X36PJC
12. přednáška

Abstraktní třídy. Úvod do šablon. Výjimky.

Minulá přednáška

- Dědění
- Polymorfismus
  - Statická vazba
  - Dynamická vazba
  - Tabulka Virtuálních Metod (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 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

- 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

- 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.

Šablony

- Generické programování:
  - Mechanismus umožňující vytvářet generické funkce/třídy.
  - Polymorfismus mezi třídami, které nemají společného předka.
  - Generický program „instancujeme“ na konkrétní datové typy.

Šablony funkcí

- Chceme vytvořit funkci compare, která vrací informaci který za dvou vstupních objektů je větší. V našem programu chceme porovnávat např. double.

```
int compare(const double &v1, const double &v2) {
        if (v1 < v2) return -1;
        if (v2 < v1) return 1; return 0;
}
```

Šablony funkcí

- Program se nám rozrůstá a najednou zjistíme, že by se nám hodilo stejně porovnávat i stringy.
- Řešení:
  - Přetížíme funkci compare i pro string.
  - Vytvoříme šeblonu funkce pro jakýkoli typ.

Šablony funkcí

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Šablony funkcí

- Šablona funkce slouží k zobecnění použití funcke pro různé datové typy.
- (typy/hodnoty) Šablonových parametrů musí být odvoditelné z parametrů funkce.
- Definice:

  template < typename *i1,* typename *i2, …* >

  definice_funkce

Šablony funkcí

```
int main () {
    // T je int;
    // int compare(const int&, const int&)
    cout << compare(1, 0) << endl;
    // T je string;
    // int compare(const string&, const string&)
     string s1 = "hi", s2 = "world";
    cout << compare(s1, s2) << endl;
    return 0;
}
```

Šablony funkcí

```
// inicializace pole
 <class T, size_t N> void array_init(T (&parm)[N])
   {
        for (size_t i = 0; i != N; ++i) {
                parm[i] = 0;
        }
   }
   int x[42]; double y[10];
   array_init(x); // instantiates array_init(int(&)[42]
   array_init(y); // instantiates array_init(double(&)[10]
```

Šablony tříd

- Využivá se hojně v std pro implementaci kontejnerů (vector, list, queue, ...)
- Šablony umožňují vytvořit šablonu třídy, která není závislá na konkrétním typu.
- Konkrétní třída se vytvoří při překladu jejím instancováním pro konkrétní datový typ.

Šablony tříd

```
template <class Type> class Queue {
public:
    Queue (); // default constructor
    Type &front (); // return element from head of Queue
    const Type &front () const;
    void push (const Type &); // add element to back of
    Queue
    void pop(); // remove element from head of Queue
    bool empty() const; // true if no elements in the Queue
private: // ...
};
```

Duck typing vs. Generic programming

- Vyhodnocováno při běhu programu.
- Pracuje na principu zpráv. Objekt buď zprávě rozumí nebo ne.
- Kód se provádí ve VM.
- Chyby v programu musí odchytit programátor důkladným testováním.

- Vyhodnocováno při překladu.
- Pro každý datový typ nový překlad funkce/třídy.
- Překládáno přímo do binárního kódu.
- Většinu chyb odchytí překladač.
- Staticky typované.

Literatura

- C++ Primer (4th Edition) by Stanley B. Lippman, kapitola 15 - 16
- C++ Primer Plus (5th Edition) by Stephen Prata
- Slajdy Ladislava Vagnera

Děkuji Vám za pozornost