

# Memory Management

Yu Zhang

**Course web site:** <http://staff.ustc.edu.cn/~yuzhang/pldpa>

# Outline

- Memory Management and Problems
- Some Solutions
  - Coding standards
  - Region-based memory management
  - Smart pointer
  - Garbage collection
- Rust Language

# References

- [Region-based memory management](#)
- [Smart Pointers in C++](#)
- [Garbage Collection](#), Richard Jones
- [Inside the JVM](#)
- [Rust](#): [book](#), [github](#)

# Memory Management

- Static data section
- Stack: stack frame
- Heap: store dynamically-allocated objects
  - **C: malloc, free**
    - Glibc's [ptmalloc](#), [Doug Lea](#)'s dlmalloc
    - Efficient concurrent memory allocator
      - [jemalloc](#), [TBBmalloc](#), [TCMalloc](#) ([gperftools](#))
  - **Java: new, **Garbage Collection****
    - [Richard Jones](#)'s the [Garbage Collection Page](#)

# Memory Management Problems

- Safety

- Do not access freed memory

- Dangling references to Stack frames or Heap
- double free, use after free

→ Abnormal program termination, denial-of-service attacks

- Free dynamically allocated memory when no longer needed

- Memory leak

→ denial-of-service attack

- Allocate and copy structures containing a flexible array member dynamically

→ Undefined behavior

- Only free memory allocated dynamically

→ Depend on the implementation

# Memory Management Problems

- Safety (cont'd)
  - Allocate sufficient memory for an object
    - Buffer overflows, the execution of arbitrary code vulnerabilities
  - .....
- Performance
  - Memory hierarchy
    - Program locality (temporal, spatial)
  - Memory allocator
    - VM-intensive: frequent memory allocation and deallocation

# Some Solutions

- Coding standards
- Region-based memory management
- Smart pointers ( modern C++)
- Garbage collection

# Coding Standards

- [SEI CERT Coding Standards](#) (CMU)
  - [Android](#)、[C](#)、[C++](#)、[Java](#)、[Perl](#)
  - [Safety and Security Coding Standards for C](#), 2016  
Author: [Robert C. Seacord](#) (SEI, CMU)
    - [Securing Coding in C and C++](#), 2<sup>nd</sup> Edition, 2013
    - C安全编码标准, 译著, 机械工业出版社, 2015
- [C++ Core Guidelines](#),
  - **Editors:** [Bjarne Stroustrup](#), [Herb Sutter](#)
- [MISRA publications](#) (UK): C/C++
  - **MISRA: Motor Industry Software Reliability Association**
- [MITRE](#) (USA)
  - [Common Weakness Enumeration](#) 公共弱点 (缺陷) 枚举
  - [Common Vulnerabilities and Exposures](#) 公共漏洞和暴露



# Region-based Memory Management

- Region (or zone, arena, memory context)
    - A collection of allocated objects that can be efficiently *deallocated all at once*
      - time performance
  - History and concepts
    - 1967 Douglas T. Ross's [AED Free Storage Package](#)
    - [[SPP 1990](#)] [David R. Hanson](#), explicit regions in C
    - Used in Apache HTTP Server, PostgreSQL, etc.
- but, do not provide memory safety
- Memory leak, dangling pointer

# Region-based Memory Management

- Region inference → safe memory allocation
  - [\[POPL1988\]](#) Ruggieri and Murtagh
    - a *region* is created at the beginning of each function and deallocated at the end
    - use *data flow analysis* to determine a lifetime for each static allocation expression, and assign it to the youngest region
  - [\[POPL 1994\]](#) Tofte and Talpin
    - Polymorphic region type and region calculus, used in SML
    - Extended lambda calculus including regions
      - $e_1$  at  $\rho$ : Compute the result of the expression  $e_1$  and store it in region  $\rho$ ;
      - letregion  $\rho$  in  $e_2$  end**: Create a region and bind it to  $\rho$ ; evaluate  $e_2$ ; then deallocate the region.

# Region-based Memory Management

- Generalization to Other PLs
  - C
    - Cyclone [[PLDI 2002](#)], [RC](#), Control-C [[CASES 2002](#)]
  - Java
    - Real time Java, combined with ownership types
  - Logic PLs such as Prolog, Mercury
- Disadvantages
  - A large proportion of dead data in large regions
  - Shorter-lifetime regions: difficulty in region inference

# Smart Pointers

Enable automatic, exception-safe, object lifetime management

- [Dynamic memory management in C++](#)

- **Pointer categories, implemented as class templates**

An object can only be referenced by a single smart pointer

- [auto\\_ptr](#)(removed in C++17): **strict** object **ownership** semantics
    - [unique\\_ptr](#)(C++11): **unique** object **ownership** semantics
    - [shared\\_ptr](#)(C++11): **shared** object **ownership** semantics
      - Reference counted pointer
    - [weak\\_ptr](#)(C++11): weak reference to an object managed by [std::shared\\_ptr](#), call **wp.lock()** to check whether the object is deleted
      - Used to break circular references of [std::shared\\_ptr](#).
- [Smart Pointers - What, Why, Which?](#)

# Garbage Collection

- Garbage

- allocated space that is no longer usable by the program

```
let x = [[1; 2; 3]; [4]] in  
let y = [2] :: List.tl x in y
```

- x is never used again and becomes garbage

- Reachability

- **Roots**: pointers that appear in the env.
- **GC**: reclaims blocks that are no longer reachable from a set of roots
- **Heap**: a directed graph in which the nodes are blocks of memory and the edges are the pointers between these blocks.
- **Reachability**: computed as a graph traversal

# Why Garbage Collection (GC)

- Eliminate a common source of defects
  - Storage leak
  - Dangling pointer
- Improve abstraction and modularity
  - A class need not include code to deal with storage deallocation
- Programs written in OCaml and Java require GC

# GC Techniques

- Requirements
  - Should identify most garbage
  - Anything it identifies as garbage must be garbage
  - Should impose a low added time overhead
  - Program pauses made by GC should be short
- Basic GC Techniques
  - Reference counting
  - **Mark-and-sweep**: mark all reachable objects from a set of roots, and sweep through memory, deallocating all unmarked objects
  - **Copying collection**: *copy* - move reachable objects from the heap to a new area called the *to-space*
  - **Mark-and-compact**: *compact* – compute forwarding addresses, update pointers and relocate blocks

# Identifying Pointers

- **Way1:** Reserve a **tag bit** in each word
  - Use up 3% memory
  - Limit the range of integers (and pointers): a 32-bit machine can address about 2GB ( $2^{31}$ )
  - Small run-time cost: arithmetic or dereference
- **Way2:** Have the **compiler** record info that the GC can query to find out the **types** of locations
  - More complicated: tightly coupling the GC and compiler
- **Way3:** **GC considers** memory unreachable only if there is nothing that looks like it might be a pointer to it
  - Work well in practice: integers are small, pointers look like large integers



# Other Issues on GC

- Traversing the heap
  - **Recursive traversal**: hard to do when low on free space
  - **Traversal *without* recursion or external stack**:
- Program pauses
  - **Generational GC**: new and old (long-lived) generations, minor GC (only scan new generation), major GC
    - *Intergenerational* pointers: pointers located in an old-generation block that points to a new-generation block
  - **Incremental GC**: let the garbage collector run *concurrently* with the rest of the program, instead of pausing the program
    - enabling *predictable real-time* performance
    - **Complicated synchronization** needed between the garbage collector and the rest of the program

# Garbage Collection and Java

- Parallel GC vs. Concurrent GC
  - **Parallel GC**: a stop-the-world, multithreaded collector
  - **Concurrent GC**: a mostly concurrent, low-pause collector
- Hashcodes
  - **Object.hashCode()**: *typically* the address of the heap block
- Finalization
  - **finalize()**: the GC calls it before reclaiming an object's block
- Package [java.lang.ref](#)
  - GC cannot reclaim **strong** references
  - GC can reclaim **soft** reference (内存不足时被回收), **weak** reference (一旦发现即回收), **phantom** reference (加到 ReferenceQueue中, 使程序可以对队列中引用的对象在回收前采取行动)