Osa
VII

24. oppitunti

Poikkeukset ja virheiden käsittely

Tässä kirjassa esitetty koodi on tarkoitettu havainnollistamaan asioita. Listauksissa ei ole käsitelty virheitä, jotta lukijan huomio ei kääntyisi pois opiskeltavista asioista. Todellisissa käytännön ohjelmissa on virheet kuitenkin otettava huomioon. Itse asiassa juuri virheiden käsittely onkin usein suurin osa koodia!

Tämän viimeisen luvun aiheita ovat:

- ❏ Mitä poikkeukset ovat
- ❏ Kuinka poikkeuksia käytetään ja millaisia seikkoja niihin liittyy
- ❏ Kuinka luodaan virheetöntä koodia
- ❏ Kuinka tästä eteenpäin

Erilaiset viat ja virheet

On tosiasia, että liian monissa suurimpien ohjelmistotalojen kaupallisissa ohjelmissa on virheitä. Vakavia virheitä.

Pelkkä tämän toteaminen ei riitä. Toimivien, virheettömien ohjelmien luominen tulisi olla ensisijaisena ohjelmoinnin tavoitteena. Mielestäni suurin yksittäinen ohjelmistoteollisuuden ongelma on epävakaa koodi. Ohjelmistoprojektien suurimmat kustannuserät koituvat testauksesta, vikojen hausta ja niiden korjaamisesta.

Ongelmia aiheuttavia virheitä on useaa eri tyyppiä. Ensimmäisenä virhetyyppinä on huono logiikka: ohjelma toimii niin kuin pitääkin, mutta algoritmeja ei ole mietitty tarpeeksi tarkkaan. Toinen virhetyyppi on syntaksivirheet: käytät väärää lauserakennetta, funktiota tai rakennetta. Edellä olevat virhetyypit ovat yleisimpiä ja ohjelmoijat yleensä hakevatkin ne esille. On paljon vaikeampi löytää virheitä, jotka ponnahtavat esille käyttäjän tehdessä jotakin odottamatonta. Nuo pienet logiikkapommit vaanivat kätköissään kuin maamiinat. Mielestäsi kaikki on kunnossa ja sitten yhtä äkkiä - POM! - joku astuu väärään paikkaan ja ohjelma possahtaa.

Tutkimus ja kokemus osoittavat, että mitä myöhemmin löydät ongelman, sitä enemmän sen korjaaminen maksaa. Halvimmat ongelmat ovat ne, jotka jäävät luomatta. Seuraavaksi halvimpia ovat kääntämisaikaiset virheet. C++ -standardit pakottavat kääntäjän käyttämään runsaasti energiaa, jotta mahdollisimman paljon virheitä tulisi esille kääntämisen yhteydessä.

Virheet, jotka säilyvät kääntämisen yli, mutta jotka havaitaan ensimmäisessä testauksessa - virheet, joihin ohjelma aina kaatuu - ovat halvempia löytää ja korjata kuin virheet, jotka tulevat esille silloin tällöin, satunnaisesti.

Syntaksi- ja logiikkavirheitä suurempi ongelma ovat erikoistilanteet (poikkeustilanteet): ohjelmasi toimii hyvin, kun käyttäjä syöttää lukuja. Mutta kun käyttäjä syöttää kirjaimen, ohjelma kaatuu. Toiset ohjelmat kaatuvat muistin riittämättömyyteen ja jotkut ehkä siksi, että levykettä ei ole laitettu asemaan tai modeemiyhteys katkeaa.

Tällaisten ongelmatilanteiden takia pyrkivät ohjelmoijat luomaan iskunkestäviä ohjelmia. Tällaiset ohjelmat pystyvät käsittelemään kaikki mahdolliset ajonaikaiset tapahtumat aina käyttäjän virheellisestä syötöstä muistin riittämättömyyteen.

On tärkeää erottaa toisistaan syntaksivirheet ja logiikkavirheet sekä poikkeukset. Syntaksivirheet ovat usein kirjoitusvirheitä ja logiikkavirheet taas johtuvat siitä, että ohjelmoija ei ymmärrä itse ongelmaa tai ei osaa kehittää ratkaisua ongelmalle. Poikkeukset syntyvät epätavallisista mutta ennustettavista ongelmista kuten resurssien (muisti tai levytila) loppuminen.

Odottamattoman tilanteen käsittely

Ohjelmoijat käyttävät tehokkaita kääntäjiä ja ympäröivät koodinsa assert-lauseisiin siepatakseen ohjelmavirheet. He käyttävät suunnittelukatselmuksia ja lisätestaamista logiikkavirheiden löytämiseksi.

Poikkeukset ovat kuitenkin erilaisia. Et voi poistaa poikkeavia olosuhteita; voit vain valmistautua niihin. Käyttäjäsi kuluttavat aika-ajoin kaiken muistin ja kysymys kuuluukin: mitä tulisi asialle tehdä. Vaihtoehdot ovat seuraavassa:

- ❏ Anna ohjelman kaatua
- ❏ Informoi käyttäjää ja lopeta ohjelma
- ❏ Informoi käyttäjää ja anna käyttäjän yrittää palauttaa tilanne ennalleen ja jatkaa
- ❏ Tee korjaavat toimenpiteet ja jatka häiritsemättä käyttäjää

Vaikkakaan jokaisen ohjelman ei ole välttämätöntä tai toivottavaa toipua automaattisesti ja hiljaa kaikista poikkeustilanteista, on selvää, että ohjelman kaataminen on viimeinen vaihtoehto.

C++ -kielen poikkeuskäsittely tarjoaa tyyppisuojatun, integroidun menettelyn toipua ennustettavasta mutta epätavallisesta tilanteesta, joita ohjelman ajon aikana syntyy.

Poikkeukset

C++ -kielessä on poikkeus olio, joka viedään koodialueelta, jossa virhe esiintyy, koodialueelle, joka käsittelee virheen ja olion antamaa tulostetta voidaan käyttää palautteen antajana käyttäjälle.

Poikkeusten perusidea on melko suoraviivainen:

Resurssien todellinen varaaminen (esimerkiksi muistin varaaminen tai tiedoston lukitseminen) tehdään tavallisesti hyvin alhaisella tasolla ohjelmassa.

Logiikka sen suhteen, mitä tehdään, kun operaatio (muistia ei voida varata tai tiedostoa ei voida lukita) epäonnistuu, on tavallisesti korkealla ohjelmassa yhdessä sen koodin kanssa, joka on vuorovaikutuksessa käyttäjän kanssa.

Poikkeukset tarjoavat erikoispolun resursseja varaavasta koodista koodiin, joka käsittelee virhetilanteen. Jos joitakin funktioita on kesken, niille annetaan mahdollisuus vapauttaa muistivarauksensa, mutta niiden ei tarvitse sisältää koodia, jonka tarkoituksena olisi virhetilannetiedon siirtäminen.

Kuinka poikkeuksia käytetään

try-lohkot sijoitetaan mahdollisia ongelmia aiheuttavan koodin ympärille. Esimerkiksi:

```
try {
JokuVaarallinenFunktio();
}
```

catch-lohkot käsittelevät try-lohkosta heitetyt poikkeukset. Esimerkiksi:

```
try {
  JokuVaarallinenFunktio();
}
catch(OutOfMemory)
{
  // toimintoja
}
```

Poikkeusten käsittelyn perusvaiheet:
1. Identifioi ne ohjelma-alueet, joista voi syntyä poikkeustilanteita ja laita ne try-lohkoihin.

2. Luo catch-lohkot sieppaamaan heitetyt poikkeukset ja puhdista varattu muisti ja informoi käyttäjää. Listaus 24.1 havainnollistaa try- ja catch-lohkojen käyttämistä.

Poikkeukset ovat olioita, jotka lähettävät tietoa ongelmasta.

try-lohko on lohko, aaltosulkujen välissä, josta poikkeus voidaan heittää.

catch-lohko on lohko, joka sijoittuu välittömästi try-lohkon perään ja joka käsittelee poikkeukset.

Kun poikkeus syntyy, suoritus siirtyy catch-lohkolle.

Listaus 24.1. Poikkeuksen syntyminen.

```
1:  #include <iostream.h>
2:  const int DefaultSize = 10;
3:
4:  // Poikkeusluokka
5:  class xBoundary
6:  {
7:  public:
8:    xBoundary() {}
9:    ~xBoundary() {}
10: private:
11: };
12:
13:
```

```
14: class Array
15: {
16: public:
17:   // muodostimet
18:   Array(int itsSize = DefaultSize);
19:   Array(const Array &rhs);
20:   ~Array() { delete [] pType;}
21:
22:   // operaattorit
23:   Array& operator=(const Array&);
24:   int& operator[](int offSet);
25:   const int& operator[](int offSet) const;
26:
27:   // käsittelijät
28:   int GetitsSize() const { return itsSize; }
29:
30:   // ystävät
31:   friend ostream& operator << (ostream&, const Array&);
32:
33: private:
34:   int *pType;
35:   int  itsSize;
36: };
37:
38:
39: Array::Array(int size):
40: itsSize(size)
41: {
42:   pType = new int[size];
43:   for (int i = 0; i<size; i++)
44:     pType[i] = 0;
45: }
46:
47:
48: Array& Array::operator=(const Array &rhs)
49: {
50:   if (this == &rhs)
51:     return *this;
52:   delete [] pType;
53:   itsSize = rhs.GetitsSize();
54:   pType = new int[itsSize];
55:   for (int i = 0; i<itsSize; i++)
56:     pType[i] = rhs[i];
57:   return *this;
58: }
59:
60: Array::Array(const Array &rhs)
61: {
62:   itsSize = rhs.GetitsSize();
63:   pType = new int[itsSize];
64:   for (int i = 0; i<itsSize; i++)
65:     pType[i] = rhs[i];
66: }
67:
68:
69: int& Array::operator[](int offSet)
70: {
71:   int size = GetitsSize();
72:   if (offSet >= 0 && offSet < GetitsSize())
73:     return pType[offSet];
74:   throw xBoundary();
```

```
75:   return pType[offSet]; // to appease MSC!
76: }
77:
78:
79: const int& Array::operator[](int offSet) const
80: {
81:   int mysize = GetitsSize();
82:   if (offSet >= 0 && offSet < GetitsSize())
83:     return pType[offSet];
84:   throw xBoundary();
85:   return pType[offSet]; // to appease MSC!
86: }
87:
88: ostream& operator<< (ostream& output, const Array& theArray)
89: {
90:   for (int i = 0; i<theArray.GetitsSize(); i++)
91:     output << "[" << i << "] " << theArray[i] << endl;
92:   return output;
93: }
94:
95: int main()
96: {
97:   Array intArray(20);
98:   try
99:   {
100:    for (int j = 0; j< 100; j++)
101:    {
102:      intArray[j] = j;
103:      cout << "intArray[" << j << "] okay..." << endl;
104:    }
105:  }
106:    catch (xBoundary)
107:    {
108:      cout << "Unable to process your input!\n";
109:    }
110:  cout << "Done.\n";
111:  return 0;
112: }
```

Tulostus

```
intArray[0] okay…
intArray[1] okay…
intArray[2] okay…
intArray[3] okay…
intArray[4] okay…
intArray[5] okay…
intArray[6] okay…
intArray[7] okay…
intArray[8] okay…
intArray[9] okay…
intArray[10] okay…
intArray[11] okay…
intArray[12] okay…
intArray[13] okay…
intArray[14] okay…
intArray[15] okay…
intArray[16] okay…
```

```
intArray[17] okay…
intArray[18] okay…
intArray[19] okay…
Unable to process your input!
Done.
```

Analyysi

Listaus sisältää hieman suppean Array-luokan, jonka avulla havainnollistetaan poikkeustilanteiden käsittelyä. Riveillä 5-11 esitellään hyvin yksinkertainen poikkeusluokka, xBoundary. Tärkein luokkaan liittyvä asia, joka kannattaa huomata, on se, että siinä ei ole yhtään mitään sellaista, joka tekisi siitä muita poikkeavamman. Itse asiassa mikä tahansa luokka, minkä tahansa nimisenä, toimisi aivan hyvin poikkeusluokkana. Ainoa ero on siinä, että vain tämä luokka heitetään (rivi 74) ja se siepataan (rivi 107).

Siirtymäoperaattorit heittävät xBoundaryn, kun luokan asiakas yrittää käsitellä tietoa taulukon rajojen ulkopuolella. Tämä menettely on paljon parempi kuin se tapa, jolla tavalliset taulukot käsittelevät sellaista pyyntöä; ne palauttavat sen, mitä sattuu olemaan muistissa tuossa paikassa, idioottivarma tapa kaataa ohjelma.

Rivillä 99 aloittaa try lohkon, joka päättyy rivillä 106. Tuossa try-lohkossa lisätään 100 kokonaislukua taulukkoon, joka esiteltiin rivillä 98.

Rivillä 107 esitellään catch-lohko, joka sieppaa xBoundary-poikkeukset.

try-lohkot

try-lohko on joukko lauseita, jotka alkavat sanalla try. Lohkoa seuraa aloittava aaltosulku, koodi ja lohko päättyy lopettavaan aaltosulkuun.

Esimerkiksi:

```
try
{
  Funktio();
};
```

catch-lohkot

catch-lohko on joukko lauseita, jotka alkavat catch-sanalla. Sen jälkeen tulee poikkeustyyppi sulkumerkeissä ja aloittava aaltosulku. Lohko päättyy lopettavaan aaltosulkuun.

Esimerkiksi

```
try
{
  Funktio();
};
```

```
catch (MuistiLoppu)
{
  // toimintoja
}
```

try- ja catch-lohkojen käyttäminen

Sen selvittäminen, mihin try-lohkot laitetaan, lienee vaikeinta poikkeuksien käyttämisessä. Aina ei ole selvää, mitkä toiminnot aiheuttavat poikkeuksen. Seuraava kysymys on se, missä kohtaa siepata poikkeus. Haluat varmaankin heittää kaikki muistiin liittyvät poikkeukset sieltä, missä muistia varataan, mutta nuo poikkeukset haluaisit yleensä siepata korkeammalta tasolta, läheltä käyttöliittymää.

Yritettäessä selvittää try-lohkojen paikkaa kannattaa katsoa ne kohdat, joissa muistia varataan tai käytetään muita resursseja. Muita seikkoja ovat rajojen ylitykset, väärät syöttötiedot, jne.

Poikkeustilanteiden sieppaaminen

Poikkeustilanteiden sieppaaminen tapahtuu seuraavasti: kun poikkeus heitetään, tutkitaan kutsupino. Kutsupino on lista funktiokutsuja, jotka luodaan, kun yksi ohjelman osa kutsuu toista funktiota.

Kutsupino jäljittää suorituspolun. Jos main() kutsuu funktiota Animal::GetFavoriteFood(), joka taas kutsuu funktiota Animal::LookupPreferences(), joka puolestaan kutsuu funktiota fstream::operator>>(), kaikki nuo kutsut joutuvat kutsupinoon. Rekursiivinen funktio voi joutua kutsupinoon moneen kertaan.

Poikkeus viedään kutsupinoon. Pinoa purettaessa kutsutaan kunkin pinossa olevan paikallisen olion tuhoajafunktiota ja oliot tuhotaan.

Jokaisen try-lohkon perässä on yksi tai useampia catch-lauseita. Jos poikkeus vastaa jotakin catch-lausetta, sen odotetaan tulevan käsitellyksi tuo lause suorittamalla. Jos vastinetta ei löydy, pinon purkaminen jatkuu.

Jos poikkeus saavuttaa ohjelman alun (main()) eikä sitä ole vielä käsitelty, kutsutaan sisään rakennettua käsittelijää lopettamaan ohjelma.

On tärkeää huomata, että poikkeuksen aiheuttama pinon purkaminen kulkee yhteen suuntaan. Purkamisen edistyessä oliot tuhotaan. Takaisin paluuta ei ole: kun poikkeus on käsitelty, ohjelma jatkaa eteenpäin siitä try-lohkosta, jonka catch-lause käsitteli poikkeuksen.

Niinpä listauksen 24.1 suoritus jatkuu riviltä 99, joka on sen try-lohkon jälkeen, jonka catch-lause käsitteli xBoundary-poikkeuksen. Muista, että poikkeuksen syntyessä ohjelman kulku jatkuu catch-lohkon perästä, ei siitä kohtaa, mistä poikkeus heitettiin.

Enemmän kuin yksi catch-määrittely

On mahdollista, että useampi kuin yksi olosuhde aiheuttaa poikkeuksen. Siinä tapauksessa catch-lauseet sijoitetaan peräkkäin aivan kuin switch-lauseessa. Vastine default-osalle olisi nyt "sieppaa kaikki" -lause, joka osoitetaan merkinnällä catch(...).

Sieppaaminen viittauksena ja monimuotoisuus

Voit hyödyntää sitä tosiasiaa, että poikkeuksetkin ovat luokkia ja käyttää niitä monimuotoisesti. Viemällä poikkeuksen viittauksena voit käyttää periytymishierarkiaa tekemään sopivan toiminnon poikkeuksen tyypin mukaan. Listaus 24.2 havainnollistaa poikkeusten käyttöä monimuotoisesti.

Listaus 24.2. Monimuotoiset poikkeukset.

```
1:  #include <iostream.h>
2:  const int DefaultSize = 10;
3:
4:  // poikkeusluokat
5:  class xBoundary {};
6:
7:  class xSize
8:  {
9:  public:
10:   xSize(int size):itsSize(size) {}
11:   ~xSize(){}
12:   virtual int GetSize() { return itsSize; }
13:   virtual void PrintError()
14:   { cout << "Size error. Received: " << itsSize << endl; }
15: protected:
16:   int itsSize;
17: };
18:
19: class xTooBig : public xSize
20: {
21: public:
22:   xTooBig(int size):xSize(size){}
23:   virtual void PrintError()
24:   { cout << "Too big! Received: ";
25:   cout << xSize::itsSize << endl; }
26: };
27:
28: class xTooSmall : public xSize
29: {
30: public:
31:   xTooSmall(int size):xSize(size){}
32:   virtual void PrintError()
33:   { cout << "Too small! Received: ";
34:   cout << xSize::itsSize << endl; }
```

```
35: };
36:
37: class xZero  : public xTooSmall
38: {
39: public:
40:   xZero(int size):xTooSmall(size){}
41:   virtual void PrintError()
42:   { cout << "Zero!!. Received: ";
43:   cout << xSize::itsSize << endl; }
44: };
45:
46: class xNegative : public xSize
47: {
48: public:
49:   xNegative(int size):xSize(size){}
50:   virtual void PrintError()
51:
52:   { cout << "Negative! Received: ";
53:   cout << xSize::itsSize << endl; }
54: };
55:
56:
57: class Array
58: {
59: public:
60:   // muodostimet
61:   Array(int itsSize = DefaultSize);
62:   Array(const Array &rhs);
63:   ~Array() { delete [] pType;}
64:
65:   // operaattorit
66:   Array& operator=(const Array&);
67:   int& operator[](int offSet);
68:   const int& operator[](int offSet) const;
69:
70:   // käsittelijät
71:   int GetitsSize() const { return itsSize; }
72:
73:   // ystävät
74:   friend ostream& operator<< (ostream&, const Array&);
75:
76:
77: private:
78:   int *pType;
79:   int  itsSize;
80: };
81:
82: Array::Array(int size):
83: itsSize(size)
84: {
85:   if (size == 0)
86:     throw xZero(size);
87:
88:   if (size < 0)
89:     throw xNegative(size);
90:
91:   if (size < 10)
92:     throw xTooSmall(size);
93:
94:   if (size > 30000)
95:     throw xTooBig(size);
```

```
96:
97:
98:    pType = new int[size];
99:    for (int i = 0; i<size; i++)
100:     pType[i] = 0;
101: }
102:
103: int& Array::operator[] (int offset)
104: {
105:   int size = GetitsSize();
106:   if (offset >= 0 && offset < GetitsSize())
107:     return pType[offset];
108:   throw xBoundary();
109:   return pType[offset];
110: }
111:
112: const int& Array::operator[] (int offset) const
113: {
114:   int size = GetitsSize();
115:   if (offset >= 0 && offset < GetitsSize())
116:     return pType[offset];
117:   throw xBoundary();
118:   return pType[offset];
119: }
120:
121: int main()
122: {
123:
124:   try
125:   {
126:     int choice;
127:     cout << "Enter the array size: ";
128:     cin >> choice;
129:     Array intArray(choice);
130:     for (int j = 0; j< 100; j++)
131:     {
132:       intArray[j] = j;
133:       cout << "intArray[" << j << "] okay..." << endl;
134:     }
135:   }
136:   catch (xBoundary)
137:   {
138:     cout << "Unable to process your input!\n";
139:   }
140:   catch (xSize& theException)
141:   {
142:     theException.PrintError();
143:   }
144:   catch (...)
145:   {
146:     cout << "Something went wrong, but I've no idea what!" << endl;
147:   }
148:   cout << "Done.\n";
149:   return 0;
150: }
```

Tulostus

```
Enter the array size: 5
Too small! Received: 5
Done.
```

```
Enter the array size: 50000
Too big! Received: 50000
Done.

Enter the array size: 12
intArray[0] okay…
intArray[1] okay…
intArray[2] okay…
intArray[3] okay…
intArray[4] okay…
intArray[5] okay…
intArray[6] okay…
intArray[7] okay…
intArray[8] okay…
intArray[9] okay…
intArray[10] okay…
intArray[11] okay…
Unable to process your input!
Done.
Press any key to continue
```

Analyysi

Listaus 24.2 esittelee virtuaalin metodin xSize-luokassa. Metodi, PrintError() tulostaa virheilmoituksen ja luokan todellisen koon. Se korvataan jokaisessa johdetussa luokassa.

Rivillä 140 esitellään poikkeusolio viittauksena. Kun PrintError()-metodia kutsutaan viittauksena olioon, monimuotoisuus aikaansaa sen, että oikeaa PrintError()-versiota kutsutaan. Ensimmäisellä kerralla taulukon kooksi annetaan 5. Tällöin heitetään TooSmall-poikkeus, ja tuo xSize-poikkeus siepataan rivillä 140. Toisella kerralla annetaan taulukon kooksi 50000, jolloin heitetään TooBig-poikkeus. Myös se siepataan rivillä 140, mutta monimuotoisuuden ansiosta tulostetaan oikea virheilmoitus. Kun taulukon kooksi annetaan lopuksi 12, taulukko täytetään, kunnes xBoundary-poikkeus heitetään ja se siepataan rivillä 136.

Seuraavat vaiheet

Mallien ja poikkeusten avulla olet paremmin varustautunut hyödyntämään joitakin C++ -kielen kehittyneitä piirteitä. Ennen kuin laitat kirjan pois, tutkikaamme vielä kuitenkin joitakin seikkoja, jotka koskevat ammattitason koodia. Kun olet päässyt harrastelijavaiheesta ja työskentelyt osana kehitystiimiä, on sinun kirjoitettava koodia, joka ei pelkästään toimi vaan on myös luettavaa. Koodiasi tulee myös voida ylläpitää ja tukea, sekä itsesi toimesta, koska asiakkaan vaatimukset voivat muuttua, että muiden toimesta, kun lähdet projektista.

Tyyli

On tärkeää omaksua johdonmukainen koodaustyyli, vaikkakaan ei ole juuri väliä, millaisen tyylin omaksut. Johdonmukaisen tyylin ansiosta on helpompi arvata, mitä tietty koodi tarkoittaa ja samalla vältytään sen tutkimiselta, aloititko funktion nimen isolla kirjaimella vai etkö, kun viimeksi kutsuit sitä.

Seuraavat ohjeet ovat suuntaa antavia; ne perustuvat niihin menettelyihin, joita olen käyttänyt viime projekteissani ja ne ovat toimineet aivan hyvin. Voit laatia yhtä helposti oman tyylisi, mutta tästä pääset ainakin liikkeelle.

Kuten Emerson sanoi, "Tyhmä johdonmukaisuus ei vie eteenpäin", mutta jonkinlainen johdonmukaisuus on aina hyvä olla olemassa. Kehitä omasi, mutta kohtele sitä myöhemmin kuin kuninkaan ohjeena.

Aaltosulut

Aaltosulkujen asettelu voi olla yksi kiistellyimpiä alueita C- ja C++ -ohjelmoijien välillä. Seuraavassa ovat omat vinkkini:

Aaltosulut sijoitetaan vaakatasossa pareittain samalle tasolle.
Uloimmat esittelyn tai määrittelyn aaltosulut sijoitetaan uloimmaksi vasemmalle. Sisällä olevat lauseet tulee sisentää. Kaikki muut aaltosulkuparit tulisi olla linjassa alkulauseen kanssa.
Mitään koodia ei saisi olla samalla rivillä aaltosulun kanssa. Esimerkiksi:

```
if (ehto == true)
{
  j = k;
  SomeFunction();
}
m++;
```

Pitkät rivit

Laita rivien pituus siten, että rivit näkyvät näytöllä. Koodi, joka on oikean laidan ulkopuolella, jää helposti ottamatta huomioon ja on tehtävä vaakasuora vieritys. Kun rivi katkaistaan, sisennä seuraavat rivit. Yritä katkaista rivi järkevästä paikasta ja yritä jättää kesken oleva operaattori edellisen rivin loppuun, jolloin on selvää, ettei rivi ole yksinään ja että lisää koodia liittyy riviin.

C++ -kielessä ovat funktiot usein paljon lyhyempiä kuin C-kielessä, mutta vanha ja hyvä neuvo pätee edelleenkin. Yritä pitää funktiot tarpeeksi pieninä, jotta ne voidaan tulostaa yhdelle A4-paperille.

Sarkaimen koon tulisi olla neljä välilyöntiä. Varmista, että editorisi muuttaa sarkaimen neljäksi välilyönniksi.

switch-lauseet

Sisennä switch-lauseet seuraavasti, jotta vaakasuora tila riittäisi:

```
switch(muuttuja)
{
  case ValueOne;
    ActionOne();
  break;
  case ValueTwo;
    ActionTwo();
  break;
  default;
    assert("bad Action");
  break;
}
```

Ohjelman teksti

Seuraavassa on useita vinkkejä, joita käyttäen voit luoda koodia, jota on helppo lukea. Helppolukuinen koodi on myös helppo ylläpitää.

Käytä välilyöntejä tukemaan luettavuutta.

Oliot ja taulukot viittaavat todellakin yhteen asiaan. Älä käytä välilyöntejä olioviittauksien sisällä (., ->, []).

Unaariset operaattorit liittyvät operandiinsa, joten älä laita välilyöntiä niiden väliin.
Laita välilyönti operandin oikealle puolelle. Unaarisia operaattoreita ovat !, ~, ++, --, -, * (osoittimille), & (muunnokset) ja sizeof.

Binääristen operaattoreiden kummallekin puolelle tulisi laittaa välilyönnit: +, =, *, /, %, >>, <<, >, <, ==, !=, &, |, &&, ||, ?:, =, +=, jne.

Älä käytä välilyönnin pois jättämistä osoittamaan prioriteettia (4+ 3*2)

Laita välilyönti pilkkujen ja puolipisteiden jälkeen, ei ennen.

Sulkumerkkien molemmille puolille tulisi laittaa välilyönti.

Varatut sanat, kuten if, tulisi erottaa välilyönnillä: if (a == b).

Kommentin runko aloitetaan välilyönnillä merkinnän // jälkeen.

Sijoita osoitin- tai viittausilmaisin tyypin nimen jälkeen, ei muuttujan nimen jälkeen. Tee näin

```
char* foo;
int& theInt;
```

Eikä näin

```
char *foo;
int &theInt;
```

Älä esittele enempää kuin yksi muuttuja samalla rivillä.

Tunnisteiden nimet

Seuraavassa on joitakin suuntaviivoja tunnisteiden nimeämiseen:

Tunnisteiden nimien tulisi olla tarpeeksi pitkiä ollakseen kuvaavia.

Vältä hämääviä lyhenteitä.

Käytä aikaa ja energiaa löytääksesi hyvät nimet.

Lyhyitä nimiä (i, p, x, jne) tulisi käyttää vain silloin, kun lyhyt nimi tekee koodista luettavampaa ja kun käyttö on niin selvää, ettei kuvaavampia nimiä tarvita.

Muuttujan nimen pituuden tulisi olla suhteessa näkyvyysalueeseen.

Varmista, että tunnisteet eroavat toisistaan sekaannusten välttämiseksi.

Funktion (tai metodin) nimet ovat tavallisesti verbejä tai verbi-substantiivi-yhdistelmiä: Search(), reset(), FindParagraph(), ShowCursor(). Muuttujien nimet ovat yleensä abstrakteja substantiiveja ja ehkä yhdyssanoja: count, state, windSpeed, windowHeight. Boolean-muuttujat tulisi nimetä sopivasti: winodwIconized, fileIsOpen.

Kirjoitustapa ja iso alkukirjain

Kirjoitustapa ja isojen kirjainten käyttö ovat tärkeitä seikkoja luotaessa omaa tyyliä. Seuraavassa on joitakin vinkkejä tältä alueelta:

Käytä isoja kirjaimia ja alaviivoja erottamaan loogisia sanoja nimiyhdistelmissä, kuten SOURCE_FILE_TEMPLATE. Huomaa kuitenkin, että ne ovat harvinaisia C++ -kielessä. Harkitse vakioiden ja mallien käyttöä useimmissa tapauksissa.

Kaikissa tunnisteissa tulisi käyttää isoja ja pieniä kirjaimia - ei alaviivoja. Funktioiden, metodien, luokkien, typedef-määrittelyn ja struct-nimien tulisi

alkaa isolla kirjaimella. Jäsenten, kuten tietojäsenten nimien tulisi alkaa pienellä kirjaimella.

Lueteltujen vakioiden nimet tulisi aloittaa muutamalla pienellä kirjaimella ikään kuin luettelon lyhenteinä. Esimerkki:

```
enum TextStyle
{
  tsPlain,
  tsBold,
  tsItalic,
  tsUnderscore
};
```

Kommentit

Kommenteilla tuetaan koodin ymmärtämistä. Joskus ohjelmaan ei palata uudelleen moneen päivään tai jopa kuukauteen. Tällä välin voit jo unohtaa, mitä tietty koodi tekee tai miksi se on mukana. Ongelmia syntyy helposti myös silloin, kun joku toinen lukee koodiasi. Kommentit, jotka on sijoitettu johdonmukaisesti ja toteutettu järkevästi, voivat maksaa itsensä nopeasti takaisin. Seuraavassa on muutamia kommentteja koskevia vinkkejä:

Käytä aina, kun mahdollista C++ -tyylistä kommentointia (//) C-tyylisen kommentoinnin (/* */) sijaan.

Korkeamman tason kommentit ovat ehdottomasti tärkeämpiä kuin yksityiskohtiin liittyvät. Lisää arvoa; älä pelkästään kuvaa koodia uudelleen. Esimerkiksi:

```
n++;  // n-arvoa kasvatetaan yhdellä
```

Tuo kommentti ei maksa vaivaa. Keskity funktioiden ja koodilohkojen toiminnan ydinkohtiin. Sano, mitä funktio tekee. Kerro vaikutukset, parametrien tyypit ja palautusarvot. Kuvaa kaikki olettamukset, jotka on tehty (tai ei ole), kuten "olettaen, että n on positiivinen" tai "palauttaa -1, jos x on epäkelpo". Kun logiikka on monimutkainen, käytä kommentteja osoittamaan ehdot, jotka ovat voimassa tuossa kohdassa koodia.

Käytä kokonaisia lauseita ja oikeita lauserakenteita. Ylimääräinen kirjoittaminen maksaa vaivan. Kirjoita selvästi äläkä käytä lyhennyksiä. Se, mikä tällä hetkellä näyttää kristallin kirkkaalta, saattaa parin kuukauden kuluttua olla hämmästyttävän sekavaa.

Käytä tyhjiä rivejä auttamaan lukijaa ymmärtämään, mitä on meneillä. Erottele lauseet loogisiin ryhmiin.

Käsittelymääritteet

Tapa, jolla ohjelmasi osia voidaan käyttää, tulisi kehittää johdonmukaiseksi. Seuraavassa on joitakin vinkkejä:

Käytä aina public:, private: ja protected-otsikoita; älä nojaudu oletuksiin.

Luettele public-jäsenet ensin, sitten protected-jäsenet ja lopuksi private-jäsenet. Luettele tietojäsenet ryhmänä metodien jälkeen.

Sijoita muodostimet ensin sopivaan jaksoon ja sitten tuhoajafunktiot. Luettele ylikuormitetut metodit toisiinsa liittyvillä nimillä. Ryhmittele käsittelyfunktiot yhteen, mikäli mahdollista.

Harkitse metodien nimien laittamista aakkosjärjestykseen ryhmän sisällä samoin kuin jäsenmuuttujienkin. Varmista, että laitat include-lauseiden tiedostonimet aakkosjärjestykseen.

Vaikkakin virtual-sanan käyttö on valinnaista korvauksen yhteydessä, käytä sitä kuitenkin; se auttaa muistuttamaan, että funktio on virtuaalinen ja pitää esittelyn johdonmukaisena.

Luokkamäärittely

Yritä pitää metodien määrittelyt samassa järjestyksessä kuin esittelyt. Se helpottaa asioiden löytämistä.

Sijoita funktion palautustyyppi ja muut muuntimet edelliselle riville, jotta luokan nimi ja funktion nimi alkaisi vasemmasta marginaalista. Menettely auttaa funktioiden löytämisessä.

Sisällytettävät tiedostot

Yritä välttää tiedostojen sisällyttämistä otsikkotiedostoihin. Ihanneminimi on otsikkotiedosto luokalle, josta tämä yksi johdetaan. Muut mahdolliset sisällyttämiset koskevat olioita, jotka ovat esiteltävän luokan jäseniä. Luokat, joihin pelkästään osoitetaan tai viitataan tarvitsevat vain ennakkoviittaukset muotoon.

Älä jätä pois include-tiedostoa otsikosta vain siksi, koska oletat, että se, mitä .CPP-tiedostoon sisällytetään, tarvitsee aina oman include-lauseensa.

Vinkki
Kaikkien otsikkotiedostojen tulisi käyttää sisällyttämisen tarkistusta.

assert()

Käytä assert()-funktiota vapaasti. Se auttaa virheiden löytämisessä, mutta tukee myös lukijaa selvittämään, mitkä ovat olettamukset. Se auttaa myös kiinnittämään huomion siihen, mitä kirjoittajan mielestä on oikein ja mikä ei.

const

Käytä const-sanaa aina, kun se on sopivaa: parametrien, muuttujien ja metodien yhteydessä. Usein on tarvetta sekä vakiolle että ei-vakiolle metodiversiolle: älä käytä tätä selityksenä toisen pois jättämiselle. Ole huolellinen muuntaessasi const-muodosta ei-const-muotoon ja päinvastoin - joskus se on ainoa keino tehdä jotakin - mutta varmista, että se on järkevää ja laita kommentti selitykseksi.

Seuraavat vaiheet

Olet nyt työskennellyt kovasti ja voit pitää itseäsi hyvänä C++ -ohjelmoijana, mutta et kuitenkaan ole millään tavoin valmis. On vielä paljon opittavaa ja saatava runsaasti lisää eväitä siirtyäksesi noviisista asiantuntijaksi.

Seuraavissa jaksoissa suositellaan erityislähteitä, joiden esille ottaminen perustuu omaan kokemukseeni ja mielipiteisiini. Kustakin aihealueesta on olemassa tukuttain resursseja, joten yritä kysellä mielipiteitä muilta ennen niiden hankkimista.

Mistä saada apua ja neuvoja

Ensimmäisenä kannattaa C++ ohjelmoijan olla yhteydessä yhteen tai useampaan on-Internet-keskusteluryhmään. Näiden ryhmien kautta pääsee nopeaan yhteyteen satojen ja jopa tuhansien C++ -ohjelmoijien kanssa; he voivat vastata kysymyksiisi, antaa neuvoja ja pohtia ideoitasi.

Hyvää luettavaa

Kirja, joka kannattaa hankkia ja lukea on

Effective C++ (ISBN: 0-201-56364-9), Scott Meyer, Addison-Wesley, 1993.

Tämä on ollut hyödyllisin lukemani kirja ja olen lukenut sen kolmeen kertaan.

Lehdet

On tärkeää lukea useitakin kirjoja alueelta ja olla yhteydessä keskusteluryhmiin, mutta voit parantaa taitojasi vielä yhtä kanavaa pitkin: tilaa itsellesi hyvä C++ -ohjelmointiin liittyvä ammattilehti. Ehdottomasti paras tällainen lehti on mielestäni C++ Report, jota julkaisee SIGS. Säästä hyödylliset artikkelit; voit tarvita niitä jo pian.

Voit tilata lehteä osoitteesta: SIGS Publications, P.O.Box 2031, Langhorne, PA 19047-9700. Minulla ei ole osuutta lehteen (työskentelen muille julkaisijoille!), mutta lehti on paras.

Lähetä palautetta

Jos sinulla on kommentteja, ehdotuksia tai ideoita koskien tätä tai muita kirjoja, haluaisin kuulla ne. Lähetä minulle sähköpostia osoitteeseen jliberty@libertyassociates.com tai käy www-sivuillani www.libertyassociates.com. Odotan yhteydenottoasi.