Práce s pamětí, zásobník, halda

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Ukazatele, pole v dynamické paměti
  - Modifikátor const
- Část 2 – Práce s pamětí, pamětové třídy
  - Výpočetní prostředky, paměť
  - Rozsah platnosti proměnných
  - Alokace dynamické paměti
- Část 3 – Ladění
  - GDB
  - Valgrind
- Část 4 – Zadání 6. domácího úkolu

Část I

Ukazatele a práce s pamětí

I. Ukazatele a práce s pamětí

Modifikátor const

Modifikátor `const`

- Uvedením klíčového slova const můžeme označit proměnnou jako konstantu
  - Překladač kontroluje přiřazení
- Pro definici konstant můžeme použít např.

```
const float pi = 3.14159265;
```

- Na rozdíl od symbolické konstanty

```
#define PI 3.14159265
```

  - mají konstantní proměnné typ
  - překladač tak může provádět typovou kontrolu

Ukazatele na konst. proměnné a konst. ukazatele

- Klíčové slovo `const` můžeme zapsat před jméno typu nebo před jméno proměnné
- Dostáváme 3 možnosti jak definovat ukazatel s `const`
  1. `const int *ptr;` – ukazatel na konstantní proměnnou
     - Nemůžeme použít pointer pro změnu hodnoty proměnné
     - `const int *` lze též zapsat jako `int const *`
  2. `int *const ptr;` – konstantní ukazatel
     - Pointer nemůžeme nastavit na jinou adresu než tu při inicializaci
  3. `const int *const ptr;` – konstantní ukazatel na konstantní hodnotu
     - Kombinuje předchozí dva případy
     - `const int * const` lze též zapsat jako `int const * const`

Ukazatel na konstantní proměnnou (hodnotu)

- Prostřednictvím ukazatele na konstantní proměnnou nelze tuto proměnnou měnit

```
int v = 10;
int v2 = 20;
const int *ptr = &v;
printf("*ptr: %d\n", *ptr);
*ptr = 11;  /* NELZE! */
v = 11;     /* lze menit promennou */
printf("*ptr: %d\n", *ptr);
ptr = &v2; /* lze priradit novou adresu ukazateli */
printf("*ptr: %d\n", *ptr);
```

Konstatní ukazatel

- Hodnotu konstantního ukazatele nelze po inicializaci měnit
- Zápis `int *const ptr;` můžeme číst zprava doleva
  - `ptr` – proměnná, která je `*const` – konstantním ukazatelem
  - `int` – na proměnnou typu int

```
int v = 10;
int v2 = 20;
int *const ptr = &v;
printf("v: %d *ptr: %d\n", v, *ptr);
*ptr = 11; /* lze zmenit odkazovanou promennou */
printf("v: %d\n", v);
ptr = &v2; /* NELZE! */
```

Konstantní ukazatel na konstantní proměnnou

- Hodnotu konstantního ukazatele na konstantního proměnnou nelze po inicializaci měnit a ani nelze prostřednictvím takového ukazatele měnit hodnotu adresované proměnné.
- Zápis `const int *const ptr;` můžeme číst zprava doleva

  `ptr` – proměnná, která je `*const` – konstantním ukazatelem `const int` – na proměnnou typu const int

```
int v = 10;
int v2 = 20;
const int *const ptr = &v;
printf("v: %d *ptr: %d\n", v, *ptr);
ptr = &v2; /* NELZE! */
*ptr = 11; /* NELZE! */
```

Část II

Práce s pamětí, paměťové třídy

II. Práce s pamětí, paměťové třídy

Výpočetní prostředky, paměť

Rozsah platnosti proměnných

Alokace dynamické paměti

Rozdělení paměti

1. **Zásobník (stack)**
   - lokální proměnné, argumenty funkcí, návratová hodnota funkce

   spravováno automaticky
2. **Halda (heap)**
   - dynamická paměť

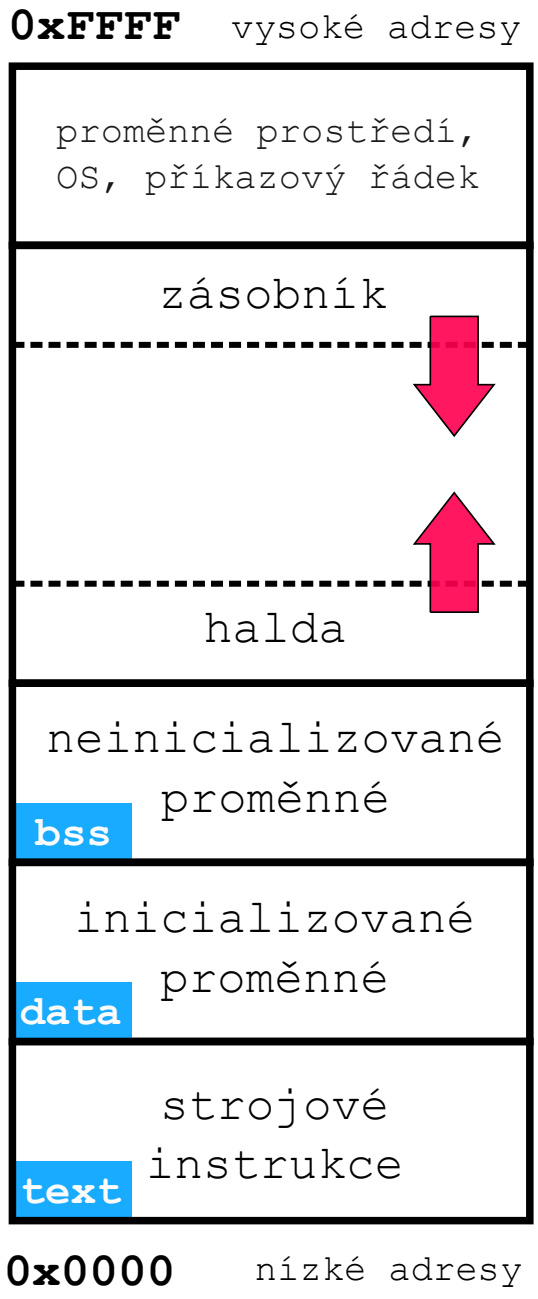
   spravuje programátor
3. **Statická (bss)**
   - globální nebo "lokální" static proměnné

   inicializace při startu na 0

   block started by symbol
4. **Literály (data, data segment)**
   - hodnoty zapsané ve zdrojovém kódu programu, např. textové řetězce

   inicializace při startu, RO/RW
5. **Program (text, code segment)**
   - strojové instrukce

   inicializace při startu, RO



Přidělování paměti proměnným

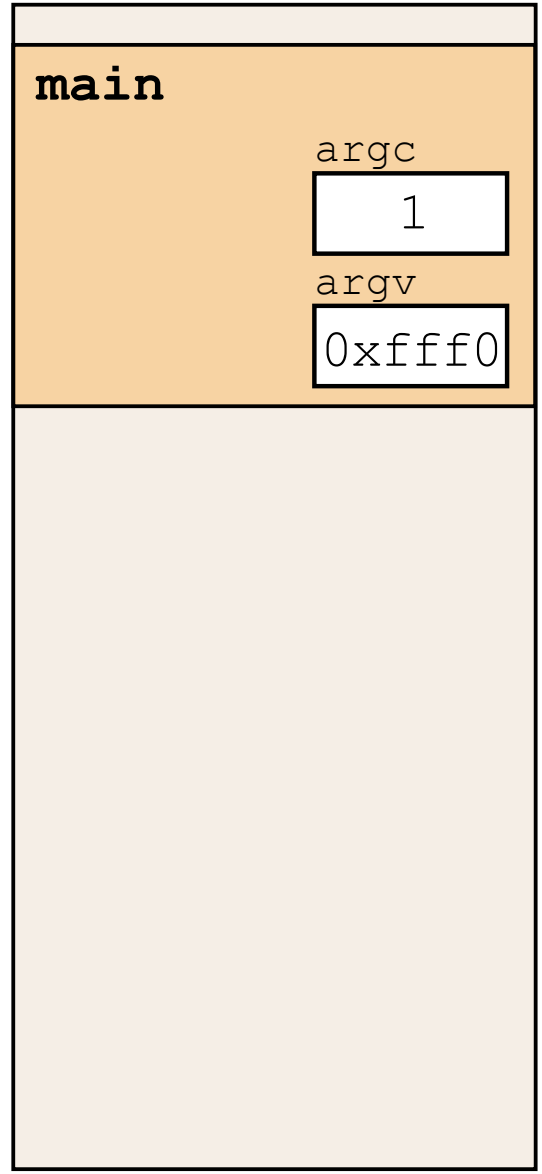
- Určení paměťového místa pro uložení hodnoty proměnné v paměti
- Lokálním proměnným a parametrům funkce se paměť přiděluje při volání funkce
  - Paměť zůstane přidělena jen do návratu z funkce
  - Paměť se automaticky alokuje z rezervovaného místa – zásobník
  - Při návratu funkce se přidělené paměťové místo uvolní
  - Výjimku tvoří lokální proměnné s modifikátorem static
    - Z hlediska platnosti rozsahu mají charakter lokálních proměnných
    - Jejich hodnota je však zachována i po skončení funkce / bloku
    - Jsou umístěny ve statické části paměti
- Dynamické přidělování paměti
  - Alokace paměti se provádí funkcemi standardní knihovny
  - Paměť se alokuje z rezervovaného místa – halda

Zásobník

- Úseky paměti přidělované lokálním proměnným a parametrům
  - Úseky se přidávají a odebírají
  - Vždy se odebere naposledy přidaný úsek – LIFO (last in, first out)
  - Na zásobník se ukládá "volání funkce"
- Na zásobník se ukládá
  - návratová hodnota funkce
  - hodnota čítače programu před voláním funkce
- Ze zásobníku se alokují proměnné parametrů funkce
  - Argumenty (parametry) jsou de facto lokální proměnné
  - Opakovaným rekurzivním voláním funkce můžeme zaplnit velikost přiděleného zásobníku program skončí chybou.

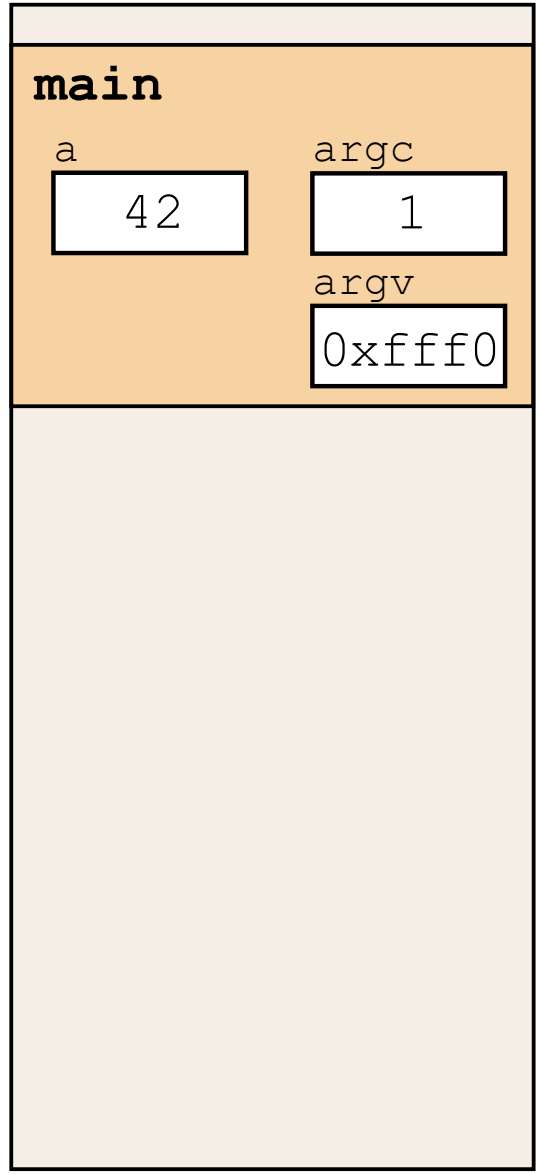
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



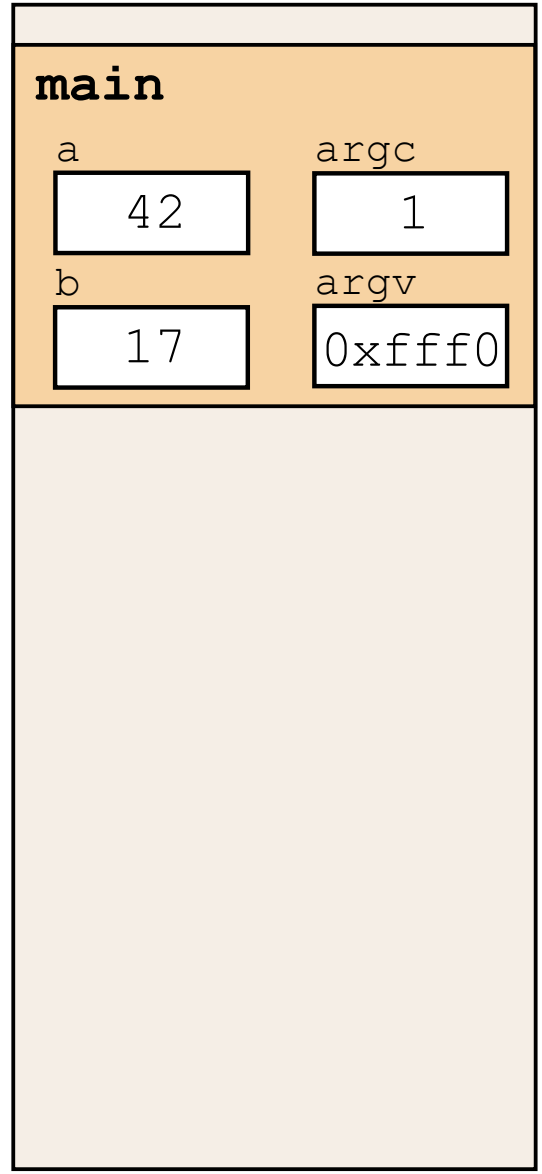
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
16 void func2()
17 {
18   int d = 0;
19 }
```



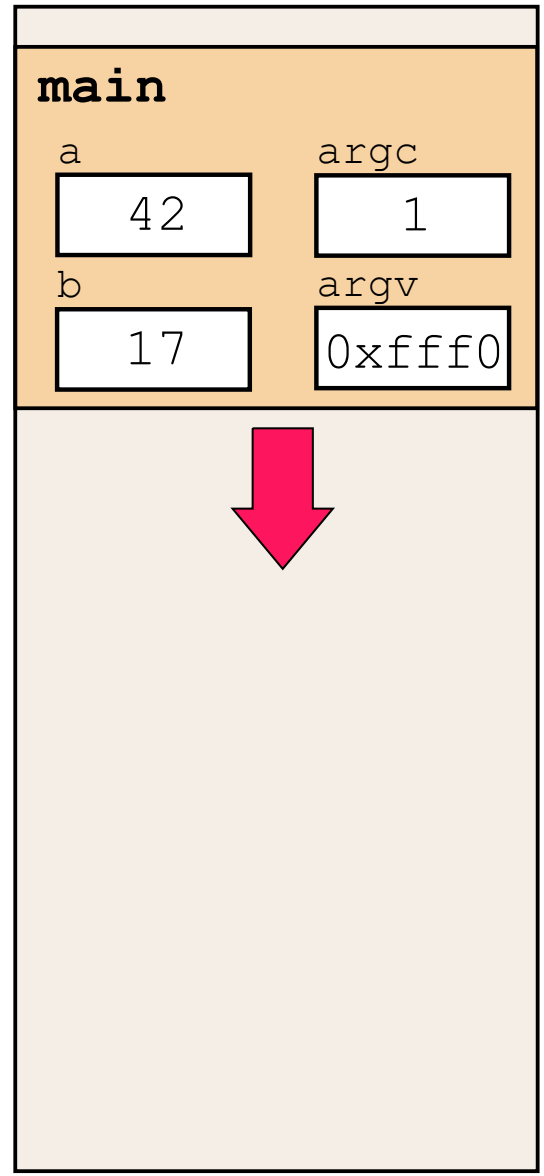
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



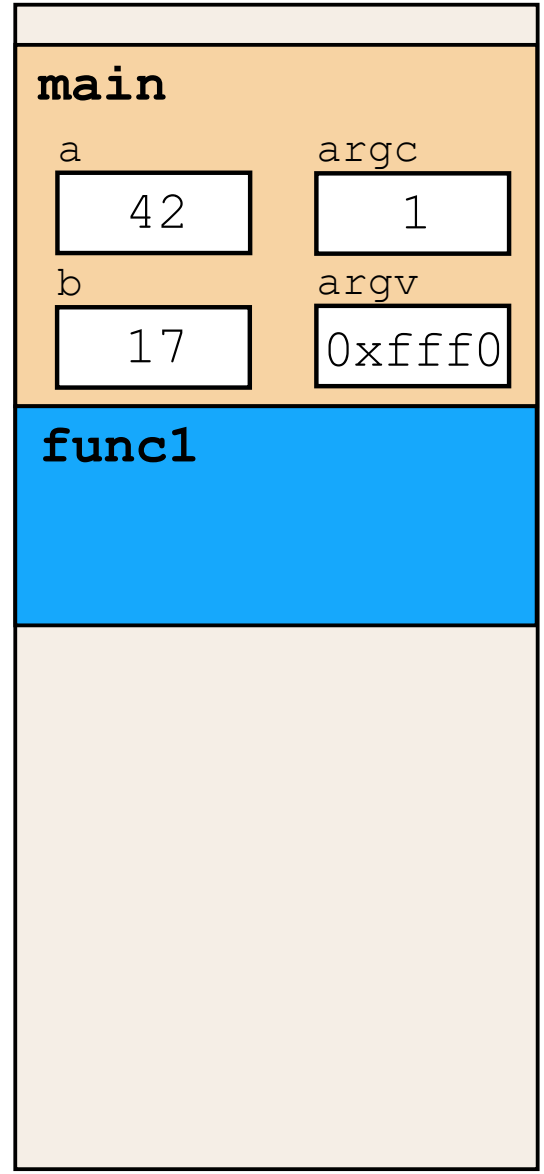
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



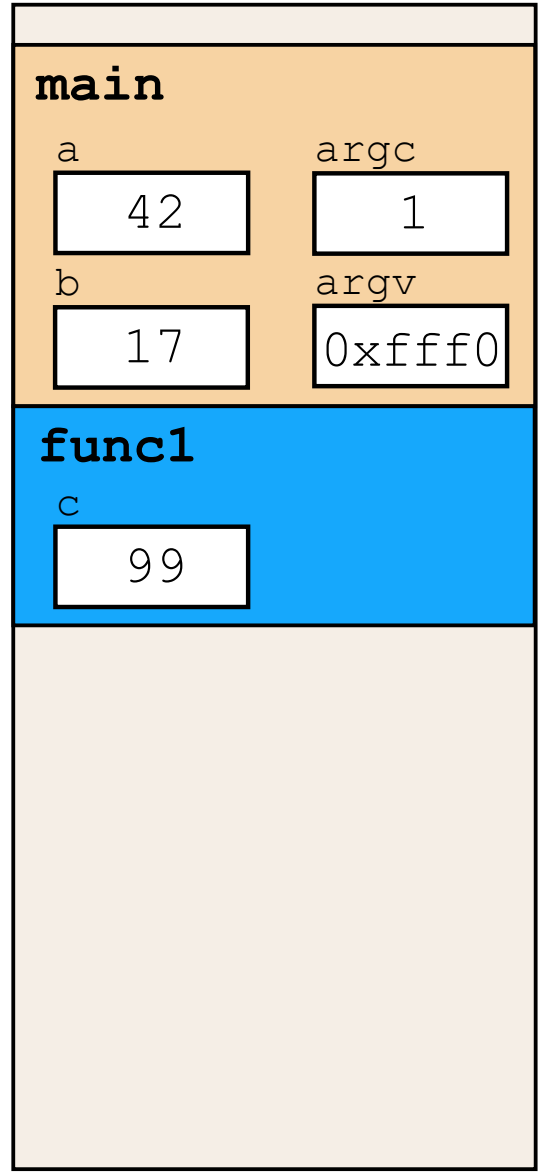
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }

10 void func1()
11 {
12   int c = 99;
13   func2();
14 }

16 void func2()
17 {
18   int d = 0;
19 }
```



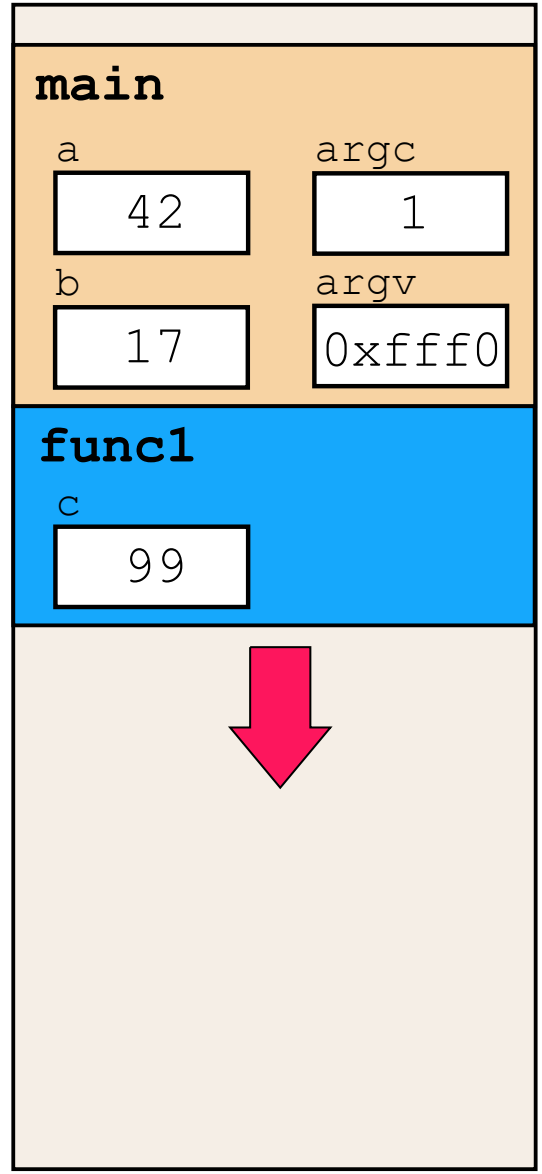
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }

10 void func1()
11 {
12   int c = 99;
13   func2();
14 }

16 void func2()
17 {
18   int d = 0;
19 }
```



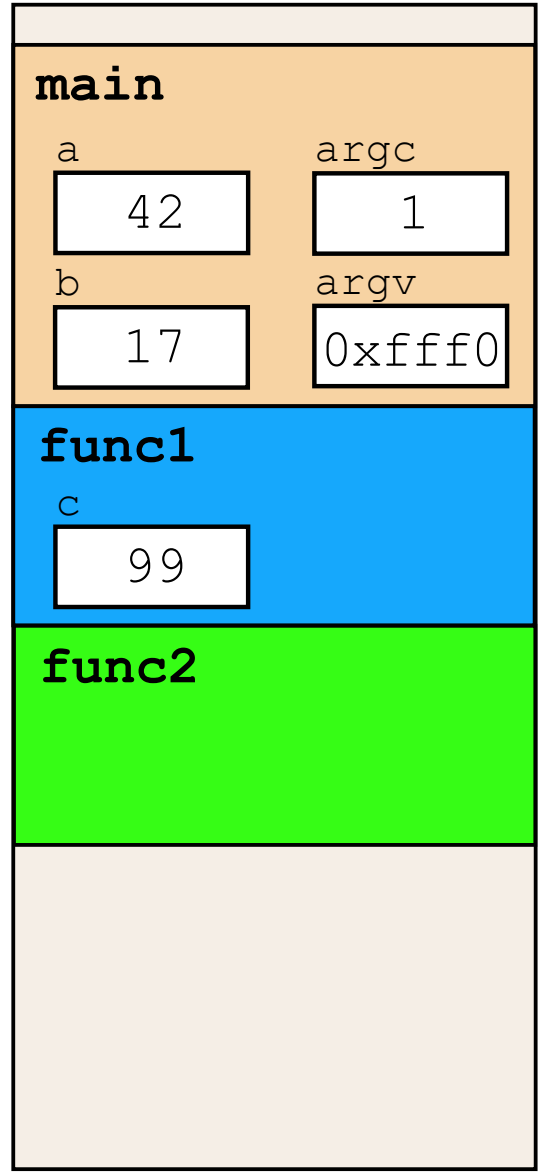
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }

10 void func1()
11 {
12   int c = 99;
13   func2();
14 }

16 void func2()
17 {
18   int d = 0;
19 }
```



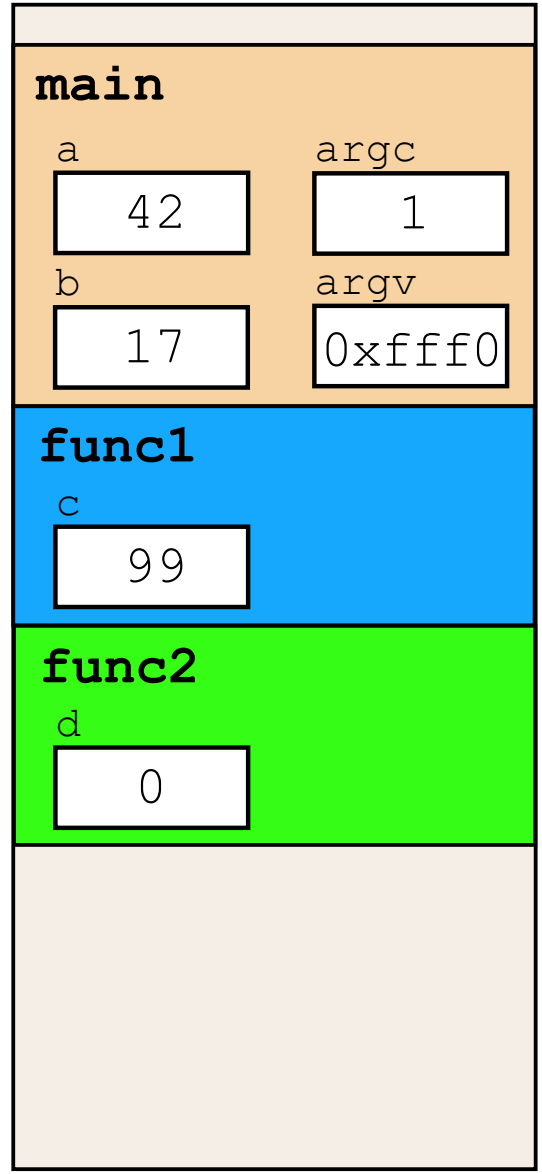
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }

10 void func1()
11 {
12   int c = 99;
13   func2();
14 }

16 void func2()
17 {
18   int d = 0;
19 }
```



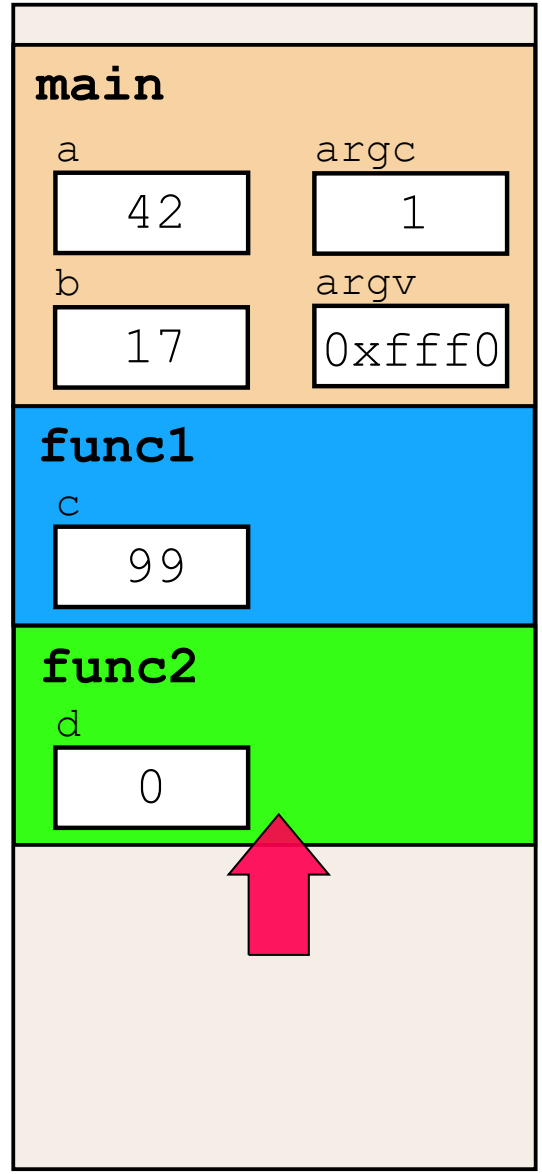
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }

10 void func1()
11 {
12   int c = 99;
13   func2();
14 }

16 void func2()
17 {
18   int d = 0;
19 }
```



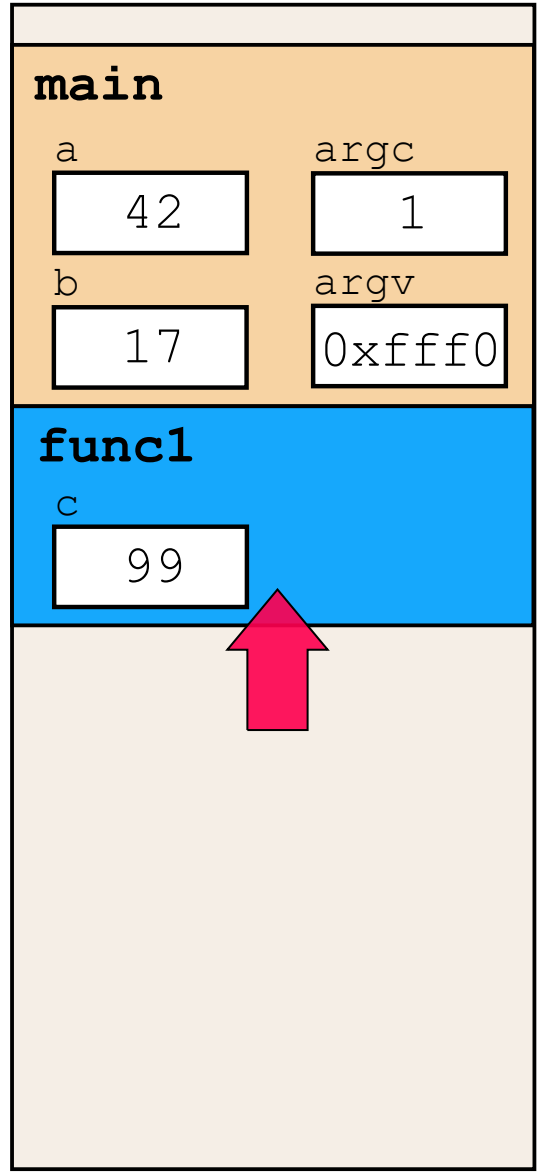
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
16 void func2()
17 {
18   int d = 0;
19 }
```



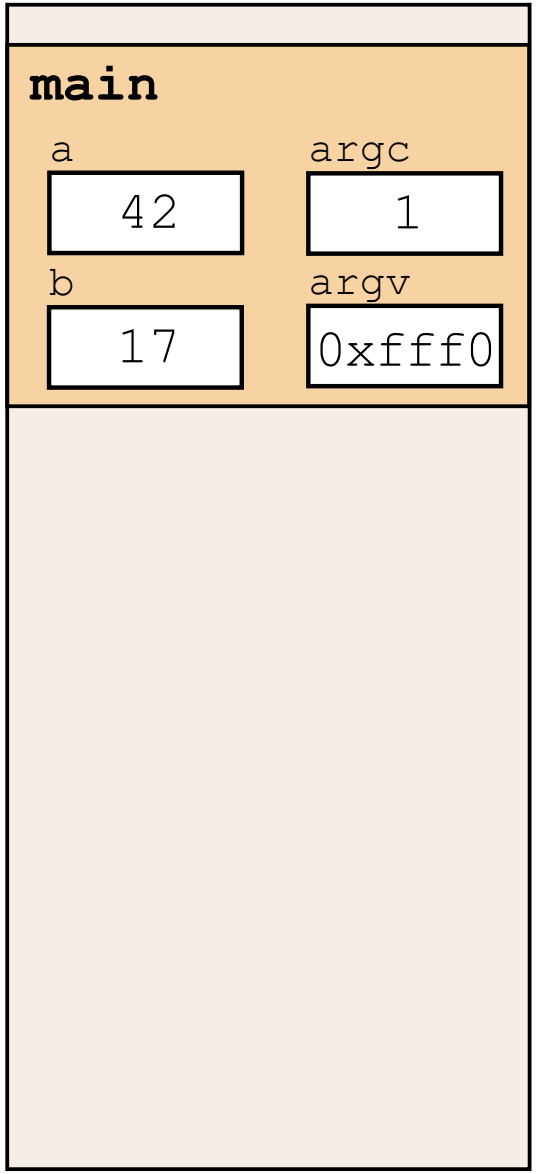
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
16 void func2()
17 {
18   int d = 0;
19 }
```



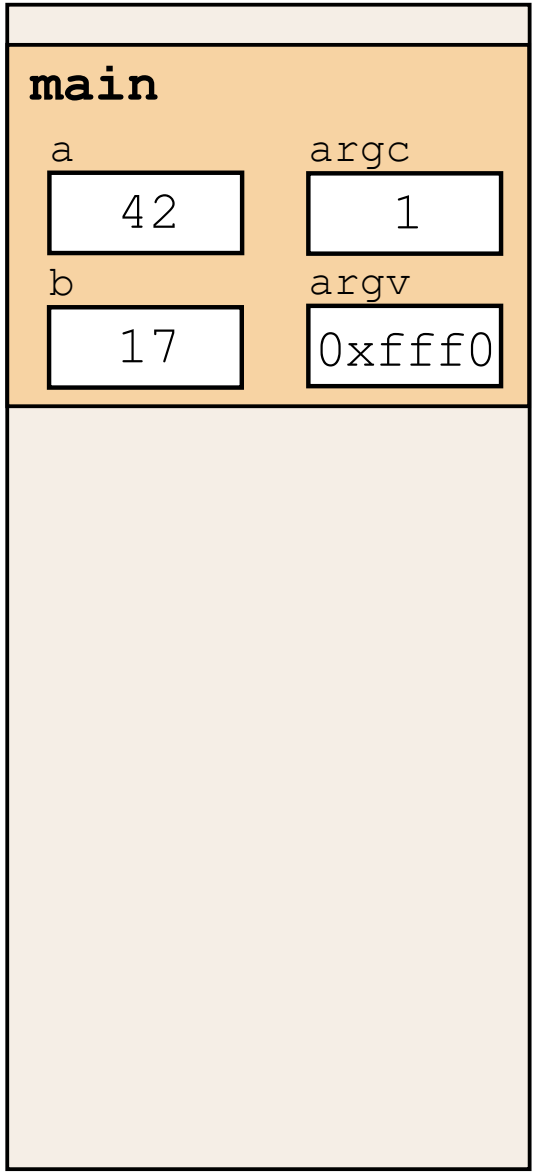
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



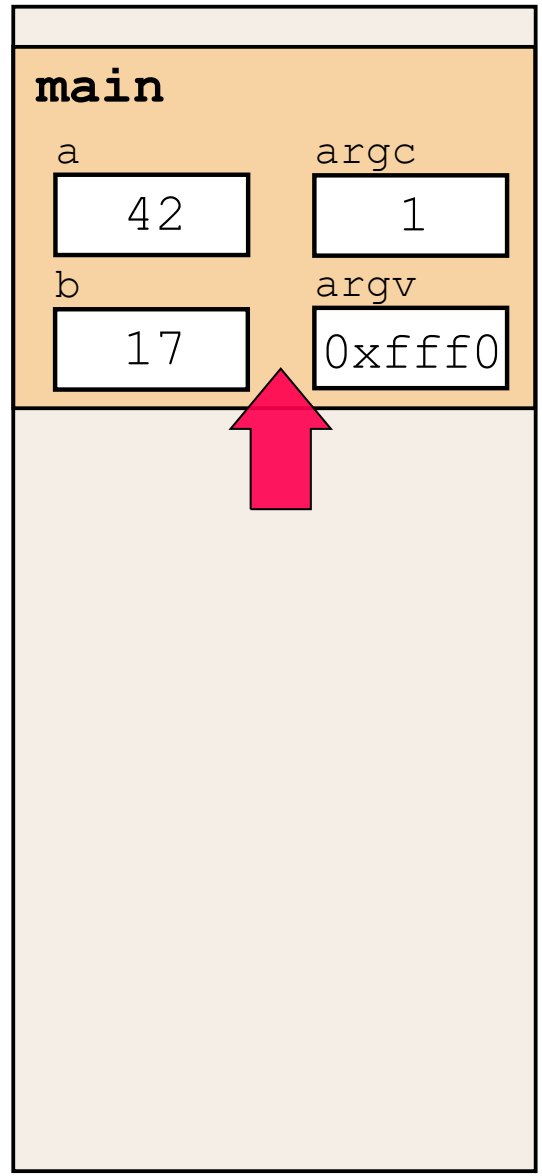
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



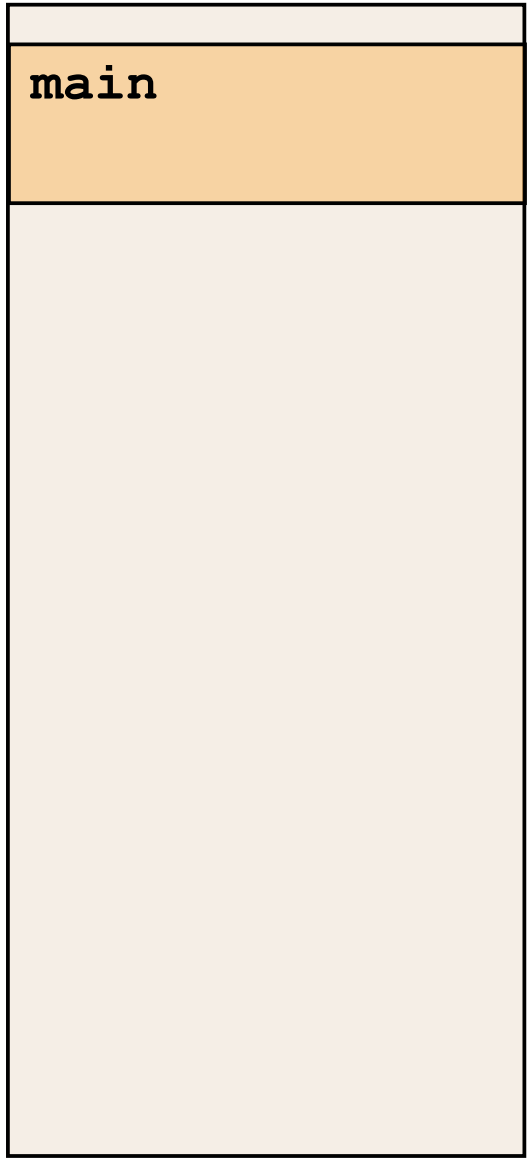
Příklad – lokální proměnné při volání funkce

```
1  int main(int argc, char *argv[])
2  {
3    int a = 42;
4    int b = 17;
5    func1();
6    printf("Done.");
7    return 0;
8  }
9
10 void func1()
11 {
12   int c = 99;
13   func2();
14 }
15
16 void func2()
17 {
18   int d = 0;
19 }
```



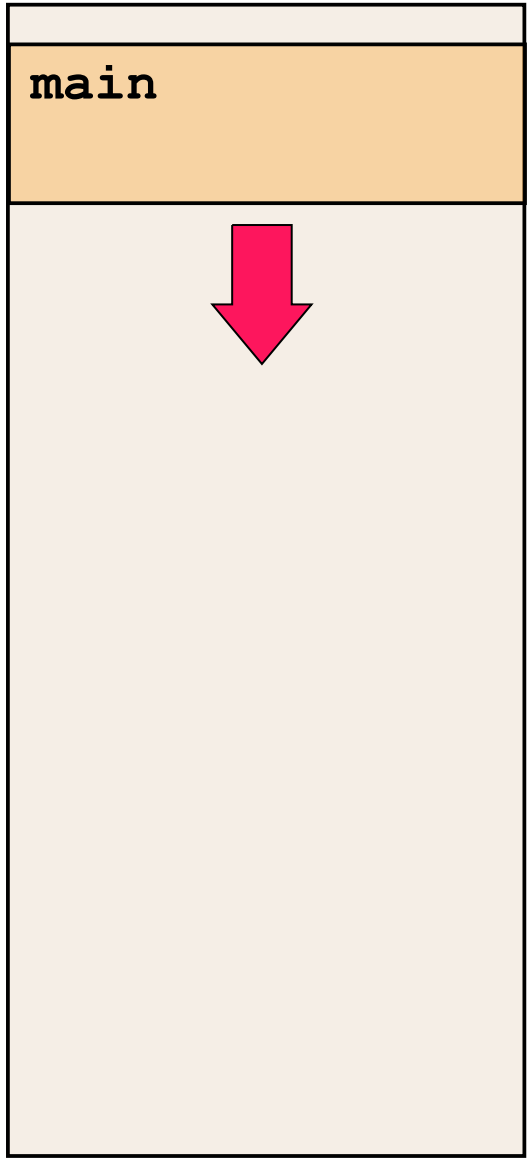
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



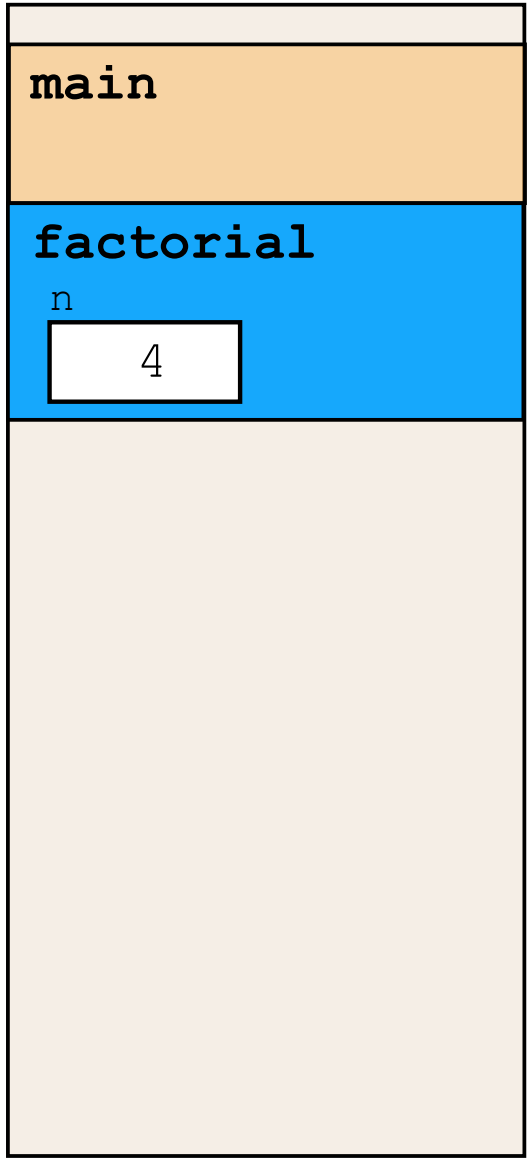
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



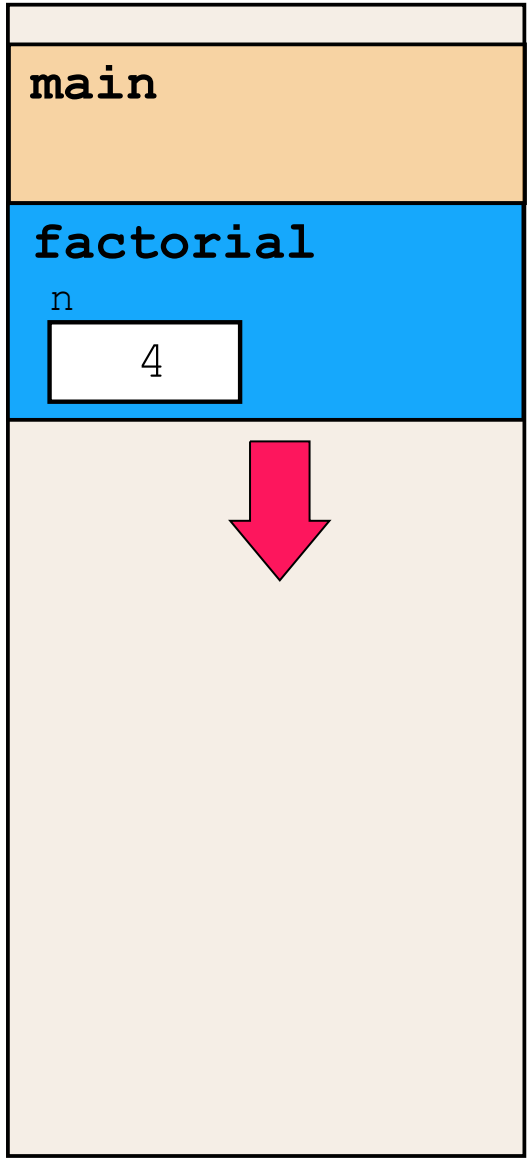
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



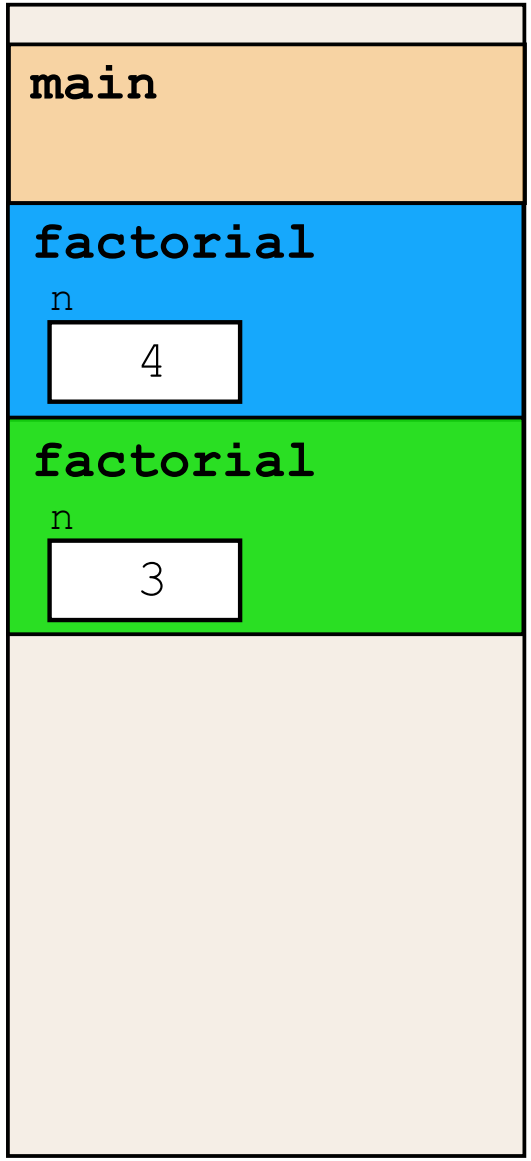
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



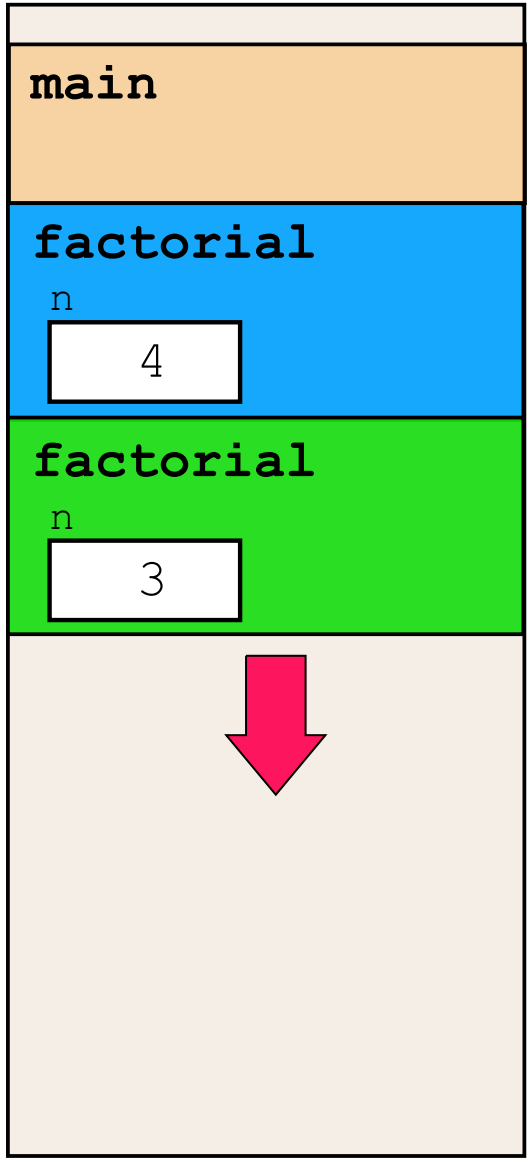
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



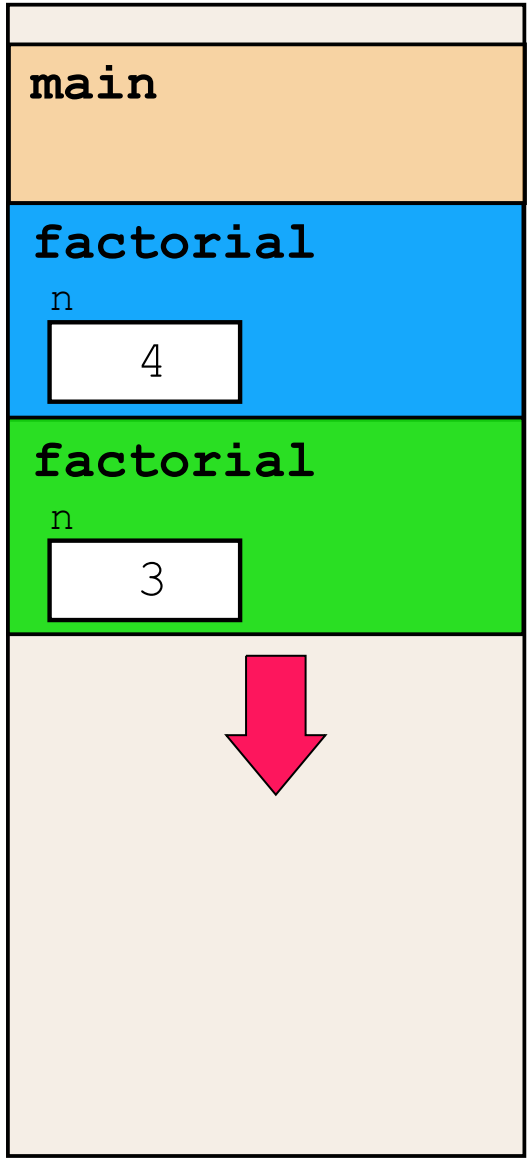
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



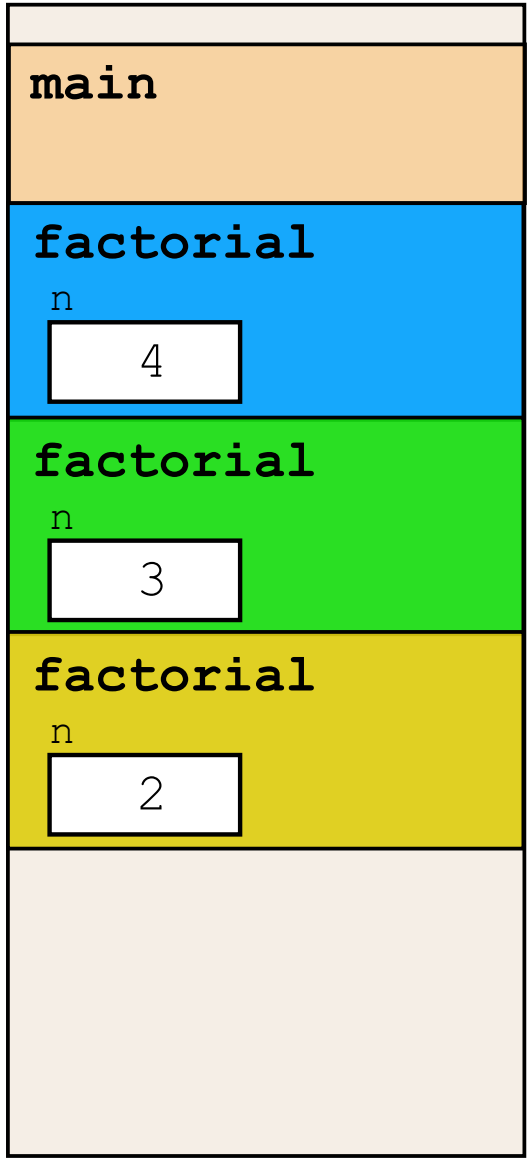
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



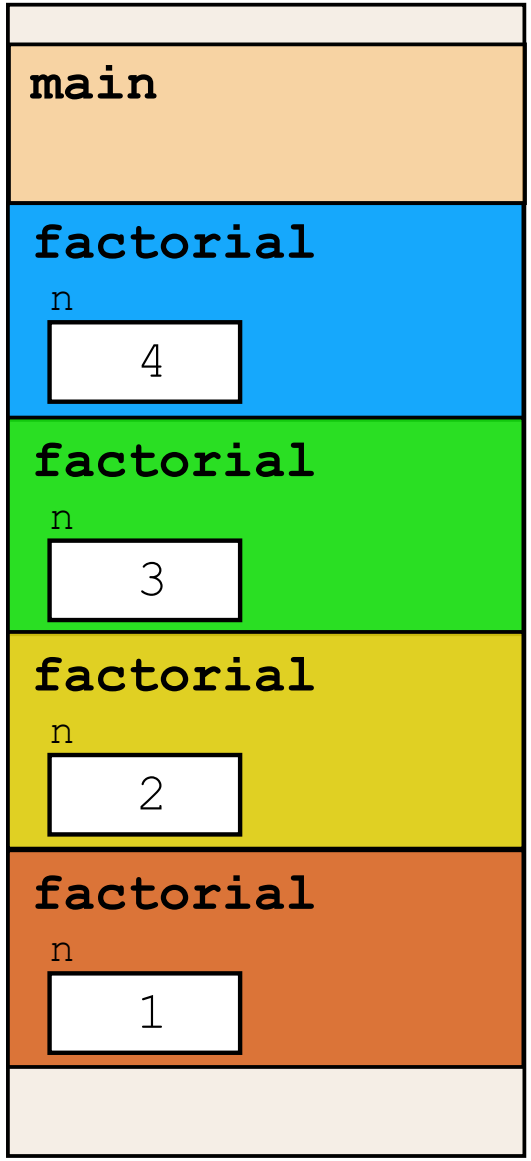
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



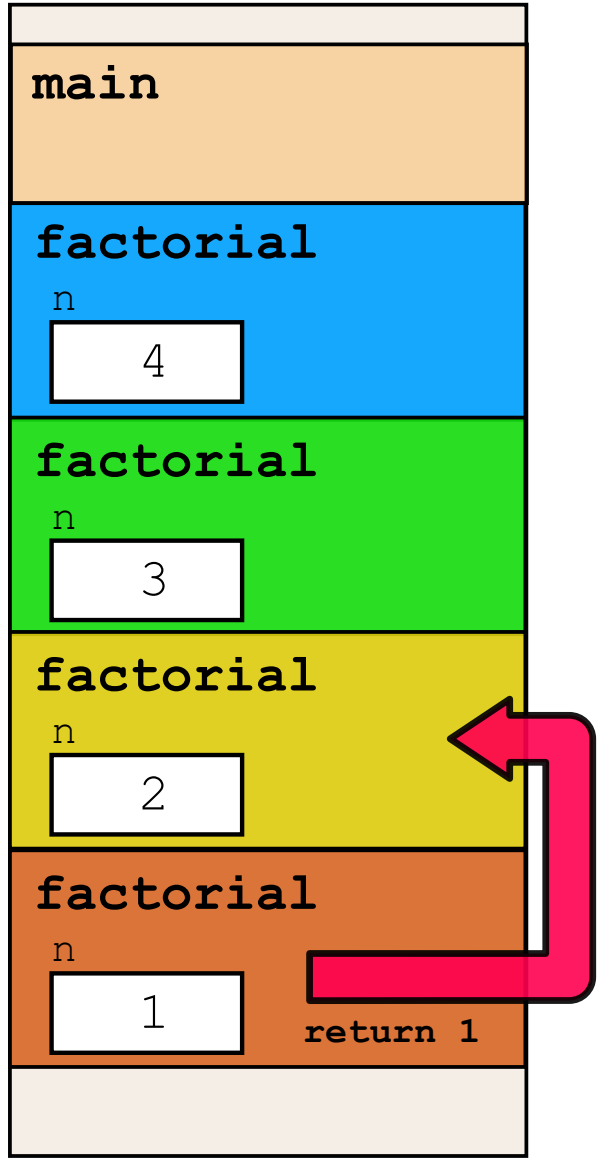
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



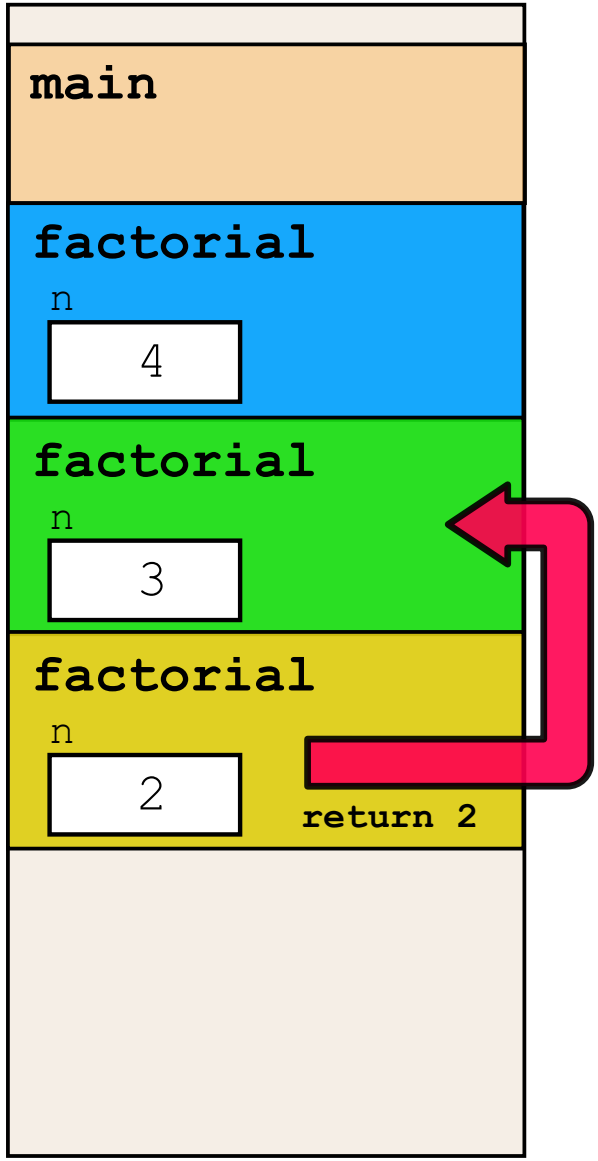
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



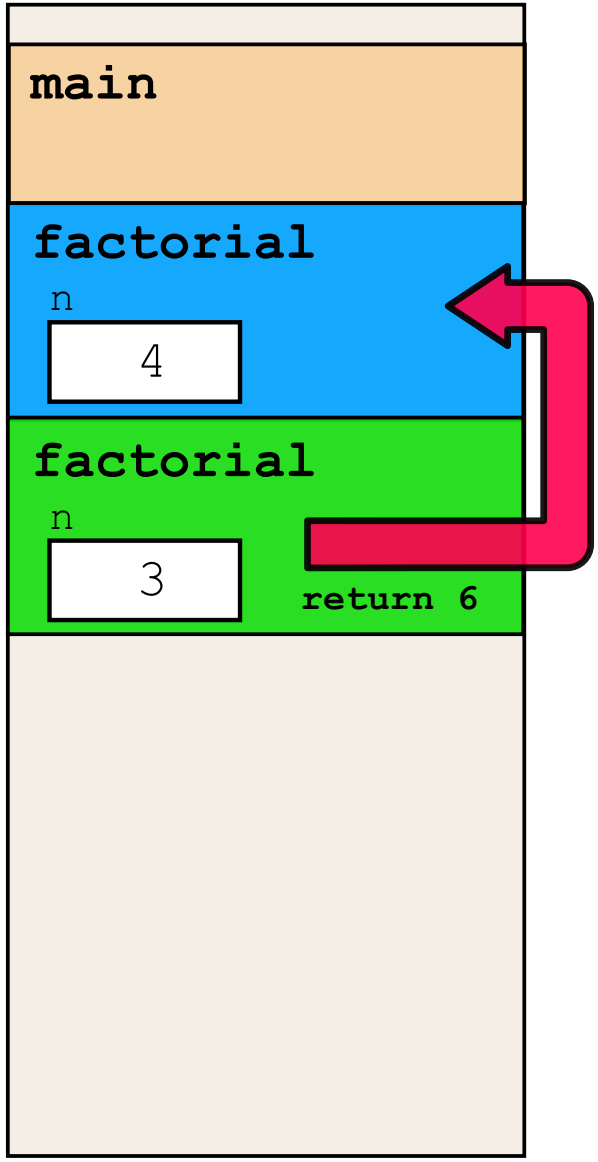
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



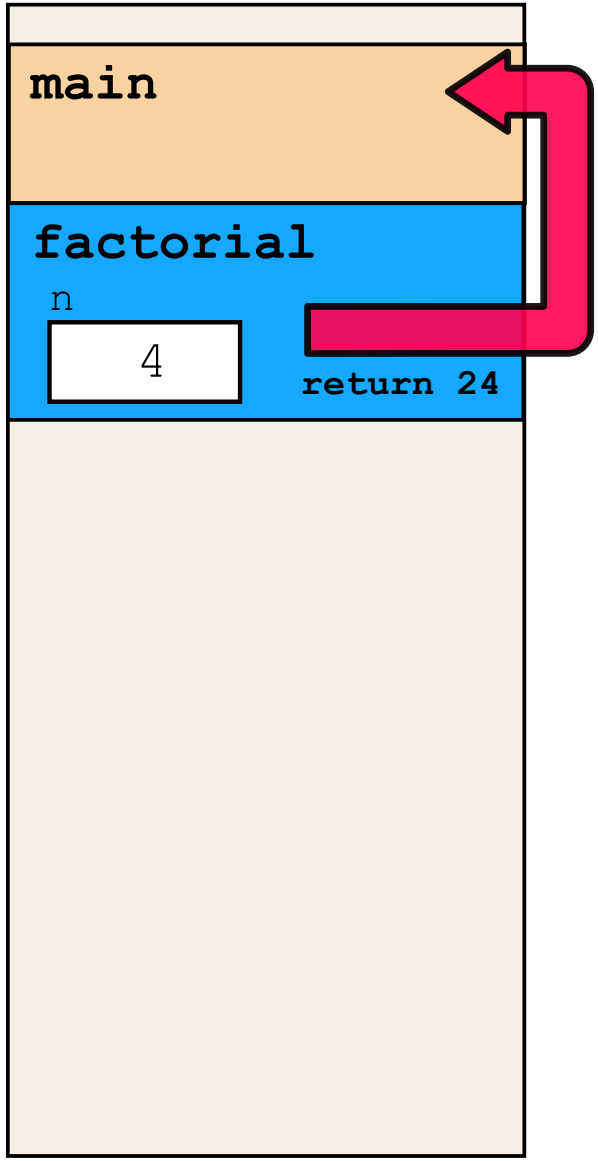
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



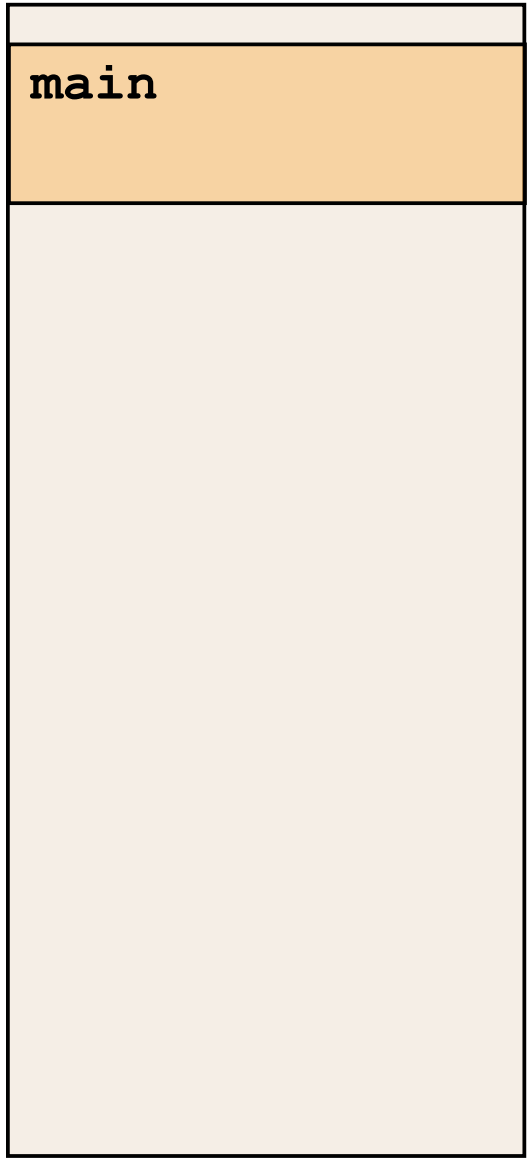
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



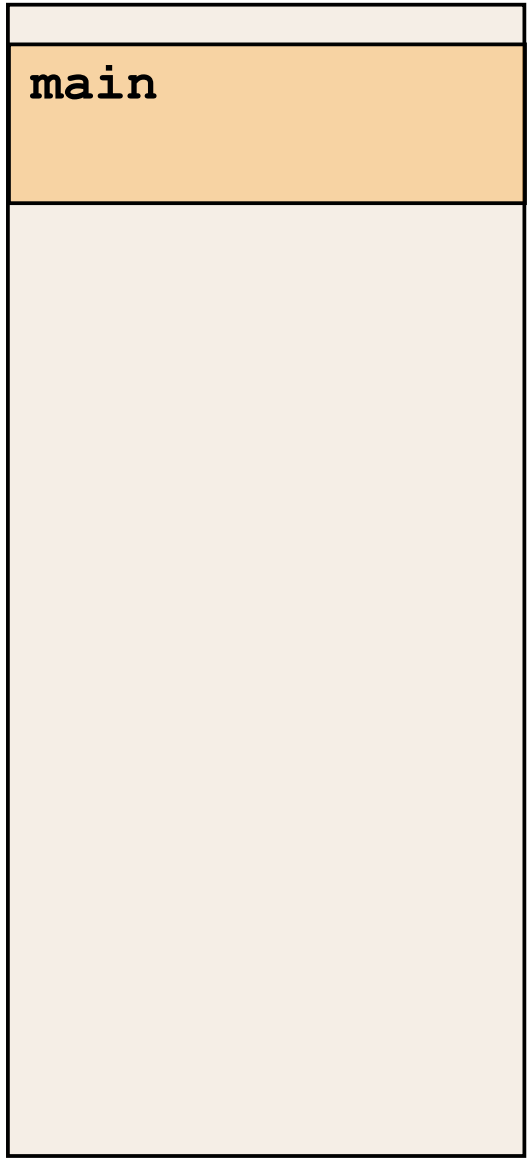
Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



Příklad – rekurzivní volání funkce

```
1  #include <stdio.h>
2
3  int factorial (int n)
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n * factorial(n-1);
9  }
10
11 int main (void)
12 {
13   printf("%d", factorial(4));
14   return 0;
15 }
```



Rekurzivní volání funkce

```c
#include <stdio.h>
void funkce(int v)
{
  printf("hodnota: %i\n", v);
  funkce(v + 1);
}
int main(void)
{
  funkce(1);
}
```

- Vyzkoušejte si program pro omezenou velkost zásobníku

  ```
  $ ulimit -s 1000
  ```

Proměnná

Vymezená oblast paměti a v C je můžeme rozdělit podle způsobu alokace

- **Statická alokace**
  - provede se při deklaraci statické nebo globální proměnné
  - paměťový prostor je alokován při startu programu a nikdy není uvolněn
- **Automatická alokace**
  - probíhá automaticky v případě lokálních proměnných (nebo argumentů funkce)
  - paměťový prostor je alokován na zásobníku a paměť proměnné je automaticky uvolněna s koncem platnosti proměnné. např. po ukončení bloku funkce.
- **Dynamická alokace**
  - není podporována přímo jazykem C, ale je přístupná knihovními funkcemi (stdlib, malloc)

Paměťové třídy

- **auto** (lokální)
  - definuje proměnnou jako dočasnou (automatickou)
  - typicky lokální proměnná deklarovaná uvnitř funkce
  - implicitní nastavení, platnost proměnné je omezena na blok
  - proměnná je v zásobníku.
- **register**
  - doporučuje překladači umístit proměnnou do registru procesoru.
  - překladač může, ale nemusí vyhovět
  - nelze ziskat adresu
  - jinak stejné jako auto
- **static**
  - **uvnitř bloku** – proměnnou je statická – ponechává si hodnotu i při opuštění bloku. Je uložena v datové oblasti.
  - **vně bloku** – kde je implicitně proměnná uložena v datové oblasti (statická) omezuje její viditelnost na modul.
- **extern**
  - rozšiřuje viditelnost statických proměnných z modulu na celý program
  - globální proměnné s extern jsou definované v datové oblasti

Příklad deklarace proměnných

```
// program.h
extern int globalni;  // deklarace
// extern int globalni = 10; by bylo definici

// program.c
int globalni = 10;

void funkce() {
  int lokalni = 0;     // lokalni promenna
  int statlok = 0;     // staticka lokalni promenna
  printf("lokalni: %d, staticka: %d\n",
    ++lokalni, ++statlok);
}

int main() {
  funkce();
  funkce();
  funkce();
}
```

II. Práce s pamětí, paměťové třídy

Výpočetní prostředky, paměť

Rozsah platnosti proměnných

Alokace dynamické paměti

Rozsah platnosti proměnných

```
int a = 10;       // globalni promenna
int main ()
{                 // začátek bloku 1
  int a = 100;    // lokalni promenna, zastini globalni
  {               // zacatek bloku 2
    int a = 1, b = 2;
    a +=  b;      // vysledek?
  }               // konec bloku 2
  b = 20;         // promenna b nena platna
}                   // konec bloku 1
```

- Globální proměnné mají rozsah platnosti "kdekoliv" v programu
- Zastíněný přístup lze řešit modifikátorem `extern` (v novém bloku)

Definice vs. deklarace

- platí pro proměnné i funkce
- definice je přidělení paměťového místa
- deklarace je oznámení, že proměnná (funkce) je někde definována
- Zřejmě:
  - definici je možné provést pouze jednou
  - pokus o vícenásobnou definici skončí chybou překladu (linkování) programu

Definice vs. deklarace

```c
// definice.h
int global = 5;
int funkce (int);

// definice.c
#include "definice.h"
static int modul;

int funkce (int a)
{
  printf ("arg: %d, global: %d",
     a, global);
  return 0;
}
```

```c
#include "definice.h"

int main ()
{
  global += 1;
  funkce (1);
  funkce (1);
  global += 1;
  funkce (1);
  return 0;
}
```

II. Práce s pamětí, paměťové třídy

Výpočetní prostředky, paměť

Rozsah platnosti proměnných

Alokace dynamické paměti

Alokace dynamické paměti

- Přidělení bloku paměti velikosti `size` lze realizovat funkcí

```
void* malloc(size);
```

z knihovny `<stdlib.h>`

  - Velikost alokované pameti je uložena ve správci pameti
  - Velikost není soucástí ukazatele
  - Návratová hodnota je typu `void*` – pretypování nutné
- Příklad alokace paměti pro 10 proměnných typu int

```
int *int_array;
int_array = (int*)malloc(10 * sizeof(int));
```

- Operace s více hodnotami v pamětovém bloku je podobná poli
  - Používáme pointerovou aritmetiku
- Uvolnění paměti

```
void* free(pointer);
```

  - Správce paměti uvolní pamět asociovanou k ukazateli
  - Hodnotu ukazatele však nemění!

Stále obsahuje predešlou adresu, která však již není platná.

Příklad alokace dynamické paměti 1/3

- Alokace se nemusí nutne povést – testujeme návratovou hodnotu funkce `malloc()`
- Pro vyplnění adresy alokované paměti předáváme proměnnou jako ukazatel na proměnnou typu ukazatel na int

```
void* allocate_memory(int size, void **ptr)
{
// ukazatel **ptr k uchovani ukazatele na nove alokovanou
// pamet (t.j.  adresu mista, kde je uchovana adresa)
  *ptr = malloc(size);
  if (*ptr == NULL) {
    fprintf(stderr, "Error: allocation fail");
    exit(-1); /* alokace se nepovedla, program konci */
  }
  return *ptr;
}
```

Příklad alokace dynamické paměti

- Pro vyplnění hodnot pole alokovaného v dynamické paměti stačí předávat hodnotu adresy paměti pole

```
void fill_array(int* array, int size) {
  for (int i = 0; i < size; ++i) {
    *(array++) = random();
  }
}
```

- Po uvolnění paměti odkazuje ukazatel stále na původní adresu, proto lze explicitne nulovat.
- Předání ukazatele na ukazatele je nutné, jinak nulovat nelze.

```
void deallocate_memory(void **ptr) {
  if (ptr != NULL && *ptr != NULL) {
    free(*ptr);
    *ptr = NULL;
  }
}
```

Příklad alokace dynamické paměti

```
1  int main(int argc, char *argv[])
2  {
3    int *int_array;
4    const int size = 4;
6    allocate_memory(sizeof(int) * size, (void**)&
       int_array);
7    fill_array(int_array, size);
8    int *cur = int_array;
9
10   for(int i = 0; i < size; ++i, cur++) {
11     printf("Array[%d] = %d\n", i, *cur);
12   }
13   deallocate_memory((void**)&int_array);
14   return 0;
15 }
```

Část III

Ladění

III. Ladění

GDB

Valgrind

GDB – spuštění

- řádkově orientovaný debugger
- existuje grafická nadstavba **ddd** a semigrafické **gdbtui**
- je třeba kompilovat s debugovacími symboly (`-g`)

```
int main()
{
  int i = 1337;
  return 0;
}
```

```
$ gcc -g program.c
$ gdb ./a.out
```

Než pořádně začneme

```
(gdb) print 1 + 2
$1 = 3

(gbd) print (int) 2147483648
$2 = -2147483648
```

GDB – základní příkazy

run – spustí běh

list – ukáže 10 řádků kódu

break [název funkce nebo číslo řádku] – nastaví breakpoint

clear [název funkce nebo číslo řádku] – smaže breakpoint

info break – zobrazí seznam breakpointů

step – provede jeden krok program (zkratka s)

step [počet kroků] – provede uvedený počet kroků programu

backtrace – vypíšte backtrace

info locals – zobrazí lokální proměnné

info args – zobrazí argumenty rámce

info variables – zobrazí všechny statické a globální proměnné

info functions – zobrazí všechny definované funkce

GDB - základní práce, nastavení breakpointů

```
(gdb) break main
(gdb) run
```

Program se zastavil na řádce 3, těsně před inicializací proměnné i

---

```
(gdb) print i
$3 = 32767
```

Výpis obecně náhodné hodnoty

```
(gdb) next
(gdb) print i
$4 = 1337
```

Posun o jeden řádek, proměnná i je již inicializovaná

GDB – inspekce paměti

```
(gdb) print &i
$5 = (int *) 0x7fff5fbff584
(gdb) print sizeof(i)
$6 = 4
```

Zjevně disponuji strojem, kde má int 4 byty

```
(gdb) x/4xb &i
0x7fff5fbff584:  0x39 0x05 0x00 0x00
```

4 bajty od adresy &i, little endian!

```
(gdb) set var i = 0x12345678
(gdb) x/4xb &i
0x7fff5fbff584:  0x78 0x56 0x34 0x12
```

GDB – inspekce datových typů

```
(gdb) ptype i
type = int

(gdb) ptype &i
type = int *

(gdb) ptype main
type = int (void)
```

GDB - pole a ukazatele

```
int main()
{
  int a[] = {1,2,3};
  return 0;
}
```

```
(gdb) break main
(gdb) run
(gdb) next
```

```
(gdb) print a
$1 = 1, 2, 3

(gdb) ptype a
type = int [3]

(gdb) x/12xb &a
0x7fff5fbff56c:  0x01 0x00
0x00 0x00 0x02 0x00 0x00 0x00
0x7fff5fbff574:  0x03 0x00
0x00 0x00
```

GDB - pole a ukazatele

```
(gdb) print a[0]
$4 = 1

(gdb) print *(a + 0)
$5 = 1

(gdb) print a[1]
$6 = 2

(gdb) print *(a + 1)
$7 = 2

(gdb) print a[2]
$8 = 3

(gdb) print *(a + 2)
$9 = 3
```

```
(gdb) ptype &a
type = int (*)[3]

(gdb) print a + 1
$10 = (int *) 0x7fff5fbff570

(gdb) print &a + 1
$11 = (int (*)[3]) 0x7fff5fbff578

(gdb) print &a[0]
$11 = (int *) 0x7fff5fbff56c
```

GDB – složitější případ

```
1  #include <stdio.h>
3  int factorial (int n
       )
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n *
       factorial(n-1);
9  }
11 int main (void)
12 {
13   printf("%d",
       factorial(4));
14   return 0;
15 }
```

```
(gdb) list
```

Výpis části zdrojového kódu

```
(gdb) break factorial
```

Nastavení breakpointu

```
(gdb) run
(gdb) print(n)
$1 = 8
```

Spuštění programu a výpis parametru funkce

```
(gdb) continue
Continuing.
(gdb) print(n)
$2 = 7
```

Pokračování v běhu a opětovný výpis parametru

```
(gdb) clear main
Deleted breakpoint 1
(gdb) run
```

Odstranění breakpointu

GDB – složitější případ

```
1  #include <stdio.h>
3  int factorial (int n
       )
4  {
5    if (n == 1)
6      return 1;
7    else
8      return n *
       factorial(n-1);
9  }
11 int main (void)
12 {
13   printf("%d",
       factorial(4));
14   return 0;
15 }
```

```
(gdb) break factorial
```

breakpoint na vstupní bod funkce nazvané f

```
(gdb) info breakpoints
```

získáme informaci o všech breakpointech

```
(gdb) ignore 1 5
(gdb) r
```

breakpoint ignoruje prvních pět průchodů

```
(gdb) bt
```

historie volání – backtrace, zkratka bt

```
(gdb) bt 4
```

zajímá nás jen část

III. Ladění

GDB

Valgrind

Valgrind

- Dynamická analýza kódu
- Detekce
  - podmíněných skoků závislých na neinicializované proměnné
  - neoprávněné čtení / zápis do paměti
  - neuvolňování paměti (memory leak)

Část IV

Zadání 6. domácího úkolu

Zadání 6. domácího úkolu (HW06)

Téma: Maticové počty

- **Motivace:** Práce s dvourozměrným polem
- **Cíl:** Práce s polem variabilní délky a předávání ukazatelů
- **Zadání:** `https://cw.fel.cvut.cz/wiki/courses/b0b99prpa/hw/hw06`
  - Načtení vstupních hodnot dvou matic a znaku operace ('*' – násobení).
  - Volitelné zadání rozširuje úlohu o další operace s maticemi sčítání ('+') a odčítání ('-'), které mohou být zapsány ve výrazu.
- **Termín odevzdání:** **16.11.2019, 23:59:59**

Shrnutí přednášky

Diskutovaná témata

- Rozdělení paměti
- Rozsah platnosti proměnných
- Alokace v dynamické paměti
- Ladící prostředky

- Příště: vícerozměrná pole v dynamické paměti, reprezentace čísel v počítači

Diskutovaná témata

- Rozdělení paměti
- Rozsah platnosti proměnných
- Alokace v dynamické paměti
- Ladící prostředky

- Příště: vícerozměrná pole v dynamické paměti, reprezentace čísel v počítači