TOBIAS WRIGSTAD

SPECIFIKATION FÖR PROJEKTARBETE PÅ IOOPM'15

IOOP/M 2015

1 Inledning

Projektarbetet består av en specifikation (detta dokument) som skall implementeras i programspråket C. Arbetet skall utföras i team om 6 personer som jobbar i roterande par. Vårt mål är att två team skall bilda en grupp som kommer att redovisa samtidigt. Denna koordination sköts internt av kursledningen.

Projektet har många syften: att fördjupa kunskaperna i C, att bygga programmeringserfarenhet på djupet och bygga på den kunskap som byggts upp under föregående faser, samt att ge en plattform för ytterligare redovisning av mål inom ramarna för ett ”riktigt program”. Ett av huvudsyftena är också att introducera element från *programvarutekniken*, d.v.s. den ingenjörsdisciplin som sysslar med utveckling av mjukvara inom givna tids-, kostnads- och kvalitetsramar och här specifikt testning. Genom att utföra en litet större uppgift än enkla laborationer och inlämningsuppgifter, och införa samarbetselement mellan par som utför olika delar av uppgiften, kommer ni att få uppleva vikten av klart definierade processer och roller och klart definierade gränssnitt mellan programmoduler. Målet är inte ”att visa hur man gör”, utan snarare att försöka ge en bakgrund till varför metoder och tekniker som tas upp på senare kurser är nödvändiga för systematisk utveckling av mjukvara.

WORK BREAKDOWN STRUCTURE

1. Se till att sätta teamet i samband
2. Läs dessa instruktioner noggrant, sedan en gång till
3. Löpande under projektet, dokumentera & versionshantera
4. Planering och design (≈2 dgr)
   (a) Gör en övergripande design för systemet
   (b) Dela upp systemet i lika många ”delsystem” som det finns par
   (c) Definiera gränssnitten mellan delsystemen
5. Implementation, parallellt i paren (>2 v)
   (a) Dela upp ditt delsystem i delar $\Delta, \Delta' \ldots$
   (b) Fundera ut hur man testar del Δ
   (c) Implementera testerna T_Δ för Δ,
   (d) Implementera Δ och testa löpande mot T_Δ

   (Upprepa steg b–d…)
6. Integration (≈1 v)
   (a) Sätt ihop delsystemen, testa, åtgärda fel
7. Reflektera kort över projektet

1.1 *Processen*

Arbetet skall utföras i team om 6 personer updelade i 3 programmeringspar[1] uppdelade i två team. Arbetet delas upp i minst lika många uppgifter som par. Teamen slumpas fram, som vanligt. Ni skall i den mån det är lämpligt följa Scrum[2] som utvecklingsprocess.

Varje vecka skall paren roteras; en person fortsätter med samma uppgift ytterligare en vecka, och den andra personen byter till ett annat par. När ni programmerar bör ni förstås också byta vem som sitter framför tangentbordet ofta, t.ex. efter varje logisk uppgift, dvs. gärna flera gånger i timmen!

Vid projektets slut skall varje projektmedlem skriva ett antal reflektioner om parprogrammering, egna prestationen etc., se § 2.1.

I slutet av projektet skall teamet skriva en gemensam rapport/reflektion om hur ni har arbetat, vilka rutiner/processer/hjälpmedel som har fungerat och inte, och vilka svårigheter som uppstod under implementation och integration.

[1] Ett programmeringspar består naturligtvis av två personer som skall tillämpa *parprogrammering*. Försök att följa instruktionerna på http://www.wikihow.com/Pair-Program.

[2] En ganska lagom beskrivning finns på http://sv.wikipedia.org/wiki/Scrum

Det skall inte ta mer än en dag att göra reflektionsarbetet och skriva ned dokumentationen, och även om skriftlig framställning är extremt viktigt i all form av mjukvaruutveckling så avser uppgiften inte främst rapportskrivning. Se § 2.2 för vidare information.

Ni måste själva göra uppdelningen av uppgiften i ”delsystem”. Då delarna med stor sannolikhet är beroende av varandra är det av stor vikt att ni tidigt definierar *gränssnitten* mellan delarna så att *integrationsfasen*, d.v.s. då delarna sätts samman till ett fungerande bibliotek, fungerar så smärtfritt som möjligt.

Gränssnitten mellan delarna specificeras i headerfilen `gc.h` (given) som inte får modifieras, samt andra ”privata” headerfiler som ni själva skapar.

1.2 *Coachen*

Varje team får en coach tilldelad sig, för att få hjälp och svara på frågor. Dessa anslås på kursens webbsida i samband med grupperna. Tidigt under projektets gång bör man ha ett möte med coachen. Vid detta första möte skall teamet presentera sin tänkta högnivådesign för coachen, samt sin planering, d.v.s. hur systemet är uppdelat i delsystem, gränssnittet mellan delsystemen, hur delsystemen är fördelade över programmeringspar, och några första grova deadlines.

Teamet ansvarar för att boka ett avstämningsmöte med sin coach någon gång under projektets gång. Vid detta möte skall teamet kort rapportera om hur arbetet fortskrider, om man räknar med att bli klar i tid, eventuella stora problem, etc. Vid behov kan ytterligare möten bokas. Vid problem skall man i första hand kontakta sin coach.

1.3 *Aktivt deltagande*

Studenter som inte aktivt deltar i projektet får göra om projektdelen av kursen ett annat år. Teamen uppmanas att göra kursansvariga uppmärksamma på sådana studenter. Poängen med projektet är lärdomarna från att göra det, inte att leverera ett färdigt system. Om man låter någon åka snålskjuts gör man vederbörande en otjänst!

1.4 *Planering och uppföljning*

Under projektet uppmuntrar vi till aktivt användande av något verktyg, gärna Trello (`http://trello.com`). Ni ansvarar själva för att sätta upp ett ”bräde” på Trello med lämpliga rättigheter. Bjud in er coach så att hen kan följa arbetet! Använd listorna i Trello för att fånga enheter att implementera i olika kort, tilldela ansvar genom att knyta personer till kort, etc.[3]

[3] En möjlighet med Trello är att använda olika listor för olika moduler, etc.

1.5 *Versionshantering och issue tracking*

Under projektet skall ni använda er av Github för att versionshantera koden. Ni kommer att få ett *privat* konto på Github, som ni *måste* använda. Av uppenbara

och fuskrelaterade skäl får koden inte göras publik eller delas med andra utanför teamet (undantaget coachen och kursledningen). Versionshistoriken på Github visar om versionshantering använts på ett vettigt sätt.

Github har ett utmärkt stöd för issue tracking, d.v.s. buggrapporter och diskussioner kring buggar. Spårbarhet är oerhört viktigt i systemutveckling, så använd en issue tracker/bug tracker, även om alla sitter i samma rum.

1.5.1 Övriga obligatoriska Verktyg

Det är ett krav att använda följande verktyg:

`valgrind`	för att verifiera att koden inte läcker minne eller gör andra dumma läsningar eller skrivningar utanför allokerat minne.
`cunit`	för att skriva tester; ingen funktion utan tester!
`gcov`	för att kontrollera hur stor del av koden som faktiskt testas!

1.6 Personliga Produktivitetsmått

Möjligen något inspirerad av Watts Humphrey's "Personal Software Process" är ett av målen med projektet att uppmuntra till kontinuerlig förbättring av ditt eget arbetssätt, bland annat:

1. Bli bättre på att planera och uppskatta tidsåtgång för uppgifter
2. Bli bättre på att göra "commitments" som du kan hålla
3. Bli bättre på att reducera mängden defekter i din kod

För att börja jobba med dessa aspekter av ditt arbetssätt *skall* du kontinuerligt skriva ned[4]:

- Alla uppgifter du skall utföra. (Det är vettigt att ha ett antal uppgifter per dag. Försök att bryta ned arbetet i enheter om 30–60 minuter. Bryt upp enheter som visar sig vara större än vad du trodde, etc.)
- Din *uppskattning* av uppgiftens tidsåtgång.
- Det *faktiska utfallet*, dvs. om du utförde uppgiften och den tid det tog.
- "Biggest fail" och "biggest win", dvs. det största problem som du stötte på och det smartaste eller bästa du gjorde.
- Logga din arbetstid och kategorisera varje timme som möte, planering, design, implementation, testning, dokumentation eller postmortem[5].

I slutet av projektet skall du sammanställa hur många timmar du faktiskt arbetade och hur många timmar du uppskattade att du skulle behöva arbeta. Antalet timmar du arbetade har ingen inverkan på betyget – du gör detta mest för din egen skull så det är superviktigt att du är ärlig i dina siffror!

Finns det återkommande wins och fails? Underskattar du konsekvent tiden det kommer att ta att utföra en uppgift? Verkar det som om du tar på dig för mycket i relation till din kapacitet?

[4] En bra idé är att ha t.ex. ett kalkylark i Google docs i vilket du gör löpande anteckningar om nedanstående. (Det går också att ha mer publika och mer ostrukturerade noter i Trello.) På så sätt blir det enkelt att visualisera trender etc.

[5] Postmortem är t.ex. all reflektion och sammanfattning och rapport i projektets slut.

2 Inlämning

Vid det datum som angivits på kurswebben är det dags att lämna in, oavsett status på implementationen. För projektet skall följande lämnas in (och motsvarar målet Y68):

Övergripande designdokument. Uppdelningen i delsystem, delsystemens moduler om lämpligt och deras gränssnitt, lämpligen dokumenterat med hjälp av doxygen. Målgruppen för denna dokumentation är alltså en *de andra utvecklarna* av biblioteket och målet är att möjliggöra parallell utveckling – dvs. att flera programmerare samtidigt kan skriva olika delar av ett system som i slutändan skall prata med varandra. Det betyder att interna implementationsdetaljer som t.ex. bitmönster etc. skall finnas med i dokumentationen[1].

[1] I viss utsträckning, främst initialt, betyder det att man kopierar vissa detaljer från denna specifikation för att ha allt på samma ställe.

Koddokumentation på gränssnittsnivå. Se motsvarande mål eller ta ledning av man-sidorna för C eller JavaDoc. Målgruppen för denna dokumentation är alltså *en klient* av biblioteket. Informationen här skall vara precis tillräcklig för att skriva programmen i § 3.7.

Själva koden. Inklusive en makefil som kan bygga den på Linux (X86) och Solaris (både X86 och SPARC). (Alltså 3 plattformar totalt.)

Enhetstester i CUnit. Med instruktioner om hur de kan köras (helst en makefil som kör alla), tillsammans med en sammanställning av hur många tester som finns, hur många som passerar, samt code coverage-data från `gcov`[2] på lämpligt format.

[2] Börja t.ex. på `http://en.wikipedia.org/wiki/Gcov`.

Vid behov – dokumentation av vad som saknas och varför. Om projektet inte är komplett bör man *utöver ovanstående* lämna in ett dokument som beskriver vilka funktioner som återstår, och en översiktlig beskrivning av hur dessa kan implementeras i den existerande koden.

Exempelvis, projektet funkar inte på SPARC (beskriv varför, vad måste till, när uppstår felen, etc. – använd ert historikdata för att göra en informerad gissning om hur lång tid det skulle ta att fixa!).

Prestandatester & Integration med existerande program Se § 3.7 och § 3.8.

Vid inlämningen bedöms projektet och en av tre saker händer: (a) projektet blir godkänt, (b) projektet får en ny deadline för restinlämning eller (c) projektet blir underkänt. I fall (b) sker en ny inlämning och redovisning senare, där betyget godkänt eller underkänt ges. *Ett underkänt projekt kan kompletteras först nästa gång kursen går.*

Bra beskrivningar av bristerna och hur dessa skulle kunna åtgärdas kan medföra godkänt.

Det kan hända att implementationen inte är färdigställd vid deadline. En ofärdig implementation skall ändå lämnas in och ackompanjeras av ett dokument som beskriver vilka funktioner som återstår, och en översiktlig beskrivning av hur dessa kan implementeras i den existerande koden. En buggig implementation bör ackompanjeras av ett testfall som reproducerar felet, och om möjligt en beskrivning av varför buggen uppstår.

2.1 Individuella reflektioner

Nedan beskriver vi individuella aspekter av arbetet.

2.1.1 Parprogrammering

Varje projektmedlem skall skriva en kort reflektion[3] om parprogrammering. Exempel på saker att reflektera kring är:

[3] Ca 1000 tecken.

- Upplever du att det bidrar till bättre kod, eller borde/kan man sitta enskilt och att få dubbel produktionstakt?
- Fungerar vissa par bättre än andra? Varför – är det t.ex. bättre med en jämnare kunskapsnivå, eller sämre?
- Vilket råd skulle du ge någon som aldrig har varit i en parprogrammeringssituation förut?

2.1.2 Självreflektion

Varje projektmedlem skall skriva en kort självreflektion[4] om sin prestation. Vilka är dina styrkor och svagheter i ett projektarbete? Vilka egenskaper bör du förstärka och vilka behöver du bli bättre på? Titta på t.ex. wins och fails här. Ytterligare exempel på saker att reflektera kring är:

[4] Ca 1000 tecken.

- Hur fungerar du i ett team? Faller du in i ett särskilt mönster, eller tar du återkommande en särskild roll?
- Får du ut något av att jobba i en grupp, eller jobbar du helst ensam?
- Jämför hur mycket du lärt dig under projektarbetet kontra kursens övriga delar, och fundera över hur du bäst lär dig saker.

2.2 Gemensam reflektion för teamet

I en kort text[5] Reflektera tillsammans över hur det har gått att:

[5] Ca 1000–1500 tecken. Använd tydliga rubriker och passa gärna på att bocka av S52.

- *Kommunicera* inom projektgruppen och mot externa.
- *Samarbeta* inom projektgruppen – diskutera även parprogrammeringen då kanske inte alla delar samma uppfattning om hur det har fungerat.
- *Koordinera* aktiviteterna i projektet.

– *Ta beslut* speciellt kring planering, eller när saker gått fel eller man varit oense.
– *Komma fram till – och följa – en process* och om processen har varit ett stöd.

Sammanställ också samtliga gruppmedlemmars tidsloggar och ta fram en gemensam sammanställning av tidsåtgång och fördelning över olika kategorier.

2.3 *Bedömningskriterier*

Projektet/teamet bedöms på:

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

Observera att det inte är ett strikt krav att ha ett fullt fungerande system vid deadline för att bli godkänd. Däremot krävs att man gjort ett allvarligt *försök* att leverera ett fullt fungerande system med för 5 HP rimlig arbetsinsats (≈133 arbetstimmar). Alla brister i systemet[6] skall vara dokumenterade, kvarvarande buggar skall ha testfall som exponerar dem, och det skall finnas en plan för hur arbetet skall fortsätta så att systemet skall uppfylla specifikationen.

[6] Inklusive utelämnade funktioner.

Enhetstester skriver ni själva. Integrationstest blir minst de prestandatester som ni skall skriva i § 3.7 och integrationen i § 3.8.

2.3.1 *Rest*

En ofullständig inlämning vid deadline kan[7] medföra rest. Teamet får då en skriftlig beskrivning av vad som måste åtgärdas före en ny inlämning kan ske, samt ett nytt *sista leveransdatum*. Ett team som inte lämnar in ett system som uppfyller specifikationen vid detta datum får göra om projektdelen av kursen ett senare år.

[7] En oseriös eller undermålig inlämning kan medföra underkänt.

2.4 *Redovisning av mål andra än Y68*

Som en sidoeffekt av projektet får gruppen också möjlighet att bocka av vissa mål utöver Y68. Dessa mål kräver särskild dokumentation som kan vara rimlig att baka ihop med projektreflektionen, och ibland inte. Ni måste klart och tydligt skriva ut vilka mål ni vill bli examinerade på, samt vilken ”dokumentation” som skall användas. Som vanligt gäller att man måste peka ut relevant information – att mer eller mindre skriva ”det finns begravet någonstans i projektdokumentationen” är alltså inte okej.

Y66 – Kodgranskning Skicka med kodgranskningsprotokoll och en kort rapport. Vilka buggar som hittades. Varför valde ni just den del av koden som ni valde? Var det rätt beslut i efterhand? Hur lång tid tog det? Hände

något som är värt att nämna i övrigt? På detta/dessa protokoll bör man ha skrivit vilka som var med och vill ha målet avbockat.

Y63 – Testdriven utveckling Utvärdera skriftligt hur testdriven utveckling har fungerat i projektet. Vad har fungerat bra? Vad har fungerat dåligt? Etc. Det kan eventuellt vara lämpligt att baka ihop med reflektionen i projektreflektionen eller som ett separat dokument.

Y64 – Tillämpa Scrum eller Kanban Här fungerar förhoppningsvis gruppens gemensamma reflektion över processen bra tillsammans med en beskrivning av den valda processen samt vad i den valda processen som fungerade och inte.

Y65 – Kodstandard En kort diskussion om nyttan av kodstandard, samt en länk till den kodstandard som har använts (alternativt en beskrivning av den kodstandard som tagits fram). Har det givit något att ha en kodstandard? Har läsbarheten påverkats?

X67 – Parprogrammering Här fungerar de individuella reflektionerna och gruppens gemensamma reflektion över parprogrammering och processen bra som dokumentation.

X69 – Tillämpa regressionstestning under projektet Tala om var man kan hitta enhetstesterna, vilken code coverage som finns, vilka skript för att köra enhetstesterna, etc.

Givetvis kan man använda koden i projektet för att redovisa ”vanliga” mål i labbsal, precis som i resten av kursen, så länge som det är delar av systemet och kod som man själv har varit aktivt inblandad i.

3 Uppgiftsbeskrivning

Denna del av specifikationen är ett *levande dokument* som kan komma att uppdateras och förändras under projektets gång.

Uppgiften går ut på att utveckla ett bibliotek, för enkelhets skull kallat $\mathcal{GC}$, för minneshantering i form av en *konservativ kompakterande skräpsamlare*. En användare kan skapa en ”egen heap” – ett konsekutivt minnesblock[1] – i vilket man sedan kan allokera minne. Allokeringar i den egna heapen skall sedan hanteras automatiskt – när minnet tar slut[2] skall skräpsamling automatiskt triggas, och alla objekt i detta minne som inte är nåbart via någon rot i systemet tas bort[3]. Ett korrekt implementerat projekt kan (och skall – se § 3.8) integreras med en inlämningsuppgift från tidigare del av kursen I det förändrade programmet skall all allokering skall ske med hjälp av $\mathcal{GC}$-biblioteket och ingen manuell avallokering skall ske, utan att programmets minne skall ta slut.

[1] Skapas under huven t.ex. med hjälp av `posix_memalign` i `stdlib.h`, eller `mmap` i `sys/mman.h`.

[2] Eller något annat villkor som användaren anger.

[3] Vi gör en förenkling och utgår från att programmen är enkeltrådade och att endast en heap skapas per program.

I detta kapitel beskrivs uppgiften. Av pedagogiska skäl beskriver vi först skräpsamling med hjälp av mark–sweep (och som redan beskrivits på föreläsning), som vi *inte* skall använda innan vi går in på den kompakterande skräpsamlaren som använder en liknande algoritm.

3.1 Skräpsamling med mark–sweep

Skräpsamling med mark–sweep vandrar genom (traverserar) den graf som heapen utgör för att identifiera objekt som fortfarande används. Alla objekt som inte används anses vara skräp och kan frigöras utan att programmet kraschar. Vi går igenom algoritmen steg-för-steg nedan.

Vi kan tänka oss att varje objekt innehåller en extra bit[4], den s.k. *mark-biten*. När denna bit är satt (1) anses objektet vara ”vid liv”. Annars är objektet skräp som kan tas bort.

[4] Tekniskt kan det också vara en bit om man har en över. Ibland kan man packa in bitar i annat data – vi skall se exempel på det senare i denna text!

Vid skräpsamling sker följande (logiskt sett):

Steg 1 Iterera över samtliga objekt på heapen och sätter mark-biten till 0. Detta innebär att alla objekt anses vara skräp initialt.

Steg 2 Sök igenom stacken efter pekare till objekt på heapen[5], och med utgångspunkt från dessa, traversera heapen och markera alla objekt som påträffas genom att mark-biten sätts till 1.

[5] Dessa pekare kallar vi också för ”rötter”.

Steg 3 Iterera över samtliga objekt på heapen och frigör alla objekt vars

mark-bit fortfarande är 0.

Steg 2 kallas för ”mark-fasen” och steg 3 för ”sweep-fasen”, härav algoritmens namn, *mark–sweep*.

3.1.1 Att traversera heapen

Att traversera heapen i C försvåras av att minnet som standard allokeras utan metadata. T.ex. så allokerar detta anrop

```
void *p = malloc(sizeof(binary_tree_node));
```

plats som rymmer en `binary_tree_node`, det sparas ingen information om innehållet i detta utrymme, mer än hur stort utrymmet är som `p` pekar på. Vi skulle vilja ”fråga” minnet vilka pekare det innehåller och hur stort det är, men det kan vi alltså inte göra. Vi måste själva implementera stöd för detta.

Rimligtvis har en `binary_tree_node` åtminstone två pekare till höger respektive vänster subträd – så hur gör man för att hitta dem?

Ett sätt är att leta igenom det minne som pekas ut av `p` och tolka varje möjlig `sizeof(void *)` i detta utrymme som en adress. Om adressen pekar in i den aktuella heapens[6] anser vi att den är en pekare till det objekt som finns lagrat där (observera att pekaren inte måste peka på starten av det objektet). Då skall vi markera detta objekt som levande (dess mark-bit sätts till 1), varefter dess utrymme också letas igenom på samma sätt som `binary_tree_node`:en i jakt på andra pekare in i den aktuella heapen. Om ett objekt redan markerats och traverserats behöver man inte göra det igen.

Men hur vet man då vilka pekare som finns som pekar in i heapen? För att hitta dessa, de s.k. ”rotpekarna”, måste man leta igenom stacken efter pekare till heapen på samma sätt som ovan, alltså gå igenom hela stackens adressrymd, inklusive register och de statiska dataareorna och leta efter pekare in i heapens adressrymd.

[6] Alltså dess pekaren är en adress i den egna heapens adressrymd – dvs. mellan dess startadress och slutadress.

Tillåtna förenklingar map. ovanstående

Vi tillåter flera förenklingar i denna uppgift – vi kräver inte stöd för pekare ”in i objekt” (alltså som inte pekar till starten av ett objekt)[7], eller scanning av den statiska dataarean[8]. Vi uppmuntrar förstås till stöd för dessa vanliga C-idiom, men det är inte nödvändigt.

[7] Implementerar du inte stöd för detta blir det heller inte säkert att använda sådana pekare i de program som använder minneshanteraren.

[8] Dvs. globala variabler – samma som föregående not gäller. Notera att vissa förenklingar kan kräva att man ändrar de program som man vill integrera med sitt bibliotek senare.

Nu har vi sett hur man kan leta igenom både stacken och heapen efter pekare. Då skall vi titta på hur man kompakterar heapen i syfte att minska fragmentering.

3.2 Kompakterande skräpsamlare

En kompakterande skräpsamlare är en relativt vanlig skräpsamlartyp som vid skräpsamling flyttar samman objekt i minnet. Det finns åtminstone tre goda skäl till att göra detta:

1. Det undviker fragmentering, eftersom allt använt minne och allt icke-använt minne ligger var för sig, konsekutivt.
2. Objekt som pekar på varandra tenderar att hamna nära varandra vilket förbättrar minneslokaliteten hos programmet.
3. Det ger möjlighet till en mycket effektiv implementation av allokering.

3.2.1 *Effektiv allokering och avallokering*

Om alla levande objekt flyttas samman vid allokering kommer allt ledigt minne att vara konsekutivt och vi behöver inte föra bok över var ledigt minne finns, vilket vore fallet för mark–sweep. Därför kan allokering implementeras med så-kallad ”bump pointer”. Det går till så att man har en pekare till starten av det fria minnet, ”fronten”, och att allokering av n bytes returnerar den nuvarande addressen till fronten, varefter fronten flyttas n bytes. Denna typ av allokering är betydligt snabbare än en implementation som söker bland en lista av fria block för att hitta ett av lämplig storlek[9].

Vidare, om vi enbart opererar på levande objekt och ignorerar skräp blir tidskomplexiteten $O(\#\text{levande objekt})$ istället för $O(\#\text{objekt})$. Det tillåter oss att skriva en implementation som undviker alltså steg 1 och steg 3 i beskrivningen av mark–sweep i § 3.1.

Det är inte ovanligt att 90–95% av alla objekt är skräp vid en skräpsamling[10], så detta är en stor tidsvinst, även om kopiering är dyrt.

[9] Jmf. `malloc` som vi diskuterat på föreläsning

[10] Den s.k. ”weak generational hypothesis.”

3.2.2 *Naiv implementation: Två minnesareor (eng. two-space)*

Det enklaste sättet att implementera en kompakterande skräpsamlare är att dela upp minnet i två olika minnesareor, en passiv och en aktiv. Alla objekt finns i den aktiva arean, och all allokering sker där – den passiva arean används inte.

Om man fyller den aktiva arean triggas skräpsamlingen. Den utgår från samtliga rötter och traverserar samtliga levande objekt i den aktiva arean. Varje objekt som hittas på detta sätt kopieras över[11] in i den passiva heapen, och vi noterar kopians adress[12]. Varje pekare vi hittar i objekt under traverseringens gång ersätter vi med adressen till dess överflyttade kopia så att vi till slut kopierat över alla levande objekt från den aktiva arean till den passiva, och uppdaterat alla pekare mellan objekten så att kopiorna pekar ut varandra. På samma sätt uppdaterar vi också alla rotpekare att peka på kopiorna. (Hela detta motsvarar alltså steg 2 i beskrivningen av mark–sweep i § 3.1.)

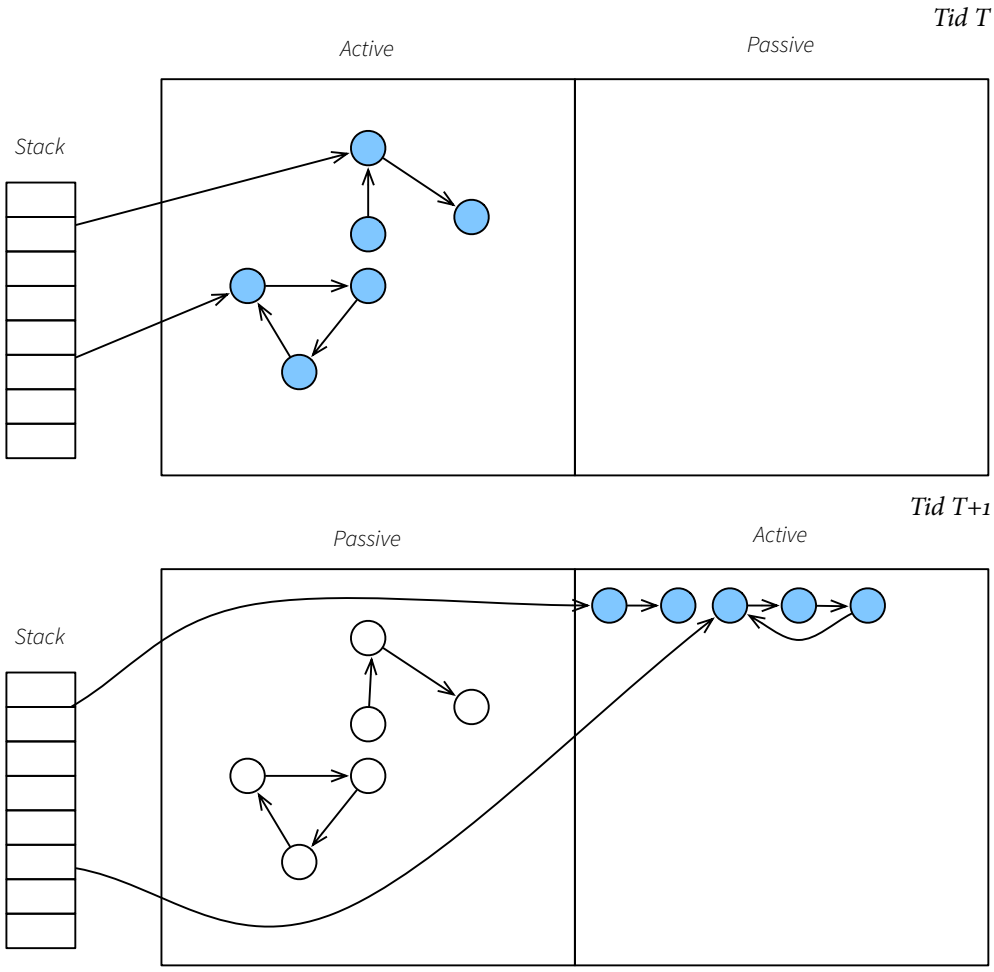
När traverseringen och kopieringen är klar byter vi så att den passiva arean blir aktiv, och den aktiva passiv.

Vad vi har åstadkommit nu är alltså att alla objekt innan skräpsamlingen finns i den numer passiva arean och betraktas som skräp. Endast de objekt som programmet kunde nå har flyttats över in i den nya aktiva arean vilket betyder att den använder minsta möjliga minne som fortfarande garanterar att programmets alla pekare är korrekta.

Vi kommer att implementera en variant av denna efter Bartlett, se § 3.2.4. För en illustration av den naiva implementationen, se 3.1.

[11] Vid denna kopiering används bump pointer-allokering i den passiva arean och den initiala fronten är areans start.

[12] En s.k. forwarding-adress.



Figur 3.1: Naiv implementation av en kopierande skräpsamlare. Minnet är indelat i två areor, en aktiv och en passiv. När skräpsamling triggas kopieras alla levande objekt över från den aktiva till den passiva arean, varefter den passiva arean blir aktiv och den aktiva passiv. Notera att i den nya aktiva arean är objekten kompakterade, dvs. lagda intill varandra. På detta vis undviks minnesfragmentering.

Notera att uppdelningen av minnet i två areor varav endast den ena är i bruk vid varje givet tillfälle (förutom vid skräpsamlingen då båda används) effektivt dubblar ett programs minnesanvändande. Detta har inte hindrat denna teknik från att användas i praktiken; de flesta program använder relativt lite minne, och smidig och korrekt minnesanvändning är ofta viktigare än *yteffektiv*. (Generationsbaserade skräpsamlare där flera olika skräpsamlingstekniker kombineras kan också hjälpa till att minska ”slöseriet” med minne.)

3.2.3 En kompakterande, konservativ skräpsamlare för C

Ett problem med skräpsamling i språk som C är avsaknaden av metadata i minnet. Eftersom en pekare och en integer ser identiska ut (och en adress är ett positivt heltal!) är det möjligt att tolka *heltalet* 3786230 (som kanske avser slutpriset på en enrummare i Stockholms innerstad) som *pekaren* 0x39C5F6 (samma tal skrivet i bas 16). Har vi otur kan 0x39C5F6 råka vara en valid adress i den heap som hanteras av $\mathcal{GC}$.

Vi kommer att hantera detta problem med en kombination av fyra tekniker:

1. Konservativ kompaktering efter Bartlett (§ 3.2.4)
2. Allokeringskarta (§ 3.2.5) (*frivilligt att implementera*)
3. Höga adresser (§ 3.2.6) (*frivilligt att implementera*)
4. Allokering med metadata (§ 3.2.7)

3.2.4 *Konservativ kompaktering efter Bartlett*

Se 3.2 för en bild som visar denna implementation.

Bartlett skiljer mellan säkra och osäkra pekare. En säker pekare är en adress som vi säkert vet är en pekare. En osäker pekare är en vars data vi inte säkert vet är en pekare. Ett typiskt användande av Bartletts teknik är för skanning av stacken i ett C-liknande språk, där vi inte vet vad det är för data vi tittar på.

Vi klassificerar alltså alla pekare vi hittar som säkra eller osäkra. Att vara konservativ innebär att vi måste utgå från att en osäker pekare faktiskt är en pekare i bemärkelsen att vi måste betrakta dess utpekade objekt som levande, och samtidigt att vi måste utgå från att den osäkra pekaren faktiskt inte är en pekare vilket innebär att vi inte kan flytta dess utpekade objekt i minnet eftersom det kräver att vi ändrar pekarvärdet till den nya adressen. Exempel, om vi hittar 0x39C5F6 (slutpriset på en lägenhet) på stacken måste det objekt som ligger på den adressen överleva och inte flyttas. Flyttade vi det till t.ex. adressen 0x1C0030 måste vi uppdatera värdet på stacken till 0x1C0030 (peka om ”pekaren”), vilket skulle betyda att vi ändrat slutpriset på lägenheten!

För att använda Bartletts trick för att hantera osäkerhet delar vi in minnet som vi hanterar i ett antal diskreta ”sidor”[13]. Istället för att dela in hela heapen i två delar – passiv och aktiv – ger vi *varje sida* statusen passiv eller aktiv. En osäker pekare till en adress *A* medför nu att den omslutande sidan *P* inte får flyttas vid kompaktering.

[13] För enkelhets skull använder vi storleken 2048 bytes per sida. Observera att man måste se till att objekt inte korsar sidgränser. För enkelhets skull kan vi sätta en maxgräns på objektstorlek så att alla objekt ryms på en sida.

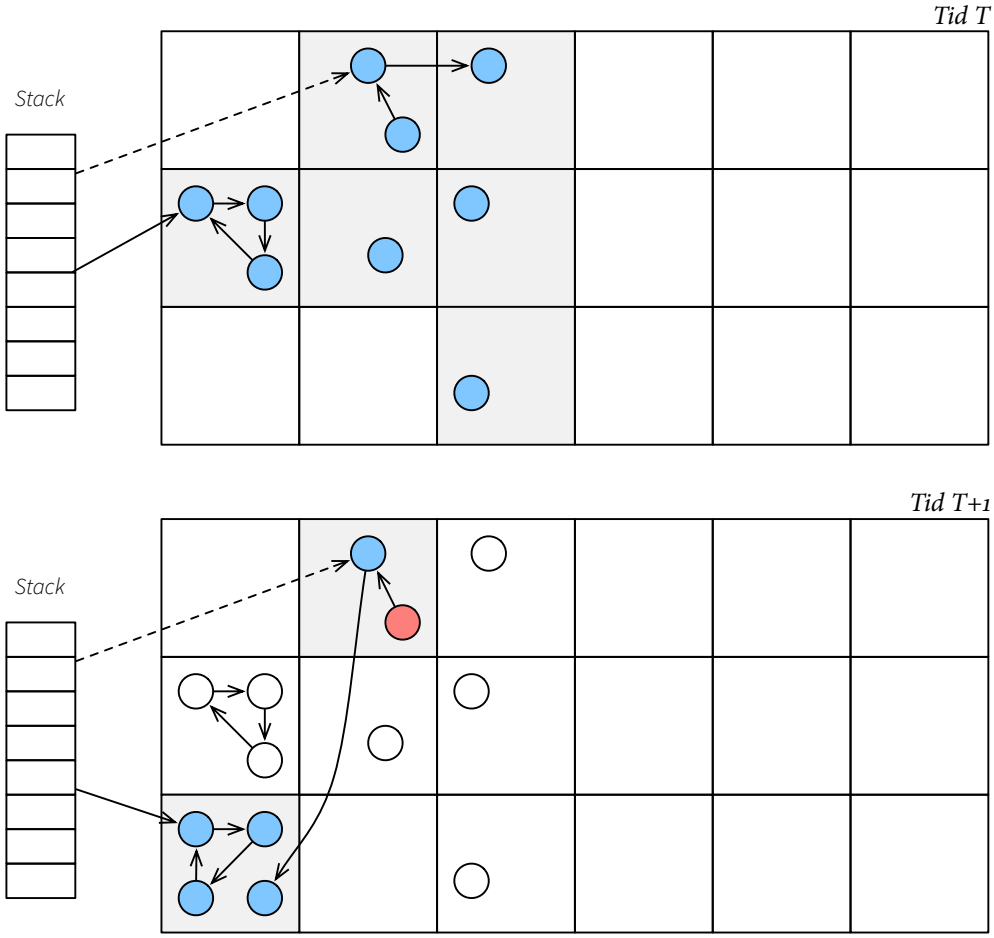
När man skapar sin minnesarea skall man kunna kontrollera om pekare på stacken skall anses som säkra eller osäkra.

3.2.5 *Allokeringskarta*

Ytterligare ett sätt att minska risken för felaktiga pekarvärden är att använda allokeringskarta. En allokeringskarta är en array av booleans där varje plats i arrayen motsvarar en valid adress för en allokering, och där `true` betyder att något allokerats på den platsen (`false` – inte). Om man t.ex. har (vilket är rimligt) en minsta objektstorlen på 16 bytes[14] behöver man alltså en array med 1024 element för att hålla reda på 16 kb. Om man använder en `bitvektor` där en enskild bit är en boolean behövs alltså bara 128 bytes för att hålla reda på 16 kb, vilket är <1% overhead.

[14] Inklusive header, se § **??**.

Om vi återgår till vårt exempel där vi hittar 0x39C5F6 (slutpriset på en lägenhet) på stacken kan vi se om den boolean som motsvarar den adressen i allokeringskartan är `true` eller `false`. Om värdet är `false` kan vi helt



Figur 3.2: I en skräpsamlare av Bartlett-typ är minnet indelat i många små sidor som var och en kan vara aktiv (grå) eller passiv (vit). Pekare kan också vara säkra eller osäkra (streckad pil). Objekt som utpekas av osäkra pekare får inte flyttas. Detta implementeras genom att hela sidan som objektet ligger på är oförändrad. Detta leder i exemplet i figuren till att ett skräp-objekt inte tas bort för att det råkar ligga på samma sida som ett osäkert utpekat objekt.

ignorera 0x39C5F6. Om värdet är `true` måste vi behandla det som en säker eller osäker pekare.

3.2.6 *Höga adresser*

Detta är enkelt att implementera och brukar ge hög avkastning. Använd t.ex. `posix_memalign` för att allokera minnet till programmets egen heap och ange en mycket hög adress som alignment. Det medför att alla pekaradresser som skapas kommer att vara väldigt stora. Eftersom program sällan manipulerar väldigt stora tal minskar risken för att ett heltal i programmet skulle råka sammanfalla med en valid minnesadress.

3.2.7 *Allokering med metadata*

För att slippa leta igenom heapen på samma sätt som stacken kommer vi att använda ett format för allokering som kräver att användaren ger den information vi behöver. Vårt skall stödja tre typer av allokering:

1. `h_alloc_struct` – där programmeraren anger en slags formatsträng som beskriver minneslayouten hos objektet som skall allokeras[15]. Formatsträngen beskriver var i ett objekt eventuella pekare finns som också skall traverseras för att markera objekt som levande.
2. `h_alloc_raw` – där programmeraren anger storleken på ett utrymme som skall reserveras[16]. Detta utrymme *får* inte innehålla pekare till andra objekt.
3. `h_alloc_union` – där programmeraren utöver storlek också skickar med en pekare till en funktion som används för att traversera objekt av denna typ, se vidare § 3.3.4.

I alla fall ovan skall det allokerade minnet nollställas, dvs. samma beteende som `calloc`.

[15] Analogt med hur en formatsträng till `printf` beskriver hur en utskriven sträng ser ut och var olika värden skall stoppas in. Se nedan.

[16] Analogt med `malloc`.

Formatsträng för `h_alloc_struct`

Formatsträngen förklaras enklast genom exempel. Antag att vi har en typ `binary_tree_node`, deklarerad enligt följande.

```
struct binary_tree_node {
  void *value;
  struct binary_tree_node *left;
  struct binary_tree_node *right;
  int balanceFactor;
}
```

Detta utrymme kan beskrivas av formatsträngen `"***i"` som betyder att utrymme skall allokeras för 3 pekare, följt av en **`int`**, dvs.,

```
alloc("***i");
```

är analogt med

```
alloc(3 * sizeof(void *) + sizeof(int));
```

Notera att data alignment kan påverka en strukturs layout för mer effektiv minnesåtkomst i en strukt. Detta kan betyda att två fält efter varandra i en strukt har ”tomt utrymme” mellan sig för att värdens plats i minnet skall bättre passa med ord-gränser. Man kan antingen sätta sig in i hur detta fungerar[17] – notera att det är plattformsberoende – eller fundera ut hur man stänger av det.[18] Ange tydligt i dokumentationen hur detta har hanterats.

En bra första iteration i implementationen av stödet för allokering med formatsträng implementerar stöd för `"*"` och `"r"`, där det sistnämnda står för **`sizeof(int)`**, vilket på en 64-bitars platform ger möjligheten att allokera antingen i ”byggklossar” om 8 respektive 4 bytes[19].

Åtminstone följande styrkoder skall kunna ingå i en formatsträng:

`*`	pekare	`i`	int	`f`	float
`c`	char	`l`	long	`d`	double

[17] En bra plats att börja på är `http://en.wikipedia.org/wiki/Data_structure_alignment`

[18] En bra plats att börja på är `http://gcc.gnu.org/onlinedocs/gcc/Structure_002dPacking-Pragmas.html` och `http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html`

[19] Faktiskt så räcker detta bra som en ”intern representation” av formatsträngen, oavsett vad användaren skriver (se nästa §).

Ett heltal före ett specialtecken avser repetition; till exempel är `"***ii"` ekvivalent med `"3*2i"`. Man kan se det som att default-värdet 1 inte måste sättas ut explicit, alltså `*` är kortform för `1*`. En tom formatsträng är inte valid. En formatsträng som bara innehåller ett heltal, t.ex. `"32"`, tolkas som `"32c"`. Detta innebär att `h_alloc_struct("32")` är semantiskt ekvivalent med `h_alloc_raw(32)`.

3.3 Implementationsdetaljer

Eftersom objekt i C inte har något metadata måste implementationen hålla reda på två saker:

1. Hur stort varje objekt är (annars kan vi inte kopiera det), samt
2. Var i objektet dess pekare till andra objekt finns.

Formatsträngen innehåller information för att räkna ut båda dessa, men formatsträngen är inte helt oproblematisk, t.ex. eftersom den ägs av klienten som kan förändra den[20] och därmed få en formatsträng att avvika från ett objekt som det förväntas beskriva, och också för att den inte stöder unioner i struktar. Om varje formatsträng kopierades med motsvarande `strdup` skulle vi vara mer skyddade mot fel på grund av förändringar i formatsträngar, men det skulle bli ett påtagligt slöseri att skapa många kopior av strängar. En bättre implementation skulle använda en mer kompakt representation av formatsträngen *som kunde bakas in i det allokerade objektet*.

[20] Eller avallokera den.

I implementationen av den kompakterande skräpsamlaren skall varje objekt ges en *header* (metadata), men vi är intresserade av att denna header är så liten som möjligt eftersom program som allokerar många små objekt annars blir för ineffektiva. En god design är att spara headern precis innan varje objekt i minnet. Låt oss börja med att titta på vad headern skall kunna innehålla för information:

1. En pekare till en formatsträng
2. En mer kompakt representation av objektets layout
3. En pekare till en funktion som hanterar skräpsamling av ett objekt med komplex layout (t.ex. struktar med unioner, se § 3.3.4, om detta implementeras)
4. En forwarding-adress
5. En flagga som anger om objektet redan är överkopierat till den passiva arean vid skräpsamling

Lyckligtvis kan vi representera samtliga dessa data i ett enda utrymme av storlek `sizeof(void *)` med hjälp av litet klassisk C-slughet. Vi kan börja med att notera att alternativ 1–3 är ömsesigt uteslutande, dvs., finns en kompakt representation av objektets layout behövs varken formatsträng eller en objektspecifik skräpsamlingsfunktion, osv. Vidare behövs forwarding-

adress enbart när en kopia redan har gjorts av objektet, vilket t.ex. betyder att forwarding-adressen kan skriva över objektets data eftersom allt överskrivet data går att hitta om man bara följer forwarding-pekaren. Slutligen kan vi konstatera att flaggan i 5 enbart behövs i samband med 4.

Om vi använder adresser som är ”alignade” mot ord i minnet, och varje ord är minst 4 bytes, så kommer alla addresser att vara en multipel av 4, vilket betyder att de sista två bitarna i en adress i praktiken inte används. (Virtuellt minne kan medföra att vi har ett stort antal insignifikanta bitar i varje adress, detta är maskin och OS-specifikt.) Notera att man kan använda t.ex. `-falign-functions=16` för att även styra vilka adresser funktionspekare får.

3.3.1 Vilken information finns i headern

För att inte slösa med minnet skall vi använda de två minst signifikanta bitarna i en pekare för att koda in information om vad som finns i headern. Alltså, om en 32-bitars pekare binärt är (med little-endian) `1001000111101100-01100101000001000` pratar vi om att gömma information i de sista två, dvs. `10010001111011000110010100000010__`.

Två bitar är tillräckligt för att koda in fyra olika tillstånd, t.ex.:

Mönster	Headern är en…
`00`	pekare till en formatsträng (alt. 1)
`01`	forwarding-adress (alt. 4)
`10`	pekare till en objektspecifik skräpsamlingsfunktion (alt. 3)
`11`	bitvektor med layoutinformation (alt. 2, se nedan)

Notera att de två minst signifikanta bitarna måste ”maskas ut” ur pekaren innan pekaren används – annars kan pekarvärde bli ogiltigt på grund av att datat vi gömt där tolkas som en del av adressen. Det betyder att varje läsning av headern som en pekare skall sätta de två minst signifikanta bitarna[21] till 0 i det utlästa resultatet.

[21] Notera skillnader mellan big-endian och little-endian i hur adresser representeras binärt.

3.3.2 Objektpekare pekar förbi headern

Objektets header ligger alltid först i objektet, men skall inte vara synlig i några struktar (det skulle göra programmet beroende av en specifik skräpsamlare, vilket vore dåligt). Därför kommer en pekare till ett objekt alltid att peka på struktens första byte, dvs. den ”pekar förbi” headern. Och om man vill komma åt headern måste man använda pekararitmetik och ”backa” `sizeof(header)` bytes. Denna typ av design tillåter att skräpsamlaren modiferas så att headern växer och krymper utan att program som använder skräpsamlaren måste modifieras.

3.3.3 En mer kompakt layoutspecifikation

Vi skall använda en bitvektor för att koda in en layoutspecifikation på ett sätt som är betydligt mer yteffektivt än en formatsträng. Vi kan t.ex. använda en bit för att ange antingen en pekare eller ”data”, t.ex. `11001` är samma som formatsträngen `"**ii*"`, som ger en objektstorlek på 32 bytes[22] om en pekare är 8 bytes och `sizeof(int)` är 4 bytes. Vi behöver också information om bitvektorns längd.

[22] 8 + 8 + 4 + 4 + 8 = 32

För större allokeringar av enbart data behöver layoutspecifikationen enbart vara en storleksangivelse. I likhet med headern kan vi reservera en bit för att

ange om layoutspecifikationen är en storlek i bytes, eller om det är en vektor med mer precis layoutinformation.

På en maskin där en pekare är 64 bitar skulle alltså 2 bitar gå åt till metadata om headern, ytterligare 1 bit gå åt till att koda in typ av layoutspecifikation, och resterande 61 bitar antingen vara en storlek i bytes eller en bitvektor och dess längd[23].

Notera att eftersom den kompakta layoutspecifikationen har en fix längd fungerar denna representation bara för data av begränsad storlek (som också styrs av huruvida headern är 32 eller 64 bitar).

[23] Faktiskt behövs inte längden. Om man kodar varje byggkloss som två bitar, t.ex. `01` för `r` och `11` för `*` och `00` för inget mer så räcker det med att scanna bitvektorn tills man hittar `00` för att avgöra längden och maxlängden blir 30.

3.3.4 *Objektspecifika skräpsamlingsfunktioner*

OBS! Det är frivilligt att implementera stöd för denna funktion (§ 3.3.4).

Objektspecifika skräpsamlingsfunktioner är användbara för objekt med komplex layout, t.ex. en union mellan en pekare och annat data. En sådan skräpsamlingsfunktion tar lämpligen som argument objektet, heapen, samt den funktion som normalt anropas för samtliga pekare i objektet[24]. Skräpsamlingsfunktionen ansvarar också för att kopiera objektet självt (det är enkelt om man byter aktiv och passiv area först i skräpsamlingsfunktionen) och returnerar en pekare till kopian som resultat.

[24] Vi kallar detta för en trace-funktion eftersom syftet med denna funktion är att ”trace:a” – traversera alla pekare och därigenom besöka alla nåbara objekt på heapen.

```
// Typen för den interna trace-funktionen
typedef void *(*trace_f)(heap_t *h, void *obj);

// Typen för objektspecifika trace-funktioner
typedef void *(*s_trace_f)(heap_t *h, trace_f f, void *obj);
```

(Se § 3.6 för information om typen `heap_t`.) Låt säga att vi ville ha en strukt med en union så här:

```
enum type { INT, PTR }; // integer or pointer

struct example {
  enum type type;
  union { // type indicates type in union
    void *pointer;
    uint32_t integer;
  };
};
```

Att ange en formatsträng för denna strukt går inte med vårt begränsade språk eftersom vi varken har notation för unioner eller möjlighet att uttrycka att om variabeln `type` har ett visst värde är värdet i unionen en pekare (som bör trace:as), annars är värdet i unionen inte en pekare (i detta fall en 32 bitars **`unsigned int`**). Därför måste vi använda en objektspecifk funktion som till exempel kan se ut så här (ofärdigt kodskelett):

```
void *trace_example_struct(heap_t *h, trace_f f, void *obj)
{
  struct example *e = (struct example *)obj;
```

```
    // ... kopiera e till e', spara addressen till e' i e:s header ...

    if (e->type == PTR)
      {
        f(h, e->pointer);
      }
    else
      {
        // inget behöver göras, e->integer är inte en pekare
      }
  }
```

3.4 *Att skapa och riva ned en heap*

Funktionen `h_init` som ni skall implementera för skapar en ny heap med en angiven storlek och returnerar en pekare till den. Utöver storlek skall det gå att ställa in två ytterligare parametrar:

1. Huruvida pekare på stacken skall anses som säkra eller osäkra
2. Vid vilket minnestryck skräpsamling skall köras

Eftersom det måste vara möjligt att resonera om minneskraven för en applikation skall *allt* metadata om heapen också rymmas i det angivna storleksutrymmet. Funktionen `h_avail` returnerar antalet tillgängliga bytes i en heap, dvs. så många bytes som kan allokeras innan minnet är fullt (dvs. minnestrycket är 100%).

Det skall finnas två funktioner för att frigöra en heap och återställa allt minne:

1. `h_delete` som frigör allt minne som heapen använder.
2. `h_delete_dbg` som utöver ovanstående också ersätter alla variabler på stacken som pekar in i heapens adressrymd med ett angivet värde, t.ex. `NULL` eller `0xDEADBEEF` så att ”skjutna pekare” (eng. dangling pointers) lättare kan upptäckas.

3.5 *Att hitta rötterna för skräpsamlingen*

Att hitta rötterna (eng. root set) kräver att man letar igenom stacken efter samtliga bitmönster som kan tolkas som pekare och som har en adress som pekar in i den aktuella heapen. Detta kan man göra genom betrakta stacken som en array från *B* till *E* och pröva alla möjliga `sizeof(void *)`-block mellan *B* och *E*. Eftersom värden kan hållas i register kan det vara lämpligt att använda någon C-funktion som tvingar alla register att sparas på stacken. Här är ett lämpligt makro som gör det. Man behöver *inte* scanna `env` på något sätt, utan innehållet dumpas på stacken (verifiera gärna detta genom att ta reda på

hur `env` är definierad på de aktuella maskiner du vill köra på genom att läsa deras `setjmp.h`).

```
#include <setjmp.h>

#define Dump_registers() \
  jmp_buf env; \
  if (setjmp(env)) abort(); \
```

Toppen på stacken kan man approximera genom att t.ex. ta adressen till en stackvariabel på den översta stack-framen. Ett bättre sätt, som dock inte fungerar på alla kompilatorer, är att använda `__builtin_frame_address(lvl)` som returnerar adressen till den översta framen på stacken när `lvl` är 0, den anropande funktionens stack frame när `lvl` är 1, etc. För att få toppen på stacken i användarens program i `h_gc` kan man alltså bara skriva `void *top = __builtin_frame_address(1)`. Botten på stacken kan man också få fram genom att läsa adressen till den globala variabeln `environ` som enligt C-standarden skall ligga ”under” starten på stacken.

Läs mer på https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html.

Åtkomst till `environ` ges genom att man deklarerar den som en *extern*, analogt med en global variabel:

```
extern char **environ;
```

Notera att huruvida stacken växer uppåt eller nedåt i adressrymden är plattformsspecifikt. Betänk också *data alignment* vid genomsökning av stacken – på vilka adresser kan man hitta adresser?

3.6 *Gränssnittet* `gc.h`

Nedanstående headerfil sammanfattar det publika gränssnitt som skall implementeras. En doxygen-dokumenterad version finns också tillgänglig i kursens repo.

```
#include <stddef.h>
#include <stdbool.h>

#ifndef __gc__
#define __gc__

typedef struct heap heap_t;

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

heap_t *h_init(size_t bytes, bool unsafe_stack, float gc_threshold);
void h_delete(heap_t *h);
void h_delete_dbg(heap_t *h, void *dbg_value);

void *h_alloc_struct(heap_t *h, char *layout);
void *h_alloc_union(heap_t *h, size_t bytes, s_trace_f f);
```

```
void *h_alloc_raw(heap_t *h, size_t bytes);

size_t h_avail(heap_t *h);
size_t h_used(heap_t *h);
size_t h_gc(heap_t *h);
size_t h_gc_dbg(heap_t *h, bool unsafe_stack);

#endif
```

3.7 *Enkla prestandamätningar*

Beroende på vilken implementation av `malloc` du använder används olika strategier för att allokera minne. Gör några enkla prestandatest för ett program som allokerar många objekt[25] och mät:

1. För ett stort program som ryms i minnet (alltså där skräpsamlaren aldrig körs), går det att observera prestandaskillnader mellan er minneshanterare och `malloc`? Detta test mäter allokeringens effektivitet.

2. För ett stort program som *inte* ryms i minnet, går det att observera prestandaskillnader mellan er minneshanterare och `malloc`? Detta test mäter allokeringens effektivitet, men också skräpsamlingens. Traversering av objekt kostar, men samtidigt krävs endast att man bearbetar data som är levande, till skillnad från `malloc`/`free` där allt skräp måste explicit lämnas tillbaka, vilket förstås tar tid.

3. Skriv ett program som skapar 4 länkade listor av heltal[26], där varje lista håller i tal inom ett visst intervall, $[0, 1 \times 10^9)$, $[1 \times 10^9, 2 \times 10^9)$, etc. upp till 4×10^9. Slumpa fram M tal i intervallet $[0, 4 \times 10^9)$ och stoppa in dem i rätt listor[27]. $(*)$ Slumpa sedan fram N tal och sök igenom rätt lista och svara på om talet finns där.

   Använd både `malloc` och er egen minneshanterare i ovanstående program och jämför körtiderna. Storleksförhållandet mellan M och N bör vara $M \approx 10 \times N$. Pröva också att göra en skräpsamling vid punkten $(*)$ i programmet och justera N uppåt utan att ändra M. Kan man se en skillnad i körtider?

[25] 10-tals megabyte minne totalt för programmet.

[26] Som alla skall rymmas i den allokerade heapen.

[27] Det är viktigt att varje slumptal tas fram ur intervallet $[0, 4 \times 10^9)$ och inte att man först slumpar lista ett, sedan lista två etc.

Resultatet av prestandatesterna kommer att efterfrågas i samband med redovisningen. (Grafer är ett bra sätt att förmedla information!)

Körtid kan man mäta t.ex. med `time` (ett POSIX-program) eller genom att använda C:s inbyggda funktioner för att läsa av systemklockan (t.ex. `time.h`).

3.8 *Integration med existerande program*

Kronan på verket när skräpsamlaren är klar är att integrera den med ett existerande program. Välj valfritt program från en godkänd redovisning av

Z101 (fas 1/sprint 2). Detta program skall inte ha några globala pekare och allokera sitt minne dynamiskt (jmf. krav på denna inlämningsuppgift).

Här är ett förslag för hur detta kan göras:

1. Ersätt alla anrop till `malloc` med anrop till en funktion som räknar ut summan av alla allokeringar, samt anropar `malloc` och returnerar pekaren till det allokerade minnet, ungefär i denna stil (och motsvarande för `calloc`):

   ```
   void malloc_indirection(size_t size)
   {
     global_counter += size;
     return malloc(size);
   }
   ```

   I slutet av programmet kan du skriva ut hur mycket minne (M) det använde:

   ```
   printf("Total memory: %zu\n", global_counter);
   ```

2. Döp om `main`-funktionen till `old_main` och lägg till ytterligare en parameter:

   ```
   int __attribute__ ((noinline)) old_main(int argc, char *argv[], heap_t *h)
   ```

   Attributet `__attribute__ ((noinline))` förhindrar att kompilatorn slår samman funktionerna `main` och `old_main`. Vi vill inte det eftersom vi vill vara säkra på att `old_main`:s stack frame har förstörts när vi triggar en sista skräpsamling nedan.

3. Skapa en ny `main`-funktion så här:

   ```
   int main(int argc, char *argv[])
   {
     // med lämpliga parametrar, se nedan
     heap_t *h = h_init(...);
     int result = old_main(argc, argv, h);
     h_gc_dbg(h, true);

     size_t b = h_used(h);
     printf("Memory used at end of program: %zu b\n", b);
     h_delete(h);
     return result;
   }
   ```

   För enkelhets skull kan du vilja göra `h` till en global variabel. Storleken på heapen som du skapar kan vara t.ex. hälften av den siffra M som du nyss tog fram. Eftersom de flesta objekt som skapas är temporära är det rimligt att tänka sig att minnestrycket vid varje givet tillfälle är mindre än M. (Experimentera gärna med olika heapstorlekar och/eller tröskelvärden för skräpsamling!)

4. Skriv om alla anrop till `malloc_indirection` till motsvarande `h_alloc`-funktion (t.ex. `_raw` för strängar, `_struct` för noder i länkade listor och träd, etc.). Ta också bort alla anrop till `free`.

När programmet körs *bör* slutligen ”Memory used at end of program: o b” skrivas ut, eftersom inget minne längre används av programmet.

Demonstrera detta vid slutseminarium!

Plattformsoberoende

Det bibliotek som ni implementerar *skall* fungera under Linux på X86 (t.ex. `tussilago.it.uu.se`) och Solaris på X86 (t.ex. `fries.it.uu.se`) och SPARC (t.ex. `yxan.it.uu.se`). Alla dessa miljöer finns tillgängliga på institutionen. Notera att t.ex. bitmanipulering är högeligen plattformsberoende[28] (även storleken på en **`int`**!) så det lönar sig snabbt att ha tester och köra dem på flera plattformar löpande under utvecklingens gång.

[28] T.ex. big endian vs. little endian.

4 Frivilliga utökningar

Dessa kan förstås användas för att redovisa mål.

Stöd för dynamiskt växande heap (Enkelt) Istället för att ha en heap vars storlek är fix kan man tänka sig att heapens storlek växer dynamiskt. Detta är möjligt genom att helt enkelt lägga till fler sidor i Barlett-kollektorn. Extra plus i kanten om minne också lämnas tillbaka när minnestrycket är lågt.

Stöd för grundläggande telemetri (Enkelt) Implementera funktioner som monitorerar minnestrycket, dvs. hur mycket minne är faktiskt levande vid varje skräpsamling.

Parallellisera skräpsamlingen. (Svår) Ett problem med denna typ av automatisk skräpsamling är att allt skräp samlas vid en enda tidpunkt. I interaktiva program kan detta upplevas som en tydlig *paus* under körning. Ett sätt att minska paustiden är att traversera heapen med hjälp av flera parallella trådar. Notera att detta inte är detsamma som stöd för multitrådade program (se utökning nedan) – trådningen är i detta fall helt inkapslad i skräpsamlaren och inte synlig utanför. De stora problemen som måste lösas är:

1. Koordination av trådar som vill kopiera samma objekt samtidigt.
2. Koordination av trådar som vill kopiera över ett objekt från den aktiva arean till den passiva arean samtidigt – var finns starten på det lediga minnet?

Stöd för att använda heapen med multipla trådar. (Mycket svår) Förenklingen att endast fungera med enkeltrådade program är tyvärr inte en rimlig förenkling i praktiken i väldigt många program. Utöka biblioteket för att fungera med skräpsamling med multipla trådar. Detta kan inte lösas lika transparent som för enkeltrådar. De stora problemen som måste lösas är:

1. Koordination av trådar som vill allokera minne samtidigt.
2. Samtliga trådars stackar måste scannas för att rötterna skall hittas.
3. Skräpsamling kan inte ske samtidigt med vare sig *allokering* eller *användning* av levande objekt eftersom objekt flyttas i minnet.[1] Samtliga trådars exekvering måste följaktligen ”avbrytas” för att skräpsamling skall kunna ske.

[1] Det finns förstås flera skräpsamlare som klarar av detta, men deras arbetssätt skiljer sig från det som vi har beskrivit här.

Det enklaste sättet att implementera denna utökning är att utöka protokollet så att ett program som använder multipla trådar måste göra vissa saker för att skräpsamlingen skall fungera korrekt.