Osa
VI

21. oppitunti

Esikäsittelijä

Lähdekooditiedostoihin kirjoitetaan pääosin C++ -koodia. Ne käännetään kääntäjän toimesta ajettaviksi ohjelmiksi. Ennen kääntäjän osallisuutta ajetaan kuitenkin esikäsittelijä, joka antaa mahdollisuuden ehdolliseen kääntämiseen. Tämän luvun aiheita ovat:

- Mitä ehdollinen kääntäminen on ja kuinka se hoidetaan
- Kuinka kirjoitetaan makroja esikäsittelijää hyödyntäen
- Kuinka esikäsittelijää käytetään virheiden etsintään

Esikäsittelijä ja kääntäjä

Joka kerta kääntäjää ajettaessa ajetaan esikäsittelijä ensin. Esikäsittelijä hakee esikäsittelijäkomennot, jotka alkavat risuaitamerkillä (#). Näiden ohjeiden tarkoituksena on muuttaa komennot lähdekoodiksi. Tuloksena on uusi lähdekooditiedosto: tilapäinen tiedosto, jota ei normaalisti nähdä, mutta jonka voi tallentaa kääntäjän avulla haluttaessa tutkittavaksi.

Kääntäjä ei lue alkuperäistä lähdekooditiedostoa; se lukee esikäsittelijän tulostuksen ja kääntää tuon tiedoston. Komento #include neuvoo esikäsittelijää etsimään tiedoston, jonka nimi on komennon perässä ja kirjoittamaan sen tilapäistiedostoon kyseiseen kohtaan. Kääntäjän alkaessa tutkia lähdekoodia, on lisätty tiedosto jo mukana.

Miltä välitiedosto näyttää?

Lähes kaikissa kääntäjissä on "kytkin", joka neuvoo kääntäjää tallentamaan välitiedoston. Voit asettaa kytkimen joko Ide-ympäristössä tai komentorivillä. Katso kääntäjäsi manuaalista, kuinka kytkin asetetaan, kun haluat tutkia välitiedostoa.

#define-säännön käyttäminen

#define-sääntö määrittää merkkijonon korvauksen. Jos kirjoitat

```
#define BIG 512
```

olet neuvonut esikääntäjää korvaamaan merkkijonon BIG merkkijonolla 512. Kyseessä ei kuitenkaan ole C++ -kielen mukainen merkkijono. Merkit 512 korvataan lähdekoodissa aina, kun valtuutus BIG tulee eteen. Valtuutus on merkkijono, jota voidaan käyttää aina merkkijonon tai vakion tai muun merkkijoukon paikalla. Siksipä kirjoittaessasi

```
#define BIG 512
int myArray[BIG];
```

on esikääntäjän tuottamassa välitiedostossa koodi

```
int myArray[512];
```

Huomaa, että #define-lause on kadonnut. Kaikki esikääntäjää koskevat lauseet ovat poissa välitiedostosta; ne eivät ole lainkaan mukana lopullisessa lähdekoodissa.

#define vakioiden yhteydessä

#define-sääntöä voidaan käyttää vakioiden korvaamisessa. Se ei ole kuitenkaan juuri koskaan hyvä idea, koska #define tekee pelkästään merkkijonokorvauksen eikä tee tyyppitarkastusta. On huomattavasti parempi käyttää const-avainsanaa #definen sijaan.

#definen käyttäminen testaukseen

Toinen tapa käyttää #defineä yksinkertaisesti tietyn merkkijonon määrittämisen kertomiseen. Voit siis kirjoittaa

```
#define BIG
```

Voit myöhemmin testata, onko BIG määritetty ja toimia sen mukaan. Vastaavat esikäsittelijän komennot ovat #ifdef (onko merkkijono määritelty) ja #ifndef (eikö merkkijonoa ole määritelty). Kumpaakin komentoa tulee seurata komento #endif ennen lohkon päättymistä (ennen lopettavaa aaltosulkua).

#ifdef antaa tuloksen TRUE, jos testattava merkkijono on jo määritelty. Voit siis kirjoittaa seuraavasti:

```
#ifdef DEBUG
cout << "Debug defined";
#endif
```

Kun esikäsittelijä lukee koodin #ifdef, se katsoo muodostamastaan taulukosta, onko DEBUG määritelty. Jos on, palauttaa testi tuloksen TRUE ja kaikki seuraavat lauseet ennen seuraavaa #else- tai #endif-lausetta kirjoitetaan välitiedostoon. Jos tulos on FALSE, ei lauseita oteta mukaan.

Huomaa, että #infdef on #ifdef-komennon looginen vastakohta. Komento antaa tuloksen TRUE, jos merkkijonoa ei ole määritetty.

#else-komento

Kuten voit kuvitella, käsite #else voidaan sijoittaa joko #ifdef- tai #ifndef-komennon ja sulkevan "endif-komennon väliin. Listaus 21.1 havainnollistaa noita käsitteitä.

Listaus 21.1. #define-sääntö.

```
1:  #define DemoVersion
2:  #define DOS_VERSION 5
3:  #include <iostream.h>
4:
5:
6:  int main()
7:  {
8:
9:    cout << "Checking on the definitions of DemoVersion, DOS_VERSION”;
10:   cout << “and WINDOWS_VERSION...\n";   _
11:   #ifdef DemoVersion
12:     cout << "DemoVersion defined.\n";
13:   #else
14:     cout << "DemoVersion not defined.\n";
15:   #endif
16:
17:   #ifndef DOS_VERSION
```

```
18:     cout << "DOS_VERSION not defined!\n";
19:   #else
20:     cout << "DOS_VERSION defined as: " << DOS_VERSION << endl;
21:   #endif
22:
23:   #ifdef WINDOWS_VERSION
24:     cout << "WINDOWS_VERSION defined!\n";
25:   #else
26:     cout << "WINDOWS_VERSION was not defined.\n";
27:   #endif
28:
29:   cout << "Done.\n";
30:   return 0;
31: }
```

Tulostus

```
Checking on the definitions of DemoVersion, DOS_VERSION and
WINDOWS_VERSION…
DemoVersion defined.
DOS_VERSION defined as: 5
WINDOWS_VERSION was not defined.
Done.
```

Analyysi

Riveillä 1-2 määritellään DemoVersion ja DOS_VERSION. DOS_VERSION määritetään merkkijonoksi 5. Rivillä 11 testataan DemoVersion määritys ja koska se on määritetty, on testi tosi ja rivin 12 merkkijono tulostetaan.

Rivillä 17 testataan DOS_VERSION. Koska se on määritetty, testi epäonnistuu ja suoritus siirtyy riville 20. Siellä korvataan merkkijono 5 sanalla DOS_VERSION ja kääntäjä näkee lauseen seuraavanlaisena:

```
cout << "DOS_VERSION defined as: " << 5 << endl;
```

Huomaa, että ensimmäistä DOS_VERSION-esiintymää ei korvata, koska se on lainausmerkeissä. Toinen esiintymä sen sijaan korvataan. Kääntäjä näkee siis arvon 5 aivan kuin se olisi kirjoitettu sinne itse.

Lopuksi ohjelma testaa WINDOWS_VERSION-määrityksen rivillä 23. Koska sitä ei ole määritetty, testi antaa tuloksen epätosi ja rivin 26 viesti tulostetaan.

Tiedostojen sisällyttämisen periaatteet

Projekteissa voi olla mukana useita erilaisia tiedostoja. Usein projekti organisoidaan niin, että kullakin luokalla on oma otsikkotiedostonsa (.HPP), jossa on luokan esittely ja oma toteutustiedostonsa (.CPP), joka sisältää luokan metodien lähdekoodin.

main()-funktiosta tulee oma .CPP-tiedostonsa ja kaikki .CPP-tiedostot käännetään .OBJ-tiedostoiksi, jotka sitten linkitetään yhdeksi ohjelmaksi linkittäjän toimesta.

Koska ohjelmat käyttävät monien luokkien metodeita, sijoitetaan useita otsikkotiedostoja kuhunkin tiedostoon. Otsikkotiedostot on lisäksi usein tarve sisällyttää toinen toisiinsa. Esimerkiksi johdetun luokan esittelyn otsikkotiedoston tulee sisältyä sen perusluokan otsikkotiedostoon.

Olettakaamme, että Animal-luokka on esitelty tiedostossa ANIMAL.HPP. Dog-luokan, joka johdetaan Animal-luokasta, tulee sisältää tiedosto ANIMAL.HPP tiedostossaan DOG.HPP tai muutoin ei Dog-luokkaa voida johtaa Animal-luokasta. Myöskin Cat-luokan otsikkotiedosto sisällyttää ANIMAL.HPP-tiedoston samasta syystä. Jos luot metodin, joka käyttää sekä Cat- että Dog-luokkaa, voit joutua tilanteeseen, jossa ANIMAL.HPP sisällytetään kaksi kertaa.

Tällöin syntyy kääntämisvirhe, koska ei ole hyväksyttävää esitellä luokkaa (Animal) kahta kertaa, vaikka esittelyt olisivatkin identtisiä. Tämä ongelma voidaan ratkaista ehdollisilla toimenpiteillä. ANIMAL-otsikkotiedoston alkuun kirjoitetaan seuraavat rivit:

```
#ifndef ANIMAL_HPP
#define ANIMAL_HPP
…   // koko tiedosto tulee tähän
#endif
```

Koodi kertoo seuraavaa: "Jos käsitettä ANIMAL_HPP ei ole määritelty, siirry eteenpäin ja määrittele se nyt". Koko tiedoston sisältö tulee lauseiden #define ja #endif väliin.

Ensimmäisellä kerralla, kun ohjelmaan sisällytetään tämä tiedosto, ohjelma lukee ensimmäisen rivin ja testin tuloksena on tosi; tällöin ei ANIMAL_HPP ole vielä määritelty. Niinpä ohjelma menee eteenpäin ja määrittelee ANIMAL_HPP:n ja sisällyttää sitten koko tiedoston.

Toisella kerralla, kun ohjelmassa on ANIMAL.HPP, on testin tulos epätosi; ANIMAL_HPP on määritelty. Siksi ohjelmassa hypätään seuraavaan #else-lauseeseen (nyt sitä ei ole) tai seuraavaan #endif-lauseeseen. Siten koko tiedoston sisältö tulee ohitetuksi eikä luokkaa esitellä kahdesti.

Määritellyn symbolin (ANIMAL_HPP) nimi ei ole merkitsevä, vaikkakin on hyvä tapa käyttää isoja kirjaimia ja korvata piste (.) alaviivalla.

Huom! Ehdollisten testien käyttäminen ei ole koskaan vahingollista. Usein ne säästävät aikaa vianhakua tehtäessä.

Määrittely komentorivillä

Lähes kaikki C++ -kääntäjät sallivat #define-arvojen antamisen joko komentoriviltä tai IDE-ympäristöstä (usein molemmista). Useimpien kääntäjien kohdalla voitkin jättää pois rivit 1 ja 2 listauksesta 21.1 ja määritellä merkkijonot DemoVersion ja DOS_VERSION komentoriviltä.

On yleistä sijoittaa ohjelmaan erityistä vianhakukoodia, joka on lauseiden #ifdef DEBUG ja #endif välissä. Tällöin voidaan melkein kaikki vianhakukoodi poistaa koodista helposti lopullista versiota käännettäessä: käsite DEBUG jätetään vain määrittelemättä.

Määrittelyn poistaminen

Jos jokin käsite on määritelty eikä sitä haluta siivota pois koodista, voidaan käyttää #undef-lausetta. Se kuolettaa #define-lauseen.

Ehdollinen kääntäminen

Yhdistämällä #define tai komentorivimäärittelyt lauseisiin #ifdef, #else ja #ifndef voidaan luoda ohjelma, joka kääntää eri koodia sen mukaan, mitä jo on määritelty. Menettelyä voidaan käyttää luomaan yksi lähdekoodi, joka käännetään kahdessa eri ympäristössä, kuten Dos- ja Windows-ympäristössä.

Toinen yleinen käyttötapa on tehdä kääntäminen ehdollisesti sen mukaan, onko DEBUG määritelty, kuten pian näet.

Makrofunktiot

#define-sääntöä voidaan käyttää myös makrofunktioiden luomiseen. Makrofunktio on symboli, joka luodaan #define-sääntöä käyttäen ja joka ottaa argumentteja lähes samoin kuin funktiotkin tekevät. Esikäsittelijä korvaa merkkijonon argumentin mukaan. Jos määrittelemme esimerkiksi makron TWICE seuraavasti

```
#define TWICE(x) ( (x) * 2)
```

ja kirjoitamme koodiin

```
TWICE(4)
```

koko merkkijono TWICE(4) poistetaan ja korvataan arvolla 8! Kun esikäsittelijä näkee arvon 4, se korvataan lausekkeella (84) * 2), josta tulee 4 * 2 eli 8.

Makrossa voi olla useampiakin kuin yksi parametri ja kutakin parametria voidaan käyttää toistuvasti korvaustekstissä. MAX ja MIN ovat kaksi yleistä makroa:

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
```

Huomaa, että makron nimen jälkeen tulee välittömästi sulkumerkki eikä siinä saa olla välilyöntejä. Esikäsittelijä ei anna anteeksi esimerkiksi seuraavaa koodia:

```
#define MAX (x,y) ( (x) > (y) ? (x) : (y) )
```

Jos nyt kirjoitat koodiin seuraavasti

```
int x = 5, y = 7, z;
z = MAX(x,y);
```

olisi välikoodi

```
int x = 5, y = 7, z;
z = (x,y) ( (x) > (y) ? (x) : (y) ) (x,y)
```

Tällöin suoritettaisiin yksinkertainen tekstin korvaus eikä makrofunktiota. Termin MAX paikalle tulee (x,y) ((x) > (y) ? (x) : (y)), jota seuraa sitten (x,y), joka on MAX-termin perässä.

Kun välilyönti poistetaan MAX-termin ja sulkumerkin välistä, on välikoodi oikea:

```
int x = 5, y = 7, z;
z = 7;
```

Miksi kaikki nuo sulkumerkit?

Olet saattanut ihmetellä sulkumerkkien määrää esittelemissämme makrofunktioissa. Esikäsittelijä ei vaadi sulkumerkkejä argumenttien ympärillä korvaavassa merkkijonossa. Kuitenkin sulkumerkeillä selvitään joistakin sivuvaikutuksista, kun makrolle viedään monimutkaisia arvoja. Olettakaamme, että MAX on määritelty seuraavasti:

```
#define MAX(x,y) x > y ? x : y
```

ja makrolle viedään arvot 5 ja 7. Makro toimii kuten pitikin. Mutta, jos makrolle viedään monimutkainen ilmaus, saadaan odottamattomia tuloksia, kuten listaus 21.2 osoittaa.

Listaus 21.2. Sulkumerkit makroissa.

```
1:  // Listaus 21.2 Makrolaajennus
2:  #include <iostream.h>
3:
4:  #define CUBE(a) ( (a) * (a) * (a) )
5:  #define THREE(a) a * a * a
6:
7:  int main()
8:  {
9:    long x = 5;
10:   long y = CUBE(x);
11:   long z = THREE(x);
12:
13:   cout << "y: " << y << endl;
14:   cout << "z: " << z << endl;
15:
16:   long a = 5, b = 7;
17:   y = CUBE(a+b);
18:   z = THREE(a+b);
19:
20:   cout << "y: " << y << endl;
21:   cout << "z: " << z << endl;
22:   return 0;
23: }
```

Tulostus

```
y: 125
z: 125
y: 1728
x: 82
```

Analyysi

Rivillä 4 määritellään makro CUBE, jonka argumentti on suluissa. Rivillä 5 määritellään makro THREE ilman sulkumerkkejä.

Ensimmäisellä makrojen käyttökerralla annetaan parametriksi arvo 5 ja molemmat makrot toimivat hienosti. CUBE(5) laajenee lausekkeeksi ((5) * (5) * (5)), jonka tulos on 125. THREE(5) laajenee lausekkeeksi 5 * 5 * 5, ja tulos on 125.

Toisella käyttökerralla (rivit 16-18) on parametrina lauseke 5 + 7. Tässä tapauksessa CUBE(5 + 7) laajenee lausekkeeksi

((5+7) * (5+7) * (5+7)), josta tulee ((12) * (12) * (12)) ja tulokseksi tulee 1728.

THREE(5+7) laajenee lausekkeeksi 5+7*5+7*5+7. Koska kertolasku suoritetaan ennen yhteenlaskua, saadaan välitulos

5 + (7*5) + (7*5) + 7, ja tulokseksi tulee 82.

Makrot vastaan funktiot ja mallit

Makrojen käyttämiseen liittyy neljä eri ongelmaa. Ensinnäkin ne ovat laajempina sekavia, koska ne on aina määriteltävä yhdellä rivillä. Riviä voi tietenkin hajauttaa kenoviivalla (\), mutta laajoja makroja on vaikea hallita.

Toisena ongelmana on se, että makrot laajennetaan riville joka kerta niitä käytettäessä. Jos makro esiintyy koodissa vaikkapa kymmenen kertaa, on korvaus tehtävä ohjelmassa yhtä monesti. Funktiota sen sijaan kutsuttaisiin vain kerran. Toisaalta makro on usein nopeampi kuin funktion kutsu, koska funktion kutsuun liittyy ylimääräisiä toimintoja.

Makron laajentaminen riville aiheuttaa kolmannen ongelman, koska tällöin makro ei esiinny kääntäjän käyttämässä välitiedostossa eikä siihen voida tällöin tehdä makrokohtaista vianhakua.

Viimeinen ongelma on kuitenkin kaikkein suurin; makrot eivät ole tyyppisuojattuja. Vaikka onkin mukavaa, että makroissa voidaan käyttää mitä tahansa argumentteja, sotii menettely kuitenkin C++ -kielen vahvaa tyypitystä vastaan. Mallien avulla päästään tästä ongelmasta, kuten luvussa 23, "Mallit", kerrotaan.

Merkkijonon muokkaaminen

Esikäsittelijä tarjoaa kaksi erikoisoperaattoria makrojen merkkijonojen muokkaamiseen. Operaattori # korvaa operaattorin jäljessä olevan merkkijonon lainausmerkeissä olevalla merkkijonolla. Liittämisoperaattori sitoo kaksi merkkijonoa toisiinsa.

#-operaattori

#-operaattori sijoittaa lainausmerkit sitä seuraavien merkkien ympärille, aina seuraavaan välilyöntiin (tai muuhun tyhjään väliin) saakka. Jos kirjoitat

```
#define WRITESTRING(x) cout << #x
```

ja käytät sitten kutsua

```
WRITESTRING(This is a string);
```

esikääntäjä muuttaa lauseen seuraavaksi:

```
cout << "This is a string";
```

Merkkijono sijoitetaan siis lainausmerkkeihin, kuten cout vaatii.

Liittäminen (##)

Liittämisoperaattori mahdollistaa yhden tai useamman termin yhdistämisen yhdeksi sanaksi. Uusi sana on todellisuudessa merkintä, jota voidaan käyttää luokan nimenä, muuttujan nimenä, taulukon siirtymänä tai kaikkialla, missä merkkisarja voi esiintyä.

Olettakaamme, että meillä on viisi funktiota: fOnePrint, fTwoPrint, fThreePrint, fFourPrint ja fFivePrint. Voimme tehdä esittelyn

```
#define fPRINT(x) f ## x ## Print
```

ja käyttää sitten lausetta fPRINT(Two) generoimaan fTwoPrint ja fPRINT(Three) generoimaan fThreePrint.

Luvussa 19 kehitettiin PartsList-luokka. Tuo lista saattoi käsitellä vain List-tyyppisiä olioita. Olettakaamme, että haluaisimme tehdä listoja eläimistä, autoista, tietokoneista, jne.

Yhtenä lähestymistapana olisi luoda AnimalList, CarList, jne leikkaamalla ja liittämällä koodia. Siitä tulisi pian painajainen, koska jokainen yhteen listaan tehty muutos tulisi kirjoittaa kaikkiin muihin.

Vaihtoehtona on käyttää makroja ja liittämisoperaattoria. Voisimme tehdä esimerkiksi seuraavan määrittelyn:

```
#define Listof(Type) class Type##List \
{ \
public: \
  Type##List() {} \
private: \
  int itsLength; \
};
```

Esimerkki on hieman hajanainen, mutta ideana on laittaa sisälle kaikki välttämättömät metodit ja tiedot. Kun AnimalList halutaan luoda, kirjoitetaan

```
Listof(Animal)
```

joka muutetaan AnimalList-luokan esittelyksi. Tässä lähestymistavassa on muutamia ongelmia, joita käsitellään luvussa 23, "Mallit".

Esimääritellyt makrot

Monet kääntäjät esimäärittelevät joukon hyödyllisiä makroja, kuten *_DATE_*, *_TIME_*, *_LINE_* ja *_FILE_*. Nimien molemmin puolin on alaviiva, joka yleensä erottaa ne muista ohjelmassa määritellyistä tarkenteista.

Kun esikäsittelijä kohtaa jonkun noista makroista, se tekee vastaavan korvauksen. Esimerkiksi *_DATE_* korvataan päivämäärällä, *_TIME_* ajalla ja *_LINE_* ja *_FILE_* lähdekoodin rivinumerolla ja tiedostonimellä. Huomaa, että korvaamiset tehdään esikääntämisen yhteydessä, ei ajon aikana. Jos pyydät ohjelmaa tulostamaan *_DATE_*, ei nykyinen päivämäärä tulostu, vaan kääntämispäivämäärä. Esimääritellyt makrot ovat hyödyllisiä vianhaussa.

assert()

Monissa kääntäjissä on assert()-makro. Se palauttaa arvon tosi, jos sen parametri saa arvon tosi ja suorittaa jonkin toiminnon, jos parametri saa arvon epätosi. Monet kääntäjät keskeyttävät ohjelman, jos assert() saa arvon epätosi, toiset taas generoivat poikkeustapahtuman (katso lukua 20, "Erikoisluokat ja -funktiot").

Eräs tehokas assert()-makron piirre on se, että esikäsittelijä ei suorita sitä, jos DEBUGia ei ole määritelty. Se on hyvänä tukena kehitystyössä ja lopullisessa versiossa ei ole suoritukseen tai tiedoston kokoon negatiivisesti vaikuttavia tekijöitä.

Kääntäjän tarjoaman assert()-makron sijaan voit kirjoittaa oman assert()-makrosi. Listaus 21.3 sisältää yksinkertaisen assert()-makron ja käyttää sitä.

Listaus 21.3. Yksinkertainen assert()-makro.

```
1:  // Listaus 21.3 ASSERT
2:  #define DEBUG
3:  #include <iostream.h>
4:
5:  #ifndef DEBUG
6:    #define ASSERT(x)
7:  #else
8:    #define ASSERT(x) \
9:  if (! (x)) \
10: { \
11:   cout << "ERROR!! Assert " << #x << " failed\n"; \
12:   cout << " on line " << __LINE__  << "\n"; \
13:   cout << " in file " << __FILE__ << "\n";  \
14: }
15: #endif
16:
17:
18: int main()
19: {
20:   int x = 5;
21:   cout << "First assert: \n";
22:   ASSERT(x==5);
23:   cout << "\nSecond assert: \n";
24:   ASSERT(x != 5);
25:   cout << "\nDone.\n";
26:   return 0;
27: }
```

Tulostus

```
First assert:

Second assert:
ERROR! Assert X!=5 failed
on line 24
in file test2103.cpp
```

Analyysi

Rivillä 2 määritellään termi DEBUG. Se voidaan tehdä joko komentoriviltä tai IDEstä käsin kääntämisen aikana, jotta se voidaan kääntää pois ja päälle tarvittaessa. Riveillä 8-15 määritellään assert()-makro. Se voidaan tehdä otsikkotiedostossa, joka voidaan sitten sisällyttää kooditiedostoon.

Rivillä 5 testataan DEBUG. Jos sitä ei ole määritelty, määritellään assert() olemaan luomatta koodia. Jos DEBUG on määritelty, toteutetaan rivit 8-14.

assert() itse on yksi pitkä lause, joka on jaettu usealle lähdekoodiriville ajateltaessa esikääntäjää. Rivillä 9 testataan viety parametri; jos se on tosi, suoritetaan rivit 11-13, jotka tulostavat virheilmoituksen. Jos parametri on epätosi, ei mitään toimintoja toteuteta.

Vianhaku assert()-makrolla

Ohjelmaa kirjoitettaessa on usein hyvä tietää joitakin toiseikkoja: onko funktiolla tietty arvo, onko osoitin sopiva, jne. Usein virheet ovat sellaisia, että ne näkyvät vain tietyissä tilanteissa. Tiedät esimerkiksi, että osoitin on sopiva ja kuitenkin ohjelma kaatuu. assert() voi auttaa löytämään tuollaisia virheitä, mutta vain, jos käytät sitä säännöllisesti. Aina, kun sijoitat tai viet osoittimen parametrina tai palautat sen funktiosta, varmista että todennat assert()-makrolla osoittimen sopivuuden. Aina, kun koodi riippuu tietystä muuttujan avrosta, käytä assert()-makroa todentamaan se.

assert()-lauseiden runsaastakaan käytöstä ei ole vahinkoa; ne poistetaan koodista, kun määrittelet vianhaun pois. Ne parantavat myös sisäistä dokumentaatiota muistuttaen lukijaa siitä, mikä on uskottavasti oikein kussakin kohtaa ohjelman kulkua.

Sivuvaikutukset

Ei ole lainkaan epätavallista, että virheitä esiintyy sen jälkeen, kun assert()-lauseet on poistettu. Se johtuu melkein aina siitä, että ohjelma on riippuvainen niistä sivuvaikutuksista, jotka assert() ja muu vianhakukoodi aiheuttavat. Jos kirjoitat esimerkiksi

```
ASSERT (x = 5)
```

kun tarkoituksenasi on testata lauseke x == 5, luot erityisen ilkeän virheen.

Oletetaan, että ennen assert()-lausetta x saa arvon 0. Lauseke siis asettaa x:n arvoksi 5 eikä täten tee testausta. Samalla koko lause palauttaa arvon tosi, koska arvo 5 on tosi.

Kun assert()-lause on ohitettu, on x todellakin 5 ja ohjelma suoriutuu hyvin. Kuitenkin siinä on paha virhe. Kun jakeluversio muodostetaan, poistetaan siitä vianhakukoodi, jolloin myös assert()-lause poistuu. Enää lause ei aseta x:n arvoksi lukua 5, vaan siinä on arvo 0. Tällöin ohjelma kaatuu.

Ohjelmoija kääntää nyt vianhaun uudelleen päälle. Voila! Virhe on kadonnut. Ole siis varovainen vianhakukoodin aiheuttamien sivuvaikutusten suhteen. Jos kohtaat virheen, joka ilmenee vain vianhaun ollessa poissa päältä, tutki vianhakukoodiasi tarkoin.

Sopimattomat luokat

Useimpien luokkien kohdalla ilmenee ehtoja, joiden tulee olla aina tosia jäsenfunktion ajamisen jälkeen. Esimerkiksi CIRCLE-olion säde ei saa olla koskaan nolla tai ANIMAL-olion iän täytyy olla aina yli nolla ja alle sadan.

Voi olla hyvin järkevää esitellä Invariants()-metodi, joka palauttaa arvon tosi vain, jos kukin näistä ehdoista on yhä tosi. Voit sitten sisällyttää Assert(Invariants())-lauseen jokaisen luokkametodin alkuun. Invariants()-metodin ei odoteta palauttavan arvoa tosi ennen muodostimen ajamista ja tuhoajafunktion päättymisen jälkeen. Listaus 21.4 esittelee Invariants()-metodin käyttöä yksinkertaisessa luokassa.

Listaus 21.4. Invariants()-metodin käyttö.

```
1:  #define DEBUG
2:  #define SHOW_INVARIANTS
3:  #include <iostream.h>
4:  #include <string.h>
5:
6:
7:  #ifndef DEBUG
8:    #define ASSERT(x)
9:  #else
10:   #define ASSERT(x) \
11: if (! (x)) \
12: { \
13:   cout << "ERROR!! Assert " << #x << " failed\n"; \
14:   cout << " on line " << __LINE__  << "\n"; \
15:   cout << " in file " << __FILE__ << "\n";  \
16: }
17: #endif
18:
19: class String
20: {
21: public:
22: // muodostimet
23:   String();
```

```
24:   String(const char *const);
25:   String(const String &);
26:   ~String();
27:
28:   char & operator[](int offset);
29:   char operator[](int offset) const;
30:
31:   String & operator= (const String &);
32:   int GetLen()const { return itsLen; }
33:   const char * GetString() const { return itsString; }
34:   bool Invariants() const;
35:
36: private:
37:   String (int);         // yksit. muodostin
38:   char * itsString;
39:   unsigned short itsLen;
40: };
41:
42: // oletusmuodostin luo merkkijonon
43: String::String()
44: {
45:   itsString = new char[1];
46:   itsString[0] = '\0';
47:   itsLen=0;
48:   ASSERT(Invariants());
49: }
50:
51:
52: // tukimuodostin auttaa tietyn kokoisen merkkijonon
53: // muodostamisessa.
54: // Täytetään nulleilla.
55: String::String(int len)
56: {
57:   itsString = new char[len+1];
58:   for (int i = 0; i<=len; i++)
59:     itsString[i] = '\0';
60:   itsLen=len;
61:   ASSERT(Invariants());
62: }
63:
64: // Muuntaa merkkitaulukon merkkijonoksi
65: String::String(const char * const cString)
66:
67: {
68:   itsLen = strlen(cString);
69:   itsString = new char[itsLen+1];
70:   for (int i = 0; i<itsLen; i++)
71:   itsString[i] = cString[i];
72:   itsString[itsLen]='\0';
73:   ASSERT(Invariants());
74: }
75:
76: // kopiomuodostin
77: String::String (const String & rhs)
78: {
79:   itsLen=rhs.GetLen();
80:   itsString = new char[itsLen+1];
81:   for (int i = 0; i<itsLen;i++)
82:     itsString[i] = rhs[i];
83:   itsString[itsLen] = '\0';
84:   ASSERT(Invariants());
```

```
85: }
86:
87: // tuhoaja vapauttaa muistin
88: String::~String ()
89: {
90:   ASSERT(Invariants());
91:   delete [] itsString;
92:   itsLen = 0;
93: }
94:
95: // == operaattori vapauttaa muistin ja
96: // kopioi merkkijonon ja sen koon.
97: String& String::operator=(const String & rhs)
98: {
99:   ASSERT(Invariants());
100:  if (this == &rhs)
101:    return *this;
102:  delete [] itsString;
103:  itsLen=rhs.GetLen();
104:  itsString = new char[itsLen+1];
105:  for (int i = 0; i<itsLen;i++)
106:    itsString[i] = rhs[i];
107:  itsString[itsLen] = '\0';
108:  ASSERT(Invariants());
109:  return *this;
110: }
111:
112: //ei-vakio indeksi, palauttaa viittauksen
113: // merkkiin, joten sitä voidaan
114: // muuttaa!
115: char & String::operator[](int offset)
116: {
117:  ASSERT(Invariants());
118:  if (offset > itsLen)
119:    return itsString[itsLen-1];
120:  else
121:    return itsString[offset];
122:  ASSERT(Invariants());
123: }
124:
125: // vakioindeksi vakio-olioiden
126: // käsittelyyn (kts kopiomuodostin!)
127: char String::operator[](int offset) const
128: {
129:  ASSERT(Invariants());
130:  if (offset > itsLen)
131:    return itsString[itsLen-1];
132:  else
133:    return itsString[offset];
134:  ASSERT(Invariants());
135: }
136:
137:
138: bool String::Invariants() const
139: {
140:  #ifdef SHOW_INVARIANTS
141:    cout << " String OK ";
142:  #endif
143:  return ( (itsLen && itsString) || (!itsLen && !itsString) );
144: }
145:
```

```
146: class Animal
147: {
148: public:
149:   Animal():itsAge(1),itsName("John Q. Animal")
150:   {ASSERT(Invariants());}
151:   Animal(int, const String&);
152:   ~Animal(){}
153:   int GetAge() {  ASSERT(Invariants()); return itsAge;}
154:   void SetAge(int Age)
155: {
156:   ASSERT(Invariants());
157:   itsAge = Age;
158:   ASSERT(Invariants());
159: }
160:   String& GetName() { ASSERT(Invariants()); return itsName;  }
161: void SetName(const String& name)
162:   {
163:     ASSERT(Invariants());
164:     itsName = name;
165:     ASSERT(Invariants());
166:   }
167:   bool Invariants();
168: private:
169:   int itsAge;
170:   String itsName;
171: };
172:
173: Animal::Animal(int age, const String& name):
174: itsAge(age),
175: itsName(name)
176: {
177:   ASSERT(Invariants());
178: }
179:
180: bool Animal::Invariants()
181: {
182:   #ifdef SHOW_INVARIANTS
183:     cout << " Animal OK ";
184:   #endif
185:   return (itsAge > 0 && itsName.GetLen());
186: }
187:
188: int main()
189: {
190:   Animal sparky(5,"Sparky");
191:   cout << "\n" << sparky.GetName().GetString() << " is ";
192:   cout << sparky.GetAge() << " years old.";
193:   sparky.SetAge(8);
194:   cout << "\n" << sparky.GetName().GetString() << " is ";
195:   cout << sparky.GetAge() << " years old.";
196:   return 0;
197: }
```

Tulostus

```
String OK String OK String OK String OK
String OK String OK String OK String OK
Animal OK Animal OK
Sparky is Animal OK 5 years old. Animal OK Animal OK Animal OK
Sparky is Animal OK 8 years old. String OK String OK
```

Analyysi

Riveillä 8-17 määritellään assert()-makro. Jos DEBUG on määritelty, makro tulostaa virheilmoituksen aina, kun assert() antaa tuloksen epätosi.

Rivillä 34 esitellään String-luokan jäsenfunktio Invariants(); se määritellään riveillä 138-144. Muodostin esitellään riveillä 43-49 ja rivillä 46, kun olio on täysin luotu, kutsutaan Invariants-metodia vahvistamaan rakenne.

Tätä menettelyä toistetaan muille muodostimille ja tuhoajafunktio kutsuu Invariants()-metodia vain ennen kuin se asetetaan tuhoamaan olio. Muut luokkafunktiot kutsuvat Invariants()-metodia sekä ennen toimintaa ja sitten uudelleen ennen päättymistä. Menettely kertoo C++ -kielen perusperiaatteen: jäsenfunktioiden, jotka eivät ole muodostimia tai tuhoajafunktioita, tulisi työskennellä oikeiden olioiden kanssa ja jättää ne oikeaan tilaan. Rivillä 152 esittelee Animal-luokka oman Invariants()-metodinsa, joka toteutetaan riveillä 178-186. Huomaa riveiltä 149, 152, 154, 159 ja 161, että inline-funktiot voivat kutsua Invariants()-metodia.

Nykyisten arvojen tulostaminen

Sen lisäksi, että todennetaan assert()-makrolla tiettyjen asioiden toteutuminen, saatetaan joskus haluta tulostaa osoittimien, muuttujien ja merkkijonojen nykyiset arvot. Sellaisesta voi olla paljon hyötyä tarkistettaessa oletuksia ohjelman kulusta ja paikannettaessa silmukoiden ylimääräisiä kierroksia. Listaus 21.5 havainnollistaa tätä ajattelua.

Listaus 21.5. Arvojen tulostaminen DEBUG-moodissa.

```
1:  // Listaus 21.5 - Tulostetaan arvot DEBUG-moodissa
2:  #include <iostream.h>
3:  #define DEBUG
4:
5:  #ifndef DEBUG
6:    #define PRINT(x)
7:  #else
8:    #define PRINT(x) \
9:    cout << #x << ":\t" << x << endl;
10: #endif
11:
12: enum BOOL { FALSE, TRUE } ;
13:
14: int main()
15: {
16:   int x = 5;
17:   long y = 73898l;
18:   PRINT(x);
19:   for (int i = 0; i < x; i++)
20:   {
21:     PRINT(i);
22:   }
23:
24:   PRINT (y);
25:   PRINT("Hi.");
26:   int *px = &x;
27:   PRINT(px);
```

```
28:   PRINT (*px);
29:   return 0;
30: }
```

Tulostus

```
x:  5
i:  0
i:  1
i:  2
i:  3
i:  4
y:  73898
"Hi.":  Hi.
px: 0x2100 (Voit saada jonkun muun arvon.)
*px:    5
```

Analyysi

Riveillä 5-10 oleva makro tulostaa annetun parametrin nykyisen arvon. Huomaa, että ensimmäisenä saa cout parametrin lainausmerkeissä; eli, kun viedään x, cout saa arvon "x".

Seuraava cout vastaanottaa lainausmerkeissä olevan merkkijonon ":\t", joka tulostaa puolipisteen ja sarkaimen. Kolmanneksi cout vastaanottaa parametrin (x) arvon ja sitten lopuksi endl-merkkijonon, joka kirjoittaa rivinvaihdon ja tyhjentää puskurin.

Vianhakutasot

Laajoissa, monimutkaisissa projekteissa saattaa olla tarvetta paremmalle kontrollille kuin DEBUGin kääntäminen päälle ja pois antaa. Tällöin voidaan määritellä vianhakutasoja ja testata niitä päätettäessä, mitä makroja käytetään ja mitä jätetään pois.

Taso määritellään sijoittamalla lauseen #define DEBUG jälkeen numero. Tasoja voi olla useita, mutta yleensä neljä kappaletta; HIGH, MEDIUM, LOW ja NONE. Listaus 21.6 havainnollistaa tämän toteuttamista. Siinä käytetään String- ja Animal-luokkia listauksesta 21.4. Luokkien metodit lukuunottamatta Invariants()-metodia on jätetty pois tilan säästämiseksi, koska ne ovat samoja kuin listauksessa 21.4.

Huom! Kääntääksesi koodin kopioi rivit 40-132 listauksesta 21.4 tämän listauksen rivien 60 ja 61 väliin.

Listaus 21.6. Vianhaun tasot.

```
1:  enum LEVEL { NONE, LOW, MEDIUM, HIGH };
2:  const int FALSE = 0;
3:  const int TRUE = 1;
4:  typedef int BOOL;
```

```
5:
6:   #define DEBUGLEVEL HIGH
7:
8:   #include <iostream.h>
9:   #include <string.h>
10:
11:  #if DEBUGLEVEL < LOW  // keskisuuri tai korkea
12:    #define ASSERT(x)
13:  #else
14:
15:    #define ASSERT(x) \
16:    if (! (x)) \
17:    { \
18:      cout << "ERROR!! Assert " << #x << " failed\n"; \
19:      cout << " on line " << __LINE__  << "\n"; \
20:      cout << " in file " << __FILE__ << "\n";  \
21:    }
22:  #endif
23:
24:  #if DEBUGLEVEL < MEDIUM
25:    #define EVAL(x)
26:  #else
27:    #define EVAL(x) \
28:  cout << #x << ":\t" << x << endl;
29:  #endif
30:
31:  #if DEBUGLEVEL < HIGH
32:    #define PRINT(x)
33:  #else
34:    #define PRINT(x) \
35:  cout << x << endl;
36:  #endif
37:
38:
39:  class String
40:  {
41:  public:
42:    // muodostimet
43:    String();
44:    String(const char *const);
45:    String(const String &);
46:    ~String();
47:
48:    char & operator[](int offset);
49:    char operator[](int offset) const;
50:
51:    String & operator= (const String &);
52:    int GetLen()const { return itsLen; }
53:    const char * GetString() const
54:    { return itsString; }
55:    BOOL Invariants() const;
56:
57:  private:
58:    String (int);         // yks. muodostin
59:    char * itsString;
60:    unsigned short itsLen;
61:  };
62:
63:  BOOL String::Invariants() const
64:  {
65:    PRINT("(String Invariants Checked)");
```

```
66:    return ( (BOOL) (itsLen && itsString) ||
67:    (!itsLen && !itsString) );
68: }
69:
70: class Animal
71: {
72: public:
73:    Animal():itsAge(1),itsName("John Q. Animal")
74:    {ASSERT(Invariants());}
75:
76:    Animal(int, const String&);
77:    ~Animal(){}
78:
79: int GetAge()
80: {
81:    ASSERT(Invariants());
82:    return itsAge;
83:
84: }
85:
86: void SetAge(int Age)
87: {
88:    ASSERT(Invariants());
89:    itsAge = Age;
90:    ASSERT(Invariants());
91: }
92: String& GetName()
93: {
94:    ASSERT(Invariants());
95:    return itsName;
96: }
97:
98: void SetName(const String& name)
99: {
100:
101:   ASSERT(Invariants());
102:   itsName = name;
103:   ASSERT(Invariants());
104: }
105:
106:   BOOL Invariants();
107: private:
108:
109:   int itsAge;
110:   String itsName;
111: };
112:
113: BOOL Animal::Invariants()
114: {
115:
116:   PRINT("(Animal Invariants Checked)");
117:   return (itsAge > 0 && itsName.GetLen());
118:
119: }
120:
121: int main()
122: {
123:   const int AGE = 5;
124:   EVAL(AGE);
125:   Animal sparky(AGE,"Sparky");
126:   cout << "\n" << sparky.GetName().GetString();
```

```
127:   cout << " is ";
128:   cout << sparky.GetAge() << " years old.";
129:   sparky.SetAge(8);
130:   cout << "\n" << sparky.GetName().GetString();
131:   cout << " is ";
132:   cout << sparky.GetAge() << " years old.";
133:   return 0;
134: }
```

Tulostus

```
AGE:    5
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)
(String Invariants Checked)

Sparky is (Animal Invariants Checked)
5 years old. (Animal Invariants Checked)
(Animal Invariants Checked)
(Animal Invariants Checked)

Sparky is (Animal Invariants Checked)
8 years old. (String Invariants Checked)
(String Invariants Checked)

// run again with DEBUG = MEDIUM

AGE:    5
Sparky is 5 years old.
Sparky is 8 years old.
```

Analyysi

Riveillä 11-21 määritellään assert()-makro jätettäväksi pois, jos DEBUGLEVEL on pienempi kuin LOW (tällöin DEBUGLEVEL on siis NONE). Jos vianhaku sallitaan, assert() toimii. Rivillä 25 esitellään EVAL jätettäväksi pois, jos DEBUG on pienempi kuin MEDIUM.

Riveillä 31-36 esitellään PRINT-makro jätettäväksi pois, jos DEBUGLEVEL on pienempi kuin HIGH. Makroa ei siis suoriteta, jos DEBUGLEVEL on MEDIUM. Tällöin suoritetaan kuitenkin EVAL ja assert().

PRINT-makroa käytetään Invariants()-metodin kanssa tulostamaan ilmoitus. EVALia käytetään rivillä 124 tutkimaan vakion kokonaisluvun AGE nykyinen arvo.

Tee/Älä tee Käytä isoja kirjaimia makrojen nimissä. Kyseessä on yleinen käytäntö ja muut ohjelmoijat voivat hämääntyä, jos et tee niin.

Älä salli makrojen aiheuttaa sivuvaikutuksia. Älä kasvata muuttujia tai sijoita niihin arvoja makrojen sisällä.

Laita kaikki makrofunktioiden sisällä olevat argumentit sulkuihin.

Yhteenveto

Tässä luvussa käsiteltiin esikäsittelijän toimintaa hieman tarkemmin. Aina, kun kääntäjä ajetaan, ajetaan esikäsittelijä ensin. Se kääntää esikäsittelijäkomennot kuten #define ja #ifdef.

Esikäsittelijä tekee tekstikorvaukset, vaikkakin makroja käytettäessä ne voivat olla jokseenkin monimutkaisia. Käyttämällä #ifdef-, #else- ja #ifndef-lauseita voidaan toteuttaa ehdollinen kääntäminen. Tällöin käännetään jokin lausejoukko tiettyjen ehtojen vallitessa ja jokin toinen lausejoukko toisten ehtojen vallitessa. Tällainen menettely voi tukea ohjelmointia useaan eri ympäristöön ja sitä voidaan hyödyntää sisällyttämään koodiin vianhakuinformaatiota ehdollisesti.

Makrofunktioilla voidaan korvata tekstejä sen mukaan, mitä argumentteja makrolle viedään kääntämisen aikana. On tärkeää sijoittaa sulkumerkit makron argumenttien ympärille oikean korvaamisen takaamiseksi.

Makrofunktiot ja esikäsittelijä yleensäkin eivät ole niin tärkeitä C++ -kielessä kuin ne olivat C-kielessä. C++ sisältää runsaasti ominaisuuksia kuten const-muuttujat ja mallit, jotka ovat parempia vaihtoehtoja kuin esikäsittelijän käyttö.

Kysymyksiä ja Vastauksia

K
Jos C++ tarjoaa parempia vaihtoehtoja esikäsittelijälle, miksi esikäsittelijä on edelleenkin käytettävissä?
V
Ensiksin C++ on taaksepäin yhteensopiva C-kielen kanssa ja C++-kielen tulee tukea kaikkia merkittäviä C-kielen osia. Toiseksi C++ -kielessä käytetään edelleenkin esikäsittelijää, erityisesti tiedostojen sisällyttämiseen (#include).

K
Miksi makrofunktioita tulisi lainkaan käyttää, kun tavalliset funktiotkin ovat käytettävissä?
V
Makrofunktiot toteutetaan riveillään ja niitä käytetään korvaamaan samojen komentojen toistuva kirjoittaminen lyhyemmillä muodoilla. Sanottakoon vielä kerran, että mallit tarjoavat paremman vaihtoehdon.

K
Onko olemassa vaihtoehtoa esikäsittelijän käytölle sisäisten arvojen tulostamisessa vianhaun aikana?
V
Paras vaihtoehto on watch-lauseiden käyttäminen debuggerissa. Tutki kääntäjäsi manuaalia saadaksesi tarvittaessa lisätietoa watch-ominaisuuden käytöstä.