unnamed) variable. The latter is of type `int`.
- The unnamed variable comes into being when we call `malloc`.
- After `foo` returns, it's still there.
 - In fact, we return a pointer to it so the caller can continue to use it.
- The unnamed variable is an **explicit heap-dynamic** variable and has **infinite extent**.
- The pointer variable `a` has dynamic extent, as before.
- Example:

```
int g;

void foo (void)
{
    g = 42;
}
```

- The variable `g` always exists.
- Storage allocated for `g` at compile time. (At least conceptually.)
- The variable `g` has **static extent**.
- **Closures**
 - Recall that some languages allows us to define and return functions.
 - Example (Lisp):


```
(defun printer (f)
  (princ (funcall f 3)))

(defun foo (a)
  (let ((f (lambda (b) (+ a b))))
    (printer f)
    (incf a)
    (printer f)))
```

 - `(foo 7)` prints 10 and then 11
 - The anonymous function refers to `a`.
 - When the function is called (within `printer`), it needs to be able to resolve `a`.
 - What is passed to `printer` is not just the code of the function, but also a pointer to the activation record in which it was defined. Thus the binding of `a` is preserved.
 - Example (Lisp):


```
(defun make-adder (a)
  (lambda (b) (+ a b)))

(mapcar (make-adder 3) '(1 3 7)) => '(4 6 10)
```

 - The parameter `a` should have dynamic extent.
 - The function defined by `(lambda (b) (+ a b))` has infinite extent, and is returned.
 - But the function uses the name `a`.
 - Thus, `a` must be given infinite extent as well.
 - Typically done by "lifting" the activation record of `make-adder` from the stack to the heap.