Ownership

Edward Z. Yang
The plan:

- What's the problem?
  - Manual memory management

- C++
  - Unique pointers and ownership
  - Rules of ownership and borrowing

- Rust
  - Statically checked ownership
Last time...

infinite memory

refcounting / garbage collection

finite memory
Last time...

but there are tradeoffs!

atomic incref/decref, pauses

infinite memory

refcounting / garbage collection

finite memory
I DON'T ALWAYS MANUALLY MANAGE MEMORY

BUT WHEN I DO, I USE OWNERSHIP
What is manual memory management?

Memory management on the stack:

```c
void f() {
    int x[20];
    // array is available
}
```

X when scope exits, array is freed
What is manual memory management?

Memory management on the stack:

```c
void f() {
    int x[20];
}
```

- array is available

- what if I want the array to live on?
Memory management on the heap

In C:

```c
int* p = (int*) malloc(sizeof(int) * 4);
// do some stuff to p
free(p);
```

In C++:

```cpp
A* p = new A();
// do some stuff to p
delete p;
```
Memory management on the heap

\[ A^* \ p = \ \text{foobar}(\); \]

// do some stuff with \( p \)

ok... now what?
inline void
THPUtils_packInt64Array(
    PyObject *tuple,
    size_t size,
    const int64_t *sizes
) {
    for (size_t i = 0; i != size; ++i) {
        PyObject *i64 = THPUtils_packInt64(sizes[i]);
        if (!i64) {
            throw python_error();
        }
        PyTuple_SET_ITEM(tuple, i,
            THPUtils_packInt64(sizes[i]));
    }
}

Memory management on the heap

spot the bug...
inline void
THPUtils_packInt64Array(
    PyObject *tuple,
    size_t size,
    const int64_t *sizes
) {
    for (size_t i = 0; i != size; ++i) {
        PyObject *i64 = THPUtils_packInt64(sizes[i]);
        if (!i64) {
            throw python_error();
        }
        PyTuple_SET_ITEM(tuple, i,
            THPUtils_packInt64(sizes[i]));
    }
}
inline void THPUtils_packInt64Array(
    PyObject *tuple,
    size_t size,
    const int64_t *sizes)
{
    for (size_t i = 0; i != size; ++i) {
        PyObject *i64 = THPUtils_packInt64(sizes[i]);
        if (!i64) {
            throw python_error();
        }
        PyTuple_SET_ITEM(tuple, i,
            THPUtils_packInt64(sizes[i]));
    }
}

Memory management on the heap

what about this one?

spot the bug...
Bug farm

- I have a pointer: should I destroy it when I'm done using it?
- How should I destroy a pointer when I'm done with it?
- Did I delete exactly once in all codepaths?
- Did I already free this pointer? (Is it dangling?)
Hierarchy of Safety

Rust

Provably safe

C++

"C++11 can help you... ...a little, anyway"

C

"You're on your own, buster!"
What is ownership? It's a set of rules

Every allocation has a unique owner

The owner is responsible for deallocating memory when it is done using it.
```c
inline void
THPUtils_packInt64Array(
    PyObject *tuple,
    size_t size,
    const int64_t *sizes
) {
    for (size_t i = 0; i != size; ++i) {
        PyObject *i64 = THPUtils_packInt64(sizes[i]);
        if (!i64) {
            throw python_error();
        }
        PyTuple_SET_ITEM(tuple, i, THPUtils_packInt64(sizes[i]));
    }
}
```
Ownership is everywhere

(P.S. PyObject is refcounted: still has notion of ownership for incref/decref)
Changed in version 2.5: This function returned an int type. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GetItem(PyObject* p, Py_ssize_t pos)

Return value: Borrowed reference.
Return the object at position pos in the tuple pointed to by p. If pos is out of range, return NULL and sets an IndexError exception.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GET_ITEM(PyObject* p, Py_ssize_t pos)

Return value: Borrowed reference
Like PyTuple_GetItem(), but does no checking of its arguments.

Changed in version 2.5: This function used an int type for pos. This might require changes in your code for properly supporting 64-bit systems.

PyObject* PyTuple_GetSlice(PyObject* p, Py_ssize_t low, Py_ssize_t high)

Take a slice of the tuple pointed to by p from low to high and return it as a new object.

Changed in version 2.5: This function used an int type for low and high. This might require changes in your code for properly supporting 64-bit systems.

int PyTuple_SetItem(PyObject* p, Py_ssize_t pos, PyObject* o)

Insert a reference to object o at position pos of the tuple pointed to by p.
PyObject *p = alloc_obj();
borrow_obj(p);
Py_CLEAR(p);

(borrow function is not responsible for freeing)
C: you still have to do everything by hand: no compiler help.

Rust

Provably safe

C++

"C++11 can help you... ...a little, anyway"

C

"You're on your own, buster!"
C++11 to the rescue: smart pointers

std::shared_ptr<T>
std::unique_ptr<T>

const T&
T*

owning pointers
non-owning references
C

T* allocate();
void borrow(T*);
void free(T*);

Basics

T* p = allocate();
borrow(p);
free(p);  \(\text{still own } p\)
C++11

`std::unique_ptr<T> allocate();`
`void borrow(T*);`
`void free(T*);`

Basics

`auto p = allocate();`
`borrow(p.get());`

(automatically freed)

How? `std::unique_ptr` is itself an object allocated on the stack! (RAII)
C

T* allocate();
void steal(T*);

Ownership transfer

T* p = allocate();
steal(p);

(we no longer own p; we must not use it, and must not free it!)
C++11

```cpp
std::unique_ptr<T> allocate();
void steal(
    std::unique_ptr<T> );
```

compare with:

```cpp
void borrow(T*);
```

C different!

Ownership transfer

```cpp
auto p = allocate();
steal(std::move(p));
// p is now nullptr!
```

The exact mechanism by which this works is complex; it involves implicit move construction of a new unique_ptr. This was not added until C++11.
C++11 benefits

- Automatic disposal of owning pointers (e.g., `unique_ptr`) when they exit scope

- Type level distinction between owning and non-owning pointers
C++11: The ugly bits

```c++
#include <memory>

struct T {
    int x = 0;
};

int main() {
    auto* p = std::make_unique<T>().get();
    p->x = 2;
    return 0;
}
```

↑ is this OK?
no complaints from the compiler...
no complaints even with warnings!

```
ezyang@sabre:~$ gcc test.cpp -std=c++14 -fsanitize=address
ezyang@sabre:~$ ./a.out
=================================================================
==10991==ERROR: AddressSanitizer: heap-use-after-free
 on address 0x60200000eff0 at pc 0x0000004009cc
 bp 0x7ffcdb7b1090 sp 0x7ffcdb7b1080
WRITE of size 4 at 0x60200000eff0 thread T0
  #0 0x4009cb in main (/home/ezyang/a.out+0x4009cb)
  #1 0x7fbf7f20282f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
  #2 0x400888 in _start (/home/ezyang/a.out+0x400888)
```
C++11: The ugly bits

Automatic disposal and type-system encoded ownership are all you get.

You MUST ensure that borrows don't extend beyond lifetime of owning pointer.
```cpp
#include <memory>

struct T {
    int x = 0;
};

int main() {
    auto* p = std::make_unique<T>().get();
    p->x = 2;
    return 0;
}
```
It can be non-obvious

class Foo {
    Foo(const Foo&); // copy-constructor
    Field* mutable_field();
}

class Field {
    void set_int(int);
}

const Foo& borrow_foo();
void run_with_borrow(Field*);

void run_with_modified_copy_of_field() {
    auto* field = Foo(borrow_foo()).mutable_field();
    field->set_int(23);
    run_with_borrow(field);
}
C++: you still have to get all the borrows right

Rust

Provably safe

C++

"C++11 can help you... a little, anyway"

C

"You're on your own, buster!"
Rust
Rust

- Conceptually, same principles as C++
  - No mutable aliases: fearless concurrency

- All borrows checked by the borrow checker

- Some programs not expressible in Safe Rust (doubly-linked list)
  → unsafe escape hatch
Rust: safe, but less programs allowed!

Rust
Provably safe

C++
“C++11 can help you... ...a little, anyway”

C
“You’re on your own, buster!”