COMP 737011 - Memory Safety and Programming Language Design

Lecture 5: Auto Memory Reclaim

XU, Hui
xuh@fudan.edu.cn

Auto Reclaim Challenge

- Memory units for local data allocated on stack are automatically reclaimed when a function returns
- Heap is hard to be reclaimed automatically
  - There could be multiple references across functions
  - Pointer analysis is NP-hard in general

Outline

- 1. Compile-time Approach
- 2. Smart Pointers
- 3. Garbage Collection

1. Compile-time Approach

Cleanup Attribute

- Set a cleanup function to be executed before the function returns
- Ineffective if an exception occurs

```
void free_buffer(char **buffer) {
    printf("Freeing buffer\n");
    free(*buffer);
}

void toy() {
    char *buf __attribute__ ((__cleanup__(free_buffer))) = malloc(10);
    snprintf(buf, 10, "%s", "any chars");
    printf("Buffer: %s\n", buf);
}
```

```
0x00000000004011a0 <+0>:     push   rbp
...
0x00000000004011ed <+77>:    call   0x401040 <printf@plt>
0x00000000004011f2 <+82>:    lea    rdi,[rbp-0x8]
0x00000000004011f6 <+86>:    mov    DWORD PTR [rbp-0x10],eax
0x00000000004011f9 <+89>:    call   0x401160 <free_buffer>
0x00000000004011fe <+94>:    add    rsp,0x10
0x0000000000401202 <+98>:    pop    rbp
0x0000000000401203 <+99>:    ret
```

C++ Auto Destruction

- Execute the destuctor of objects on the stack automatically

```
class MyClass{
  private:
    int id;
  public:
    MyClass(int v) { id = v; }
    ~MyClass() { cout << "delete:"<< id << endl; }
};

void toy() {
    MyClass c1 = MyClass(100);
    MyClass* c2 = new MyClass(200);
}
```

```
./a.out
delete:100
```

C++ Auto Destruction: Assembly Code

```
0x0000000000401250 <+0>:     push   rbx
0x0000000000401251 <+1>:     sub    rsp,0x10
0x0000000000401255 <+5>:     lea    rdi,[rsp+0x8]
0x000000000040125a <+10>:    mov    esi,0x64
0x000000000040125f <+15>:    call   0x4012b0 <_ZN7MyClassC2Ei>
0x0000000000401264 <+20>:    mov    edi,0x4
0x0000000000401269 <+25>:    call   0x401090 <_Znwm@plt>
0x000000000040126e <+30>:    mov    rdi,rax
0x0000000000401271 <+33>:    mov    esi,0xc8
0x0000000000401276 <+38>:    call   0x4012b0 <_ZN7MyClassC2Ei>
0x000000000040127b <+43>:    lea    rdi,[rsp+0x8]
0x0000000000401280 <+48>:    call   0x4012c0 <_ZN7MyClassD2Ev>
0x0000000000401285 <+53>:    add    rsp,0x10
0x0000000000401289 <+57>:    pop    rbx
0x000000000040128a <+58>:    ret
0x000000000040128b <+59>:    mov    rbx,rax
0x000000000040128e <+62>:    lea    rdi,[rsp+0x8]
0x0000000000401293 <+67>:    call   0x4012c0 <_ZN7MyClassD2Ev>
0x0000000000401298 <+72>:    mov    rdi,rbx
0x000000000040129b <+75>:    call   0x401100 <_Unwind_Resume@plt>
```

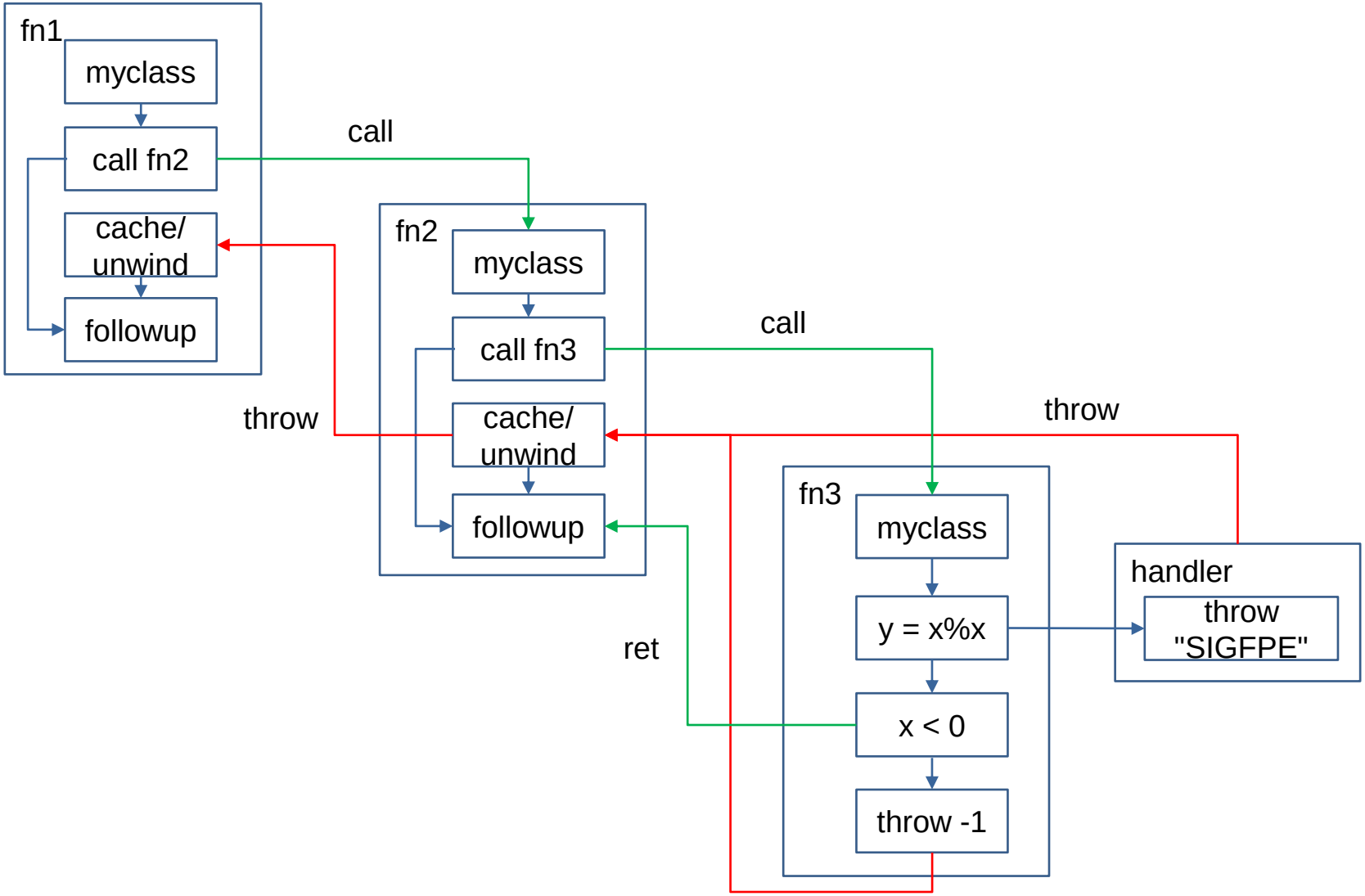
A More Complicated Example

```
void fn3(int x) {
    MyClass c3 = MyClass{300};
    double y = x%x;
    if(x < 0) throw -1;
}
void fn2(int x) {
    MyClass c2 = MyClass{200};
    try{
        fn3(x);
    }catch (const int msg) {
        cout << "Land in fn2:"
             << msg << endl;
    }
}
void fn1(int x) {
    MyClass c1 = MyClass{100};
    try{
        fn2(x);
    }catch (const char* msg) {
         cout << "Land in fn1:"
              << msg << endl;
    }
}
```

```
void handler(int signal) {
    throw "SIGFPE Received!!!";
}

int main(int argc, char** argv) {
    signal(SIGFPE, handler);
    int x;
    scanf("%d", &x);
    fn1(x);
}
```

Inter-procedural CFG



Execution Result

```
#: ./a.out
0
delete:200
Land in fn1: SIGFPE Received!!!
delete:100
#: ./a.out
-1
delete:300
Land in fn2:-1
delete:200
delete:100
```

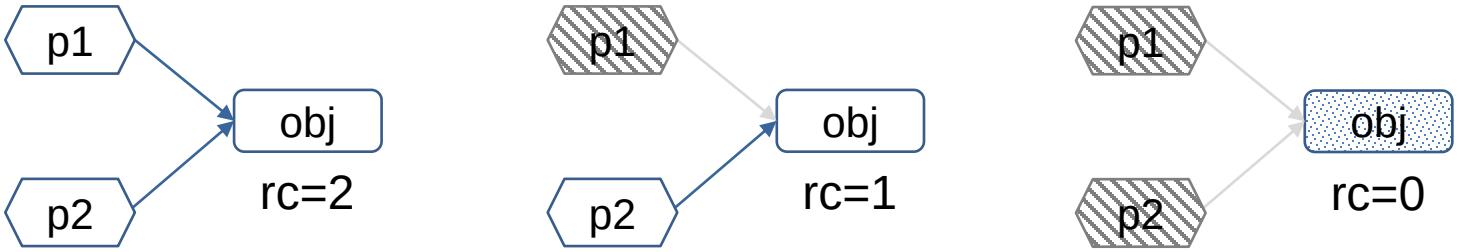
Landing Pad: Check gcc_except_tables

```
#: clang++ -S unwind-reclaim.cpp
#: cat unwind-reclaim.s
...
GCC_except_table5:
.Lexception2:
        .byte   255                     # @LPStart Encoding = omit
        .byte   3                       # @TType Encoding = udata4
        .uleb128 .Lttbase1-.Lttbaseref1
.Lttbaseref1:
        .byte   1                       # Call site Encoding = uleb128
        .uleb128 .Lcst_end2-.Lcst_begin2
.Lcst_begin2:
        .uleb128 .Lfunc_begin2-.Lfunc_begin2 # >> Call Site 1 <<
        .uleb128 .Ltmp13-.Lfunc_begin2  #   Call between .Lfunc_begin2 and .Ltmp13
        .byte   0                       #     has no landing pad
        .byte   0                       #   On action: cleanup
        .uleb128 .Ltmp13-.Lfunc_begin2  # >> Call Site 2 <<
        .uleb128 .Ltmp14-.Ltmp13        #   Call between .Ltmp13 and .Ltmp14
        .uleb128 .Ltmp15-.Lfunc_begin2  #     jumps to .Ltmp15
        .byte   0                       #   On action: cleanup
        .uleb128 .Ltmp16-.Lfunc_begin2  # >> Call Site 3 <<
        .uleb128 .Ltmp17-.Ltmp16        #   Call between .Ltmp16 and .Ltmp17
        .uleb128 .Ltmp18-.Lfunc_begin2  #     jumps to .Ltmp18
        .byte   3                       #   On action: 2
        .uleb128 .Ltmp17-.Lfunc_begin2  # >> Call Site 4 <<
...
```

2. Smart Pointers

Smart Pointers

- Why? Static analysis cannot handle pointers
- Dynamically track the number of object pointers
- Reclaim the memory once no variable owns it



Smart Pointer in C++: shared_ptr

- Share an object among multiple pointers with a reference counter
- Destroy the object when the last remaining shared_ptr owning the object is destroyed or reassigned

```
void toy() {
    shared_ptr<MyClass> p1(new MyClass(100));
    //cout << "Ref counter: " << p1.use_count() << endl;
    shared_ptr<MyClass> p2 = p1;
    //cout << "Ref counter: " << p1.use_count() << endl;
}
```

How to Implement shared_ptr<T>

```
0x0000000000401290 <+0>:     push   r14
0x0000000000401292 <+2>:     push   rbx
0x0000000000401293 <+3>:     sub    rsp,0x28
0x0000000000401297 <+7>:     mov    edi,0x4
0x000000000040129c <+12>:    call   0x4010a0 <_Znwm@plt>
0x00000000004012a1 <+17>:    mov    rbx,rax
0x00000000004012a4 <+20>:    mov    rdi,rax
0x00000000004012a7 <+23>:    mov    esi,0x64
0x00000000004012ac <+28>:    call   0x401380 <_ZN7MyClassC2Ei>
0x00000000004012b1 <+33>:    lea    r14,[rsp+0x18]
0x00000000004012b6 <+38>:    mov    rdi,r14
0x00000000004012b9 <+41>:    mov    rsi,rbx
0x00000000004012bc <+44>:    call   0x401390 <_ZNSt10shared_ptrI7MyClassEC2IS0_vEEPT_>
0x00000000004012c1 <+49>:    lea    rbx,[rsp+0x8]
0x00000000004012c6 <+54>:    mov    rdi,rbx
0x00000000004012c9 <+57>:    mov    rsi,r14
0x00000000004012cc <+60>:    call   0x4013a0 <_ZNSt10shared_ptrI7MyClassEC2ERKS1_>
0x00000000004012d1 <+65>:    mov    rdi,rbx
0x00000000004012d4 <+68>:    call   0x4013b0
<_ZNSt12__shared_ptrI7MyClassLN9__gnu_cxx12_Lock_policyE2EED2Ev>
0x00000000004012d9 <+73>:    mov    rdi,r14
0x00000000004012dc <+76>:    call   0x4013b0
<_ZNSt12__shared_ptrI7MyClassLN9__gnu_cxx12_Lock_policyE2EED2Ev>
0x00000000004012e1 <+81>:    add    rsp,0x28
0x00000000004012e5 <+85>:    pop    rbx
0x00000000004012e6 <+86>:    pop    r14
0x00000000004012e8 <+88>:    ret
```

Create a shared_ptr

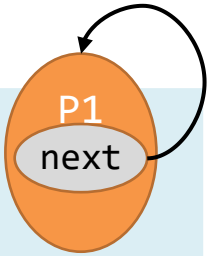
Increase the counter

Decrease the counter

Decrease the counter

Problem of Shared Pointer

- Problem of shared_ptr: reference cycles




```
class MyList{
private:
    int id;
public:
    MyList(int v) { id = v; }
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< id << endl; }
};

int main() {
    shared_ptr<MyList> p(new MyList(100));
    p->next = p;
    cout << "Ref counter: " << p.use_count() << endl;
}
```

```
#:./a.out
Ref counter: 2
```

Use weak_ptr Instead

- weak_ptr: do not update the reference counter

```
class MyList{
private:
    int id;
public:
    MyList(int v) { id = v; }
    weak_ptr<MyList> next;
    ~MyList() { cout << "delete obj:"<< id << endl; }
};

int main() {
    shared_ptr<MyList> p(new MyList(100));
    p->next = p;
    cout << "Ref counter: " << p.use_count() << endl;
}
```

```
#:./a.out
Ref counter: 1
delete obj:100
```

Smart Pointer: unique_ptr

- Object is uniquely ownd by one pointer
- Checked during compile time (similar to Rust ownership)
- User can transfer ownership through move()

```
int main() {
    unique_ptr<MyClass> p1(new MyClass(100));
    cout << "Before move: p1 = " << p1.get() << endl;
    //unique_ptr<MyClass> p2 = p1;
    unique_ptr<MyClass> p2 = move(p1);
    cout << "After move: p1 = " << p1.get() << endl;
    //cout << p1->val << endl;
    cout << p2->val << endl;
}
```

```
#:./a.out
Before move: p1 = 0x1476eb0
After move: p1 = 0
100
delete:100
```

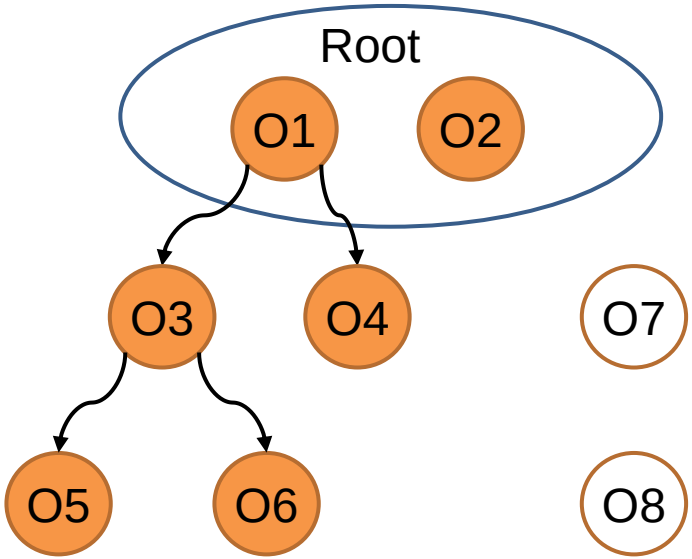
3. Garbage Collection

Garbage Collection

- When should the GC be triggered?
- Which objects should be recycled?
  - Rechability analysis
- How to recycle?
  - May cause slowdown due to intensive GC operation
  - Memory fragmentation issue

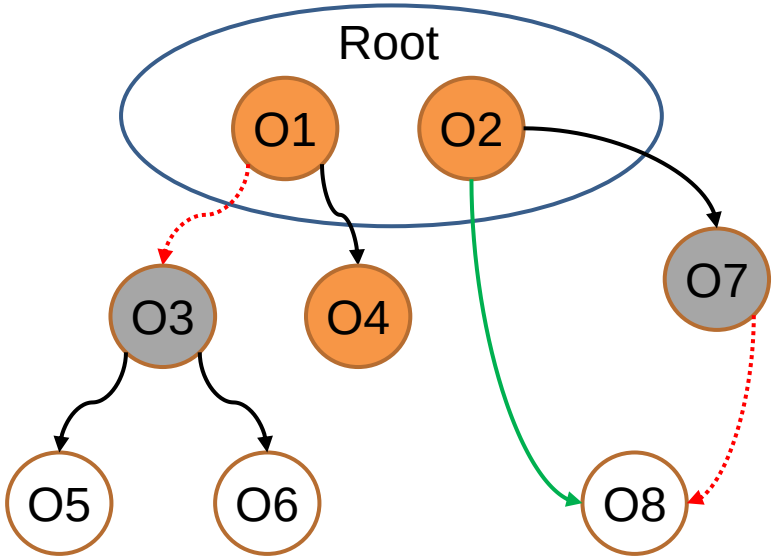
Reachability Analysis

- Stop the world
- Analyze from the root
- Unreachable objects should be recycled immediately



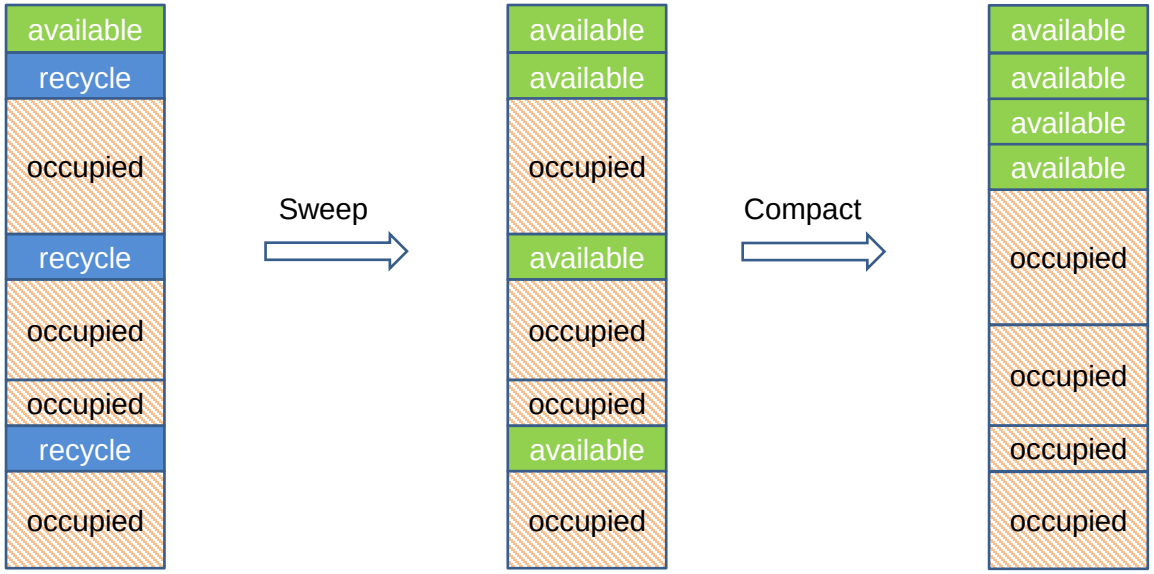
Incremental Analysis

- Do not need to stop the world
- Use three colors to record the temporary result
  - Orange: reached, and analysis (to other objects) is done
  - Gray: reached, but analysis is not finished
  - White: unreached object
- false negative?
- false positive?



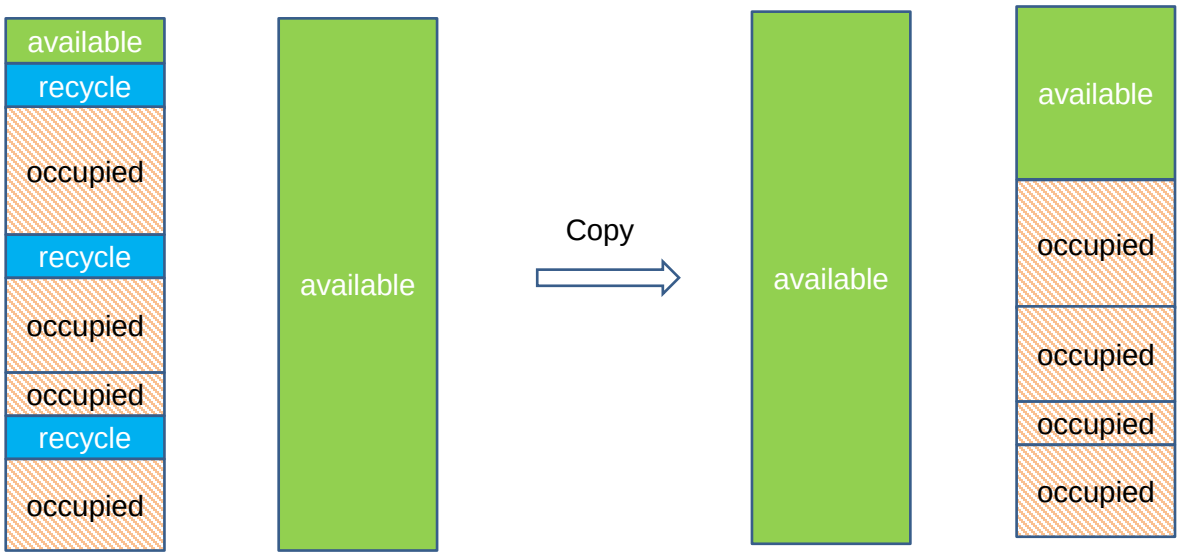
How to Recycle?

- For consecutive memory chunks (e.g., program break)
- Mark-sweep: suffers fragmentation issue
- Mark-compact: move all used units to one side
  - nontrivial overhead for moving data
  - when should the process be triggered?



Mark-Copy

- Two pieces of memory with the same size
  - the memory piece is still usable during copy
  - tradeoff between time and space

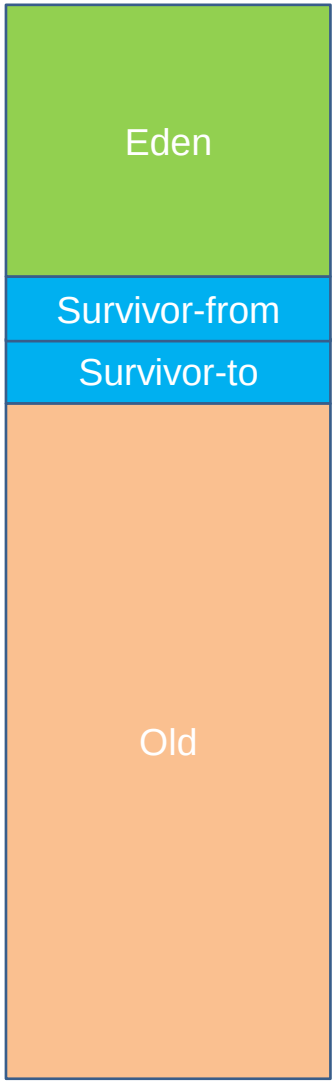


Observation

- Newly created objects tend to be recycled
- The objects survived after several GC rounds has a high chance to survive in the following round
- How can we utilize the observation for optimization?
  - Avoid frequent copy of old objects

Generational Collection

- Eden: for new objects
  - trigger minor GC if no space available
- Survivor: to host survived objects after minor GC
  - with two sub areas: from, to
  - Minor GC(eden+from)=>to,
  - Minor GC(eden+to)=>from
- Old: for objects survived after several rounds of minor GC
  - trigger major GC if no space available
  - large objects are saved to this area directly to avoid the overhead of copy.



Implementing GC for C?

- Enumerate the Root node:
  - Variables of pointer types
  - Variables of data structures with pointers
- Check unreachable objects and delete them:
  - The allocator maintains the info of all allocated chunks
  - When? Before a function returns
- More reference:
  - BoehmGC: https://www.hboehm.info/gc/#details
  - Writing a Simple Garbage Collector in C: https://maplant.com/gc.html

In-class Practice

1) Write a C++ code snipet with use after free bugs
   - You cannot use delete/free
   - Based on the auto delete or shared_ptr mechanism
2) Design a library of shared pointer features for C
   - Implement the API for create/increase/decrease