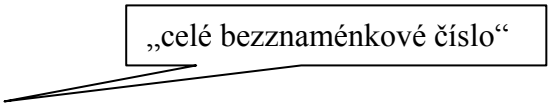
Správa paměti

Motivace a úvod

- v C (skoro vždy) ručně
- statické proměnné
  - datový typ, počet → znám v době překladu
  - zabírají paměť po celou dobu běhu programu
- problém velikosti definovaných proměnných – „jak velké pole?“
- Dynamické proměnné
  - datový typ, počet → znám v době překladu
  - velikost → při běhu programu
- dynamické proměnné v C → výhradně přes pointery
  - **každá dyn. proměnná musí mít svůj pointer**
- stdlib.h → funkce pro práci s dynamickými proměnnými

Alokace a dealokace paměti

- alokace → OS programu vyhradí část volné paměti
- memory allocation




```
void * malloc(size_t Size)
```

  - `Size` = počet požadovaných **Bytů**
  - Vrací pointer na alokovaný blok (pointer na `void` → nutno vždy přetypovat)
    - alokace se zdařila → platný pointer
    - alokace se nezdařila (není dost paměti) → `NULL`
- Dealokace (uvolnění) paměti → program vrací paměť zpět OS

```
void free(void *Memory)
```

  - `Memory` je pointer na uvolňovaný blok
- Realokace – změna velikosti alokovaného bloku

```
void *realloc(void *ptr, size_t size)
```

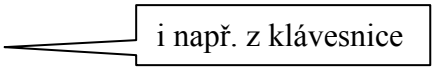
  - změní velikost alokované paměti v bloku `ptr` na `size` (obsah zachován)
  - vrací
    - realokace OK: pointer na nový blok
    - realokace false: `NULL`
  - nový blok > původní → přidá paměť za původní (obsah náhodný)
  - nový blok < původní → odřízne konec

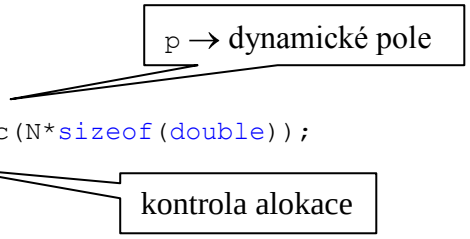
Příklad

```
#include <stdio.h>
#include <stdlib.h>

#define N   10

int main(void)
{
      double *p;
      int i;
      // alokace
      p = (double *)malloc(N*sizeof(double));
      if (p == NULL)
            return 1;
      // prace s polem
      for (i = 0; i < N; i++)
            p[i] = i;
      for (i = 0; i < N; i++)
            printf("%lf\n", *(p + i));
      // dealokace
      free((void *) p);
      p = NULL;
      return 0;
}
```





- Poznámky:
  - pro jednoduché typy se dyn. alokace nevyplatí – režie
  - alokace mnoha malých bloků = vyčerpání paměti dříve, než při alokaci jednoho velkého bloku
  - práce s dynamickým polem = práce se statickým pole (až na definici)

Vícerozměrná dynamická pole

- dimenze neomezeně, dále 2D
- „polodynamické pole“ = statické pole pointerů na dynamické řádky
  - počet řádků staticky, počet sloupců dynamicky

```
int *pPoloDynPole[RADKU];
for(i = 0; i < RADKU; i++)
      pPoloDynPole[i] = (int *)malloc(SLOUPCU*sizeof(int));
```

  - každý řádek může mít i jinou velikost
- plně dynamické pole = dynamické pole pointerů na dynamické řádky

```
int **pDynPole;
pDynPole = (int **)malloc(RADKU*sizeof(int *));
for(i=0; i<RADKU; i++)
      pDynPole[i] = (int *)malloc(SLOUPCU*sizeof(int));
```

Program v paměti počítače

- Paměť → několik segmentů (závisí na architektuře počítače a překladači)

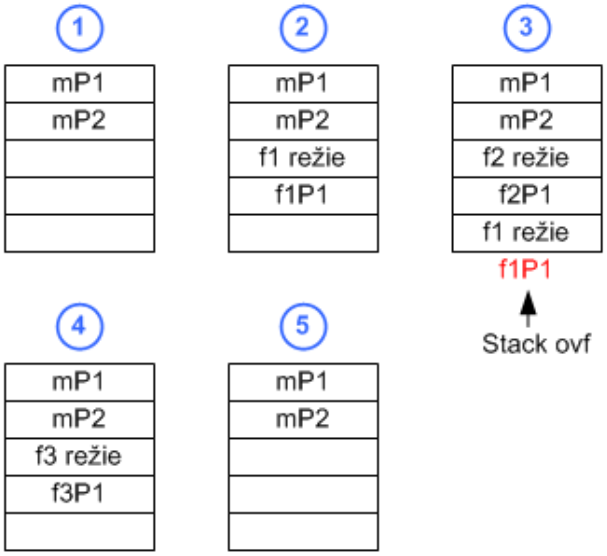
  - code – kód vlastního programu (instrukce)
  - globální data – všechny globální proměnné
  - zásobník (stack) –lokální proměnné + režie funkcí
  - hromada (halda, heap) – volná paměť pro dynamické proměnné
  - nějaké další
- MS-DOS
  - v paměti vždy pouze OS, ovladače a (1) program
  - zásobník → max. 64 kB (typicky 4 kB)
  - heap → 64 kB (cca 1x `double Pole[90][90]` !!!)
  - globální proměnné → max. 64 kB
  - (MS-DOS + code + stack + heap + global + ovladače) < 640 kB
- Win32 (W95 a novější)
  - každý program – k dispozici 4 GB (virtuální paměť)
    - virtuální paměť = RAM + pevný disk (swap file)
  - zásobník = 1 MB (lze zvětšit)
  - (code + stack + globální data + heap) < 2GB

Program a zásobník

- program s více funkcemi – obr. obsazení zásobníku

```
void f1(void);
void f2(void);
void f3(void);

int main(void)
{
      int mP1,mP2;      // 1
      f1();             // 2
      f2();             // 3
      f3();             // 4
      return 0;         // 5
}
void f1(void)
{
      int f1P1;
}
void f2(void)
{
      int f2P1;
      f1();
}
void f3(void)
{
      int f3P1;
}
```

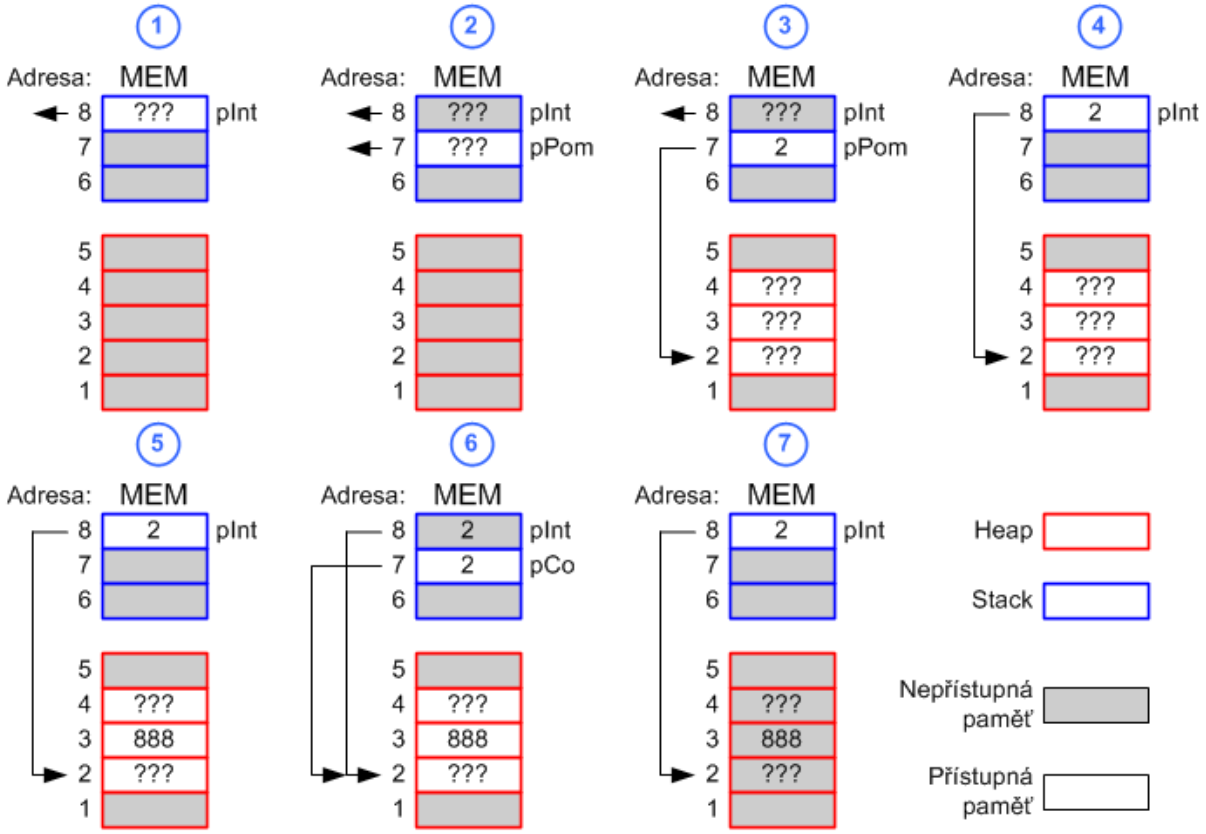


Dynamické proměnné a funkce

- Příklad – heap vs. stack

```
int main(void)
{
        int *pInt;              // 1
        pInt = Alokuj(3);
        // po Alokuj()             4
        pInt[1] = 888;          // 5
        Tisk(pInt);
        free(pInt);             // 7
        return 0;
}
int *Alokuj(N)
{
        int *pPom;              // 2
        pPom = malloc(N);       // 3
        return pPom;
}
void Tisk(int* pco)
{
        // printf(pco)          // 6
}
```

  - Obr. vždy po provedení příslušného příkazu na řádku

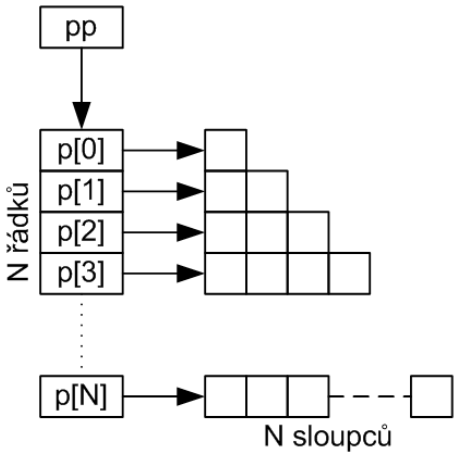


- statické vs. dynamické proměnné alokované a vracené funkcí
  - statické → na stacku → nelze používat mimo funkci
  - dynamické → na heapu → lze používat mimo funkci

```
int *AlokujPole1(void)  // spatne
{
      int poleStat[100];
      return poleStat;  // compiler warning
}
int *AlokujPole2(void)  // OK
{
      int *pPom;
      Pom = (int *)malloc(100*sizeof(int));
      return pPom;
}
```

Příklad

Napište funkci, která alokuje paměť pro kompletně dynamické dvourozměrné „zubaté“ pole ve tvaru dolní trojúhelníkové matice. Napište také funkci, která alokovanou paměť uvolní




```c
#include <stdio.h>
#include <stdlib.h>

double **AlokujMem(int N);
void UvolniMem(double **pp, int N);

int main(void)
{
	double **TrojMatice;

	TrojMatice = AlokujMem(10);

	*(TrojMatice[3]+1) = -123.456;
	printf("%lf\n", *(*(TrojMatice+3)+1));
	return 0;
}

double **AlokujMem(int N)
{
	double **ppPom;
	int i;

	ppPom = (double **)malloc(N*sizeof(double *));
	if (ppPom == NULL) {
		return NULL;
	}
	for (i=0; i<N; i++) {
		ppPom[i] = (double *)malloc((i+1)*sizeof(double));
		if (ppPom[i] == NULL) {
			return NULL;
		}
	}
	return ppPom;
}

void UvolniMem(double **pp, int N)
{
	int i;
	for(i=0; i<N; i++) {
		free((void *)pp[i]);
	}
	free((void **)pp);
}
```

`pole[i][j]` nelze –
řádky nemusí ležet za sebou!!!

Počítání referencí

- vylepšené funkce pro správu paměti

```
#include <stdlib.h>

unsigned int numberOfReferences = 0;
void *getmem(size_t Size);
void freemem(void **ppBlock);

int main(void)
{
      return 0;
}

void *getmem(size_t Size)
{
      void *p;
      p = malloc(Size);
      if (p != NULL)
            numberOfReferences++;
      return p;
}

void freemem(void **ppBlock)
{
      if (*ppBlock != NULL)
            numberOfReferences--;
      free(*ppBlock);
      *ppBlock = NULL;
}
```

globální proměnná;
= počet alokací;
na konci programu musí být 0

** → funkce bude pointer měnit

změna pointeru