Class 12: Memory Management

Elaine Li (efl9013@nyu.edu)
Office Hours Fridays 1pm – 2pm

Nisarg Patel (nisarg@nyu.edu)
Office Hours Monday 11am – 12pm

Recap: program memory

- Program memory:
  - Data segment: static objects, read-only
    - Global variables
    - Program text
    - Compiler tables
  - Stack: objects needed by activation records
    - Local variables
    - Arguments and return values
    - Return address
    - Pointer to the stack frame of the caller
  - Heap: objects with dynamic size or lifetime
    - Resizable arrays
    - Function closures
- Stack vs. heap:
  - Stack is LIFO, heap requires explicit memory management
  - Accessing data from the heap is slower than from the stack

Heap memory management

- Two broad paradigms:
    - **Manual** memory management by the programmer,
    - **Automatic** memory management by the language runtime environment
- Manual memory management is often buggy!
    - Use after free errors (accessing dangling pointers)
    - Double free errors (deallocating objects multiple times)
    - Memory leaks (not deallocating objects)
- Automatic memory management techniques:
    - Garbage collection
    - Reference counting
    - Ownership types

Garbage Collection

- Broadly refers to a class of algorithms for **automatic memory deallocation**
- Common variants:
  - Mark/sweep: compacting, non-recursive
  - Copying: incremental, generational
- Goal of garbage collection: deallocate objects that are no longer in use, i.e. dead

Garbage Collection

- An object x is live if:
  - x is pointed to by either a global variable or a variable on the stack
  - there is a register that points to x
  - there is another object on the heap that is live and points to x
- All live objects can be found via graph traversal:
  - Start at the roots:
    - Local variables on the stack
    - Global variables
    - Registers
  - Any object not reachable from the roots is dead and can be deallocated

Garbage Collection: Mark/Sweep

- Each object is marked with an extra bit
- Mark phase: collector traverses the heap and sets the mark bit of each live object found
- Sweep phase: add all objects whose mark bit is not set to a free list

```
GC()
  for each root pointer p do
    mark(p);
  sweep();

mark(p)
  if p->mark != 1 then
    p->mark = 1;
    for each pointer field p->x do
      mark(p->x);

sweep()
  for each object x in heap do
    if x.mark == 0 then insert(x, free_list);
                   else x.mark = 0;
```

Garbage Collection: Copying

- Heap is split into a FROM space and a TO space; memory allocation only happens in FROM space
- When FROM space is full, invoke garbage collection
- During graph traversal, move each live object to TO space; flip the two spaces
- Moving objects is done via forwarding addresses

```
GC()
  for each root pointer p do
    p = traverse(p);

traverse(p)
  if *p contains forwarding address then
    p = *p;  // follow forwarding address
    return p;
  else
    new_p = copy (p, TO_SPACE);
    *p = new_p; // write forwarding address
    for each pointer field p->x do
      new_p->x = traverse(p->x);
    return new_p;
```

Garbage Collection: Generational Copying

- Observation: the older an object gets, the longer it is expected to stay around
  - Many objects are very short lived (e.g. intermediate values)
  - Objects that are live for a long time tend to be key data structures in the program
- Idea: replace 2 heaps with many heaps, one for each “generation”
  - Younger generations are collected more frequently than older generations
  - During generation traversal, move live objects to the next older generation
  - When a generation is full, invoke garbage collection

Garbage Collection: Disadvantages

- Explicit invocation of garbage collection halts or slows down the program
- Garbage collected languages are unusable for real-time applications with strict timing guarantees, e.g. embedded controllers, video games
- Alternative idea: split up the work of garbage collection into smaller tasks
- Enter: reference counting

Reference Counting

- Idea: keep track of how many references point to an object
  - Initialize count to 1 for newly created objects
  - Increment counter whenever the pointer is copied
  - Decrement counter whenever a pointer goes out of scope or stops pointing to the object
  - Deallocate objects whose counter value hits 0

Reference Counting: Smart Pointers

- Reference counting is commonly implemented in smart pointers
  - Example implementation: https://github.com/nyu-pl-fa22/class12
- Design choice: counters are stored as separate objects on the heap, each pointer stores a reference to the counter for the object it is pointing to
  - Avoids duplicated counter bookkeeping for multiple pointers pointing to the same object
  - Easy to access

Reference Counting: Smart Pointers

- Constructor

```
Ptr(T* _addr = 0) : addr(_addr), counter(new size_t(1)) {}
```

- Copy

```
Ptr(const Ptr<T>& other) : addr(other.addr), counter(other.counter) {
  ++(*counter);
}
```

Reference Counting: Smart Pointers

- Assignment

```
Ptr& operator=(const Ptr& right) {
  if (addr != right.addr) {
    if (0 == --(*counter)) {
      delete addr;
      delete counter;
    }
    addr = right.addr;
    counter = right.counter;
    ++(*counter);
  }
  return *this;
}
```

Reference Counting: Smart Pointers

- Destructor

```
~Ptr() {
  if (0 == --(*counter)) {
    delete addr;
    delete counter;
  }
}
```

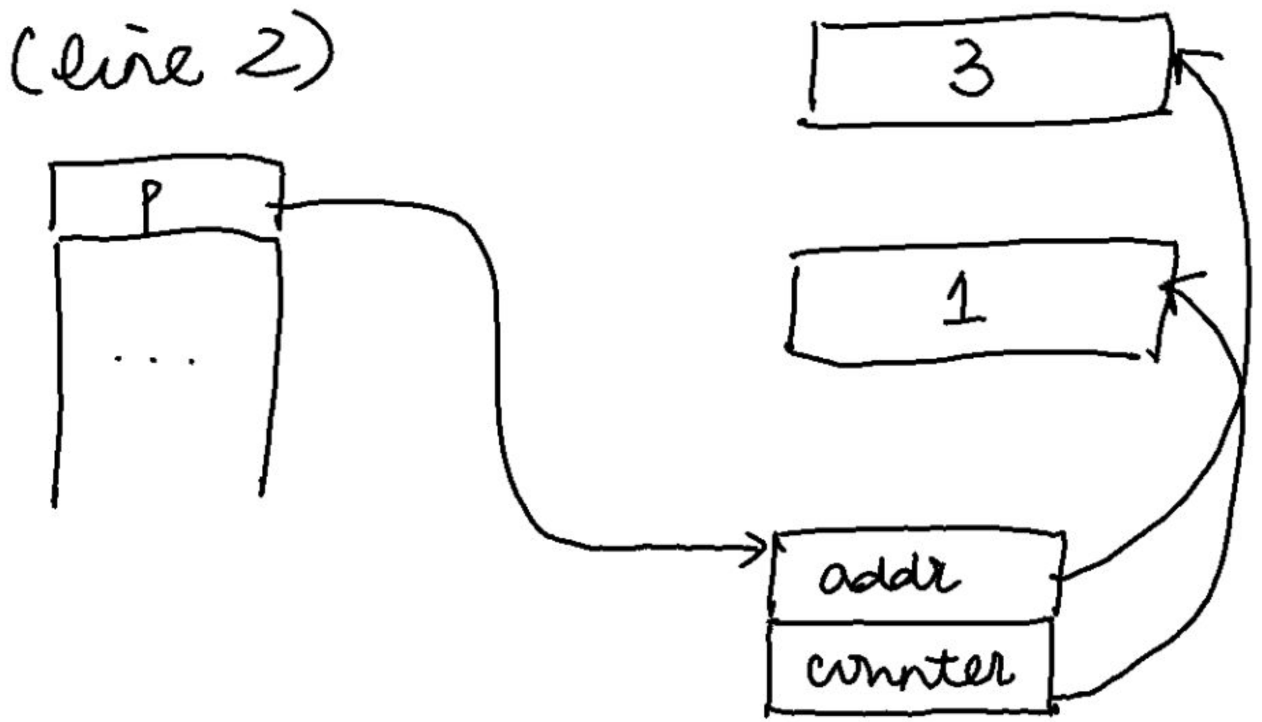
Reference Counting: Smart Pointers Example

```
{
    Ptr<int> p = new int(3);
    Ptr<int> q = p;
    cout << *q << endl;
}
```

Reference Counting: Smart Pointers example

```
2:    Ptr<int> p = new int(3);
```

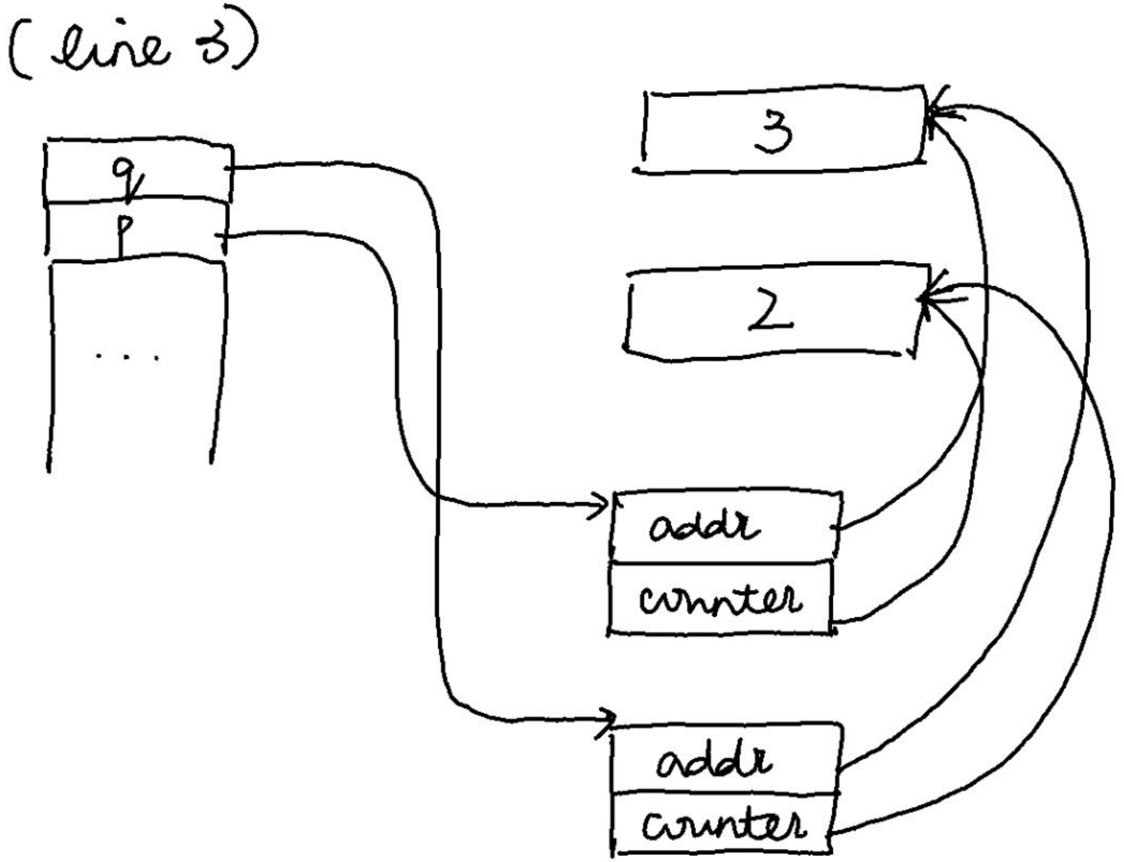
- Line 2:
- Allocate int on the heap, initialized to 3
- Call constructor Ptr(int*) to create p on the stack, with argument as the address of newly initialized int



Reference Counting: Smart Pointers example

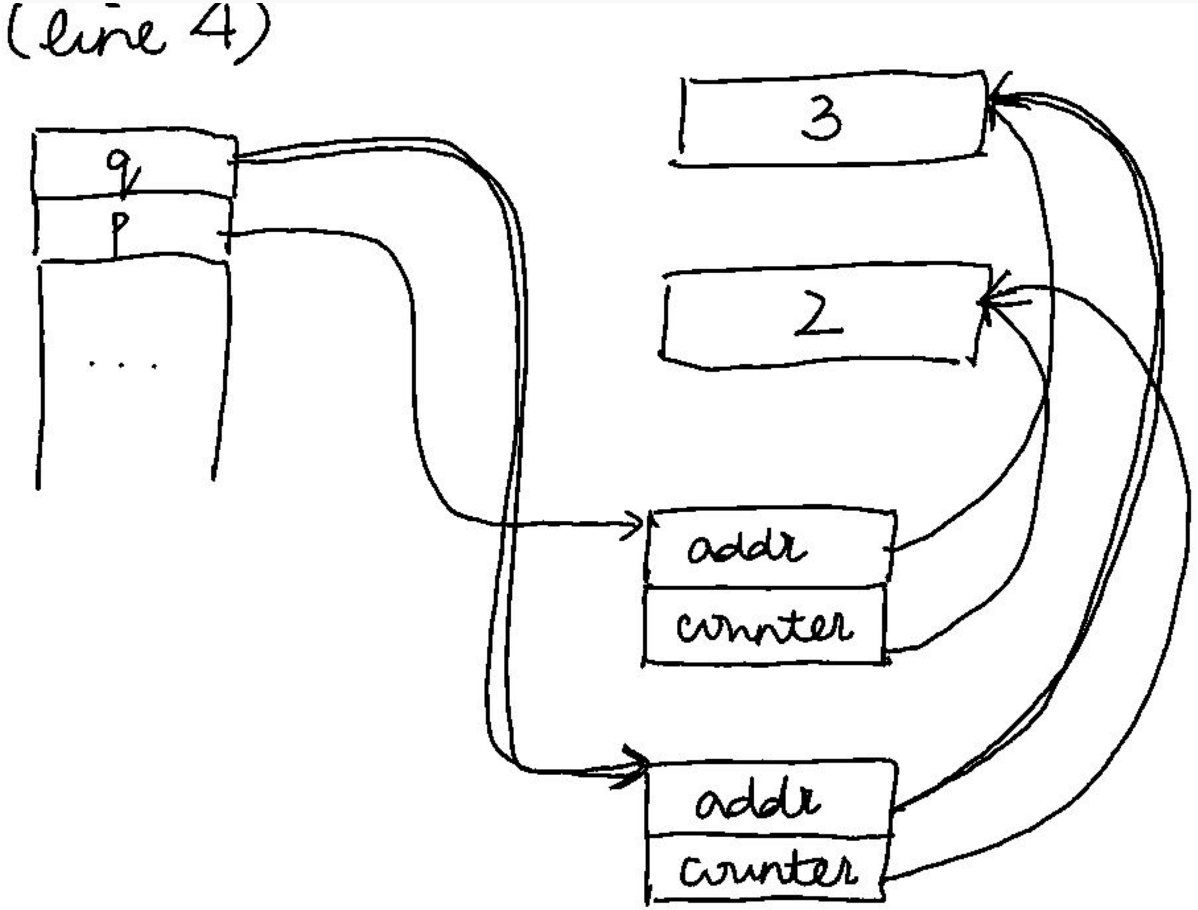
- Line 3:
- Call constructor Ptr(const &Ptr<int>) to create q on the stack, with argument p

```
3:    Ptr<int> q = p;
```



Reference Counting: Smart Pointers example

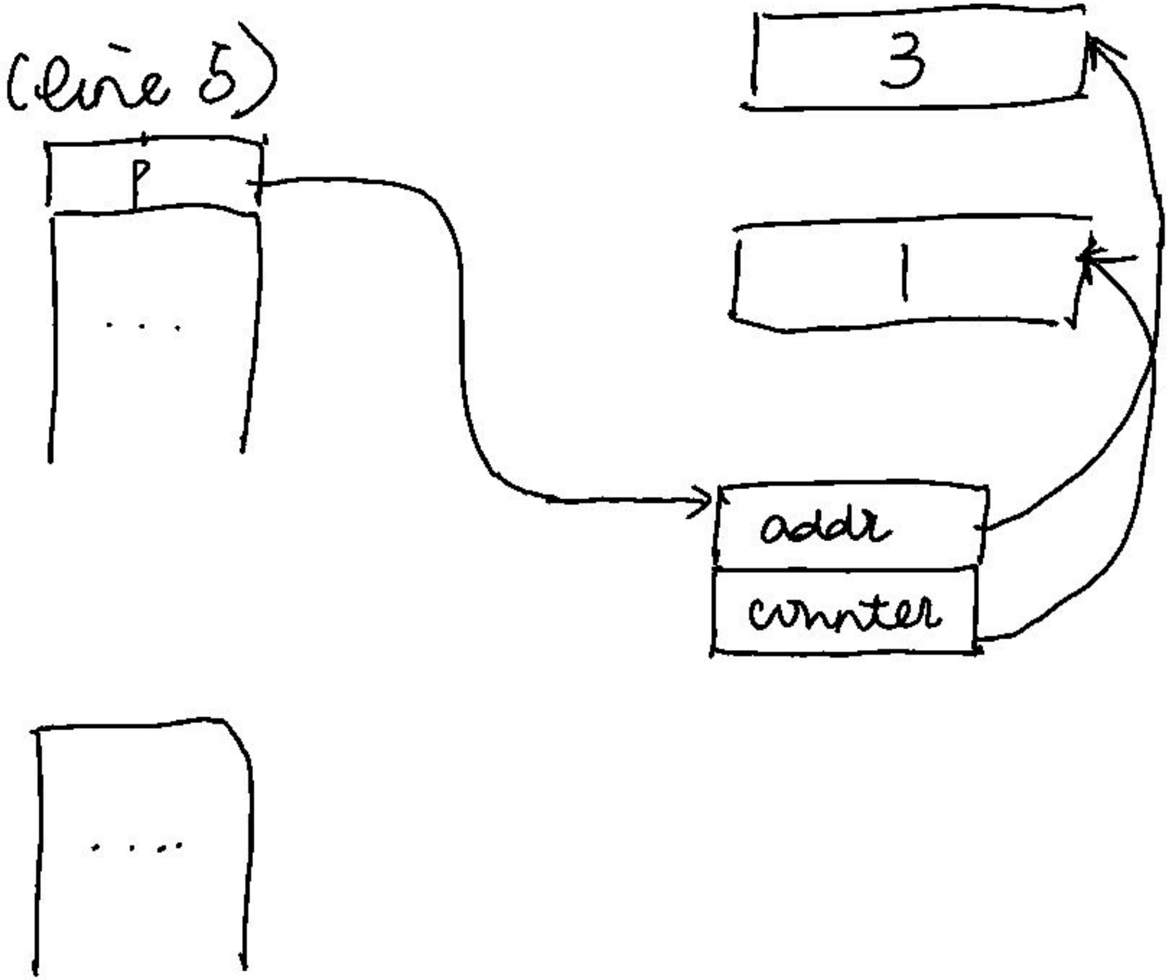
```
4:    cout << *q << endl;
```



- Line 4:
- Call Ptr::operator*
- Return value is set to value pointed to by field addr of p, 3
- Pass return value to cout, print 3

Reference Counting: Smart Pointers example

- Line 5:
- Both p and q go out of scope
- ~Ptr is first called for q, counter value is decremented to 1
- ~Ptr destructor is then called for p, counter value is decremented to 0, object and smart pointer are deallocated



```
10-19-45-56:class12 elaineli$ ./ptr_test
main:32:declaration of p
Ptr:23:0x600000fdc030
Printing counter value: 1
main:33:declaration of q
Ptr:23:0x0
Printing counter value: 1
main:34:assignment of q
operator=:49:0x0
Printing counter value: 2
main:35:dereferencing q
operator*:65:0x600000fdc030
Printing counter value: 2
5
main:36:return
~Ptr:38:0x600000fdc030
Printing counter value: 1
~Ptr:38:0x600000fdc030
Count equals zero, dereferencing: 0
Printing counter value: 19006602351476
```

```
21 #include <iostream>
22 #include "ptr.h"
23
24 using namespace std;
25
26 struct Node {
27   Ptr<Node> next;
28 };
29
30 int main(void) {
31   // Example 1:
32   TRACE("declaration of p");          Ptr<int> p = new int(5
33   TRACE("declaration of q");          Ptr<int> q;
34   TRACE("assignment of q");           q = p;
35   TRACE("dereferencing q");           cout << *q << endl;
36   TRACE("return");                    return 0;
37 }
```

Reference Counting: Advantages and Disadvantages

- Advantages:
  - Distributes the work of garbage collection, reduces overhead
  - Can be implemented by programmer without reliance on programming language runtime environment
- Disadvantages:
  - Requires additional space for keeping track of reference count
  - Non-zero runtime overhead
  - Does not work on circular reference structures!

Reference Counting: Circular References

- At line 7, the counter associated with object p is set to 2: one reference from p and another reference from p’s next field
- When p goes out of scope in line 8 and the destructor is called, the counter is decremented to 1
- Memory leak!

```
1: struct Node {
2:     Ptr<Node> next;
3: };
4:
5: {
6:     Ptr<Node> p = new Node();
7:     p->next = p;
8: }
```

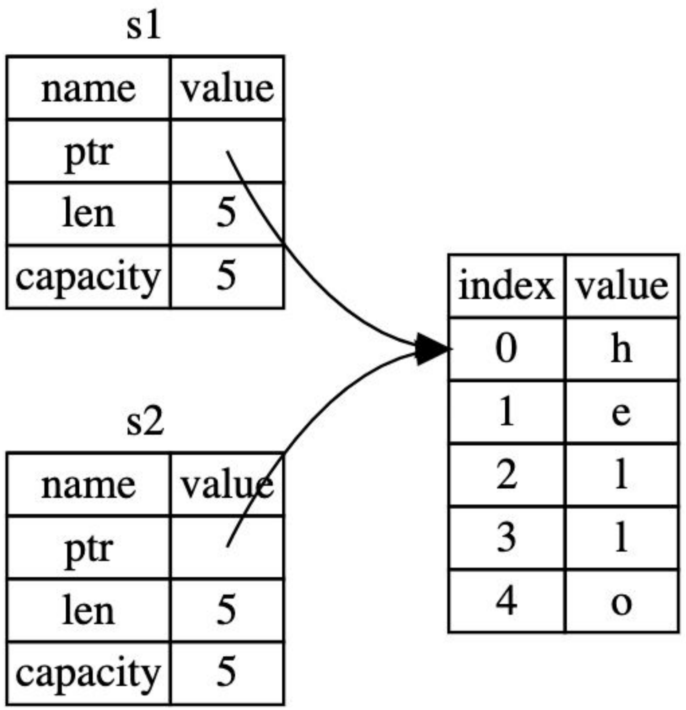
Ownership Types: Rust

- Objects are owned by a single variable at a time
- An object is deallocated when the variable that owns it go out of scope
- Statically eliminates double free errors

Ownership Types: Rust

- What happens when two pointers reference the same object?

```
let s1 = String::from("hello");
let s2 = s1;
```

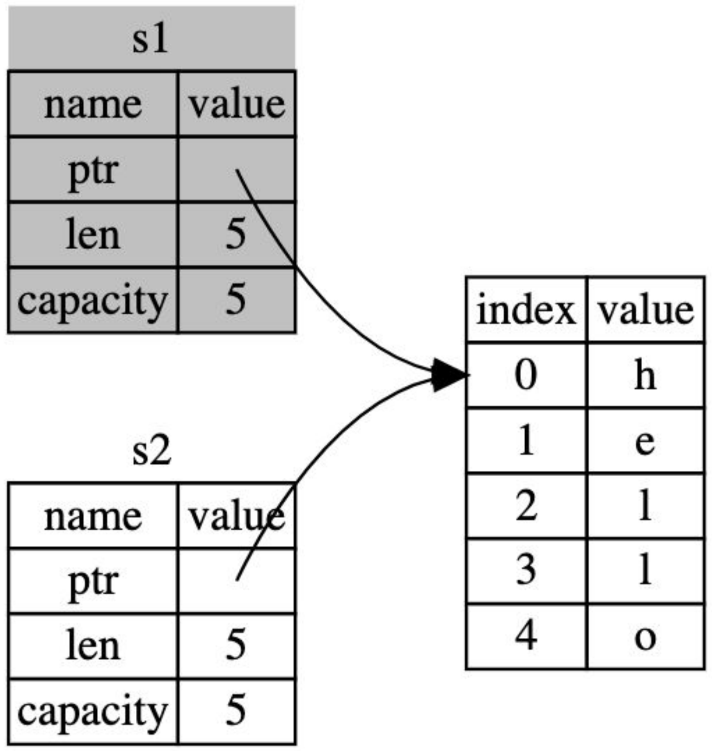


Ownership Types: Rust

- What happens when two pointers reference the same object?

```
let s1 = String::from("hello");
let s2 = s1;
```

- Ownership of s1 is moved to s2
- s1 becomes invalid



Ownership Types: Rust

- What happens to a variable when passed as an argument to a function?
- The ownership of heap objects are moved when passed as function arguments, i.e. s
- The ownership of stack objects are not moved when passed as function arguments
  - Integer, boolean types
  - Floating points, chars
  - Tuples containing the above

```
let s = String::from("hello");

takes_ownership(s);


let x = 5;

makes_copy(x);
```