Organizace, úvod do C++, reference

PB161 Programování v jazyce C++

Nikola Beneš

20. února 2024

There are only two kinds of languages: the ones people complain about and the ones nobody uses. *(Bjarne Stroustrup)*

https://www.stroustrup.com/quotes.html

Cíle předmětu

- ukázat základní prvky C++
  - moderní C++ podle standardu C++20
- podpořit praktické programátorské schopnosti
  - intenzivním programováním
  - s důrazem na kvalitu kódu

Organizace předmětu

Hlavní studijní materiál: Učební text

- organizační informace (kapitola A)
- sbírka příkladů
- blokové větší úkoly

Cvičení

- přípravné příklady (typ **p**) – vyřešit do soboty před cvičením
- rozšířené příklady (typ **r**) – řešeny na cvičeních (body za aktivitu)

Komunikace

- diskuse na cvičení, přednášce
- diskusní fórum v ISu

Bodování, hodnocení

- *přečtěte si organizační informace!*

Dokumentace

http://cppreference.com

Nástroje

Překladač – GCC (verze 10 a vyšší)

- binárka `g++`, nikoli `gcc`
- použití: `g++ -std=c++20 -Wall -Wextra soubor.cpp`
  - vytvoří binárku `a.out`
    (možno změnit pomocí `-o jméno_binárky`)
  - `-Wall -Wextra` zapínají varování překladače
- alternativně můžete použít `clang++` dostatečně nové verze

Statická analýza – clang-tidy

- (balík `clang` nebo `llvm`)
- odhalování podezřelých konstrukcí / potenciálních chyb
- „varování navíc“
- použití: `clang-tidy soubor.cpp -- -std=c++20`
- vyžaduje konfigurační soubor `.clang-tidy`

Nástroje

Analýza programu za běhu – valgrind

- odhalování různých problémů (zejména správy paměti)
  - použití neinicializované paměti
  - čtení/zápis mimo alokovanou paměť
  - únik paměti (memory leak)
- použití `valgrind --leak-check=full cesta_k/binárce`

Nástroje na serveru `aisa`

- `module add gcc-13.1; module add llvm-16.0.6`
- příklady ve sbírce obsahují `makefile`
  - stačí použít `make`
  - řešení se zkompiluje, zkontroluje pomocí `clang-tidy` a spustí pod `valgrind`em

Plán semestru

Blok 1

- základy syntaxe a sémantiky C++
- životní cyklus a vlastnictví
- složené typy, metody a operátory

Blok 2

- práce s pamětí a zdroji, ukazatele, princip RAII
- realizace OOP v C++, výjimky
- anonymní funkce, lexikální uzávěry (lambdy)

Blok 3

- součtové typy
- knihovna algoritmů
- práce s řetězci, vstup a výstup

Předpoklady

Základní schopnost programování a algoritmizace

- na úrovni kurzů IB111, IB002

Základní znalost nízkoúrovňového programování

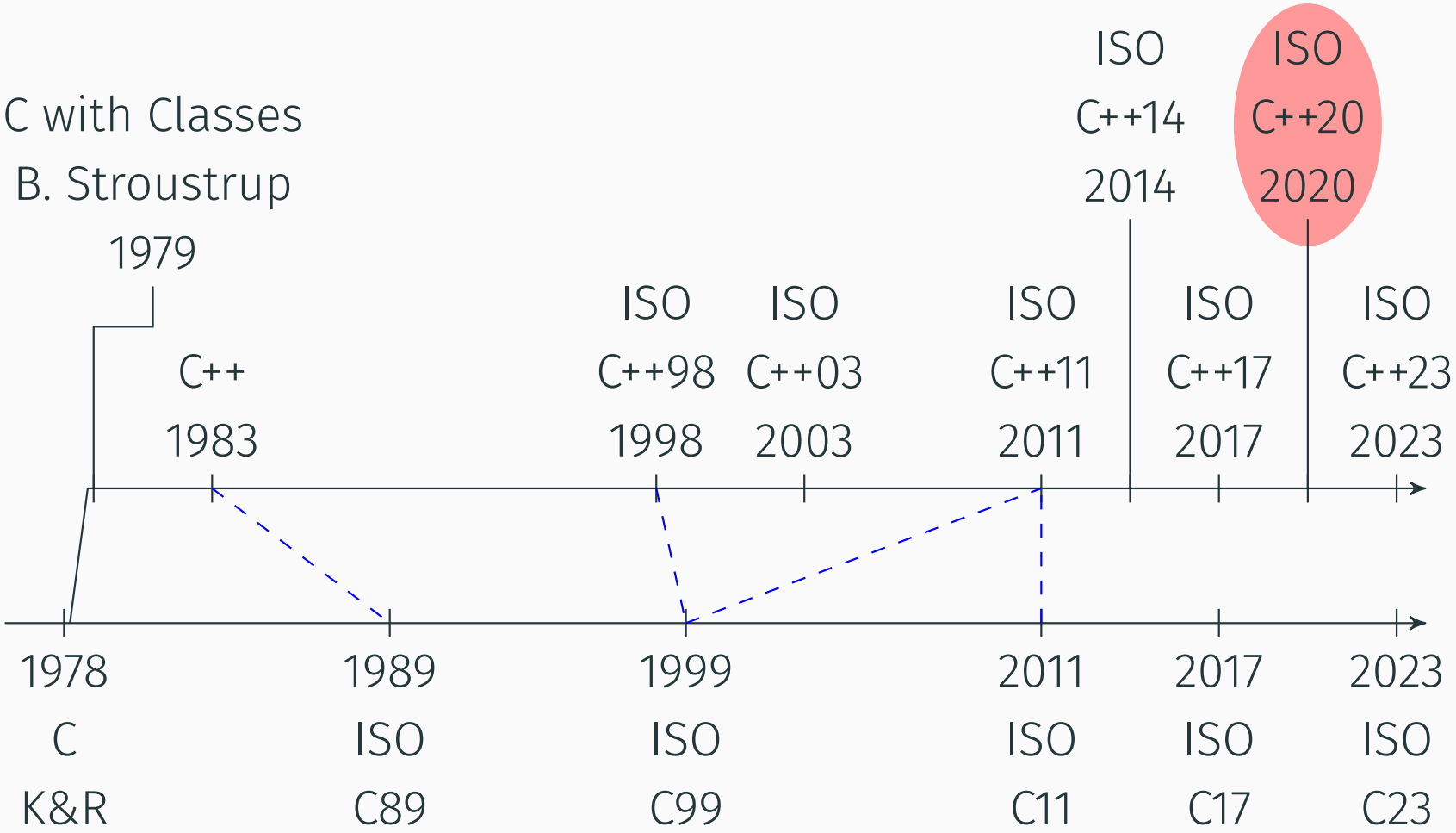
- na úrovni kurzu PB071
- povědomí o paměti, zásobník vs. halda

Znalost základů OOP a principů programovacích jazyků

- na úrovni kurzu PB006
- volání hodnotou vs. volání odkazem
- časná vs. pozdní vazba (virtuální metody)

Programovací jazyk C++

Historie a vývoj



Charakteristika C++

- imperativní, staticky typovaný jazyk
- objektově-orientovaný; *hodnotová sémantika*
- s funkcionálními prvky
- podporuje generické programování a metaprogramování
- *částečně* zpětně kompatibilní s C
- rozsáhlá standardní knihovna
- nové standardy přinášejí významné změny

Charakteristika C++

- přímý vztah kódu a hardware
- abstrakce, které (téměř) nic nestojí

https://twitter.com/timur_audio/status/1227282748685115393

- neplatíte za to, co nepoužíváte
- co používáte, je alespoň tak rychlé, jako co byste napsali sami

https://en.cppreference.com/w/cpp/language/Zero-overhead_principle

https://isocpp.org/wiki/faq

Charakteristika C++

- vysoká rychlost kódu
- rozsáhlé možnosti metaprogramování (tvorba abstrakcí)
- **RAII** (deterministická správa zdrojů)
- jiný pohled na sémantiku objektů
    - ve srovnání s většinou v současnosti populárních jazyků

- vhodné pro
    - větší projekty
    - systémové aplikace
    - rychlou grafiku
    - embedded zařízení

První program

hello.cpp

```cpp
#include <iostream>
#include <string>

int main() {
    std::cout << "What is your name? ";
    std::string name;
    std::cin >> name;
    /* try instead: */
    // std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!\n";
}
```

Poznámka: O typu `std::string` a vstupu/výstupu bude řeč později.

Použití hlavičkových souborů

- `#include <jméno_souboru>`
- `<...>` hledá mezi systémovými soubory, `"..."` lokálně
- čistě textové vložení
  - standard C++20 obsahuje *moduly*, ale podpora je zatím slabá
- soubory ze standardní knihovny nemají příponu
  - vlastní hlavičkové soubory: typicky `.h`, `.hpp`, `.hh` apod.

fun.cpp

Podprogramy (funkce)

- hlavička funkce:
  `typ jméno_funkce(formální argumenty)`
  - typ `void` pro funkce, které nic nevrací
- seznam argumentů ve tvaru `typ jméno`
- tělo funkce – blok příkazů
- vrácení hodnoty z funkce – příkaz **`return`**

```
int clamp(int value, int low, int high) {
    return value < low ? low
         : value > high ? high
         : value;
}
```

Funkce `main` – speciální, stejně jako v C

Jednoduché typy

Pravdivostní hodnoty – `bool`

- literály **`true`**, **`false`**

Celočíselné typy

- `int`
  - základní typ pro celá čísla
  - znaménkový typ, rozsah dle platformy
  - (různé modifikátory: `unsigned`, `long` atd.)
- `std::size_t`
  - neznaménkový knihovní typ pro velikosti objektů, polí apod.
  - hlavičkový soubor `<cstddef>` (nebo jiné, viz dokumentaci…)
- `std::int8_t`, `std::uint8_t`, `std::int16_t`, …
  - knihovní typy pro celá čísla s konkrétní bitovou šířkou
  - předpona `u` – neznaménkový typ
  - hlavičkový soubor `<cstdint>`

Celočíselné literály

- zapsané desítkově: `42`, `17`, …
- zapsané osmičkově: `052`, `021`, …
- zapsané šestnáctkově: `0x2a`, `0x11`, …
- zapsané dvojkově: `0b101010`, `0b10001`, …

- typ je `int` nebo vyšší, pokud je hodnota mimo rozsah
- suffix `u` znamená neznaménkový typ (modifikátor `unsigned`)

- (suffix `zu` pro `std::size_t` až od C++23)

Jednoduché typy

Čísla s plovoucí řádovou čárkou (floating-point)

- `double`
  - běžně používaný typ pro floating-point čísla
  - formát IEEE-754 binary64 (pokud to platforma podporuje)
- `float` (binary32), `long double` (dle platformy)
- literály s desetinnou tečkou: `3.14`, `1.0`, …
  - implicitně typu `double`
  - jiné typy pomocí suffixů `f` nebo `l`

Znaky

- `char` (typicky jednobajtový)
- `char32_t` (UTF-32 codepoint) a jiné; více o nich později
- literály v apostrofech: `'a'`, `'\n'`, …

Typové konverze

Implicitní

- mezi různými číselnými typy
- konverze na `bool` – nula na `false`, ostatní na `true`

Explicitní

- `static_cast<typ>(výraz)`
  - základní konverze, *většinou* bezpečná
- `(typ) výraz` (možná znáte z C)
  - vždy nebezpečné, nepoužívejte
  - varování `-Wold-style-cast`
- `typ(výraz)`
  - stejný význam jako výše, jen pro „jednoslovné“ typy
  - považujeme za OK pro číselné typy, `char`, `bool`, (volání konstruktoru); jinak nepoužívejte

více o typových konverzích později

Proměnné, inicializace

scope.cpp

Deklarace (hodnotové) proměnné

- bez inicializace: `typ jméno;`
  - u jednoduchých typů – neinicializovaná (lokální) proměnná
  - (u složených typů záleží na konstruktoru, viz později)
- s inicializací: `typ jméno = výraz;`
  - tohle není přiřazení
- vícenásobná deklarace: jména oddělená čárkami

Rozsah platnosti proměnné (scope)

- lokální proměnné: od místa deklarace do konce akt. bloku
- deklarace uvnitř některých příkazů (za chvíli): do konce příkazu
- parametry funkce: od hlavičky do konce těla funkce
- globální proměnné: od místa deklarace do konce souboru
- (jmenné prostory, třídy, ...)
- hledání jména (lookup): od vnitřních rozsahů k vnějším

Výrazy, přiřazení

Výrazy

- literály, proměnné
- volání funkcí (skutečné parametry jsou výrazy)
- aplikace operátorů (operandy jsou výrazy)
- ...
- výraz zakončený `;` je jednoduchý příkaz

Přiřazení

- `=` je operátor
- `a = b` je výraz
  - vrací *referenci* na `a` (viz dále)
  - vedlejší efekt: modifikuje `a`
  - `a` musí být modifikovatelná *l-hodnota*
    - proměnná, položka složeného datového typu, ...

Podmínkové příkazy

if.cpp

Příkaz `if`

- `if (podmínka) příkaz`
  - podmínka může být i deklarace
- volitelně následováno `else příkaz`
- od C++17 varianta `if (inicializace; podmínka)`
  - buď deklarace nebo jednoduchý příkaz
- rozsah deklarovaných proměnných – do konce příkazu `if`
  - včetně případných `else` větví

```
if (int x = answer(); x < 10) {
    std::cout << "small " << x << '\n';
} else {
    std::cout << "large " << x << '\n';
}
```

Podmínkové příkazy

Příkaz `switch`

- `switch (výraz) příkaz`
  - výraz může být i deklarace
  - příkaz je typicky blok
- od C++17 varianta `switch (inicializace; výraz)`
  - jako u `if`
- uvnitř příkazu návěští `case konstanta:`, `default`
  - vykonávání programu skočí na odpovídající `case` nebo `default`, pokud žádný není
  - dále se pokračuje v bloku bez ohledu na návěští
  - opuštění příkazu `switch` pomocí `break`
- rozsah deklarovaných proměnných – do konce příkazu `switch`

Příkazy cyklu

while.cpp

Příkaz `while`

- `while (podmínka) příkaz`
  - podmínka může být i deklarace
- rozsah deklarovaných proměnných – do konce příkazu `while`
- podmínka se testuje vždy před vykonáním příkazu
  - má-li hodnotu `false`, cyklus se ukončí

Příkaz `do while`

- `do příkaz while (podmínka);`
  - podmínka *nesmí* být deklarace
- podmínka se vyhodnocuje vždy *po* vykonání příkazu

Příkazy cyklu

Příkaz `for` obecný

- `for (inicializace; podmínka; iterace) příkaz`
  - inicializace je deklarace nebo výraz
  - iterace je libovolný výraz
- sémantika:

```
{
    inicializace;
    while (podmínka) {
        příkaz
        iterace;
    }
}
```

Příkaz `for` po prvcích (range-based) – uvidíme příště

for.cpp

Příkaz `break`

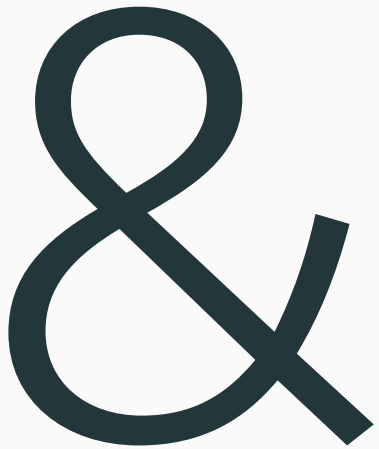
- ukončí aktuální cyklus (nebo příkaz **`switch`**)

Příkaz `continue`

- skočí na konec těla cyklu
- v cyklu `for` se tedy po skoku ještě provede *iterace*

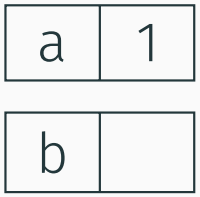
```
for (int i = 0; i < 10; ++i) {
    if (i == 3)
        continue;
    std::cout << i << '\n';
}
```

Hodnotová sémantika, reference a const

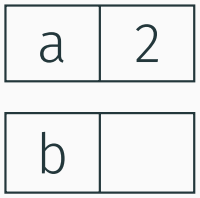


Hodnotová sémantika

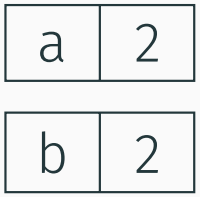
```
int a, b;
a = 1;
```




```
a = 2;
```




```
b = a;
```

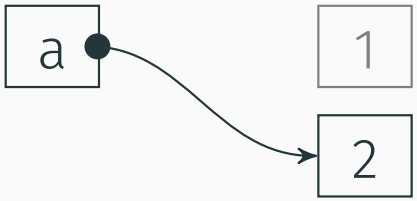


Jazyk C++

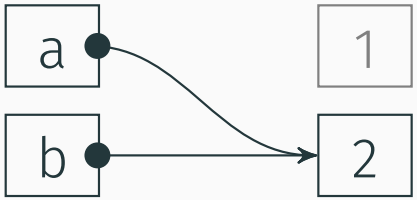
Přiřazení mění hodnotu

```
a = 1
```


```
a = 2
```




```
b = a
```



Jazyk Python

Přiřazení přesměruje odkaz

Hodnotová sémantika

Vazba proměnné na hodnotu

- vazba proměnné na hodnotu („místo v paměti“) je fixní
  - po dobu existence proměnné
  - nemůžeme „přesměrovat“ proměnnou jinam
- hodnota se vytváří inicializací
- přiřazení hodnotu přímo mění
- inicializace i přiřazení tedy vytvářejí *kopie*

Volání funkcí

- hodnoty předaných výrazů se použijí pro inicializaci parametrů

Volání funkcí

Volání hodnotou (call by value)

- hodnota skutečného argumentu funkce se uloží do parametru (formálního argumentu funkce) – jde o *kopii* hodnoty
- modifikace parametru se navenek nijak neprojeví
- (jediný způsob volání funkcí v C)

Volání odkazem (call by reference)

- skutečný argument musí být *l-hodnota*
  - „to, co může stát vlevo u přiřazení“
  - proměnná, položka složeného datového typu, ...
- parametr je svázán přímo s hodnotou skutečného argumentu
- modifikace parametru přímo mění hodnotu předané entity
- (v C můžeme simulovat pomocí ukazatelů)
- v C++ k tomuto účelu používáme tzv. *reference*

Reference

ref1.cpp

Deklarace nového jména pro existující hodnotu

- `typ& jméno = výraz`
  - výraz musí být *l-hodnota*[1]
    (proměnná, položka složeného typu, ...)
- alias, jiné jméno pro stejnou věc

```
int a = 1;
int b = 2;
int& ref = a;
// whatever happens to ref, happens to a
ref = b;
ref += 7;
```

- reference (`typ&`) můžeme předávat do funkcí a vracet z funkcí
  - reference vrácená z funkce je *l-hodnota*

[1] pokud typ nemá kvalifikátor `const`, viz dále

Předávání parametrů hodnotou vs. referencí

ref2.cpp

```
void by_val(int x) { x += 4; }
void by_ref(int& x) { x += 4; }

int main() {
    int a = 38;
    by_val(a);
    std::cout << a << '\n';
    by_ref(a);
    std::cout << a << '\n';
}
```

ref3.cpp

Na co si dát pozor?

- reference neřeší problém životnosti
- nedržet referenci na objekt, který přestal existovat
  - *dangling reference*
- nevracet z funkce referenci na lokální proměnnou nebo *hodnotový* parametr

```
int& bad() {
    int x = 1;
    return x;
}

int& also_bad(int x) {
    return x;
}
```

```
int& good(int& x) {
    return x;
}

int& also_good(int& x) {
    int& y = x;
    return y;
}
```

const.cpp

Klíčové slovo `const`

- deklaruje záměr neměnit hodnotu (*read-only*)
- použití s hodnotovými proměnnými – deklarace konstant
  - globálních i lokálních

```
const int answer = 42;

int main() {
    for (int i = 0; i < 10; ++i) {
        const int m = i * answer + 10;
        std::cout << m << '\n';
    }
}
```

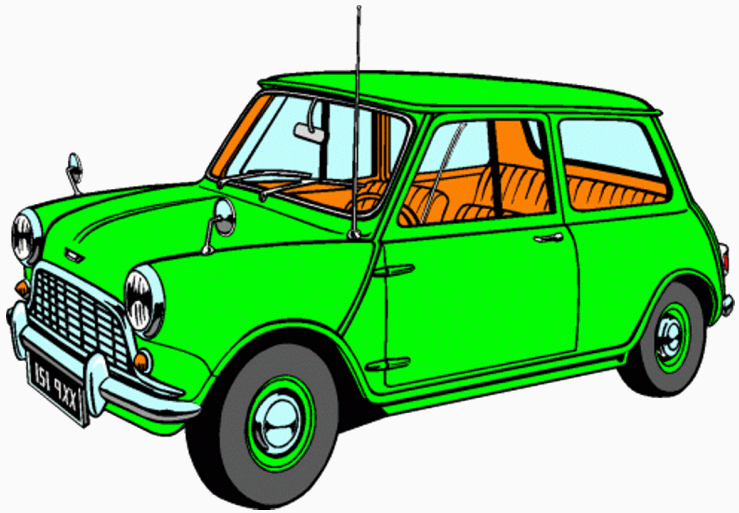
Použití s referencemi

- `const typ&`
- deklaruje záměr neměnit hodnotu *skrze tuto referenci*
  - tj. reference jen pro čtení
- (může se vázat i k hodnotám, které nejsou *l-hodnoty*)

```
int a = 1;
int& ref = a;
const int& cref = a;
```

- použití při volání funkcí
  - nevytváříme kopii
  - vhodné pro předávání **větších objektů** (složených typů)
  - nedává smysl pro jednoduché hodnoty a jiné malé objekty
  - uvidíme příště

Automatická dedukce typu



auto.cpp

Klíčové slovo `auto`

- smíme použít místo typu v místě deklarace
- od C++14 i jako návratový typ funkcí
- od C++20 i pro parametry funkcí
  - *generické funkce*
- skutečný typ dedukuje kompilátor
  - podle inicializace
  - podle výrazu za `return`
  - podle místa volání

- samotné `auto` znamená vždy *hodnotu*, ne referenci
- `auto&` znamená referenci
- `const auto&` znamená konstantní referenci

auto.cpp

Klíčové slovo `decltype`

- chceme-li deklarovat proměnnou stejného typu jako jiná proměnná nebo výraz
- použití zejména v generických funkcích

```
auto slow_pow(auto num, int exp) {
    decltype(num) result = 1;
    for (; exp > 0; --exp) {
        result *= num;
    }
    return result;
}
```

Nespecifikované a nedefinované chování

Chování závislé na implementaci

Implementačně závislé chování (implementation-defined behaviour)

- závisí na konkrétní implementaci překladače
- jeho efekt musí být dokumentován
- příklad: počet bitů v bajtu, velikosti číselných typů, ...

Nespecifikované chování (unspecified behaviour)

- závisí na konkrétní implementaci překladače
- jeho efekt nemusí být dokumentován
- příklad: pořadí vyhodnocování skutečných argumentů funkce

Standard definuje *meze*, v nichž se chování implementace pohybuje.
Na konkrétní implementaci není vhodné se spoléhat.

Nedefinované chování – *undefined behaviour*

https://en.cppreference.com/w/cpp/language/ub

- efekt může být úplně libovolný
- překladač smí předpokládat, že takové chování nenastává
  - to umožňuje některé optimalizace
  - souvisí se zero-overhead principle
- příklady:
  - indexace mimo hranice pole
  - přístup do uvolněné paměti
  - čtení z neinicializované proměnné
  - použití reference na již neexistující hodnotu
  - nekonečný cyklus bez vedlejších efektů
  - přetečení u znaménkových typů
  - porušení vstupní podmínky funkce standardní knihovny

Program, který obsahuje nedefinované chování, je *vždy nekorektní*.

Testování

Testování

Programy obsahují chyby

- je to tak

Testování programů

- důležitá součást vývoje
- různé úrovně testování
- *unit testing* – testování malých jednotek kódu

Nástroje pro unit testing

- v základu si vystačíme bez nich
- velká řada různých frameworků
  - např. https://github.com/catchorg/Catch2

Ověření platnosti tvrzení

Makro `assert`

- hlavičkový soubor `<cassert>`
- `assert(podmínka)`
  - tvrzení, že v tomto místě programu podmínka platí
- použití:
  - pro jednoduché testování
  - formulace vstupních / výstupních podmínek
  - jiná pomocná tvrzení (invarianty cyklů, ...)
- při neplatnosti podmínky program selže
  - při kompilaci v režimu „debug“ (implicitně)

Poznámka: Znaky čárka `,` uvnitř podmínky musí být chráněny závorkami. (Příklad uvidíme někdy později.)

Příklad použití assert

test_example.cpp

```
auto digit_sum(auto num, auto base) {
    // preconditions
    assert(num >= 0 && base >= 2);

    /* ... */

}

int main() {
    assert(digit_sum(42, 10) == 6);
    assert(digit_sum(1337, 10) == 14);
    assert(digit_sum(1337, 2) == 6);
    assert(digit_sum(1330, 11) == 30);
}
```

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

(Edsger W. Dijkstra)