9 Variables and lifetime

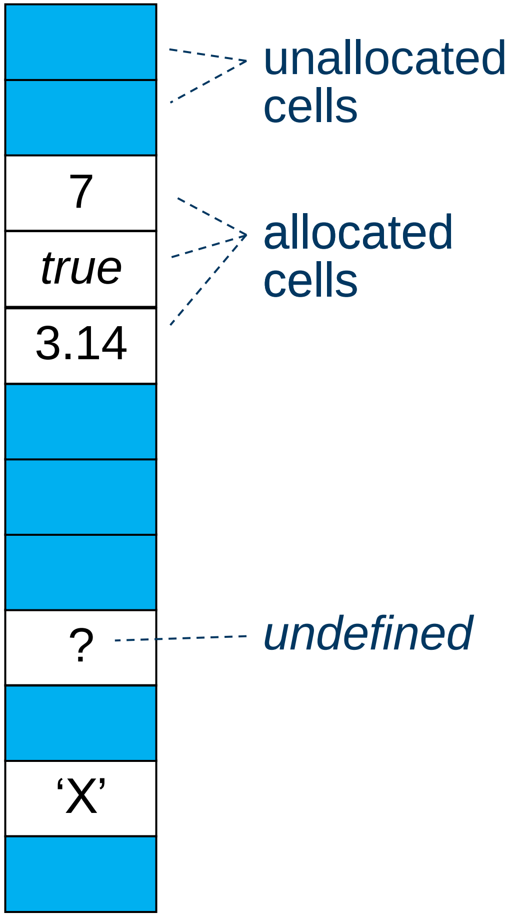
- Variables and storage
- Simple *vs* composite variables
- Lifetime: global, local, heap variables
- Pointers
- Commands
- Expressions with side effects

What are variables?

- In functional PLs (and in mathematics), a **variable** stands for a fixed (but possibly unknown) value.
- In imperative and OO PLs, a **variable** *contains* a value. The variable may be *inspected* and *updated* as often as desired.
  - Such a variable can be used to model a real-world object whose state changes over time.

Storage

- To understand such variables, assume an abstract storage model:
  – A **store** is a collection of **cells**, each of which has a unique **address**.
  – Each cell is either *allocated* or *unallocated*.
  – Each allocated cell contains either a simple value or *undefined*.
  – An allocated cell can be **inspected**, unless it contains *undefined*.
  – An allocated cell can be **updated** at any time.



Simple *vs* composite variables

- A **simple variable** is one that contains a primitive value or pointer.
  - A simple variable occupies a single allocated cell.
- A **composite variable** is one that contains a composite value.
  - A composite variable occupies a group of adjacent allocated cells.

Composite variables

- A variable of a composite type has the same structure as a value of the same type. For instance:
  - A tuple variable is a tuple of component variables.
  - An array variable is a mapping from an index range to a group of component variables.
- Depending on the PL, a composite variable can be:
  - **totally updated** (all at once), and/or
  - **selectively updated** (one component at a time).

Example: C composite variables

- Declaration and updating of struct variables:

```
struct Date {int y, m, d;}
struct Date xmas, today;

xmas.d = 25;
xmas.m = 12;
xmas.y = 2008;

today = xmas;
```

selective updating (xmas.d = 25; xmas.y = 2008;)

total updating (today = xmas;)

- Declaration and updating of array variable:

```
float a[10];

a[i] = 3.1417;
```

selective updating (a[i] = 3.1417;)

Lifetimes

- Every variable is **created** at some definite time, and **destroyed** at some later time when it is no longer needed.
- A variable's **lifetime** is the interval between its creation and destruction.
- A variable occupies cells only during its lifetime. When the variable is destroyed, these cells may be deallocated.
  - And these cells may subsequently be re-allocated to other variable(s).

Global *vs* local *vs* heap variables

- A **global variable**'s lifetime is the program's entire run-time. It is created by a global declaration.
- A **local variable**'s lifetime is an activation of a block. It is created by a declaration within that block, and destroyed on exit from that block.
- A **heap variable**'s lifetime is arbitrary, but bounded by the program's run-time. It can be created at any time (by an **allocator**), and may be destroyed at any later time. It is accessed through a pointer.

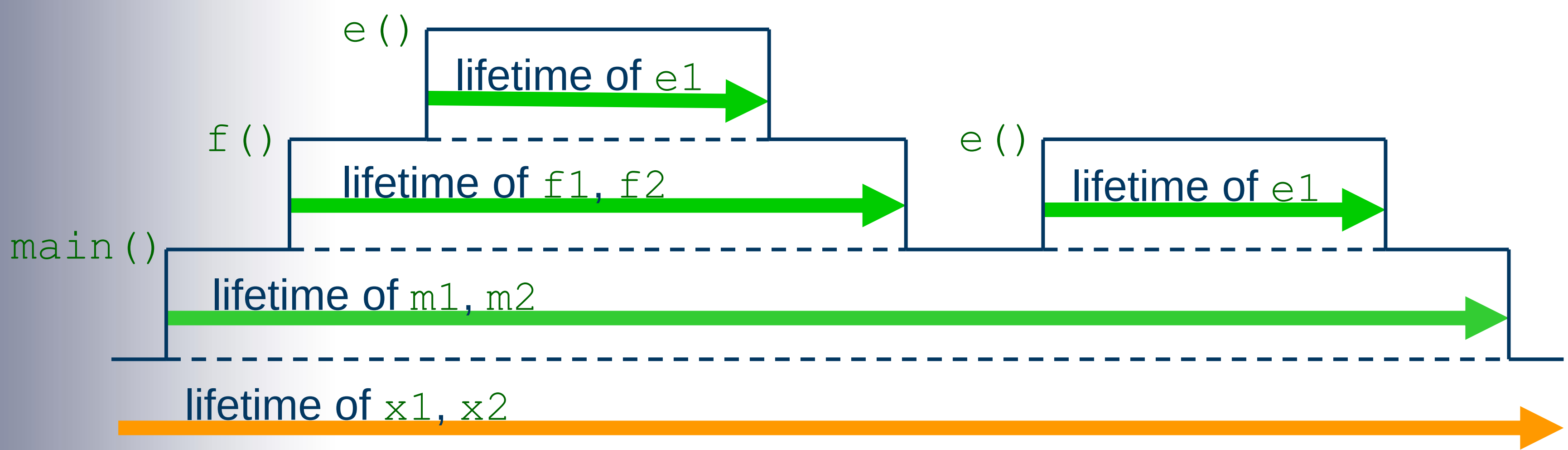
Example: C global and local variables *(1)*

- Outline of C program:

```
extern int x1, x2;                  global variables

void main () {
    int m1; float m2;               local variables
    … f(); … e(); …
}

void f () {
    float f1; int f2;               local variables
    … e(); …
}

void e () {
    int e1;                         local variable
    …
}
```

Example: C global and local variables *(2)*

- Lifetimes of global and local variables:



- Global and local variables' lifetimes are nested. They can never be overlapped.

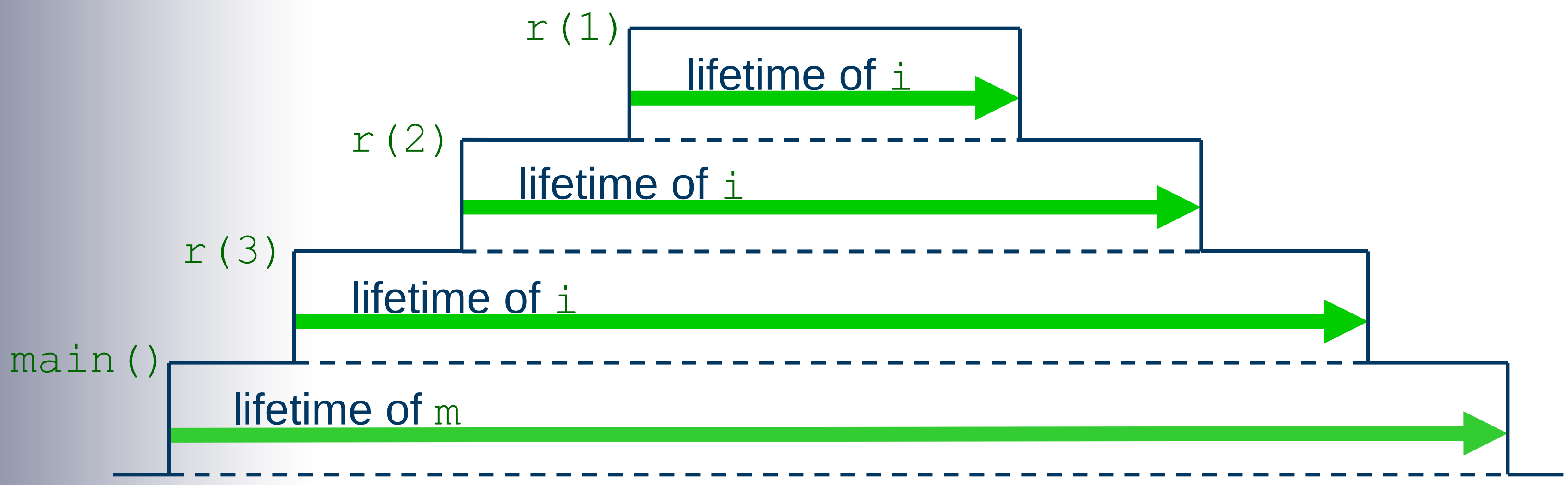
Example: local variables and recursion *(1)*

- Outline of C program:

```
void main () {
    float m;
    … r(3); …
}

void r (int n) {
    int i;
    if (n > 1) {
        … r(n-1); …
    }
}
```

Example: local variables and recursion *(2)*

- Lifetimes of global and local variables:



- *Note:* A local variable of a recursive procedure/function has several nested lifetimes.

Example: C heap variables *(1)*

- Outline of C program:

```
struct IntNode {int elem; IntList succ;}
typedef struct IntNode * IntList;

IntList c (int h, IntList t) {
    // Return an IntList with head h and tail t.
    IntList ns =
        (IntList) malloc (sizeof IntNode);
    ns->elem = h; ns->succ = t;
    return ns;
}

void d (IntList ns) {
    ns->succ = ns->succ->succ;
}
```

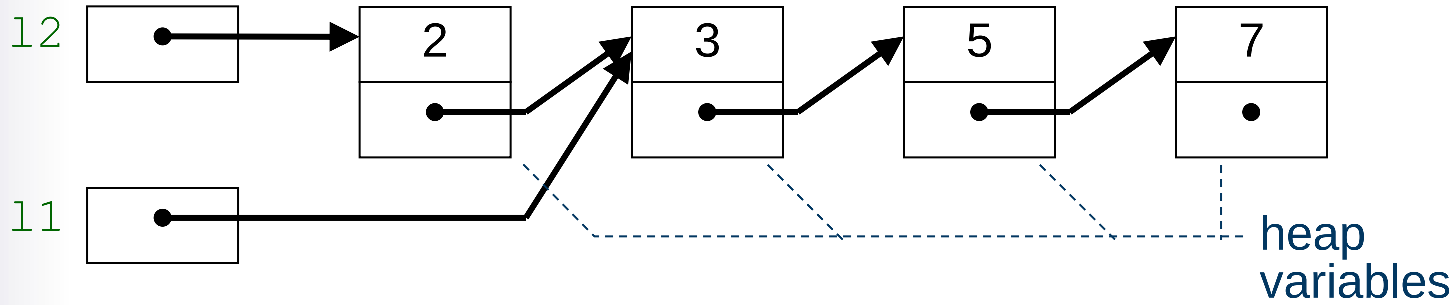
Example: C heap variables *(2)*

- Outline *(continued)*:

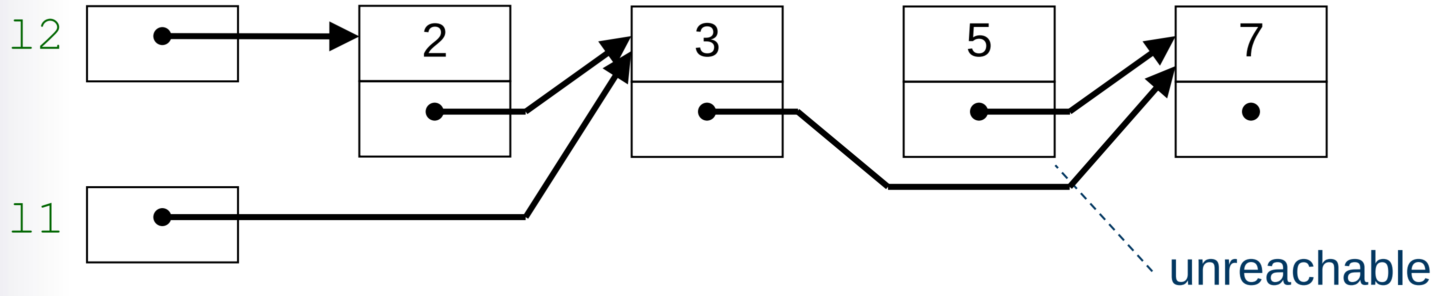
```
void main {
    IntList l1, l2;
    l1 = c(3, c(5, c(7, NULL)));
    l2 = c(2, l1);
    d(l1);
}
```

Example: C heap variables *(3)*

- After initializing `l1` and `l2`:

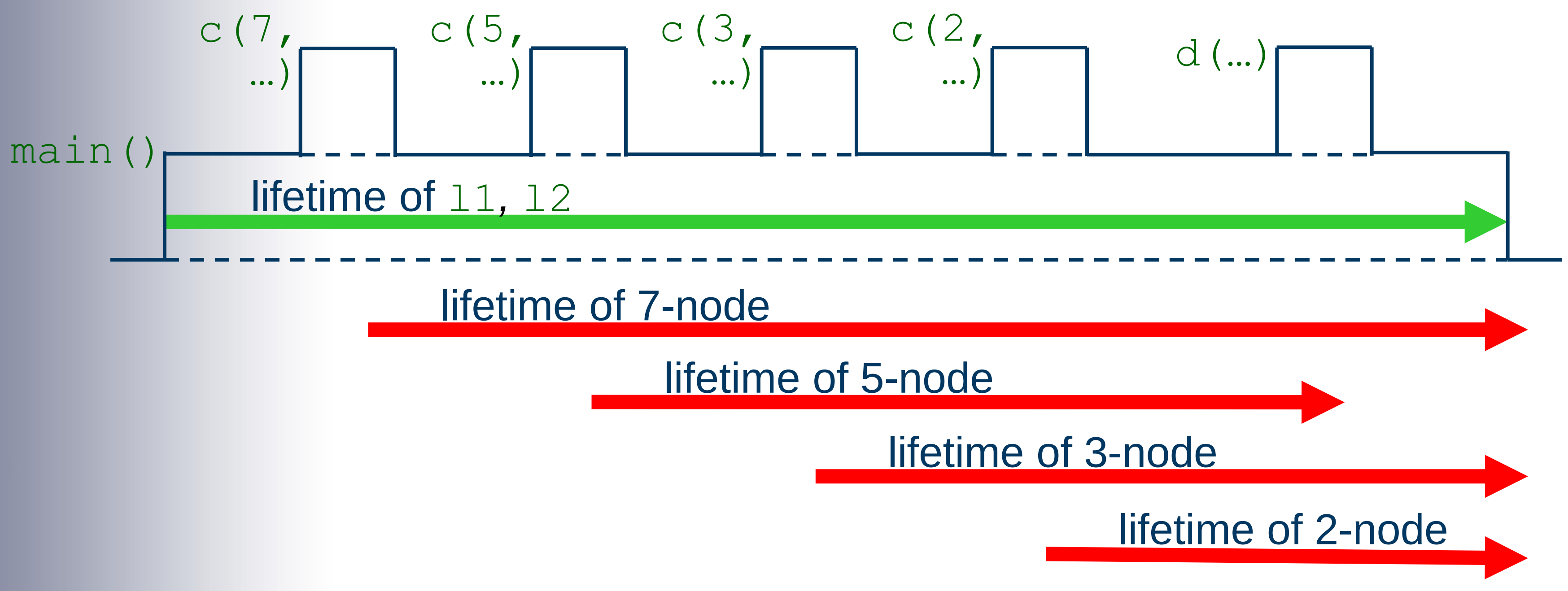


- After calling `d(l1)`:



Example: C heap variables *(4)*

- Lifetimes of local and heap variables:



- Heap variables' lifetimes *can* overlap one another and local/global variables' lifetimes.

Allocators and deallocators

- An **allocator** is an operation that creates a heap variable, yielding a pointer to that heap variable. E.g.:
  - C's allocator is a library function, `malloc()`.
  - Java's allocator is an expression of the form "**`new`** *C*`(…)`".
- A **deallocator** is an operation that *explicitly* destroys a designated heap variable. E.g.:
  - C's deallocator is a library function, `free()`.
  - Java has no deallocator at all.

Reachability

- A heap variable remains **reachable** as long as it can be accessed by following pointers from a global or local variable.
- A heap variable's lifetime extends from its creation until:
  - it is destroyed by a deallocator, or
  - it becomes unreachable, or
  - the program terminates.

University of Glasgow

Pointers *(1)*

- A **pointer** is a reference to a particular variable. (In fact, pointers are sometimes called **references**.)
- A pointer's **referent** is the variable to which it refers.
- A **null pointer** is a special pointer value that has no referent.
- A pointer is essentially the address of its referent in the store.
  - However, each pointer also has a *type*, and the type of a pointer allows us to infer the type of its referent.

Pointers *(2)*

- Pointers and heap variables can be used to represent recursive values such as lists and trees.
- But the pointer itself is a low-level concept. Manipulation of pointers is notoriously error-prone and hard to understand.
- For example, the C assignment "`p->succ = q;`" appears to manipulate a list, but which list? Also:
  - Does it delete nodes from the list?
  - Does it stitch together parts of two different lists?
  - Does it introduce a cycle?

University of Glasgow

Dangling pointers *(1)*

- A **dangling pointer** is a pointer to a variable that has been destroyed.
- Dangling pointers arise when:
  - a pointer to a *heap variable* still exists after the heap variable is destroyed by a deallocator
  - a pointer to a *local variable* still exists at exit from the block in which the local variable was declared.
- A deallocator immediately destroys a heap variable; all existing pointers to that heap variable then become dangling pointers.
  - Thus deallocators are inherently unsafe.

Dangling pointers *(2)*

- C is highly unsafe:
  - After a heap variable is destroyed, pointers to it might still exist.
  - At exit from a block, pointers to its local variables might still exist (e.g., if stored in global variables).
- Java is very safe:
  - It has no deallocator.
  - Pointers to local variables cannot be obtained.

Example: C dangling pointers

- Consider this C code:

```
struct Date {int y, m, d;}
typedef Date * DatePtr;

DatePtr date1 = (DatePtr)
   malloc(sizeof Date);
date1->y = 2008;
date1->m = 1;
date1->d = 1;

DatePtr date2 = date1;

free(date2);

printf("%d4", date1->y);
date2->y = 2009;
```

makes `date1` point to a newly-allocated heap variable

makes `date2` point to that same heap variable

deallocates that heap variable – `date1` and `date2` now contain dangling pointers

behaves unpredictably

Commands *(1)*

- A **command** (often called a **statement**) is a program construct that will be **executed** to update variables.
- Commands are characteristic of imperative and OO PLs (but not functional PLs).
- Simple commands:
  - A **skip command** is a command that does nothing.
  - An **assignment command** is a command that uses a value to update a variable.
  - A **procedure call** is a command that calls a proper procedure with argument(s). Its net effect is to update some variables.

University of Glasgow

Commands *(2)*

- Compound commands:
  - A **sequential command** is a command that executes its sub-commands in sequence.
  - A **conditional command** is a command that chooses *one* of its sub-commands to execute.
  - An **iterative command** is a command that executes its sub-command repeatedly. This may be:
    - **definite iteration** (where the number of repetitions is known in advance)
    - **indefinite iteration** (where the number of repetitions is not known in advance).
  - A **block command** is a command that contains declarations of local variables, etc.

University of Glasgow

Example: Java assignment commands

- Java single assignment:

```
m = n + 1;
```

- Java multiple assignment:

```
m = n = 0;
```

- Java assignment combined with binary operator:

`m += 7;` ---------------------- equivalent to "`m = m+7;`"

`n /= b;` ---------------------- equivalent to "`n = n/b;`"

University of Glasgow

Example: Java conditional commands

- Java if-command:

```
if (x > y)
    out.print(x);
else
    out.print(y);
```

- Java switch-command:

```
Date today = …;
switch (today.m) {
    case  1:  out.print("JAN"); break;
    case  2:  out.print("FEB"); break;
    …
    case 11:  out.print("NOV"); break;
    case 12:  out.print("DEC");
}
```

Breaks are essential here!

Example: Java iterative commands *(1)*

- Java while-command:

```
Date[] dates;
…
int i = 0;
while (i < dates.length) {
    out.print(dates[i]);
    i++;
}
```

- Java for-commands (both forms):

```
for (int i = 0; i < dates.length; i++)
    out.print(dates[i]);

for (Date d : dates)
    out.print(d);
```

Example: Java iterative commands *(2)*

- Java do-while-command:

```
static String render (int n) {
    String s = "";
    int m = n;
    do {
        char d = (char)(m % 10) + '0';
        s = d + s;
        m /= 10;
    } while (m > 0);
    return s;
}
```

- Here the loop condition is evaluated *after* each repetition of the loop body.

Example: Java block command

- Java block command:

```
if (x > y) {
    int z = x;
    x = y;
    y = z;
}
```

University of Glasgow

Expressions with side effects

- The primary purpose of evaluating an expression is to yield a value.
- In most imperative and OO PLs, evaluating an expression can also update variables – these are **side effects**.
- In C and Java, the body of a function is a *command*. If that command updates global or heap variables, calling the function has side effects.
- In C and Java, assignments are in fact expressions with side effects: "$V = E$" stores the value of E in V as well as yielding that value.

Example: side effects

- The C function `getchar(fp)` reads a character and updates the file variable that `fp` points to.
- The following C code is correct:

```
char ch;
while ((ch = getchar(fp)) != NULL)
   putchar(ch);
```

- The following C code is incorrect (why?):

```
enum Gender {FEMALE, MALE};
Gender g;
if (getchar(fp) == 'F')   g = FEMALE;
else if (getchar(fp) == 'M')   g = MALE;
else …
```