CS 131 Compilers: Discussion 13: Garbage Collection

杨易为 季杨彪 尤存翰

{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 5 月 31 日

1 C++ garbage collection

C++ actually have runtime garbage collection, once put in 2008 and deprecated in 2013. If you use some of the implementation of C++, they have runtime resource garbage collector. Now, we only talk about the compile time resource allocation.

1.1 Resource Acquisition Is Initialization

RAII, which is the compiled time garbage collector first introduced in this thread, guarantees that the resource is available to any function that may access the object (resource availability is a class invariant, eliminating redundant runtime tests). It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition. Likewise, if resource acquisition fails (the constructor exits with an exception), all resources acquired by every fully-constructed member and base subobject are released in reverse order of initialization. This leverages the core language features (object lifetime, scope exit, order of initialization and stack unwinding) to eliminate resource leaks and guarantee exception safety. Another name for this technique is Scope-Bound Resource Management (SBRM), after the basic use case where the lifetime of an RAII object ends due to scope exit.

RAII can be summarized as follows:

1. encapsulate each resource into a class, where
   (a) the constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
   (b) the destructor releases the resource and never throws exceptions;
2. always use the resource via an instance of a RAII-class that either
   (a) has automatic storage duration or temporary lifetime itself, or
   (b) has lifetime that is bounded by the lifetime of an automatic or temporary object

   Move semantics make it possible to safely transfer resource ownership between objects, across scopes, and in and out of threads, while maintaining resource safety.

   Classes with open()/close(), lock()/unlock(), or init()/copyFrom()/destroy() member functions are typical examples of non-RAII classes:

```
#include <bits\stdc++.h>
class ResourceGuard{
  private:
    const std::string resource;
    enum RG{
      DEAD,
      LIVE,
      ACVITE
    }
  public:
    ResourceGuard(const std::string& res):resource(res){
      std::cout << "Acquire the " << resource << "." <<  std::endl;
    }
    ~ResourceGuard(){
      std::cout << "Release the "<< resource << "." << std::endl;
    }
};

int main() {
  ResourceGuard resGuard1{"memoryBlock1"};
  std::cout << "\nBefore local scope" << std::endl; {
    ResourceGuard resGuard2{"memoryBlock2"};
  }
  std::cout << "After local scope" << std::endl;
  std::cout << std::endl;
  std::cout << "\nBefore try-catch block" << std::endl;
  try {
      ResourceGuard resGuard3{"memoryBlock3"};
      throw std::bad_alloc();
  }
  catch (std::bad_alloc& e){
      std::cout << e.what();
  }
  std::cout << "\nAfter try-catch block" << std::endl;
  std::cout << std::endl;
}
```

1.2 Modern C++ Resource Lifetime

1.2.1 string_view

The string ”move” util to make the char like variable, whatever it come from, the fastest to deal with the memory reallocation.

```
#include <string>
std::size_t length(const std::string &s){
  return s.size();
}
int main(){
  return length("hello world! long string");
}
```

We get the resources allocated on heap, no exception. it requires the function call’s full prologue and epilogue.

```
  length(std::__cxx11::basic_string<char, ;allocator statically calculated
    std::char_traits<char>, std::allocator<char> > const&):
  mov     rax, QWORD PTR [rdi+8]
  ret
main:
  push    r12
  xor     edx, edx
  push    rbx
  sub     rsp, 56
  lea     rdi, [rsp+16]
  lea     rbx, [rsp+32]
  mov     QWORD PTR [rsp+8], 24
  lea     rsi, [rsp+8]
  mov     QWORD PTR [rsp+16], rbx
  call    std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> >::_
    M_create(unsigned long&, unsigned long)
  mov     rdx, QWORD PTR [rsp+8]
  movdqa  xmm0, XMMWORD PTR .LC0[rip] ;create a standard string using implicit conversion
  movabs  rcx, 7453010373645639783
  mov     QWORD PTR [rsp+16], rax
  mov     QWORD PTR [rsp+32], rdx
  movups  XMMWORD PTR [rax], xmm0 ; mov in an efficient way.
  mov     rdx, QWORD PTR [rsp+16]
  mov     QWORD PTR [rax+16], rcx
  mov     rax, QWORD PTR [rsp+8]
  mov     QWORD PTR [rsp+24], rax
  mov     BYTE PTR [rdx+rax], 0
  mov     rdi, QWORD PTR [rsp+16]
  mov     r12d, DWORD PTR [rsp+24]
  cmp     rdi, rbx
  je      .L3
  mov     rax, QWORD PTR [rsp+32]
  lea     rsi, [rax+1]
  call    operator delete(void*, unsigned long) ;resource deallocation
.L3:
  add     rsp, 56
  mov     eax, r12d
  pop     rbx
  pop     r12
  ret
.LC0:
  .quad   8031924123371070824
  .quad   7957697951841938546
```

Instead, we can simply change *string* to *string_view*, which is very similar to *string*, but a view to it. And we still have the long string size at compile time.

```
length(std::basic_string_view<char, std::char_traits<char> > const&):
  mov     rax, QWORD PTR [rdi]
  ret
main:
  mov     eax, 24
  ret
```

1.2.2 PMR

PMR is for you to have a more sophisticated control of memory resources in C++, all about how, when and where to have your garbage may lie in, the design is something refer to channel in golang. Basically, it has at least 4 pros:

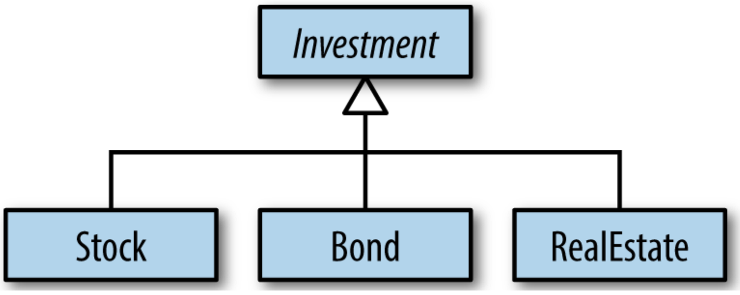
1. Fewer calls to new and delete
2. Reduced thread contention
3. Reduced false sharing
4. Reduced memory diffusion

In PMR, you can allocate a monotonic buffer resource or local thread stack resource. This solve the following 3 problems, just let the memory allocation unique to one thread. You can get a test of this code

1.2.3 smart pointer

Use *std* :: *unique_ptr* for exclusive-ownership resource management.

```
  class Investment { … };
  class Stock:
    public Investment { … };
  class Bond:
    public Investment { … };
  class RealEstate:
    public Investment { … };
```



A factory function for such a hierarchy typically allocates an object on the heap and returns a pointer to it, with the caller being responsible for deleting the object when it' s no longer needed. That' s a perfect match for std::unique_ptr, because the caller acquires responsibility for the resource returned by the factory (i.e., exclusive ownership of it), and the std::unique_ptr automatically deletes what it points to when the std::unique_ptr is destroyed. A factory function for the Investment hierarchy could be declared like this:

```
  template<typename... Ts>              // return std::unique_ptr
  std::unique_ptr<Investment>           // to an object created
  makeInvestment(Ts&&... params);       // from the given args

  auto delInvmt = [](Investment* pInvestment)       // custom
  {                                   // deleter
    makeLogEntry(pInvestment);        // (a lambda
    delete pInvestment;               // expression)
  };

template<typename... Ts>                              // revised
std::unique_ptr<Investment, decltype(delInvmt)>     // return type
makeInvestment(Ts&&... params) {
  std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
  pInv(nullptr, delInvmt);                      // returned

  if ( /* a Stock object should be created */ )
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  else if ( /* a Bond object should be created */ )
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  else if ( /* a RealEstate object should be created */ )
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));

  return pInv;
}
```

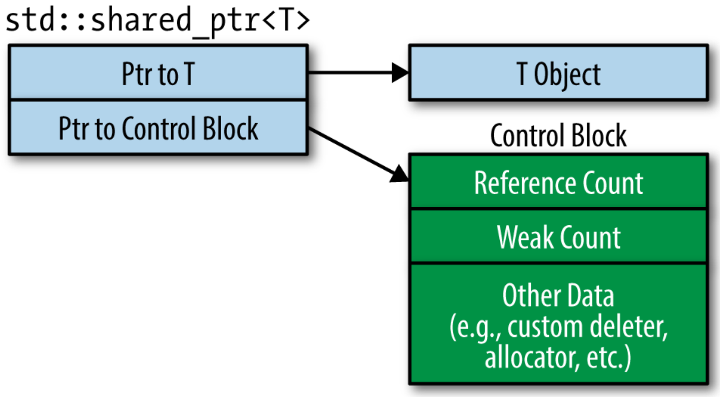
1.2.4 Use std::shared_ptr for shared-ownership resource management.

1. std::shared_ptrs are twice the size of a raw pointer, because they internally contain a raw pointer to the resource as well as a raw pointer to the resource' s reference count.2
2. Memory for the reference count must be dynamically allocated. Conceptually, the reference count is associated with the object being pointed to, but pointed-to objects know nothing about this. They thus have no place to store a reference count. (A pleasant implication is that any object—even those of built-in types—may be managed by std::shared_ptrs.) Item 21 explains that the cost of the dynamic

allocation is avoided when the std::shared_ptr is created by std::make_shared, but there are situations where std::make_shared can’ t be used. Either way, the reference count is stored as dynamically allocated data.

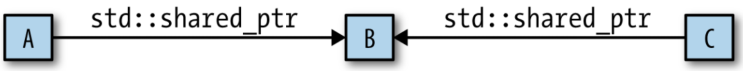
3. Increments and decrements of the reference count must be atomic, because there can be simultaneous readers and writers in different threads. For example, a std::shared_ptr pointing to a resource in one thread could be executing its destructor (hence decrementing the reference count for the resource it points to), while, in a different thread, a std::shared_ptr to the same object could be copied (and therefore incrementing the same reference count). Atomic operations are typically slower than non-atomic operations, so even though reference counts are usually only a word in size, you should assume that reading and writing them is comparatively costly.

```
auto loggingDel = [](Widget *pw)        // custom deleter
  {                         // (as in Item 18)
    makeLogEntry(pw);
    delete pw;
  };

std::unique_ptr<                        // deleter type is
 Widget, decltype(loggingDel)          // part of ptr type
 > upw(new Widget, loggingDel);

std::shared_ptr<Widget>                 // deleter type is not
 spw(new Widget, loggingDel);          // part of ptr type
```



1.2.5 Use std::weak_ptr for std::shared_ptr-like pointers that can dangle.

As a final example of std::weak_ptr’ s utility, consider a data structure with objects A, B, and C in it, where A and C share ownership of B and therefore hold std::shared_ptrs to it:



Suppose it’ d be useful to also have a pointer from B back to A. What kind of pointer should this be?

1. A raw pointer. With this approach, if A is destroyed, but C continues to point to B, B will contain a pointer to A that will dangle. B won’ t be able to detect that, so B may inadvertently dereference the dangling pointer. That would yield undefined behavior.
2. A std::shared_ptr. In this design, A and B contain std::shared_ptrs to each other. The resulting std::shared_ptr cycle (A points to B and B points to A) will prevent both A and B from being destroyed. Even if A and B are unreachable from other program data structures (e.g., because C no longer points to B), each will have a reference count of one. If that happens, A and B will have been leaked, for all practical purposes: it will be impossible for the program to access them, yet their resources will never be reclaimed.
3. A std::weak_ptr. This avoids both problems above. If A is destroyed, B’ s pointer back to it will

dangle, but B will be able to detect that. Furthermore, though A and B will point to one another, B’ s pointer won’ t affect A’ s reference count, hence can’ t keep A from being destroyed when std::shared_ptrs no longer point to it.

Potential use cases for std::weak_ptr include caching, observer lists, and the prevention of std::shared_ptr cycles.

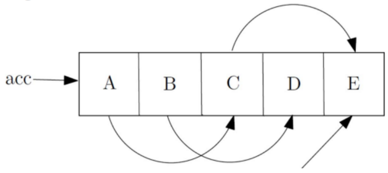
1.3 Runtime Garbage Collector

An object instance x is **Reachable** on heap iff some variable (either in register or in memory) points to x, or another Reachable object y contains a pointer to x. Unreachable objects are called **Garbage**, and is desired to get recycled by *automatic memory management*.

> The concept of Reachability is *sound* (*safe*) but not *complete*, since Unreachable objects are definitely useless, but not all Reachable objects will be used later.

A example snapshot of the heap during execution can be:

- e.g.



- Arrows indicate reference pointings
- **Roots** include all references coming from outside the heap (in `ACC` or on stack)

Various strategies of doing **Garbage Collection (GC)** exist. Three simple strategies are introduced below.

Mark & Sweep

When running out of memory conduct the following two stages:

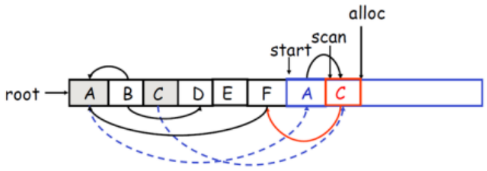
1. Start from Roots, mark all Reachable objects
2. Erase all Unreachable objects, while leaving Reachable ones unmoved

Will *fragment* the memory, but no need to update pointers since unmoved.

Stop & Copy

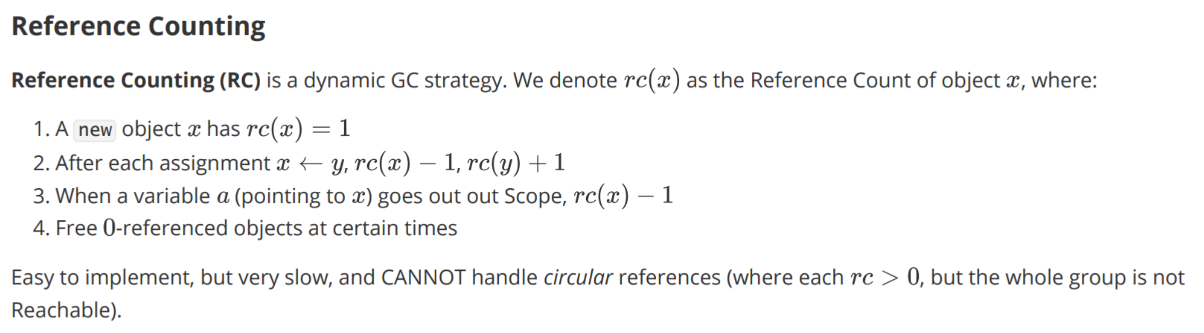
Memory is partitioned into two equal areas S_{old}, S_{new}, while S_{old} is the one under use currently. When S_{old} runs full, copy all Reachable objects to the beginning of S_{new}, and the rest of the memory is then considered free.

- e.g.



- Notice the order:
  1. First copy a Root A
  2. Follow its out-going reference to C, copy C
  3. Update the pointer in A
  4. Repeat, starting from C
  5. If a referenced child is already copied, simply update the pointer

Avoids fragmentations, but is time- and memory-expensive, since pointers need to be updated, and only half of memory is available.



Reference Counting

Reference Counting (RC) is a dynamic GC strategy. We denote $rc(x)$ as the Reference Count of object x, where:

1. A `new` object x has $rc(x) = 1$
2. After each assignment $x \leftarrow y$, $rc(x) - 1$, $rc(y) + 1$
3. When a variable a (pointing to x) goes out out Scope, $rc(x) - 1$
4. Free 0-referenced objects at certain times

Easy to implement, but very slow, and CANNOT handle *circular* references (where each $rc > 0$, but the whole group is not Reachable).

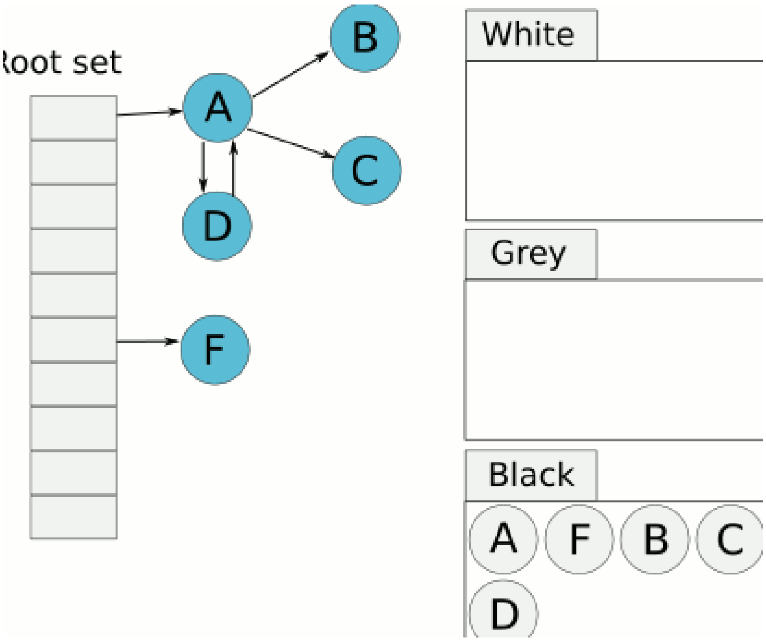
1.3.1 Case Study: Go GC

There are two main types of garbage collection algorithms, namely, tracing garbage collection algorithm and reference counting method. The three-color tagging method is one of the tracing garbage collection algorithms.

The core idea of tracing algorithm is to determine whether an object is reachable or not, because once the object is not reachable, it can be immediately reclaimed by GC. So how do we determine whether an object is reachable or not? The first step is to find all global variables and variables in the current function stack and mark them as reachable. In the second step, starting from the data already marked, further mark the variables they are accessible to, and so on, in the technical terminology called passing closures.

The Go language garbage collector has been evolving since day one, except for a few versions without major updates, almost every minor release improves the performance of garbage collection, and along with the performance improvement is the complexity of the garbage collector code, this section will analyze the evolution of the garbage collector starting from the Go language v1.0 version.

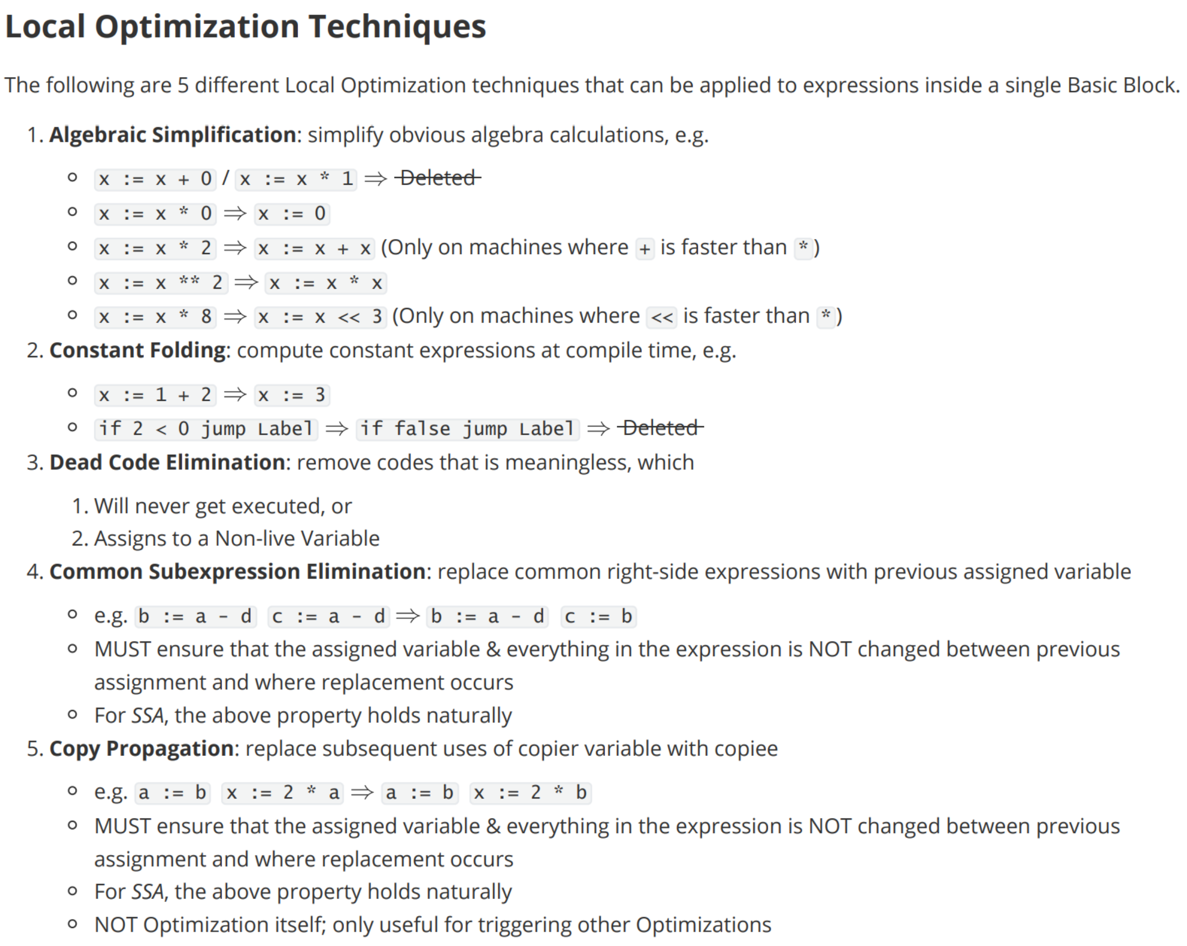
1.4 Three color label algorithm



2 Hard-core Optimization

The optimization can be classified as

1. Local Optimizations apply inside a Basic Block
2. Global (Intra-procedural) Optimizations apply to a CFG across Basic Blocks
3. Inter-procedural Optimizations (过程间优化) apply across method boundaries



Local Optimization Techniques

The following are 5 different Local Optimization techniques that can be applied to expressions inside a single Basic Block.

1. **Algebraic Simplification**: simplify obvious algebra calculations, e.g.
   - `x := x + 0` / `x := x * 1` ⇒ ~~Deleted~~
   - `x := x * 0` ⇒ `x := 0`
   - `x := x * 2` ⇒ `x := x + x` (Only on machines where `+` is faster than `*`)
   - `x := x ** 2` ⇒ `x := x * x`
   - `x := x * 8` ⇒ `x := x << 3` (Only on machines where `<<` is faster than `*`)
2. **Constant Folding**: compute constant expressions at compile time, e.g.
   - `x := 1 + 2` ⇒ `x := 3`
   - `if 2 < 0 jump Label` ⇒ `if false jump Label` ⇒ ~~Deleted~~
3. **Dead Code Elimination**: remove codes that is meaningless, which
   1. Will never get executed, or
   2. Assigns to a Non-live Variable
4. **Common Subexpression Elimination**: replace common right-side expressions with previous assigned variable
   - e.g. `b := a - d` `c := a - d` ⇒ `b := a - d` `c := b`
   - MUST ensure that the assigned variable & everything in the expression is NOT changed between previous assignment and where replacement occurs
   - For *SSA*, the above property holds naturally
5. **Copy Propagation**: replace subsequent uses of copier variable with copiee
   - e.g. `a := b` `x := 2 * a` ⇒ `a := b` `x := 2 * b`
   - MUST ensure that the assigned variable & everything in the expression is NOT changed between previous assignment and where replacement occurs
   - For *SSA*, the above property holds naturally
   - NOT Optimization itself; only useful for triggering other Optimizations

2.1 Inter-procedural Optimizations

循环不变、归纳表达式、常量传播。