2014 CEE-SEC(R) Software Engineering Conference in Russia

Precise Garbage Collection for C++ with a Non-Cooperative Compiler

Daniil Berezun

JetBrains

Daniil.Berezun@jetbrains.com

23.10.2014

Context

C++:

- Manual memory management;
- Fine-tune using unsafe primitives.

Sometimes GC is desirable:

- Non-time-critical components;
- Library interface simplification;
- Safety requirement.

Library-based approach advantages:

- Portability;
- Compiler independence;
- Separation of managed and non-managed parts.

Garbage Collection

Reference counting:

- each pointer has a counter;
- pointer operation overhead;
- circular references is a problem;
- unpredictable deallocation time.

Tracing garbage collection:

- ”useful” objects;
- root set;
- reachability;
- precise / conservative.

Conservative vs. Precise GC

Precise:

- precise pointer identification;
- can reclaim all unreachable memory.
- safe.

Conservative: (e.g. Boehm GC)

- heuristic pointer identification;
- disadvantages:
  - compactification cannot be implemented;
  - some ”dead” objects may not be collected;
  - unsafe.

Our GC Features

- Precise;
- Safety: nothing can “go wrong” because of the GC;
- No compiler cooperation is needed;
- Managed and unmaneged objects can coexist.

Library Interface — Key Primitives

- Smart pointer class gc_ptr:

```
template <class T> class gc_ptr { ... };
```

- Memory allocation template function gc_new:

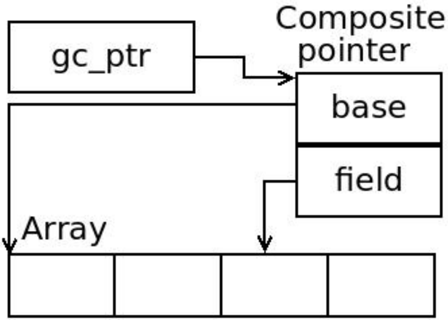
```
template <class T , typename ... Types>
gc_ptr<T> gc_new (Types ... types, size_t count = 1);
```

```
Class * element = new Class (a1, ... , an);
// --- replaced with --->
gc_ptr<Class> element =
  gc_new<Class, T1, ... , Tn>(a1, ... , an);
```

Library Interface — Other Features

- Pointers ”inside” objects:

```
template <typename F , typename B>
gc_ptr<F> derive (const gc_ptr<B> base, const F * field);
```



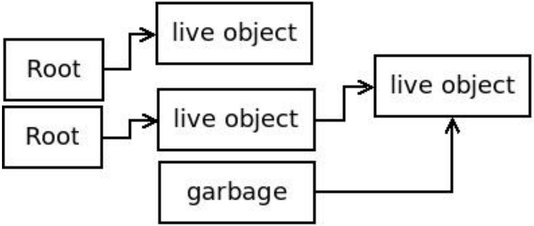
Library Interface — Non-managed Objects

Managed and unmanaged objects can coexist:

- References from managed to non-managed objects — OK;
- References from non-managed to managed objects require user assistance:
  1. `void register_object (void *);`
  2. `void unregister_object (void *).`

```
struct str1 {
  gc_ptr<char> p;
};
// ...
str1 * s = (str1 *) malloc (sizeof(str1));
s->p = gc_new<char>(10);
register_object(s->p);
```

Implementation — Overview



mark and sweep

Solved problems:

- Root set identification;
- Constructing and maintaining meta-information;
- Implementing mark-and-sweep phase.

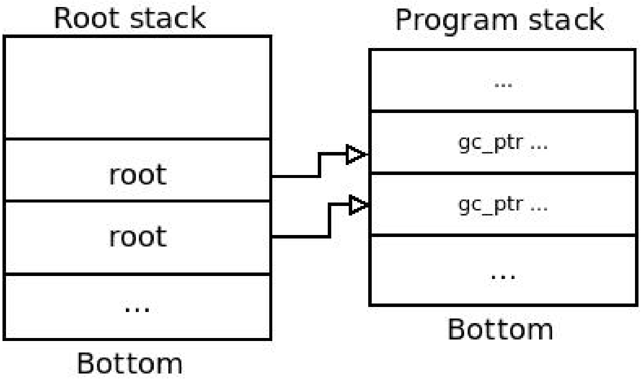
Implementation — Cooperative Heap

- Cooperative heap (Doug Lea's malloc):

  1. Distinguish heap pointers from non-heap;
  2. Implement efficient sweep phase;
  3. Maintain mark bit and managed bit.

Root Set

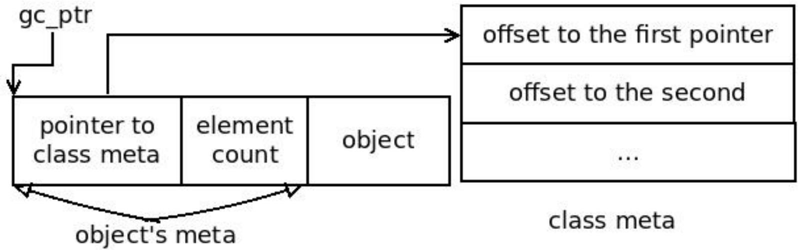
- gc_ptr maintains root set via its constructor / destructor.



Meta-information

Meta-information:

1. class meta;
2. object meta.



Properties

- Precise;
- Safety;
- No compiler cooperation;
- Managed and unmaneged objects can coexist.

Demonstaration

Limitations

- 64-bit Linux;
- Single-thread;
- Not thread-safety;
- Time overhead up to an order.

Future Work

- Port to another platforms;
- Thread-safety;
- Other garbage collection algorithms;
- Minimize overhead;
- Memory leaks detector.