Muistinhallinta

C++-kielen muistinhallintaan liittyvät huolet sijoittuvat kahteen yleiseen leiriin: kuinka saada se toimimaan ja kuinka saada se toimimaan tehokkaasti. Hyvät ohjelmoijat ymmärtävät, että nämä huolet pitäisi käsitellä tuossa järjestyksessä, koska ohjelmasta, joka on häikäisevän nopea ja ällistyttävän pieni, ei ole mitään hyötyä, jos se ei käyttäydy odotetulla tavalla. Asioiden oikeellisuus tarkoittaa useimmille ohjelmoijille muistinvaraamis- ja -vapauttamisrutiinien kutsumista oikein. Se, että saat asiat toimimaan tehokkaasti, tarkoittaa toisaalta usein sitä, että kirjoitat omat versiosi muistinvaraamis- ja -vapauttamisrutiineista. Tällöin on jopa tärkeämpää, että saat ohjelman toimimaan oikein.

Mitä tulee virheettömyyteen, C++-kieli perii C-kieleltä yhden sen suurimmista päänsäryistä, eli mahdolliset muistivuodot. Jopa näennäismuisti, niin ihmeellinen keksintö kuin se onkin, on rajallinen, ja kaikilla ei ensinnäkään edes ole käytössä näennäismuistia.

C-kielessä muistivuoto ilmenee aina silloin, kun `malloc`-funktiolla varattua muistia ei koskaan vapauteta `free`-funktiolla. C++-kielen näyttelijät ovat nimeltään `new` ja `delete`, mutta tarina on melkein sama. Tilanne paranee kuitenkin hieman muodostinfunktioiden olemassaololla. Ne tarjoavat mukavan talletuspaikan `delete`-kutsuille, joita kaikkien olioiden täytyy kutsua silloin, kun ne tuhotaan. Samaan aikaan muodostuu lisää huolehdittavaa, koska `new` kutsuu implisiittisesti muodostinfunktioita ja `delete` tuhoajafunktioita. Lisäksi on vielä olemassa se komplikaatio, että voit sekä luokkien sisällä että ulkopuolella määrittää omat versiosi `operator new`- ja `operator delete` -funktiosta. Tämä mahdollistaa kaikenlaisten virheiden tekemisen. Seuraavat kohdat auttavat yleisimpien virheiden välttämisessä.

Kohta 5: Käytä samaa muotoa `new`- ja `delete`-operaattoreiden vastaavissa käytöissä

Mitä vikaa tässä on?

```
string *stringArray = new string[100];
...
delete stringArray;
```

Kaikki tuntuu olevan kunnossa - `new`-operaattoria käytetään samalla tavalla kuin `delete`-operaattoria - mutta jokin on silti pielessä: ohjelmasi käyttäytyminen on määrittelemätöntä. Enimmillään yhdeksääkymmentäyhdeksää sadasta `stringArray`-taulukon osoittamasta `string`-oliosta tullaan tuskin tuhoamaan kunnolla, koska niiden tuhoajafunktioita ei luultavasti koskaan kutsuta.

Kun käytät `new`-operaattoria, tapahtuu kaksi asiaa. Ensinnäkin, muisti varataan (`operator new` -funktion avulla, tästä minulla on enemmän kerrottavaa Kohdissa 7-10). Toiseksi tähän muistiin kutsutaan yhtä tai useampaa muodostinfunktiota. Kun käytät `delete`-operaattoria, tapahtuu kaksi muutakin asiaa: muistiin kutsutaan yhtä tai useampaa tuhoajafunktiota, sitten muisti vapautetaan (funktion `operator delete` avulla - katso Kohta 8). Iso kysymys `delete`-operaattorin kohdalla on tämä: *kuinka monta oliota* oleilee muistissa odottaen poistamista? Vastaus tähän määrittää sen kuinka montaa tuhoajafunktiota pitää kutsua.

Kysymys on itse asiassa yksinkertaisempi: osoittaako poistettava osoitin yksittäiseen olioon vai oliotaulukkoon? Ainoa tapa, jolla `delete`-operaattori voi tietää tämän, on se, että sinä kerrot sen sille. Jos et käytä hakasulkeita, kun käytät `delete`-operaattoria, `delete` olettaa, että vain yhteen olioon osoitetaan. Muuten se olettaa, että taulukkoon osoitetaan:

```
string *stringPtr1 = new string;
string *stringPtr2 = new string[100];
...
delete stringPtr1;              // poista olio
delete [] stringPtr2;           // poista olio-
                                // taulukko
```

Mitä tapahtuisi, jos käyttäisit muotoa `"[]" stringPtr1`-oliossa? Ratkaisu on määrittelemätön. Mitä tapahtuisi, jos et käyttäisi muotoa `"[]" stringPtr2`-oliossa? Tämä on myös määrittelemätöntä. Tämä on edelleen myös määrittelemätöntä sisäänrakennetuille tyypeille, kuten `int`-tyyppi, vaikka näillä tyypeillä ei ole tuhoajafunktioita. Niinpä sääntö on yksinkertainen: jos käytät muotoa `[]`, kun kutsut `new`-operaattoria, sinun täytyy käyttää muotoa `[]`, kun kutsut `delete`-operaattoria. Jos et käytä `[]`-muotoa, kun kutsut `new`-operaattoria, älä käytä `[]`-muotoa, kun kutsut `delete`-operaattoria.

Tämä on erityisen tärkeä sääntö, joka kannattaa muistaa varsinkin silloin, kun olet kirjoittamassa luokkaa, joka sisältää osoitintietojäsenen, ja tarjoat myös useita muodostinfunktioita, koska tällöin sinun pitää olla huolellinen ja käyttää *samaa muotoa* `new`-operaattorista kaikissa muodostinfunktioissa osoittimen jäsenen alustamiseksi. Jos et tee niin, kuinka voit tietää mitä muotoa `delete`-operaattorista käytetään tuhoajafunktiossasi? Jos haluat tutkia lisää tätä aihetta, lue Kohta 11.

Tämä sääntö on myös tärkeä niille, jotka ovat innokkaita käyttämään `typedef`-komentoa, koska tämä tarkoittaa sitä, että `typedef`-komennon kirjoittajan täytyy dokumentoida, mitä muotoa `delete`-operaattorista pitää käyttää, kun `new`-operaattoria käytetään taikomaan esiin `typedef`-tyyppisiä olioita. Tutki esimerkiksi tätä `typedef`-komentoa:

```
typedef string AddressLines[4];    // henkilön osoitteella
                                   // on 4 riviä, joista
                                   // jokainen on merkki-
                                   // jono
```

Koska `AddressLines` on taulukko, tämän `new`-operaattorin käyttötavan

```
string *pal = new AddressLines;    // huomaa, että "new
                                   // AddressLines" palaut-
                                   // taa string*-tyypin,
                                   // aivan samoin kuin
                                   // "new string[4]"
```

täytyy täsmätä `delete`-operaattorin *taulukko*muodon kanssa:

```
delete pal;                        // määrittelemätön!

delete [] pal;                     // toimii
```

Jos haluat välttää tämän kaltaista sekaannusta, on varmaan paras kieltäytyä `typedef`-komennoista ja siirtyä taulukkotyyppeihin. Tämän pitäisi kuitenkin olla helppoa, koska perus-C++-kirjasto (katso Kohta 49) sisältää `string`- ja `vector`-malleja, jotka vähentävät sisäänrakennettujen taulukoiden tarpeen lähes nollaan. `AddressLines` voitaisiin tässä esimerkiksi määrittää tyyliin `string`-tyyppisten muuttujien `vector`-malli. Eli `AddressLines` voisi olla tyyppiä `vector<string>`.

Kohta 6: Käytä `delete`-operaattoria tuhoajafunktioiden osoitinjäseniin.

Luokat, jotka suorittavat lähes koko ajan muistin dynaamista varaamista, käyttävät `new`-operaattoria muodostinfunktiossa/-funktioissa muistin varaamiseksi, ja myöhemmin tuhoajafunktiossa `delete`-operaattoria muistin vapauttamiseksi. Tätä ei ole liian vaikea toteuttaa, kun aluksi kirjoitat luokan, olettaen tietenkin, että muistat työllistää `delete`-operaattorin kaikkiin jäseniin, joihin olisi voitu osoittaa muistia *missä tahansa* muodostinfunktiossa.

Tilanne tulee kuitenkin vaikeammaksi, kun luokkia ylläpidetään ja laajennetaan, koska ne ohjelmoijat, jotka tekevät muutoksia luokkaan, eivät ehkä ole niitä, jotka kirjoittivat sen alunperin. Kun ottaa nämä ehdot huomioon, on helppo unohtaa, että osoitinjäsenen lisääminen vaatii melkein aina kaikkia seuraavista:

- Osoittimen alustus jokaisessa muodostinfunktiossa. Jos muistia ei varata osoittimeen tietyssä muodostinfunktiossa, osoitin pitäisi alustaa nollalla (eli null-arvoisella osoittimella).
- Olemassa olevan muistin poistaminen ja uuden muistin liittäminen sijoitusoperaattorilla (Katso myös Kohta 17.)
- Osoittimen poistaminen tuhoajafunktiossa.

Jos unohdat alustaa muodostinfunktion osoittimen tai käsitellä sen sijoitusoperaattorissa, ongelma tulee ilmeiseksi kohtalaisen pian, joten käytännössä näillä kysymyksillä ei ole taipumuksena vaivata sinua ruton lailla. Jos osoittimen poisto tuhoajafunktiossa epäonnistuu, mitään ulkoisia oireita ei välttämättä ilmene. Se ilmenee sen sijaan hienoisena muistivuotona, eli hitaasti kasvavana syöpänä, joka lopulta ahmaisee koko osoitetilasi ja ajaa ohjelmasi ennenaikaiseen päätökseen. Koska tämä erityinen ongelma ei yleensä kiinnitä itseensä huomiota, on tärkeää, että muistat asian aina, kun lisäät osoitinjäsenen luokkaan.

Huomaa muuten, että null-arvoisen osoittimen poistaminen on aina turvallista (se ei tee mitään). Täten, jos kirjoitat muodostinfunktioitasi, sijoitusoperaattoreitasi ja muita jäsenfunktioita niin, että luokan jokainen osoitinjäsen joko osoittaa aina voimassa olevaan muistiin tai on tyhjä, voit huoletta käyttää tuhoajafunktiossasi `delete`-operaattoria ilman, että sinun tarvitsee ottaa huomioon sitä, mitä käytit `new`-komennossa kysymyksessä olevaan osoittimeen.

Ei ole mitään syytä heittäytyä fasistiseksi tästä Kohdasta. Et esimerkiksi varmastikaan halua käyttää `delete`-operaattoria osoittimessa, jota ei ole alustettu `new`-operaattorilla, etkä varmaan halua koskaan käyttää `delete`-operaattoria osoittimeen, joka välitettiin alunperin sinulle. Luokkasi tuhoajafunktion ei toisin sanoen pitäisi yleensä käyttää `delete`-operaattoria, paitsi jos luokkasi jäsenet olivat ne jäsenet, jotka käyttivät `new`-operaattoria alunperin.

Kohta 7: Varaudu muistin loppumisen ehtoihin.

Silloin kun `operate new` -funktio ei voi varata pyytämääsi muistia, se aiheuttaa poikkeuksen. (Ennen se palautti arvon 0 ja eräät vanhemmat kääntäjät tekevät vieläkin niin. Jos haluat, niin kääntäjäsi voivat edelleen tehdä niin, mutta siirrän keskustelun tästä tämän Kohdan loppuun.) Tiedät sydämesi sopukoissa, että muistin loppumiseen liittyvien ehtojen käsittely on ainoa oikea moraalinen toimintatapa. Olet samaan aikaan tietoinen siitä tosiasiasta, että asioiden tekeminen tällä tavalla on varsinaista tuskaa. Tämän tuloksena on mahdollista, että jätät aika ajoin tämän kaltaisen käsittelyn tekemättä. Tai aina, ehkä.

Sinun pitää silti idätellä kasvavaa syyllisyydentuntoa. Tarkoitan, että mitä jos `new`-operaattori todellakin *aiheuttaa* poikkeuksen?

Ajattelet varmaan, että yksi järkevä tapa suoriutua tästä asiasta on vetäytyä taaksepäin ja palata viemäriojan päiviin, toisin sanoen käyttämään esikääntäjää. Esimerkiksi tyypillinen lauseparsi C-kielessä on määrittää tyypistä itsenäinen makro, jolla muisti varataan ja sitten varmistaa tarkistamalla, että muistin varaaminen onnistui. Tällainen makro voisi C++-kielessä näyttää tältä:

```
#define NEW(PTR, TYPE)                    \
   try { (PTR) = new TYPE; }              \
   catch (std::bad_alloc&) { assert(0); }
```

(Ihmettelet varmaan, että "Hetkinen! Mikä on tämä `std::bad_alloc` -juttu?" `bad_alloc` on sen tyyppinen poikkeus, jonka `operator new` -funktio muodostaa silloin, kun se ei voi tyydyttää muistin varaamiseen liittyvää pyyntöä, ja `std` on sen nimiavaruuden nimi (katso Kohta 28), jossa `bad_alloc` on määritetty. "Hyvä", jatkat, "mikä sitten on tämä `assert`-makro?" Jos tutkit C-kielen perus-`include`-otsikkotiedostoa `<assert.h>` (tai C++-kielen nimiavaruutta säästävää vastinetta, `<cassert>` - katso Kohta 49), tulet huomaamaan, että `assert` on makro. Makro tarkistaa, onko sen välittämä ilmaisu muu kuin nolla, ja jos ei, se tekee virheilmoituksen ja kutsuu `abort`-funktiota. Se tekee sen vain silloin, kun perusmakroa `NDEBUG` ei ole määritetty, eli debug-tilassa. Tuotantotilassa, eli silloin kun `NDEBUG` *on* määritetty, `assert` ei laajennu minnekään - `void`-lauseeseen. `assert`-makrojen käyttö tarkistetaan täten vain silloin, kun virheitä jäljitetään.)

Tämä `NEW`-makro kärsii yleisestä virheestä, jossa `assert`-makroa käytetään testaamaan tilannetta, joka voi tapahtua tuotannon lähdetekstissä (muisti voi joka tapauksessa loppua koska tahansa), mutta makrossa on myös C++-kielelle erityinen haitta: se epäonnistuu ottamaan huomioon ne lukemattomat tavat, joilla `new`-operaattoria voidaan käyttää. On olemassa kolme yleistä kieliopillista muotoa, joilla voidaan saada uusia olioita, jotka ovat tyyppiä `T`, ja sinun pitää ottaa huomioon poikkeuksien mahdollisuus kaikkien näiden kolmen muodon kohdalla:

```
new T;
new T(muodostinfunktion argumentit);
new T[size];
```

Tämä kuitenkin pelkistää liioitellusti ongelman, koska asiakkaat voivat määrittää omat (kuormitetut) versionsa `operator new` -funktioista, joten ohjelmat voivat sisältää mielivaltaisen määrän erilaisia kieliopillisia muotoja `new`-operaattorin käytöstä.

Kuinka tämän asian kanssa sitten menetellään? Jos olet halukas tyytymään erittäin yksinkertaiseen virheidenhallintastrategiaan, voit määrittää niin, että jos muistiin kohdistuvaa pyyntöä ei voida tyydyttää, määrittelemääsi muistinkäsittelyfunktiota kutsutaan. Tämä strategia perustuu siihen sopimukseen, että silloin kun `operator new` -funktio ei voi tyydyttää pyyntöä, se kutsuu asiakasohjelmalle ominaista virheidenkäsittelyfunktiota - kutsutaan usein nimellä *new-käsittelijä* - ennen kuin se muodostaa poikkeuksen. (Tosiasiassa se, mitä `operator new` todella tekee, on

lievästi sanoen hieman monimutkaisempaa. Yksityiskohdat ovat luettavissa Kohdasta 8.)

Kun asiakasohjelmat määrittelevät muistin loppumiseen liittyvän funktion, ne kutsuvat `set_new_handler`-funktiota, joka on määritetty enemmän tai vähemmän tähän tyyliin `<new>`-otsikossa:

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

Kuten näet, `new_handler` on tyypiltään `typedef`-komento sen funktion osoittimeen, joka ei ota eikä palauta mitään ja `set_new_handler` on funktio, joka ottaa ja palauttaa `new_handler`-arvon.

Funktion `set_new_handler` parametri on osoitin funktioon, jota `operator new` -funktion pitäisi kutsua, jos se ei voi varata pyydettyä muistia. `set_new_handler` -funktion paluuarvo on osoitin funktioon, joka on toiminnassa siihen tarkoitukseen ennen kuin `set_new_handler`-funktiota kutsuttiin.

`set_new_handler`-funktiota käytetään tähän tyyliin:

```
// kuts. funktio jos op. new ei voi varata tarpeeksi muistia
void noMoreMemory()
{
  cerr << "Muistipyyntöä ei voida tyydyttää\n";
  abort();
}

int main()
{
  set_new_handler(noMoreMemory);

  int *pBigDataArray = new int[100000000];

  ...

}
```

Jos, ja kuten tuntuu todennäköiseltä, `operator new` -funktio ei pysty varaamaan tilaa miljoonalle kokonaisluvulle, funktiota `noMoreMemory` kutsutaan, ja ohjelman suoritus keskeytetään virheilmoituksen jälkeen. Tämä on marginaalisesti ottaen parempi tapa päättää ohjelma kuin yksinkertainen dumppaus. (Harkitse muuten, mitä tapahtuu, jos muisti pitää varata dynaamisesti sillä aikaa, kun virheilmoitus pitää kirjoittaa `cerr`-tulostusvirtaan...)

Kun `operator new` -funktio ei voi tyydyttää muistipyyntöä, se ei kutsu new-käsittelijäfunktiota vain kerran, vaan *toistuvasti,* kunnes se *voi* löytää tarpeeksi muistia. Kohdassa 8 nähdään lähdeteksti, joka mahdollistaa nämä toistuvat kutsut, mutta tämä korkean tason kuvaus riittää antamaan sen kuvan, että hyvinsuunnitellun new-käsittelijäfunktion täytyy tehdä jotain seuraavista:

- **Saada lisää muistia saataville.** Tämä sallinee `operator new` -funktion seuraavan yrityksen varata muistia onnistuneesti. Yksi tapa, jolla tämä strategia voidaan toteuttaa, on varata iso muistilohko ohjelman aloitusvaiheessa, ja vapauttaa se sitten, kun new-käsittelijäfunktiota pyydetään avuksi. Tämän kaltaisen vapautuksen mukana seuraa yleensä jonkinlainen käyttäjälle tarkoitettu varoitus, että muistia on vähän ja tulevat pyynnöt voivat epäonnistua, mikäli muistia ei jollain tavalla tule saataville.
- **Asenna eri new-käsittelijä.** Jos nykyinen new-käsittelijä ei pysty saamaan lisää muistia käyttöön, se tietää ehkä toisen new-käsittelijän, joka sisältää enemmän resursseja. Jos näin on, aktiivinen new-käsittelijä voi asentaa toisen new-käsittelijän tilalleen (kutsumalla `set_new_handler`-funktiota). Seuraavalla kerralla, kun `operator new` -funktio kutsuu new-käsittelijäfunktiota, se saa sen, joka on viimeksi asennettu. (Tämän teeman vaihteluna on se, että new-käsittelijä muuttaa *omaa* käyttäytymistään, joten seuraavalla kerralla, kun sitä pyydetään avuksi, se tekee jotain erilaista. Yksi tapa saada tämä aikaan on, että new-käsittelijä muuttaa staattista tai globaalia tietoa, joka vaikuttaa new-käsittelijän käytökseen.)
- **Poista new-käsittelijän asennus**, eli välitä null-arvoinen osoitin `set_new_handler`-funktiolle. Kun new-käsittelijää ei ole asennettu, `operator new` -funktio saa aikaan poikkeuksen, joka on tyyppiä `std::bad_alloc` silloin, kun sen yritys varata muistia epäonnistuu.
- **Muodosta poikkeus,** jonka tyyppi on `std::bad_alloc` tai jokin tyyppi, joka on periytetty `std::bad_alloc`-funktiosta. `operator_new` -funktio ei löydä tämän tyyppisiä poikkeuksia, joten ne leviävät siihen paikkaan, joka on lähtöisin muistia koskevasta pyynnöstä. (Silloin kun muodostetaan poikkeus, joka on eri tyyppiä, vahingoitetaan `operator new` -funktion poikkeusmäritystä. Näissä tapauksissa oletuksena on kutsua `abort`-funktiota, joten jos new-käsittelijäsi aiheuttaa poikkeuksen, haluat ehdottomasti varmistaa, että se on peräisin `std::bad_alloc`-hierarkiasta.)
- **Ei paluuarvoa**, tapahtuu tavallisesti kutsumalla `abort`- tai `exit`-funktiota, nämä molemmat löytyvät perus-C-kirjastosta (ja täten myös perus-C++-kirjastosta, katso Kohta 49).

Näillä valinnoilla saadaan melkoinen joustavuus new-käsittelijäfunktioiden toteuttamiseen.

Haluaisit varmaan joskus käsitellä muistin varaamisen aiheuttamia epäonnistumisia erilaisin tavoin, riippuen varaamisen kohteena olevan olion luokasta:

```
class X {
public:
  static void outOfMemory();

  ...
```

```
};
class Y {
public:
  static void outOfMemory();

  ...
};
X* p1 = new X;          // jos varaus epäonnistuu,
                        // kutsu X::outOfMemory

Y* p2 = new Y;          // jos varaus epäonnistuu,
                        // kutsu Y::outOfMemory
```

C++-kieli ei sisällä tukea luokkakohtaisille new-käsittelijöille, mutta se ei tarvitsekaan. Voit toteuttaa tämän käyttäytymisen itse. Varustat jokaisen luokan omalla versiollaan `set_new_handler`-funktiosta ja `operator new` -funktiosta. Luokan `set_new_handler`-funktio antaa asiakkaiden määrittää luokalle new-käsittelijän (aivan kuten perus-`set_new_handler` antaa asiakkaiden määrittää globaalin new-käsittelijän). Luokan `operator new` -funktio varmistaa, että luokalle ominaista new-käsittelijää käytetään globaalin new-käsittelijän sijasta silloin, kun luokan olioille varataan muistia.

Tutki luokkaa `X`, jolla haluat käsitellä muistin varaamiseen liittyvät epäonnistumiset. Sinun pitää seurata kutsuttavan funktion reittiä silloin, kun `operator new` -funktio ei voi varata tarpeeksi muistia X-tyyppiselle oliolle. Esittelet staattisen jäsenen, joka on tyyppiä `new_handler`, joka osoittaa luokan new-käsittelijäfunktioon. Sinun `X`-luokkasi näyttää tältä:

```
class X {
public:
  static new_handler set_new_handler(new_handler p);
  static void * operator new(size_t size);

private:
  static new_handler currentHandler;
};
```

Luokan staattiset jäsenet täytyy olla määritettyinä luokan määrityksen ulkopuolella. Koska haluat käyttää staattisten olioiden oletuksena olevaa alustustapaa, 0, määrität `X::currentHandler`-funktion ilman alustusta:

```
new_handler X::currentHandler;     // asettaa current-
                                   // Handler-muuttujan
                                   // arvoksi 0 (eli null)
                                   // oletus
```

Luokan `X` `set_new_handler`-funktio tallentaa kaikki sille välitetyt osoittimet. Se palauttaa kaikki osoittimet, jotka on tallennettu ennen kutsua. Tämä on tarkasti ottaen se, mitä `set_new_handler`-funktion perusversio tekee:

```
new_handler X::set_new_handler(new_handler p)
{
  new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}
```

Luokan `X operator new` -funktio tekee lopulta seuraavat asiat:

1. Kutsuu perus-`set_new_handler`-funktiota `X`-luokan virheenkäsittelyfunktiolla. Tämä asentaa `X`-luokan new-käsittelijän globaalina new-käsittelijänä. Huomaa, kuinka `std`-nimiavaruuden näkyvyysalueeseen (siihen, missä perus-`set_new_handler` sijaitsee) viitataan allaolevassa lähdetekstissä eksplisiittisesti käyttämällä esitystapaa `"::"`.
2. Kutsuu globaalia `operator new` -funktiota pyydetyn muistin varsinaiseen varaamiseen. Jos alustava yritys varata muistia epäonnistuu, globaali `operator new` -funktio pyytää avuksi `X`-luokan new-käsittelijää, koska tuo funktio asennettiin juuri globaaliksi new-käsittelijäksi. Jos globaali `operator new` -funktio ei viime kädessä pysty löytämään tapaa varata pyydettyä muistia, se muodostaa `std::bad_alloc`-poikkeuksen, jonka `X`-luokan `operator new` -funktio tunnistaa. `X`-luokan `operator new` -funktio palauttaa sitten sen globaalin new-käsittelijän, joka oli alunperin sen paikalla ja se palauttaa muodostamalla poikkeuksen.
3. Jos oletetaan, että globaali `operator new` -funktio pystyi menestyksellisesti varaamaan tarpeeksi muistia oliolle, joka on tyyppiä `X`, `X`-luokan `operator new` -funktio kutsuu uudelleen tavallista `set_new_handler`-funktiota palauttaakseen globaalin virheenkäsittelyfunktion siihen, mikä se alunperin oli. Se palauttaa sitten osoittimen varattuun muistiin.

Tässä on tapa, jolla asia hoidetaan C++-kielellä:

```
void * X::operator new(size_t size)
{
  new_handler globalHandler =              // asenna X:n
    std::set_new_handler(currentHandler);  // käsittelijä

  void *memory;

  try {                                    // yritä
    memory = ::operator new(size);         // varaamista
  }
  catch (std::bad_alloc&) {                // palauta
    std::set_new_handler(globalHandler);   // käsittelijä;
    throw;                                 // aiheuta
  }                                        // poikkeus
```

```
  std::set_new_handler(globalHandler);     // palauta
                                           // käsittelijä
  return memory;
}
```

Luokan X asiakkaat käyttävät sen new-käsittely-taitoja tähän tyyliin:

```
void noMoreMemory();              // funktion esittely jota
                                  // kutsutaan, jos X-
                                  // olioiden muistin
                                  // varaaminen epäonnistuu
X::set_new_handler(noMoreMemory);
                                  // aseta noMoreMemory X:n
                                  // new-käs.funktioksi
X *px1 = new X;                   // jos muistin varaaminen
                                  // epäonnistuu, kutsu
                                  // noMoreMemory-funktiota
string *ps = new string;          // jos muistin varaaminen
                                  // epäonnistuu, kutsu
                                  // globaalia new-käsitte-
                                  // lijäfunktiota
                                  // (jos sellainen on)
X::set_new_handler(0);            // aseta X-kohtainen
                                  // new-käsitt.funktion
                                  // arvoksi 0(eli null)
X *px2 = new X;                   // jos muistin varaaminen
                                  // epäonnistuu, muodosta
                                  // poikkeus välittömästi
                                  // (Luokalle X ei ole
                                  // new-käsitt.funktiota)
```

Huomaat varmaan, että tämän skeeman toteuttamiseen tarvittava koodi on sama riippumatta luokasta, joten olisi järkevää käyttää sitä uudelleen muissa paikoissa. Kuten Kohdassa 41 selvitetään, uudelleen käytettävissä olevan lähdetekstin luomiseksi voidaan käyttää sekä periytyvyyttä että malleja.

Sinun tarvitsee vain luoda "sekoitustyylinen" kantaluokka, toisin sanoen kantaluokka, joka on suunniteltu niin, että se sallii periytettyjen luokkien periä yksittäinen tietty taito - tässä tapauksessa taito asettaa luokkakohtainen new-käsittelijä. Kantaluokka palautetaan tämän jälkeen malliin. Työtavan kantaluokka-osa antaa periytettyjen luokkien periä `set_new_handler`- ja `operator new` -funktiot, joita ne kaikki tarvitsevat, kun taas työtavan malliosa varmistaa sen, että jokainen periytyvä luokka saa eri `currentHandler`-tietojäsenen. Tulos voi kuulostaa hieman monimutkaiselta, mutta tulet huomaamaan, että lähdeteksti näyttää rauhoittavan tutulta. Itse asiassa melkein ainoa todellinen ero on siinä, että mikä tahansa luokka voi halutessaan käyttää sitä uudelleen:

```
template<class T>              // "sekoitus"kantaluokka
class NewHandlerSupport {      // luokkakohtaiselle
public:                        // set_new_handler-tuelle

  static new_handler set_new_handler(new_handler p);
  static void * operator new(size_t size);

private:
  static new_handler currentHandler;
};

template<class T>
new_handler NewHandlerSupport<T>::set_new_handler(new_handler p)
{
  new_handler oldHandler = currentHandler;
  currentHandler = p;
  return oldHandler;
}

template<class T>
void * NewHandlerSupport<T>::operator new(size_t size)
{
  new_handler globalHandler =
    std::set_new_handler(currentHandler);

  void *memory;

  try {
    memory = ::operator new(size);
  }
  catch (std::bad_alloc&) {
    std::set_new_handler(globalHandler);
    throw;
  }

  std::set_new_handler(globalHandler);

  return memory;
}

// tämä asettaa jokaisen currentHandlerin arvoksi 0
template<class T>
new_handler NewHandlerSupport<T>::currentHandler;
```

`set_new_handler`-funktion tuen lisääminen luokkaan `X` on helppoa tällä luokkamallilla: `X` vain periytyy `newHandlerSupport<X>`-mallista:

```
class X: public NewHandlerSupport<X> { // huomaa
periytyvyys
                                       // sekatyylisestä
                                       // luokkamallista

  ...                // kuten ennenkin, mutta ei esitt.
};                   // set_new_handler tai operator new
```

Luokan `X` asiakkaat ovat edelleen tietämättömiä kaikesta kulissien takana tapahtuvasta toiminnosta; heidän vanha lähdetekstinsä toimii edelleen. Tämä on hyvä, koska asiakkaiden tietämättömyyteen voidaan tavallisesti luottaa.

`set_new_handler`-funktion käyttäminen on mukava ja helppo tapa suoriutua siitä tilanteesta, että muisti loppuu. Se on varmasti paljon viehättävämpää kuin jokaisen `new`-operaattorin käytön kietominen `try`-lohkoon. Lisäksi `NewHandler-Supportin` kaltaisten mallien avulla on helppoa lisätä luokkakohtaisia new-käsittelijöitä jokaiseen sitä haluavaan luokkaan. Sekatyylinen periytyvyys johtaa kuitenkin muuttumattomasti moniperintä-aiheeseen, ja ennen kuin haluat joutua tähän liukumäkeen, haluat varmasti lukea Kohdan 43.

C++-kieli edellytti vuoteen 1993 asti, että `operator new` -funktio palauttaa arvon 0 silloin, kun se ei pysty tyydyttämään muistipyyntöä. Nykyään `operator new` -funktio aiheuttaa `std::bad_alloc`-poikkeuksen, mutta C++-kielistä lähdetekstiä oli kirjoitettu paljon ennen kuin kääntäjät alkoivat tukea muutettua määritystä. C++-kielen standardointikomitea ei halunnut hylätä vakiintunutta 0-arvon testaukseen perustuvaa lähdetekstikantaa, joten se tarjosi erilaisia muotoja `operator new-` ja `operator new[]` -funktioista (katso Kohta 8). Nämä sisältävät edelleen perinteisen "epäonnistuminen aiheuttaa nollan" -käyttäytymisen. Näitä muotoja kutsutaan nimellä "nothrow", koska ne eivät käytä `throw`-lauseita, ja niissä sovelletaan `no-throw`-attribuutin tyyppisiä olioita (ne on määritetty standardi otsikkotiedostossa `<new>` kohdassa, jossa `new`-operaattoria käytetään):

```
class Widget { ... };

Widget *pw1 = new Widget;       // aiheuttaa std::bad_alloc
                                // jos varaus epäonnistuu

if (pw1 == 0) ...               // tämän testin täytyy
                                // epäonnistua

Widget *pw2 =
  new (nothrow) Widget;         // palauttaa 0 jos varaus
                                // epäonnistuu

if (pw2 == 0) ...               // tämä testi voi onnistua
```

Riippumatta siitä käytätkö "normaalia" (eli poikkeuksen aiheuttavaa) `new`-operaattoria tai `nothrow`-attribuutin tyyppistä `new`-operaattoria, on tärkeää, että varaudut käsittelemään muistinvaraamiseen liittyvät epäonnistumiset. Helpoin tapa toteuttaa tämä on hyötyä `set_new_handler`-funktiosta, koska se toimii molemmissa muodoissa.

Kohta 8: Noudata sopimusta, kun kirjoitat `operator new` - ja `operator delete` -funktioita.

Kun ryhdyt kirjoittamaan `operator new` -funktioita (Kohdassa 10 selvitetään, miksi sinun kannattaa tehdä näin), on tärkeää, että funktiosi tarjoavat käytöksen, joka

on yhdenmukainen oletuksena olevan `operator new` -funktion kanssa. Käytännössä tämä tarkoittaa sitä, että palautat oikean arvon, kutsut virheenkäsittelyyn tarkoitettua funktiota silloin, kun muistia ei ole tarpeeksi saatavilla (katso Kohta 7) ja varaudut suoriutumaan pyynnöistä, joille ei ole tarpeeksi muistia. Sinun kannattaa myös välttää sitä, että piilotat huomaamatta "tavallisen" muodon `new`-operaattorista, mutta tälle asialle on oma Kohtansa 9.

Paluuarvon sisältävä osa on helppo. Jos pystyt toimittamaan pyydetyn muistin, palautat vain osoittimen siihen. Jos et voi, noudata Kohdassa 7 kuvattua sääntöä, ja aiheuta poikkeus, joka on tyyppiä `std::bad_alloc`.

Tämä ei kuitenkaan ole aivan niin yksinkertaista, koska `operator new` -funktio yrittää kuitenkin itse asiassa varata muistia enemmän kuin kerran kutsumalla virheidenkäsittelyn funktiota jokaisen epäonnistumisen jälkeen, ja olettamuksena on se, että virheidenkäsittelyfunktio voisi pystyä tekemään jotain vapauttaakseen hieman muistia. Vain silloin, kun virheidenkäsittelyfunktion osoitin on arvoltaan `null`, `operator new` -funktio aiheuttaa poikkeuksen.

Perus-C++-kieli vaatii lisäksi sen, että `operator new` -funktio palauttaa laillisen osoittimen myös silloin, kun pyydettyjen tavujen määrä on arvoltaan 0. (Usko tai älä, tämän oudolta kuulostavan käyttäytymisen edellyttäminen tekee itse asiassa elämästä yksinkertaisempaa muualla kielessä.)

Kun asia on näin, ei-jäsen-tyyppisen `operator new` -funktion pseudokoodi näyttää tällaiselta:

```
void * operator new(size_t size)  // operator new -funktiosi
                                  // voisi ottaa lisä-
                                  // parametrejä
{
  if (size == 0) {                // käsittele 0-tavuiset
                                  // pyynnöt
    size = 1;                     // 1-tavun pyyntöinä
  }

  while (true) {
    yritä varata size-arvon mukaisesti tavuja;

    if (varaus oli onnistunut)
      return (osoitin muistiin);

    // muistin varaus epäonnistui; ota selvää mikä on sen-
    // hetkinen virheenkäs. funktio (katso Kohta 7)
    new_handler globalHandler = set_new_handler(0);
    set_new_handler(globalHandler);

    if (globalHandler) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

Temppu, jolla hoidetaan nolla-tavuihin liittyvät pyynnöt siten, kuin ne olisivat yhden tavun pyyntöjä, näyttää limaiselta, mutta on yksinkertainen, laillinen ja se toimii, ja kuinka usein muuten luulet, että sinulta pyydetään nolla tavua?

Katsot varmasti myös hieman halveksuvasti pseudokoodin siihen kohtaan, jossa virheenkäsittelyfunktion osoittimen arvoksi asetetaan null-arvo, ja sen arvoksi asetetaan viipymättä sen jälkeen alkuperäinen arvo. Virheenkäsittelyfunktion osoittimeen ei valitettavasti päästä mitenkään suoraan, joten sinun täytyy kutsua `set_new_handler`-funktiota saadaksesi selville sen arvon. Julmaa, mutta myös tehokasta.

Kohdassa 7 huomautetaan, että `operator new` sisältää loputtoman silmukan, ja tämä silmukka ilmaistaan edellä olevassa lähdetekstissä eksplisiittisesti - ei ole paljon loputtomampaa kuin `while(true)`-lause. Ainoa tapa päästä silmukasta pois, on varata muisti onnistuneesti tai tehdä new-käsittelyyn liittyvälle funktiolle jokin Kohdassa 7 kuvattu asia: hankkia lisää muistia käytettäväksi, asentaa eri new-käsittelijä, poistaa new-käsittelijä, muodostaa poikkeus, joka on tai on periytetty `std::bad_alloc`-funktiosta, tai epäonnistua arvon palauttamisessa. Nyt pitäisi olla jo aika selvää, miksi new-käsittelijän pitää tehdä yksi näistä asioista. Jos näin ei tapahdu, `operator new` -funktion sisällä oleva silmukka ei lopu koskaan.

Yksi asia, jota monet ihmiset eivät käsitä `operator new` -funktiosta, on se, että se on periytetty aliluokista. Tämä voi johtaa eräisiin aika kiinnostaviin komplikaatioihin. Huomaa edellä olevasta `operator new` -funktion pseudokoodista, että funktio yrittää varata `size`-operaattorin mukaisen arvon tavuissa (paitsi jos `size`-operaattorin arvo on 0). Tämä on täydellisen järkevää, koska se on se muuttuja, joka välitettiin funktiolle. Useimmat luokkakohtaiset versiot `operator new` -funktiosta (mukaan lukien Kohdassa 10 oleva) on kuitenkin suunniteltu *määrätylle* luokalle, *ei* luokalle *tai* millekään sen aliluokalle. Tämä tarkoittaa sitä, että luokan `X` `operator new` -funktiolle sen funktion käyttäytyminen on melkein aina huolellisesti viritetty olioille, joiden koko on `sizeof(X)` - ei yhtään suurempi tai pienempi. Periytyvyyden ansiosta on kuitenkin mahdollista, että kantaluokassa olevaa `operator new` -funktiota kutsutaan muistin varaamiseksi periytetyssä luokassa olevalle oliolle:

```
class Base {
public:
  static void * operator new(size_t size);
  ...
};

class Derived: public Base     // Derived ei esittele
{ ... };                       // operator new

Derived *p = new Derived;      // kuts. Base::operator new!
```

Jos `Base`-luokan luokkakohtaista `operator new` -funktiota ei ole suunniteltu selviytymään tästä - ja hyvin mahdollista on, että sitä ei ole - paras tapa hoitaa tilanne

on kuolettaa kutsut, jotka pyytävät "vääränkokoista" määrää muistia, perus- `operator new` -funktiolle, tähän tyyliin:

```
void * Base::operator new(size_t size)
{
  if (size != sizeof(Base))        // jos koko on "väärä",
    return ::operator new(size);   // anna perus- operator
                                   // new -funktion hoitaa
                                   // pyyntö
  ...                              // hoida pyyntö
                                   // muuten täällä
}
```

"Hetkinen!" kuulen sinun valittavan, "Unohdit tarkistaa sen patologisen-mutta-kuitenkin-mahdollisen-tapauksen, jossa `size`-operaattorin arvo on nolla!" Itse asiassa en unohtanut ja pyydän, että et käytä väliviivoja silloin, kun valitat. Testi on edelleen tallella, se on vain sisällytetty testiin, jossa verrataan `size`-operaattorin arvoa `sizeof(Base)`-muuttujaan. Tutkimattomat ovat C++-standardin tiet, ja yksi tapa on määrittää asiat niin, että kaikki itsenäiset luokat ovat kooltaan jotain muuta kuin nolla. `sizeof(Base)`-operaattorin arvo ei määrityksen mukaisesti koskaan voi olla 0 (vaikka sillä ei ole jäseniä), joten jos `size` on arvoltaan nolla, pyyntö tullaan välittämään `::operator new` -funktiolle, ja vastuu pyynnön hoitamisesta järkevällä tavalla siirtyy funktiolle.

Jos haluat hallita taulukoiden muistin varaamista luokkakohtaisin perustein, sinun täytyy toteuttaa `operator new` -funktion taulukolle erityinen serkku, `operator new[]`. (Tätä funktiota kutsutaan yleensä nimellä "array new", koska on vaikeaa hahmottaa, kuinka lausutaan "operator new[]".) Jos päätät kirjoittaa `operator new[]` -funktion, muista, että olet itse asiassa varaamassa raakaa muistia - et voi tehdä mitään niille taulukon olioille, joita ei ole vielä olemassa. Et itse asiassa vielä edes pysty hahmottamaan, kuinka monta oliota taulukossa tulee olemaan, koska et tiedä, kuinka iso kukin olio on. Kantaluokan `operator new[]` -funktiotahan voidaan kuitenkin kutsua periytyvyyden kautta varaamaan muistia taulukolle, joka koostuu periytetyistä luokkaolioista, ja periytetyt luokkaoliot ovat yleensä suurempia kuin kantaluokan oliot. Tästä johtuu, että et voi `Base::operator new[]` -funktion sisällä olettaa, että jokaisen taulukkoon menevän olion koko on arvoltaan `sizeof(Base),` ja tämä tarkoittaa sitä, että et voi olettaa, että taulukossa olevien olioiden määrä on arvoltaan (*`pyydetyt tavut`*`)/sizeof(base)`.

Tämä riittänee sopimuksista, joita tarvitaan kirjoitettaessa `operator new` (ja `operator new[]`) -funktioita. Mitä tulee `operator delete` -funktioon (ja sen taulukkovastineeseen, `operator delete[]` -funktioon), asiat ovat yksinkertaisemmin. Sinun tarvitsee vain muistaa, että C++-kieli takaa sen, että null-arvoisen osoittimen poistaminen on aina turvallista, joten kaikki kunnia näille takuille. Tässä on pseudokoodi, joka on tarkoitettu ei-jäsen-tyyppiselle `operator delete` -funktiolle:

```
void operator delete(void *rawMemory)
{
  if (rawMemory == 0) return;   // älä tee mitään jos null
                                // osoitin poistetaan

  vapauta muisti, johon rawMemory osoittaa;

  return;
}
```

Tämän funktion jäsenversio on myös yksinkertainen, paitsi että sinun pitää varmistaa, että tarkistat poistettavan koon. Jos oletetaan, että luokkakohtainen `operator new` -funktio välittää `::operator new` -funktiolle pyynnöt, jotka ovat "väärää" kokoa, niin sinun täytyy välittää "vääränkokoiset" poistopyynnöt `::operator delete` -funktiolle:

```
class Base {                    // sama kuin ennen, paitsi
public:                         // op. delete on esitelty
  static void * operator new(size_t size);
  static void operator delete(void *rawMemory, size_t size);
  ...
};

void Base::operator delete(void *rawMemory, size_t size)
{
  if (rawMemory == 0) return;       // tark. null-osoitin

  if (size != sizeof(Base)) {       // jos koko on "väärä",
    ::operator delete(rawMemory);   // anna perus- operator
    return;                         // deleten hoitaa pyyntö
  }

  vapauta muisti, johon rawMemory osoittaa;

  return;
}
```

Funktioita `operator new` ja `operator delete` (ja niiden taulukkovastineita) koskevat sopimukset eivät välttämättä ole kovin hankalia, mutta on tärkeää noudattaa niitä. Jos muistinvaraamisrutiinisi tukevat new-käsittelijäfunktioita ja ne käsittelevät oikeaoppisesti pyynnöt, jotka ovat arvoltaan nolla, sinulla on asiat hyvin, ja jos muistinvapauttamisrutiinisi suoriutuvat null-arvoisista osoittimista, ei ole paljon lisättävää. Lisää funktiosi jäsenversioille tuki periytyvyyttä varten ja *presto!* - olet valmis.

Kohta 9: Vältä `new`-operaattorin "normaalin" muodon piilottamista.

Sisäisessä näkyvyysalueessa olevan nimen esittely piilottaa saman nimen ulkoisissa näkyvyysalueissa, joten funktion `f` kohdalla jäsenfunktio piilottaa globaalin funktion sekä globaalissa että luokan näkyvyysalueessa:

```
void f();                       // globaali funktio

class X {
public:
  void f();                     // jäsenfunktio
};

X x;

f();                            // kutsuu globaalia f

x.f();                          // kutsuu X::f
```

Tämä ei ole yllättävää eikä tavallisesti aiheuta hämmennystä, koska globaaleja funktioita pyydetään yleensä avuksi käyttämällä erilaisia kieliopillisia muotoja. Jos kuitenkin lisäät tähän luokkaan sen `operator new` -funktion, joka vastaanottaa ylimääräisiä parametrejä, tulos avaa luultavasti silmät:

```
class X {
public:
  void f();

  // operator new sallii
  // new-käsittelijäfunktion
  // määrityksen
  static void * operator new(size_t size, new_handler p);
};

void specialErrorHandler();     // määritys on muualla

X *px1 =
  new (specialErrorHandler) X;  // kutsuu X::operator new

X *px2 = new X;                 // virhe!
```

Kun esittelet luokan sisällä funktion nimeltä "operator new", estät huomaamatta pääsyn `new`-operaattorin "normaalin" muodon sisälle. Syy tähän käsitellään Kohdassa 50. Olemme tässä yhteydessä kiinnostuneempia hahmottamaan sen, kuinka ongelma voidaan välttää.

Yksi ratkaisu on kirjoittaa luokkakohtainen `operator new` -funktio, joka tukee "normaalia" avuksi pyydettyä muotoa. Jos se tekee saman kuin globaali versio, se voidaan tehokkaasti ja hienostuneesti kapseloida avoimena funktiona:

```
class X {
public:
  void f();

  static void * operator new(size_t size, new_handler p);

  static void * operator new(size_t size)
  { return ::operator new(size); }
};
```

```
X *px1 =
  new (specialErrorHandler) X;  // kutsuu X::operator
                                // new(size_t, new_handler)
X* px2 = new X;                 // kutsuu X::operator
                                // new(size_t)
```

Vaihtoehtona on tarjota oletuksena oleva parametrin arvo (katso Kohta 24) jokaiselle `operator new` -funktioon lisättävälle ylimääräiselle parametrille:

```
class X {
public:
  void f();

  static
    void * operator new( size_t size,          // huomaa p:n
                         new_handler p = 0);   // oletusarvo
};

X *px1 = new (specialErrorHandler) X;// fine

X* px2 = new X;                              // toimii myös
```

Jos päätät myöhemmin muokata `new`-operaattorin "normaalin" muodon käyttäytymistä, sinun pitää molemmissa tapauksissa vain kirjoittaa funktio uudelleen; kutsujat vastaanottavat muutetun käyttäytymisen automaattisesti, kun he linkittävät ohjelman uudelleen.

Kohta 10: Jos kirjoitat `operator new` -funktion, kirjoita myös `operator delete` -funktio.

Hellitetään hetkeksi ja palataan perusasioihin. Miksi joku ensinnäkään haluaisi kirjoittaa oman versionsa `operator new`- tai `operator delete` -funktiosta?

Vastaus on useimmiten: tehokkuus. `operator new-` ja `operator delete` -funktioiden oletusversiot vastaavat täydellisesti tarkoitustaan yleisluonteiseen käyttöön, mutta niiden joustavuus antaa väistämättömästi tilaa suorituksen kohdennukselle määritetymmässä asiayhteydessä. Tämä on totta varsinkin niille sovelluksille, joka varaavat dynaamisesti suuren joukon pieniä olioita.

Tutki esimerkkinä lentokoneita edustavaa luokkaa, jossa `Airplane`-luokka sisältää vain osoittimen lentokone-olioiden varsinaiseen esittelyyn (tekniikka käsitellään Kohdassa 34):

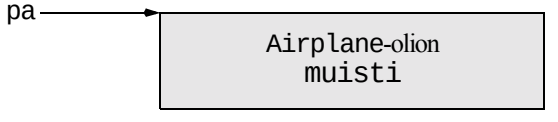
```
class AirplaneRep { ... };      // Airplane-olion
                                // esitys
class Airplane {
public:
  ...
private:
  AirplaneRep *rep;             // osoitin esitykseen
};
```

`Airplane`-olio ei ole kovin iso; se sisältää vain yhden osoittimen. (Kuten selvitettiin Kohdassa 14, olio voi implisiittisesti sisältää toisen osoittimen, jos `Airplane`-luokka esittelee virtuaalifunktiot.) Kun varaat muistia `Airplane`-oliolle kutsuen `operator new` -funktiota, saat kuitenkin melko varmasti enemmän kuin tarpeeksi muistia tämän osoittimen (tai osoitinparin) tallentamiseen. Syyn tähän näennäisesti arvaamattomaan käyttäytymiseen täytyy liittyä `operator new-` ja `operator delete` -funktioiden tarpeeseen kommunikoida keskenään.

Koska `operator new` -funktion oletusversio on yleiskäyttöinen muistivaraaja, sen täytyy olla varautunut varaamaan muistilohkoja, jotka ovat minkä tahansa kokoisia. `operator delete` -funktion oletusversion täytyy vastaavasti olla varautunut vapauttamaan minkä tahansa kokoisia `operator new` -funktion varaamia muistilohkoja. Jotta `operator delete` -funktio tietäisi, kuinka paljon muistia täytyy vapauttaa, sillä täytyy olla jokin keino, kuinka paljon muistia `operator new` on ensinnäkin varannut. Yleinen tapa, jolla `operator new` kertoo `operator delete` -funktiolle, kuinka paljon muistia se on varannut, on teeskennellä muistille, että se palauttaa jotain ylimääräistä tietoa, joka määrittää varatun muistilohkon koon. Tämä tarkoittaa sitä, että kun kirjoitat

```
Airplane *pa = new Airplane;
```

et välttämättä saa takaisin muistilohkoa, joka näyttää tältä:



Saat sen sijaan usein muistilohkon, joka näyttää ennemminkin tältä:

`Airplane`-luokan kaltaisille pienille olioille tämä kirjanpidossa oleva ylimääräinen tieto voi enemmän kuin kaksinkertaistaa sen muistin määrän, jota tarvitaan jokaiselle dynaamisesti varatulle oliolle (varsinkin jos luokka ei sisällä virtuaalifunktioita).

Jos kehität ohjelmistoja ympäristöön, jossa muisti on kallisarvoista, voi olla, että sinulla ei ole varaa tämänkaltaiseen tuhlarimaiseen muistin varaamiseen. Kun kirjoitat oman `operator new` -funktion `Airplane`-luokalle, voit hyötyä siitä tosiasiasta, että kaikki `Airplane`-oliot ovat saman kokoisia, joten ei ole mitään tarvetta pitää kirjaa jokaisessa varatussa lohkossa säilytetystä tiedosta.

Luokkakohtainen `operator new` voidaan myös toteuttaa pyytämällä oletuksena olevalta `operator new` -funktiolta isoja lohkoja raakaa muistia, joista jokainen lohko on riittävän kokoinen säilyttämään suuren määrän `Airplane`-olioita. `Airplane`-olioiden omat muistimöhkäleet otetaan näistä isoista lohkoista. Käyttämättömät möhkäleet järjestetään linkitettyihin listoihin - *vapaa lista* - jotka koostuvat möhkäleistä, jotka ovat käytettävissä tulevaan `Airplane`-käyttöön. Tämä voi kuulostaa hieman siltä, että sinun täytyisi maksaa jokaisen olion `next`-tyyppisen kentän kustannuksia (tukeaksesi listaa), mutta näin ei ole: `rep`-tyyppisen kentän vaatima tila (joka on välttämätön vain `Airplane`-olioiden käyttämille muistimöhkäleille) toimii myös paikkana, jonne tallennetaan `next`-osoitin (koska tuo osoitin tarvitaan vain niille muistimöhkäleille, joita Airplane-oliot *eivät* käytä). Järjestäydyt tähän työnjakoon tavallisella tavalla: käytät `union`-avainsanaa.

Jos haluat muuttaa tämän työtavan käytäntöön, sinun täytyy muuttaa `Airplane`-olion määritystä niin, että se tukee omaa muistinhallintaa. Teet sen seuraavalla tavalla:

```
class Airplane {        // muutettu luokka – tukee nyt
public:                 // omaa muistinhallintaa

  static void * operator new(size_t size);

  ...

private:
  union {
    AirplaneRep *rep;          // käytössä oleville
objekteille
    Airplane *next;            // vapaalla listalla
oleville objekteille
  };

  // tämä luokkakohtainen vakio (katso Kohta 1) määrittelee
  // kuinka monta Airplane-oliota
  // sopii isoon muistilohkoon;
  // se on alustettu alla
  static const int BLOCK_SIZE;

  static Airplane *headOfFreeList;

};
```

Olet tässä lisännyt esittelyt `operator new` -funktiolle eli liittoon, joka sallii `rep`- ja `next`-tyyppisten kenttien varata saman muistin, ja luokkakohtaisen vakion, joka määrittelee, kuinka iso jokaisen varatun lohkon pitäisi olla. Staattinen osoitin pitää lukua vapaan listan yläosasta. Tälle viimeiselle tehtävälle on tärkeää käyttää staattista jäsentä, koska koko *luokalle* on käytettävissä vain yksi vapaa lista, eli ei yhtä vapaata listaa jokaiselle `Airplane`-*oliolle*.

Seuraavaksi kirjoitat uuden `operator new` -funktion:

```
void * Airplane::operator new(size_t size)
{
  // lähettää "väärän" kokoisen pyynnön ::operator new()
  // -funktiolle; lisää yksityiskohtia Kohdassa 8
  if (size != sizeof(Airplane))
    return ::operator new(size);

  Airplane *p =               // p on nyt osoitin
    headOfFreeList;           // vapaan listan yläosaan

  // jos p kelpaa, siirrä listan yläosassa oleva
  // vapaan listan seuraavaan elementtiin
  if (p)
    headOfFreeList = p->next;

  else {
    // Vapaa lista on tyhjä. Varaa muistilohko
    // joka on tarpeeksi iso tallentamaan BLOCK_SIZE -arvon
    // mukaisen määrän Airplane-objekteja
    Airplane *newBlock =
      static_cast<Airplane*>(::operator new(BLOCK_SIZE *
                                            sizeof(Airplane)));

    // muodosta uusi vapaa lista linkittämällä muisti-
    // möhkäleet yhteen; ohita 0.-elementti, koska palau-
    // tat sen operator new -funktion kutsujalle
    for (int i = 1; i < BLOCK_SIZE-1; ++i)
      newBlock[i].next = &newBlock[i+1];

    // päätä linkitetty lista null-arvoisella osoittimella
    newBlock[BLOCK_SIZE-1].next = 0;

    // aseta p listan etuosaan, arvoon headOfFreeList
    // sitä heti seuraavaan lohkoon
    p = newBlock;
    headOfFreeList = &newBlock[1];
  }

  return p;
}
```

Jos olet lukenut Kohdan 8, tiedät, että silloin, kun `operator new` -funktio ei pysty tyydyttämään muistipyyntöä, sen oletetaan suorittavan sarjan rituaalisia toimia mukaan lukien new-käsittelijäfunktiot ja -poikkeukset. Yllä tämän kaltaisista toimista ei ole jälkeäkään. Tämä johtuu siitä, että tämä `operator new` saa `::operator new` -funktiolta kaiken muistin, minkä pystyy. Tämä taas tarkoittaa sitä, että tämä `operator new` -funktio voi epäonnistua vain silloin, jos `::operator new` epäonnistuu. Mutta jos `::operator new` epäonnistuu, *sen* täytyy aloittaa taistelu new-käsittelyyn liittyvässä rituaalissa (joka mahdollisesti kulminoituu poikkeuksen muodostamiseen), joten `Airplane`-olion `operator new` -funktion ei myöskään tarvitse tehdä tätä. Toisin sanoen, new-käsittelijä-käytös on tallella, mutta et vain näe sitä, koska se on piilotettu `::operator new` -funktioon.

Kun `operator new` -funktio on määritetty, ainoa jäljellä oleva asia on määrittää `Airplane`-olioiden staattisten tietojäsenten pakolliset määritykset:

```
Airplane *Airplane::headOfFreeList;   // nämä määrityks.
                                      // menevät toteutus-
const int Airplane::BLOCK_SIZE = 512; // tiedostoon, eivät
                                      // otsikkotiedostoon
```

`headOfFreeList`-muuttujaa ei ole tarpeellista asettaa null-arvoiseksi osoittimeksi eksplisiittisesti, koska staattiset jäsenet alustetaan oletusarvoisesti nollalla. `BLOCK_SIZE`-muuttujan arvo määrittää tietysti sen jokaisen muistilohkon koon, jonka saamme `::operator new` -funktiolta.

Tämä versio `operator new` -funktiosta tulee toimimaan erittäin hienosti. Se ei pelkästään käytä paljon vähemmän muistia `Airplane`-olioihin kuin oletuksena oleva `operator new`, vaan se on myös luultavasti nopeampi, ehkä jopa *kahta suuruusluokkaa* nopeampi. Sen ei pitäisi olla yllätys. `Operator new` -funktion yleisen versionhan täytyy sen lisäksi suoriutua erikokoisista muistipyynnöistä, sen täytyy huolehtia sisäisestä ja ulkoisesta pirstoutumisesta ja niin edelleen, siinä missä oma versiosi `operator new` -funktiosta pelkästään käsittelee paria osoitinta linkitetyssä listassa. On helppo olla nopea, kun ei tarvitse olla joustava.

Olemme vihdoin viimein tilanteessa, jossa voimme keskustella `operator delete` -funktiosta. Muistathan `operator delete` -funktion? Tämä Kohta *käsittelee* `operator delete` -funktiota. Tällä hetkellä `Airplane`-luokkasi esittelee `operator new` -funktion, mutta se ei esittele `operator delete` -funktiota. Mieti mitä tapahtuu, kun asiakas kirjoittaa seuraavaa, mikä ei ole erityisen järkevää:

```
Airplane *pa = new Airplane;    // kutsuu funktiota
                                // Airplane::operator new
...
delete pa;                      // kutsuu ::operator delete
```

Jos olet tarkkaavainen kirjoittaessasi tätä koodia, voit kuulla lentokoneen murskautuvan ja palavan, ja ohjelmoijat, jotka tiesivät tämän, itkevät ja valittavat. Ongelma on siinä, että `operator new` -funktio (se, joka on määritetty `Airplane`-oliossa) palauttaa osoittimen muistiin *ilman mitään otsikkotietoa*, mutta `operator delete` -funktio (oletuksena oleva globaali) olettaa, että sille välitetty muisti *sisältää* otsikkotiedon! Tässäpä ainekset onnettomuuteen.

Tämä esimerkki havainnollistaa yleisen säännön: funktioiden `operator new` ja `operator delete` täytyy olla kirjoitettuina yhteistyössä niin, että ne jakavat samat olettamukset. Jos aiot kehittää oman muistinvaraamisrutiinisi, varmista, että kehität myös muistinvapauttamisrutiinin.

Tässä on tapa, jolla ratkaiset `Airplane`-luokan ongelman:

```
class Airplane {       // sama kuin ennen, paitsi nyt
public:                // operator delete on esitelty
  ...
  static void operator delete(void *deadObject,
                              size_t size);
};
// operator delete -funkt. välitetään muistimöhkäle, joka
// vain lisätään vapaista möhkäleistä koostuvan listan
// eteen, jos se on oikean kokoinen
void Airplane::operator delete(void *deadObject,
                               size_t size)
{
  if (deadObject == 0) return;        // katso Kohta 8
  if (size != sizeof(Airplane)) {     // katso Kohta 8
    ::operator delete(deadObject);
    return;
  }
  Airplane *carcass =
    static_cast<Airplane*>(deadObject);
  carcass->next = headOfFreeList;
  headOfFreeList = carcass;
}
```

Koska olit huolellinen `operator new` -funktion kohdalla ja varmistit, että "väärää" kokoa olevat kutsut välitettiin globaalille `operator new` -funktiolle (katso Kohta 8), sinun täytyy osoittaa samanlaista huolellisuutta varmistamalla, että `operator delete` -funktion globaali versio käsittelee "kelvottoman kokoiset" oliot. Jos et olisi ollut huolellinen, olisit joutunut täsmälleen siihen ongelmaan, jonka välttämistä olet niin voimia kysyvästi työstänyt - semanttinen sopimattomuus `new`- ja `delete`-operaattoreiden välillä.

Kiinnostavaa kyllä arvo `size_t`, jonka C++ välittää `operator delete` -funktiolle, voi olla *väärä*, jos poistettava olio periytettiin kantaluokasta, josta puuttui virtuaalinen tuhoajafunktio. Tässä on tarpeeksi syytä varmistaa se, että kantaluokat sisältävät virtuaaliset tuhoajafunktiot, mutta Kohdassa 14 kuvataan toinen, kiistattomasti parempi syy. Tällä hetkellä riittää yksinkertaisesti se, että huomaat, että jos jätät tekemättä kantaluokkien virtuaaliset tuhoajafunktiot, `operator delete` -funktiot eivät ehkä toimi oikein.

Vaikka kaikki toimii ihan hienosti, voin silti nähdä otsasi juonteista, että asia, josta todella olet huolestunut, on muistivuoto. Kaiken sovelluskehitystyössä saamasi kokemuksen avulla et voi olla huomaamatta, että `Airplane`-olion `operator new` -funktio kutsuu `::operator new` -funktiota saadakseen isoja muistilohkoja, mutta `Airplane`-olion `operator delete` -funktio ei onnistu vapauttamaan

näitä lohkoja.† *Muistivuoto! Muistivuoto!* Voin melkein kuulla hälytyskellojen soivan päässäsi.

Kuuntele minua tarkasti: *mitään muistivuotoa ei tapahdu.*

Muistivuoto tapahtuu silloin, kun muistia varataan, ja kaikki siihen muistiin kohdistuvat osoittimet menetetään. Vaikka apuna olisi poistettu jätelajitelma tai jokin muu ylimääräistä kielellinen mekanismi, muistia ei voida pelastaa. Mutta tämä työtapa ei sisällä muistivuotoa, koska muistin kaikkia osoittimia ei yksinkertaisesti voida menettää. Muistin jokainen lohko hajotetaan ensin `Airplane`-olion kokoisiin möhkäleisiin ja nämä möhkäleet sijoitetaan sitten vapaana olevaan listaan. Kun asiakkaat kutsuvat `Airplane::operator`-funktiota, möhkäleet poistetaan vapaasta listasta ja asiakkaat saavat niihin liittyvät osoittimet. Kun asiakkaat kutsuvat `operator delete` -funktiota, möhkäleet laitetaan takaisin vapaaseen listaan. Tämän suunnittelun avulla kaikki muistimöhkäleet ovat käytössä joko `Airplane`-olioina (siinä tapauksessa asiakkaiden velvollisuus on huolehtia muistivuodon välttämisestä) tai ne ovat vapaassa listassa (siinä tapauksessa muistiin on osoitin). Muistivuotoa ei tapahdu.

`Airplane::operator delete` -funktio ei kuitenkaan koskaan vapauta funktion `::operator new` palauttamia muistilohkoja, ja tälle tapahtumalle täytyy olla olemassa *jokin* nimi. Sille on. Olet luonut *muistialtaan*. Jos haluat, voit kutsua tätä semanttiseksi voimisteluksi, mutta muistivuodon ja muistialtaan välillä on selkeä ero. Muistivuoto voi kasvaa loputtomasti, myös silloin, kun asiakkaat käyttäytyvät hyvin, mutta muistiallas ei koskaan kasva isommaksi kuin se muistin enimmäismäärä, jota asiakkaat ovat pyytäneet.

`Airplane`-olion muistinhallintarutiineja ei ole vaikeaa muuttaa niin, että `::operator new` -funktion palauttamat muistilohkot vapautettaisiin automaattisesti sen jälkeen, kun ne eivät enää ole käytössä, mutta on olemassa kaksi syytä, miksi et varmaan halua tehdä niin.

Ensimmäinen koskee sinun todennäköistä motivaatiotasi käydä oman muistinhallintasi kimppuun. On olemassa monta syytä miksi tekisit niin, mutta todennäköisin syy on se, että olet päättänyt, että oletuksena olevat `operator new` ja `operator delete` -funktiot käyttävät liian paljon muistia tai ovat liian hitaita (tai molempia). Kun näin on asia, jokainen lisätavu ja jokainen lisälause, jonka omistat näiden isojen muistilohkojen jäljittämiseen ja vapauttamiseen, tulee suoraan ytimestä: ohjelmasi toimii hitaammin ja käyttää enemmän muistia, kuin jos olisit ottanut käyttöön allasstrategian. Niille kirjastoille ja sovelluksille, joille suorituskyky on bonusta ja altaiden kokojen odotetaan olevan kohtuullisen rajoitettuja, allas-tyyppinen lähestymistapa saattaa olla paras.

---

† Kirjoitan tämän kieltämättömänä tosiasiana, koska epäonnistuin tämän kirjan ensimmäisessä painoksessa tämän asian osoittamisessa, ja *monet* lukijat hirttivät minut laiminlyönnistäni. Mikään ei vedä vertoja muutamalle tuhannelle oikolukijalle, jotka demonstroivat toisen erehtyväisyyden.

Toisen syyn täytyy johtua patologisesta käyttäytymisestä. Oletetaan, että `Airplane`-olion muistinhallintarutiineja muutetaan niin, että olion `operator delete` -funktio vapauttaa minkä tahansa vapaan muistilohkon, jossa ei ole yhtään aktiivista oliota. Tutki seuraavaksi tätä ohjelmaa:

```
int main()
{
  Airplane *pa = new Airplane;    // ensimmäinen varaus: ota
                                  // iso lohko, muodosta vapaa
                                  // lista ja niin edelleen;
  delete pa;                      // lohko on nyt tyhjä;
                                  // vapauta se
  pa = new Airplane;              // oh hoh, ota lohko
                                  // uudelleen, muodosta
                                  // vapaa lista, jne.
  delete pa;                      // ok, lohko on tyhjä
                                  // taas; vapauta se
  ...                             // nyt varmaan ymmärrät...
  return 0;
}
```

Tämä pieni ilkeä ohjelma toimii hitaammin ja käyttää enemmän muistia kuin *oletuksena* olevilla `operator new` ja `operator delete` -funktioilla varustettu, ja nämä versiot funktioista ovat paljon vähemmän allaspohjaisia!

On tietenkin olemassa tapoja toimia tämän tautiopin kanssa, mutta mitä enemmän kirjoitat lähdetekstiä harvinaisille erikoistapauksille, sitä enemmän lähestyt sitä, että toteutat uudelleen muistinhallinnan oletuksena olevat funktiot, ja mikä on lopputulos? Muistiallas ei ole vastaus kaikkiin muistinhallintaan liittyviin kysymyksiin, mutta se on järkevä vastaus moniin niistä.

Itse asiassa se on usein tarpeeksi järkevä vastaus, varsinkin kun olet vaivaantunut siitä, että sinun täytyy toteuttaa se uudelleen eri luokille. Ajattelet itseksesi, että "on varmasti olemassa tapa kääriä kiinteänkokoisen muistinvaraajan käsite pakettiin niin, että se on helposti uudestaan käytettävissä". Niin onkin, vaikka tämä Kohta onkin kuhnaillut tarpeeksi kauan, niinpä jätän yksityiskohdat pelätyn harjoituksen muodossa lukijalle.

Esitän sen sijaan minimaalisen rajapinnan (katso Kohta 18) `Pool`-luokkaan, jossa jokainen `Pool`-tyyppinen olio on muistinvaraaja olioille, joiden koko on määritetty `Pool`-luokan muodostinfunktiossa:

```
class Pool {
public:
  Pool(size_t n);                 // luo varaaja olioille,
                                  // joiden koko on n
```

```
  void * alloc(size_t n);       // Varaa tarpeeksi muistia
                                // yhdelle oliolle; noudata
                                // operator new -funktion
                                // sopimuksia Kappaleesta 8

  void free(void *p, size_t n); // Palauta altaaseen p:n
                                // osoittama muisti;
                                // seuraa operator delete
                                // -sopimuksia Kohdasta 8

  ~Pool();                      // Vapauta kaikki altaassa
                                // oleva muisti
};
```

`Pool`-oliot luodaan, ne suorittavat muistin varaamis- ja vapauttamistoimintoja ja ne tuhotaan tämän luokan ansiosta. Kun `Pool`-olio tuhotaan, se vapauttaa kaiken varaamansa muistin. Tämä tarkoittaa sitä, että nyt on olemassa tapa välttää muistivuodon tapainen käyttäytyminen, jonka `Airplane`-olion funktiot ovat esittäneet. Tämä tarkoittaa myös sitä, että jos `Pool`-luokan tuhoajafunktiota kutsutaan liian pian (ennen kuin kaikki sen muistia käyttävät oliot on tuhottu), eräät oliot huomaavat, että niiden käyttämä muisti riuhtaistaan niiden alta, ennen kuin ne ovat lopettaneet sen käytön. On suorastaan jalomielistä sanoa, että tuloksena oleva käyttäytyminen on määrittelemätöntä.

Jopa Java-ohjelmoijat voivat lisätä omat muistinhallintaominaisuudet `Airplaneen` tällä `Pool`-luokalla ilman turhaa hikeä:

```
class Airplane {
public:
  ...                     // tavalliset Airplane-funktiot

  static void * operator new(size_t size);
  static void operator delete(void *p, size_t size);

private:
  AirplaneRep *rep;       // osoitin esitykseen

  static Pool memPool;    // Airplane-luokkien muistiallas
};                        // katso alla oleva huom. tähän

inline void * Airplane::operator new(size_t size)
{ return memPool.alloc(size); }

inline void Airplane::operator delete(void *p,
                                      size_t size)
{ memPool.free(p, size); }

// luo uusi allas Airplane-olioille; se menee
// luokan toteutustiedostoon
Pool Airplane::memPool(sizeof(Airplane));
```

Tämä on paljon puhtaampi työtapa kuin aikaisempi työtapa, koska `Airplane`-olion luokkaan ei enää ole kytkettynä siihen liittymättömiä yksityiskohtia. Poissa ovat

`union`, vapaan listan yläosa, vakio, joka määrittää kuinka iso jokaisen puhtaan muistilohkon pitäisi olla ja niin edelleen. Ne on kaikki piilotettu `Pool`-luokan sisälle, ja tämä onkin se paikka, missä niiden pitää olla. Anna `Pool`-luokan kirjoittajan huolehtia muistinhallinnan kirjanpidosta. Sinun tehtäväsi on saada `Airplane`-luokka toimimaan kunnolla.

Sivuhuomautuksena sanottakoon, että `Airplane::memPool` on se, johon perus-C++-kielessä viehättävästi viitataan termillä "epäpaikallinen staattinen olio". Sinun on tarkasti huolehdittava siitä, että sellaisia olioita ei ole mahdollista käyttää ennen kuin ne on alustettu. Kaikkiin näihin kysymyksiin löytyy vastaus Kohdasta 47.

Nyt on kiinnostavaa nähdä, kuinka omat muistinhallinnan rutiinisi voivat parantaa ohjelman suoritusta, ja kannattaa katsoa, kuinka rutiinit voidaan kapseloida luokan kuten `Pool` sisälle, mutta älkäämme sokaistuko pääkohdalle. Pääkohta on se, että `operator new`- ja `operator delete` -funktioiden pitää toimia yhdessä, joten jos kirjoitat `operator new` -funktion, varmista, että kirjoitat myös `operator delete` -funktion.