

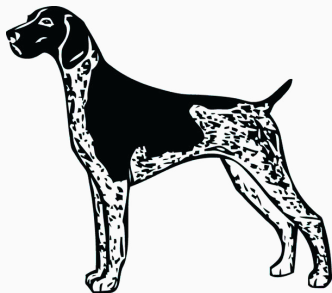
SPRÁVA PAMĚTI; UKAZATELE OBYČEJNÉ I CHYTRÉ

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

14. března 2023

UKAZATELE



Datové ukazatele `typ*`

- hodnotou je
 - adresa objektu
 - adresa *těsně* za objektem (typické použití pro pole)
 - žádná adresa (*null*) – ukazatel **`nullptr`**
 - nevalidní adresa (objektu po konci života)
- `typ` může být i **`void`**
- místo typu můžeme použít **`auto`**

Funkční ukazatele – uvidíme později

Členské ukazatele – mimo záběr předmětu

Poznámka: klasickým ukazatelům se někdy říká „syrové“ (*raw*).

Adresa objektu – unární &

- jen pro l-hodnoty
- vrací hodnotu typu ukazatel
- (& může být přetíženo, proto máme `std::addressof`)

Dereference – unární *

- jen pro validní ukazatele na objekt
- vrací (l-hodnotovou) *referenci* na odkazovaný objekt
- `ptr->x` je ekvivalent `(*ptr).x`

Rovnost ==

- oba jsou *null* nebo oba reprezentují stejnou adresu

Uspořádání < apod.

- specifikováno pouze pro ukazatele dovnitř stejného objektu (pole, záznamu, ntice)
- menší ukazatel je ten s adresou dříve deklarované položky
 - pro pole: podle indexu
- ukazatel *těsně za* objekt je větší než ukazatel na tento objekt nebo libovolnou jeho položku

Aritmetika

- pro ukazatele dovnitř polí¹
- přičtení / odečtení celého čísla, ++, --
 - posun v poli o daný počet položek
 - je možné se posunout *těsně za* pole
- odečtení dvou ukazatelů (dovnitř stejného pole)
 - vzdálenost položek v poli (se znaménkem)
 - výsledek typu `std::ptrdiff_t` (znaménkové celé číslo)
- „indexování“ ukazatele – `ptr[i]` je ekvivalent `*(ptr + i)`

¹ostatní objekty se pro účely ukazatelové aritmetiky chovají jako pole o velikosti 1

Použití s `const`

- rozdíl mezi `const` „před hvězdičkou“ a „za hvězdičkou“
- `const typ*` ukazatel – dereference vrátí `const typ&`
 - nesmíme měnit odkazovaný objekt
 - ale ukazatel jako takový můžeme (přiřazením, ++ apod.)
- `typ* const` ukazatel – dereference vrátí `typ&`
 - smíme měnit odkazovaný objekt
 - ukazatel měnit nesmíme
 - (není moc užitečné, typicky spíš chcete referenci)
- `const typ* const` – nesmíme měnit nic

Implicitní konverze

- `nullptr` na libovolný ukazatelový typ
- `typ*` na `const typ*`
- libovolný ukazatel na `bool`
 - `false` pro `null`, `true` jinak
- `typ*` na `void*`; `const typ*` na `const void*`

Explicitní konverze – pomocí `static_cast`

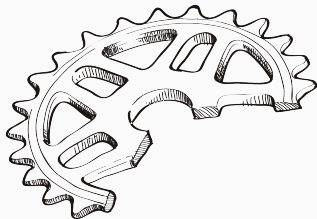
- libovolná implicitní
- `void*` na `typ*` (případně s `const`)
 - je třeba zaručit, že skutečně ukazuje na objekt daného typu (resp. dle pravidel pro *type aliasing* – mimo záběr předmětu)

Poznámka: o konverzích při dědičnosti bude řeč později.

Používat reference nebo ukazatele?

- „*use references if you can, pointers if you must*“
- dávejte přednost použití referencí
- ukazatele jen tam, kde
 - dává smysl **nullptr** (a jste připraveni se s ním vyrovnat)
 - dává smysl přesměrování v průběhu života ukazatele

ITERÁTORÝ



Motivace

- zobecnění ukazatelů do pole
 - nevlastní položky, jen na ně ukazují
 - „lehké objekty“, je v pořádku je kopírovat
- pro různé druhy kontejnerů, ale nejen pro ně
 - (iterátor pro čtení ze / zápis do souboru, ...)
- umožňují generické algoritmy
 - nezávislé na konkrétní datové struktuře
 - (více o knihovně algoritmů později)
- různé druhy iterátorů podporují různou podmnožinu „ukazatelových“ operací
 - <https://en.cppreference.com/w/cpp/iterator>

Operace

- dereference `*`
 - dereference pro čtení / zápis dle druhu iterátoru
 - nemusí nutně vracet referenci
 - některé druhy navíc i operátor `->`
- inkrement `++`
- typicky také: porovnání `== / !=`
 - minimálně s iterátorem za *konec*²
- další operace podle druhu iterátoru
 - obousměrný (bidirectional): dekrement `--`
 - s náhodným přístupem (random-access): „ukazatelová“ aritmetika `+`, `-`, `[]`, `<` apod.

²od C++20 obecnější zarážka (*sentinel*)

Minimum nutné pro použití cyklu **for** po prvcích

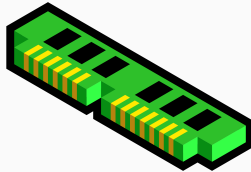
- metody `begin()`, `end()`
- dereference `*`, prefixový inkrement `++`,
porovnání `!=` s iterátorem *za konec* (zarážkou)

Přetížený operátor unární `*`

- typická implementace jako metoda
- vrací (pokud možno) referenci na položku kontejneru

Poznámka: operátor `->` se automaticky nevygeneruje, je třeba jej případně přetížit (složitější, mimo záběr předmětu)

SPRÁVA (DYNAMICKÉ) PAMĚTI



Nízkoúrovňová

- „zděděno“ z C: `std::malloc` / `std::free`
 - v C++ nepoužíváme, jen při nutnosti interagovat s C
- s inicializací: `new` / `delete`
 - v moderním C++ spíše nepoužíváme
 - ale je vhodné vědět, jak funguje
- bez inicializace, s pomocí tzv. alokátorů
 - mimo záběr předmětu

Vysokoúrovňová

- souvisí s konceptem vlastnictví
 - automatická dealokace v rámci „úklidu“
- prostředky standardní knihovny, příp. jiných knihoven
 - `std::vector` a jiné kontejnery
- chytré ukazatele

Operátor **new** – alokace s případnou inicializací

- jeden objekt nebo („Céčkové“) pole objektů
- výsledkem je ukazatel na objekt nebo na první položku pole

```
int* ptr = new int;  
int* array = new int[size];
```

- když se alokace nezdaří, vyhodí výjimku **std::bad_alloc**

Operátor **delete** – destrukce a dealokace

```
delete [] array;  
delete ptr;
```

- musí odpovídat použitému **new**
- nehlídá kompilátor! (někdy ovšem umí dát varování)

Inicializace alokovaných objektů

```
int* ptr1 = new int;    // uninitialised!  
int* ptr2 = new int{ 17 };  
int* ptr3 = new int{}; // initialised to zero  
  
auto* ptr4 = new point; // calls point()  
auto* ptr5 = new point{ 4, 2 };
```

Problémy správy paměti

- *memory leak* (alokovaná paměť, na kterou již nemáme ukazatel)
- zápis do/čtení z nealokované paměti
- čtení z neinicializované paměti
- volání **delete** na špatný ukazatel
- volání **delete** místo **delete []** a naopak
- volání **free** na paměť alokovanou **new**, nebo **delete/delete[]** na paměť alokovanou **malloc**

Motivace pro chytré ukazatele

- chceme ukazatel, který sám provede **delete**, když je potřeba

CHYTRÉ UKAZATELE



Myšlenka

- chceme zřetěžený seznam s automatickou dealokací
- *vlastnictví*: seznam vlastní první uzel; uzel vlastní následníka
- k úklidu použijeme destruktorku (princip RAII)

```
struct node_ptr {  
    node* ptr;  
    node_ptr(node* ptr = nullptr) : ptr(ptr) {}  
    ~node_ptr() { release(); }  
};
```

```
void node_ptr::release() {  
    delete ptr;  
}
```

Kopírování, přesun

- pamatujme na *rule of five!*
- kopírování zakážeme (nedává smysl)
- přesun povolíme (a explicitně implementujeme)

```
node_ptr(node_ptr&& other) noexcept
    : ptr(other.ptr) {
    other.ptr = nullptr;
}
```

```
node_ptr& operator=(node_ptr&& other) noexcept {
    release();
    ptr = other.ptr;
    other.ptr = nullptr;
    return *this;
}
```

... a to je vše.

- napsali jsme si vlastní *chytrý ukazatel*
- vlastní dynamicky alokovanou paměť pro uzel
 - uklízí (dealokuje) v destrukturu
- vlastnictví umíme *explicitně předat* pomocí `std::move`
- *všimněte si*: stále můžeme používat syrové ukazatele
 - tam, kde nechceme přebírat vlastnictví
 - např. v implementaci iterátoru

Chytré ukazatele

- ve standardní knihovně (od C++11³), příp. i jinde (*boost*)
- úklid ve chvíli, kdy už paměť není potřeba (v destruktoru)
- souvisí s konceptem *vlastnictví*

`std::unique_ptr`

- unikátní vlastník alokované paměti
- nulová režie za běhu

`std::shared_ptr`

- sdílené vlastnictví; „poslední zhasne“
- režie spojená s počítáním odkazů; pomalejší

³v C++03 existoval `std::auto_ptr`, ale byl nahrazen `std::unique_ptr`

`std::unique_ptr`

- jednoduchý, ale pokrývá většinu potřeb
- nulová režie za běhu
 - výsledný kód je stejně rychlý⁴ jako s ruční dealokací
- použitelný pro jednotlivé objekty i pole
 - pro dynamická pole ovšem preferujte `std::vector`
- základní princip: unikátní vlastnictví
 - `std::unique_ptr` uvnitř drží ukazatel na alokovanou paměť
 - na konci života objektu se zavolá `delete` na tento ukazatel
 - vlastnictví se nedá sdílet – `std::unique_ptr` se nedá kopírovat, ale může se přesouvat – explicitní *předání vlastnictví*

⁴za předpokladu kompilace s *optimalizacemi*, samozřejmě;
viz <https://godbolt.org/z/zx5EzY8Yx>

Vytvoření ukazatele

- prázdná inicializace – `nullptr`
- inicializace ukazatelem na alokovanou paměť
- lepší řešení (od C++14) – `std::make_unique`

```
auto ptr = std::make_unique<thing>(17);
```

Práce s ukazatelem

- dereference `*`, `->` jako u klasických ukazatelů

```
ptr->do_something();  
run(*ptr);
```

Předání vlastnictví

- přesun; použití `std::move`

```
ptr1 = std::move(ptr2);
```

Uvolnění paměti (destrukce objektu + dealokace)

- automatická na konci života `std::unique_ptr`
- automatická na levé straně při přiřazení
- explicitní pomocí `reset()`

Typová konverze na `bool`

- totéž jako `ptr != nullptr`

Přímý přístup k syrovému ukazateli

- metoda `get()`
- např. potřebujeme-li volat funkci s ukazatelovým parametrem
 - nezapomeňte: preferujte reference před ukazateli

Pozor! Metoda `release()` rovněž vrátí ukazatel, ale zruší vlastnictví (`std::unique_ptr` už jej dále nespravuje); raději ji nepoužívejte!

Jaký typ parametru zvolit?

- hodnota typu `std::unique_ptr`
 - chceme-li přebrat vlastnictví
 - volající se musí vlastnictví explicitně vzdát (`std::move`)
- syrový ukazatel na objekt (`const`) `thing*`
 - volání `fun(ptr.get())`
 - jasné sdělení, že nedochází ke změně vlastnictví
 - pokud dává smysl, aby funkce dostala `nullptr`
- reference na objekt (`const`) `thing&`
 - volání `fun(*ptr)`
 - nejlepší možnost, pokud nehodláme měnit vlastnictví
 - to, že ukazatel není `nullptr`, si zajistí volající

Nevhodné / nedoporučované způsoby volání

- (l-hodnotová) reference na `std::unique_ptr`
 - nedoporučované, volaná funkce může sebrat vlastnictví
- konstantní (l-hodnotová) reference na `std::unique_ptr`
 - nedoporučované
 - funkce smí modifikovat odkazovaný objekt
 - chová se jako `thing* const`
- r-hodnotová reference na `std::unique_ptr`
 - volající musí použít `std::move`
 - ale nemá záruku, že se skutečně vzdal vlastnictví
 - spíše nedoporučované (trochu otázka vkusu)

- je třeba rozmyslet strukturu vlastnictví
 - nesmí být cyklická
 - jeden objekt nesmí mít dva různé vlastníky
 - tj. musí být „stromová“
- příklad: oboustranně zřetěžený seznam
 - (uzly) `next` je `std::unique_ptr`, `prev` je syrový ukazatel
 - (seznam) `first` je `std::unique_ptr`, `last` je syrový ukazatel
 - nebo naopak
- příklad: binární strom
 - vztah „rodič vlastní potomky“
 - `left` a `right` jsou `std::unique_ptr`, `parent` je syrový ukazatel

`std::shared_ptr`

- komplikovanější, založený na počítání odkazů (*reference counting*)
- základní princip: sdílené vlastnictví
 - více různých vlastníků; `std::shared_ptr` udržuje jejich počet
 - kopie `std::shared_ptr` je sdílení (tj. nekopíruje se odkazovaný objekt)
 - zanikne-li poslední vlastník, zavolá se **delete**
- nutná režie s počítáním vlastníků
 - funguje i v paralelním prostředí
 - atomické počítadlo
- ukážeme si jen základní použití, `std::shared_ptr` je složitější (https://en.cppreference.com/w/cpp/memory/shared_ptr)

Vytvoření ukazatele

- podobně jako `std::unique_ptr`
- funkce `std::make_shared`

Práce s ukazatelem

- operátory `*`, `->`

Sdílení / předání vlastnictví

- kopírování – sdílení: `ptr1 = ptr2;`
- přesun – předání: `ptr1 = std::move(ptr2);`

Uvolnění paměti

- sníží-li se počet vlastníků na nulu
 - konec života vlastníka, přiřazení do vlastníka, `reset()`


```
struct node {  
    thing value;  
    std::shared_ptr<node> next;  
};
```

```
int main() {  
    auto n1 = std::make_shared<node>();  
    auto n2 = std::make_shared<node>();  
    n1->next = n2;  
    n2->next = n1;  
}
```

- cyklická závislost (n1 vlastní n2 a obráceně)
- nedojde k uvolnění paměti (a destrukci objektů typu **thing**)

Řešení

- je třeba „rozbít cykly“
 - struktura vlastnictví musí být acyklický graf (*dag*)
 - ostatní ukazatele nic nevlastní
- jedna možnost: syrové ukazatele
 - pokud jejich existence zaručeně nepřezíje datovou strukturu
- druhá možnost: „slabý“ chytrý ukazatel
 - `std::weak_ptr` – mimo záběr předmětu

Problém se špatným vícenásobným vlastnictvím

- konstruktory `std::unique_ptr`, `std::shared_ptr` berou syrový ukazatel a přebírají jeho vlastnictví
- pokud už je ale ukazatel někým vlastněn, dojde k chybě!

```
auto ptr1 = std::make_unique<thing>();  
auto* raw = ptr1.get();  
std::unique_ptr<thing> ptr2{ raw };
```

- destruktory obou vlastníků zavolají **delete**
- totéž v případě `std::shared_ptr`
 - extra informace nutné pro počítání referencí nejsou vázány na syrový ukazatel
- pointa: používejte `std::make_unique`, `std::make_shared`, předávejte vlastnictví (přesouvacím) přiřazením / inicializací

Kdy použít dynamickou alokaci?

- vyhněte se pokud možno „syndromu kladiva“
 - „právě jsem se naučil používat kladivo a všechno v okolí mi připadá jako hřebík“
- potřebujete vůbec dynamicky alokovat?
 - lokální hodnotové proměnné jsou v pořádku
 - hodnotové položky jsou v pořádku
- nenabízí požadovanou funkcionalitu standardní knihovna?
 - `std::vector` nestojí o moc víc než dynamicky alokované pole
 - a lépe se s ním zachází

Kdy a jak použít `std::unique_ptr`?

- dá se určit unikátní vlastník (zodpovědný za úklid)?
 - bude žít alespoň tak dlouho jako alokovaná paměť?
- pokud ano, použijte `std::unique_ptr`
- jasně si rozmyslete, kdo bude *vlastníkem*
- ostatní uživatelé mohou mít syrový ukazatel na objekt (nebo ještě lépe, referenci)
 - vlastník musí přežít všechny ostatní uživatele

Kdy a jak použít `std::shared_ptr`?

- nedá se určit unikátní vlastník
 - (datová struktura, v níž existuje více rovnocenných „rodičů“)
- cykly je třeba přerušit pomocí nevlastnických ukazatelů
- uživatelé, kteří nemají být vlastníky, ale mohou je přežít, musí použít `std::weak_ptr`
- syrové ukazatele pro uživatele, kteří nikdy nepřezijí vlastníky

Kdy a jak použít `new` a `delete`?

- v moderním C++ pokud možno *nikdy* a *nijak*
- je ovšem vhodné rozumět tomu, co dělají
- chcete-li vytvářet novou datovou strukturu
 - rozmyslete se, jestli skutečně potřebujete vlastní správu paměti
 - většinou stačí `std::unique_ptr`, vzácněji `std::shared_ptr`
 - pokud skutečně chcete vlastní správu paměti:
naučte se používat alokátory (nad rámec tohoto předmětu)

A `shared_ptr` represents shared ownership but shared ownership isn't always ideal: By default, it is better if an object has a definite owner and a definite, predictable lifespan. Prefer the following in order: stack or member lifetime (stack variables or by-value member variables); heap allocation with `unique_ptr` unique ownership; and heap allocation via `make_shared` with shared ownership. (C++ FAQ)