

PB071 – Programování v jazyce C

Jaro 2014

Uživatelské datové typy, dynamické
struktury a jejich ladění

Uživatelské datové typy

Uživatelské datové typy

- Známe primitivní datové typy
- Známe pole primitivních datových typů
- Můžeme vytvářet vlastní uživatelské datové typy:
 - enum
 - struct
 - union
 - typedef

enum – výčtový typ

```
enum race_t {  
    elf,  
    human,  
    hobbit  
};  
enum race_t race1 = elf;
```

- Motivace: proměnná s omezeným rozsahem hodnot
 - např. rasa hráče (elf, human, hobit):
 - `const int elf = 0; const int human = 1; const int hobbit = 2;`
 - `int race = human; race = 18; ???`
 - jak zajistit přiřazení jen povolené hodnoty?
- Typ proměnné umožňující omezit rozsah hodnot
 - povolené hodnoty specifikuje programátor
 - kontrolováno v době překladu (pozor, jen pro pojmenované hodnoty)
- Deklarace typu:
 - `enum jméno_typu { pojmenované_hodnoty };`
- Vytvoření proměnné:
 - `enum jméno_typu jméno_proměnné;`
- (Další možné varianty syntaxe)
 - <http://msdn.microsoft.com/en-us/library/whbyts4t%28v=vs.80%29.aspx>

enum – ukázka

```
enum race_t {  
    elf,  
    human,  
    hobit  
};  
enum race_t race1 = elf;
```

- enum nelze vynechat
 - lze vyřešit pomocí nového typu (typedef, později)
- Lze vynechat výčet pojmenovaných hodnot, pokud již bylo zavedeno dříve
- Nelze zavést ve stejném jmenném prostoru stejně pojmenovaný index

enum - detailněji

- V C je enum realizován typem int
 - položky enum jsou pojmenování pro konstanty typu int
- První položka je defaultně nahraditelná za 0
 - druhá za 1, atd.
 - `enum race_t {elf, human, hobit};`
 - `// elf == 0, hobit == 2`
- Index položky lze ale změnit
 - `enum race_t {elf, human=3, hobit};`
 - `// elf == 0, hobit == 4`
- Index položky může být i duplicitní
 - `enum race_t {elf=1, human=1, hobit=-3};`

enum - využití

- Pro zpřehlednění zápisu konstant
 - elf, human a hobit přehlednější než 0, 1, 2
 - přiřazení hodnoty, switch...
- Pro kontrolu rozsahu hodnot
 - pouze při specifikaci pojmenovaným indexem
 - rozsah číselného indexu ale není kontrolován

```
enum race_t {  
    elf,  
    human,  
    hobit  
};  
enum race_t race1 = elf;  
enum race_t race3 = chicken;  
enum race_t race2 = -15;
```

struct - motivace

- Motivace: avatar
 - avatar má několik atributů (nick, energy, weapon...)
- Nepraktické implementační řešení
 - proměnná pro nick, energy, weapon, ...
- Jak předávat avatara do funkce?
 - velké množství parametrů? ☹
 - globální proměnné? ☹
- Jak zachytit svět s více avatary?
 - proměnné s indexy? ☹
- Jak přidávat avatary průběžně?
 - OMG



struct

- Datový typ obsahující definovatelnou sadu položek
- Deklarace typu:
 - **struct** jméno_typu { výčet_položek};
- Vytvoření proměnné:
 - **struct** jméno_typu jméno_proměnné;
 - struct nelze vynechat, výčet položek se opakovaně nespecifikuje
- Velikost proměnné typu **struct** odpovídá součtu velikostí všech položek (+ případné zarovnání)

```
enum weapon_t {sword,axe,bow};  
struct avatar_t {  
    char nick[32];  
    int energy;  
    enum weapon_t weapon;  
};  
struct avatar_t myAvatar = {"Hell", 100, axe};
```

Inicializace struktur

- Při deklaraci proměnné

- `struct` avatar_t myAvatar = {"Hell", 100, axe};
- nelze `myAvatar = {"Hell", 100, axe};`
 - není konstanta typu `struct`

- Po jednotlivých položkách

- `myAvatar.energy = 37; myAvatar.weapon = bow;`
- `strcpy(myAvatar.nick, "PetrS");`

- Pojmenovaným inicializátorem (od C99)

- `struct` avatar_t avatar2 = {.energy=107};

- (Vynulováním paměti)

- `memset(&avatar2, 0, sizeof(struct avatar_t))`

struct – předání do funkce

hodnotou

hodnotou
ukazatele

```
enum weapon_t {sword,axe,bow};
struct avatar_t {
    char nick[32];
    int energy;
    enum weapon_t weapon;
};

void hitAvatar(struct avatar_t avatar, int amount) {
    avatar.energy -= amount;
}

void hitAvatar2(struct avatar_t* pAvatar, int amount) {
    (*pAvatar).energy -= amount;
}

int main(void) {
    struct avatar_t myAvatar = {"Hell", 100, axe};
    hitAvatar(myAvatar, 10);
    hitAvatar2(&myAvatar, 10);

    return EXIT_SUCCESS;
}
```

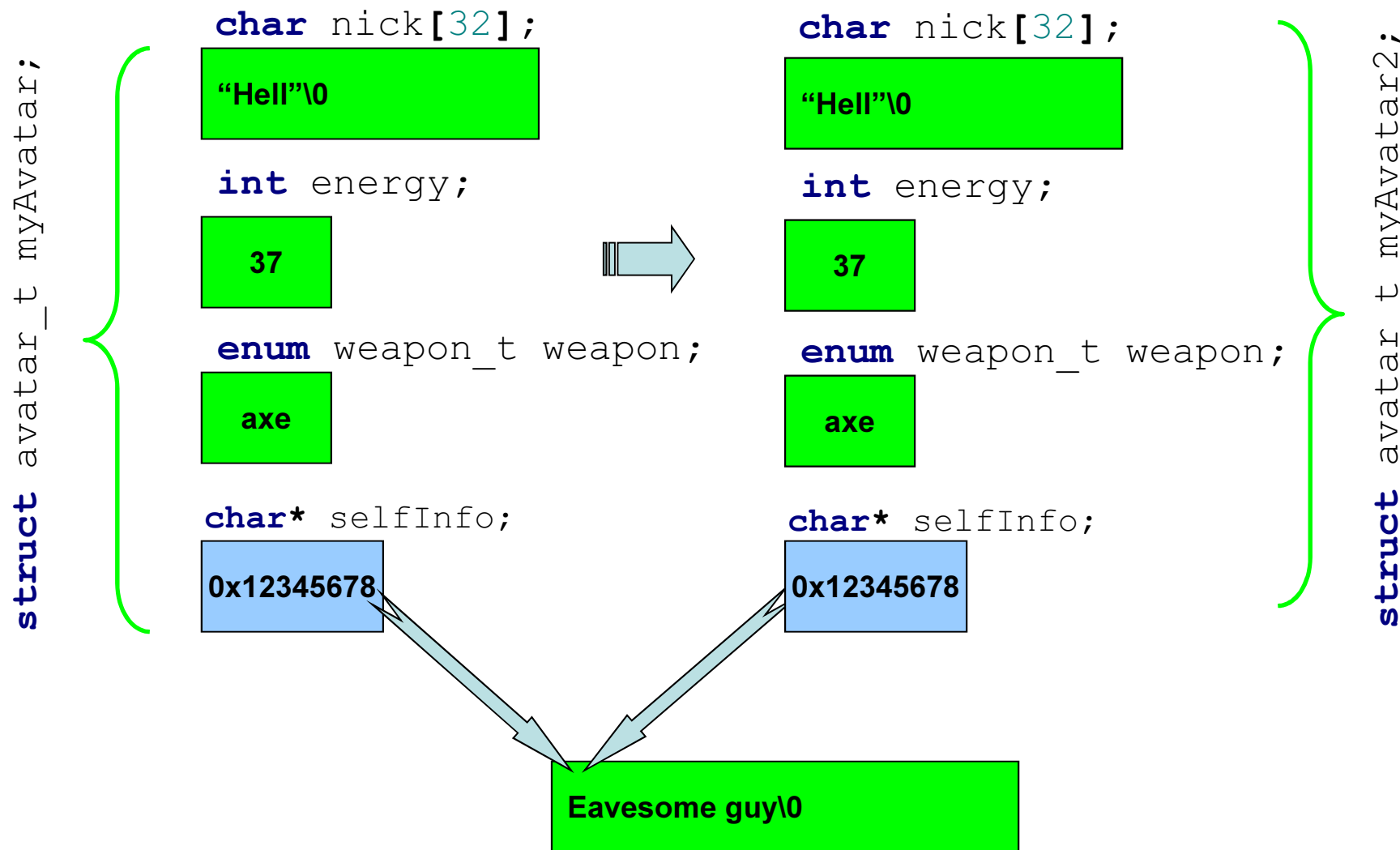
Kopírování struktur – přiřazovací operátor

- Obsah struktury lze kopírovat pomocí operátoru přiřazení

```
struct avatar_t myAvatar = {"Hell", 100, axe};  
struct avatar_t avatar3;  
  
avatar3 = myAvatar;
```

- Dochází ke kopírování obsahu jednotlivých položek
 - položka primitivního datového typu (int, float...)?
 - zkopíruje se hodnota položky
 - položka typu pole (char nick[32])?
 - zkopíruje se hodnota jednotlivých prvků pole
 - položka typu ukazatel
 - zkopíruje se adresa (v ukazateli)
- Analogické ke kopírování celé paměti se strukturou
 - např. pomocí funkce memcpy()

Kopie struktur - ilustrace



struct: plytká vs. hluboká kopie

- Co když je kopírovaná položka ukazatel?
 - zkopíruje se hodnota ukazatele (nikoli obsah odkazované paměti)
 - původní i nová struktura ukazují na společnou paměť
 - navzájem si mohou přepisovat
 - pokud jedna uvolní, tak druhá ukazuje na neplatnou paměť
- Kopie je “plytká”
- Pokud chceme vytvořit samostatnou odkazovanou paměť
 - vlastní položka `selfInfo` pro každého uživatele
 - tzv. “hluboká” kopie
 - musíme provést explicitně dodatečným kódem
 - `malloc()` + `memcpy()`

Využití struct: pole s pamatováním délky

- array + length vs. **struct** { array, length}
- Vhodné např. pro dynamicky alokované pole

```
struct int_blob_t {  
    int* pData;  
    unsigned int length;  
};  
struct int_blob_t array = {NULL, 0};  
array.length = 100;  
array.pData = malloc(array.length * sizeof(int));  
for (int i = 0; i < array.length; i++) array.pData[i] = i;  
free(array.pData);
```

- Stále nezajišťuje implicitní kontrolu přístupu!
 - lze číst a zapisovat mimo délku pole
 - máme ale pole i délku pohromadě

Dynamická alokace celých struktur

- Struktury lze dynamicky alokovat pomocí malloc()
 - stejně jako jiné datové typy
- Do jedné proměnné:
 - `struct avatar_t* avat = NULL;`
 - `avat = malloc(sizeof(struct avatar_t));`
- Do pole ukazatelů
 - `struct avatar_t* avatars[10];`
 - `avatars[2] = malloc(sizeof(struct avatar_t));`

Operátor `->` vs. operátor `.`

- Operátor `.` se použije pro přístup k položkám struktury
 - `struct` avatar_t myAvatar;
 - `myAvatar.energy = 10;`
- Pokud strukturu alokujeme dynamicky, máme ukazatel na strukturu
 - `struct` avatar_t* pMyAvatar = malloc(sizeof(struct avatar_t));
 - operátor `.` nelze přímo použít
 - musíme nejprve dereferencovat `(*pMyAvatar).energy = 10;`
- Pro zjednodušení je dostupný operátor `->`
 - `(*pStruct).atribut == pStruct->atribut`
 - `pMyAvatar->energy = 10;`

typedef

typedef

- Jak zavést nový datový typ použitelný v deklaraci proměnných?
 - **typedef** typ nové_synonymum;
- Ukázky
 - **typedef int** muj_integer;
 - **typedef int**[8][8][8] cube; cube myCube;
 - **typedef struct** avatar_t avatar; avatar myAvatar1;
 - **typedef** avatar* pAvatar; pAvatar pMyAvatar1 = NULL;
- Vhodné využití pro:
 - zkrácení zápisu dlouhých typů (např. **int**[8][8][8])
 - abstrakce od konkrétního typu
 - **typedef int** return_value;
 - **typedef int** node_value;
 - odstranění nutnosti psát struct, union, enum u deklarace proměnné

```
typedef struct node {struct node* pNext;int value;} node_t;  
  
node_t node1;
```

Kombinace typedef + struct

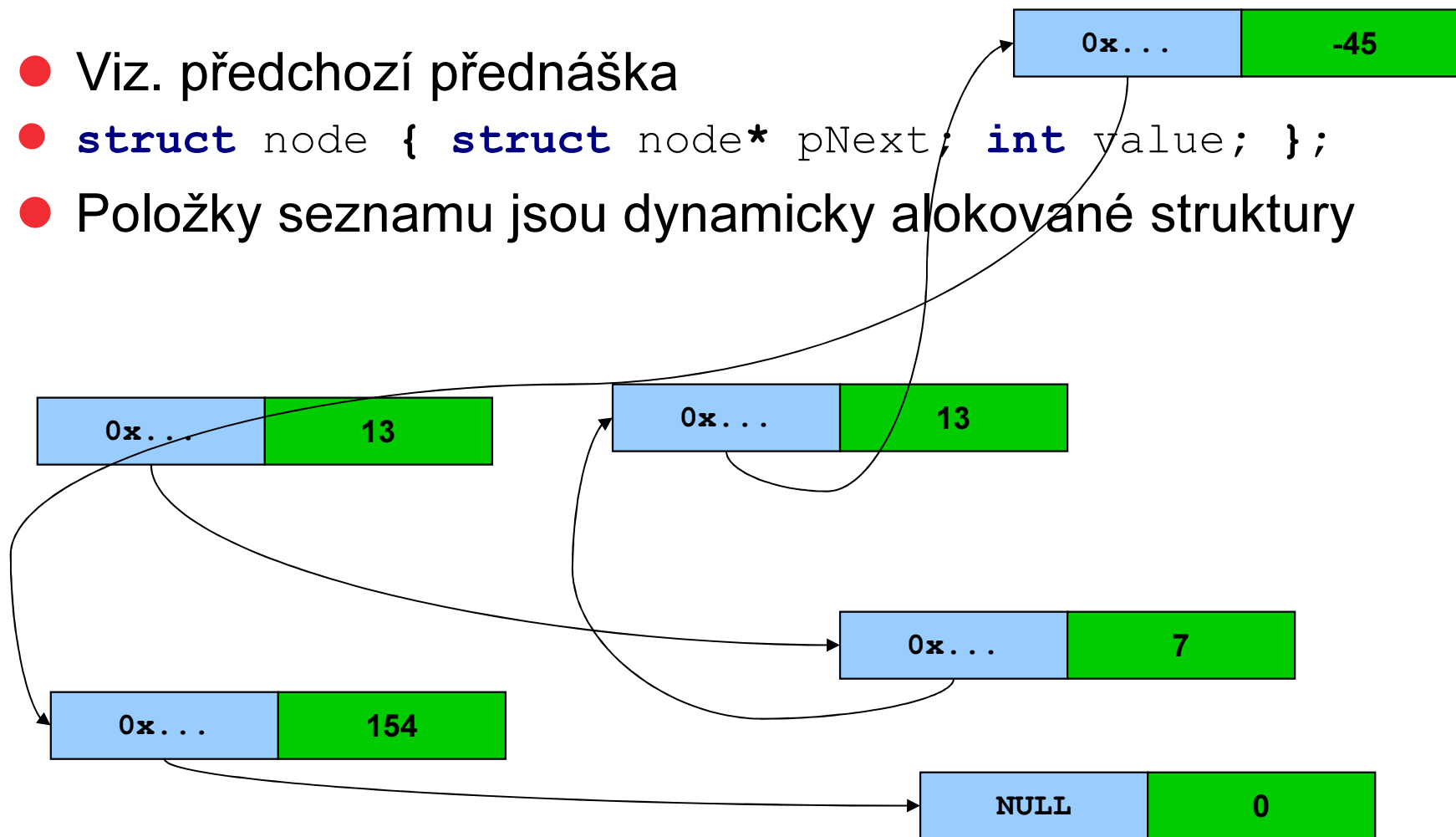
```
struct priority_queue {  
    priority_queue_item* first;  
    priority_queue_item* last;  
    uint size;  
};  
struct priority_queue mojePromenna;
```

- **typedef** **starý_typ** **nový_typ**;

```
typedef  
struct priority_queue {  
    priority_queue_item* first;  
    priority_queue_item* last;  
    uint size;  
}  
priority_queue;
```

Dynamická alokace – zřetězený seznam

- Viz. předchozí přednáška
- `struct node { struct node* pNext; int value; };`
- Položky seznamu jsou dynamicky alokované struktury



Zřetězený seznam - použití

- Pro velké datové struktury s proměnnou délkou
- Pro často se měnící struktury
 - průběžně vkládáme a ubíráme prvky
- Obecně použitelná struktura, pokud nemáme speciální požadavky ani neoptimalizujeme
 - tj. nejde nám příliš o rychlost, ale zároveň nechceme zbytečně „pomalou“ nebo „velkou“ strukturu
 - nalezneme často ve standardních knihovnách jazyků
 - C++ `std::list<E>`, Java `LinkedList<E>`, C# `List<E>`
- Nesená hodnota může být složitější struktura
 - nejen `int`, ale např. celý `struct` `avatar_t`

Zřetězený seznam – typické operace

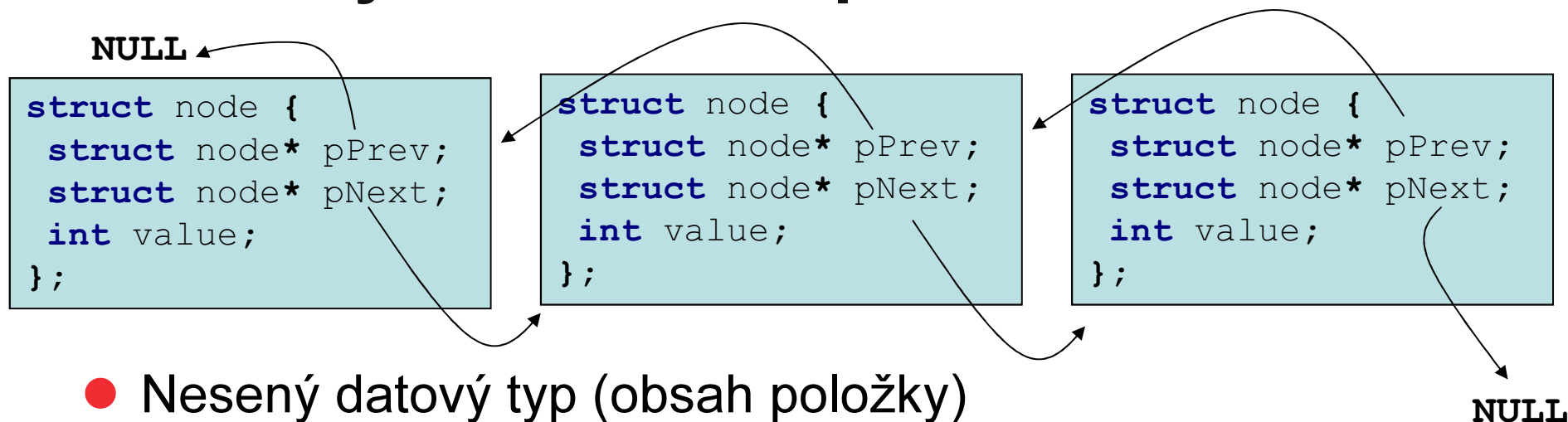
- Vložení na začátek/konec
- Nalezení položky dle hodnoty
- Přidání nové hodnoty za/před nalezenou položku
- Odstranění nalezené položky
- Změna hodnoty položky
- Přesun položky na jiné místo

Zřetězený seznam - vlastnosti

- (Srovnání typicky s dynamicky alokovaným polem)
- Výhody
 - Potenciálně neomezený počet položek
 - Složitost vložení prvku na začátek/konec?
 - Složitost zařazení prvku do setříděné posloupnosti?
 - Složitost vložení prvku za daný prvek?
 - Složitost daného odstranění prvku?
 - Složitost nalezení prvku?
- Nevýhody
 - Větší paměťová náročnost (ukazatele)
 - Není konstantní složitost přístupu na i-tý prvek
 - Není vhodné pro dotazy typu klíč → hodnota
 - Vložení hodnoty je sice $O(1)$, ale náročnější než $a[i] = 5;$

$O(1)$
$O(n)$
$O(1)$
$O(1)$
$O(n)$

Zřetěžený seznam – implementace I.



- Nesený datový typ (obsah položky)
 - např. `int` nebo `struct` `avatar_t`
- Ukazatel na následující/předchozí prvek
 - např. `int` nebo `struct` `avatar_t`
- Signalizace prvního prvku
 - ukazatel na předchozí (`pPrev`) je `NULL`
- Signalizace posledního prvku
 - ukazatel na následující (`pNext`) je `NULL`

Zřetězený seznam – implementace II.

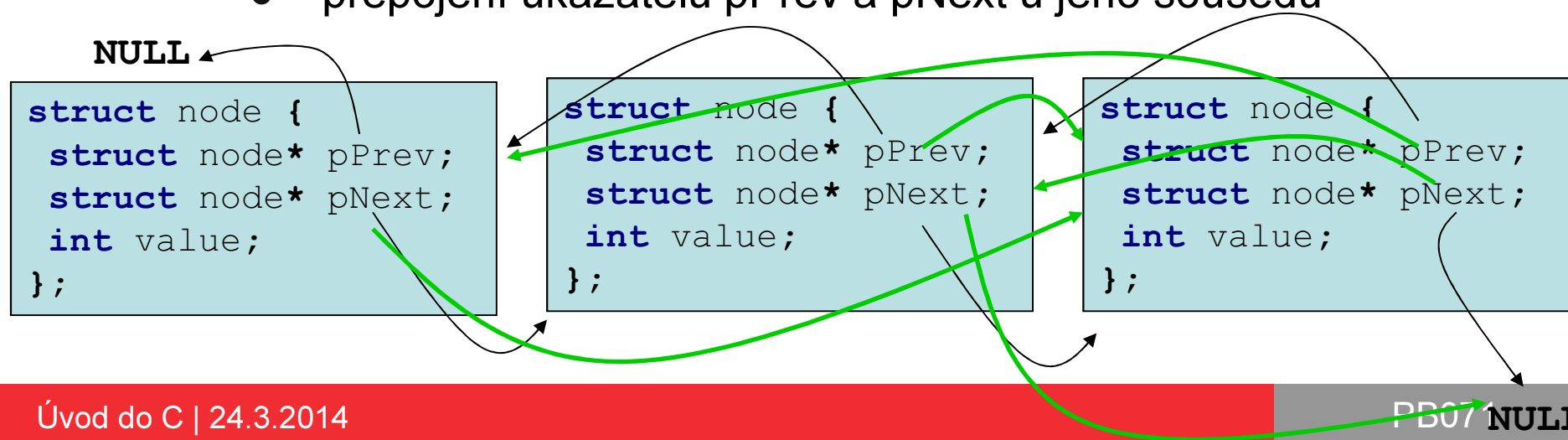
- V programu máme trvale ukazatel na první prvek
 - ostatní položky jsou z něj dostupné
- Často se udržuje i ukazatel na poslední prvek
 - protože častá operace je vložení na konec
- Typické procházení posloupnosti pomocí **while**

```
pNode = pFirstNodeInList;
while (pNode != NULL) {
    // Do something with pNode->value
    printf("%d", pNode->value);
    // Move to next node
    pNode = pNode->pNext;
}
```

Zřetězený seznam – implementace III.

● Přesun prvku

- není nutná dealokace (=> potenciálně větší rychlost)
- 1. nalezneme prvek
- 2. odpojíme jej korektně ze stávajícího umístění
 - přepojení ukazatelů pPrev a pNext u jeho sousedů
- 3. nalezneme novou pozici pro umístění
- 4. vložíme mezi stávající prvky
 - přepojení ukazatelů pPrev a pNext u jeho sousedů



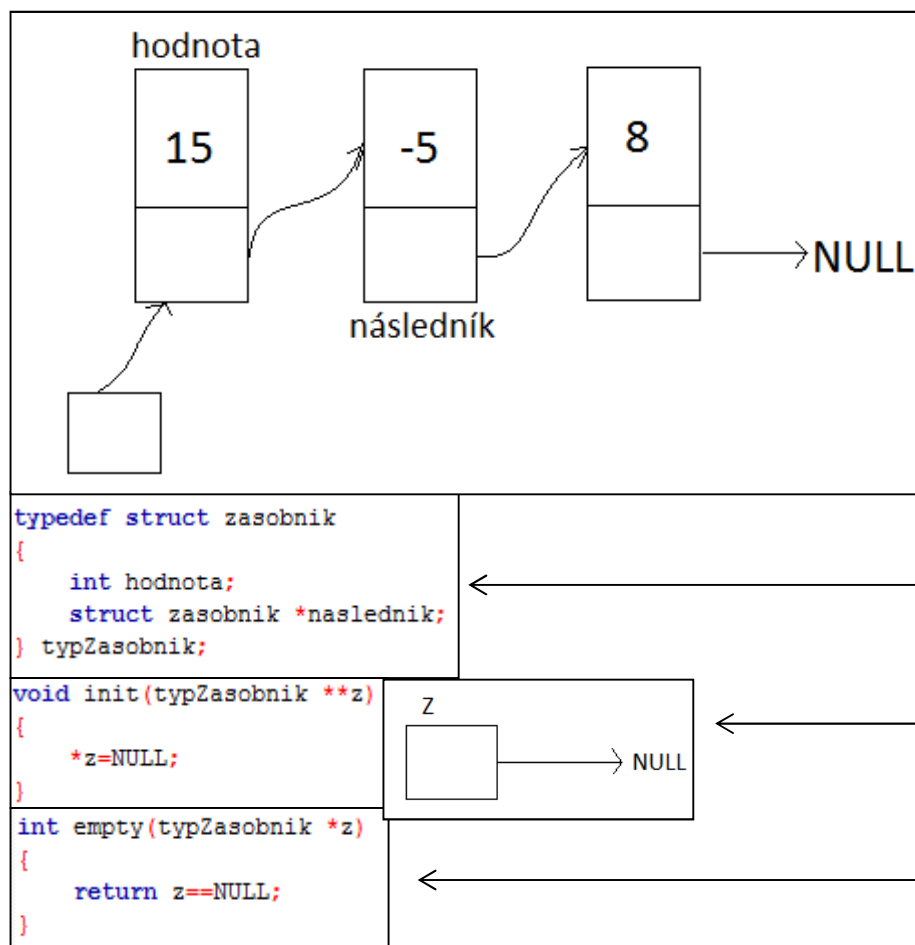
Zřetězený seznam – implementace III.

- V programu si musíme držet alespoň ukazatel na začátek seznamu
- Lze vytvořit dodatečnou strukturu `List`, která bude obsahovat ukazatel na začátek(konec) seznamu

```
typedef struct _list {  
    node* first;  
    node* last;  
} list;
```

- Do funkcí pak předáváme ukazatel na `List`, nikoli ukazatel na první prvek
- Lze uchovávat další informace, např. počet prvků

Zásobník - implementace



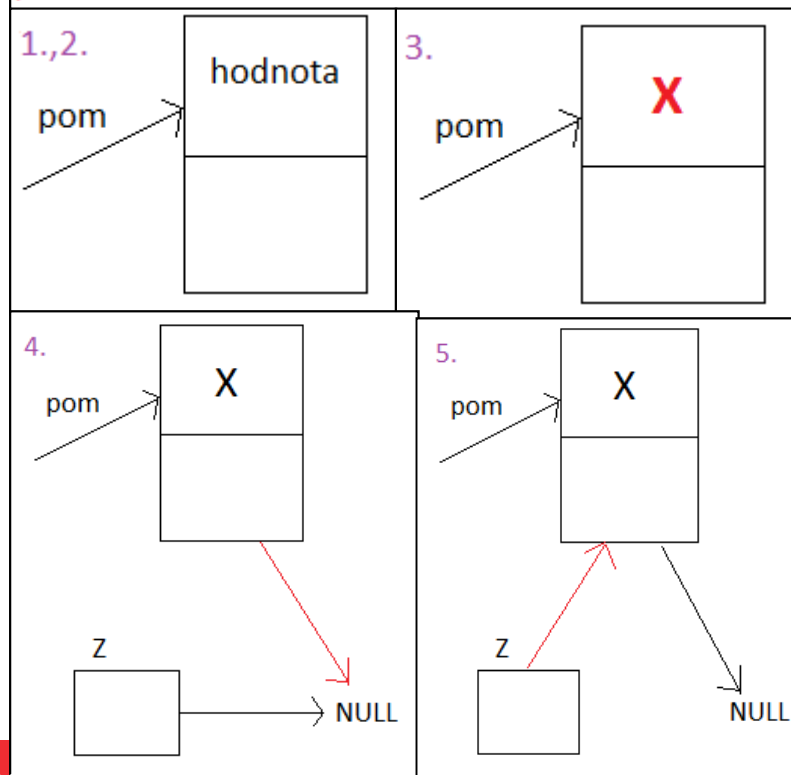
- Zjednodušené schéma
- Možné operace zásobníku – init, push, pop, empty
- Datová struktura obsahující hodnotu a ukazatele následníka
- init - Inicializace zásobníku před jejím prvním použitím
- empty – testování, zda zásobník je prázdný

Slidy pro zásobník vytvořil Martin Paulík

Zásobník – ukázka push

```
void push(typZasobnik **z, int x)
{
    1. typZasobnik *pom;

    2. pom=(typZasobnik *)malloc(sizeof(typZasobnik));
    3. pom->hodnota=x;
    4. pom->naslednik=*z;
    5. *z=pom;
}
```



- push – vkládání hodnoty na vrchol zásobníku
- 1. definuji ukazatel pom na datovou strukturu
- 2. ukazatel pom alokuji v paměti na velikost typZasobnik
- 3. hodnota v datové struktuře se změní na hodnotu X
- 4. ukazatel v datové struktuře bude ukazovat na to, kam ukazuje *z, tedy NULL
- 5. *z ukazuje na datovou strukturu a tím jsme dokončili vložení hodnoty na vrchol

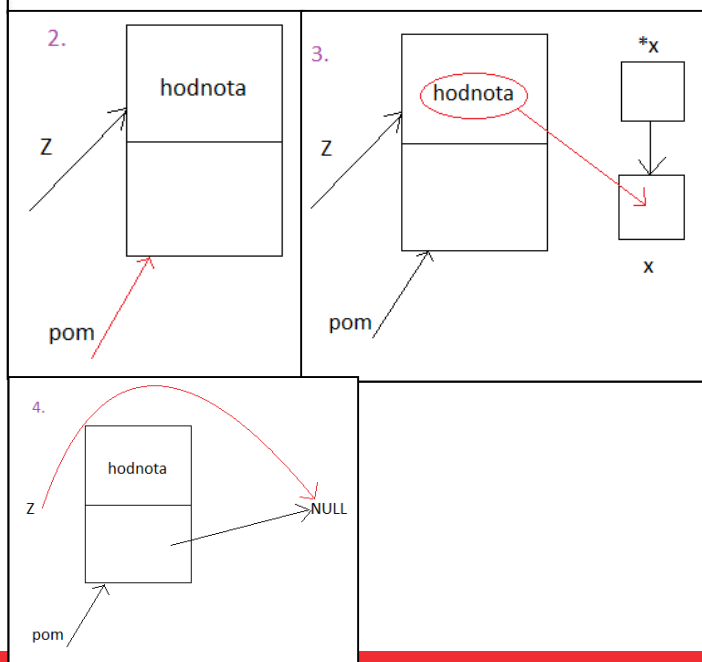
Zásobník – ukázka pop

```
int pop(typZasobnik **z, int *x)
{
    1. typZasobnik *pom;

    if (empty(*z)) return 0;

    2. pom=*z;
    3. *x=pom->hodnota;
    4. *z=pom->naslednik;
    5. free(pom);

    return 1;
}
```



- pop – odebírání ze zásobníku položky
- 1. stejné jako u předchozího (push)
- příkazem **if(empty(*z)) return 0;** testujeme, zda zásobník je prázdný, pokud ano, končíme (protože není z čeho vybírat), pokud ne, pokračujeme ve funkci pop
- 3. hodnota v datové struktuře je zkopírována do místa, kam ukazuje ukazatel `*x`
- 4. `*z` bude ukazovat tam, kde ukazoval ukazatel `naslednik` v datové struktuře, bude to buď další struktura nebo NULL
- 5. Datová struktura se uvolní, tím jsme vybrali jednu položku, aniž by zůstala a operace pop je dokončena.

Ladění dynamických struktur

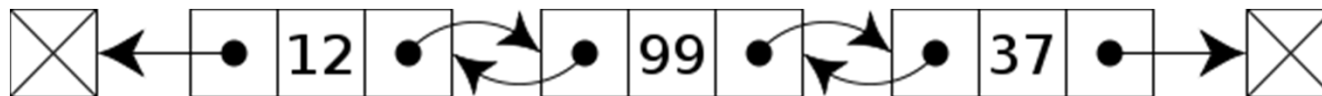
Ladění dynamických struktur

- Situace:
 - dynamicky alokované položky dostupné přes ukazatel
 - nemáme pro každou položku proměnnou obsahující ukazatel
- Typickým příkladem je dynamicky alokovaný list
 - a operace přidání, odebrání, vyhledávání
 - např. domácí úkol na rotaci obrázků



Kombinace více metod vede typicky k rychlejšímu odladění

Ladění – „manuální“ metody



- Kreslení struktury pastelkama
 - ukazatele, přepojování
- Výpis obsahu struktury (seznamu)
 - volání po každé operaci a kontrola obsahu
 - např. postupně vložit na konec 10 prvků
 - výpis po každém vložení
 - jako hodnotu v položce seznamu si zvolte unikátní číslo/řetězec
 - umožní vám snadno detekovat chybně vložený prvek
- Speciální funkce na kontrolu integrity struktury
 - očekávaný počet položek
 - validita ukazatelů (NULL na koncích)
 - validita ukazatelů na celou strukturu (první resp. poslední prvek)
 - výhodný kandidát na unit testing a integrační testování!
- Volejte po každé operaci
 - po odladění se odstraní, např. makro VERBOSE nebo assert()

Typické problémy u dynamických struktur

- Nedojde k propojení všech ukazatelů při
 - např. pouze pNext a ne pPrev
 - musíme aktualizovat ukazatele u tří prvků (vlevo, aktuální, vpravo)
- Chybné propojení v případě manipulace u prvního/posledního prvku
 - pád programu při pokusu o přístup na adresu NULL
 - poškození nebo neuložení aktualizovaného ukazatele na první/poslední prvek
- Přepsání stávajícího ukazatele adresou na nový prvek => ztráta ukazatele => memory leaks
- Chybí korektní dealokace po ukončení práce => memory leaks

Ladění – využití debuggeru

- Nevyhýbejte se použití debuggeru
 - i prosté krokování funkce dá výrazný vhled do chování!
 - lze brát jako učitele ukazující postupně jednotlivé kroky algoritmu
- Zobrazte si hodnoty proměnných
 - v případě dynamické struktury je ale obtížnější
 - typicky máte aktuální prvek při procházení seznamu
 - a můžete si zobrazit jeho adresu i obsah
 - některé debuggeru umožní procházet postupně i další prvky
 - klikáním na položky „next“ a „previous“ (u seznamu)
- Pokud je nutné, lze si poznačit adresy/hodnoty do obrázku
 - např. kontrola, zda je seznam pospojovaný opravdu správně

Zobrazení dalších položek v debuggeru

```
// START WITH NODE START_NODE_INDEX
// SEARCH FOR PATH TO NODE TARGET_NODE_INDEX
list pathList;
if (DFS_BFS_search(&g, g.items[START_NODE_INDEX], TARGET_NODE_INDEX, &pathList, TRUE) != NULL) {
    numSearches++;
    // PRINT PATH
    printf("PATH:");
    node* pNode = pathList.begin;
    while (pNode != NULL) {
        printf(" -> %d", pNode->pItem->value);
        pNode = pNode->next;
        avgLength++;
    }
}
```

Watch 1

Name	Value
pNode	0x003018b0 { pItem=0x00301550 next=0x00301868 prev=0x00000000 }
pItem	0x00301550 { value=1 bVisited=1 parent=0x00000000 }
next	0x00301868 { pItem=0x003015e0 next=0x00301820 prev=0x003018b0 }
pItem	0x003015e0 { value=3 bVisited=1 parent=0x00301550 }
next	0x00301820 { pItem=0x00301670 next=0x00000000 prev=0x00301868 }
pItem	0x00301670 { value=5 bVisited=1 parent=0x003015e0 }
next	0x00000000 { pItem=??? next=??? prev=??? }
prev	0x00301868 { pItem=0x003015e0 next=0x00301820 prev=0x003018b0 }
prev	0x003018b0 { pItem=0x00301550 next=0x00301868 prev=0x00000000 }
prev	0x00000000 { pItem=??? next=??? prev=??? }

adresa aktuální položky

ukazatel na následující

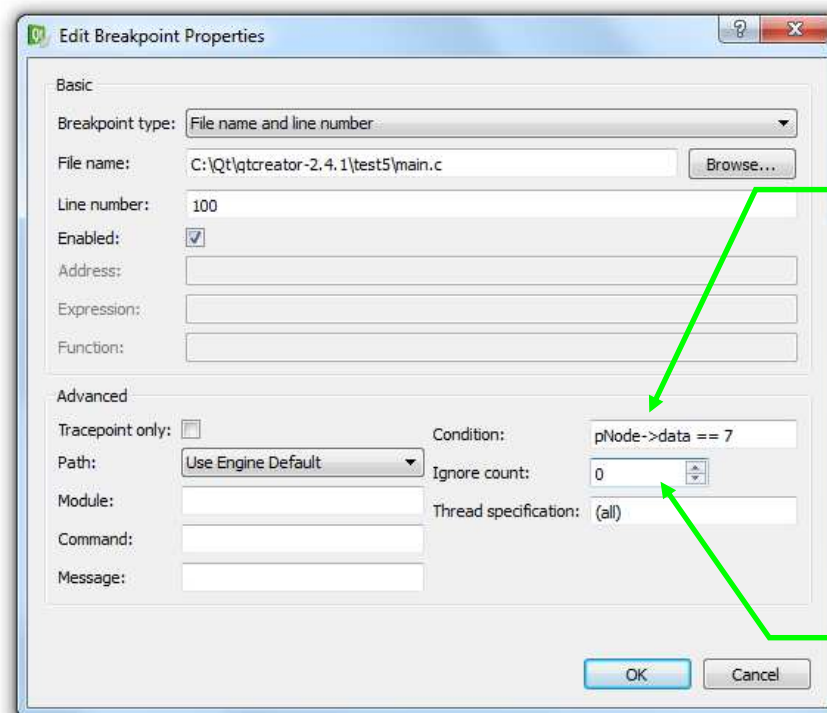
ukazatel na následující od následující

Ladění – využití debuggeru 2

- Struktury mohou být velké
 - např. tisíce prvků, nelze procházet vždy od začátku
 - víme, že problém nastává po vkládání prvku s hodnotou '1013'
- Podmíněný breakpoint
 - má (snad) každý debugger
 - vložte breakpoint, pravé myšítko a editujte podmínku
 - např. zastav pokud je `hodnota == 1013`
 - (můžete samozřejmě použít i jinou podmínku)

Podmíněný breakpoint – QT Creator

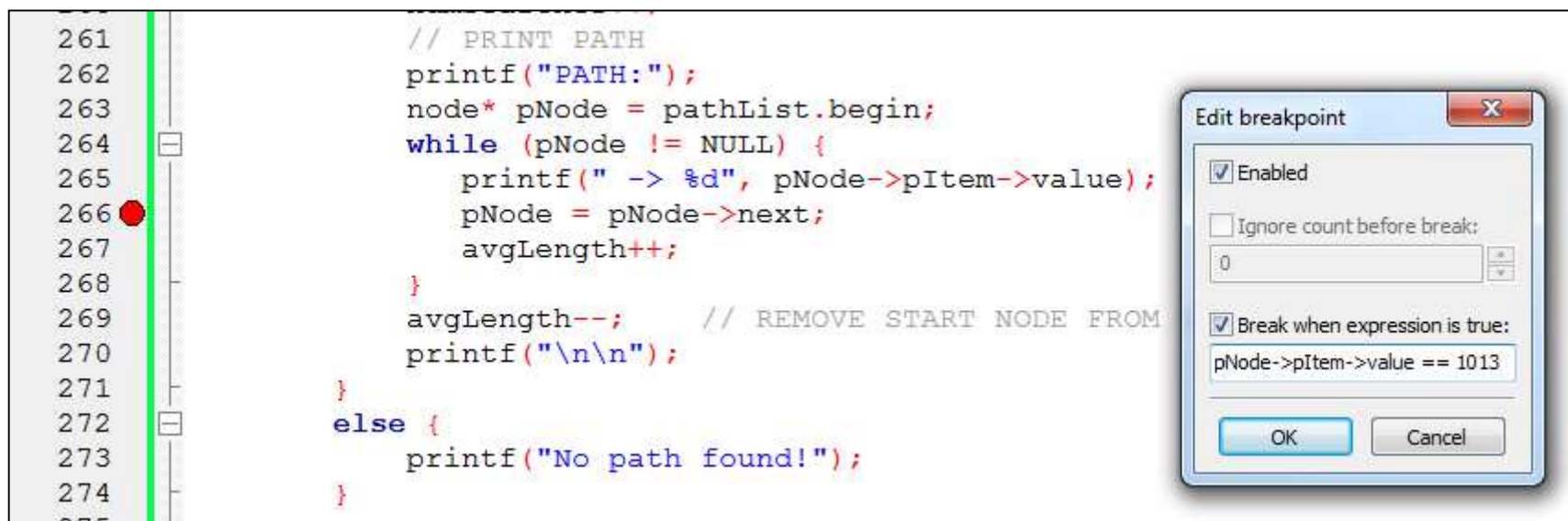
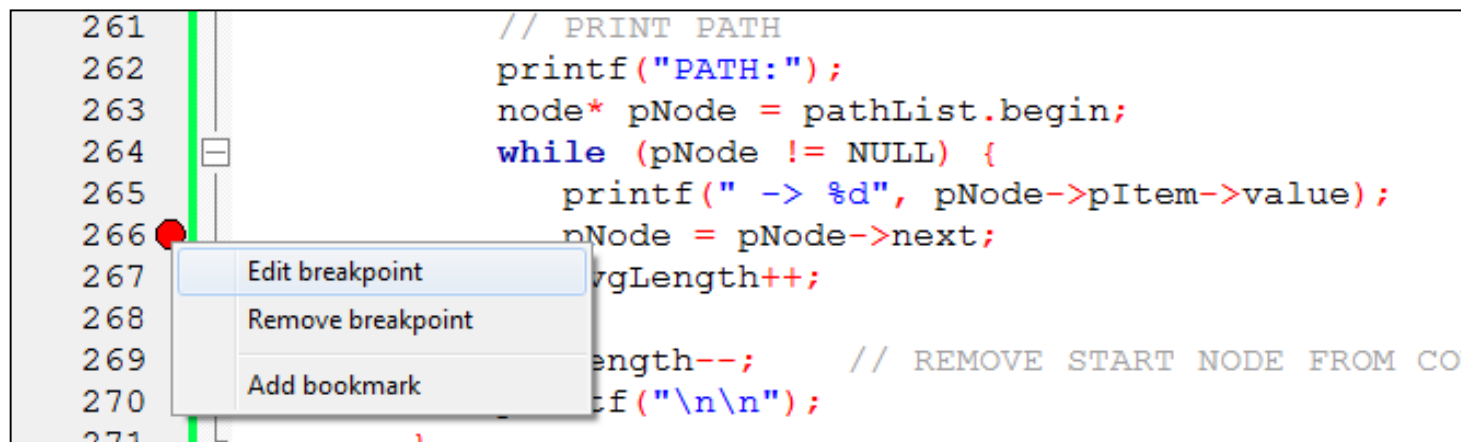
- Toggle breakpoint (F9)->R-Click->Edit breakpoint



podmínka

počet ignorování
„splnění“ breakpointu

Podmíněný breakpoint – Code::Blocks



Podmíněný breakpoint ve VS2010

The image shows a screenshot of the Visual Studio 2010 IDE. In the background, a C++ code file is open, displaying a linked list traversal function. A context menu is open over a breakpoint on the line `pNode = pNode->next;`. The menu options are: Delete Breakpoint, Disable Breakpoint (Ctrl+F9), Location..., Condition... (highlighted), Hit Count..., Filter..., When Hit..., Edit labels..., and Export....

In the foreground, the 'Breakpoint Condition' dialog box is open. It contains the following text: 'When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.' Below this, there is a checked checkbox for 'Condition:'. The text box next to it contains the expression `pNode->pItem->value == 1013`. At the bottom, there are two radio buttons: 'Is true' (which is selected) and 'Has changed'. The dialog has 'OK' and 'Cancel' buttons at the bottom right.

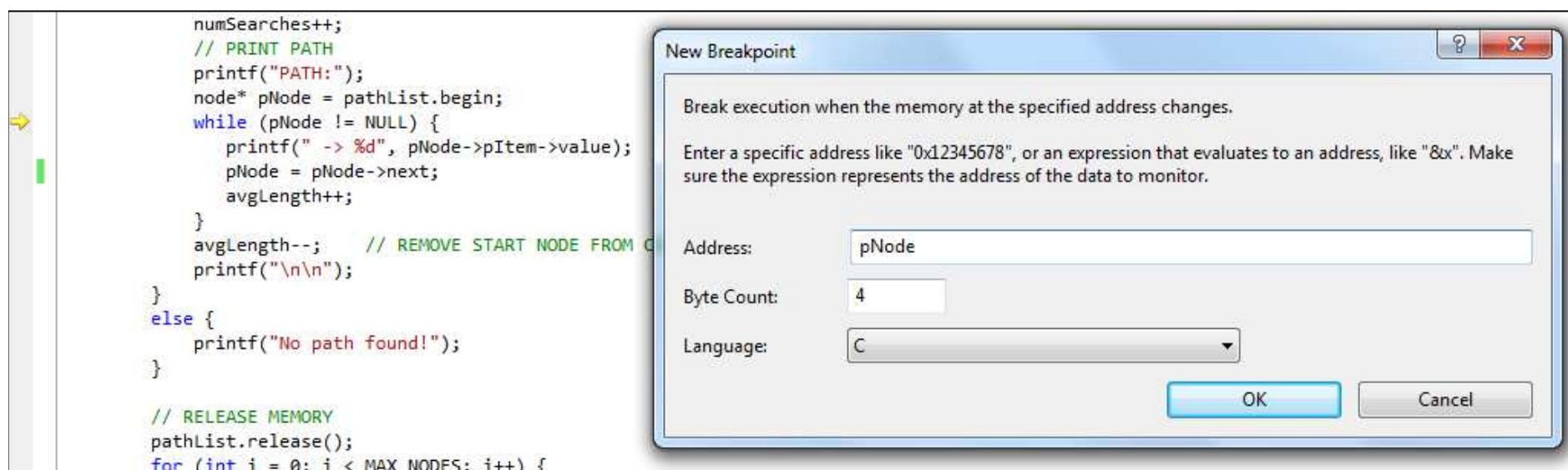
```
node* pNode = pathList.begin;
while (pNode != NULL) {
    printf(" -> %d", pNode->pItem->value);
    pNode = pNode->next;
}

// PRINT PATH
printf("PATH:");
node* pNode = pathList.begin;
while (pNode != NULL) {
    printf(" -> %d", pNode->pItem->value);
    pNode = pNode->next;
}
```

Ladění – paměť je klíčová

- U dynamických struktur je klíčová paměť
 - přidejte si výpisy ukazatelů (&polozka, polozka->next)
 - výpis obsahu paměti nebo Memory watch
- QtCreator
 - Windows→View→Memory
- Code::Blocks
 - Debug→Debugging windows→Examine memory
- Visual Studio 2010
 - Debug→Windows→Memory (až po spuštění ladění)
- Memory breakpoint
 - pokročilá vlastnost debuggeru (např. VS2010)
 - podmíněný breakpoint na změnu v paměti (nutná podpora v CPU)
 - typicky jen několik bajtů
 - Debug→Breakpoint→New data breakpoint
- Kontrola memory leaks
 - nenechávat na konec, hůř se pak odstraňuje
 - memory leak může znamenat i chybu v kódu, nejen zapomenutý delete

Paměťový breakpoint – Visual Studio 2010



Hodnoty ukazatelů

- Debugger umožní zobrazit i hodnoty ukazatelů
 - pozor, mohou se mezi různými spuštěními měnit
 - první vložená položka nemusí být vždy na stejném místě na haldě
- Ukazatel na následující položku by měl odpovídat adrese této položky
 - typicky již ukazatel máte nebo získáte pomocí operátoru &
- V debug režimu může překladač nastavovat dosud nenastavené hodnoty ukazatelů na „speciální“ hodnoty
 - např. `0xBAADF00D`
 - poznáte podle toho při ladění neinicializovaný ukazatel
 - POZOR: existuje pouze v debug režimu!
 - v Release je „smetí“ z paměti
 - => nelze použít pro zjištění validity ukazatele!!!

Bonus – překvapení 😊

MOOC

- MOOC – Massive open online courses
 - http://en.wikipedia.org/wiki/Massive_open_online_course
 - až stovky tisíc studentů v jednom kurzu (270k rekord)
 - podpora studentů formou sociální sítě (studenti si radí navzájem)
- Špičkové kurzy světových universit zdarma
 - Coursera, <https://www.coursera.org/>
 - edX, <https://www.edx.org/>
 - UDACITY, <https://www.udacity.com/>
- Velmi aktuální a rychle se rozvíjející věc
 - ale založena na dlouhotrvajícím výzkumu o optimalitě učení
 - cca 10 minutové kusy přednášek s následným cvičením

Search by course name, category, university, or instructor

Sort by

Starting soon ▾

STARTING SOON

☐ Starting soon

ELIGIBLE FOR

☐ Signature Track

LANGUAGE

☐ English

☐ Spanish

☐ French

☐ Chinese

☐ Italian

CATEGORY

☐ Arts

☐ Biology & Life Sciences

☐ Business & Management

☐ Chemistry

☐ CS: Artificial Intelligence

☐ CS: Software Engineering

☐ CS: Systems & Security

☐ CS: Theory

☐ Economics & Finance

☐ Education

☐ Energy & Earth Sciences

☐ Engineering

☐ Food and Nutrition

☐ Health & Society

☐ Humanities

☐ Information, Tech, and Design

☐ Law



Rice University

An Introduction to Interactive Programming in Python

with Joe Warren, Scott Rixner, John Greiner & Stephen Wong

Apr 15th 2013

9 weeks long

Signature Track



Duke University

Healthcare Innovation and Entrepreneurship

with Marilyn M. Lombardi & Bob Barnes

Apr 15th 2013

6 weeks long

Signature Track



University of Washington

The Hardware/Software Interface

with Gaetano Borriello & Luis Ceze

Apr 15th 2013

8 weeks long



Johns Hopkins University

Mathematical Biostatistics Boot Camp

with Brian Caffo

Apr 16th 2013

7 weeks long



University of Washington

Computational Neuroscience

with Rajesh P. N. Rao & Adrienne Fairhall

Apr 19th 2013

8 weeks long



University of California, San Diego

Drug Discovery, Development & Commercialization

with Williams S. Ettouati & Joseph D. Ma

Apr 19th 2013

9 weeks long

Signature Track



Johns Hopkins University

Apr 22nd 2013

Zajímavé kurzy (Coursera)...

- Zajímavé kurzy <https://www.coursera.org/courses>
- Python
(<https://www.coursera.org/course/interactivepython>)
- Machine learning (<https://www.coursera.org/course/ml>)
- The Hardware/Software Interface
(<https://www.coursera.org/course/hwswinterface>)
- Statistics (<https://www.coursera.org/course/introstats>)
- Computer Architecture
(<https://www.coursera.org/course/comparch>)
- C++ For C Programmers
(<https://www.coursera.org/course/cplusplus4c>)
- ...

Absolvoval jsi TY něco zajímavého?

- Sbíráme prověřený seznam MOOC kurzu dotýkajících se programování
- <http://www.cecko.eu/public/code@fimu>
- Napiš prosím na svenda@fi.muni.cz