TOBIAS WRIGSTAD & JUSTIN PEARSON

PROJEKTSPEC

IOOP/M 2013

Meta

Introduktion

Projektarbetet består av en specifikation (detta dokument) som skall implementations i programspråket C. Arbetet skall utföras i team om 6 personer som jobbar i roterande par. Två team bildar en grupp som kommer att redovisa samtidigt och ha ett gemensamt uppstartsmöte.

Projektet har många syften: att fördjupa kunskaperna i C, att bygga programmeringserfarenhet på djupet och bygga på den kunskap som byggts upp under fas 0 och 1, 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

Processen

Arbetet skall utföras i team om 6 personer[1] updelade i 3 programmeringspar[2] uppdelade i två team. Arbetet delas upp i minst lika många uppgifter som par. Teamen slumpas fram som vanligt och presenteras på http://wrigstad.com/ioopm/groups-fas2.html. Utgå från Scrum[3] vid designen av er process.

Varje vecka skall paren roteras; en person fortsätter med samma uppgift ytterligare en vecka, och den andra personen byter till ett annat par.

Parrotationerna, samt vem som jobbat på vad, bokförs i en projektdagbok som skall lämnas in i slutet av projektet. I denna skall även

[1] Säkerligen med något undantag.

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

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

REGEL FÖR PARROTATION

Om det är möjligt *måste* man välja en person som man inte redan arbetat med i projektet.

arbetstiden per uppgift och hur många timmar varje enskild programmerare har arbetat bokföras (se sid 3).

I slutet av projektet skall även en kort reflektion på 500–750 ord lämnas in och undertecknas av alla teammedlemmar. I denna skall ni reflektera över hur ni har arbetat, vilka rutiner/processer/hjälpmedel som har fungerat och inte, och vilka svårigheter som uppstod under implementation och integration. Det skall inte ta mer än en dag att skriva denna reflektion, och även om skriftlig framställning är viktigt så avser inte uppgiften rapportskrivning.

Ni måste själva göra uppdelningen av uppgiften i ”delsystem”, och motivera den i er projektdagbok. 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.

GRÄNSSNITTSDOKUMENTATION

Gränssnitten mellan delarna specificeras i headerfilerna `imalloc.h` som är utdelad kod som inte får modifieras, samt i `priv_imalloc.h` för privata funktioner.

Möten med 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 (lämpligen ca 25/11) 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 grova deadlines. Presentationen vid avsparksmötet skall göras i form av en poster där individuella delar skrivs ut på A4- eller A3-ark som skall rymmas på ett A1-ark.[4] Vi kommer att försöka samköra möten mellan minst två grupper så att man kan få ta del av andra gruppers design.

[4] Detta A1-ark kommer att ligga på ett bord och skall kunna betraktas av alla personer som står runt bordet, så använd lämplig fontstorlek.

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

PROJEKTDAGBOKENS INNEHÅLL

• En sammanställning över hur många timmar varje projektmedlem har arbetat samt en fördelning av tiden över kategorierna *möten*, *implementation*, *testning* och *annat*. • En förklaring av systemets uppdelning i delsystem. • En sammanställning av hur många timmar varje delsystem tagit att implementera. • Parrotationerna.

Projektavslut och inlämning

Den 17/12 är det dags att lämna in. Ni skall, oavsett status på implementationen, lämna in följande leverabler L1–L6:

L1 Projektdagbok (*skrivs löpande under projektet!*)

L2 Övergripande designdokument

L3 Koddokumentation på gränssnittsnivå

L4 Gränssnitten mellan modulerna (i kommentarer i koden eller som separat dokument)

L5 Själva koden (med fungerande makefile)

L6 Tester

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.*

OBSERVERA

Utmärkta beskrivningar av brister i implementationen, och hur dessa skulle kunna rättas till, kan leda till att man blir *godkänd trots bristerna.*

Det kan hända att implementationen inte är färdigställd den 17/12. En ofärdig implementation bör 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.

Make skall användas för att underlätta egna utvecklingen, d.v.s. bygga med beroenden, exekvera tester, etc. Därutöver skall det finnas en regel `final` som skall bygga all kod i imalloc till en objektkodsfil `imalloc.o` som sedan enkelt kan länkas in med ett testprogram av rättarna.

Bedömningskriterier

Projektet/teamet bedöms på:

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

Observera att det inte är ett strikt krav att ha ett fullt fungerande system den 17/12 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, dvs. $3\frac{1}{3}$ ”vanliga” heltidsveckor[5]). 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.

All inlämnad kod kommer att testas mot ett antal ospecificerade testfall. Det måste därför finnas en fungerande makefile och koden måste kunna kompileras utan handpåläggning på institutionens Solaris-maskiner (både på X86 och Sparc-arkitekturerna). Något enstaka testfall kommer att lämnas ut mot slutet av projekttiden.

[5] Notera att en heltidsvecka på utbildningen är ca 50 timmar eftersom terminerna är kortare än 20 veckor.
[6] Inklusive utelämnade funktioner

Observera: Det är ett strikt krav att använda verktyg av typen valgrind för att verifiera avsaknaden av minnesläckage.

Rest

En ofullständig inlämning vid 17/12 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 – 10:e januari 2014*. 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.

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!

Planering och uppföljning

Under projektet skall ni *aktivt* använda er av verktyget Trello (`http://trello.com`). Ni ansvarar själva för att sätta upp ett ”bräde” på Trello med lämpliga rättigheter, och bjuda 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.[8] Om något strul uppstår kommer vi att använda Trello för att spåra arbetet så det ligger i ert intresse att det som sker där faktiskt stämmer överens med verkligheten.

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

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 av er coach, 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 det är viktigt att använda en issue tracker/bug tracker, även om man sitter i samma rum.

Uppgiften

OBSERVERA

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, ”imalloc” för minneshantering. Med funktionen `iMalloc` kan man reservera ett konsekutivt minnesblock[9] i vilket man sedan kan allokera minne med hjälp av biblioteksfunktioner. Det skall finnas stöd för manuell minneshantering, á la vanliga C-malloc, men det skall också gå att ställa in hur den s.k. free-listan skall vara sorterad, vilket påverkar olika programs prestanda på olika sätt[10]. Det skall finnas stöd för referensräkning, och stöd för automatisk skräpsamling med hjälp av en skräpsamlare som traverserar minnet och hittar minne som inte längre är nåbart från programmet[11]. De sista två metoderna kallar vi här för *managed* och den första för *manual*. Ett minnesblock skapat av `iMalloc` är alltid antingen manual eller managed – aldrig båda. Två olika block som existerar samtidigt behöver inte använda samma metoder. Observera att metoden managed kan innebära enbart referensräkning, enbart automatisk skräpsamling, *eller båda samtidigt*.

[9] Med hjälp av malloc i stdlib

[10] Se screencast om minneshantering

[11] En beskrivning av dessa funktioner, F1–F3, finns nedan och samtliga skall alltså implementeras.

F1: Manual, minneshantering med `alloc` och `free`

Beskrevs på screencast om minneshantering, se även utdelad kod. Man skall kunna finjustera beteedet hos allokeringsfunktionen `alloc` genom att ställa in free-listans sortering: snabb allokering på bekostnad av ytterligare fragmentering[12], mindre fragmentering[13], eller högre grad av referenslokalitet[14].

[12] Listan sorterad på minskande storlek

[13] Listan sorterad på ökande storlek

[14] Listan sorterad i adressordning

F2: Managed, referensräknare

Beskrevs på screencast om minneshantering, se även utdelad kod. Notera att cykliska strukturer kan ge upphov till minnesläckage.

Stödet för referensräkning i imalloc implementeras med funktionerna `retain`, `release` samt `count`. Den första räknar upp referensräknaren med 1 för ett objekt[15], den andra räknar ned, och den sista returnerar ett objekts *refcount*.

[15] Från och med nu kallar vi en allokerad strukt i minnet, eller ett minnesblock, för ett objekt.

```
void *p = gc->alloc(gc, sizeof(foo));
gc->refcount->count(p); // refcount = 1
```

```
gc->refcount->retain(p);
gc->refcount->count(p); // refcount = 2
gc->refcount->release(gc, p);
gc->refcount->count(p); // refcount = 1
gc->refcount->release(gc, p); // objektet tas bort
```

F3: Managed, automatisk skräpsamling

Tanken bakom automatisk skräpsamling är att presentera en minnesabstraktion för programmeraren där minnet är oändligt. Detta uppnås genom att skräpsamlaren håller reda på om ett objekt kan deallokeras eller inte, istället för programmeraren. Så länge som minnet räcker till fungerar allokering i stort sett som vanligt, men så fort en allokering misslyckas på grund av att minnet är fullt söker allokeraren igenom allt minne efter objekt som inte längre används, och frigör dessa. Därefter fullföljer man allokeringen, och i regel lyckas detta eftersom de flesta objekt som skapas endast lever en kort tid innan de blir skräp.

I imalloc finns även funktionen `gc` som startar en skräpsamling, oavsett om det behövs eller inte.

Algoritmen Mark-Sweep

Skräpsamlaren som vi skall implementera här skall använda sig av en så kallad ”mark-sweep-algoritm” för att identifiera objekt som säkert kan deallokeras utan att programmet kraschar. Vi gå igenom algoritmen steg-för-steg nedan, men först skall vi diskutera några implementationsdetaljer.

Varje objekt innehåller en extra bit-flagga, den s.k. *mark-biten*. När flaggan är satt (1) anses objektet vara ”vid liv”. Annars är objektet skräp som kan tas bort.

Vid varje skräpsamlingstillfälle sker följande:

Steg 1 Iterera över listan över samtliga objekt på heapen och sätt mark-biten till 0.

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

Steg 3 Iterera över listan över samtliga objekt på heapen och frigör alla vars mark-bit fortfarande är 0.

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

Att traversera heapen

Att traversera heapen i C är inte så enkelt eftersom minnet som standard allokeras utan metadata. T.ex. allokerar anropet

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

plats som rymmer en `BinaryTreeNode`, men C har ingen information om innehållet i detta utrymme, mer än hur stort utrymmet är som `p` pekar på. Rimligtvis har en `BinaryTreeNode` åtminstone två pekare till höger respektive vänster subträd – 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(int)` i detta utrymme som en adress. Om adressen pekar in i den aktuella heapens adressrymd måste vi anse 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), och sedan skall dess utrymme också letas igenom på samma sätt som `BinaryTreeNode`:en. 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. Kod för detta delas ut och innehåller bl.a. en funktion som letar reda på alla rotpekare och för var och en av dem anropar en funktionspekare:

```
/* Scans the stack, CPU registers, and static data to build a
 * root set, R s.t. for all r in R, h->start <= r && r <= h->end.
 * Then, for all r in R, calls f(r, p), where p is just some
 * additional user-provided payload.
 */
void traverseStack(AddressSpace h, MarkFun f, void *p);
```

För att t.ex. skriva ut alla rotpekare skulle man kunna skapa en funktion `printPtr`:

```
void printPtr(void *ptr, void *ignore) {
  printf("%p\n", ptr);
}
```

och anropa `traverseStack` så här:

```
addressspace as;
as.start = ... ; // Bör imalloc hålla reda på
as.end = ... ; // Bör imalloc hålla reda på
traverseStack(&as, printPtr, NULL);
```

Filerna `roots.h` och `roots.c` med koden samt typdefinitioner enligt ovan kommer att delas ut.

Typdeklarationer:

```
typedef char *RawPtr;
typedef struct {
  RawPtr start;
  RawPtr end;
} addressspace, *AddressSpace;

typedef
void (*MarkFun)(void *ptr, void *data);
```

Allokering med metadata

Vårt imalloc-bibliotek skall stödja en version av allokering där programmeraren anger en slags formatsträng[16] som beskriver minneslayouten hos objektet som skall allokeras. Formatsträngen kan sedan användas för att slippa det kostsamma letandet efter pekare som beskrevs ovan.

[16] Analogt med `printf`.

Ponera typen `BinaryTreeNode` deklarerad enligt följande.

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

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

```
alloc("***d");
```

är analogt med

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

En definition av formatsträngen finns på sidan 11 i detta dokument.

Observera att man fortfarande måste identifera rotpekare genom att leta igenom stacken på samma sätt som nämnts i föregående avsnitt. Notera att C-kompilatorer använder ”packing” 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.

[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

Minneshanterare

För att använda imalloc måste man först skapa en minneshanterare. En sådan skapas med funktionen `iMalloc`. Denna funktion tar ett antal bitflaggor som parametrar[19] som anger vilken typ av minneshanterare som skall skapas, samt storleken på det minne som skall hanteras och som returnerar sedan en pekare till en strukt. Denna strukt innehåller allt minneshanterarens data, t.ex. en pekare till det minne som hanteras, men också en samling pekare till de funktioner[20] som skall användas för att interagera med minneshanteraren.

[19] Se screencast om bitmanipulering

[20] Se screencast om funktionspekare.

Att alla anrop går via strukten och dess funktionspekare istället för via globala funktioner gör det enkelt att ha flera aktiva minneshanterare samtidigt och biblioteket tillåter detta, t.ex. en minnesarea som hanteras manuellt och en som hanteras med hjälp av en automatisk

skräpsamlare. Pekare mellan objekt i olika minnesareor sker dock på egen risk.

Exempel på skapande av minneshanterare

Följande kodsnutt allokerar ett en megabyte stort minnesutrymme som hanteras manuellt. Variabeln `mem` pekar på den struct som håller i alla funktioner som sedan används för att allokera och avallokera minne. Notera flaggorna `MANUAL` och `ASCENDING_SIZE` som anger att utrymmet skall hanteras manuellt (med anrop till `alloc` och `free`), samt att frilistan skall vara sorterad på minnesblockens storlek.

```
// Allocate 1 megabyte of memory into mem
Manual mem = (Manual) iMalloc(1 Mb, MANUAL + ASCENDING_SIZE);

// Allocate 1 kilobyte of memory from mem into foo
void *foo = mem->alloc(1 kb);

// Free foo
mem->free(foo);
```

Följande kodsnutt anger att det minne som allokeras (2 megabyte) skall hanteras med hjälp av referensräkning, samt en automatisk skräpsamlare. Anropen av `retain` och `release` räknar upp respektive ned ett objekts referensräknare.

```
// Allocate 2 megabytes of memory into mem, sort
// the free list on ascending chunk size and use
// reference counting, plus a copying garbage
// collector for memory management
Managed mem = (Managed) iMalloc(2 Mb, REFCOUNT + GC + ASCENDING_SIZE);

// Allocate 1 kilobyte of memory from mem into
// foo, with an initial reference count of 1
void *foo = mem->alloc(1 kb);

// Retain foo, release foo
mem->rc->retain(foo); // Refcount = 2
mem->rc->release(foo); // Refcount = 1

// Collect garbage (and cycles)
mem->gc->collect(mem);
```

Det sista exemplet använder bara automatisk skräpsamlare och frilistan är sorterad efter stigande minnesadress.

```
// Allocate 2 megabytes of memory into mem, sort
// the free list on memory address and use an
// automatic garbage collector for memory management
Managed mem = (managed) iMalloc(2 Mb, GCD + ADDRESS);

// Allocate 1.5 megabyte of memory from mem into foo,
```

```
// with an initial reference count of 1
void *foo = mem->alloc(1 Mb + 512 Kb);

printf("%d\n", mem->avail()); // 1536 Kb

// Use the typed allocate function to provide the GC with more
// information about object layout
void *bar = mem->gc->alloc("****dd");

// Run the garbage collector
mem->gc->collect(mem);

// Will crash, as refcounts not turned on
mem->rc->retain(foo);
```

Formatsträng för `alloc` i GC

Följande specialtecken kan ingå i en formatsträng till den automatiska skräpsamlarens `alloc`.

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

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"`.

Hjälp att komma igång med implementationen

Som exemplen ovan visar är funktionen `iMalloc` ”startpunkten” för ett klientprograms interaktion med imalloc. Nedan finns en utgångspunkt för er kod – ett försök att visa hur man kan tänka. För enkelhets skulle ignorerar den free-listans sortering och återanvänder (med några modifikationer) utdelad kod från smalloc som finns i `ovningar/c/smalloc.c`. *Denna kod är varken kompilerad eller testad*[21] *– det lämnas som en övning till läsaren.*

[21] =YMMV (your mileage may vary)!

Till att börja med måste vi skapa en datastruktur som hanterar det data vi behöver för att hålla koll på vilket minne som är fritt och vilket som är använt. Här återanvänder vi `Chunk` från smalloc:

```
typedef struct chunk *Chunk;
struct chunk {
  void* start; // pointer to start of memory
  unsigned size; // how big is this chunk
  Chunk next; // pointer to next chunk
  bool free; // true if the chunk is free, else false
};
```

Sedan behöver vi en funktion som skapar och initierar minnet. Vi återanvänder `init` från smalloc:

```
// Stolen and modified from smalloc
Chunk init(unsigned int bytes) {
  char *memory = (char*) malloc(bytes);
  Chunk H = (Chunk) malloc(sizeof(chunk));
  H->start = (void*) memory;
  H->size = bytes;
  H->next = NULL;
  H->free = 1;
  while (bytes) memory[--bytes] = 0;
  return H;
}
```

Sedan behöver vi funktioner för att allokera minne, frigöra minne, och svara på hur mycket ledigt minne som finns. Nedan återanvänder jag kod från smalloc igen. Den viktigaste förändringen är att funktionerna måste ta emot en pekare till minneshanteringens metadata, eftersom vi skall kunna ha flera samtidiga separata minnen, något som inte stöds av smalloc.

```
void *smalloc(Memory mem, chunkSize siz) {
  // Back up one pointer in memory to access the first chunk
  Chunk c = (Chunk) ((char*) mem)-sizeof(void*);
  while (!fits(c, bytes)) c = c->next;
  if (c) {
    return split(c, bytes);
  } else {
    return NULL;
  }
}

unsigned int avail(Memory mem) {
  // Back up one pointer in memory to access the first chunk
  Chunk c = (Chunk) ((char*) mem)-sizeof(void*);
  int avail = 0;
  for (; c; c = c->next)
    if (c->free) avail += c->size;
  return avail;
}

unsigned int sfree(Memory mem, void *ptr) {
  // Back up one pointer in memory to access the first chunk
  Chunk c = (Chunk) ((char*) mem)-sizeof(void*);
  // rest left as an exercise to the reader
}
```

Nu har vi alla funktioner som skall finnas i `manual`-strukten, och vi kan nu skriva `iMalloc`. Notera att eftersom vi använder smalloc för vår implementation kan vi inte stödja olika sortering av free-listan.

```
struct style *iMalloc(unsigned int memsiz, unsigned int flags) {
```

```
  // Ignoring free list ordering in this simple example
  if (flags & MANUAL) {
    // Allocate space for the struct of functions and metadata
    struct private_manual *mgr = malloc(sizeof(private_manual));
    // Allocate the space that the memory manager will manage
    mgr->data = init(memsiz);
    // Install the functions
    mgr->functions->alloc = smalloc;
    mgr->functions->avail = avail;
    mgr->functions->free = sfree;
    return &(mgr->functions);
  }
  // Implement all other cases
  return NULL;
}
```

Sammanfattning

Åtminstone följande publika funktioner skall implementeras. Rimligtvis behövs också ett antal privata funktioner (alltså, som inte syns i headerfilen `imalloc.h`) som sköter arbete bakom kulisserna. Minns att många små funktioner som var och en löser en och endast en uppgift är en betydligt bättre design än få stora funktioner som gör många saker.

1. Funktionen `iMalloc` som skall returnera en datastruktur med pekare till de funktioner som kan användas för att manipulera minnet, samt allt data som minneshanteraren behöver.
2. Funktioner för manuell minneshantering, se strukten `manual`.
3. Funktioner för referensräkning, se strukten `Refcount`.
4. Funktioner för automatisk skräpsamling, se strukten `GC`.

Vanliga missförstånd

M1 Minneshanterarens metadata skall också rymmas i det data som efterfrågas vid anropet till `iMalloc` då minneshanteraren skapas.

M2 Refcount-värde för en bit minne placeras lämpligen i direkt anslutning till detta minne, i likhet med istring-inluppen.

Listning av imalloc.h

```
#ifndef __imalloc_h
#define __imalloc_h

/*
```

```
 * imalloc.h
 *
 * This file contains the public specifications for using the
 * imalloc allocator used as a course project for the 2012 IOOP/M
 * course.
 *
 * Note that this specification might evolve as the project is
 * running, f.ex. due to technical as well as educational bugs and
 * insights.
 *
 * You may NOT alter the definitions in this file.
 *
 */

#define chunk_size unsigned int
#define Kb *1024
#define Mb Kb Kb

/* Enumeration constants used by to define how the freelist should
 * be sorted.
 */
enum { ASCENDING_SIZE = 1, DESCENDING_SIZE = 2, ADDRESS = 4 } FreelistStyle;

/* Enumeration constants used by to specify kind of memory
 * manager.
 */
enum { MANUAL = 8, REFCOUNT = 16, GCD = 32 } MallocStyle;

/* Not mandatory. If you want to support copying GC, you are free
 * to think about how to achieve that!
 */
enum { NON_COPYING = 64, COPYING = 128 } GCStyle;

/* The client's pointer to a memory manager. This must be passed
 * as argument to most functions of imalloc.
 */
typedef struct style *Memory;

/* Types used for function pointers in the memory manager struct.
 */
typedef void *(*RawAllocator)(Memory mem, chunk_size size);
typedef void *(*TypedAllocator)(Memory mem, char* typeDesc);
typedef unsigned int(*Manipulator)(Memory mem, void *ptr);
typedef unsigned int(*Global)(Memory mem);
typedef unsigned int(*Local)(void *ptr);

/* Functions for automatic garbage collecting memory manager
 * (mark-sweep)
 */
typedef struct {
  TypedAllocator alloc;
```

```
  Global collect;
} GC;

/* Functions for reference counting memory manager */
typedef struct {
  Local retain;
  Manipulator release;
  Local count;
} Refcount;

/* Functions for the manual memory manager */
typedef struct {
  RawAllocator alloc;
  Global avail;
  Manipulator free;
} manual, *Manual;

/* Functions for the manual memory manager */
typedef struct {
  RawAllocator alloc;
  Refcount rc;
  GC gc;
} managed, *Managed;

/* Return type specification for iMalloc */
typedef union {
  manual manual;
  managed managed;
} style;


////////////////////// Public Functions ////////////////////

/* Initiates the malloc library to be used. memsiz defines the
 * maximum amount of memory that can be used. flags specifies kind
 * of memory manager and allows fine-tunes some options.
 */
struct style *iMalloc(chunk_size memsiz, unsigned int flags);

#endif
```

Listning av priv_imalloc.h

Notera att imalloc använder en teknik som även användes av istring-biblioteket: `iMalloc`-funktionen returnerar en pekare till `manual`- eller `managed`-delen av nedanstående strukt. Den första pekaren skall användas för att peka på allt metadata som minneshanteraren behöver. Det är tillåtet att lägga till och ändra i denna fil.

```
#ifndef __priv_malloc_h
#define __priv_malloc_h

/*
 * priv_imalloc.h
 *
 * This file contains private specifications for using the imalloc
 * allocator used as a course project for the 2012 IOOP/M course.
 *
 * Note that this specification might evolve as the project is
 * running, f.ex. due to technical as well as educational bugs and
 * insights.
 *
 * You MAY ADD and ALTER the definitions in this file.
 *
 */


/* Actual return type specifications for iMalloc */
struct private_manual {
  void *data;
  manual functions;
};

struct private_managed {
  void *data;
  managed functions;
};

#endif
```