

# **PB071 – Principy nízkoúrovňového programování**

Uživatelské datové typy, dynamické struktury a jejich ladění

# Organizační

- -lm přepínač (matematická knihovna) je nyní defaultně zapnutý pro vaše odevzdání
- Využití Valgrindu při opravě domácích úkolů
  - Valgrind je důležitý pomocný nástroj pro zlepšení implementace (memory leaks, neinicializované proměnné, zápisy mimo alokované pole...)
- Valgrind je spouštěn v průběhu testů Kontrem (>HW03)
  - Pokud je nalezen problém, obdržíte za daný test pouze 3/5 bodů získaných za funkčnost
  - Využívejte Valgrind pro kontrolu vašich programů (během všech vašich implementovaných testů!)
- HW03 má 3 týdny na vypracování, ale začněte pracovat zavčas
  - Dynamická alokace se vždy obtížněji ladí
  - Vnitrosemestrálka

# Organizační

- Vnitrosemestrální písemka
  - Proběhne za dva týdny (Úterý 11.4. 20:00)
  - Nezapomeňte se přihlásit v ISU

# 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,  
    hobit  
};  
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 hobit = 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 překladač (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 u deklarace proměnné 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

## 1. Pro zpřehlednění zápisu konstant

- elf, human a hobit přehlednější než 0, 1, 2
- přiřazení hodnoty, switch...

## 2. 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)

Součet velikostí položek nemusí být roven sizeof(struct)  
Např. int nebude na 31. bajtu

```
enum weapon_t {sword,axe,bow};  
struct avatar_t {  
    char nick[31];  
    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

```
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;
}
```

hodnotou

hodnotou  
ukazatele

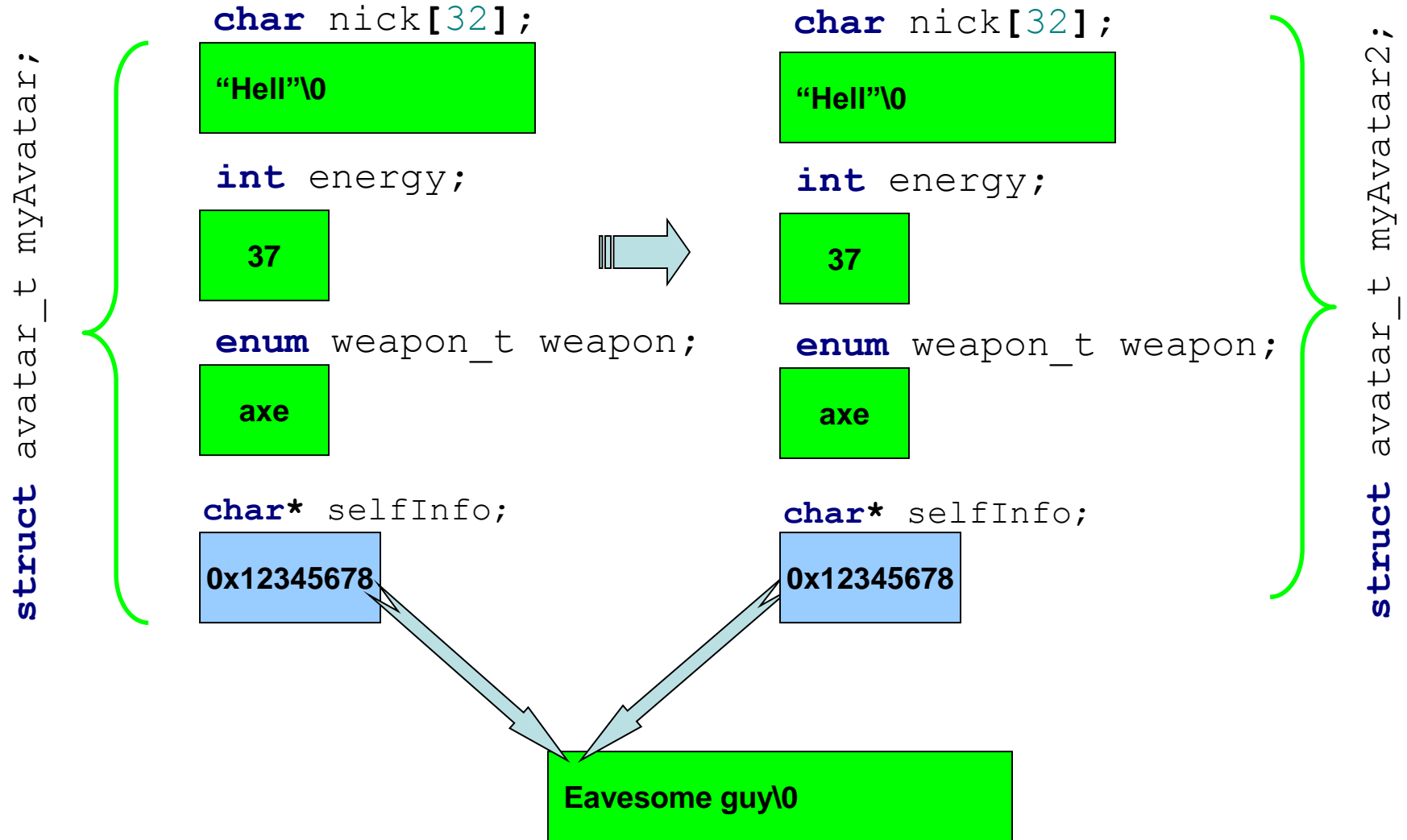
# Kopírování struktur – přiřazovací operátor

- 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;
```

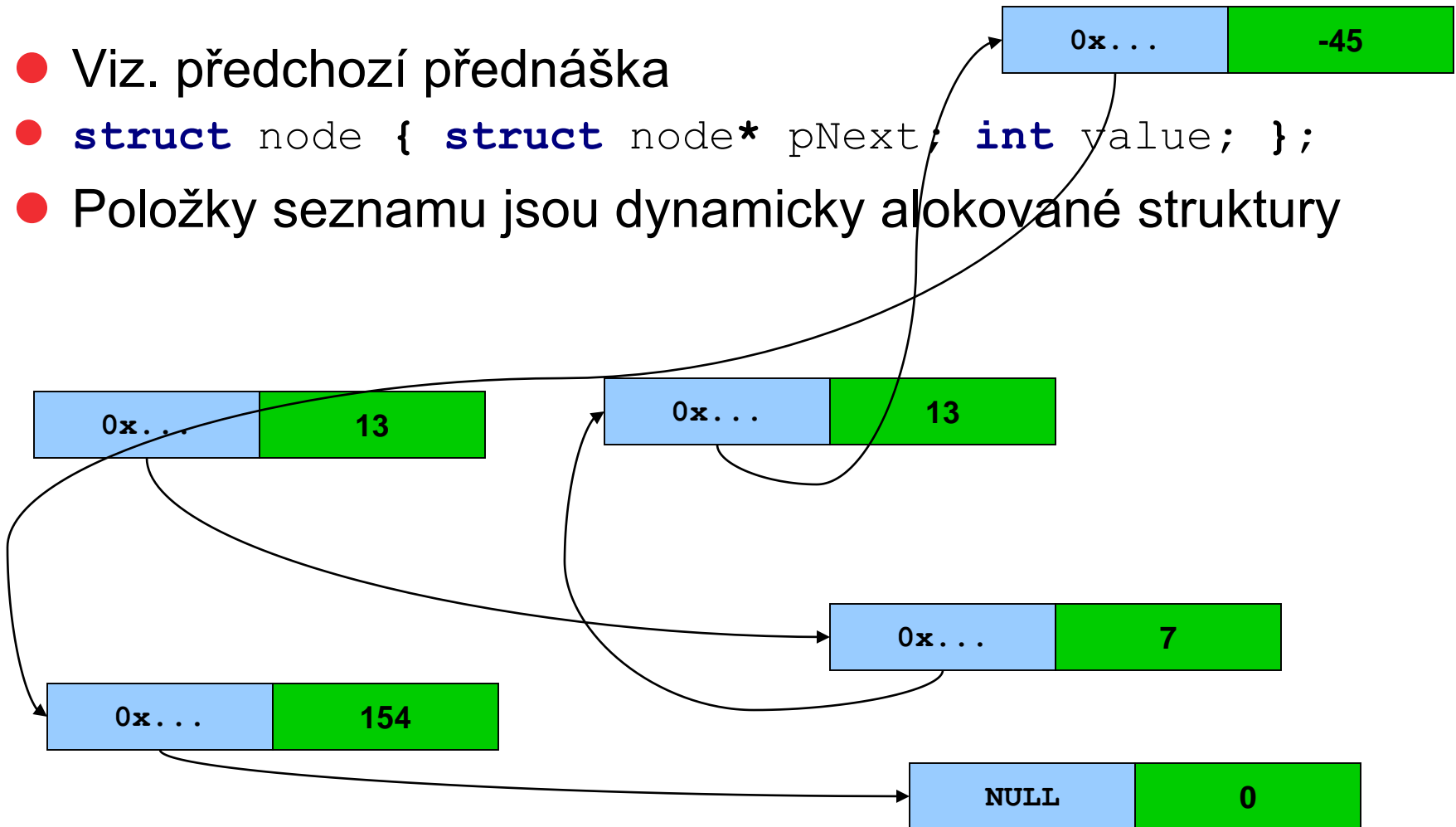
# PB071 Prednaska 07 – uživatelské typy

Kahoot!



# 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?

$O(1)$

$O(n)$

$O(1)$

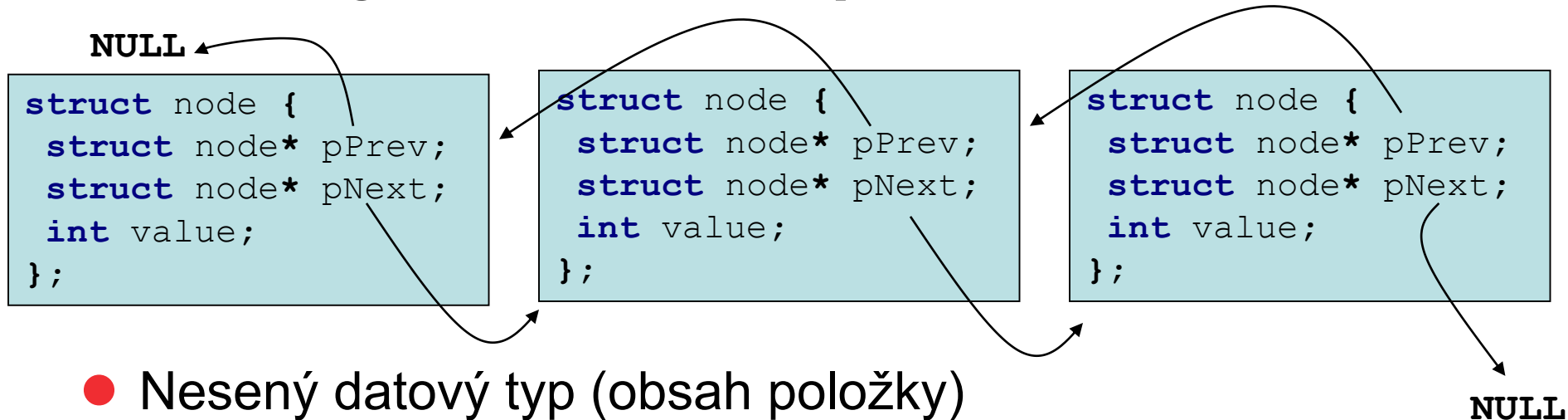
$O(1)$

$O(n)$

- Nevýhody

- Větší paměťová náročnost (ukazatele)
- Není konstantní složitost přístupu na  $i$ -tý prvek
- Není vhodné pro dotazy typu klíč  $\rightarrow$  hodnota
- Vložení hodnoty je sice  $O(1)$ , ale náročnější než  $a[i] = 5$ ;

# Zřetězený 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`

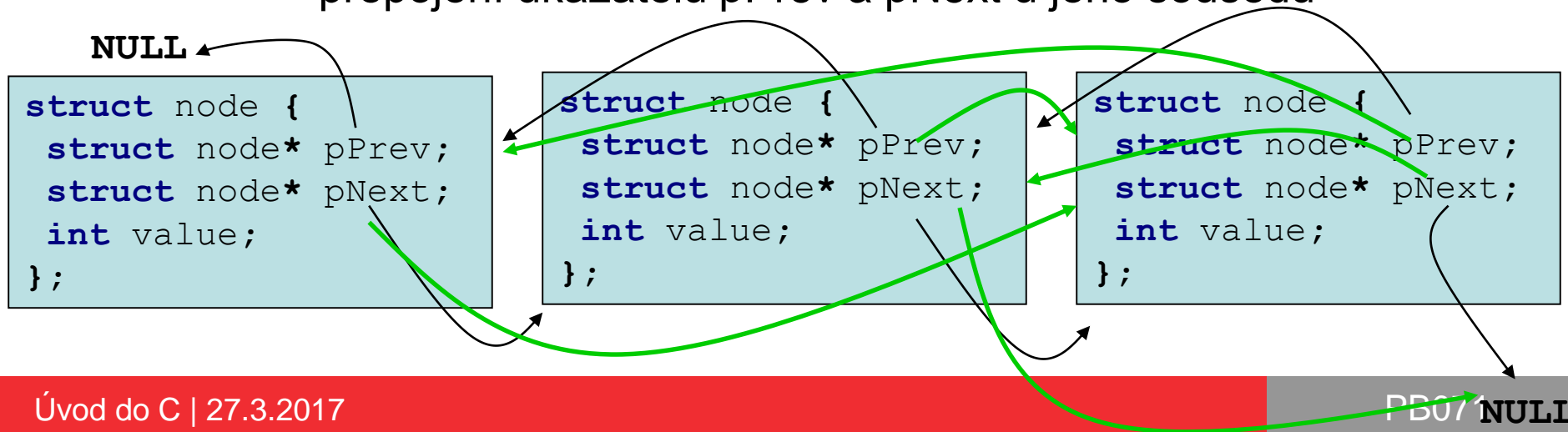
# Zřetězený seznam – implementace II.

- 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



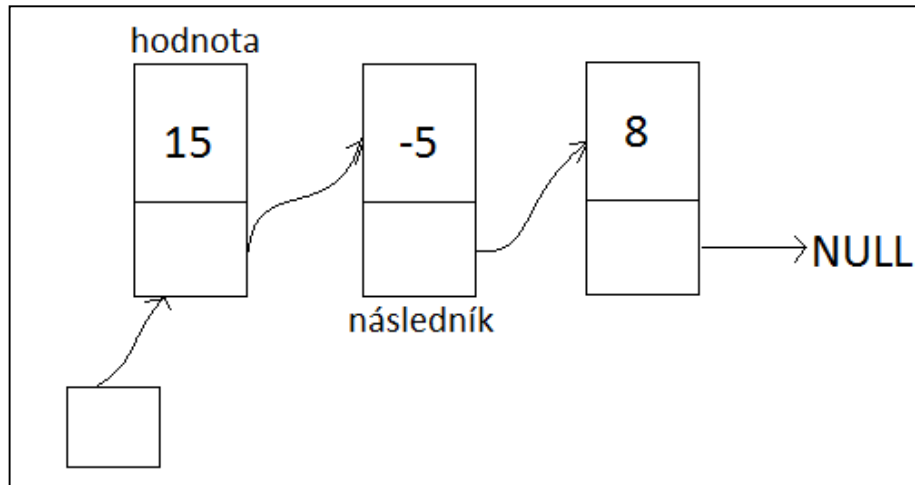
# Zřetězený seznam – implementace III.

- 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

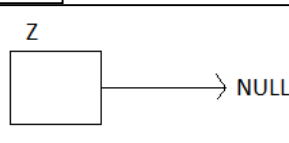
# Zásobník - implementace



```
typedef struct zasobnik  
{  
    int hodnota;  
    struct zasobnik *naslednik;  
} typZasobnik;
```

```
void init(typZasobnik **z)  
{  
    *z=NULL;  
}
```

```
int empty(typZasobnik *z)  
{  
    return z==NULL;  
}
```



- 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

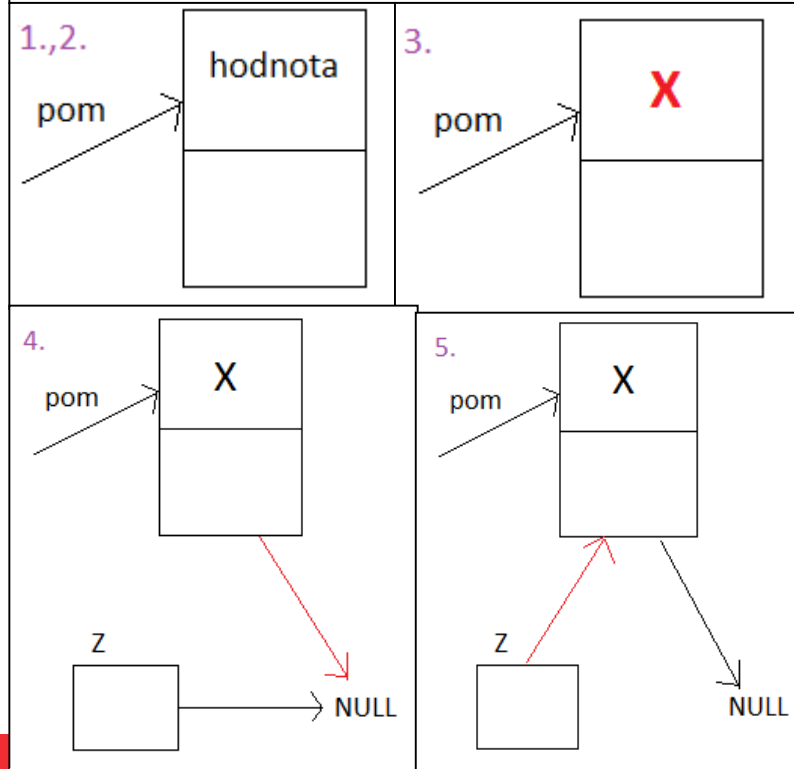
*Slidy pro zásobník vytvořil Martin Paulík*



# 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 změnil 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

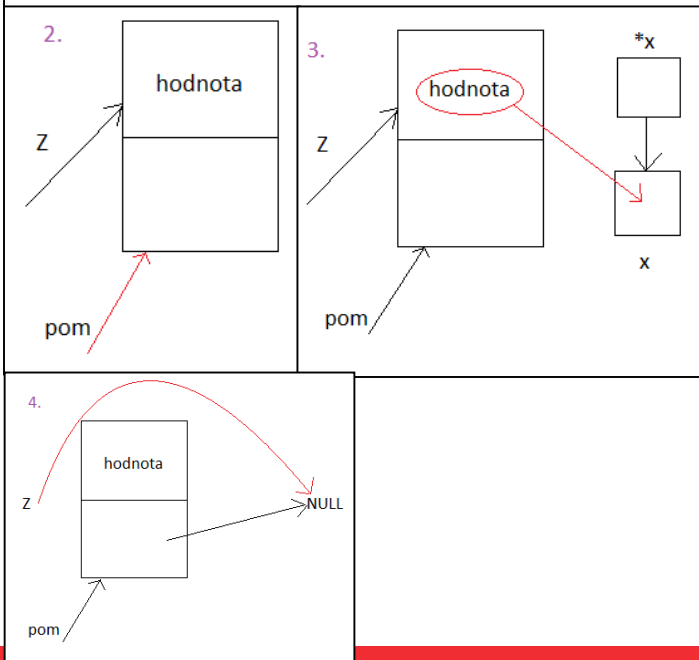
# Zásobník – ukázka pop

```
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 – odebírá 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

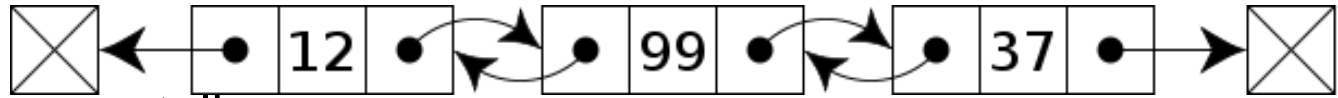
# Ladění dynamických struktur

- 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++;
    }
}
```

adresa aktuální položky

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=??? }

ukazatel na následující

ukazatel na následující od  
následující

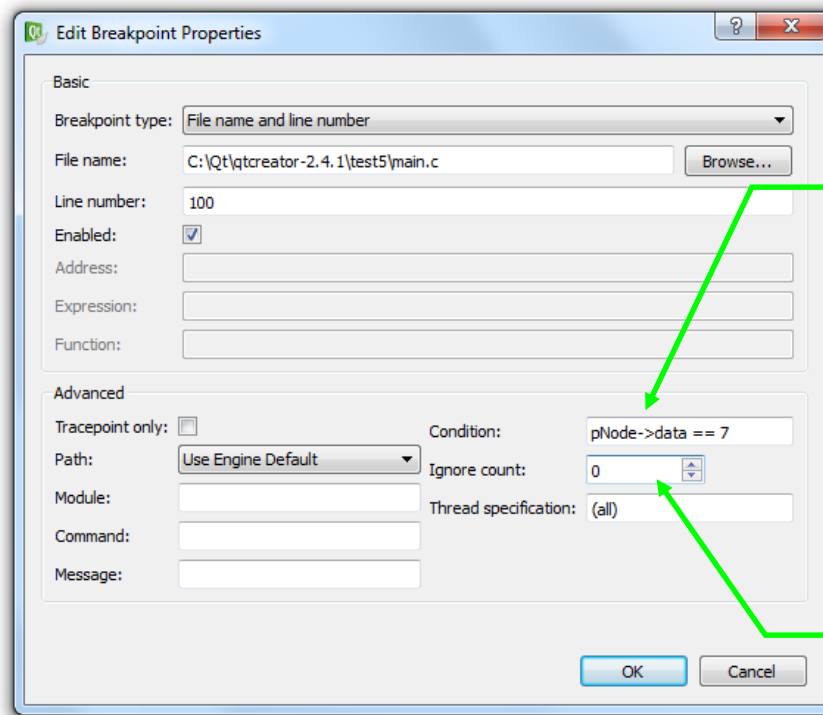


# Ladění – využití debuggeru 2

- 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

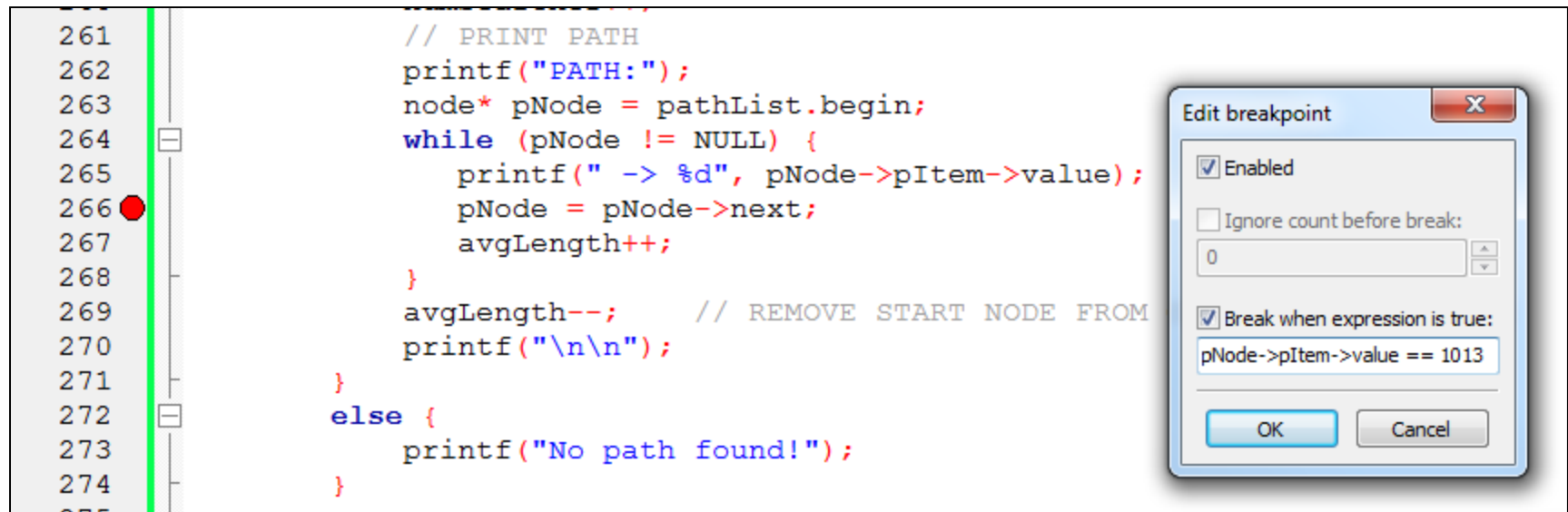
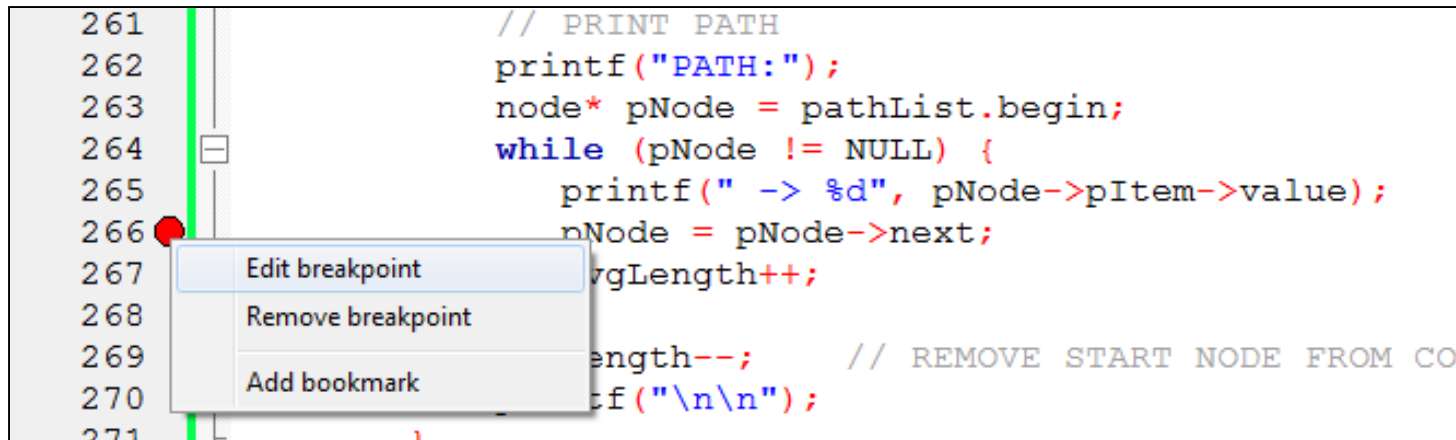
- Toggle breakpoint (F9) → R-Click → Edit breakpoint



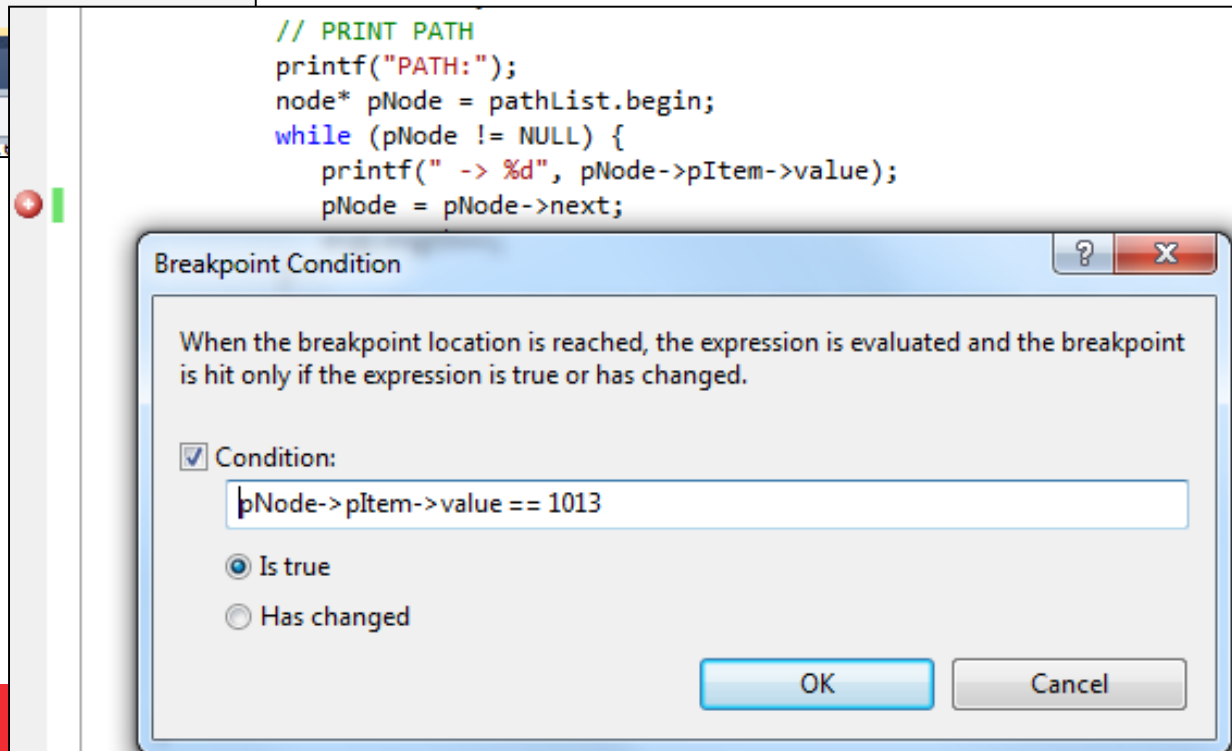
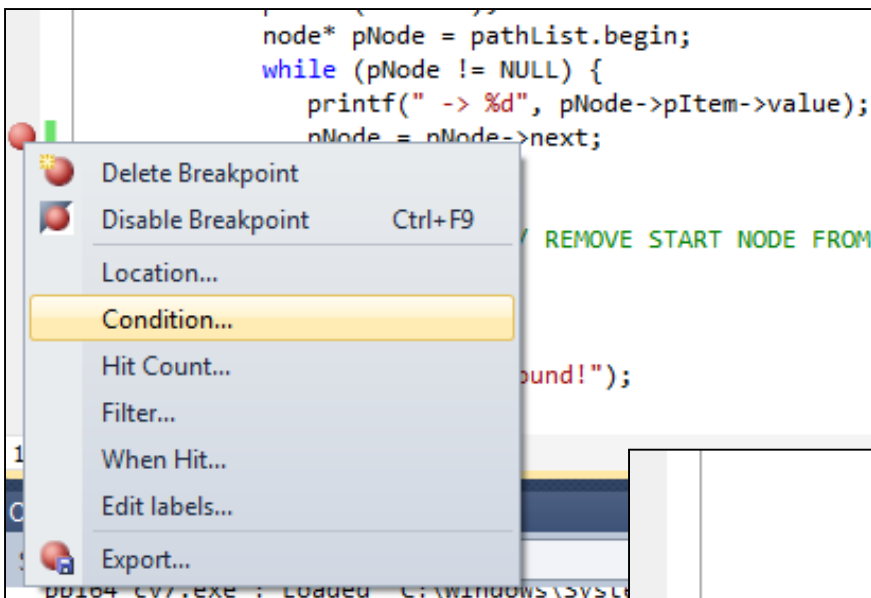
podmínka

počet ignorování  
„splnění“ breakpointu

# Podmíněný breakpoint – Code::Blocks



# Podmíněný breakpoint ve VS2010

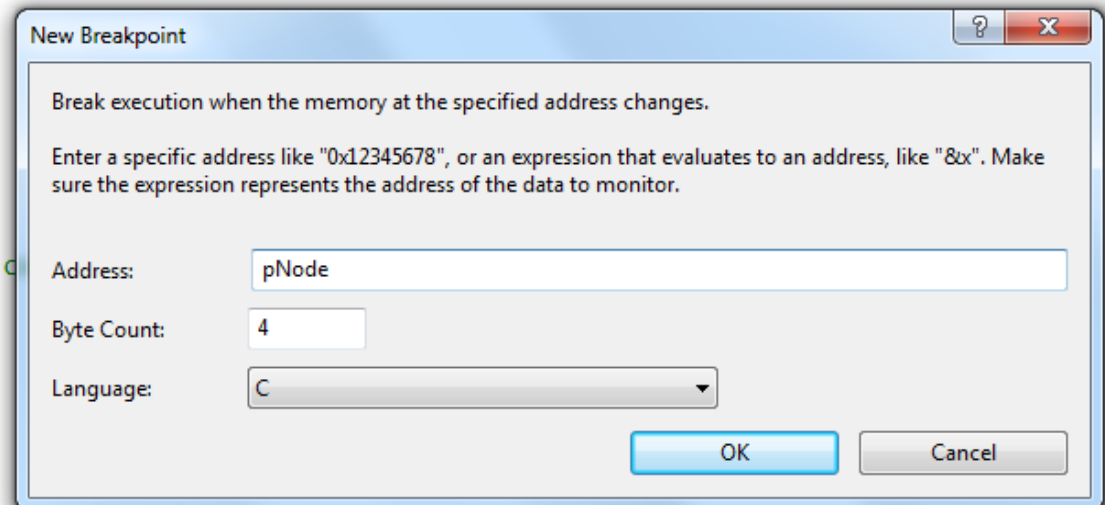


# Ladění – paměť je klíčová

- 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

```
numSearches++;  
// PRINT PATH  
printf("PATH:");  
node* pNode = pathList.begin;  
while (pNode != NULL) {  
    printf(" -> %d", pNode->pItem->value);  
    pNode = pNode->next;  
    avgLength++;  
}  
avgLength--; // REMOVE START NODE FROM C  
printf("\n\n");  
}  
else {  
    printf("No path found!");  
}  
  
// RELEASE MEMORY  
pathList.release();  
for (int i = 0; i < MAX_NODES; i++) {
```



# 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!!!

# PB071 Prednaska 07 – Dynamické struktury

The image features the Kahoot! logo in a large, white, 3D-style font with a slight shadow. The logo is centered over a background composed of a 4x4 grid of squares. The top-left half of the grid (the first two columns) is orange, and the bottom-left half (the last two columns) is yellow. The top-right half (the first two columns) is light blue, and the bottom-right half (the last two columns) is green. The overall effect is a vibrant, multi-colored backdrop for the white text.

**Kahoot!**



# Bonus – překvapení 😊

# Budoucnost programovacích jazyků

- Na internetu existuje velké množství blogů o přechodu mezi jazyky pro konkrétní tým ( $> 10^5$ )
  - “Why our team moved from X to Y”
- Erik Bernhardsson pomocí Google dotazů vytvořil tabulku přechodů  $N * N$ 
  - <https://erikbern.com/2017/03/15/the-eigenvector-of-why-we-moved-from-language-x-to-language-y.html>
  - <https://github.com/erikbern/eigenstuff>

# Na jazyk To language

## Z jazyku

From language

	c	c#	c++	clojure	cobol	erlang	fortran	go	haskell	java	kotlin	lisp	lua	matlab	node	objective c	pascal	perl	php	python	r	ruby	rust	scala	swift	visual basic
c		3619	60635	2	6	6	8	50527	10	28921		24	247	11		8	2700	20	14	15153	19	12	27	2	6	7
c#	37229		22049	5	2	1		3	4	9539			3	5	6	2749	3		4552	6089	1	30	2	16	1580	1183
c++	15631	14439		2		19	351	16	153	15959		7	21	10	1	17	12	5	19	5949	1	215	26	8	5	8
clojure								3	5	4					4					2				1		
cobol	20	6	5				1			1227		4							1	77						6
erlang	11		2	1				3	2											252						
fortran	14658	8	7900		4				2	14		4		1302			27			12	2				1	3
go	3			1		4			6	2					8					4			12	1	1	
haskell	10	2	3	5		14	1	2		7					5			2		10		4	4	13		
java	21642	11256	31778	27	2	9		23527	9		3744	12	4	7	4012	4311	1	12	12044	9263	1	3229	5	8702	334	12
kotlin										9														1		
lisp	27		7	5		4			5	5				1					2	838						
lua	22	9	16							9		1							2	10		1		2	2	
matlab	4353	2760	6387				1203		1	3838								6	1	5974	1191			1	1	8
node		1	1			1		24		8			1						4	7	1	3	1			
objective c	881	1941	7							3875		1			6		1			2		2			5216	
pascal	10286	7	1920				2			31			7	2						7		1				1
perl	6563	3	18			1		2	2	1163		1	165	1	2				1921	9409	5	4974	1	10		1
php	11014	1535	15	4				2573	1	15396			4	1	3828	3		24		5429		6427	2	17	2	
python	12079	5127	4933	97		843	1	39893	35	9723	1	13	271	4244	628	13	3	31	4524		5599	6787	15	100	8	
r	31513	1	4					1	2	2				974				2		4400				5		
ruby	485	7	10	122		4		5065	11	2430		1	4		21	1		2	1303	4031			1	1593	2	
rust	6		2		1			9	1																	
scala		3		5				1149	8	2779	7		1		3					5		6	3			
swift	3	2						1		5			4			1639				1						
visual basic	20	691	7				3			28			1					10	240	17		1		3		

# Popularita jazyků v budoucnosti

- Využití pro predikci popularity jazyků v budoucnosti (eigenvector)

16.41%	Go
14.26%	C
13.21%	Java
11.51%	C++
9.45%	Python

I took the stochastic matrix sorted by the *future popularity* of the language (as predicted by the first eigenvector).

# Analýza pro javascript frameworky, DBs...

