Projektarbete

IOOPM 2014

Tobias Wrigstad

tobias.wrigstad@it.uu.se

Parametrar

- Grupper om 6 personer, uppdelade i 3 par som roterar ”hela tiden”
- Redovisning med två grupper åt gången för att jämföra lösningar och problem

  Både kod och ”process”
- Två möjliga redovisningsdatum

  Slutet december, mitten januari — välj själva (anmälan öppnar senare)
- Varje grupp har en coach — listan anslås på portalen

  2–3 handledningsmöten som ni bokar
- Handlednings- och redovisningstillfällen som vanligt (dock annat schema)

  På inget sätt projektspecifika

Syftet med projektet

- Förståelse för automatisk minneshantering
- Förståelse för varför processer, etc. verktyg behövs
- Pröva på mjukvaruutveckling i projektarbetsform, t.ex.

  Komplicerade beroenden, svårt att modularisera, svårt att kommunicera

  Använda Trello för att spåra och lägga upp arbetet

  Förstå och navigera en 21-sidors specifikation

- Få en baseline för personliga mått för framtida tidsestimat

UU

Bedömningskriterier

- 1. den inlämnade kodens kvalitet och kompletthet,
- 2. inlämnad dokumentation,
- 3. kvalitet på testfall, samt
- 4. aktivt deltagande i utvecklingsprocessen.

- Vad händer om man inte är klar?

  Godkänt ändå: (se detaljer i specen)

  Rest: ett nytt försök med ny överenskommen deadline

  Underkänt: om för långt ifrån klar eller det saknas en plan för vad som skall fixas

Övriga mål som redovisas

- Y66 – Kodgranskning
- Y63 – Testdriven utveckling
- Y64 – Tillämpa Scrum eller Kanban
- Y65 – Kodstandard
- X67 – Parprogrammering
- (Se specen och AU-portalen för mer information)

UU

Inlämning (Y68)

- Övergripande designdokument
- Koddokumentation på gränssnittsnivå (t.ex. mha. doxygen)
- Själva koden
- Enhetstester
- Individuella reflektioner

  Parprogrammering

  Självreflektion
- Gemensam reflektion

Obligatoriska verktyg

- valgrind

  Löpande under utvecklingen – det kommer att vara GULD värt

- cunit

  För era enhetstester

- gcov

  För att se hur bra era tester täcker programmet

- gprof

  För att göra prestandamätningar i samband med integrationstesten

Projektarbetets domän

Minneshantering

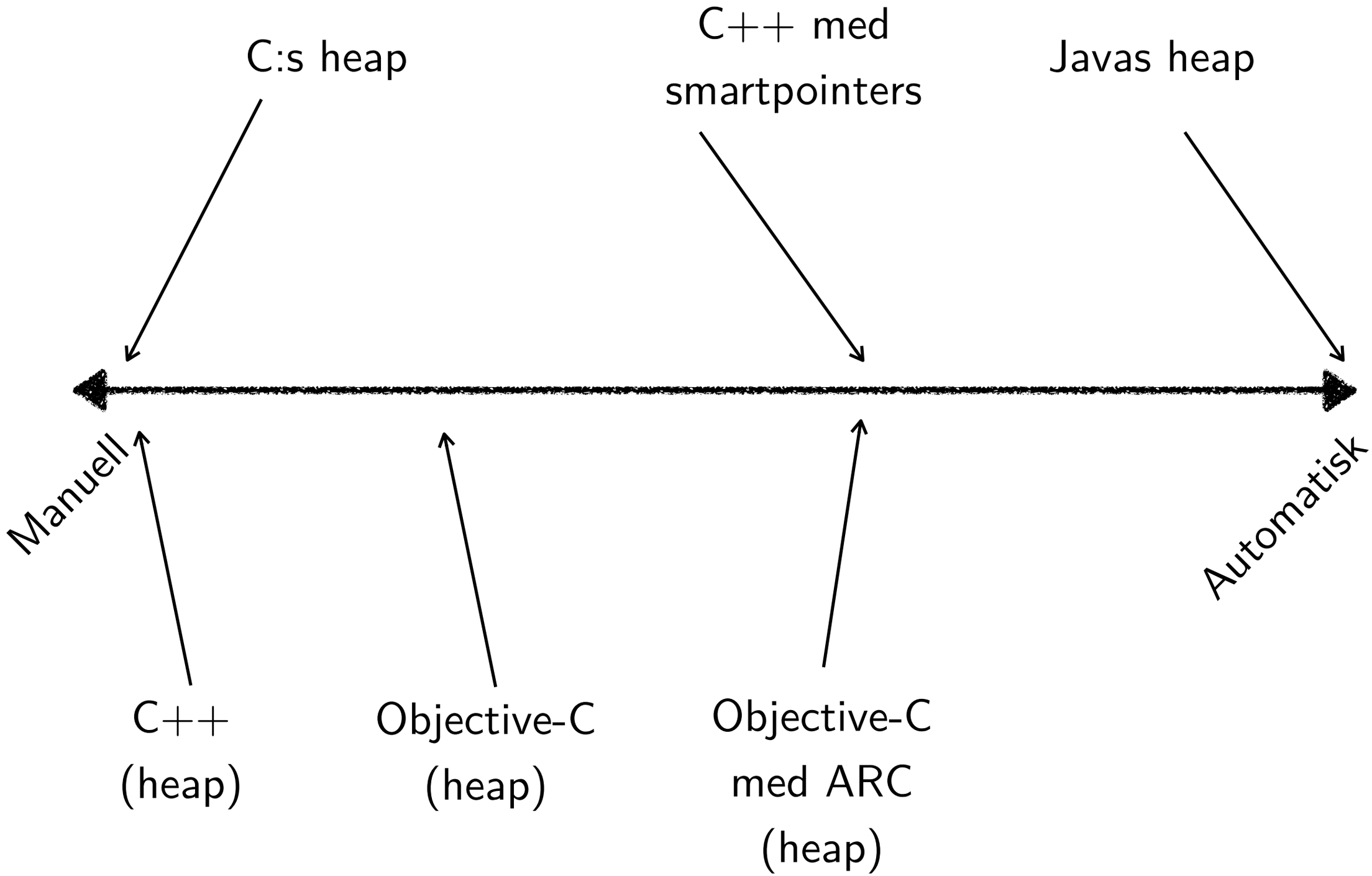
- Manuellt

  Programmeraren måste explicit begära minne av rätt storlek och återlämna

- Automatiskt

  Minne av rätt storlek begärs automatiskt och återlämnas automatiskt

En glidande skala med några exempel



UU

Olika tekniker för minneshantering

- Stacken
- Malloc & free (C, C++)
- Referensräkning (reference counting)

  Explicita retain/release-anrop (t.ex. traditionell Objective-C)

  Automatiska anrop till retain/release (t.ex. Python, Swift)
- Tracing GC

  Mark–sweep (Ruby, Python)

  Mark–compact
- Hybrider: generational GC, G1, etc. (Java)

(En naiv) Malloc & free

- Ha en lista A över allokerade block och en lista F över lediga block

  Initialt, A tom, F = [Heapen]

- Malloc(N) = p

  1. Hitta block B i F med minst N bytes, låt p vara dess startadress
  2. Dra av N bytes från B och ändra dess startadress till p + N
  3. lägg till ett N-bytes block i A med p som startadress

  Hur F är sorterad påverkar hela systemet (fragmentering, hastighet)

- Free(p)

  1. Leta upp ett block med startadress p i A, flytta det till F
  2. Om angränsande block finns i F — slå samman blocken

Referensräkning

- Varje objekt O har en räknare R_O för antalet inkommande referenser

  När objektet O skapas är $R_O = 0$

  Varje gång en referens till O sparas i en variabel, $R_O{+}{+}$

  Varje gång en referens till O förstörs, $R_O{-}{-}$

  När R_O når 0, ta bort objektet (t.ex. med free)

- Manuell referensräkning: retain(ptr) och release(ptr)

- Inget stöd för cykler!

  Python (bl.a.) använder tracing GC som komplement till referensräkning

Tracing GC

- Allokera efter någon princip så länge det finns ledigt minne
- När allokering misslyckas på grund av minnesbrist, trigga skräpsamling
- Skräpsamling i två faser: mark och sweep

  Betrakta minnet som en graf; noder = objekt, bågar = referenser

  Mark: utifrån alla objekt **som pekas ut från stacken**, besök alla objekt som går att nå från dessa och markera dem som "levande"

  Sweep: ta bort alla objekt som inte är levande

- Efter skräpsamling, försök allokera på nytt och hoppas på att ledigt minne nu finns

  90–95% av alla objekt brukar tas bort vid en GC-körning i OO-språk som Java

Mark-Sweep

- Allokering av minne sker analogt med malloc & free

  Lista A över allokerade block, lista F över lediga (”free”) block

- När malloc misslyckas, trigga GC

- När vi traverserar de levande objekten, flytta över de block i A som de ligger på till en ny lista N

- Efter alla traverseringar kan vi flytta allt som är kvar i A till F

- N blir nya A

Mark-Compact

- Effektiv allokering med ”bump pointer”

  Ha en startadress S till heapen och en slutadress E

  Allokering av ett objekt O placerar O vid S och flyttar fram S med sizeof(O) bytes

  När S når E är minnet fullt och skräpsamling måste ske

- Skräpsamling

  I samband med traversering — flytta alla levande objekt så att de ligger i ett sammanhängande minne

  Problem 1: hur vet vi vad vi kan skriva över?

  Problem 2: vi måste uppdatera alla pekare till flyttade objekt

Generationstekniker

- Dela in heapen i olika regioner för objekt av olika ålder

  Ålder bestäms av hur många GC-pass objektet överlevt

- Hantera minnet olika i olika regioner (t.ex. ung, medelålders, gammal)

- Bakomliggande tankar

  Objekt dör unga (cf. 90–95% vid nästa GC)

  Gamla objekt tenderar att leva länge

  Objekt som överlever en skräpsamling flyttas till nästa region

  Varje region skräpsamlas för sig

  Minskar upprepat arbete på långlivade objekt

Några observationer och ”jämförelser”

- Malloc & free

  Komplicerat att programmera med (vem skall free:a?) och därför felbenäget

- Referensräkning

  Betydligt enklare programmering men fungerar inte med cykler (också knepigt om objekt delas mellan trådar)

- Tracing GC

  Måste stanna världen för att samla skräp — leder till pauser i programmet

- Mark–Compact

  Behöver i regel mer minne men ger bättre lokalitet och snabbare allokering

Saker som jag inte har pratat om

- Incrementell och parallel GC

  Hur får vi GC att fungera i ett parallelexekverande program utan att stoppa alla trådar?

  (All Tracing GC har varit ”stop the world”)

- Lokalitet

  Det är bättre om objekt som används tillsammans ligger nära varandra i minnet (varför?) — hur kan man uppnå det? (Mark-compact bättre här)

- Starka och svaga pekare

- ...

Projektarbetet

Projektarbetet

- ”Implementera en automatisk minneshanterare av typen Mark–Compact som skall fungera för godtyckliga C-program”
- Förenklingar

  Vi stöder inte pekare in i objekt (t.ex. istring)

  Vi implementerar mha. two-space (enkelt men slösaktigt)

  Vi använder allokering med metadata för att identifiera pekare
- En massa rolig C-programmering

  Bitmanipulering, bitvektorer, funktionspekare
- Prestandamätningar

  Balansera overhead för skräpsamling med prestanda från bättre lokalitet

UU

Gränssnittet mot skräpsamlaren

```
typedef struct heap_t heap_s;

typedef void *(*trace_f)(heap_s *h, void *obj);
typedef void *(*s_trace_f)(heap_s *h, trace_f f, void *obj);

heap_s *h_init(size_t bytes);
void h_delete(heap_s *h);
void h_delete_dbg(heap_s *h, void *dbg_value);

void *h_alloc_struct(heap_s *h, char *layout);
void *h_alloc_union(heap_s *h, size_t bytes, s_trace_f f);
void *h_alloc_raw(heap_s *h, size_t bytes);

size_t h_gc(heap_s *h);
size_t h_avail(heap_s *h);
```

UU

Gränssnittet mot skräpsamlaren

```c
#include <stdio.h>
#include "gc.h"

int main(int argc, char *argv[])
{
  heap_s *h = h_init(1024);
  tree_node *t = h_alloc_struct(h, "d**");
  list_node *l = h_alloc_struct(h, "d*");
  printf("Available memory: %lu\n", h_avail(h));

  h_gc(h);
  printf("Available memory: %lu\n", h_avail(h));

  h_delete_dbg(h, (void *)0xDEADBEEF);
  return 0;
}
```

Ett inbyggt problem

- För att hitta rötterna till traverseringen måste vi scanna stacken efter vad som ser ut som pekare
- Risk att vi identifierar värden som pekare som inte är pekare

  För att detta skall ske måste vi hitta ett värde sizeof(void*) som är samma som en adress som returnerats av vår allokerare.
- Om detta sker — BOOM!!!

  Vad är sannolikheten? På 32bit vs. 64bit? (Vilka adresser har ni?)
- En lösning om man råkar ut för problem

  Ha en lista med rötter som hanteras explicit (add_root, del_root)

  Ett vettigt första steg i implementationen ändå och användbart i testning