Virtuální funkce

aneb třída jako šablona

- Chceme vytvořit vektorové malovátko, které kreslí různé tvary,
- každý tvar se kreslí jinak, ale všechny chceme dát do spojáku.
- Spoják zajistí třída `nakres`(litelny) a chce také vynutit funkci `sezame_nakresli`.
- `class nakres{ public: void sezame_nakresli();};`
- `class bod:public nakres{ public: void sezame_nakresli();};`
- `nakres*hlava=new bod();...`
- ... a celé to nebude fungovat, protože metody jsou reprezentovány v prototypu.
- Virtuální funkce jsou reprezentovány ve VMT.
- Tvoří se stejně jako v Pascalu:
  `virtual navr_typ jmeno(parametry)...`

Virtuální funkce

v příkladu malovátko

```
#include <stdio.h>
class nakres
{    public: virtual void sezame_nakresli()
     {    printf("Nic nekreslim!\n"); }
};
class bod:public nakres
{    public: virtual void sezame_nakresli()
     {    printf("Kreslim bod!\n");}
};
int main()
{    nakres*hlava=new bod();
     hlava->sezame_nakresli();
}
```

Statické prvky

sedí ve třídě a ne v objektech

- Někdy chceme, aby atribut (nebo metoda) existovala jen jednou.
- Takové prvky můžeme definovat, aby příslušely přímo třídě...
- ... zvány statické, definují se modifikátorem `static`. Ten se chová (syntakticky) stejně jako `virtual`, tedy:
- statická proměnná:

```
class t{ public: t()
     {    static int citac1=0; citac1++;}
}
```

Statické prvky

- Statický atribut třídy musíme explicitně definovat (nedefinuje se deklarací statického atributu):

```
class t{ static int citac1;
public: t(){t::citac1++}}
int t::citac1;
```

- statická metoda:

```
class t{...
      static int kolikrat()
      {    return citac1;
      }
};
...
printf("Konstruovali jsme %d",t::kolikrat());}
```

Friend-funkce

aneb placený zabiják v roli sheriffa

- Občas chceme, aby funkce nepříslušná třídě měla přístup k privátním atributům.
- V tom případě funkci definujeme mimo třídu – např.
  ```
  void kamarad(t a)
  {...}
  ```
  a ve třídě jen deklarujeme, že je zpřátelená:
- `friend void kamarad(t);`
- Friend-funkci využijeme především předáváme-li jako parametr svou vlastní třídu a zpřátelená funkce jí přistupuje k privátním atributům.

Čistě virtuální funkce

jsou funkce, které chceme definovat až v synovských třídách

- Motivace: Chceme napsat ovladač od tiskárny, ale každá tiskárna je jiná.
- Tiskárna obecná neexistuje (existují jehličkové, inkoustové, laserové,...) a každá tiskne jinak.
- Chceme rozhraní, které definuje společné požadavky - například metodu `tiskni`.
- Použijeme dědičnost a metoda `tiskni` bude virtuální.
- V rodiči ji nechceme definovat, tak do ní přiřadíme nulu: virtual int tiskni(char*)=0;
- Funkci definujeme až v synech.
- Takové funkci říkáme *čistě virtuální.*

Příklad

s tiskárnami

```
class tisk
{    public: virtual int tiskni(char*)=NULL;
};
class ltisk
{    public: virtual int tiskni(char*co)
     {    printf("Tisknu laserove: %s\n",co);}
}
class itisk
{    public: virtual int tiskni(char*co)
     {    printf("Tisknu inkoustem: %s\n",co);}
}
...
```

Abstraktní třída

byla na předchozím slildu

- Třída, která obsahuje aspoň jednu čistě virtuální funkci, se nazývá abstraktní.
- Nelze vytvořit instanci této třídy,
- že je třída abstraktní, překladač pozná.
- Takové třídy můžeme používat jako tzv. interface,
- tedy řekneme, co musí každá synovská třída definovat.
- Praktické použití: často omílané vektorové malovátko (a typ `drawable`).

Přetěžování operátorů

dělá něco podobného jako přetěžování funkcí

- Chceme implementovat racionální čísla, uděláme tedy třídu s čitatelem a jmenovatelem (a vybavíme metodou `toString`).

Přetěžování operátorů

dělá něco podobného jako přetěžování funkcí

- Chceme implementovat racionální čísla, uděláme tedy třídu s čitatelem a jmenovatelem (a vybavíme metodou `toString`).
- Čísla chceme sčítat, odčítat,... jako jiné číselné typy (ne `secti(rac a, rac b);`).

Přetěžování operátorů

dělá něco podobného jako přetěžování funkcí

- Chceme implementovat racionální čísla, uděláme tedy třídu s čitatelem a jmenovatelem (a vybavíme metodou `toString`).
- Čísla chceme sčítat, odčítat,... jako jiné číselné typy (ne `secti(rac a, rac b);`).
- Přetížený operátor definujeme jako funkci s divným jménem, např. `operator=`, `operator+`,...

Přetěžování operátorů

dělá něco podobného jako přetěžování funkcí

- Chceme implementovat racionální čísla, uděláme tedy třídu s čitatelem a jmenovatelem (a vybavíme metodou `toString`).
- Čísla chceme sčítat, odčítat,... jako jiné číselné typy (ne `secti(rac a, rac b);`).
- Přetížený operátor definujeme jako funkci s divným jménem, např. `operator=, operator+,...`
- Příklad:

```
rac& rac::operator=(rac A)
{    this->re=A.re;this->im=A.im;
     return A;
}
rac operator+(const rac B)
{    rac C;
     C.re=this->re+B.re;
```

Proudy

a využití přetížených operátorů

- Načítáme pomocí `getchar` a vypisujeme pomocí `printf`,

Proudy

a využití přetížených operátorů

- Načítáme pomocí `getchar` a vypisujeme pomocí `printf`,
- v C++ je možnost použít streamy (otevřít soubor jako stream)

Proudy

a využití přetížených operátorů

- Načítáme pomocí `getchar` a vypisujeme pomocí `printf`,
- v C++ je možnost použít streamy (otevřít soubor jako stream)
- Příklad:

```
ofstream vystupni;
vystupni.open("vystup.txt");
...
```

Proudy

a využití přetížených operátorů

- Načítáme pomocí `getchar` a vypisujeme pomocí `printf`,
- v C++ je možnost použít streamy (otevřít soubor jako stream)
- Příklad:

```
ofstream vystupni;
vystupni.open("vystup.txt");
...
```

- Standardní vstup/výstup reprezentován streamy `cin, cout`.

Proudy

a využití přetížených operátorů

- Načítáme pomocí `getchar` a vypisujeme pomocí `printf`,
- v C++ je možnost použít streamy (otevřít soubor jako stream)
- Příklad:

```
ofstream vystupni;
vystupni.open("vystup.txt");
...
```

- Standardní vstup/výstup reprezentován streamy `cin, cout`.
- Soubor je pak reprezentován objektem (kterému lze volat metody).

Proudy II

a teď už opravdu ty přetížené operatory

- Načítání a výpis:

```
char data[100];
cin>>data;
cout<<data<<endl;
```

- Ovšem << resp. >> známe, byly to operátory bitového posunu.
- Tady jsou tyto operátory přetížené.

Výjimky

aneb když se něco nepovede

- Když se něco nepovede, program může buďto drasticky zastavit,
- nebo chybu ignorovat,
- nebo funkce místo výsledku vrátí chybový kód...
- a tím přichází o obor hodnot.
- Výjimky reprezentují možnost dát vědět, že je problém.

Výjimky

pravidla

- klíčová slova `try`, `catch` a `throw`.
- `try` uvádí blok, ve kterém mohou padat výjimky,
- `catch` uvádí blok, kde se má výjimka daného typu ošetřit,
- `throw` hází (vyvolává) výjimky syntakticky jako `return`.

Příklad

výjimky

```
int main()
{    int a=nacti(),b=nacti();
     try
     {    if(b==0) throw 0;
          printf("%d\n",a/b);
     }
     catch(int a){printf("NELZE\n");}
     return 0;
}
```

Objektová výjimka

aneb házíme objektem

```
class mavyjimka
{    char*cosestalo;
     public: mavyjimka(char*cosestalo)
     {    this->cosestalo=cosestalo;}
}
...
throw mavyjimka("prase kozu potrkalo");
... catch(const mavyjimka & a)
{    printf("%s\n",a.cosestalo);
...}
```

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.
- Vypadneme-li dříve z programu, výjimku zachytí defaultní handler výjimek (a program ukončí).

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.
- Vypadneme-li dříve z programu, výjimku zachytí defaultní handler výjimek (a program ukončí).
- `catch`-bloků může jít více za sebou, výjimku vyřídí první nalezený blok.

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.
- Vypadneme-li dříve z programu, výjimku zachytí defaultní handler výjimek (a program ukončí).
- `catch`-bloků může jít více za sebou, výjimku vyřídí první nalezený blok.
- Pozor, pokud máte handler výjimky synovské za handlerem výjimky rodičovské!

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.
- Vypadneme-li dříve z programu, výjimku zachytí defaultní handler výjimek (a program ukončí).
- `catch`-bloků může jít více za sebou, výjimku vyřídí první nalezený blok.
- Pozor, pokud máte handler výjimky synovské za handlerem výjimky rodičovské!
- `catch(...){.....}` – mělo by zachytit všechny výjimky, jenže...

Výjimky

fungování a podrobnosti

- Výjimka propadá programem, dokud se nenajde odpovídající `catch`-blok.
- Vypadneme-li dříve z programu, výjimku zachytí defaultní handler výjimek (a program ukončí).
- `catch`-bloků může jít více za sebou, výjimku vyřídí první nalezený blok.
- Pozor, pokud máte handler výjimky synovské za handlerem výjimky rodičovské!
- `catch(...){.....}` – mělo by zachytit všechny výjimky, jenže...
- některé výjimky jsou (aspoň v některých prostředích) nezachytitelné (např. `5/0`).

Namespacy a jejich využití

- Při programování se nám jména identifikátorů divně rozlézají,
- `insertovat` má například smysl do lecčeho a ne pokaždé můžeme chtít pod tímto aliasem vidět jinou funkci.
- Proto můžeme části kódu uzavřít do jmenného prostoru (`namespace`).
- Prvky namespacu definujeme poblíž sebe syntakticky podobně jako třídu:
- `namespace jmeno{ prvky...}`
- Přistupujeme buďto operátorem čtyřtečky, nebo řekneme `using namespace jmeno`
- Často se používá předdefinovaný C++ový prostor `std`.

Příklad

jmenných prostorů

```
namespace spojak
{    pridej(int co){...}//pridej do spojaku
     int uber(){...}//ze spojaku
}
namespace btree
{    pridej(int co){...}//pridej do b-stromu
     int uber(){...}//z b-stromu
}
... btree::pridej(spojak::uber());
```

Šablony

a to bude téměř to poslední, co ode mě o jazycích uslyšíte

- Opět chceme implementovat funkce (a třídy) pro různé datové typy,
- nechceme používat makra (protože se jich bojíme).
- Šablona nám umožňuje vytvořit parametrizovanou třídu nebo funkci.
- Za parametr dosazujeme.
- Definujeme pomocí klíčového slova `template`, do špičatých závorek popíšeme parametry, se kterými pak pracujeme.
- Následuje definice funkce nebo třídy (podle toho, jestli je to šablona na třídu nebo na funkci).

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...
- typ v tomto případě může být (krom již známých) také `class` nebo `typename` (anebo další šablona).

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...
- typ v tomto případě může být (krom již známých) také `class` nebo `typename` (anebo další šablona).
- Můžeme například definovat fukci sčítající předem neznámé typy (jako u maker).

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...
- typ v tomto případě může být (krom již známých) také `class` nebo `typename` (anebo další šablona).
- Můžeme například definovat fukci sčítající předem neznámé typy (jako u maker).
- šablona v C++ se vygeneruje při kompilaci jedna pro každou hodnotu parametrů, takže použijeme-li šablonu pro sčítání intů, doublů a charů, vygenerují se nám z šablony tři funkce.

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...
- typ v tomto případě může být (krom již známých) také `class` nebo `typename` (anebo další šablona).
- Můžeme například definovat fukci sčítající předem neznámé typy (jako u maker).
- šablona v C++ se vygeneruje při kompilaci jedna pro každou hodnotu parametrů, takže použijeme-li šablonu pro sčítání intů, doublů a charů, vygenerují se nám z šablony tři funkce.
- Již v době kompilace se zkontroluje, zda s parametry lze dělat to, co chceme (například sčítat).

Slovo `template`

aneb co píšeme do špičatých závorek

- Do špičatých závorek popisujeme parametry šablony, jednotlivé parametry oddělujeme čárkou,
- každý parametr je ve formátu `typ název`, tedy to vypadá jako definice parametrů funkce, jenže...
- typ v tomto případě může být (krom již známých) také `class` nebo `typename` (anebo další šablona).
- Můžeme například definovat fukci sčítající předem neznámé typy (jako u maker).
- šablona v C++ se vygeneruje při kompilaci jedna pro každou hodnotu parametrů, takže použijeme-li šablonu pro sčítání intů, doublů a charů, vygenerují se nám z šablony tři funkce.
- Již v době kompilace se zkontroluje, zda s parametry lze dělat to, co chceme (například sčítat).
- Povšimněte si později rozdílu oproti generikům v C#

Příklad

šablony na funkci a šablon na třídu

```
template <typename T> T secti(T a, T b)
{    return a+b; }
template<int I,typename T> class uloziste
{    T sklad[I];//mame pole I prvku typu T.
}
uloziste<10,int> sklad;
printf("%d\n",secti<int>(1,2));
secti(1,2);//nastoupi dedukce typu!
```