Memory Management

Memory Organization

During run time, variables can be stored in one of three “pools”:

1. Stack
2. Global area (Static heap)
3. Dynamic heap

The machine code and data associated with it are in the “code segment”

Stack

Maintains memory during function calls:

- Argument of the function
- Local variables
- Address to return to in “code segment”

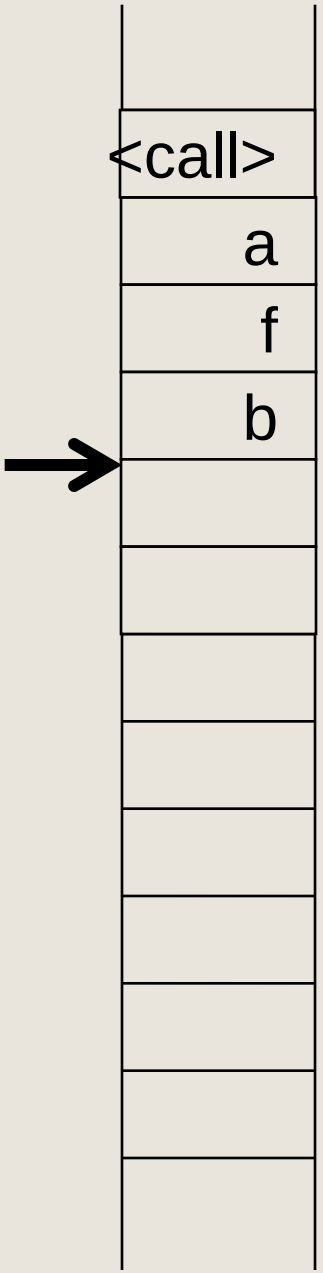
Variables on the stack have limited “life time”

Stack size must be defined at compile time

Stack - Example

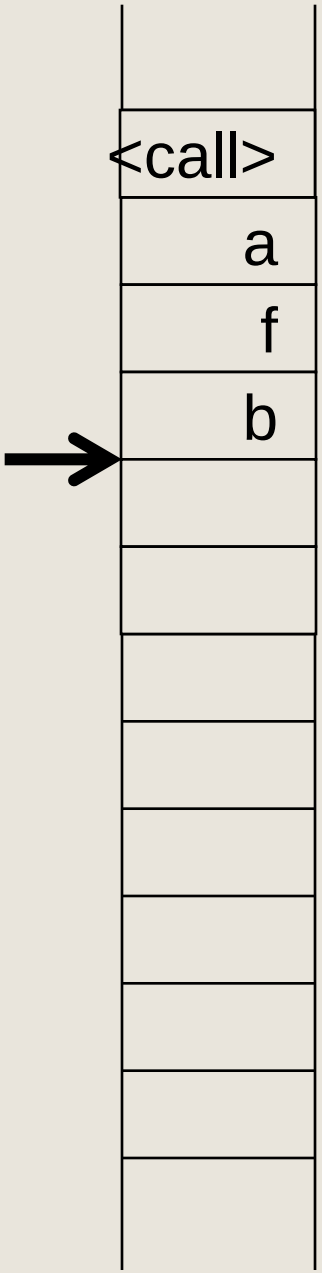
```
int foo( int a, double f )
{
    int b;
    ...

}
```



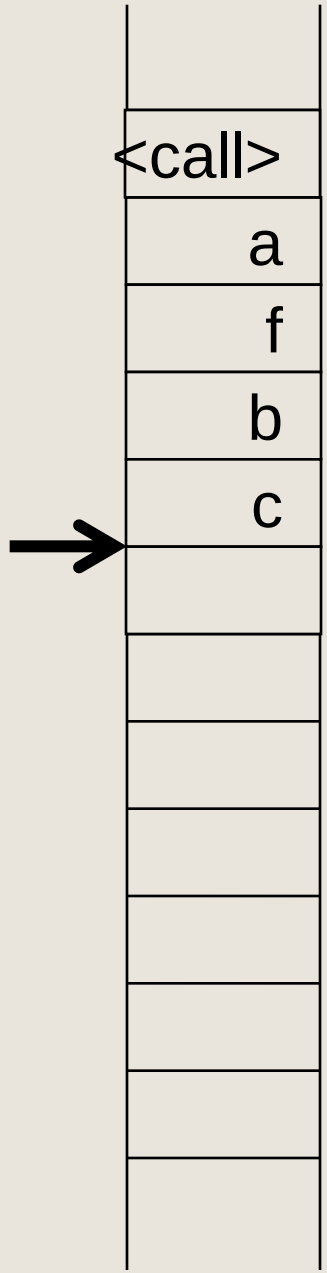
Stack - Example

```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```



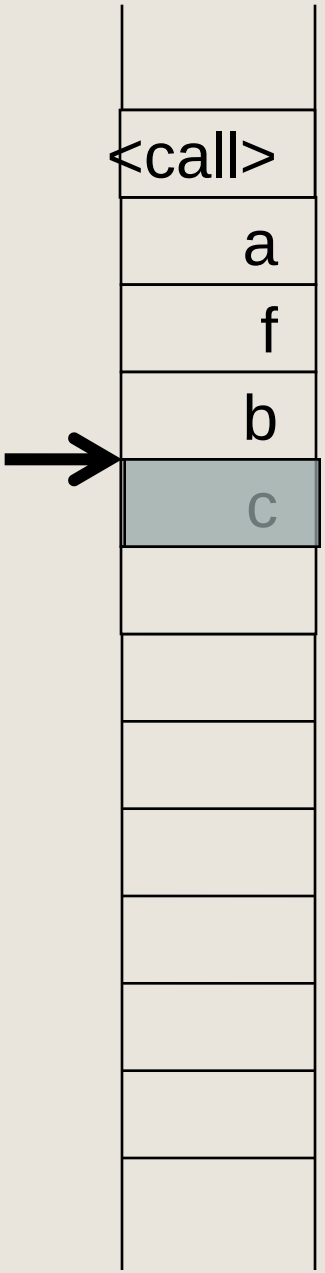
Stack - Example

```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```



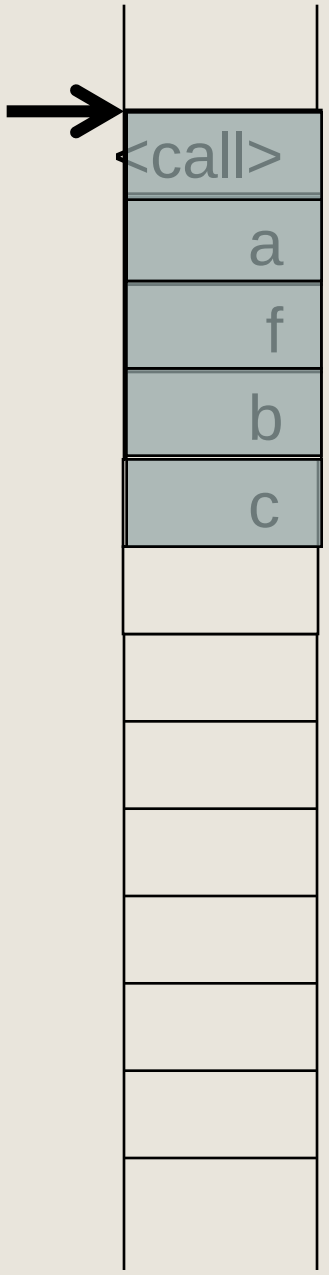
Stack - Example

```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```



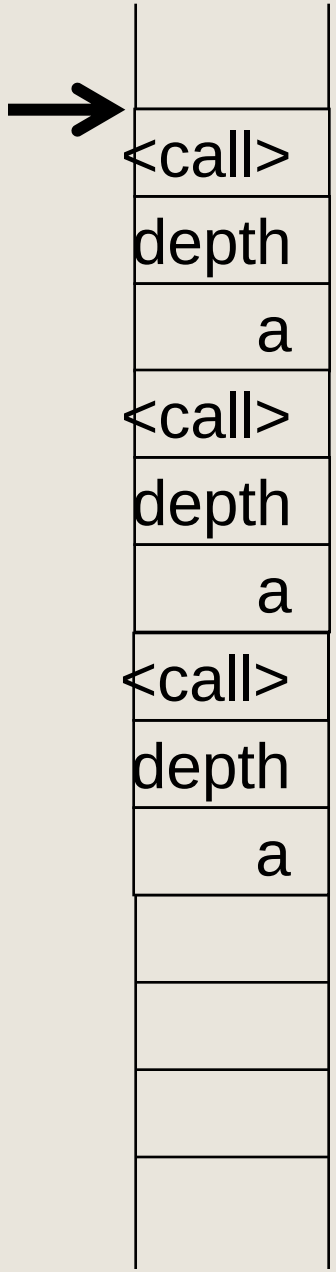
Stack - Example

```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```



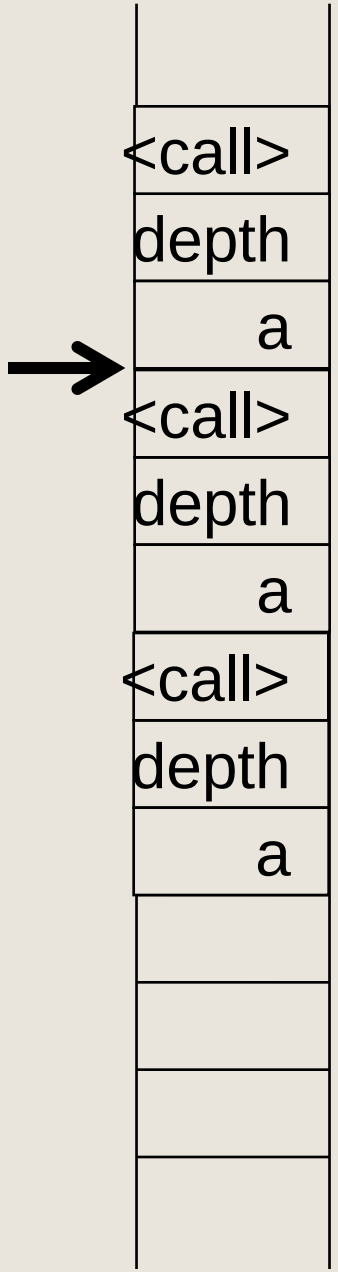
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    …
```



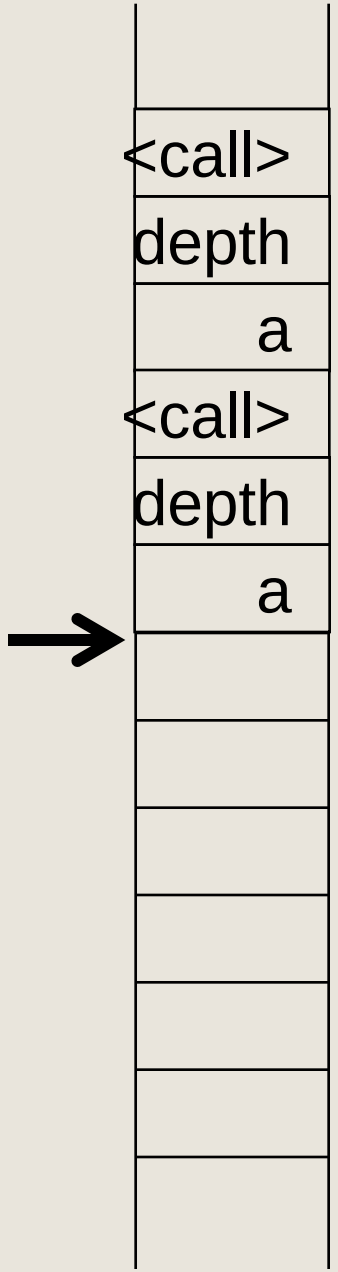
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    …
```



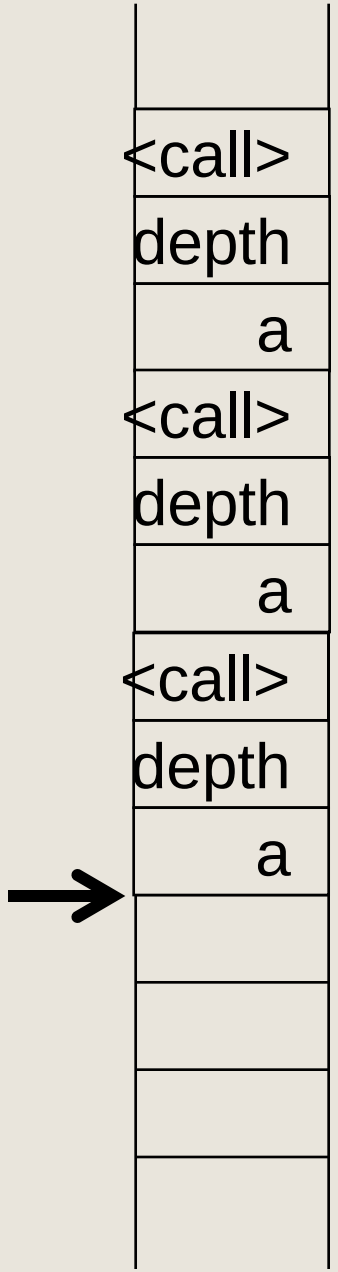
Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    …
```



Stack – recursive example

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth-1 );
    }
}
int main()
{
    foo(3);
    …
```



Stack -- errors

```
void foo( int depth )
{
    int a;
    if( depth > 1 )
    {
        foo( depth );
    }
}
```

Will result in run time error:

out of stack space

Global area

Memory for global variables – must be defined in compile time

```
#include <stdio.h>

const int LIST_OF_NUMBER_SIZE = 1000;
int ListOfNumbers[LIST_OF_NUMBER_SIZE];

int main()
{
   ...
```

Global area: `reverse` example

Example: program to reverse the order of lines of a file.

To this task, we need to

- Read the lines into memory
- Print lines in reverse

How do we store the lines in memory?

Global area: reverse example

```
const int LINE_LENGTH = 100;
const int NUMBER_OF_LINES = 10000;
char g_lines[NUMBER_OF_LINES][LINE_LENGTH];


...
int main()
{
   int n= ReadLines();
   while(n>=1) {
      --n;
      printf("%s\n", g_lines[n]);
   }
}
```

Global area: `reverse` example

This solution is problematic:

- The program cannot handle files **larger** than these specified by the **compile time choices**
- If we set `NUMBER_OF_LINES` to be very large, then the program requires this amount of memory even if we are reversing a short file
- The life-time of global variables is the entire program running time.

➔**Want to use memory on "as needed" basis**

Dynamic Heap

1. Memory that can be allocated and freed during run time.
2. The program controls how much is allocated and when.
3. Limitations based on run-time situation - available memory on the computer.

Allocating Memory from Heap

```
#include <stdlib.h>

void* malloc( size_t Size );
```

Returns a pointer to a new memory block of size `Size`, or `NULL` if it cannot allocate memory of this size.

Why do we need dynamic memory?

In static arrays we need to know the array size at compilation time:

```
int s_arr[5] = {1,2,3,4,5};
```

But, this size may not be known in advance, or might be changed during the program execution.

The solution: use dynamic array:

```
int *d_arr = (int*)malloc(sizeof(int)*arraySize);
```

We can also use it for a single int

```
int* iptr =
      (int*) malloc(sizeof(int));
```

Always check allocation success - or document that behavior is undefined otherwise

```
char *str = (char *)malloc(5*sizeof(char));
if (str==NULL)
{
  //print error message
   exit(1); //or perform relevant operation
}
```

What will be in the allocated memory?

```
int* iptr =
     (int*) malloc(sizeof(int));

struct Complex* complex_ptr =
     (struct Complex*)
     malloc(sizeof(struct Complex));
```

What will be in the allocated memory?

`*iptr` = undefined

`*complex_ptr` = undefined

Good design – an initialization function for a struct (poor constructor).

```
struct Complex*
        newComplex(double r, double i) {
   struct Complex *p =
     (struct Complex*)
     malloc (sizeof (Complex));
   p->_real = r;
   p->_imag = i;
   return p;
}
```

Good design – an initialization function for a struct (poor constructor).

```
struct Complex *complex_ptr =
            newComplex (1.0, 2.1);
```

De-allocating memory

```
void free( void *p );
```

Returns the memory block pointed by p to the pool of unused memory

No error checking! Undefined if p:

- Is not NULL and
- Was not allocated by malloc or its friends or `free`-ed before

Example of free

```
int* iptr =
     (int*) malloc(sizeof(int));
...

free(iptr);
```

How to free a pointer to struct?

```
void foo( double r, double i )
{
    struct Complex* p_c =
                    newCompelx (r,i);
    // do something with p_c
    free(p_c);
}
```

This version frees the allocated memory

What about structs that allocate memory by themselves?

```
struct Vec
{
   int *_arr;
};

struct Vec *newVec (int length)
{
   struct Vec *p = (struct Vec*) malloc (sizeof (struct Vec));
   p->_arr = (int*) malloc (sizeof(int)*length);
   return p;
}

int main ()
{
   struct Vec *v = newVec(5);
   // do something
   free (v);
}
```

Memory leak – every malloc should have a corresponding free – otherwise a bug!

```
struct Vec
{
   int *_arr;
};

struct Vec *newVec (int length)
{
   struct Vec *p = (struct Vec*) malloc (sizeof (struct Vec));
   p->_arr = (int*) malloc (sizeof(int)*length);
   return p;
}

int main ()
{
   struct Vec *v = newVec(5);
   // do something
   free (v);
}
```

Poor destructors

```
struct Vec
{
   int *_arr;
};

void freeVec (struct Vec *v)
{
   free (v->_arr);
   free (v);
}

int main ()
{
   struct Vec *v = newVec(5);
   // do something
   freeVec (v);
}
```

Poor destructors

```
struct Vec
{
   int *_arr;
};

void freeVec (struct Vec *v)
{
   free (v->_arr);
   free (v);
}

int main ()
{
   struct Vec *v = newVec(5);
   // do something
   freeVec (v);
}
```

But, what if v is NULL ?

Poor destructors

```
struct Vec
{
   int *_arr;
};
```

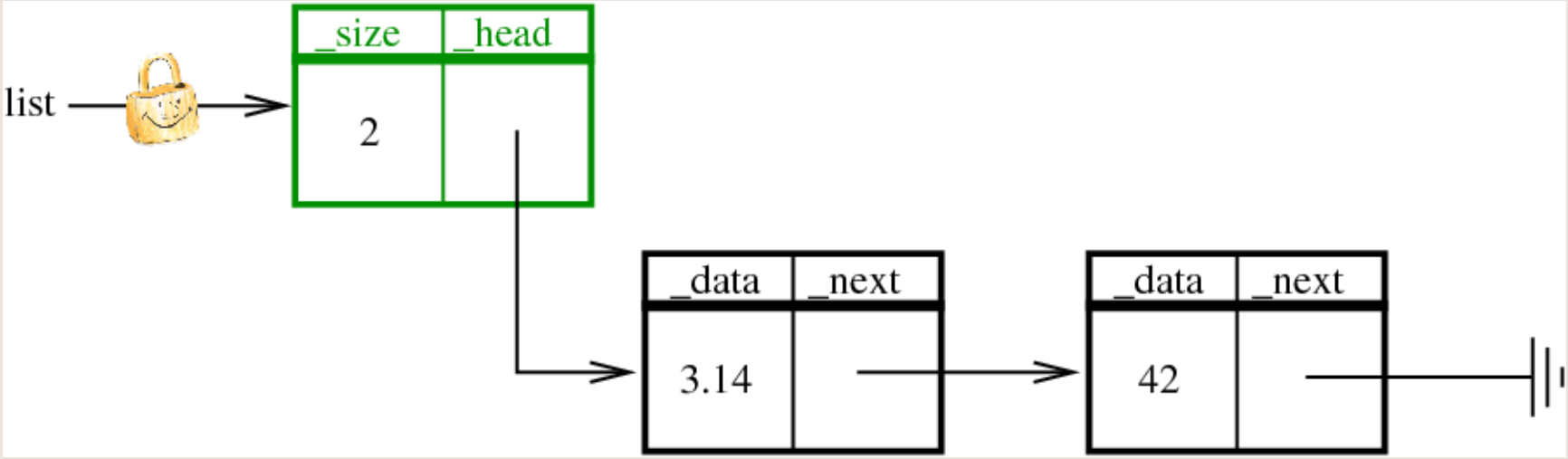
```
void freeVec (struct Vec *v) {
   if (v==NULL) return;
   free (v->_arr);
   free (v);
}
```

```
int main () {
   struct Vec *v = newVec(5);
   // do something
   freeVec (v);
}
```

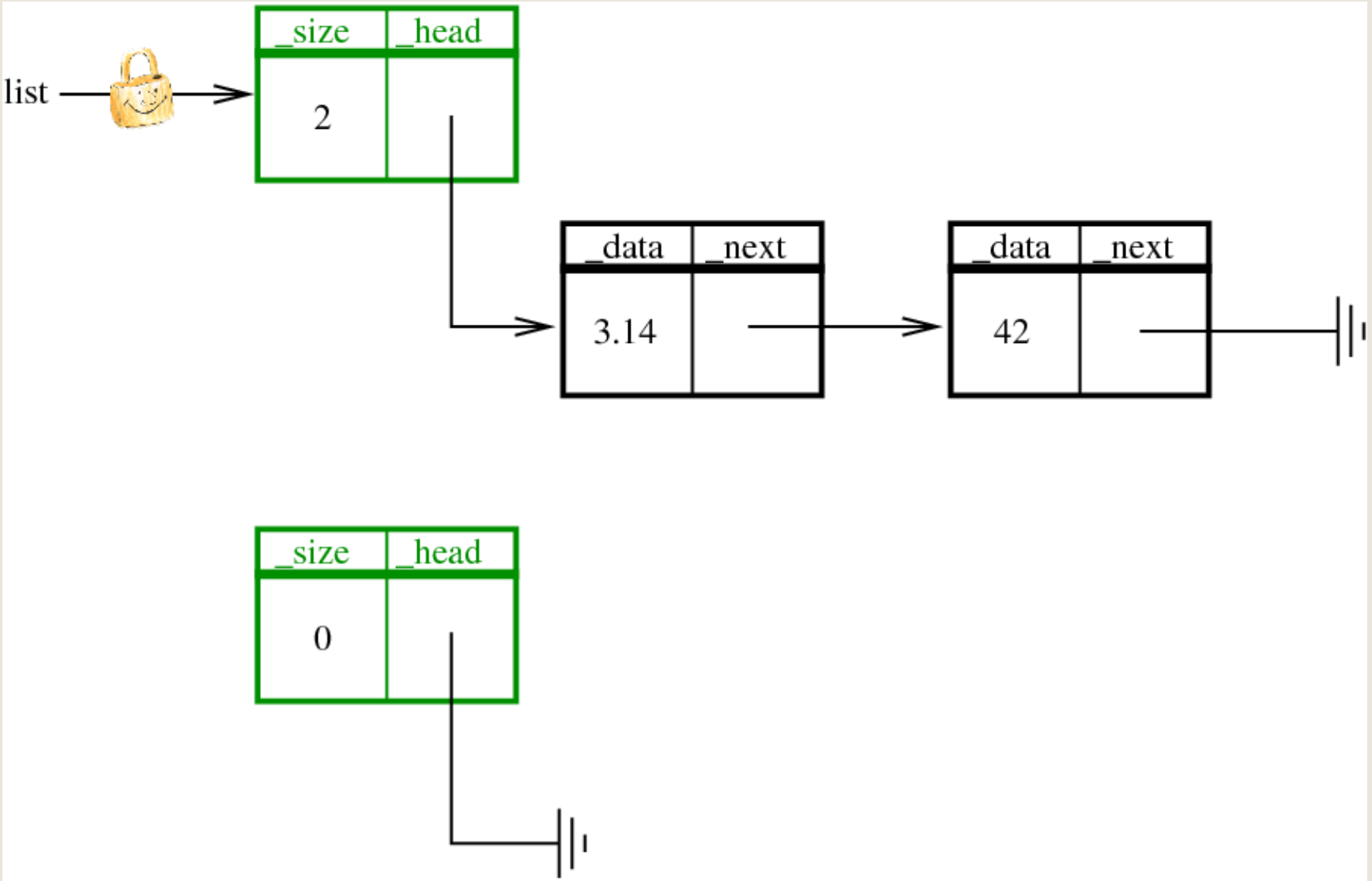
Unit Testing Reminder:

See slides: programming style and correctness

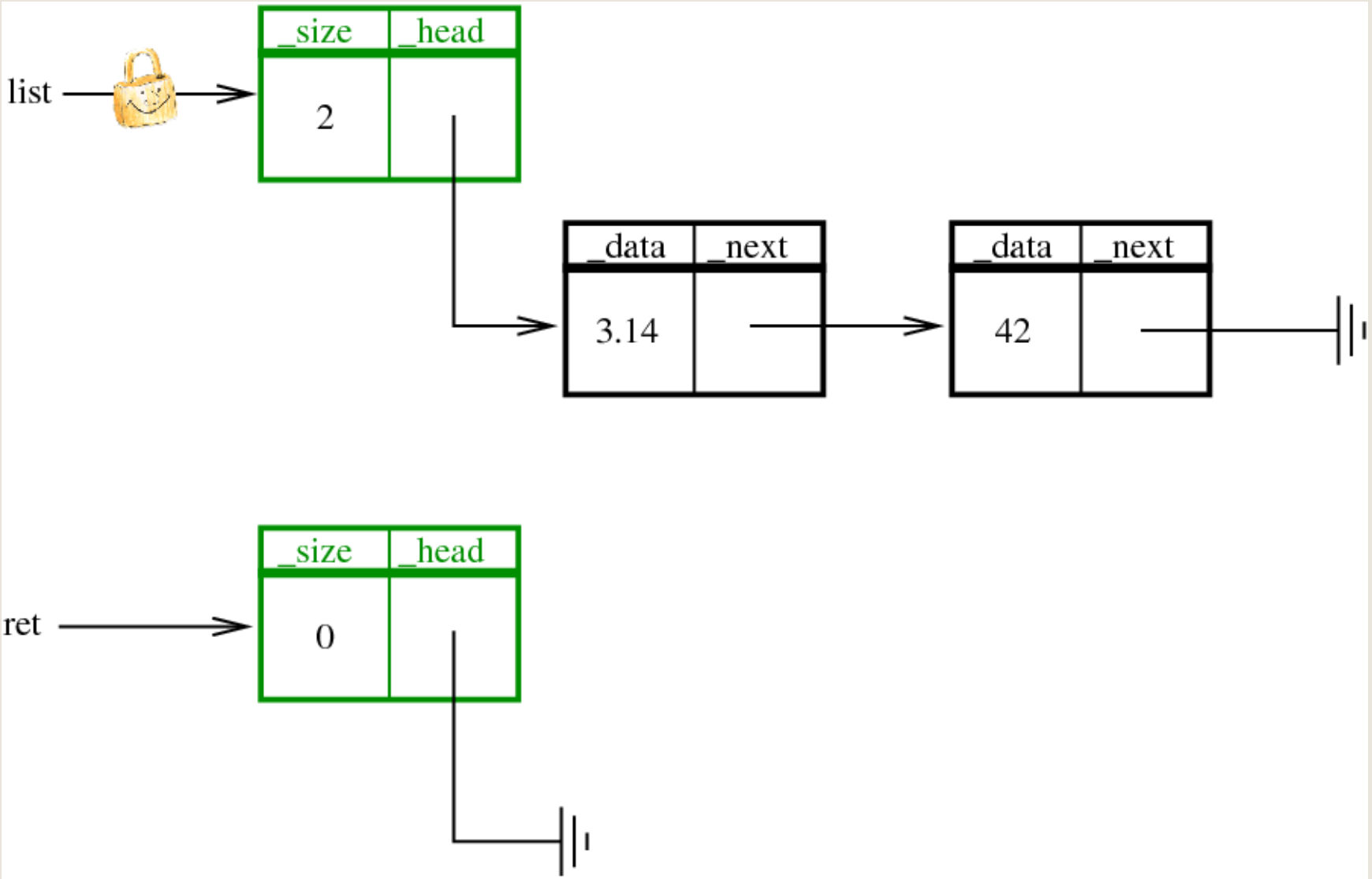
List code example



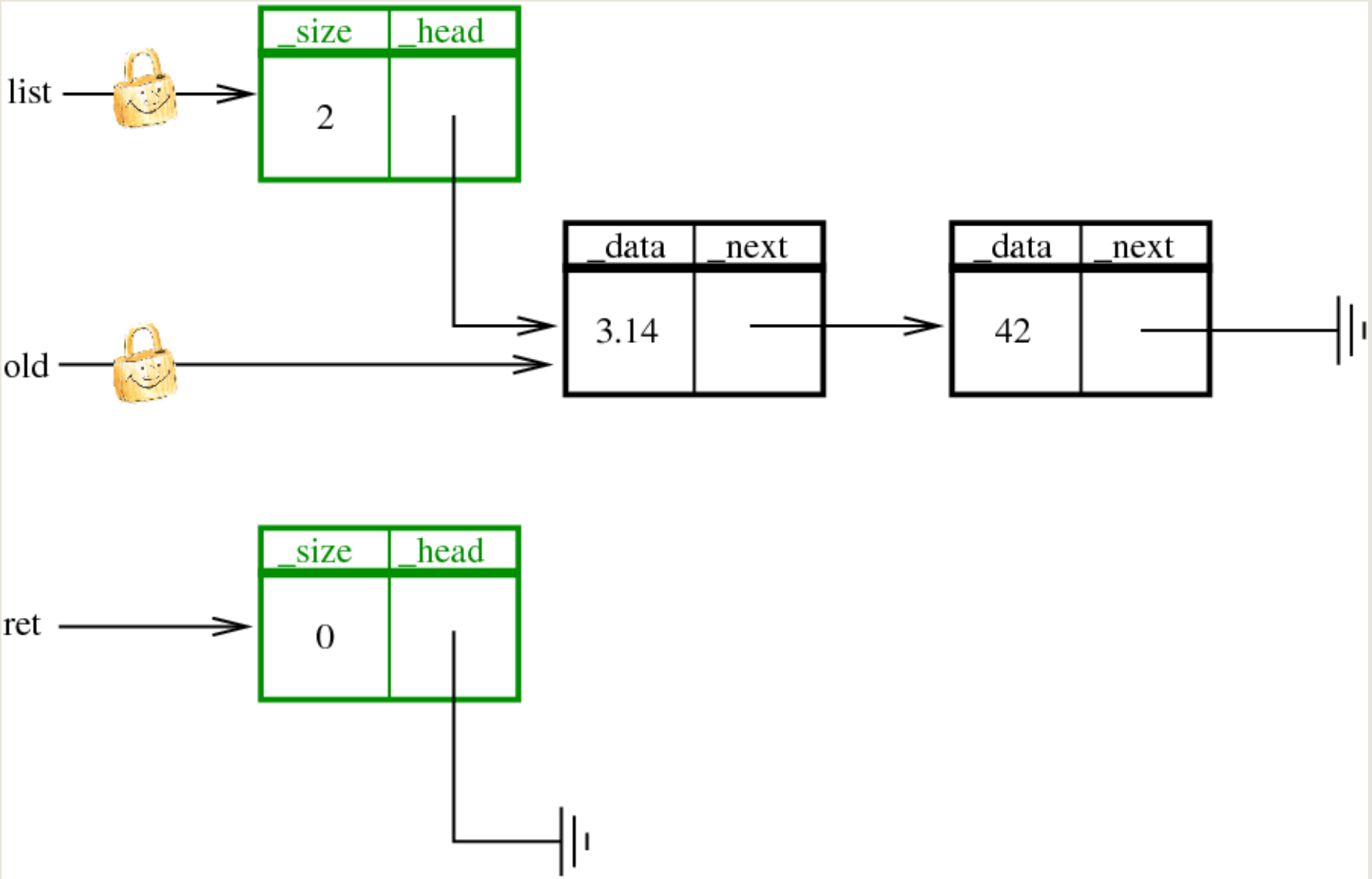
List* List_clone(const List* list) {




```
List* List_clone(const List* list) {
List* ret= List_alloc();
```

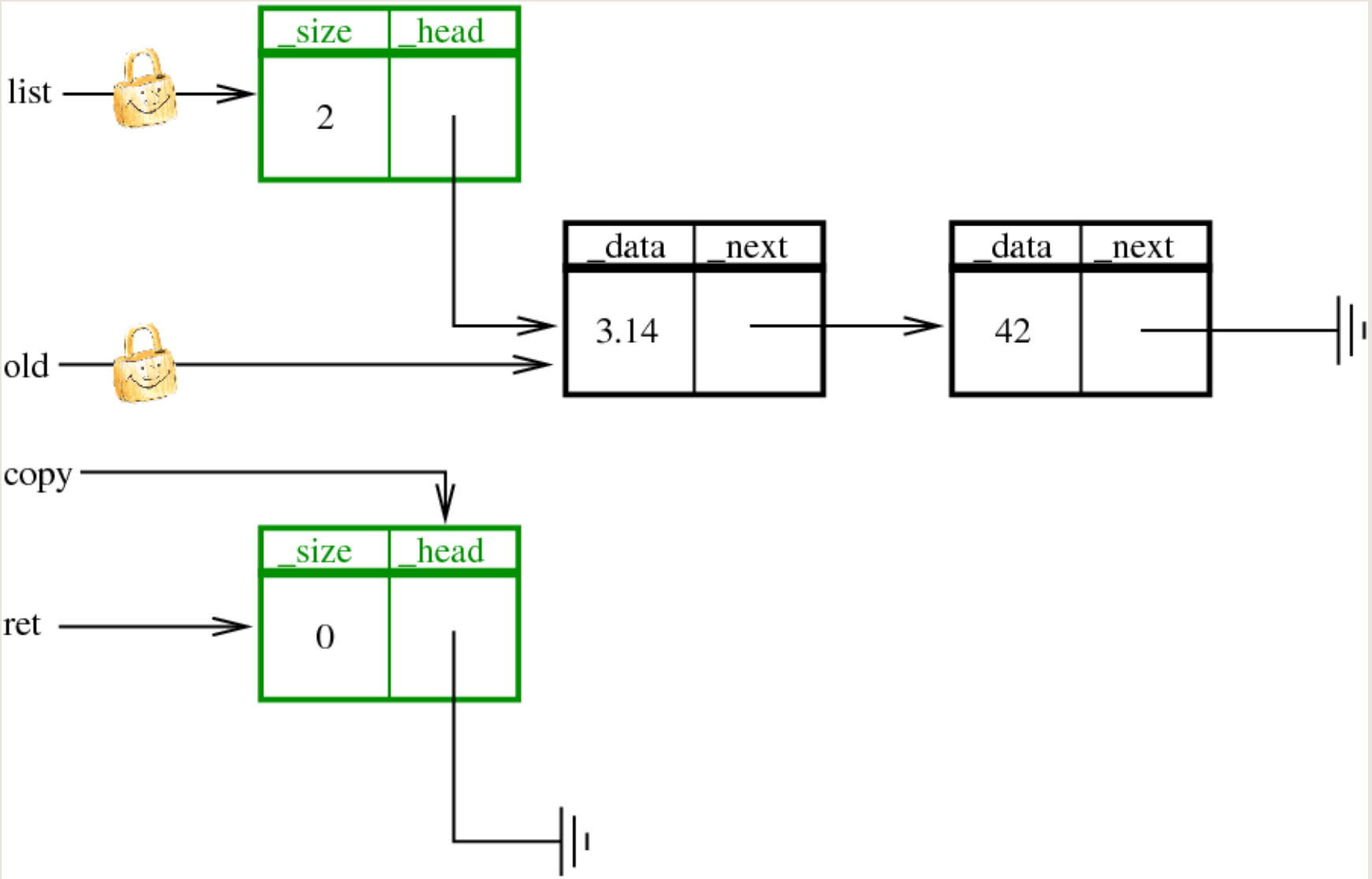



```
List* List_clone(const List* list) {
List* ret= List_alloc();
```



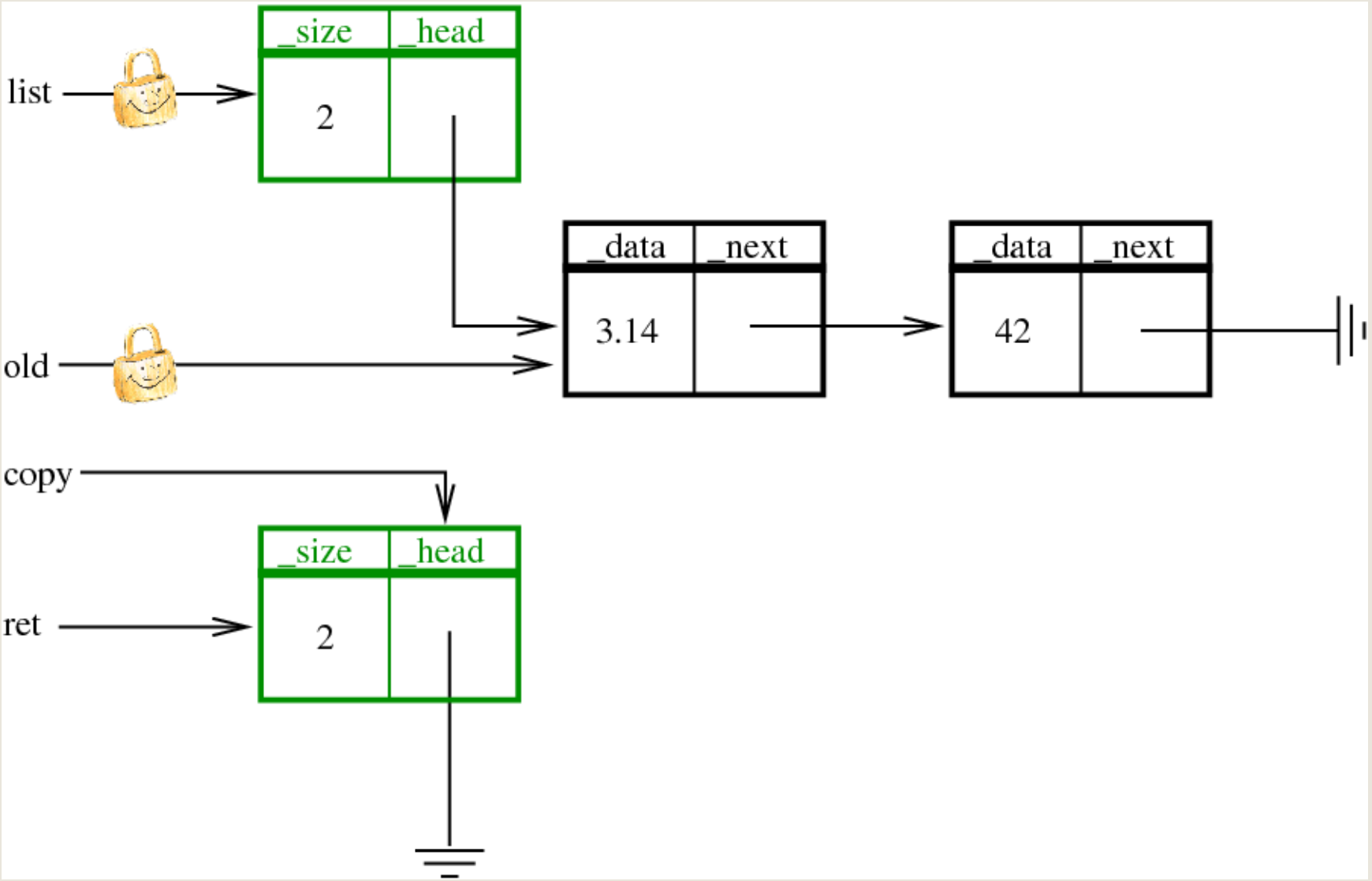
List* ret= List_alloc();

const Node* old= list->_head;

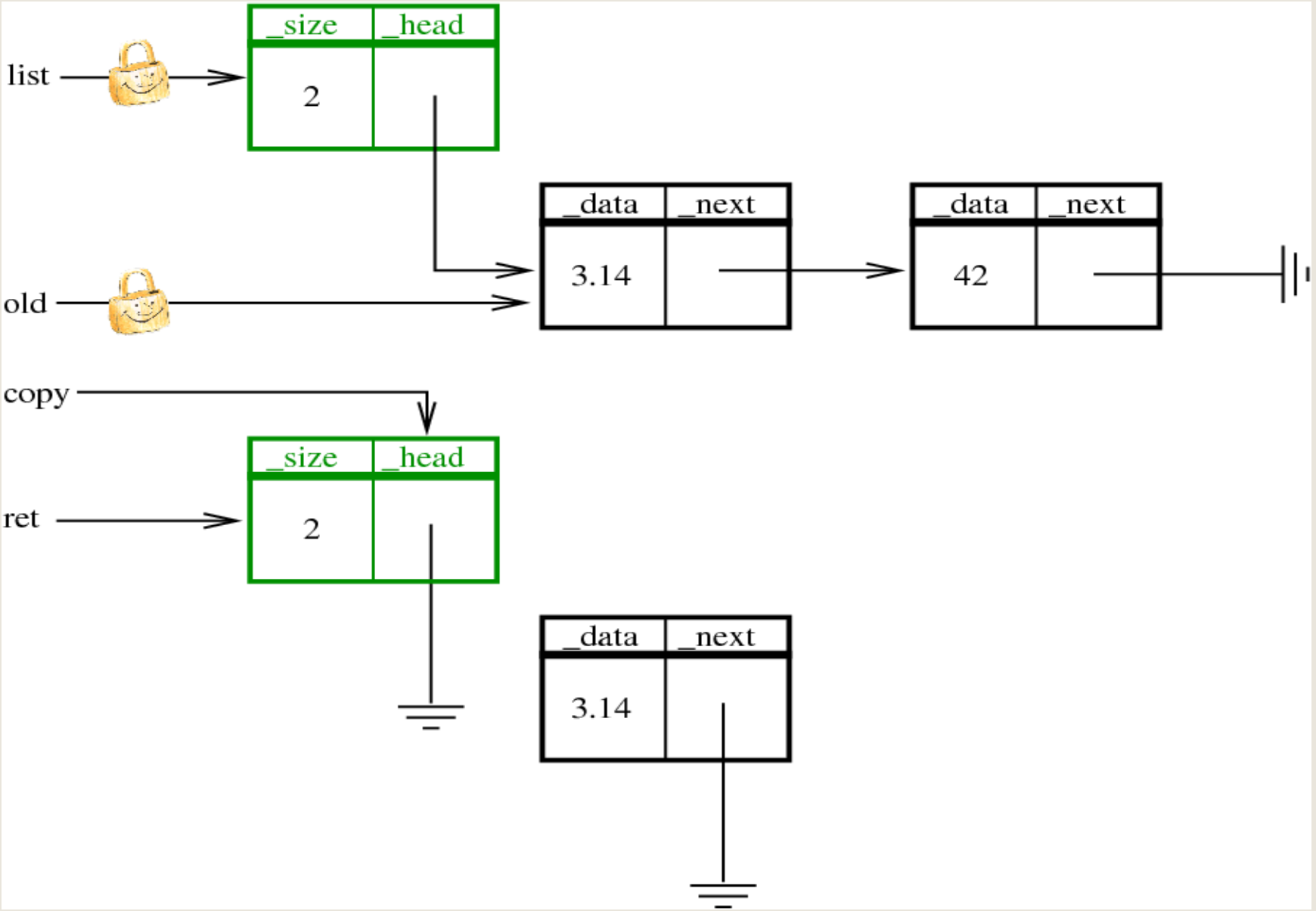


const Node* old= list->_head;

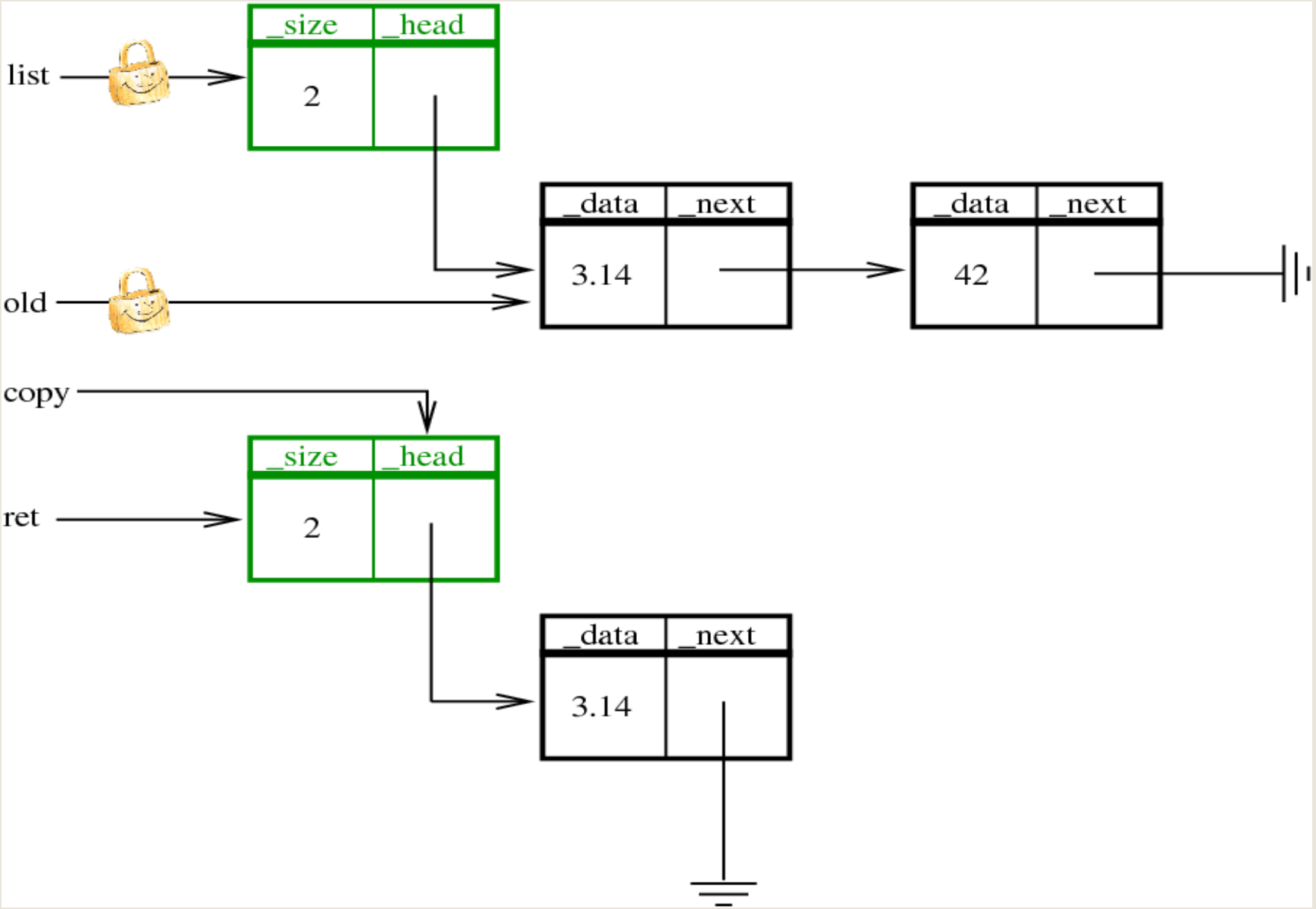
Node copy= &(ret->_head);**



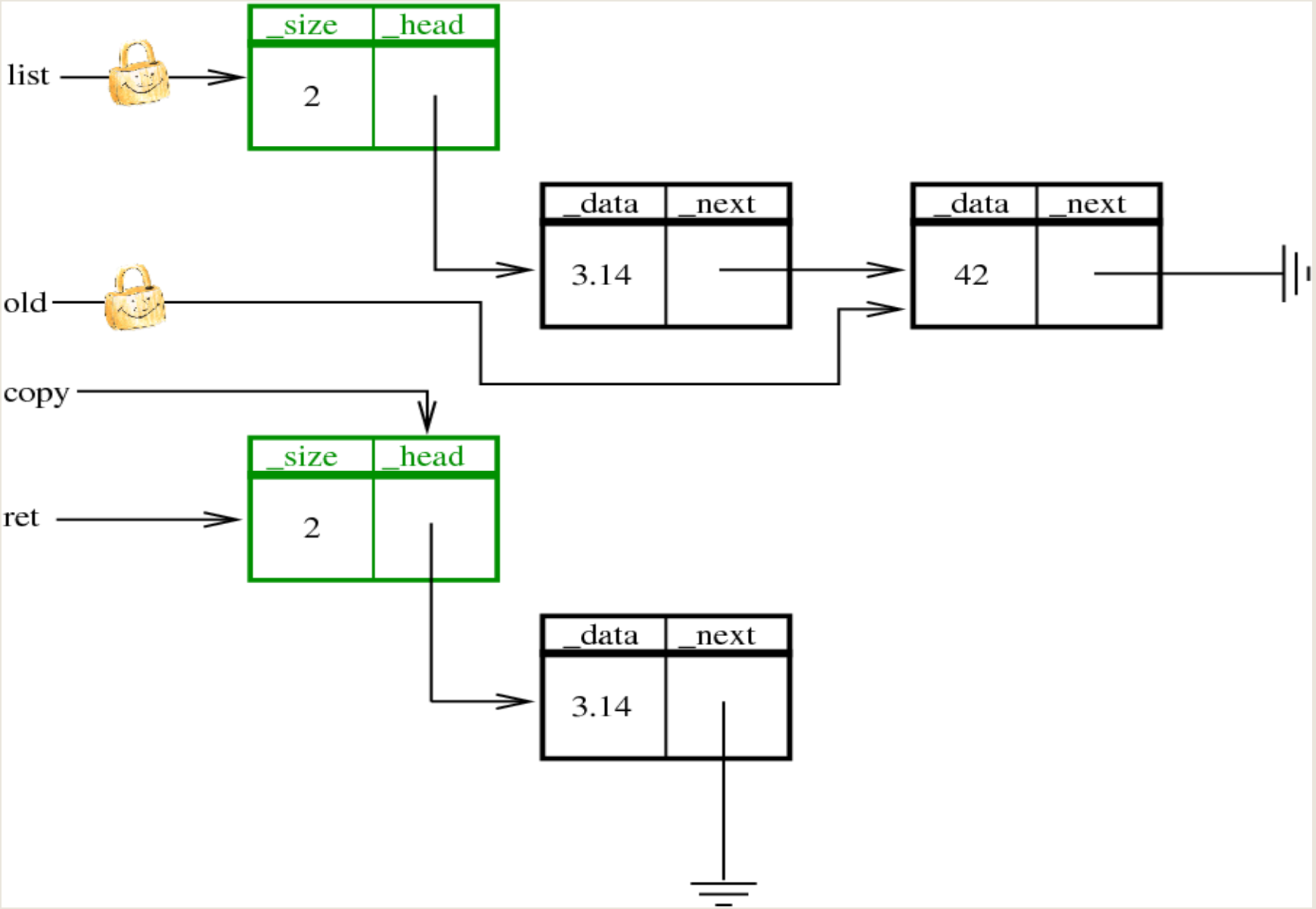

```
Node** copy= &(ret->_head);
ret->_size= list->_size;
        while(old) {
```



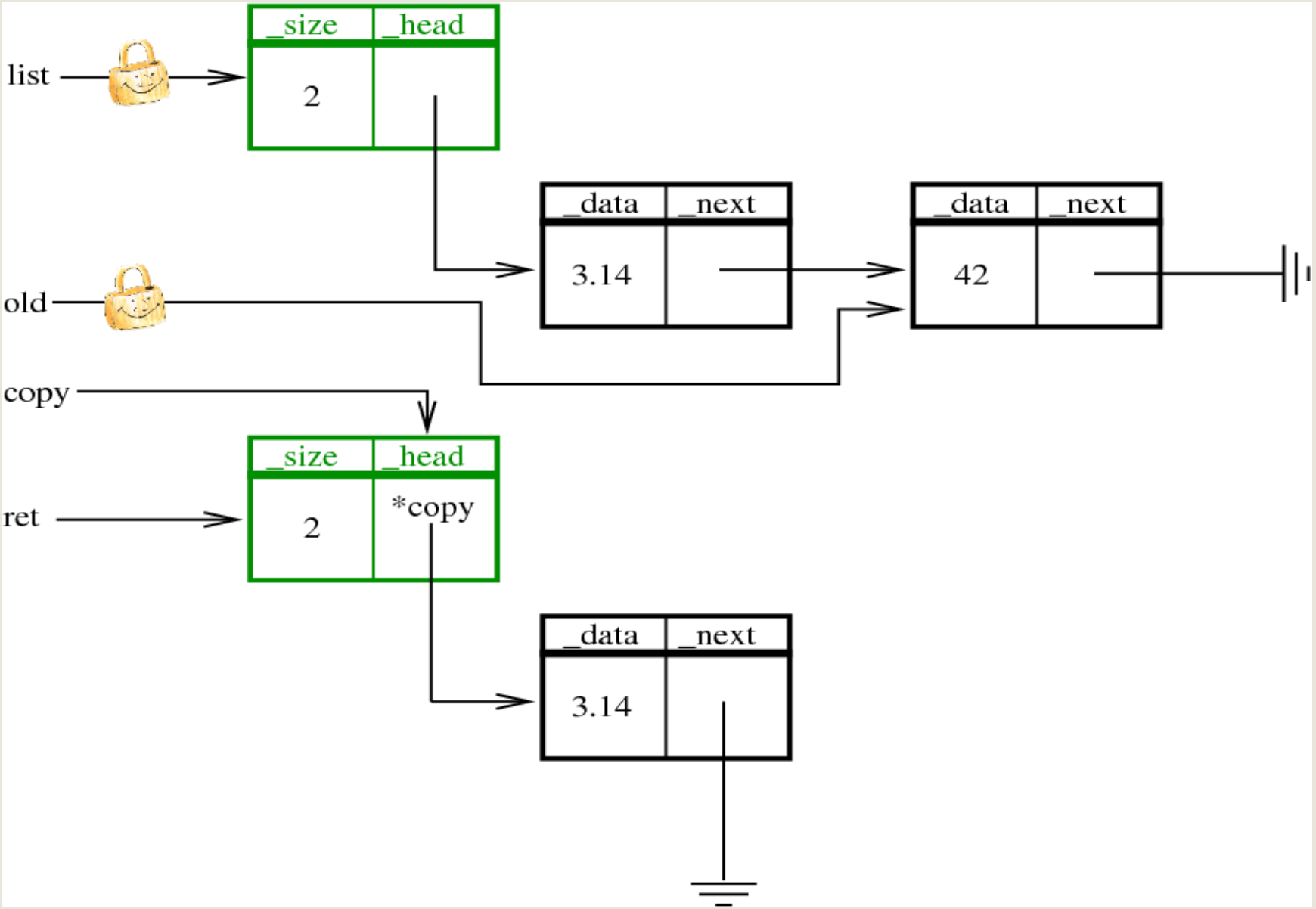
***copy= Node_alloc(old->_data,NULL);**



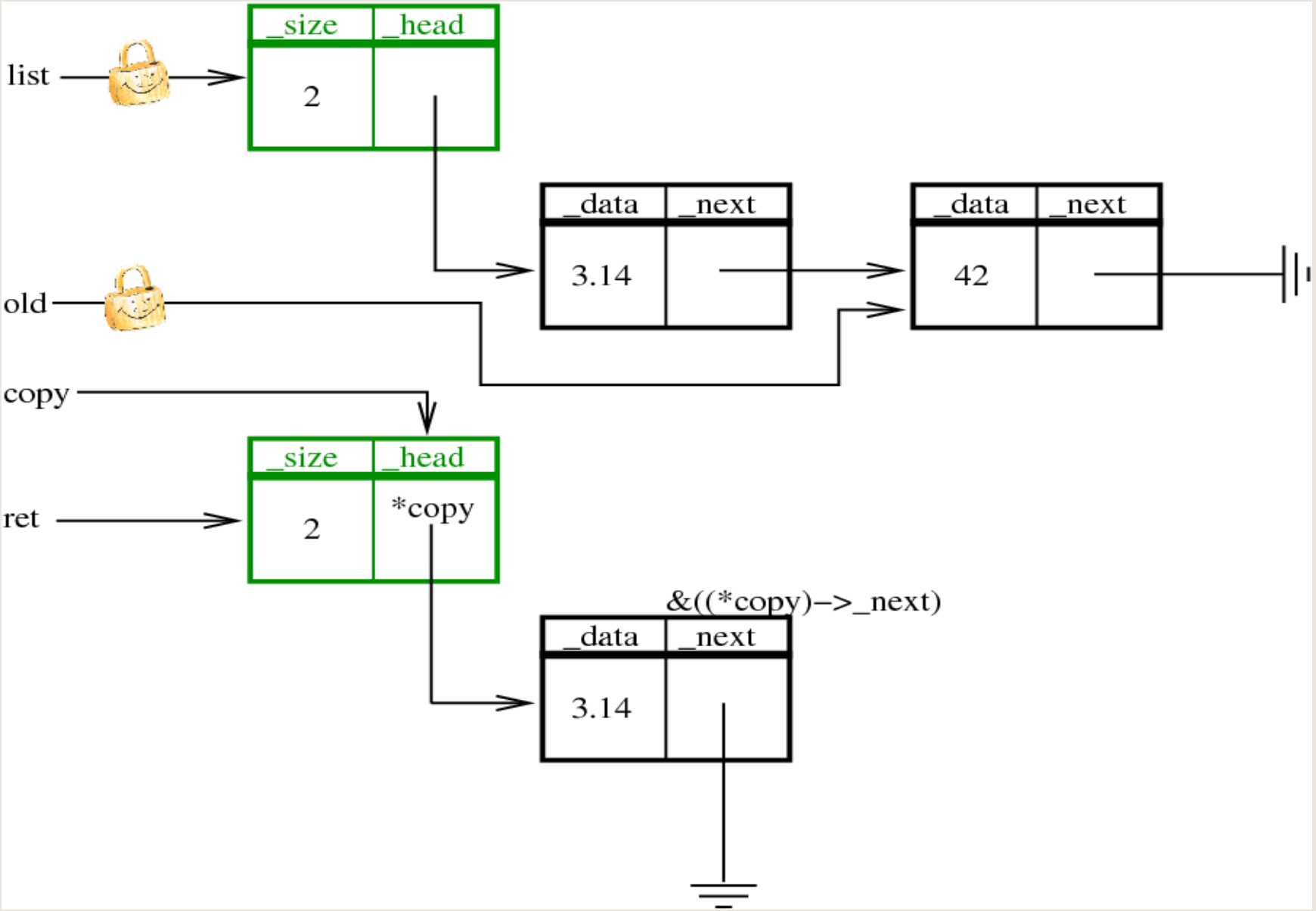
***copy= Node_alloc(old->_data,NULL);**


list
_size
_head
2
_data
_next
3.14
_data
_next
42
old
copy
ret
_size
_head
2
_data
_next
3.14

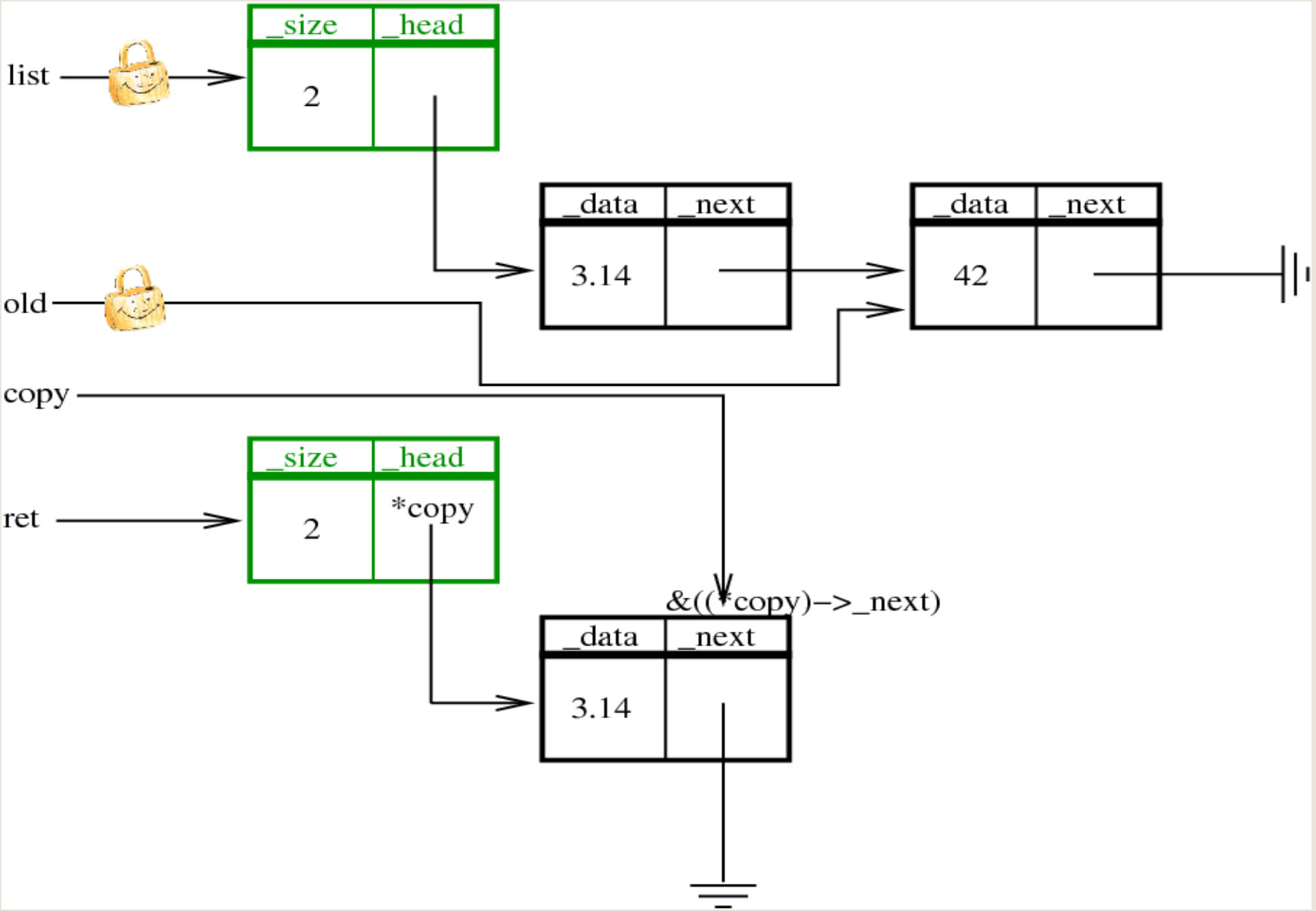
old= old->_next;



copy= &((*copy)->_next);



copy= &((*copy)->_next);



copy= &((*copy)->_next);

Array reallocation – we need to resize an array

Array reallocation – **wrong** solution

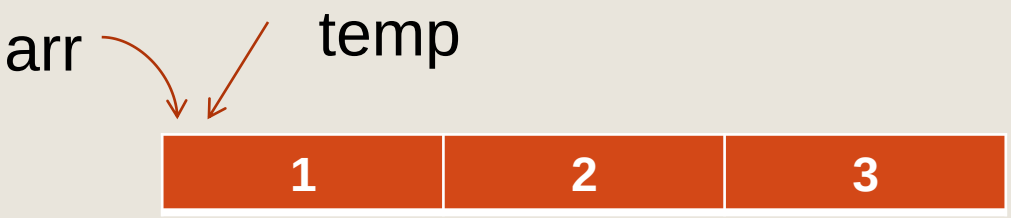
```
int *arr = (int*)malloc(sizeof(int)*arraySize);
//put some values in arr
...
int* nArr= (int*)malloc(sizeof(int)*(arraySize+1));
//copy values from arr to newArr
...
arr = nArr;
// BAD: lost address of first allocation!
```

Array reallocation

Allocate array on the heap and assign a pointer to it

Array reallocation

Use temporary pointer to point to the old array.



Array reallocation

Use temporary pointer to point to the old array.

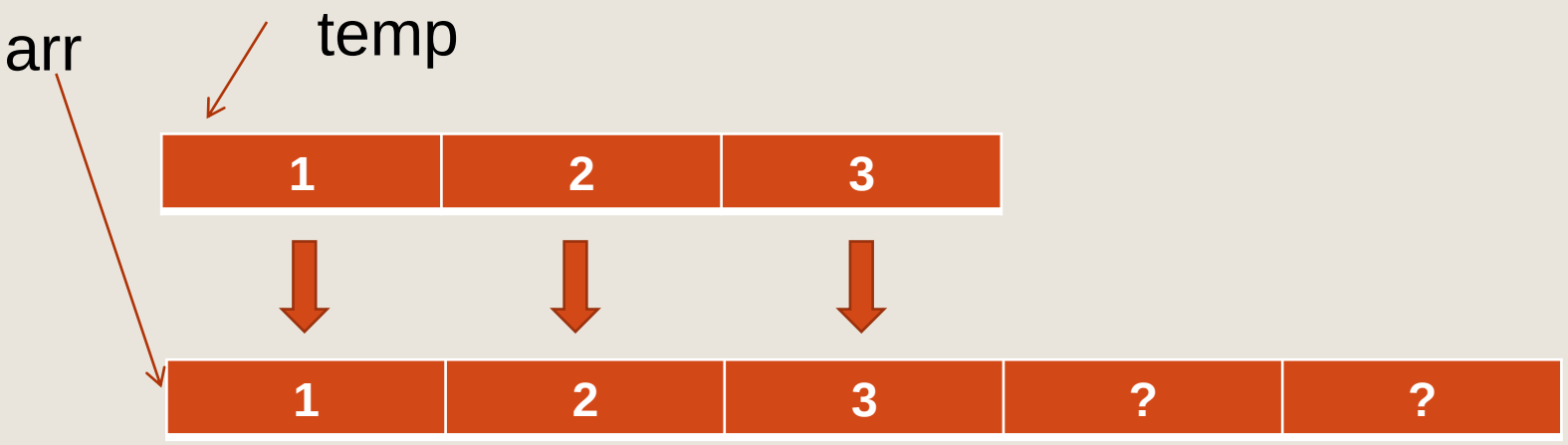
Reallocate new array.

Array reallocation

Use temporary pointer to point to the old array.

Reallocate new array.

Copy values from the old array to the new one.



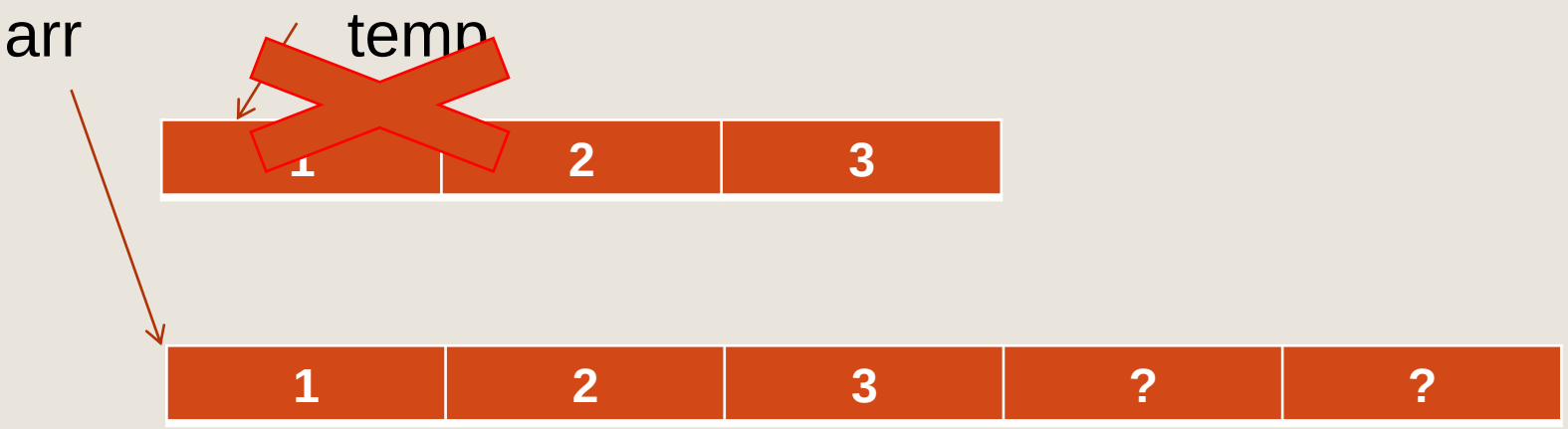
Array reallocation

Use temporary pointer to point to the old array.

Reallocate new array.

Copy values from the old array to the new one.

Free the old array.



Array reallocation – version 1

```
void reallocateMemory(int **arr,
      unsigned int oldSize, unsigned int newSize) {

   int *temp = *arr;
   //partial example-do not forget checking malloc!
   *arr = (int*)malloc(sizeof(int)*newSize);
   int i = 0;
   while(i < oldSize) {
      (*arr)[i] = temp[i];
      i++;
   }
   free(temp);

}
```

Array reallocation – version 1

```
int main() {

    int *arr = (int*)malloc(sizeof(int)*arrSize);
    //do some stuff
    ...
    reallocateMemory(&arr,arrSize,newSize);
    free(arr)
```

Array reallocation – version 2

```
int* reallocateMemory(int *arr,
      unsigned int oldSize, unsigned int newSize) {

   int *temp = arr;
   //partial example-do not forget checking malloc!
   arr = (int*)malloc(sizeof(int)*newSize);
   int i = 0;
   while(i < oldSize) {
      arr[i] = temp[i];
      i++;
   }
   free(temp);
   return arr;

}
```

Array reallocation – version 2

```
int main() {

    int *arr = (int*)malloc(sizeof(int)*arrSize);
    //do some stuff
    ...
    arr = reallocateMemor(arr,arrSize,newSize);
    free(arr);
```

Array reallocation – using realloc

```
int * arr = (int*)malloc(sizeof(int)*oldSize);


arr = (int*)realloc(arr,sizeof(int)*newSize);
```

realloc tries to reallocate the new memory in place,
if fails, tries elsewhere

The old data is preserved

The new cells contents is undefined

If `arr`=NULL, behaves like alloc

Further Knowledge

Read manual page of

malloc

calloc

realloc

free

Variable Length Arrays (VLA)

How to allocate an array on the stack with size defined only on running time?

ANSI C - use malloc:

```
size_t length= get_user_length();
int* arr= (int*) malloc(length*sizeof(int));
```

C99 - introduce VLA

```
size_t length= get_user_length();
int arr[length];
```

VLA - disadvantage

- No clear definition of the implementation
- Not accepted by C++ standard (use std::vector instead)
- Not mandatory feature in C11
- Still not supported by common compilers like visual studio 2012 (and probably will not be supported)
- **Not allowed during the course (use –Wvla or Makfile in the List code)**
- More information here:

http://www.drdobbs.com/efficient-variable-automatic-buffers/184401740?pgno=6

Common memory/pointers bugs

bug 1

```
(1)   typedef struct _Student {
(2)        int id;
(3)        char * name;
(4)   } Student;

(5)   Student * stud =
        (Student *) malloc( sizeof(Student) );
(6)   stud->id = 123456;
(7)   stud->name =
        (char *) malloc(100*sizeof(char));
      …
(8)   free(stud);
```

Memory leak of name!

bug 2

```
1)  void myFunc() {
2)    int * x = randomNumPtr();
3)    int result = *x; //unexpected !
4)    *x = 17; //accessing unallocated space!
5)  }
6)
7)   int * randomNumPtr() {
8)         int j= srand( time(0) );
9)         return &j;
10)  }
```

Never return a address of a stack-variable !

bug 3

```
1)  void myFunc(char * input) {
2)     char * name;
3)     if (input != NULL ) {
4)           name = (char*)malloc(MAX_SIZE);
5)           strcpy(output,input);
6)     }
7)     …
8)     free( name );
9)
10) }
```

free on an address that was not allocated using malloc.

Always initialize pointers to NULL!

No bug 3

```
1)  void myFunc(char * input) {
2)     char * name= NULL;
3)     if (input != NULL ) {
4)           name = (char*)malloc(MAX_SIZE);
5)           strcpy(output,input);
6)     }
7)     …
8)     free( name );
9)
10) }
```

Initialize pointers to NULL prevents these bugs

How to duplicate a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
   const char *p1 = "hi mom",
              *p2 = (char*)malloc(strlen(p1));
   strcpy(p2,p1);
   printf("%s\n",p2);
   return 0;
}
```

How to duplicate a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    const char *p1 = "hi mom",
               *p2 = (char*)malloc(strlen(p1));
    strcpy(p2,p1);
    printf("%s\n",p2);
    return 0;
}
```

How to duplicate a string?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
  const char *p1 = "hi mom",
             *p2 = (char*)malloc(strlen(p1)+1);
  strcpy(p2,p1);
  printf("%s\n",p2);
  return 0;
}
```

strdup: duplicate a string (part of the standard library)

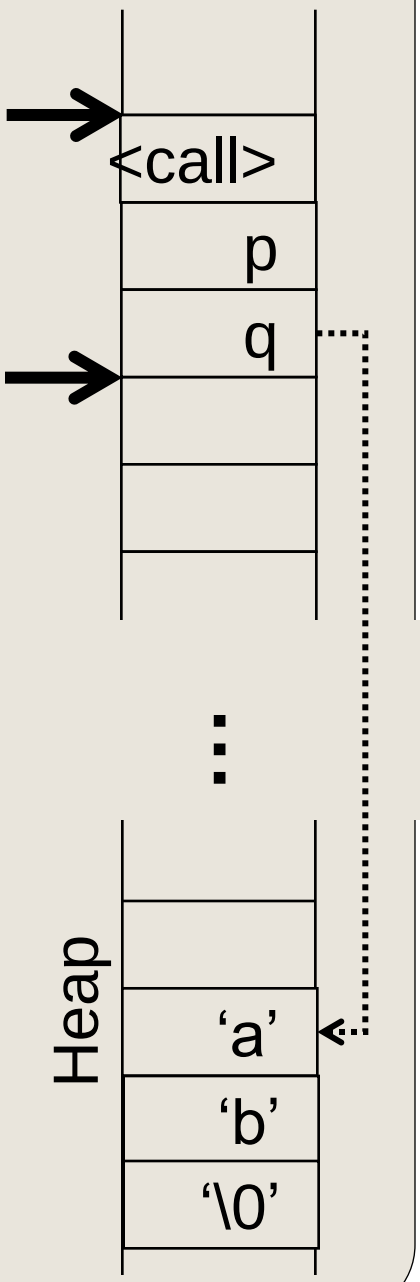
```
char* strdup( char const *p )
{
   int n = strlen(p);
   char* q =(char*)malloc(sizeof(char)*(n+1));
   if( q != NULL )
   {
      strcpy( q, p );
   }
   return q;
}
```

Memory Management

```
void
foo( char const* p )
{
  char *q;
  q = strdup( p );

  // do something with q

}
```



The allocated memory remains in use
cannot be reused later on

Memory Management

```
void
foo( char const* p )
{
  char *q = strdup( p );

  // do something with q
  free(q);
}
```

Now the memory is being freed.