16. Sigurnije programiranje

16.1 Pretvorba tipova

Pretvorba realnog broja u cijeli broj se može izvršiti naredbama:

```
int i;
double d;

i = (int) d;
```

ili:

```
i = int(d);
```

U cilju bolje kontrole pretvorbe tipova ANSI-C++ standard je definirao četiri nova operatora pretvorbe tipova: **reinterpret_cast**, **static_cast**, **dynamic_cast** i **const_cast**. Svi oni se koriste u obliku:

```
static_cast <tip>       (izraz)
reinterpret_cast <tip> (izraz)
const_cast <tip>        (izraz)
dynamic_cast <tip>      (izraz)
```

gdje <*novi_tip*> označava u koji se tip vrši pretvorba tipa (*izraza*).

static_cast

omogućuje sve pretvorbe koje se mogu i implicitno izvršiti (pr. int u float), kao i inverzne pretvorbe (čak i one koje se ne mogu implicitno izvršiti).

```
float f=3.14159265;
int i = static_cast<int>(f);          // =  (int)f
```

Kada se primijeni na pokazivače klasa, omogućuje pretvorbu pokazivača izvedene klase na pokazivač temeljne klase i obrnuto.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

reinterpret_cast

reinterpret_cast omogućuje pretvorbu pokazivača u bilo koji tip pokazivača.
Pri toj operaciji vrši se jednostavno binarno kopiranje sadržaja varijable, bez provjere tipa, pa se može koristiti kod pokazivača na objekte različitih klasa.

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

reinterpret_cast, dakle, treatira sve pokazivače na način kako je to bilo uobičajeno kada se radi s void pokazivačima

dynamic_cast

se upotrebljava s referancama i pokazivačima na objekte. Omogućuje bilo koju pretvorbu koja se implicitno može provesti, kao i inverzne pretvorbe, kada se radi s polimorfnim objektima. Za razliku od **static_cast**, **dynamic_cast** u ovom slučaju provjerava da li se operacija može izvršiti uspješno, tj. provjerava da li se dobija objekt ispravnog tipa. Ova provjera se vrši u toku izvršenja programa. Ukoliko se ne može izvršiti pretvorba pokazivača vraća se NULL pokazivač.

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1 = dynamic_cast<Derived*>(b1);   // ok
Derived* d2 = dynamic_cast<Derived*>(b2);   // greska: vraća NULL
```

const_cast

Pomoću **const_cast** može se postaviti ili skinuti **const** atribut objekta:

```
class C {};
const C * a = new C;
C * b = const_cast<C*> (a);
```

Identifikacija tipa - typeid

ANSI-C++ definira operator imena `typeid`**.** Njime se može ispitati tip nekog izraza, na način da izraz

`typeid` **(***expression***)**

vraća referencu na konstantni objekt tipa **`type_info`**, koji je definiran u zaglavlju `<typeinfo>`**.** Te reference se mogu uspoređivati pomoću operatora == i **!=**, ili se mogu koristiti za dobijanje stringa koji opisuje tip (ili ime klase). U tu svrhu koristi se članska funkcija **`type_info`** klase imena **`name()`,** koja vraća string koji sadrži ime tipa.

```
#include <iostream>
#include <typeinfo>
using namespace std;

class Cl { };

void main () {
  Cl a;
  Cl *b;
  if (typeid(a) != typeid(b)){
    cout << "Objekti a i b su razlicitog tipa:\n";
    cout << "Tip od a je: " << typeid(a).name();
    cout << "Tip od b je: " << typeid(b).name();
  }
}
```

Rezultat:

```
Objekti a i b su razlicitog tipa:
Tip od a je: class Cl
Tip od b je: class Cl *
```

Prihvat i generiranje iznimki - Exception handling

try-catch-throw

Ideja je da se program podijeli na dio u kojem može nastati iznimka i na dio u kojem se on nastavlja kada nastane iznimka. Ključna riječ ***try*** označava blok programa u kojem može nastati iznimka, a ključna riječ ***catch*** označava dio programa koji će se izvršiti kada nastane neka iznimka (Za catch-blok se često koristi naziv ***exception-handler***). Uz ključnu riječ ***catch*** u zagradama se navodi parametar koji označava tip iznimke. Neke iznimke generira izvršni sustav C++ kompajlera, a neke može generirati korisnik koristeći ključnu riječ ***throw***.
Pojednostavljeno se ovaj mehanizam može zapisati u obliku:

```
class Iznimka {}        // može biti "prazna klasa"

try {   // program koji pokušavamo izvršiti
  if (neka_greška) throw Iznimka();
  // iznimku može generirati i izvršni sustav
}
catch (Iznimka){
  // dio programa koji se izvršava kada korisnik
  // generira iznimku tipa Iznimaka
}
catch (std:exception& e){
  // dio programa koji se izvršava kada nastane
  // iznimka  koju generira standardne biblioteka
}
catch (...){
  // dio programa koji se izvršava za sve ostale
  // iznimke
}
```

Primjer: U prvom primjeru pokazat ćemo kako se generiraju uznimke koje u catch-blok šalju vrijednost prostih tipova

```
try  {
    int n;
    char *mystring = new(nothrow) char [10];
    if (mystring == NULL)
         throw "Greska alokacije memorije!";
    cout << "Unesi cijeli broj: ";
    cin >> n;

    if (n>9)
         throw n;
    else
      mystring[n]='z';
 }
 catch (int i)  {
    cout << "Iznimka: ";
    cout <<"indeks "<<i
        <<" izvan dozvoljenog intervala!"
        << endl;
 }
 catch (char * str)  {
    cout << "Iznimka: " << str << endl;
 }
```

Rezultat:

```
Unesi cijeli broj: 99
Iznimka: indeks 10 izvan dozvoljenog intervala!
```

Primjer: Pokazat ćemo kako se koriste korisnički definirane klase kao parametri catch-bloka. Klasu Range-error koristit ćemo za generiranje iznimke kada se prekorači neki dozvoljeni cjelobrojni interval:

```
class Range_error {
public:
     Range_error(int i) :m_i(i) {}
     int m_i;
};
```

Pretpostavimo da smo definirali funkciju `int2char()` koja vraća znakovnu vrijednost ako je cjelobrojni argument unutar intervala [0,255], u suprotnom generira se iznimka Range_error.

```
char int2char(int i){
     if(i < 0  || i>255)  throw Range_error(i);
     return (char) i;
}
```

Pri korištenju ove funkcije moguća su dva načina prihvata iznimke. Prvi način je da catch ne koristi argument:

```
try {
    char c = int2char (600);
}
catch(Range_error) {
    cout << "int2char: nedozvoljen argument";
}
```

Drugi je način da se catch specificira s formalnim parametrom x, pa tada može primati argumente:

```
try {
    char c = int2char (600);
}
catch(Range_error x) {
    cout << " int2char: nedozvoljen argument "
         << x.m_i <<  endl;
}
```

Neprihvaćene iznimke

Ako nastane iznimka za koju nije osiguran prihvat (i ako ne postoji catch(...)) tada se poziva specijalna funkcija **terminate()**, kojom se prekida trenutni proces (program). Zapravo, **terminate** je pokazivač na funkciju kojem je predodređena vrijednost na adresu standardne C funkciju **abort( )**, kojom se prekida program. To znači da neće biti pozvani destruktori globalnih i statičkih objekata, pa se ovakovi tip neprihvaćenih iznimaka smatra programerskom pogreškom.
Moguće je da korisnik sam definira funkciju za prihvat ovakovih iznimki. To se vrši pomoću funkcije **set_terminate()**, koja vraća pokazivač na funkciju terminate(), a prima kao argument void funkciju koja nema argumenata i koja će zamijeniti funkciju terminate(). Primjer upotrebe je dan u programu setterm.cpp

Ugniježdeni try-catch

Try-catch-blokovi mogu biti ugniježdeni. Primjerice:

```
try {
   try {
      // neki kod
   }
   catch (int n) {
      throw;        // preusmjeri na vanjski catch-blok
   }
}
catch (...) {
  cout << "Iznimka";
}
```

U slučaju ugniježdenih blokova moguće je usmjeriti iznimku iz unutarnjeg **catch-**bloka u vanjski **catch-**block. U ovom primjeru smo u tu svrhu u unutarnjem bloku koristili **throw** bez argumenta. Preporuka je da se ne koriste ugniježdeni try-catch blokovi.

Grupiranje iznimki

Često se iznimke mogu logično grupirati. Primjerice, za rad s matetematičkim operacijama mogli bi definirati sljedeće:

```
class MathErr() {};            // greska u mat. operacijama
class ZeroDevide: public MathErr {};  // dijeljenje s nulom
class Overflow: public MathErr {}:    // prekoračen maksim.

void f()
{
     try {
               //matematicke operacije
    }
    catch(ZeroDevide) {
               // dijeljenje s nulom
    }
    catch(MathErr){
                // prihvaca sve ostale mat. greske
                //osim dijeljenja s nulom
    }
}
```

U ovom slučaju se se koristi naslijeđivanje za definiranje objekta identifikacije iznimke. Zbog toga, u ovom primjeru se dijeljenje s nulom prihvaća u catch(ZeroDevide), a ostale matematičke iznimke u catch(MathErr) uključujući i Overflow (jer se Overflow izvodi iz MathErr).

Deklaracija funkcija s iznimkama

Kada deklariramo neku funkciju

```
int f ();
```

nije poznato da li se u njoj koristi ili ne koristi mehanizam generiranja iznimki.

```
int f() throw();
```

znači da se u funkciji f() ne generiraju iznimke, a

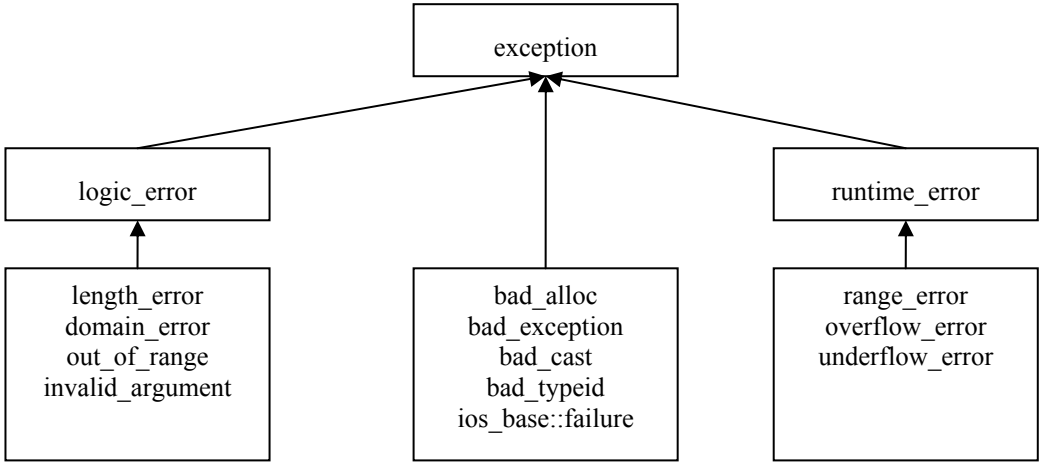
```
int f() throw(MarhErr);
```

znači da se u funkciji f() generira iznimka MathErr.

Standardne iznimke

Standardne iznimke generiraju se iz koda koji stvara sam C++ kompajler ili iz koda standardne biblioteke. Deklarirane su pomoću temeljne klase `exception`, koja je deklarirana u zaglavlju **`<exception>`**:

```
class exception {
public:
     exception() throw();
     exception(const exception&) throw();
     exception& operator = (const exception&) throw();
     virtual ~exception() throw();
     virtual const char * what() const throw();
private:
     //
}
```



Slika 16.1 Hijerarhija naslijeđivanja klase exception

```
catch(bad_alloc)
{
     // greska alokacije
}
catch (std::exception & e)  // ostale izvršne greške
{
    cout << "Exception: " << e.what();
}
```