Informatika 3

Výnimky 9

Čo je to výnimka

- Reprezentuje nejakú ***zlú*** udalosť
- Predstavuje nezvyčajný a neočakávaný stav
- Môžu byť spôsobené externými faktormi a faktormi prostredia

Ako ošetrovať chyby

- Program sa ukončí.
- Funkcie, detekujúce výnimku návratovou hodnotou - stavová hodnota, indikujúca nežiadúci stav; globálny indikátor errno a perror
- Vytvorenie spätne volanej funkcie na ošetrenie chýb, ktorú budú pri výskyte výnimky volať nízko-úrovňové funkcie.
- Vykonanie vzdialeného skoku – setjmp, longjmp zo štandardnej C knižnice
  - setjmp – uloží sa dobrý stav programu
  - longjmp – obnoví sa tento stav
- Signály - funkcie signal a raise so štandardnej C knižnice (asynchrónne)

- Ani jeden nevie volať deštruktory
- Pre C++ sa používajú výnimky

Príklad

```
class A {
  public:
    A() { cout << "A()" << endl; }
   ~A() { cout << "~A()" << endl; }
};

jmp_buf jbuf;

void fun()
{
  A a;
  for(int i=0;i<3;i++) {
    cout << "fun()\n";
  }
  longjmp(jbuf,1);
}
```

```
int main()
{
  if(setjmp(jbuf) == 0) {
    cout << "setjmp=0...\n";
    fun();
  } else {
    cout << "setjmp!=0..." << endl;
  }
  return 0;
}
```

OO ošetrovanie výnimiek

- Ošetrenie výnimiek by malo byť vlastnosťou programovacieho jazyka
- Spôsob ošetrovania výnimiek by mal byť OOP orientovaný
- Ošetrovanie výnimiek musí byť flexibilné, musí podporovať čo najviac najbežnejších typov výnimiek a ich ošetrenie
- Mechanizmus ošetrovania výnimiek musí byť prekrývateľný programátorom

Základný problém

- Indikácia výnimky
- Ošetrenie výnimky

Vyslanie výnimky

- Výnimka sa vyvoláva kľúčovým slovom **throw**

```
#include <iostream.h>

const int CHYBA = -1;

void Fun(unsigned char *data,long size)
{
     if(size >1000)
     // nedokážem spracovať tak veľa dát
     throw CHYBA;
     // ... spracovanie dát
}
```

Zachytenie výnimky

```
try {
    /* riadený blok */
}
catch(Vynimka v) {
    /* ošetrenie výnimky */
}
catch(Vynimka2 v2) {
    /* ošetrenie inej výnimky */
}
```

- Ak výnimka nie je zachytená, prejde sa do ďalšej obaľujúcej sféry
- Porovnávanie
  - Skočí do catch bloku podľa typu výnimky
  - typ musí byť rovnaký alebo potomkom typu parametra catch bloku
  - Nevykonávajú sa konverzie jedného typu na iný
  - V catch môže byť odkaz na objekt (nevolá sa copy-konštruktor)

Príklad

```
// Výnimky
class A {
  public:
  A() { cout << "A()" << endl; }
 ~A() { cout << "~A()" << endl;
}
};

void Fun()
{
  A a;
  for(int i = 0; i < 3; i++)
    cout << "fun()\n";
  throw 47;
}
```

```
int main() {
  try {
    cout << "main()" << endl;
    Fun();
  }
  catch (int) {
    cout << "catch(int) "<< endl;
  }
}
```

```
class Vynimka{
    char* text;
public:
    Vynimka(char* txt) { text=txt; }
    char* Preco() { return text; }
}
double vydel(double x, double y){
    if(y==0)
        throw Vynimka("Delenie nulou");
    return x / y;
}

main()
{   …
    try{
            int a=3, b=0;
            double x=vydel(a, b); // nastane výnimka a odvíjanie zásobníka
            …
    }
    catch(Vynimka& v){
            cout << v.Preco() << '\n';
    }
}
```

```
class Chyba {
};

class Varovanie : public Chyba {
};

class Fatal : public Chyba {
};

class A {
  public:
    void Fun() { throw Fatal(); }
};
```

```
int main() {
  A a;
  try {
    a. Fun ();
  }
  catch(Chyba &) {
    cout << "Chyba" << endl;
  }
  // Skryté
  catch(Varovanie &) {
    cout << "Varovanie" << endl;
  }
  catch(Fatal &) {
    cout << "Fatal" << endl;
  }
}
```

Modely ošetrenia výnimiek

- ukončenie – používa C++
- obnovenie – ak potrebujeme musíme si to naprogramovať

```
int main() {
  bool ok = true;
  while(ok==true) {
      try {
        throw 50;
      }
      catch(int x) {
         // OPRAVA
        if(Skoncit==ANO)
          ok=false;
      }
    }
}
```

Pravidlá používania výnimiek

- Každú výnimku treba zachytiť – neošetrená výnimka je programová chyba
- Nezachytená výnimka vyvoláva terminate() - volá abort()
  - dá sa zmeniť pomocou set_terminate()
- V deštruktore negenerovať výnimku (odporúčanie)
  - vo výnimke môže nastať ďalšia výnimka
  - V deštruktore lokálnej premennej pri odvíjaní zásobníka výnimka volá terminate()
- V konštruktore byť opatrný – ak nastala výnimka v konštruktore, deštruktor tohto objektu sa nevykoná
- Zachytávať výnimky odkazom
- Používať potomkov štandardných výnimiek
- Zachytenie akejkoľvek výnimky:

```
catch(…){
   cout << “Výnimočný stav\n”;
   throw; // prevyslanie výnimky
}
```

Obalová trieda

```
class A {
    int *Pole_d;
  public:
    A() {
      cout << "A()" << endl;
      Pole_d=new int[100];
      rob_nieco();
    };
    void rob_nieco() {
      Pole_d[0]=1;
      throw 47;
    };
   ~A() {
      cout << "~A()" << endl;
      delete []Pole_d;
    };
};
```

```
int main() {
  try {
    A ur;
  }
  catch(int) {
    cout<<"Obsluha vynimky"<<endl;
  }
}
```

A()
Obsluha vynimky

Obalová trieda - pokračovanie

```
class Obal {
    int *Pole_d;
  public:
    Obal(int vel) {
      cout << "Obal() - alokujem 'vel' int" << endl;
      Pole_d=new int[vel];
    };
   ~Obal() {
      cout << "~Obal() - dealokujem 'vel' int" << endl;
      delete [] Pole_d;
    };
    operator int *() { return Pole_d; };
};
```

Obalová trieda - použitie

```
class A {
    Obal Pole_d;
  public:
    A() : Pole_d(100) {
      cout << "A()" << endl;
      rob_nieco();
    };
    void rob_nieco() {
      Pole_d[0]=1;
      throw 47;
    };
   ~A() {
      cout << "~A()" << endl;
    };
};
```

```
int main() {
  try {
    A ur;
  }
  catch(int) {
    cout <<"Obsluha vynimky"<< endl;
  }
}
```

Obal() - alokujem 'vel' int
A()
~Obal() - dealokujem 'vel' int
Obsluha vynimky

Výnimky na úrovni funkcií

```
double vydelPole(double* pole, double y, int n)
try
{
  for(int i=0 ; i<n ; i++){
        pole[i]=vydel( pole[i], y);
  }
}
catch(Vynimka v)
{
  cout<<v.Preco()<<'\n';
}
```

- To isté platí aj pre metódy
- Často sa používa v konštruktoroch

Výnimka na úrovni funkcie

```
class cZakladnaTrieda {
  int i;
public:
  class cVynimka {};

  cZakladnaTrieda(int i) : i(i) {
    throw cVynimka();
  }
};
```

Výnimka na úrovni funkcie-pokr.

```
class cOdvodenaTrieda : public cZakladnaTrieda {
public:
  class cVynimkaOdv {
    const char* msg;
  public:
    cVynimkaOdv(const char* msg) : msg(msg) {}
    const char* Oznam() const {
      return msg;
    }
  };
  cOdvodenaTrieda(int j) // Konštruktor
  try
    : cZakladnaTrieda(j) {
    // Telo konštruktora
    cout << "Toto by sa nemalo vytlačit" << endl;
  }
  catch (cVynimka &) {
    throw cVynimkaOdv("cZakladnaTrieda subobjekt vyslal vynimku");;
  }
};
```

Špecifikácia výnimky

- Môžeme definovať, ktoré výnimky metóda generuje:

```
class complex {
   double r, i;
public:
   complex vydel(double delitel) throw(Vynimka, Vynimka2) { … }
};
```

- Ak throw(v1,v2,v3) nie je uvedené, metóda môže vyhodiť ľubovoľnú výnimku.
- throw( ) – metóda nevyhadzuje žiadnu výnmku
- unexpected() – funkcia {nastavujeme cez set_unexpected()}, ktorá sa volá ak metóda vyhodí inú výnimku ako je uvedená v hlavičke – štandardne volá terminate() {nastavujeme cez set_terminate()}
- U šablón je deklarácia výnimiek prakticky nepoužiteľná

Štandardné výnimky

- exception – základná trieda
- logic_error – chyby programovacej logiky (napr. nesprávny parameter)
- runtime_error – chyba hardvéru, vyčerpanie pamäte

Používanie výnimiek

- Nie pre asynchrónne udalosti
- Nie pre mierne chybové podmienky
- Nie pre riadenie priebehu programu

- Vyriešenie problému a opätovné zavolanie funkcie ktorá výnimku spôsobila.
- Zorganizovanie vecí a pokračovanie bez opätovného volania funkcie.
- Výpočet nejakého alternatívneho výsledku namiesto toho, čo mala poskytovať funkcia.
- Urobenie hocičoho v aktuálnom kontexte a znovu vyslanie ***tej istej*** výnimky do vyššieho kontextu.
- Urobenie hocičoho v aktuálnom kontexte a znovu vyslanie ***odlišnej*** výnimky do vyššieho kontextu.
- Ukončenie programu
- Obaľovacie funkcie (najmä C knižničné funkcie, ktoré používajú obyčajnú chybovú schému tak, aby poskytovali výnimky).
- Zjednodušenie. Ak schéma výnimiek robí veci komplikovanejšími, nie je vhodné používať ju.
- Vytvorenie bezpečnejšej knižnice a programu. Toto je dlhodobá investícia (odolnosť aplikácie).

Pravidlá

- Začnime so štandardnými výnimkami
- Vnárajme svoje vlastné výnimky
- Používajme hierarchie výnimiek
- Zachytávajme odkazom, nie hodnotou
- Vysielajme výnimky v konštruktoroch
- Negenerujme výnimky v deštruktoroch

Informatika 3

Ukladacie triedy

Deklarácia

- Deklarácia premenných

<ukladacia_trieda> <modifikátor> <deklarátor> pr1, pr2, … prN;

- Ukladacia trieda:
  - **extern**
  - **static**
  - **register**
  - **volatile**

Automatická premenná

- deklarovaná v zloženom príkaze (funkcii)
- lokálne v zloženom príkaze
- pri začiatku vykonávania zloženého príkazu premenná vzniká, pri skončení zaniká
- je alokovaná v zásobníku – rýchla alokácia
- kľúčové slovo **auto** je nepovinné – lokálne premenné sú implicitne auto

```
auto int i=0;
```

Externé premenné

- extern
  - deklarácia premennej bez alokácie pamäťového miesta
  - deklarované mimo akejkoľvek funkcie
  - globálne premenné, dostupné v celom programe
  - kľúčové slovo **extern** - použité iba pre deklaráciu premennej, ktorá bola definovaná v inom zdrojovom súbore
  - Vo všetkých zdrojových súboroch musí byť externá premenná dekladovaná slovom **extern**, ale v jednom musí byť definovaná bez kľúčového slova **extern**
  - najčastejšie používané deklarácie globálnych premenných v hlavičkových súboroch

Statická premenná

- static
  - platia pre ňu pravidlá ako pre globálnu premennú
    - **vzniká pri spustení programu**
    - **inicializuje sa na nulu alebo na hodnotu na zadanú pri deklarácií**

- static – lokálne premenné
  - deklarované v zloženom príkaze alebo vo funkcii
  - identifikátor premennej je platný v rámci zloženého príkazu alebo funkcie
  - po vykonaní zloženého príkazu/funkcie nie je premenná zrušená
  - zostáva jej hodnota z predchádzajúceho vykonávania

```
int pocitadlo()
{
    static int pocet_volani = 1;      // !!! inicializácia
                                      // iba pri spustení programu
    return pocet_volani++;
}
```

- static – globálne premenné
  - identifikátor premennej je platný iba v rámci zdrojového súboru
  - je opakom ku **extern** – premenná sa nedostane do tabuľky krížových referencií

Registrové premenné

- register
  - pre premenné ku ktorým potrebujeme rýchly prístup
  - prekladač sa pokúsi uložiť premennú do registra procesora
  - nie je zaručené, že premenná bude v registri (obmedzené množstvo registrov)
  - nemožno na premennú použiť operátor adresy

- volatile
  - pre premenné, ktoré môžu meniť svoju hodnotu mimo program (napr. cez prerušenie)
  - každá operácia s premennou je vykonávaná priamo v pamäti
  - operácie sú pomalšie

Kľúčové slová v deklarácii funkcie

- Deklarácia funkcie

<typ> <meno>(parametre);

- Pred deklaráciou funkcie môže byť:
  - **extern**
  - **static**
  - **interrupt**
  - **inline**

- extern
  - deklarácia funkcie – hlavička
  - kľúčové slovo **extern** pre funkcie je nepovinné, hlavičky funkcií sú implicitne typu **extern**
  - identifikátor funkcie sa dostane do tabuľky krížových referencií – implicitné chovanie hlavičiek funkcií
- static
  - je opakom ku **extern** – identifikátor funkcie sa nedostane do tabuľky krížových referencií

extern, static pre premenné a funkcie

- Pre globálne premenné a funkcie sa extern a static používa pri linkovaní viacerých zdrojových súborov
- **static** – funkcia/globálna premenná platná iba v danom zdrojovom súbore
- **extern** – funkcia/globálna premenná platná v celom programe

Prerušenie

- interrupt
  - funkcie, ktoré sú väčšinou volané cez prerušenie
  - pred spustením funkcie sa uchová aktuálny stav registrov
  - po ukončení funkcie je obnovený stav registrov
  - volanie funkcie je pomalšie

- inline
  - pre jednoduché funkcie
  - funkcia nie je volaná ako štandardná funkcia ale je kompilovaná priamo do volania
  - ak sa používa vo viacerých zdrojových súboroch, musí byť definovaná (aj s telom) v hlavičkovom súbore
  - nie je možné získať smerník na funkciu

Veľké projekty

- Rozdeliť program do viacerých zdrojových súborov
- Používať hlavičkové súbory
- V hlavičkovom súbore:
  - deklarácia štruktúr
  - deklarácia globálnych premenných s kľúčovým slovom **extern**
  - hlavičky funkcií
  - symbolické konštanty
- Prekladač automaticky zistí závislosti zdrojových súborov na hlavičkových súboroch
- Pri zmene hlavičkového súboru sa prekladajú len súbory, ktoré sú na ňom závislé
- #include aj v súbore, kde sa funkcia definuje
- linkovanie – tabuľka krížových referencií

Premenlivý počet parametrov

Premenlivý počet parametrov

- Potrebné typy:

va_list – typ premennej ukazujúcej do zásobníka

- Používame makrá:

va_start(va_list, last_fix_param)

- **inicializácia zásobníka**

va_arg(va_list, typ)

- **vracia aktuálny parameter zo zásobníka**

va_end(va_list)

- **upratanie zásobníka**

Príklad

- Napíšte funkciu max_double, ktorá vráti najväčší prvok so zadaných parametrov typu double. Prvý parameter bude počet prvkov.
- Použitie:

```
double x = max_double( 4, 1.3, 2.4, 0.2, 9.6 );
```

- Riešenie:

```
#include <stdio.h>
#include <stdarg.h>
double max_double ( int num, ...)
{
    int i = 0;
    double ret = -100000;
    va_list argptr;
    va_start(argptr, num);
    for( i = 0 ; i < num ; i++ ) {
            double x = va_arg( argptr, double );
            if( x > ret ) ret = x;
    }
    return ret;
}
int main()
{
    double x = max_double( 4, 1.3, 2.4f, 0.2, 9.6 ); // float sa automaticky konvertuje
    printf( “max = %f\n”, x );                       // na double
}
```

Práca s vláknami

Vytvorenie vlákna

- Vo windows

```
uintptr_t _beginthread(
        void (*ThreadFunc)(void*),
        unsigned _StackSize,
        void* _ArgList
        );
```

- V Linuxe

```
int pthread_create(
        pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg
        );
```

Čakanie na ukončenie vlákna

- Vo windows

```
DWORD WaitForSingleObject(
        HANDLE hHandle,
        DWORD dwMilliseconds
        );
```

- V Linuxe

```
int pthread_join(
        pthread_t thread,
        void **value_ptr
        );
```

Zásady multithreading

- Funkcie reentrantné –funkcie, ktoré nemajú vnútorný stav (napr. pri použití premenných typu static)
- Používať/tvoriť triedy vláknovo-bezpečné
- Využívať prvky IPC (inter-process communication)
  - Semafóry
  - Signály
  - Fronty správ
  - Rúry (pipes)
  - atď.

Príklad nereentrantnej funkcie

```
char* IntToString(int val)
{
    static char buf[512];
    sprintf(buf, “%d”, val);
    return buf;
}

…
printf(“%s, %s”, IntToString(10), IntToString(20));
…

// Čo sa zobrazí na obrazovke? Prečo?
```

Práca s vláknami

```
#include <windows.h>
#include <process.h>
#include <math.h>
using namespace std;

#define NUM_THREADS 10

void func(void* data)
{
  int id=*(int*)data;
  for (double i=0;i<100000000;i++)
  {
     sin(i);
  }
  cout << id << endl;
  delete data;
}
```

```
int main(int a, char* c[])
{
  HANDLE vlakna[NUM_THREADS];
  for (int i=0;i<NUM_THREADS;i++)
  {
     int* data=new int;
     *data=i;
     vlakna[i]=(HANDLE)
              _beginthread(func, 0, data);
  }
  WaitForMultipleObjects(
     NUM_THREADS,
     vlakna, true, INFINITE);
     return 0;
}
```

Informatika 3

Metapríkazy

Metapríkazy

- Sú vykonávané preprocesorom
- Začínajú znakom ‘#’
- Typy metapríkazov:
  - symbolické konštanty
  - makrá
  - vkladanie súboru
  - podmienený preklad
  - direktívy pre prekladač

Symbolické konštanty

- Nahrádza identifikátor textom uvedeným za ním
- Zvykom je písať ich identifikátory veľkými písmenami

```
#define PI 3.1415926535
#define BIELA “biela”
```

- Hodnota nemusí byť uvedená – využíva sa hlavne pri podmienenom preklade

```
#define WIN32
```

- V hodnote môžu byť komentáre /*…*/ aj //...

```
#define MIN 2 /* minimalna hodnota */
#define MAX 10 // maximalna hodnota

for(int i=MIN ; i<MAX ; i++){
   …
}
```

- platnosť symbolickej konštanty končí makroinštrukciou “undef”

```
#undef identifikátor
```

Makroinštrukcie - Makrá

```
#define identifikátor(zoznam parametrov) reťazec
```

- definícia makra s formálnymi parametrami
- v programe sa pri volaní makra dosadia za formálne parametre skutočné parametre

```
#define chodna(riadok,stlpec) gotoxy(stlpec,riadok)
```

- použitie

```
chodna (3,2)          // preprekladac nahradi gotoxy(2,3)
```

- operátor ## spája argumenty makra do jedného výrazu:

```
#define FUNKCIA(i,j) funkcia (i##j)
FUNKCIA (x,9)         // preprekladac nahradi funcia (x9)
```

- ak je pred formálnym parametrom znak #, skutočný parameter je konvertovaný na reťazec

```
#define vypi(premenna) printf(#premenna"=%d\n",premenna)
x=10;
vypis(x);             // preprekladac nahradi printf("x""=%d\n",x);
```

- ako identifikátory nesmú byť použité štandardné makrá C-jazyka (_STDC, _DATE, ...)
- platnosť makra končí makroinštrukciou “undef”

```
#define pocet 200
....
#undef pocet
```

KST

Makroinštrukcie - Makrá

- Pri použití je odporúčané používať zátvorky:
  - každý použitý parameter
  - celé makro

```
#define max(a,b) ((a)>(b) ? (a) : (b))
x=max(a,max(b,c));          // ked nebudu zatvorky?
```

- Výhody
  - rýchle operácie
  - podobné ako inline funkcie
  - jedno makro pre viaceré typy parametrov
- Nevýhody
  - zle sa hľadá chyba

Podmienený preklad

```
#if konštantný výraz
...
#elif konštantný výraz
...
#elif konštantný výraz
...
#else
...
#endif
```

- predkladá sa iba jedna vetva, v ktorej je konštantný výraz nenulový
- ak všetky konštantné výrazy sú nulové, bude sa preklať vetva #else
- vetva #else je nepovinná
- je možné využiť metaoperátor

```
defined (identifikátor)          // zátvorky sú nepovinné
```

  - kde:

```
defined(id) = 1      //ak bolo definované #define id, inak
defined(id) = 0      //ak id nie je definované
```

- skrátenie:

```
#if defined(id)    <=> #ifdef id
#if !defined(id)   <=> #ifndef id
```

Vloženie súboru do zdrojového programu

- #include "meno súboru”
- vkladaný súbor vyhľadáva prednostne v aktuálnom adresári, až potom v adresároch, nastavených pomocou OPTIONS/ENVIRONMENT/INCLUDE/DIRECTORIES resp. -I<dir>

- #include <meno súboru>
- súbor vyhľadáva v adresároch OPT./ENV./INCLUDE DIRECTORIES resp. -I<dir>
- namiesto mena súboru je možné použiť makro, obsahujúce názov súboru

- #define subor "c:\\student\\vsuvka.c“
- # include subor
- <=>
- #include "c:\\student\\vsuvka.c"
- vložený súbor môže opäť obsahovať direktívu #include (nie je však povolená rekurzia - ani priama, ani nepriama)

Nastavenie čísla riadku

#line konštanta súbor

- nasledujúcemu riadku priradené číslo "konštanta" a za zdrojový súbor sa považuje "súbor"
- použitie - pri zisťovaní výskytu chýb v programe

Generovanie chyby pri preklade

#error hlásenie

- Zobrazenie chybového hlásenia pri preklade a kompilácia sa považuje za neúspešnú
- Používa sa pre upozornenie na nekorektný preklad

```
#if .....
#error CHYBA: Preklada sa nespravny subor
#endif
```

Direktíva pragma

- Umožňuje zahrnúť do programu direktívy, ktoré sú pre konkrétnu implementáciu prekladača špecifické
- Ak direktíva nie je implementovaná, nedôjde k výskytu chyby - len sa nevykoná
- #pragma message(„Preklad vetvy 1")
  - Vypíše správu počas prekladu
- #pragma once
  - Preklad zdrojového súboru iba raz (hlavičkový súbor)
- #pragma saveregs
  - vkladá sa pred volanie funkcie
  - zabezpečí, že po ukončení volania funkcie budú registre mať rovnakú hodnotu, ako pred volaním funkcie