



Informační systém Masarykovy univerzity

Zodpovězení odpovědníku (student)**odpovědník Zkouška 17. 12. 2018**

Celkem 16 otázek, za každou otázku je maximálně 5 bodů. Otázka může mít více správných odpovědí (má vždy minimálně jednu). Za každou správnou odpověď je přibližně poměrná část bodového hodnocení otázky, za každou špatnou odpověď jsou -2 body. Není povoleno používat dodatečné materiály.

• [Skrýt parametry odpovědníku.](#) ▶

Základní informace

- počet otázek: 16
- čas: bez omezení
- výsledky naleznete v poznámkovém bloku
 - pouze jeden nejnovější výsledek

Průchody

- nelze odpovídat (skenovací písemky)

Prohlídka odpovědí

- přístupná po zodpovězení

[Další nastavení](#) ▼

Zeleně jsou vyznačeny správné odpovědi.

```
1. #include <iostream>
#include <vector>
#include <memory>
using namespace std;
class Base {};
class Derived : public Base {
public:
    vector<int> data;
    Derived() : data(100) {}
};
int main() {
    Derived d; // 1
    auto ptr = make_unique<Derived>(); // 2
    unique_ptr<Base> ptr2 =
        make_unique<Derived>(); // 3
}
```

Pro výše uvedený kód platí:

- ☐ Když smažeme řádky 1 a 3, kód bude obsahovat memory leak nebo nedefinované chování.
- ☐ *Kód obsahuje memory leak nebo nedefinované chování, ale když přidáme do třídy **Base** veřejný virtuální destruktork, kód už nebude obsahovat memory leak ani nedefinované chování.
- ☐ Kód neobsahuje žádný memory leak nebo nedefinované chování.
- ☐ Když smažeme řádky 2 a 3, kód bude obsahovat memory leak nebo nedefinované chování.
- ☐ Kód obsahuje memory leak nebo nedefinované chování, ale když přidáme do třídy **Base** veřejný defaultní destruktork, kód už nebude obsahovat memory leak ani nedefinované chování.
- ☐ *Když smažeme řádky 1 a 2, kód bude obsahovat memory leak nebo nedefinované chování.

body = null = 0

```

2. #include <iostream>
using namespace std;
class A {
public:
    virtual void f() { cout << "Af"; }
    virtual void g() { cout << "Ag"; }
    void h() { f(); g(); }
};
class B : public A {
public:
    void f() { cout << "Bf"; }
    void g() const { cout << "Bg"; }
    void h() { cout << "Bh"; }
};
int main() {
    B b;
    A& ref = b;
    ref.h();
}

```

Pro výše uvedený kód platí:

- ☒ ☒ *Vypíše „BfAg“.
- ☐ Vypíše „BfBg“.
- ☐ Vypíše „AfAg“.
- ☐ Vypíše „Bh“.
- ☐ Vypíše „AfBg“.
- ☐ Nezkompiluje se, protože metoda h ve třídě B se pokouší překrýt metodu h ze třídy A, která není virtuální.

body = 5 = 5

```

3. #include <algorithm>
#include <iostream>
#include <vector>
int main() {
    using namespace std;
    vector<int> v;
    v.reserve(5);
    auto it1 = v.begin() + 1;
    auto it2 = v.end() - 2;
    fill(it1, it2, 7);
    vector<int> z = v;
    z[3] = 2;
    v.push_back(9);
    for (auto x : v) {
        cout << x << " ";
    }
    cout << "\n";
    for (auto& x : z) {
        cout << x << " ";
    }
    cout << endl;
}

```

Pro výše uvedený kód platí:

- ☐ Program obsahuje syntaktickou chybu, direktivu using namespace není možno použít uvnitř funkce.
- ☐ Na druhém řádku výstupu bude „0 7 7 12 0“.
- ☐ *Program obsahuje nedefinované chování, některý z iterátorů it1 nebo it2 totiž ukazuje mimo hranice vektoru.
- ☒ ☒ Na prvním řádku výstupu bude „0 7 7 0 0 9“.
- ☐ Na prvním a druhém řádku výstupu bude „0 7 7 12 0 9“.

body = -2 = -2

```

4. #include <iostream>
#include <stdexcept>
using namespace std;
class A {};
class B : public A {};
class X {
public:

```

```

X() { cout << "1"; }
~X() { cout << "2"; }
void f() { cout << "3"; }
};
void fun() {
    cout << "4";
    throw B();
    cout << "5";
}
void nuf() {
    X x;
    cout << "6";
    fun();
    x.f();
    cout << "7";
}
int main() {
    cout << "8";
    try {
        cout << "9";
        nuf();
        cout << "0";
    }
    catch (const A& ex) {
        cout << "A" << endl;
    }
    catch (const B& ex) {
        cout << "B" << endl;
    }
}

```

Pro výše uvedený kód platí:

- ☐ Vypíše „89164A“.
- ☒ ~~X~~ Vypíše „891642B“.
- ☐ Vypíše „89164B2“.
- ☐ *Žádná z ostatních uvedených odpovědí není pravdivá.
- ☐ Vypíše „89164A2“.
- ☐ Vypíše „89164B“.

body = -2 = -2

5. Která z následujících tvrzení platí o návrhových principech SOLID:

- ☒ ☒ **Liskov Substitution Principle* tvrdí, že potomek může zastoupit předka.
- ☐ *Open/Closed Principle* tvrdí, že by atributy objektů neměly být soukromé (*private*).
- ☐ *Liskov Substitution Principle* tvrdí, že předek může zastoupit potomka.
- ☐ *Open/Closed Principle* tvrdí, že je lépe třídy zavřít přidáním klíčového slova *final*, aby je nemohl nikdo dále modifikovat.
- ☐ *Single Responsibility Principle* tvrdí, že každá třída by měla obsahovat nejvýše jeden atribut.
- ☒ ☒ **Single Responsibility Principle* tvrdí, že každá třída (modul, funkce) by měla mít na starost pouze jednu věc (tzv. jediný důvod ke změně).

body = 5 = 5

6.

```

#include <iostream>
using std::cout;
void f(const int x) { cout << x; }
void g(int x) { cout << x; }
void h(int& x) { cout << ++x; }
void h(const int& x) { cout << x; }

int main() {
    const int x = 5;
    int y = 7;
    f(x); // A
    g(x); // B
    h(y); // C
    h(9); // D
}

```

Pro výše uvedený kód platí:

- ☐ Zkompiluje se a po spuštění vypíše „5579“.
- ☐ *Zkompiluje se a po spuštění vypíše „5589“.
- ☐ Nezkompiluje se, protože není možné mít dvě funkce stejného jména (zde h), jejichž parametr se liší pouze specifikátorem const.
- ☒ ~~X~~ Nezkompiluje se kvůli chybě na řádku označeném B.
- ☐ Nezkompiluje se kvůli dvěma chybám (na řádcích B a D).
- ☐ Nezkompiluje se kvůli chybě na řádku označeném D.

body = -2 = -2

```
7. #include <iostream>
using std::cout;
class A {
public:
    A() { cout << "1"; }
    ~A() { cout << "2"; }
    A(const A&) { cout << "3"; }
    A& operator=(const A&) {
        cout << "4";
        return *this;
    }
    void f() const { cout << "!"; }
};

void fun(A x) {
    x.f();
}

int main() {
    A a;
    A b{ a };
    A c = a;
    a = b;
    fun(a);
}
```

Pro výše uvedený kód platí:

- ☐ Nelze přesně určit, co bude na výstupu, protože při volání funkce fun může docházet k tzv. copy elision.
- ☐ Za znakem '!' bude na výstupu „222“.
- ☐ Výstup bude končit znakem '! '.
- ☐ Před znakem '!' bude na výstupu „13144“.
- ☐ *Za znakem '!' bude na výstupu „2222“.
- ☐ Před znakem '!' bude na výstupu „131443“.

body = null = 0

```
8. #include <iostream>
#include <string>
#include <vector>
using namespace std;
void print(const vector<string>& v) {
    for (auto& s : v) { cout << s; }
    cout << '\n';
}

int main() {
    vector<string> v = { "A", "B", "C" };
    vector<string>& w = v;
    string& s = w[1];
    s = v[0];
    print(v);
    s = "X";
    print(v);
}
```

Pro výše uvedený kód platí:

- ☒ ~~✓~~ *Na prvním řádku výstupu bude „AAC“.
- ☐ Na druhém řádku výstupu bude „ABC“.

- ☐ Na druhém řádku výstupu bude „XBC“.
- ☐ Na prvním řádku výstupu bude „ABC“.
- ☐ Není možno přesně říct, co bude na výstupu, protože jsme zapomněli provést vyprázdnění bufferu proudu cout (např. pomocí endl nebo flush).
- ☒ ☒ *Na druhém řádku výstupu bude „AXC“.

body = 5 = 5

```
9. #include <iostream>
using namespace std;
class Val {
public:
    Val(int x) : value(x) {}
private:
    int value;
    friend Val operator+(const Val& a,
                        const Val& b) {
        cout << a.value << "+" << b.value << "|";
        return { a.value + b.value };
    }
    friend ostream& operator<<(ostream& out,
                             const Val& x) {
        return out << x.value;
    }
};

int main() {
    Val a(3);
    Val b = 7;
    Val c = 1 + 4;
    cout << (a + b + c + 0) << endl;
}
```

Pro výše uvedený kód platí:

- ☐ Vypíše „1+4|3+7|10+5|15+0|15“.
- ☐ Není možno přesně určit, co vypíše, protože pořadí vyhodnocování operátoru + není přesně definováno.
- ☐ Vypíše „3+7|5+0|10+5|15“.
- ☐ Vypíše „3+7|10+5|15“.
- ☐ Nezkompiluje se, protože operátory + a << jsou deklarovány jako soukromé (private).
- ☒ ☒ *Vypíše „3+7|10+5|15+0|15“.

body = 5 = 5

```
10. #include <iostream>
using std::cout;
class A {
public:
    A() { cout << "A"; }
    ~A() { cout << "~A"; }
};
class X {
public:
    X() { cout << "X"; }
    ~X() { cout << "~X"; }
};
class Y : public X {
    A a;
public:
    Y() { cout << "Y"; }
    ~Y() { cout << "~Y"; }
};
int main() {
    Y y;
    cout << "|";
}
```

Pro výše uvedený kód platí:

- ☒ ☒ *Před znakem ' | ' bude na výstupu „XAY“.
- ☐ *Za znakem ' | ' bude na výstupu „~Y~A~X“.

- ☐ Výstup bude končit znakem ' | '.
- ☐ Před znakem ' | ' bude na výstupu „AYX“.
- ☐ Za znakem ' | ' bude na výstupu „~X~A~Y“.
- ☒ ~~X~~ Za znakem ' | ' bude na výstupu „~X~A“; destruktory ~Y se nezavolá, protože destruktory ~X není virtuální.

body = 0.5 = 0.5


```
11. #include <iostream>
using std::cout;
class Object {
public:
    void print() const { cout << "C"; }
    void print() { cout << "N"; }
};

void f(const Object& obj) {
    obj.print();
}

void f(Object& obj) {
    obj.print();
}

int main() {
    Object x;
    const Object& y = x;
    f(x);
    f(y);
    f(Object());
}
```

Pro výše uvedený kód platí:


- ☐ Nelze říct s jistotou, co vypíše, protože výstup nebyl ukončen koncem řádku.
- ☐ Vypíše „NCN“.
- ☐ Vypíše „NC“. Na posledním řádku funkce main totiž není volání funkce, ale deklarace funkce.
- ☐ Nezkompiluje se. Poslední řádek funkce main totiž není syntakticky správně.
- ☐ Vypíše „NNN“.
- ☒  *Vypíše „NCC“.

body = 5 = 5

```
12. #include <fstream>
#include <string>
class File {
    std::ofstream log;
public:
    File(const char* file) : log(file) {
        log << "F";
    }
    void print() {
        log << "p";
    }
    ~File() {
        log << "~F";
    }
};

int main() {
    File test("test.txt");
    test.print();
}
```

Předpokládejme, že soubor test.txt v aktuálním adresáři existuje a je čitelný i zapisovatelný pro aktuálního uživatele. Co bude obsahem tohoto souboru po spuštění výše uvedeného kódu?

- ☐ Tento kód se nezkompiluje, protože konstruktor třídy File má parametr typu const char* a my mu předáváme řetězcový literál typu std::string.
- ☒  *Soubor bude obsahovat jen „Fp~F“.
- ☐ Nelze přesně říct, protože výstupní proud log není správně uzavřen (destruktory ~F měl správně obsahovat ještě


```
log.close();
```

- ☐ Soubor bude obsahovat to, co obsahoval před spuštěním kódu, a navíc „Fp~F“.
- ☐ Soubor bude obsahovat jen „Fp“ (výstup v destrukturu selže, protože tou dobou už je proud `log` uzavřen).
- ☐ Soubor bude obsahovat jen „p~F“ (výstup v konstrukturu selže, protože tou dobou proud `log` ještě nebyl otevřen).

body = 5 = 5

13. Která z následujících tvrzení o správě paměti a zdrojů v C++ jsou pravdivá?

- ☐ Operátor `new` dělá totéž, co standardní Cěčková funkce `malloc`.
- ☐ Operátor `delete` dělá totéž, co standardní Cěčková funkce `free`.
- ☐ Moderní správa paměti pomocí `unique_ptr` používá k uvolňování paměti tzv. garbage collector.
- ☐ Při alokaci pole hodnot pomocí `new int[100]()` zůstanou hodnoty v poli neinicializované.
- ☒ ☒ *Používání `unique_ptr` je z hlediska rychlosti běhu programu stejně efektivní jako používání ruční alokace/dealokace paměti pomocí `new` a `delete`.
- ☐ Dynamická alokace `new T` a `new T()` znamená vždy totéž, nezávisle na typu `T`.

body = 5 = 5

14. Která z následujících tvrzení o referencích a ukazatelích jsou pravdivá?

- ☐ K rozdílu mezi referencemi a ukazateli patří to, že reference můžeme používat pouze s objektovými typy (třídami), zatímco ukazatele můžeme používat se všemi typy včetně těch primitivních.
- ☒ ☒ *Reference se nedá přeměrovat, tj. není možné změnit, na který objekt se odkazuje.
- ☐ Je možné definovat neinicializované reference, ale nedoporučuje se to používat, protože to může vést k problémům s pamětí.
- ☐ Je-li ukazatel neinicializovaný, pak je to totéž, jako by byl inicializovaný na `nullptr`.
- ☒ ☒ *Atributy objektů, které jsou referenčního typu, nemůžeme inicializovat v těle konstrukturu. Musíme vždy použít buď inicializační sekci nebo inicializaci přímo v definici třídy.
- ☐ *Máme-li ukazatel deklarovaný jako `const Object* ptr`, pak jej můžeme přeměrovat na jiný objekt.

body = 3.34 = 3.34

15. Mějme třídu deklarovanou takto:

```
class Thing {
    static int x;
    int y;
public:
    void f();
    static void g();
    friend void h();
};
```

Která z následujících tvrzení platí?

- ☐ Kdokoli může přistoupit ke statickému atributu `x` pomocí `Thing::x`.
- ☐ Funkce `h` může přistoupit k atributu `y` pomocí `Thing::y`. Ve funkci `h` přitom nemusí existovat žádný objekt typu `Thing`.
- ☐ Kdokoli může volat metodu `f` pomocí `Thing::f()`. K tomuto volání není třeba mít objekt typu `Thing`.
- ☒ ☒ *Funkce `h` může přistoupit ke statickému atributu `x` pomocí `Thing::x`. Ve funkci `h` přitom nemusí existovat žádný objekt typu `Thing`.
- ☐ Mám-li objekt deklarovaný jako `Thing thing`, mohu na něm volat `thing.h()`.
- ☐ *Kdokoli může volat statickou metodu `g` pomocí `Thing::g()`. K tomuto volání není třeba mít objekt typu `Thing`.


body = 2.5 = 2.5

16.

```
#include <iostream>
using std::cout;
```

```
template <typename T>
class A {
    const T t{};
public:
    void f() const { t.f(); }
    void g() { t.g(); }
};
class X {
public:
    void f() { cout << "Xf"; }
    void g() { cout << "Xg"; }
};
int main() {
    A<X> x;    // 1
    A<int> y;  // 2
    x.f();    // 3
    x.g();    // 4
}
```

Jaký nejmenší počet řádků z těch označených čísly 1 až 4 musíme ve výše uvedeném kódu smazat, aby se zkompiloval? Jaký pak bude výstup programu?

- ☐ Je třeba smazat minimálně jeden řádek, a to 3. Výstup pak bude „Xg“.
- ☒  Je třeba smazat minimálně dva řádky, a to 3 a 4. Výstup pak bude prázdný.
- ☐ Je třeba smazat minimálně jeden řádek, a to 4. Výstup pak bude „Xf“.
- ☐ Je třeba smazat minimálně jeden řádek, a to 2. Výstup pak bude „XfXg“.
- ☐ Je třeba smazat minimálně dva řádky, a to 2 a 3. Výstup pak bude „Xg“.
- ☐ Je třeba smazat minimálně tři řádky, a to 2, 3 a 4. Výstup pak bude prázdný.

body = 5 = 5