Zpracoval: houzvjir@fel.cvut.cz

10. Objektově orientované programování v C++. Přetěţování operátorů, generické funkce a třídy, výjimky, knihovny. (A7B36PJC)

Obsah

OOP v C++

Třída

Třída je předpis, jak vyrobit objekt daného typu. Definuje atributy a metody.

- Reprezentace modelované skutečnosti v programu
- Soustředí související údaje do jednoho celku
- Uzavření před vnějším světem (omezení přístupu)
- Rozhraní pro práci (metody)
- Využitelné pro polymorfismus a dědění

Objekt

Konkrétní objekt, jednoznačně identifikovatelný. Je vždy instancí nějaké (právě jedné) třídy. Třída má typicky mnoho instancí.

Instanci je možno alokovat staticky:

```
int main(int argc, char ** argv){
  // staticka alokace instance tridy PrikladTridy (objekt t)
  PrikladTridy t;
}
```

- vytváří se před provedením programu
- po provedení **automaticky** zanikne

Nebo ji lze alokovat dynamicky:

- vytváří se za běhu programu
- klíčové slovo new pro alokaci
- vytváří ukazatel na instanci
- nemusí uspět (výjímka std::bad_alloc)
- **nezaniká automaticky** - paměť se musí uvolnit pomocí klíčového slova delete (nebo delete[] pro pole)

```
int main(int argc, char ** argv){
  //dynamicka alokace ukazatele na instanci tridy PrikladTridy (objekt t)
  PrikladTridy * t = new PrikladTridy;
  //dynamicka alokace ukazatele na pole instanci PrikladTridy
  PrikladTridy * t2 = new PrikladTridy[3];
  //dealokace pameti po objektu t
  delete t;
  //dealokace pameti po poli t2
  delete[] t2;
}
```

Pro připomenutí - Základní ideje OOP

Viz též otázka č. 9.

Dědičnost

Třída může být „poděděná“ od jiné třídy, což znamená, že má všechny členy (atributy, metody,..) tříd, ze kterých dědí (předků). Může přidat nové, případně upravit existující (z této možnost vychází koncept polymorfismu). V programovacích jazycích může být podporována i několikanásobná dědičnost, umožňující třídě být potomkem několika tříd (které mezi sebou nemají žádný vztah) najednou. **C++ podporuje vícenásobné dědění.**

Abstrakce

Díky dědičnosti je možné pracovat s třídami na libovolné vhodné úrovní ve stromu jejich předků (např. máme pole zvířat Zvire[] - nezajímá mě jakých - ale každý prvek pole může být nějaký (jiný) potomek třídy Zvire).

Zapouzdření

Třídy nám umožňuji nevidět konkrétní detaily fungování třídy (metoda Stekni() může využívat jiné metody atd.)

Polymorfismus

OOP nám umožňuje pracovat s potomkem třídy X jako se třídou X samotnou, ale potomek může provádět při práci s jeho členy akce, které mu odpovídají (např. třída Zvire ma metodu DelejZvuk(), kterou má Slon implementovanou jako troubení a Vlk jako vytí. A když voláme u nějakého objektu, se kterým pracujeme jako s třídou Zvire, metodu DelejZvuk(), díky polymorfismu se ozve správny zvuk - troubení, vytí atd..)

Přetěžování operátorů

Standardní operátory jsou přetížené. Např: 10 + 1 (int + int), 1.5 + 2.4 (float + float)…

V C++ lze operátory přetěžovat pro jiné typy parametrů, ALE:

- nelze zavést nové operátory,
- nelze přetěžovat operátory ., .*, ::, ?:, ::*, sizeof(), typeid() (seznam oddělený čárkami)
- nelze měnit aritu operátorů (počet operandů, se kterými se daná operace provádí),
- nelze měnit prioritu operátorů

Ostatní operátory při dodržení těchto pravidel (včetně [] a ()) přetěžovat lze.

Přetěžovat lze pomocí metod a funkcí. Klíčové slovo **operator**.

Přehled přetěžování operátorů:

- funkcemi,
- metodami,
- klíčové slovo friend,
- přetěžování operátorů =, << a >>
- přetěžování dalších operátorů (++, --, [], ()).

Přetěžování operátorů metodami

Syntaxe:

```
navratovy_typ jmeno_tridy::operator meneny_operator (parametry)
```

Výhody:

- lze pracovat s privátními proměnými

Nevýhody:

- jako 1. parametr vždy **this**, takže omezená použitelnost

Třída, jejíž 2 metody přetěžují operátory (binarni + a unarni -):

- Přetížení operátoru funkcí:
  - nelze pro všechny operátory,
  - problémy s přístupem ke členským proměnným objektů.
- Řešení – přetížení metodou:
  - přístup ke členským proměnným,
  - použitelné pro všechny operátory, které lze přetížit.
- Realizace:
  - jméno metody – operator …,
  - parametry – o jeden méně, než je arita operátoru,
  - levý operand – instance nad kterou je metoda spuštěna.

```
class CCplx //trida CCplx
{
   double re, im;
   public:
      CCplx ( double r, double i=0 ) : re(r), im(i) {}

      //pretizeni unarniho operatoru -, jako parametr se bere volajici
objekt (this)
      CCplx operator - ( void ) const;

      //pretizeni binarniho operatoru +, jako 1. parametr se bere volajici
objekt
      CCplx operator + ( const CCplx & x ) const;
};
```

Deklarace metod přetěžujících operátory:

```
CCplx CCplx::operator - ( void ) const
{
   return ( CCplx ( -re, -im ) );
}

CCplx CCplx::operator + ( const CCplx & x ) const
{
   return ( CCplx ( re + x.re, im + x.im ) );
}
```

Přetěžování operátorů funkcemi

Syntaxe:

```
navratovy_typ operator meneny_operator (parametry)
```

Výhody:

- 1. parametr může být jakýkoliv (primitivni datový typ…)

Nevýhody:

- obtížný přístup k privátním proměným

```
struct TCplx
{
double re, im;
};
TCplx operator + ( TCplx a, TCplx b )
{
TCplx c;
c . re = a . re + b . re;
c . im = a . im + b . im;
return c;
}
TCplx x = { 1, 0 }, y = { 2, 1 }, z;
z = x + y;
z = operator + ( x, y ); // jiny zapis volani op.
```

- Operátor + přetížený v příkladu nebude použit pro:

```
TCplx a, b;
a = b + 3;
a = 4 + b;
```

- Řešení 1:
  - přetížit operátor + i pro parametry typu double,
  - pro každý operátor musíme přetěžovat 3 varianty.

```
TCplx operator+ ( TCplx a, double b )
{ … }
TCplx operator+ ( double a, TCplx b )
{ … }
```

- Řešení 2:
  - využít konstruktor uživatelské konverze z typu double na typ TCplx,
  - ponechat pouze jeden přetížený operátor +.

```
TCplx::TCplx ( double a )
{
re = a;
im = 0;
}
TCplx a, b;
a = 4 + b; // a = TCplx ( 4 ) + b;
```

- Parametry funkce přetíženého operátoru:

```
TCplx operator+ ( TCplx a, TCplx b )
{ … } // ok, ale pomale pro velke instance
TCplx operator+ ( TCplx & a, TCplx & b )
{ … } // !!! mj. nebude asociativni.Proc?
TCplx operator+ ( const TCplx & a,
const TCplx & b )
{ … } // ok
```

- Návratový typ funkce přetíženého operátoru:

```
TCplx & operator+ ( TCplx a, TCplx b )
{
TCplx res;
res . re = a . re + b . re;
res . im = a . im + b . im;
return res; //!! chyba - reference na lok. prom.
}
TCplx & operator+ ( TCplx a, TCplx b )
{
static TCplx res;
res . re = a . re + b . re;
res . im = a . im + b . im;
return res; //!! chyba – asociativita, thready
}
```

Jinak:

```
TCplx & operator+ ( TCplx a, TCplx b )
{
TCplx * res = new TCplx;
res . re = a . re + b . re;
res . im = a . im + b . im;
return * res; //!! chyba – nelze smazat
```

```
}
TCplx operator+ ( TCplx a, TCplx b )
{
TCplx res;
res . re = a . re + b . re;
res . im = a . im + b . im;
return res; // ok
}
```

Přetěžování operátorů – časté chyby

- Modifikace parametrů:
  - standardní operátory s výjimkou těch, které mají vedlejší efekt, nemění své parametry.
  - první parametr mění operátory přiřazení (+=, -=, …) a operátory inkrementu/dekrementu (++, --),
  - tomu by mělo odpovídat použití const v parametrech,
- pokud Vámi přetížený operátor bude měnit operandy, bude kód pro ostatní programátory nečitelný a nepochopitelný (problémy s údržbou, předáním kódu, …).

Unární operátory (např. mínus):

```
TCplx & operator- ( TCplx & a )
{
a . re = - a . re; // !! meni parametr
a . im = - a . im;
return ( a );
}
```

Unární mínus správně:

```
TCplx operator- ( const TCplx & a )
{
TCplx res;
res . re = - a . re;
res . im = - a . im;
return ( res );
}
```

Přetěžování operátorů – friend

- Přetížit operátor+ pro (double,const CCplx&):
  - nelze metodou (int není třída),
  - musí se přetížit funkcí.
- Problém s funkcí – přístup ke členským proměnným třídy CCplx:
  - zviditelnit je public (nevhodné),
  - přístup pomocí čtecích metod (getter) – zdržení,
  - delegovat na tuto funkci právo přístupu ke členským proměnným (friend funkce).

```
class CCplx
{
 double re, im;
 public:
 CCplx ( double r, double i=0 ) : re(r), im(i) {}
 CCplx operator - ( void ) const;
 CCplx operator + ( const CCplx & x ) const;
 friend CCplx operator + ( double a,
 const CCplx & b );
};

CCplx operator + ( double a, const CCplx & x )
{ // binarni plus funkci – 2 parametry
 return ( CCplx ( a + x. re, x . im ) );
}
```

- friend dává práva přístupu:
  - funkci,
  - jiné třídě.
- Označená funkce/třída má stejná práva přístupu jako metody.
- Neexistuje friend metoda, existuje pouze frond funkce.
- Neplýtvejte friend.

```
class CCplx
{
double re, im;
public:
CCplx ( double r, double i=0 ) : re(r), im(i) {}
CCplx operator - ( void ) const;
friend CCplx operator + ( const CCplx & x );
// funkce – unarni operator +
};

CCplx CCplx::operator + ( const CCplx & x )
{ // takova metoda ve tride CCplx není deklarovana
return ( CCplx ( re + x. re, im + x . im ) );
}
```

Generické funkce a třídy (templates)

Šablony

Šablony jsou konstrukce funkcí (tříd, struktur, unií), které umožňují vytvářet funkce (…) na základě parametrů. Odpadne tak vytváření mnoha funkcí (…) pro různé typy.

Syntaxe:

```
template <parametry šablony> deklarace
```

- Klíčové slovo **template**.
- Parametry:
  - typové **class** a **typename**.
  - hodnotové - int,… - konstanta nebo výraz, který lze vyhodnotit při překladu.
- Deklarace.

Generické funkce (šablony funkcí)

Vysvětlení na příkladu: Swap je funkce, která vymění („přehodí“) obsah dvou proměnných. Představme si situaci, kdy chceme mít možnost vyměnit hodnoty proměnných mnoha typů, nebo i instancí mnoha tříd. Bez použití šablon bychom museli napsat mnoho funkcí swap, které by se od sebe lišili jen typem parametrů. Lepší řešení je vytvořit jednu šablonu funkce swap, podle které se dle potřeby vytvoří požadovaná funkce na základě parametru, kterým bude typ.

```
#include<iostream>
using namespace std;
template<class T> void sw(T &a, T &b){ //sw ma jako parametr 2x typ T
  T c(a); //prehodi a za b pomoci pomocne promene
  a = b;
  b = c;
}
int main()
{
  int a(0),b(100);
  sw<int>(a,b); //pretypovani funkce sw na int
  cout << a << " " << b << endl;
  float c(1.2),d(2.6);
  sw<float>(c,d); //pretypovani funkce sw na float
  cout << c << " " << d << endl;
  return 0;
}
```

Přetížená funkce sw, která má navíc hodnotový parametr int N a slouží k přehození polí:

```
template<class T, int N> void sw(T *a, T *b){
  T temp;
  for (register int p = 0; p < N; p++, a++, b++){
    temp = *a;
    *a = *b;
    *b = temp;
  }
}
```

Priorita volání v případě existence šablony i „obyčejné“ funkce:

1. Existuje-li pro daný typ parametrů funkce, přeloží se volání této funkce.
2. Neexistuje-li pro daný typ parametrů funkce, překladač zjistí, jestli nemůže vytvořit instanci šablony.
3. Není-li k dispozici ani funkce, ani šablona, ze které lze vytvořit potřebnou instanci, jedná se o chybu.

Generické třídy (šablony tříd)

Slouží pro specifikování šablony jedné třídy. Template hlavičku definující parametry šablony je nutné dávat jak před deklarací třídy, tak i před implementace jednotlivých metod třídy. Navíc musí být generické třídy deklarovány a implementovány v jednom souboru - nemohou být rozdělený do .h a .cpp souborů jako je u tříd zvykem.

Při použití šablony třídy uvádíme parametry explicitně vždy, protože kompilátor nemá z čeho je poznat.

```
template <class T, int jeOtevrena> class Garaz { //Pro ilustraci je zde
druhý parametr. 1. je typový, 2. hodnotový
private:
  T* parkovaciMisto;
public:
  void zaparkuj(T* car);
  T* vyparkuj();
};
template <class T, int jeOtevrena> T* Garaz<T, jeOtevrena>::vyparkuj(){ //
deklarace metody vyparkuj
  T* temp = parkovaciMisto;
  parkovaciMisto = 0;
  return temp;
}
template <class T, int jeOtevrena> void Garaz<T, jeOtevrena>::zaparkuj(T*
car){ // deklarace metody zaparkuj
  if(jeOtevrena == 0) return;
  parkovaciMisto = car;
}
```

Implicitní a explicitní šablony

Šablony lze vytvářet **implicitní** (instance se vytváří, až když jsou potřeba), nebo **explicitní** (vytváří se při překladu).

Implicitní deklarace, třída X slouží pouze k demonstraci, proto nejsou metody f() a g() deklarovany:

```
template<class T> class X {
  public:
    X* p;
    void f();
    void g();
};
```

V komentářích je naznačeno, kdy se vytvoří instance a kdy se vytvořit nemusí (záleží na překladači):

```
X<int>* q; // pouze pointer na tridu X<int>, samotna instance se vytvaret
nemusi
X<int> r; // musi se vytvorit instance tridy x<int> (vytvari se objekt)
X<float>* s; // opet pouze pointer, instance X<float> se nemusi vytvaret
r.f(); // v tuto chvili je potreba vytvorit instanci X<int>::f(), objekt r
ji zacne pouzivat
s->g(); // v tuto chvili  je potreba vytvorit instanci X<float> a
X<float>::g()
```

Explicitní deklarace - pokud chceme, aby se instance vytvářely při překladu a ne až po zavolání např. pro typ <int>, přidáme do předchozího programu řádek:

```
template class X<int>;
```

Tím zajistíme, že při práci s třídou X<int> bude instance vytvářena již při překladu.

Specializace šablon a výchozí parametry

Specializace

V některých je třeba pro konkrétní hodnoty parametrů šablonu upravit - změnit jak se bude při využití s daným parametrem šablony chovat.

Porovnání dvou hodnot a vrácení maxima:

```
#include <iostream>
using namespace std;
template <class T> T max1(T a, T b){
        return a > b ? a : b ;
}
int main(int argc, char ** argv){
    /* Vytvoří se funkce char* max1(char* a, char* b) a bude porovnávat
adresy obou stringu!
     * (ale my chceme porovnat řetězce na těch adresách)
     */
    cout << "max(\"Aladdin\", \"Jasmine\") = " << max1("Aladdin",
"Jasmine") << endl ;
}
```

Specializace pro typ char*:

```
template <> char* max1(char* a, char* b){
  return strcmp(a, b) > 0 ? a : b ;
}
```

Výchozí parametry

Tak jako pro parametry funkcí, je i pro parametry šablon možné zadat výchozí hodnoty parametru. Je tedy možné pak třídě předat méně parametrů, nebo i žádný.

Hlavička šablony třídy Garage by mohla vypadat např. takto:

```
template <class T = Porsche, int jeOtevrena = 1> class Garaz
```

Výjímky

V C++ se často používá systém návratových hodnot různých operací, nejčastěji typu int. Zpravidla hodnota -1 značí chybu.

```
int a = operace();
if(a == -1){
  cerr << "Nastala chyba!" << endl;
}
```

Pokud máme operací mnoho, vznikne nepřehledný sled if podmínek. Proto se používají výjímky - hlídá se celý úsek kódu pomocí klíčového slova **try**, za nímž se odchytávají výjímky pomocí klíčového slova **catch**. Výjímky se tvoří pomocí klíčového slova **throw**.

(Pozn. Vojtech Kral: za throw se dá vyhodit cokoliv, řetězec, číslo, třída)

Třída Vyjimka, kterou budeme používat jako výjímku, má privátní proměnou duvod a pro ni setter a getter:

```
class Vyjimka
{ // Trida vyjimek
  private:
    string duvod;
  public:
    void nastav(string d){duvod = d;}
    string dej(){return duvod;}
};
```

Třída zlomek s čitatelem, jmenovatelem a metodou vyděl, která háže výjímku v momentě, kdy je jmenovatel == 0.

```
class Zlomek{
  private:
    int C,J;
  public:
    void nastavCitatel(int c) { C = c;}
    void nastavJmenovatel(int j) {J = j;}
    double vydel() throw (Vyjimka); // Z metody vydel muze byt vyvrzena
vyjimka - instance tridy Vyjimka
};
double Zlomek::vydel() throw (Vyjimka){ // začátek bloku 2
  int *i = new int;
  try{
    if (J == 0){ //začátek bloku 1
      string s("Nejde");
      Vyjimka v;
      v.nastav(s);
      throw v;
    } // konec bloku 1
    delete i;
  }
  catch (Vyjimka v) { // odchytnuti vyjimky v metode pro dealokaci pameti
    delete i;
    throw; // V tomto případě má stejný význam jako throw v;
  }
  return ((double)C / J);
} // konec bloku 2
```

Main s demonstrací odchytávání výjímek:

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

//pretizeni operatoru << pro vypis vyjimek
ostream &operator<< (ostream &os, Vyjimka &v){
        return os << v.dej() << endl;
}
int main(int argc, char ** argv){
  Zlomek z1,z2;
  z1.nastavCitatel(10);
  z2.nastavCitatel(5);
  for(int i = 5; i > -5; i--){
    z1.nastavJmenovatel(i);
    z2.nastavJmenovatel(i);
    try{ // hlidany usek
      cout << "10 / " << i << " = " << z1.vydel() << endl;
      cout << "5 / " << i << " = " << z2.vydel() << endl;
    }
    catch (Vyjimka v){ // zde je odchyceni vyjimky tridy Vyjimka
      cout << v << endl;
    }
    /* zde muze byt dalsi catch, podle toho, kolik vyjimek odchytavame
    catch (Jina_vyjimka j){
      .....
    }
    */
  }
  return 0;
}
```

Z příkladu je vidět, že u metody vydel() je povolena vyjímka **throw (Vyjimka)**. Všechny výjímky, které se můžou vytvořit při zpracování kódu metody **musí být zahrnuty v tomto seznamu**, jinak se zavolá funkce unexpected a následně terminate. Podobně **pokud není vytvořená výjímka nikde odchycena, ani ve funkci main()**, program zavolá funkci terminated a následně aborted a skončí.

Pokud chceme odchytit jakoukoliv výjímku, píšeme **catch(…)**. Výjímky jsou třídy a lze je tedy **dědit** a vytvářet tak konkrétnější výjímky z abstraktního základu.

Výjímky v STD C++

Ač se používají spíše návratové hodnoty, C++ má několik vlastních výjímek v STD (co je STD viz. Knihovny) v sekci Language Support, <exception>, například výjímku **bad_alloc**, kterou háže např. operátor new, pokud se mu nepodaří zabrat paměť.

Knihovny