KIV/CPP – Programování v jazyce C++

05. Vícenásobná dědičnost, práce s typy, proudy

Martin Úbl

KIV ZČU

2023/2024

Dědičnost

- připomenutí
- polymorfismus (dynamický)
- klíčová slova `virtual`, `override`
- modifikátor viditelnosti dědičnosti
  - `public` – klasická dědičnost
  - `protected` – public a protected atributy a metody budou v potomkovi protected
  - `private` – totéž, ale budou v potomkovi private

```
class Potomek : public Rodic {
  ...
};
```

Tabulka virtuálních metod

- trochu odbočka: tabulka virtuálních metod
- nástroj k uskutečnění polymorfismu
- v podstatě seznam adres implementací virtuálních metod
- každý potomek si ji vyplní podle svého
- klíčovým slovem `virtual` se metoda do tabulky přidá
- obsahuje další servisní prvky „nad“ samotnou tabulkou
  - `top_offset` – offset části tabulky od vrchu (v případě skládání více tabulek do jedné velké)
  - `type_info` pointer – ukazatel na RTTI (viz dále)

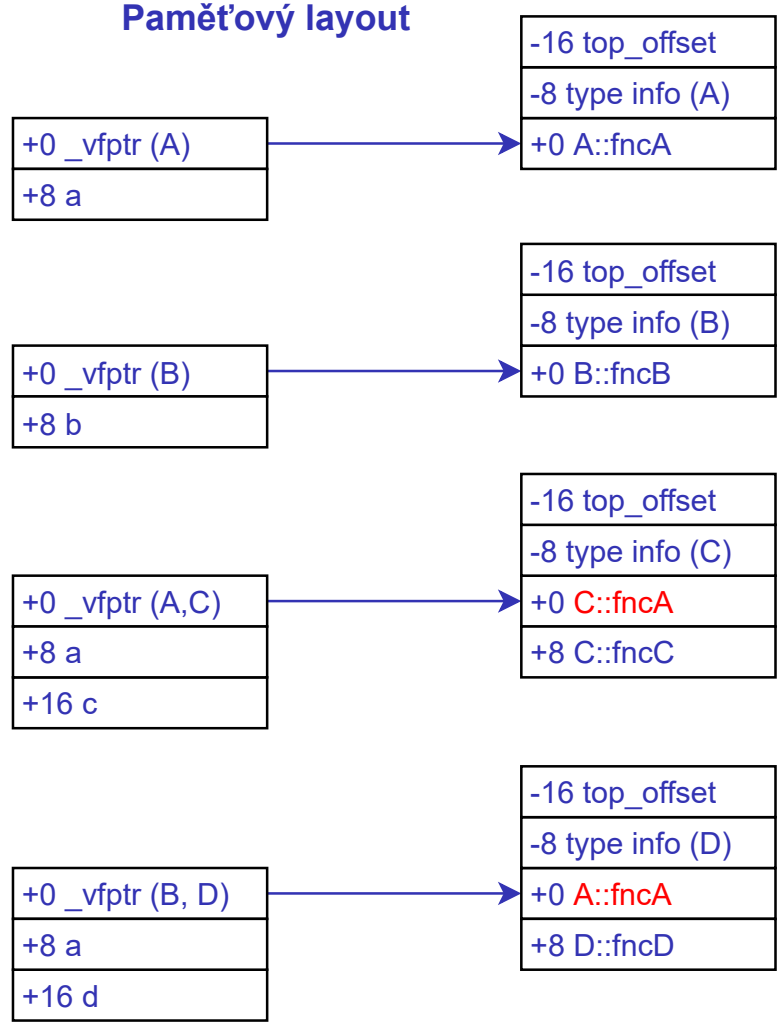
Tabulka virtuálních metod

- tabulka virtuálních metod a přepsané metody

Definice třídy

```
class A {
    public:
        virtual void fncA();
        int a;
}


class B {
    public:
        virtual void fncB();
        int b;
}


class C : public A {
    public:
        virtual void fncA() override;
        virtual void fncC();
        int c;
}


class D : public B {
    public:
        virtual void fncD();
        int d;
}
```

Paměťový layout



Vícenásobná dědičnost

- vícenásobná dědičnost
- je možná, ale občas nevyzpytatelná
- lze takto nahrazovat absenci prvku jazyka *rozhraní* v kombinaci s abstraktní třídou
- třída může dědit od více rodičů, oddělují se čárkou v definici
- pak skutečně dědí od všech rodičovských tříd

```
class Potomek : public Matka, public Otec {
  ...
};
```

Vícenásobná dědičnost

- vícenásobná dědičnost
- paměťový layout se pak „skládá“ dohromady
  - tabulky virtuálních metod taktéž
- je nutné počítat s tím, že potomek může být přetypován na libovolného z předků
  - tzn. musí jít bez újmy získat ukazatel na kteréhokoliv z nich

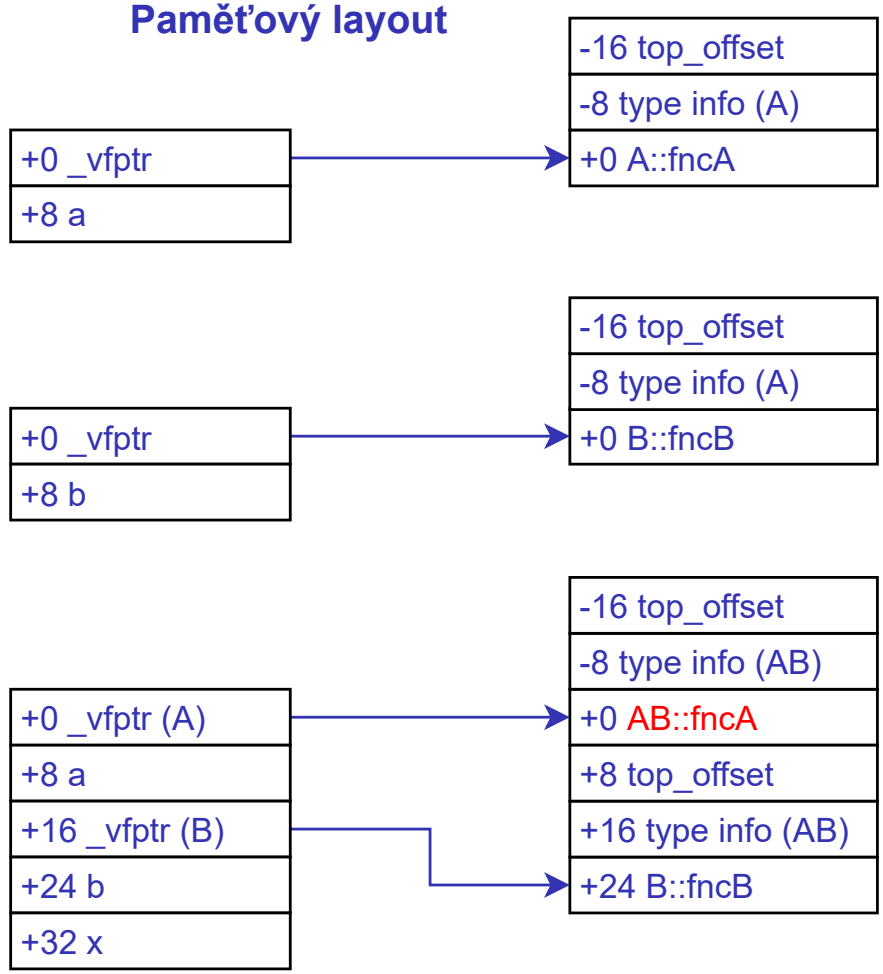
Vícenásobná dědičnost

- tabulka virtuálních metod a vícenásobná dědičnost

Definice třídy

```
class A {
    public:
        virtual void fncA();
        int a;
}

class B {
    public:
        virtual void fncB();
        int b;
}

class AB : public A, public B {
    public:
        virtual void fncA() override;
        int x;
}
```

Paměťový layout



Vícenásobná dědičnost

- ... přináší to s sebou problémy
  1. co když více rodičovských tříd definuje stejnojmennou metodu?
  2. co když rodičovské třídy dědí od stejného předka?
     - *diamond problem*
- lze předejít
  - lepším návrhem (1, 2)
  - explicitní určení předka při volání (1)
  - kompozice místo polymorfismu (1)
  - virtuální dědičnost (2)

Vícenásobná dědičnost

- stejnojmenné metody
- např. `Student` i `Employee` mají metodu `GetTimeSchedule()`
  - obě ale dělají třeba něco trochu jiného

```
class WorkingStudent : public Student,
                       public Employee {
  ...
};
```

- jak vybrat správnou verzi? explicitně

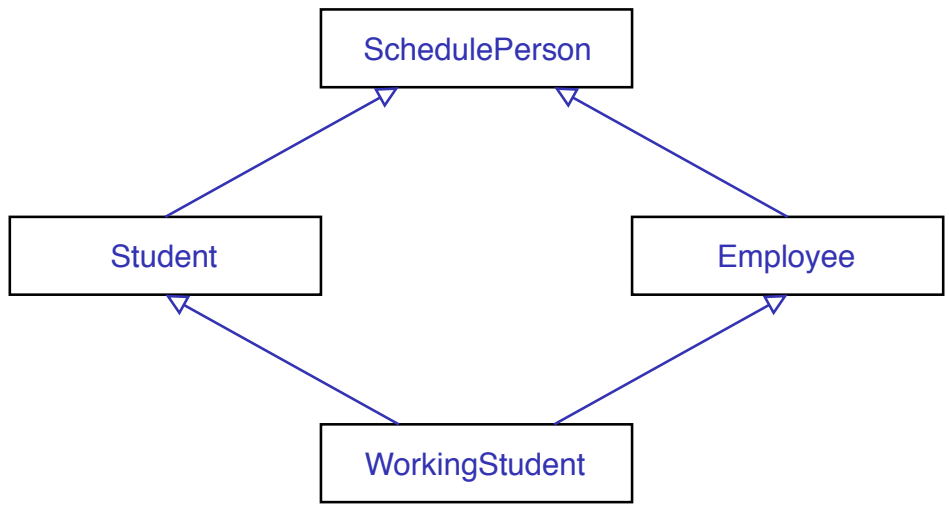
```
WorkingStudent pepa;

pepa.Student::GetTimeSchedule();
pepa.Employee::GetTimeSchedule();
```

Vícenásobná dědičnost

- metody se navzájem překrývají
- snaha se takovému schématu vyhnout
  - lepším návrhem
  - jinou dekompozicí
  - společným předkem a virtuální dědičností

Vícenásobná dědičnost

- *diamond problem*
- prarodič definuje metodu `GetTimeSchedule` vč. implementace
- rodiče oba dědí od prarodiče
- potomek dědí od obou rodičů, nepřepisuje metodu `GetTimeSchedule`
- problém: oba rodiče obsahují v tabulce virtuálních metod `GetTimeSchedule` prarodiče
  - která je ta správná?



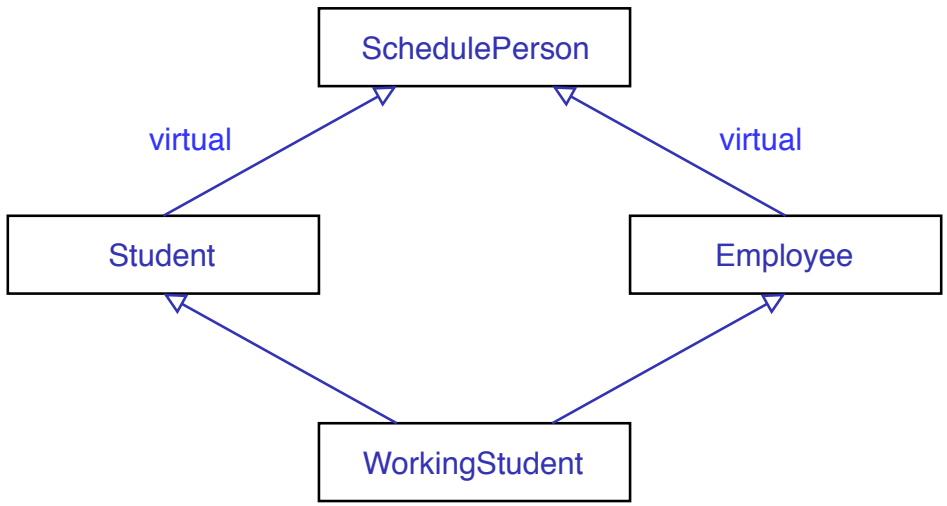
Vícenásobná dědičnost

- problém: oba rodiče obsahují v tabulce virtuálních metod `GetTimeSchedule` prarodiče
  - která je ta správná?
- obě, jsou identické
- problém řeší virtuální dědičnost

```
class Student : public virtual SchedulePerson
{
  ...
};
class Employee : public virtual SchedulePerson
{
  ...
};
```

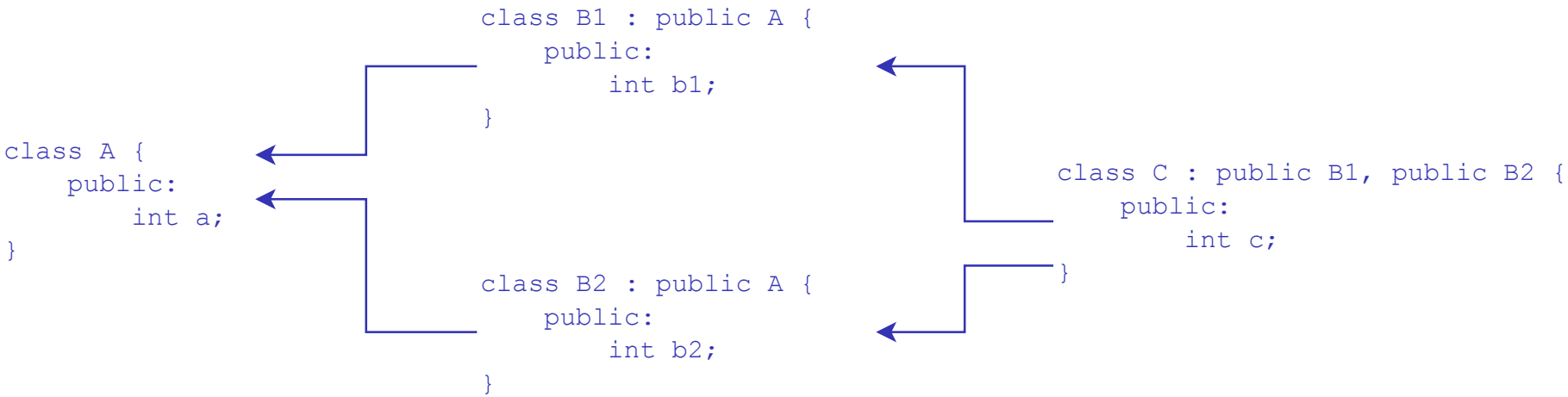
Vícenásobná dědičnost

- *diamond problem*
- virtuální dědičnost předků



Vícenásobná dědičnost

- *diamond problem*
- otázka: jak bude vypadat paměťový layout?



Vícenásobná dědičnost

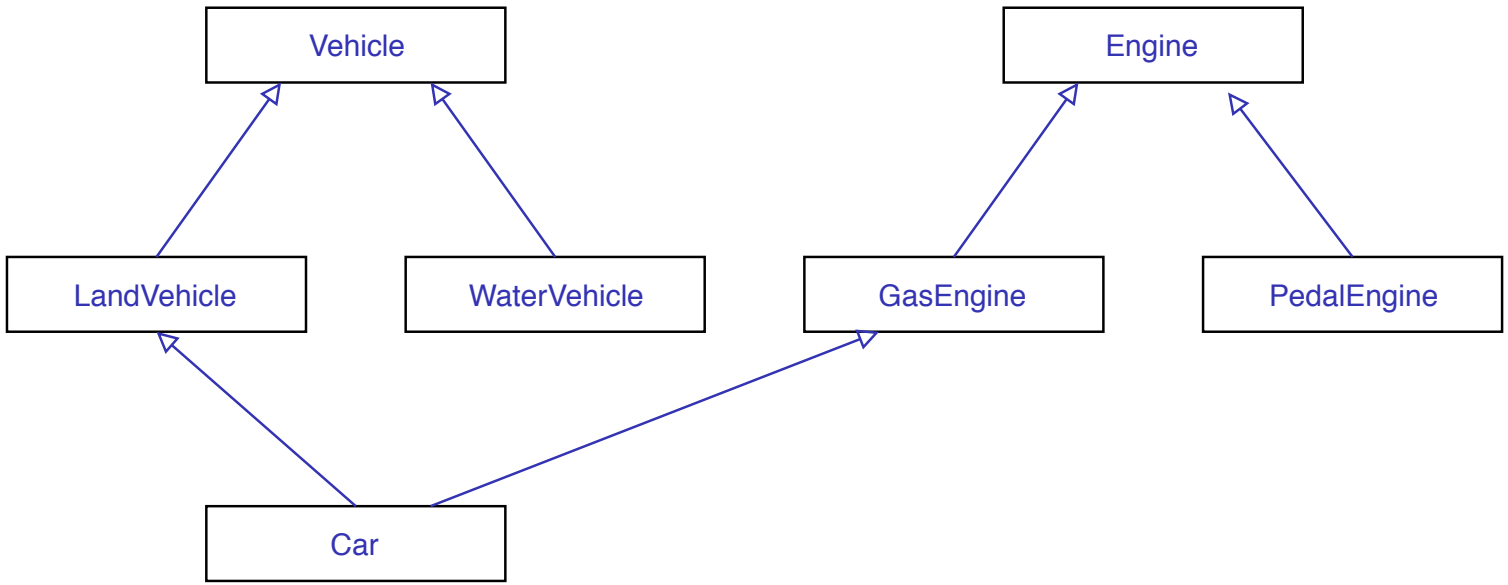
- mýtus:
  - vícenásobná dědičnost je špatná!
- pravda:
  - vícenásobná dědičnost je nenahraditelný nástroj při řešení určitých problémů
  - vždy otázka návrhu
  - může být zastoupena statickým polymorfismem (další semináře)

Vícenásobná dědičnost

- hodí se, když potomek má sdílet myšlenky a fungování několika tříd
  - sdílení implementace
  - polymorfismus
- „Rules of thumb“
  - `https://isocpp.org/wiki/faq/multiple-inheritance`
  - „*Use inheritance only if doing so will remove if / switch statements from the caller code.*“
  - „*Try especially hard to use abstract base classes when you use multiple inheritance.*“
  - „*Consider the “bridge” pattern or nested generalization as possible alternatives to multiple inheritance.*“
- pozn.: *bridge pattern* – v podstatě kompozice s výběrem typu za běhu
- pozn. 2: *nested generalization* – jedna primární hierarchie specializovaná pro každý podtyp

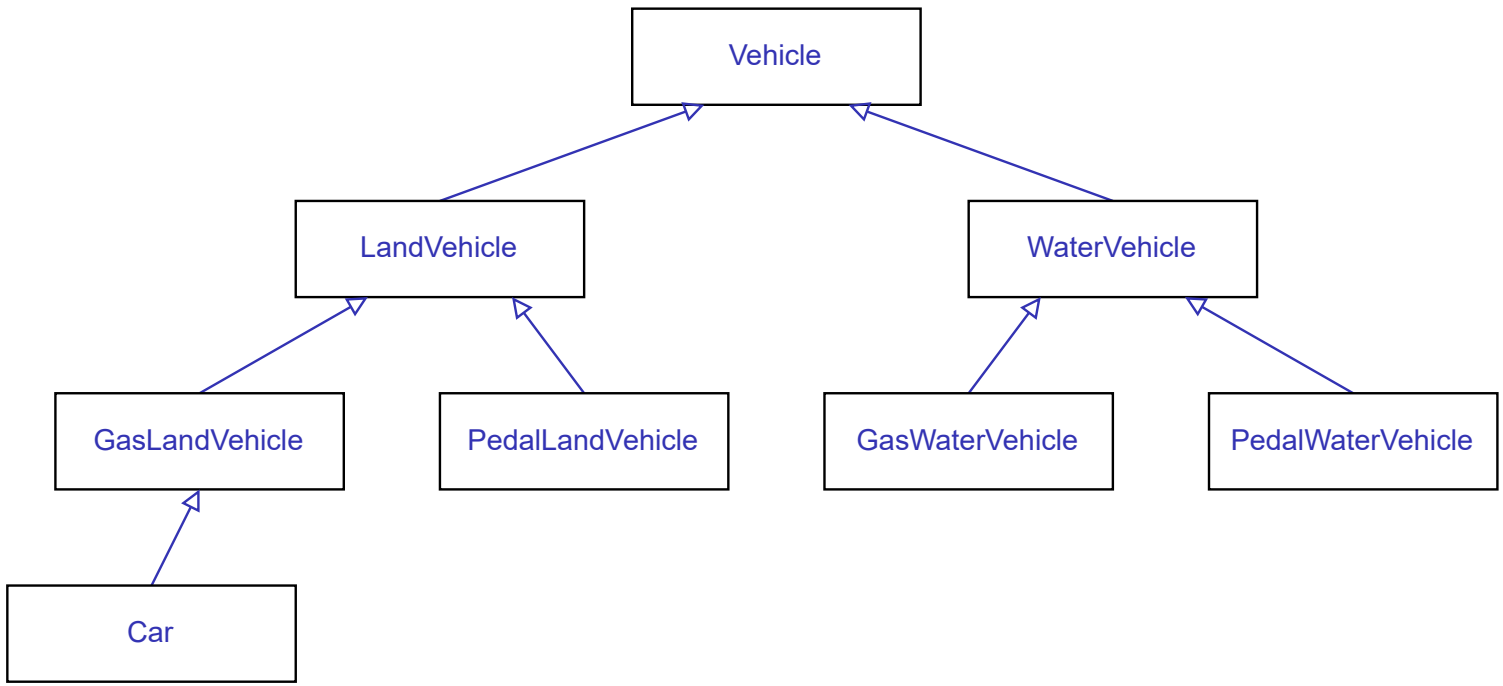
Vícenásobná dědičnost

- isocpp příklad
- vícenásobná dědičnost a zanořené zobecňování



Vícenásobná dědičnost

- isocpp příklad
- vícenásobná dědičnost a zanořené zobecňování



Práce s typy

- práce s typy
- převody mezi typy
  - implicitní
  - C-style cast
  - `static_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - `const_cast`
- pro polymorfní typy v `shared_ptr`
  - `std::dynamic_pointer_cast`

Práce s typy

- Run-Time Type Info (RTTI)
- informace o typu objektu v čase běhu
- pouze pro polymorfní typy
- dovoluje použití některých konstruktů
  - `dynamic_cast`
  - `typeid`
  - `type_info`
- na instanci RTTI odkazuje `type_info` položka tabulky virtuálních metod

Práce s typy

- `static_cast`
- „náhrada“ implicitní konverze
- „obejití“ narrowing konverze
- i např. při používání C knihoven – konverze mezi `void*` a konkrétním pointerem
- neodbourává `const`

```
int a = 5;
float b = static_cast<float>(a);
int c = static_cast<int>(b);

void* mem = ...;
uint8_t* cmem = static_cast<uint8_t*>(mem);
```

Práce s typy

- `reinterpret_cast`
- donutí kompilátor, aby se k entitě choval jako k úplně jinému typu
- může být nebezpečné – nic nás nezastaví v konverzi např. `int` na `std::string*`
- konverze ukazatele na *float* s hodnotou `2.0f` na *integer* nedá hodnotu 2
- relativně málo situací, kdy je nutný
- *UserData* v knihovnách

```
float a = 2.0f;
int b = *reinterpret_cast<int*>(&a);

// b = 1073741824
```

Práce s typy

- `dynamic_cast`
- `static_cast` s kontrolou
- používá RTTI pro převod mezi polymorfními typy
- přetypování pointeru na pointer
  - při nezdaru vrací `nullptr`
- přetypování reference na referenci
  - při nezdaru vyhodí výjimku `std::bad_cast`

```
Rodic *r = ...;
Potomek *p = dynamic_cast<Potomek*>(r);

if (p == nullptr)
  // "r" neni typu Potomek
```

Práce s typy

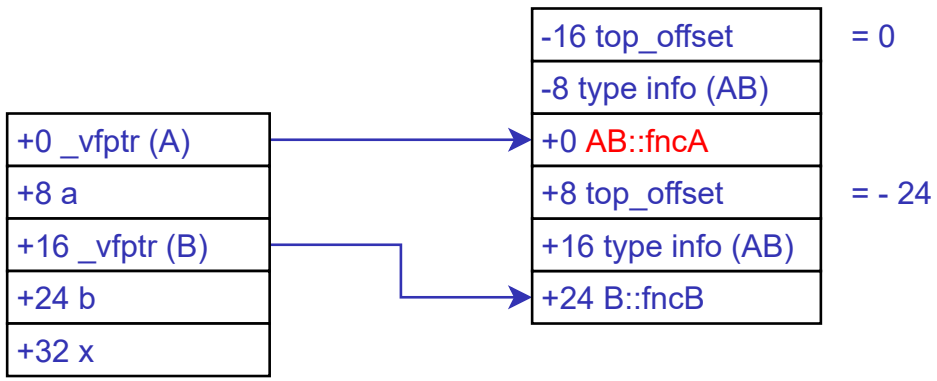
- dynamic_cast
- reference

```
Rodic &r = ...;

try
{
  Potomek& p = dynamic_cast<Potomek&>(r);
  p.metoda();
}
catch (std::bad_cast ex)
{
  // "r" neni typu Potomek
}
```

Práce s typy

- `dynamic_cast`
- může vracet jiný ukazatel
- viz paměťový layout při vícenásobné dědičnosti
- `dynamic_cast` na jiné předky může vracet jiný ukazatel
  - zpětný `dynamic_cast` na potomka využívá RTTI i `top_offset` položku tabulky virtuálních metod

```
class AB : public A, public B {
    public:
        virtual void fncA() override;
        int x;
}
```



Práce s typy

- `const_cast`
- pro přidání nebo odebrání `const` z typu
- používat velmi obezřetně pro odebírání
  - modifikace konstantní paměti může být nedefinované chování
  - lepší pozměnit návrh

```
const char* pozdrav = "ahoj";

char* m_pozdrav = const_cast<char*>(pozdrav);
m_pozdrav[0] = 'b'; // ouch!
```

- otázka: proč se tato modifikace nemusí povést?

Práce s typy

- `const_cast`
- paměť, co ve skutečnosti není const, ale je jako const předávána
- např. v `std::string`
  - `c_str()` vrací `const char*`

```
std::string s_pozdrav = "ahoj";

char* ms_pozdrav = const_cast<char*>
                              (s_pozdrav.c_str());
ms_pozdrav[0] = 'b';
```

Shrnutí

- C-style cast
  - zapomenout
- `static_cast`
  - pro běžné přetypování
  - POD mezi sebou, objekty pokud nepotřebujeme RTTI
- `dynamic_cast`
  - pro polymorfní přetypování
- `reinterpret_cast`
  - pokud možno vyhýbat se
  - jinak např. při nutnosti surové práce s pamětí
- `const_cast`
  - pokud možno nikdy
  - ideálně jen pro přidání const
  - odebrání const jen pokud už skutečně není jiná možnost

Poznámky

- klíčové slovo `decltype`
  - zjištění a aplikace typu (např. pro zajištění, že typ bude identický)

```
unsigned long long a = 15;
decltype(a) b = 25;
```

- klíčové slovo `typeid`
  - vrací RTTI informace v instanci `std::type_info`
  - může se hodit např. pro ladění

```
std::string p;
std::cout << typeid(p).name();
```

Proudy

- obecný mechanismus pro práci se vstupně-výstupními proudy
- vstupní proud
  - `std::istream`
- výstupní proud
  - `std::ostream`
- kombinace
  - `std::iostream`
- specializace pro soubory, řetězce
- přetížené operátory bitového posuvu << a >>
  - pouze pro textový vstup a výstup!

Proudy

- respektuje RAII
  - konstruktor otevírá stream (např. soubor)
  - destruktor zavírá stream
- příznaky pro otevření (vždy std::ios::)
  - `in` – z proudu budeme číst ("r")
  - `out` – do proudu budeme zapisovat ("w")
  - `app` – kurzor se bude přesouvat na konec proudu ("a")
  - `trunc` – proud se při otevření vyprázdní
  - `binary` – binární čtení/zápis ("b")

Souborové proudy

- souborové proudy
- `#include <fstream>`
- vstupní proud
  - `std::ifstream`
- výstupní proud
  - `std::ofstream`
- kombinace
  - `std::fstream`
- parametry konstruktoru
  - cesta k souboru
  - volitelně mód otevření

Souborové proudy

- souborové proudy
- výstupní textový proud

```
std::ofstream vystup("soubor.txt");

vystup << "Hello" << std::endl;
vystup << "World" << std::endl;
```

Souborové proudy

- souborové proudy
- vstupní textový proud
- implicitním oddělovačem je mezera nebo zakončení řádky

```
std::ifstream vstup("soubor.txt");

std::string a, b;
vstup >> a;
vstup >> b;
```

Souborové proudy

- souborové proudy
- binární zápis

```
std::ofstream vystup("soubor.bin",
      std::ios::out | std::ios::binary);

const char* binbuf = "Binarni\x00zapis\x00";

vystup.write(binbuf, 14);
```

Souborové proudy

- souborové proudy
- binární čtení

```
std::ifstream vystup("soubor.bin",
      std::ios::in | std::ios::binary);

char data[14];

vystup.read(data, 14);
```

Řetězcové proudy

- řetězcové proudy
- `#include <sstream>`
- vstupní proud
  - `std::istringstream`
- výstupní proud
  - `std::ostringstream`
- kombinace
  - `std::stringstream`
- parametry konstruktoru
  - řetězec co se má použít jako základ
  - volitelně i mód otevření
- obsah se pak vyzvedne metodou `str()`
- chová se jinak stejně jako souborový

Řetězcové proudy

- řetězcové proudy
- výstupní řetězcový proud

```
std::ostringstream vystup(
                   "Sedmeho␣dne␣Buh␣rekl:␣",
                   std::ios::app);

vystup << "Hello";
vystup << "_";
vystup << "World";

std::string vysledek = vystup.str();
```

- pozn.: bez *append* příznaku by se základní text přepsal

Proudy

- `std::getline`
- získá celou „řádku“
  - jinak je implicitně parsováno i do mezer
- lze definovat znak, o který se „zarazit“
- navíc vrací instanci proudu, a ten má přetížený operátor `bool()`
  - lze tedy snadno ověřit úspěšnost operace

```
std::ifstream vstup("soubor.txt");

std::string str;
while (std::getline(vstup, str))
  std::cout << str << std::endl;
```

Proudy

- `std::getline`
- např. parsuje po střednících

```
std::ifstream vstup("soubor.csv");

std::string str;
while (std::getline(vstup, str, ';'))
  std::cout << str << std::endl;
```

Doplnění k proudům

- zjištění pozice ve streamu
  - `tellg` (input), `tellp` (output)
  - vrací instanci `streampos` (často int nebo `size_t`)
- nastavení pozice ve streamu (seek)
  - `seekg` (input), `seekp` (output)
- např. zjištění velikosti souboru v bajtech:

```
std::ifstream input("soubor.txt");

input.seekg(0, input.end);
int fileSize = input.tellg();
input.seekg(0, input.beg);
```

Doplnění k proudům

- standardní proudy
  - proudové obaly nad implicitně otevřenými soubory
  - `std::cin` – standardní vstup, `stdin`
  - `std::cout` – standardní výstup, `stdout`
  - `std::cerr` – chybový výstup, `stderr`
- instance proudů lze předávat obecnou referencí
- lze tedy mít funkci schopnou zapisovat do/číst z libovolného streamu
  - parametr `std::ostream&`
    - předáním `std::cout` zapíše do konzole
    - předáním instance `std::ofstream` zapíše do souboru
  - parametr `std::istream&`
    - předáním `std::cin` přečte z konzole
    - předáním instance `std::ifstream` přečte ze souboru

Doplnění k proudům

- speciální vstupní/výstupní „znaky“
- `std::endl`
  - vloží konec řádky `\n` a zavolá `flush()`
  - pozn. `flush()` už bude volat *syscall* (může zpomalovat)
- `std::ends`
  - vloží nulový znak `\0`
- `std::ws`
  - „zahodí“ bílý znak (mezera, tabulátor) ze vstupu

Proudové manipulátory

- proudové manipulátory
- `#include <iomanip>`
- obsahují modifikátory toho, jak se zpracovává vstup/výstup proudů
- např. formát čísla
  - `hex`, `dec`, `oct`
  - `fixed`, `scientific` (float)
- délka čísla a výplň
  - `setprecision`
  - `setw`, `setfill`
- transkripce typů
  - `boolalpha` a `noboolalpha`
- a další
  - `showbase` a `noshowbase`
- modifikátory se „vkládají“ do streamu operátory << a >>

Proudové manipulátory

- proudové manipulátory
- např. hex výstup

```
std::cout << std::showbase << std::hex << 254;
// 0xfe
```

Proudové manipulátory

- existuje jich spousty
- `https://en.cppreference.com/w/cpp/io/manip`
- více na příkladech

Formátování

- `std::format` (od C++20)
- `#include <format>`
- obdoba podobných formátovacích funkcí z jiných jazyků
- dovoluje na základě formátovacího řetězce zformátovat výstup
  - zástupné sekvence v `{ ... }`
  - uvnitř volitelně pořadové číslo parametru, dvojtečka a formatter
  - escapování zdvojením
- používá variadické šablony pro variabilní počet parametrů (viz další přednášky)

```
std::format("Hello␣{}!", "world");
std::format("Hello␣{1}␣and␣{2}!␣",
                    "Atlas", "P-body");
std::format("Cislo␣{:.2f}", 15.245);
```

Formátování

- `std::format`, jak lze např. formátovat
- `{:#x}`, `{:#o}`, `{:#b}` – hexadecimální, oktální a binární výstupy, vč. prefixu (#)
- `{:0>6}`, `{:0<6}`, `{:0^6}` – zarování doprava, doleva, centrování, padding na 6 znaků, výplň nulou
  - `000066`, `660000`, `006600`
- a další...

Formátování

- `std::format` (od C++20)
- formatterů je spousty, podobají se těm z C (printf)
- `https://en.cppreference.com/w/cpp/utility/format/format`
- více na cvičení

Formátování

- `std::vformat` (od C++20)
- dynamické formátování
  - narozdíl od `std::format`, který je kompletně staticky kontrolovaný
- např. formátování seznamu jmen se zarováním na nejdelší

```
std::string fmt = std::format("{{:>{}}}",
                              longest.length());

for (const auto& p : osoby) {
  std::cout
    << std::vformat(fmt, std::make_format_args(p))
    << std::endl;
}
```

Formátování

- `std::format` (od C++20)
- podporuje rekurzivní definice
- formát definovaný formátem
- viz předchozí příklad, ale lépe, jen pomocí `std::format`

```
for (const auto& p : osoby) {
   std::cout
     << std::format("{:>{}}", p, longest.length())
     << std::endl;
}
```

Poznámky

- C++ proudy nejsou nutně to nejefektivnější
  - „mezibod“ čitelnosti, univerzálnosti a výkonu
  - lze optimalizovat pro to či ono
- proudy nejsou thread-safe
  - např. výpis do `std::cout` z více vláken na více částí výstup různě roztrhá a promíchá
  - částečně řeší `std::osyncstream`

Vsuvka – KIV/ZOS + KIV/UPS

- C++ proudy pro binární čtení a zápis
- KIV/ZOS
  - implementace virtuálního FS
  - binární soubor jako obraz disku
  - nutnost číst a zapisovat struktury
    - padding, alignment
    - endianita
    - binární data
- KIV/UPS
  - možné použití streamů pro práci s aplikačními pakety
  - textový protokol, stringstream
  - od C++2y možná i síťový stream
- příklady na cvičení