Inicializace paměti

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

04.12.2025

Neinicializované hodnoty

Neinicializované proměnné

Co se stane, když **neincializujeme proměnnou?**

Neinicializované proměnné

Co se stane, když **neincializujeme proměnnou?**

```c
int main() {
    int x;
    printf("%d\n", x);
}
```

Neinicializované proměnné

Co se stane, když **neincializujeme proměnnou?**

```c
int main() {
    int x;
    printf("%d\n", x);
}
```

```
32766
```

Neinicializované proměnné

Nastává stejný problém také u **struktur**?

```
typedef struct {
    long weight;
    char countOfDoors;
} Car;

int main() {
    Car c;
    printf("%ld %d\n", c.weight, c.countOfDoors);
}
```

Neinicializované proměnné

Nastává stejný problém také u **struktur**?

```
typedef struct {
    long weight;
    char countOfDoors;
} Car;

int main() {
    Car c;
    printf("%ld %d\n", c.weight, c.countOfDoors);
}
```

```
140732583230048 88
```

Neinicializované proměnné

Problém:

- Proměnná obsahuje **náhodná data** z paměti
  - Tzv. „garbage values“ (smetí)

Neinicializované proměnné

Problém:

- Proměnná obsahuje **náhodná data** z paměti
  - Tzv. „garbage values“ (smetí)
- Výsledek je **nedeterministický** (pokaždé jiný)

Neinicializované proměnné

Problém:

- Proměnná obsahuje **náhodná data** z paměti
  - Tzv. „garbage values“ (smetí)
- Výsledek je **nedeterministický** (pokaždé jiný)
  - Každé spuštění programu může mít jiný výsledek

Neinicializované proměnné

Problém:

- Proměnná obsahuje **náhodná data** z paměti
  - Tzv. „garbage values“ (smetí)
- Výsledek je **nedeterministický** (pokaždé jiný)
  - Každé spuštění programu může mít jiný výsledek
- Častá příčina bugů!

Správná inicializace

Vždy inicializujte proměnné a struktury!

Správná inicializace

Vždy inicializujte proměnné a struktury!

```
int x = 0;
struct Point p = { -1 };
```

Správná inicializace

Vždy inicializujte proměnné a struktury!

```
int x = 0;
struct Point p = { -1 };
```

- List initialization (C99) zajišťuje, že všechny (ostatní, nespecifikované) prvky jsou inicializovány na 0

Správná inicializace

Vždy inicializujte proměnné a struktury!

```
int x = 0;
struct Point p = { -1 };
```

- List initialization (C99) zajišťuje, že všechny (ostatní, nespecifikované) prvky jsou inicializovány na 0

Proto výše uvedený kód s inicializací bodu funguje stejně jako:

```
struct Point p = { -1, 0 };
```

Správná inicializace

```
Car c1 = { 0 }; // All members are initialized to 0
Car c2 = { .weight = 1000, .countOfDoors = 5 };
```

Vyjímka

Vyjímku pro nutnost inicializace tvoří

Vyjímka

Vyjímku pro nutnost inicializace tvoří **globální** a **statické** proměnné, ty jsou **vždy inicializované na 0**

```
int global;   // Always 0
static int s; // Always 0

void func() {
    int local; // Uninitialized = Garbage value
}
```

Statické proměnné a funkce

Klíčové slovo `static`

Klíčové slovo `static` má v C **dva hlavní účely**:

Klíčové slovo `static`

Klíčové slovo `static` má v C **dva hlavní účely**:

1. **Statické lokální proměnné** ve funkcích

Klíčové slovo `static`

Klíčové slovo `static` má v C **dva hlavní účely**:

1. **Statické lokální proměnné** ve funkcích
   - Zachovávají hodnotu mezi voláními funkce

Klíčové slovo `static`

Klíčové slovo `static` má v C **dva hlavní účely**:

1. **Statické lokální proměnné** ve funkcích
   - Zachovávají hodnotu mezi voláními funkce
2. **Statické globální proměnné/funkce**

Klíčové slovo `static`

Klíčové slovo `static` má v C **dva hlavní účely**:

1. **Statické lokální proměnné** ve funkcích
   - Zachovávají hodnotu mezi voláními funkce
2. **Statické globální proměnné/funkce**
   - Omezení viditelnosti pouze na aktuální soubor

Klíčové slovo `static`

Kde se ukládají?

Klíčové slovo `static`

Kde se ukládají?

- Statické proměnné jsou v **data** segmentu paměti

Klíčové slovo `static`

Kde se ukládají?

- Statické proměnné jsou v **data** segmentu paměti
- Existují po celou dobu běhu programu

Klíčové slovo `static`

Kde se ukládají?

- Statické proměnné jsou v **data** segmentu paměti
- Existují po celou dobu běhu programu
- Automaticky inicializované na 0 (pokud není uvedeno jinak)

Statické lokální proměnné

```c
int counter() {
    static int count;
    count++;
    return count;
}

int main() {
    printf("%d\n", counter());  // ?
    printf("%d\n", counter());  // ?
    printf("%d\n", counter());  // ?
}
```

Statické lokální proměnné

Výstupem předchozího kódu je:

```
1
2
3
```

Statické lokální proměnné

Výstupem předchozího kódu je:

```
1
2
3
```

Proměnná `count` je **statická** a tedy

- Je **inicializována** na 0 (nebylo uvedeno jinak) **pouze při prvním volání** funkce

Statické lokální proměnné

Výstupem předchozího kódu je:

```
1
2
3
```

Proměnná `count` je **statická** a tedy

- Je **inicializována** na 0 (nebylo uvedeno jinak) **pouze při prvním volání** funkce
- **Pamatuje si** svou hodnotu mezi voláními funkce!

Porovnání statické a lokální proměnné

```c
// Without static
int func() {
    int x = 0;
    x++;
    return x;
}

func(); // 1
func(); // 1
func(); // 1
```

```c
// With static
int func() {
    static int x = 0;
    x++;
    return x;
}

func(); // 1
func(); // 2
func(); // 3
```

Porovnání statické a lokální proměnné

```c
// Without static
int func() {
    int x = 0;
    x++;
    return x;
}

func(); // 1
func(); // 1
func(); // 1
```

```c
// With static
int func() {
    static int x = 0;
    x++;
    return x;
}

func(); // 1
func(); // 2
func(); // 3
```

Klíčový rozdíl: Statická proměnná „přežívá“ ukončení funkce

`static` pro omezení viditelnosti

Statické globální proměnné a statické funkce jsou „viditelné“ pouze v aktuálním souboru

`static` pro omezení viditelnosti

Statické globální proměnné a statické funkce jsou „viditelné“ pouze v aktuálním souboru

```
// file: module.c
static int internalCounter = 0;  // Visible only in module.c
static void helperFunction() {   // Visible only in module.c
    internalCounter++;
}

void publicFunction() {          // Visible everywhere
    helperFunction();
}
```

`static` pro omezení viditelnosti

Výhody:

- Zapouzdření (encapsulation)
- Prevence konfliktů jmen mezi soubory
- Lepší modularita kódu