

# C/C++ III. Pokročilý

část 1.

IT Academy, 2014

# Obsah

## **1. Pamäťové modely a menné priestory**

1. Oddelený preklad
2. Menné priestory

## **2. Dedičnosť tried**

1. Začneme jednoduchou základnou triedou
2. Dedičnosť
3. Polymorfná verejná dedičnosť
4. Riadenie prístupu: protected
5. Dedičnosť a dynamické pridelovanie pamäte

## **3. Opakované používanie kódu v C++**

1. Triedy, ktorých členy sú objekty
2. Súkromná dedičnosť
3. Viacnásobná dedičnosť
4. Šablóny tried

# 1. Pamäťové modely a menné priestory

## 1. Oddelený preklad

1. Doba trvania úložiska, rozsah platnosti a väzba
2. Rozsah platnosti a väzba
3. Automatická doba trvania úložiska
4. Premenné so statickou dobou trvania

## 2. Menné priestory

1. Tradičné menné priestory v jazyku C++
2. Nové vlastnosti menných priestorov
3. Príklad menných priestorov
4. Menné priestory v budúcnosti

# 1. Pamäťové modely a menné priestory

## **Odporúčaná literatúra**


Začiatok témy - strana 321

Opakovacie otázky - strana 349

Programovacie cvičenia - strana 350

# 1. Pamäťové modely a menné priestory

## Oddelený preklad

- viacero súborov, ako C, ktoré sa kompilujú samostatne
- **make** (viz snake **Makefile**)
- **#include**  ""
- hlavičkový súbor **.h**

### **koncovky súborov:**

<http://stackoverflow.com/questions/5171502/c-vs-cc-vs-cpp-vs-hpp-vs-h-vs-cxx>

# 1. Pamäťové modely a menné priestory

## Oddelený preklad

Program sa teda delí do:

- **hlavičkových súborov**, ktorý deklaruje štruktúry a prototypy funkcií, ktoré s týmito štruktúrami pracujú
- **súborov obsahujúcich zdrojové kódy týchto funkcií**
- **súbor programu, ktorý tieto funkcie volá**

# 1. Pamäťové modely a menné priestory

## Oddelený preklad

### Hlavičkové súbory .h .hpp:

**Nevkladáme definície funkcií do hlavičkových súborov!**

**Umiestňujeme sem:**

- ◆ Funkční prototypy.
- ◆ Symbolické konstanty definované pomocí direktivy #define nebo const.
- ◆ Deklarace struktur.
- ◆ Deklarace tříd.
- ◆ Deklarace šablon.
- ◆ Vložené funkce.

# 1. Pamäťové modely a menné priestory

Doba trvania úložiska,  
rozsah platnosti a väzba

3 odlišné schémy uloženia dát v pamäti:

1. **Automatické premenné** - platia v bloku
2. **Statické premenné** - definované mimo blokov funkcií alebo pomocou kľúčového slova **static**.  
Pretrvávajú po celú dobu behu programu
3. **Dynamická pamäť** - alokuje sa pomocou **new**,  
pretrváva až do použitia **delete** alebo do konca behu programu



# 1. Pamäťové modely a menné priestory

Doba trvania úložiska,  
rozsah platnosti a väzba

## Automatické premenné

- Unikátne pre každý blok.
- Ak sa bloky vnorujú, spravidla platí, že lokálna premenná predefinováva globálnu premennú
- môžeme použiť kľúčové slovo **register** pre použitie registra CPU

# 1. Pamäťové modely a menné priestory

Doba trvania úložiska,  
rozsah platnosti a väzba

## **Statické premenné**

- existujú po celú dobu behu programu
- inicializujú sa iba raz
- radíme sem externé, statické, aj externé statické premenné

Externé premenné - príklad 8.16

# 1. Pamäťové modely a menné priestory

Doba trvania úložiska,  
rozsah platnosti a väzba

## Statické premenné

Tabulka 8.2 Paměťové třídy

Paměťová třída	Způsob vzniku	Rozsah platnosti	Vazba	Trvání
Automatický blok	Standardně pro parametry a proměnné deklarované uvnitř funkce	Lokální	Interní	Po dobu běhu funkce
Externí	Standardně pro proměnné deklarované vně funkce	Globální	Externí	Po dobu programu
Statická	Aplikací klíčového slova static na proměnnou deklarovanou uvnitř funkce	Lokální	Interní	Po dobu programu
Externí statická	Aplikací klíčového slova static na proměnnou deklarovanou vně funkce	Globální	Interní	Po dobu programu

# 1. Pamäťové modely a menné priestory

## Menné priestory

**Mená môžu mať:**

Premenné  
Štruktúry  
Enumerátory,  
Funkcie,  
Triedy,  
Členy tried a štruktúr

**=> KONFLIKTY!**

# 1. Pamäťové modely a menné priestory

## Menné priestory

```
namespace Jack {
    double pail;
    void fetch();
    int pal;
    struct Well { ... };
}
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    int pal;
    struct Hill { ... };
}
```

- smú sa zanorovať do seba
- nesmú byť umiestnené do bloku
- globálny priestor mien
- sú otvorené -> môžeme do nich pridávať členy

# 1. Pamäťové modely a menné priestory

## Menné priestory

Použitie pomocou using

```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
char fetch;
int main()
{
    using Jill::fetch;           // vloží fetch do lokálneho priestoru jmen
    /* double fetch;           // Chyba! Již máme lokální fetch
    cin >> fetch;              // zapisuje hodnotu do Jill::fetch
    cin >> ::fetch;            // zapisuje hodnotu do globální fetch
    ...
}
```



# 1. Pamät'ové modely a menné priestory

## Menné priestory

Použitie pomocou using namespace

```
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };
}
char fetch; // globálny priestor jmen
int main()
{
    using namespace Jill; // importuje jmena priestoru jmen
    struct Hill Thrill; // vytváří Jill::Hill
    double water = bucket(2); // používa Jill::bucket();
    double fetch; // není chyba; skrývá Jill::fetch
    cin >> fetch; // zapisuje hodnotu do lokální proměnné
    fetch
    cin >> ::fetch; // zapisuje hodnotu do globální proměnné
    fetch
    cin >> Jill::fetch; // zapisuje hodnotu do proměnné Jill::fetch
    ...
}
```

# 1. Pamäťové modely a menné priestory

## Menné priestory

Použitie pomocou using namespace

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



# 2. Dedičnosť tried

## 1. Začneme jednoduchou základnou triedou

1. Odvodenie triedy
2. Konštruktory: Úvahy o prístupe
3. Použitie odvodenenej triedy
4. Vzt'ah medzi odvodenou a základnou triedou

## 2. Dedičnosť

## 2. Dedičnosť tried

- 3. Polymorfná verejná dedičnosť**
- 4. Riadenie prístupu: protected**
- 5. Dedičnosť a dynamické pridelovanie pamäti**

# 2. Dedičnosť tried

## **Odporúčaná literatúra**

Začiatok témy - strana 511

Zhrnutie - 562

Opakovacie otázky - strana 562

Programovacie cvičenia - strana 564

## 2. Dedičnosť tried

### Základná a odvodená trieda

- [http://www.tutorialspoint.com/cplusplus/cpp\\_classes\\_objects.htm](http://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm) + odkazy v spodnej časti stránky
- Program - 12.1.

### Dedičnosť

- modeluje vzťah “**je**” alebo “**je druhu**” napr. tank je vozidlo, auto je vozidlo
- nemodeluje vzťah “**má**”, ani “**implementuje ako**”, ani “**používa**”
- pridáva a predefinováva, neredukuje

## 2. Dedičnosť tried

### Základná a odvodená trieda

- Dedičnosť v CPP podporuje modifikátory prístupu pri odvodzovaní tried - public, protected, private
- <http://stackoverflow.com/a/860353>
- <http://stackoverflow.com/a/1372858>

## 2. Dedičnosť tried

Riadenie prístupu

**Public, Protected, Private**

## 2. Dedičnosť tried

### Základná a odvodená trieda

### Konštruktor základnej triedy

```
Overdraft::Overdraft(const char *s, long an, double bal, double  
                    ml, double r) : BankAccount(s, an, bal)  
{  
    maxLoan = ml;  
    owesBank = 0.0;  
    rate = r;  
}
```

Zde část

```
: BankAccount(s, an, bal)
```

## 2. Dedičnosť tried

Základná a odvodená trieda

**Konštruktor základnej triedy**

**Konštruktor sa nededí, nemôže byť virtuálny**

```
derived::derived(type1 x, type2 y) : base(x,y)
// seznam inicializátorů
|
|
|   ...
|
|
```



## 2. Dedičnosť tried

### Základná a odvodená trieda

#### **Deštruktor sa nededí, virtuálny je v prípade základnej triedy**

Constructors:

- B does not inherit constructors from A;
- Unless B's ctor explicitly calls *one of A's* ctor, the default ctor from A will be called automatically *before* B's ctor body (the idea being that A needs to be initialized before B gets created).

Destructors:

- B does not inherit A's dtor;
- *After* it exits, B's destructor will automatically call A's destructor.

## 2. Dedičnost' tried

### Referencie a pointre na objekty

- je možné pretypovať na predka, na potomka je možné pretypovať iba pri použití explicitného pretypovania

```
double x = 2.5;  
int * pi = &x;    // neplatné prirazení, neodpovídají si typy ukazatelů  
long & r1 = x;    // neplatné prirazení, neodpovídají si typy referencí
```

Ale reference nebo ukazatel na základní třídu může odkazovat na objekt odvozené třídy, aniž by bylo potřeba použít explicitní přetypování. Například následující inicializace jsou přípustné:

```
Overdraft dilly ("Annie Dill", 493222, 2000);  
BankAccount * pb = &dilly;    // ok  
BankAccount & rb = dilly;    // ok
```

## 2. Dědičnost tříd

### Virtuálně metody

#### **Co je potřeba znát o virtuálních funkcích**

Hlavní body týkající se virtuálních funkcí jsme již probrali:

- ◆ Uvození deklarace metody třídy klíčovým slovem `virtual` v základní třídě ji činí virtuální pro základní třídu a všechny třídy od ní odvozené a třídy odvozené od odvozených tříd atd.
- ◆ Jestliže je virtuální metoda vyvolána pomocí reference na objekt nebo pomocí ukazatele na objekt, použije program metodu definovanou pro typ objektu, nikoli metodu definovanou pro typ reference nebo ukazatele. Toto se nazývá dynamická nebo také pozdní vazba. Toto chování je důležité, protože platí vždy pro ukazatele či referenci základní třídy, mají-li odkazovat na objekt odvozeného typu.
- ◆ Pokud definujete třídu, která bude použita jako základní pro dědičnost, deklaruje jako virtuální ty metody třídy, u kterých je předpoklad, že budou v odvozených třídách předdefinovány.

## 2. Dedičnosť tried

### Virtuálne metódy

```
class Animal
{
    public:
        // turn the following virtual modifier on/off to see what happens
        //virtual
        std::string Says() { return "?"; }
};

class Dog: public Animal
{
    public: std::string Says() { return "Woof"; }
};

void test()
{
    Dog* d = new Dog();
    Animal* a = d;        // refer to Dog instance with Animal pointer

    cout << d->Says();    // always Woof
    cout << a->Says();    // Woof or ?, depends on virtual
}
```



## 2. Dedičnosť tried

### Virtuálne metódy

príklady pre vysvetlenie -

<http://stackoverflow.com/a/2391721>

<http://stackoverflow.com/a/1307867>

<http://stackoverflow.com/a/1307282>

<http://stackoverflow.com/a/21607>

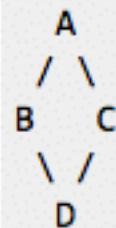
Operátor priradenia sa taktiež nededí. - strana 540

## 2. Dedičnosť tried

# Virtuálna dedičnosť

```
class A { public: void Foo() {} };  
class B : public A {};  
class C : public A {};  
class D : public B, public C {};
```

The above class hierarchy results in the "dreaded diamond" which looks like this:



An instance of D will be made up of B, which includes A, and C which also includes A. So you have two "instances" (for want of a better expression) of A.

When you have this scenario, you have the possibility of ambiguity. What happens when you do this:

```
D d;  
d.Foo(); // is this B's Foo() or C's Foo() ??
```

Virtual inheritance is there to solve this problem. When you specify virtual when inheriting your classes, you're telling the compiler that you only want a single instance.

```
class A { public: void Foo() {} };  
class B : public virtual A {};  
class C : public virtual A {};  
class D : public B, public C {};
```

This means that there is only one "instance" of A included in the hierarchy. Hence

```
D d;  
d.Foo(); // no longer ambiguous
```

## 2. Dedičnosť tried

### Dynamické pridelovanie pamäte - príklad

v konštruktore alokujeme pomocou **new**:

```
#include <string.h>
BankAccountD::BankAccountD(const char *s, long an, double bal)
{
    fullName = new char[strlen(s) + 1];
    strcpy(fullName, s);
    acctNum = an;
    balance = bal;
}
```

## 2. Dedičnosť tried

### Dynamické pridelovanie pamäte - príklad

v deštruktore dealokujeme pomocou **delete**:

```
BankAccountD::~~BankAccountD()  
{  
    delete [] fullName;  
}
```

Rôzne možnosti implementácie - strana 543



## 2. Dedičnost' tried

### Abstraktné triedy

strana 550

```
class BaseEllipse // abstraktní základní třída
{
private:
    double x; // x-ová souřadnice středu elipsy
    double y; // y-ová souřadnice středu elipsy
    ...
private:
    BaseEllipse(double x0 = 0, double y0 = 0) : x(x0), y(y0) {}
    virtual ~BaseEllipse() {}
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual Area() const = 0; // čistě virtuální funkce
    ...
}
```

Jestliže třída obsahuje čistě virtuální funkci, pak nelze vytvořit objekt této třídy. Smyslem tříd s čistě virtuálními funkcemi je sloužit jako základní třídy. Aby třída byla opravdu abstraktní základní třídou, musí mít alespoň jednu čistě virtuální funkci.

## 2. Dedičnost' tried

### Interfaces

<http://stackoverflow.com/a/318105>

The whole reason you have a special Interface type-category in addition to abstract base classes in C#/Java is because C#/Java do not support multiple inheritance.

C++ supports multiple inheritance, and so a special type isn't needed. An abstract base class with no non-abstract (pure virtual) methods is functionally equivalent to a C#/Java interface.

## 2. Dedičnost' tried

### Multiple inheritance

<http://www.learncpp.com/cpp-tutorial/117-multiple-inheritance/>

# 3. Opakované používanie kódu v C++

**1. Triedy, ktorých členy sú objekty (objektové členy)**

**2. Šablóny tried**

1. Definícia šablóny triedy
2. Použitie šablóny triedy
3. Bližší pohľad na šablónu triedy
4. Príklad šablóny pole a netypové parametre
5. Univerzálnosť šablón
6. Špecializácia šablón
7. Členské šablóny
8. Šablóny ako parametre
9. Šablónové triedy a priatelia

# 3. Opakované používanie kódu v C++

## **Odporúčaná literatúra**

Začiatok témy - strana 567

Zhrnutie - 628

Opakovacie otázky - strana 631

Programovacie cvičenia - strana 632

### 3. Opakované používanie kódu v C++

#### Triedy, ktorých členy sú objekty

Vloženie objektov do triedy je možné dvoma spôsobmi:

1. Kompozícia - strana 567, príklad na:

<http://jliusun.bradley.edu/~jiangbo/cs106/ch12.htm>

2. Súkromná dedičnosť - strana 579,

<http://jliusun.bradley.edu/~jiangbo/cs106/ch13.htm>

# 3. Opakované používanie kódu v C++

## Šablóny tried

1. prvok generického programovania
2. umožňujú písať všeobecné funkcie pre parametrizované typy
3. spravidla umiestňujeme šablóny do samostatných hlavičkových súborov
4. v programe ich treba inštancovať

```
Stack<int> kernels;      // vytvoří zásobník pro typ int  
Stack<String> colonels; // vytvoří zásobník objektů třídy String
```

Ako prepísať triedu na šablónu - strana 588

Vzorová šablóna - 13.9.10

### 3. Opakované používanie kódu v C++

#### Šablóny tried a funkcií

[http://en.cppreference.com/w/cpp/language/function\\_template](http://en.cppreference.com/w/cpp/language/function_template)

[http://en.cppreference.com/w/cpp/language/class\\_template](http://en.cppreference.com/w/cpp/language/class_template)

<http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part-1>