Y36PJC Programování v jazyce C/C++

Abstraktní třídy, RTTI

Ladislav Vagner

Dnešní přednáška

- Abstraktní třídy.
- RTTI – run-time type info.
- Přetypování v C++.
- Výjimky.

Minulá přednáška

- Dědění.
- Polymorfismus.
- Statická a dynamická vazba.
- Vnitřní reprezentace, VMT.

Abstraktní třídy

- Chceme modelovat výpočty důchodu:
  - evidujeme informace o jménu a datu narození,
  - chceme mít k dispozici rozhraní, které rozhodne, zda dané osobě již vznikl nárok na důchod.
- Nárok na důchod vzniká:
  - pro muže ve věku 65 let, od tohoto věku se odečítá délka vojenské služby,
  - pro ženy ve věku 63 let, od tohoto věku se odečítají 2 roky za každé vychované dítě (neodpovídá realitě, jen příklad).

Abstraktní třídy

- Řešení s abstraktními třídami.
- Třída **`CPerson`**:
  - společné vlastnosti (jméno, rok narození),
  - deklaruje rozhraní (metoda pro výpočet, zda již vznikl nárok na důchod),
  - nedefinuje tuto metodu – abstraktní metoda.
- Třída **`CMan`**:
  - dědí ze třídy **`CPerson`**,
  - navíc informace o vojenské službě,
  - definuje metodu výpočtu nároku na důchod.
- Třída **`CWoman`**:
  - dtto, jen ukládá počet vychovaných dětí.

Abstraktní třídy

```
class CPerson
 {
   protected:
    string name;
    int    born;
   public:
    CPerson ( string _name, int _born ) :
        name (_name), born(_born) { }

    virtual ~CPerson ( void ) { }
    virtual int retired ( int year ) const = 0;
 };
```

Abstraktní třídy

```
class CWoman : public CPerson
 {
   protected:
    int childs;
   public:
    CWoman ( string _name, int _born, int _childs ):
      CPerson ( _name, _born ), childs(_childs) { }
    virtual int retired ( int year ) const
     { return year > born + 63 - 2 * childs; }
 };
```

Abstraktní třídy

```
class CMan : public CPerson
 {
   protected:
    int milSvc;
   public:
    CMan ( string _name, int _born, int _milSvc ) :
      CPerson ( _name, _born ), milSvc(_milSvc) { }
    virtual int retired ( int year ) const
     { return year > born + 65 - milSvc; }
 };
```

Abstraktní třídy

```
CPerson * people [2];
people[0] = new CMan   ( "Novak",  1948, 2 );
people[1] = new CWoman ( "Novakova", 1948, 3 );

for ( i = 0; i < 2; i ++ )
 cout << i << ". "
      << (people[i]->retired ( 2005 ) ? "ano" : "ne")
      << endl;

for ( i = 0; i < 2; i ++ )
 delete people[i];
```

Abstraktní třídy

- Abstraktní třída deklaruje metodu:
  - je dáno rozhraní metody (jméno, parametry, ...),
  - není definované tělo metody,
  - v deklaraci označena `=0`,
  - existuje v předkovi, aby se vyhradil prostor v tabulce virtuálních metod (VMT).
- Těla metod definují potomci.
- Nelze vytvořit instanci abstraktní třídy.
- Abstraktní předek:
  - jednotný pohled na více heterogenních objektů,
  - využití rozhraní vyšší úrovně, netřeba rozlišovat detaily implementace podtříd,
  - uplatnění zejména kolekcích.

Abstraktní třídy

- Abstraktní třídy:
  - nelze vytvořit instanci abstraktní třídy,
  - v programu existují pouze instance neabstraktních tříd - potomků,
  - lze ale pracovat s ukazateli a referencemi typu abstraktní třída.
- Abstraktní metoda musí být **`virtual`**. Proč?

Abstraktní třídy

- Abstraktní metody:
  - lze vytvořit abstraktní instanční metodu,
  - nelze vytvořit abstraktní konstruktor a třídní metodu,
  - abstraktní destruktor vždy povede k chybě. Proč?

RTTI – Run-Time Type Info

- Pracujeme-li s heterogenní kolekcí objektů se společným předkem, můžeme chtít zjistit datový typ instance.
- Příklad s pojištěnci:
  - chceme zjistit, kolik je v databázi mužů (žen).
- Řešení 1:
  - společného předka doplníme o abstraktní metody **`IsMale`** a **`IsFemale`**,
  - podtřídy tyto metody implementují.
- Nevýhoda:
  - těžkopádné pro více podtříd (další podtřída -> metoda ve všech ostatních podtřídách),
  - nepoužitelné pro další rozšíření (např. GUI prvky).

RTTI – Run-Time Type Info

```
class CPerson
 { … virtual int IsMale (void) const = 0;
     virtual int IsFemale (void) const = 0; …
 };

class CMan : public CPerson
 { … virtual int IsMale (void) const { return 1; }
     virtual int IsFemale (void) const { return 0; }
 };

class CWoman : public CPerson
 { … virtual int IsMale (void) const { return 0; }
     virtual int IsFemale (void) const { return 1; }
 };
```

RTTI – Run-Time Type Info

- Řešení 2 - objekt vrací informaci o své třídě:
  - vlastní řešení (např. MFC),
  - systémové řešení - RTTI.
- Operátor **`typeid`**:
  - pro daný objekt vrací referenci na konstantní objekt třídy **`type_info`**,
  - vrácený objekt popisuje třídu dotazovaného objektu.
- Má Java RTTI? Jak je řešeno?

RTTI – Run-Time Type Info

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

CPerson * a = new CMan ( "Novak", 1948, 2 );
CPerson * b = new CWoman ( "Novotna", 1950, 3 );

const type_info & ti = typeid ( *a );
cout << ti . name () << endl;

if ( typeid ( *b ) == typeid ( CWoman ) )
  cout << "b je instance CWoman" << endl;
```

RTTI – Run-Time Type Info

- Přetypování:
  - vždy formální změna typu výrazu (např. **`int`** -> **`double`**),
  - někdy změna hodnoty výrazu (např. zaokrouhlení pro konverzi **`double`** -> **`int`**),
  - různě nebezpečné:
    - **`int`** -> **`double`** asi ok,
    - **`double`** -> **`int`** ztráta přesnosti,
    - **`int`** -> **`char *`** velmi pravděpodobně chyba.
- Standardní konverze – zavedené v systému (např. **`int`** -> **`double`**).
- Vlastní přetypování – lze zavést pro třídy přetížením operátorů přetypování.

RTTI – Run-Time Type Info

- Standardní přetypování:
  `(T) vyraz`
  - není omezené – ať je vztah typů výrazu a `T` jakýkoliv,
  - kompilátor nemůže hlídat záměr programátora a upozornit jej na případné chyby.
- Nově zavedené operátory přetypování:

```
const_cast<T>       ( vyraz )
static_cast<T>      ( vyraz )
dynamic_cast<T>     ( vyraz )
reinterpret_cast<T> ( vyraz )
```

RTTI – Run-Time Type Info

- Přetypování pomocí **`static_cast<T>`**:
  - standardní konverze,
  - přetypování v rámci hierarchie dědičnosti,
  - přetypování tam, kde je přetížen operátor přetypování nebo konstruktor uživatelské konverze,
  - přetypování na **`void/void*`**.
- Chyba překladu je hlášena pro:
  - přetypování se změnou **`const/volatile`**,
  - přetypování mezi ukazateli/referencemi na třídy mimo hierarchii dědění,
  - neportabilní přetypování (např. mezi ukazateli a číselnými typy).

RTTI – Run-Time Type Info

```
class A
 {  … operator int ( void ) { … }  };
class B : public A
 { …  B ( int x ) { … } };
class C : public A { … };

B t1   = static_cast<B> ( 4 );
int i1 = static_cast<int> ( t1 );
i1     = static_cast<int> ( 25.89 );
A * t2 = static_cast<A*> ( &t1 );
B * t3 = static_cast<B*> ( t2 );

const int *iptr = &i1;
int * i3 = static_cast<int *> ( iptr );
C   c;
B * bptr = static_cast<B*> ( &c );
char * d = static_cast<char*> ( &i1 );
```

RTTI – Run-Time Type Info

- Přetypování pomocí **`dynamic_cast<T>`**:
  - použitelné pouze pro přetypování ukazatelů/referencí na objekty s RTTI (**`T`** nebo jeho předek musí mít alespoň jednu virtual metodu),
  - podobná omezení jako **`static_cast`** ,
  - za běhu programu se kontroluje, zda typ přetypovávaného výrazu odpovídá požadovanému typu.
- Pokud přetypování neuspěje:
  - pro ukazatel vrací **`NULL`**,
  - pro referenci způsobí výjimku **`bad_cast`**.

RTTI – Run-Time Type Info

```
class A
 {  … operator int ( void ) { … }  };
class B : public A
 { …  B ( int x ) { … } };
class C : public A { … };

A a,         * aptr = &a, *ta;
B b ( 10 ), * bptr = &b, *tb;

ta = static_cast<A*> ( bptr );
tb = static_cast<B*> ( aptr );
cout << ta << "  " << tb << endl;
  // projde, ale tb je nesmyslne
ta = dynamic_cast<A*> ( bptr );
tb = dynamic_cast<B*> ( aptr );
cout << ta << "  " << tb << endl;
  // tb je NULL
```

RTTI – Run-Time Type Info

- Přetypování pomocí **`const_cast<T>`**:
  - umožní z datového typu odebrat kvalifikátory **`const/volatile`**.
- Přetypování pomocí **`reinterpret_cast<T>`**:
  - umožňuje provádět ostatní přetypování:
    - neportabilní operace,
    - přetypování z důvodu přístupu k paměťové reprezentaci.

RTTI – Run-Time Type Info

```
class A
 {  … operator int ( void ) { … }  };
class B : public A
 { …  B ( int x ) { … } };
class C : public A { … };

int i1 = 10;
const int *iptr = &i1;
int * i3 = const_cast<int *> ( iptr );

char * d = reinterpret_cast<char*> ( &i1 );
 // syntaxe ok, obcas vyuzitelne

C   c;
B * bptr = reinterpret_cast<B*> ( &c );
 // syntaxe ok, pouziti ??
```

Výjimky v C++

- Reakce na chybu za běhu programu:
  - ukončení běhu (!!),
  - výpis chyby, ukončení běhu (!),
  - zápis do logu,
  - ignorování.
- Místo, kde chyba vzniká často není místem, kde se chyba dá ošetřit:
  - chyba vzniká na nízké úrovni (např. chyba čtení z disku),
  - ke správnému ošetření chyby není dostatek informací.
- Šíření informace o chybě:
  - návratové hodnoty funkce (ošetřování !),
  - výjimky.

Výjimky v C++

- Ošetření chyb výpočtu či nestandardního stavu.
- Ukončení výpočtu v hlídaném bloku.
- Vyhledání odpovídajícího ovladače výjimky:
  - existuje – šíření výjimky se zastaví,
  - neexistuje – výjimka se šíří dále směrem k volajícímu,
  - neošetřená výjimka v **`main`** – ukončení programu.
- Hlídaný blok – klíčové slovo **`try`**.
- Vyvolání výjimky – klíčové slovo **`throw`**:
  - parametr – popis výjimky,
  - libovolná hodnota (skalární typ, strukturovaný typ),
  - často objekt s popisem příčiny vzniku.
- Ovladač výjimky – klíčové slovo **`catch`**.
- Neexistuje **`finally`** jako v Javě.

Výjimky v C++

```
int gcd ( int a, int b )
 {
   if ( a <= 0 || b <= 0 ) throw "Invalid arguments";
   while ( a != b )
    if ( a > b ) a -= b; else b -= a;
   return ( a );
 }

int a, b, c;
try {
  cin >> a >> b;
  c = gcd ( a, b );
 }
catch ( const char * Err ) { cout << Err << endl; }
catch ( … ) { /* ostatní vyjimky */ }
```

Výjimky v C++

- Rozlišení výjimek – datový typ.
- Stejná pravidla jako u výběru přetížené funkce.
- Příklady:

```
throw 10;                    catch ( int n )
throw 2.5f;                  catch ( float f )
throw 'a';                   catch ( char c )
throw "Error";               catch ( const char * s )
throw SomeClass ();          catch ( const SomeClass & x )
throw SomeClass ( 10 );      catch ( const SomeClass & x )
throw new SomeClass;         catch ( const SomeClass * x )

class Ancestor { … };
class Descent : public Ancestor { … };
throw Descent ();            catch ( const Ancestor & x )
                             catch ( const Descent & x )
```

Výjimky v C++

- Funkce/metoda může deklarovat, že se z ní mohou šířit výjimky:
  - Může šířit libovolné výjimky:

```
int foo ( void )
```

  - Může šířit pouze výjimky **Exc1** nebo **Exc2**:

```
int foo ( void ) throw (Exc1, Exc2)
```

  - Nemůže šířit žádné výjimky:

```
int foo ( void ) throw ()
```

Výjimky v C++

- Výjimka v konstruktoru:
  - provádění konstruktoru se pozastaví,
  - nová instance není inicializovaná,
  - nevolá se na ni destruktor.
- Výjimka v destruktoru:
  - ukončí se kód destruktoru,
  - provedou se destruktory ostatních lokálních objektů,
  - teprve pak se hledá ovladač výjimky.
- Výjimka v průběhu šíření jiné výjimky:
  - okamžité ukončení programu.

Dotazy...

Děkuji za pozornost.