

Konstanty I

- možnosti:

přednostně

v paměti neexistuje

žádný ;

- o preprocesor (deklarace)

```
#define KONSTANTA 10
```

- o konstantní proměnná (definice)

```
const int KONSTANTA = 10;
```

v paměti existuje

- příklad

```
#include <stdio.h>

#define PI 3.141592
#define N 10
#define POZDRAV "ahoj"

int main(int argc, char* argv[])
{
    double R, O;
    int i;
    O = 2 * PI * R;           // O = 2 * 3.141592 * R;
    for(i = 0; i < N; i++)    // for(i = 0; i < 10; i++)
        // neco
    printf(POZDRAV);         // printf("ahoj");
}
```

Datový typ pointer

- obsah (hodnota) = adresa v paměti
- využití – přístup k jiným proměnným
 - pole, řetězce
 - funkce (pointer na funkci, volání odkazem)
 - dynamické proměnné

Deklarace pointeru

- vždy svázán s datovým typem
- syntaxe: `datovy_typ *identifikator`

```
double *pA, *pB;
unsigned char *pCislo1;
```

- Pozor:

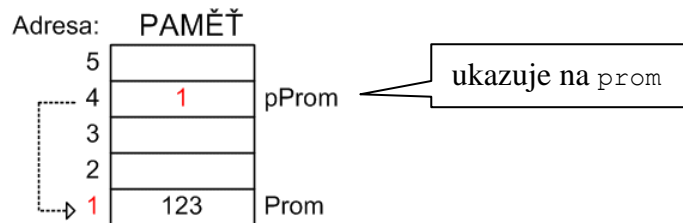
```
double *pA, pB;           // pointer na double, promenna double!!!
```

Referenční operátor &

- vrací adresu proměnné

```
int prom = 123, *pProm;
pProm = &prom;
```

žádná *



- inicializace pointeru v definici

```
int prom = 10, *pProm = &prom;
```

- & lze používat v programu libovolně

```
double prom1, cislo;
double *pProm = &prom1; // pProm ukazuje na Prom1
Cislo = 156*Prom1;      // nejaka operace
pProm = &Cislo;         // pProm nyní ukazuje na cislo
```

Dereferenční operátor *

- vrací hodnotu paměťového místa, na které ukazuje pointer:

```
int prom = 123, *p = &prom, cislo;
cislo = *p + 10;           // cislo = 123 + 10
*p = 58;                  // prom = 58
printf("prom = %d", *p);  // tisk prom
```

- významy *: násobení, dereference, definice proměnné typu pointer

Poznámky:

- vysvětlení scanf()

- hlavička

```
scanf("format", *promenna)
```

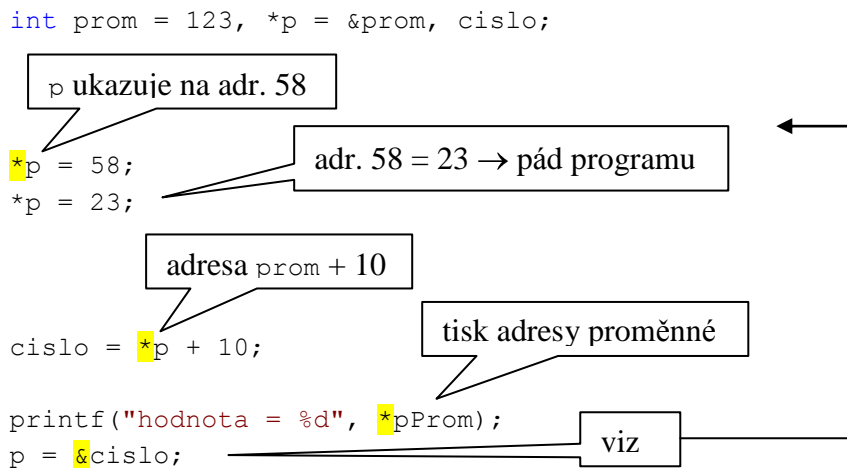
argument typu adresa

- volání

```
int prom;
scanf("%d", &prom)
```

adresa prom

- Důsledky chyb (vynecháno):



- nemíchat proměnné, pointery různých datových typů (typová konverze)

```
int *pProm;
double Cislo;
pProm = &Cislo; // Chyba (warning prekladace)
```

Práce s datovým typem pointer na ...

- příklady formálního zápisu

	„normální“ dat. typ	pointer
definice proměnné	<code>int prom</code>	<code>int *pointer</code>
přetypování	<code>(char)prom</code>	<code>(char *)pointer</code>
sizeof	<code>sizeof(int)</code>	<code>sizeof(int *)</code>
návratová hod. funkce	<code>int Funkce(...</code>	<code>int * Funkce(...</code>

- velikost pointeru vs. datového typu

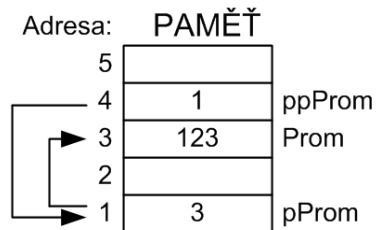
	Win32, Linux (32b CPU, AS)	AVR (8b CPU, 16b AS)
<code>sizeof(char)</code>	1	1
<code>sizeof(int)</code>	4	2
<code>sizeof(double)</code>	8	8
<code>sizeof(char *)</code>	4	2
<code>sizeof(int *)</code>	4	2
<code>sizeof(double *)</code>	4	2

Nulový pointer

- konstanta NULL (obvykle `#define NULL 0`)
- NULL pointer = „neukazuje nikam“
- lze přiřadit všem typům pointerů

Pointer na pointer

- princip



- použití

```
int prom;
int *pProm;
int **ppProm;
```

```
pProm = &prom;           // ukazuje na prom
ppProm = &pProm;        // take ukazuje na prom
```

```
scanf("%d", *ppProm);   // nacte do prom
printf("%d", **ppProm); // vytiskne prom
pProm = NULL;           // neukazují nikam
ppProm = NULL;
```

- dereferencování pointeru:

- ppProm → pointer na pointer na int
- *ppProm → pointer na int
- **ppProm → proměnná typu int

Pointerová aritmetika

- dále předpoklad: 32 b platforma (`sizeof(int) → 4`, `sizeof(int *) → 4`), adresace po B
- definované matematické operace s pointerem:
 - součet pointeru a celého čísla: + (bin), ++
 - rozdíl pointeru a celého čísla: - (bin), --
 - rozdíl dvou pointerů (součet nelze!!!)
 - porovnání pointerů: > >= == != < <=
- pouze teoretické úvahy, v praxi používat jen u polí (viz dále)
- přičítání a odečítání celého čísla k pointeru

```
#define K hodnota
TYP *pointer;
```

posun o K paměťových míst nahoru

- platí

```
pointer + K ↔ (int)pointer + K*sizeof(TYP)
pointer - K ↔ (int)pointer - K*sizeof(TYP)
```

- příklady:

```

int *pInt = 100;          // POZOR, jen teoreticky
double *pDouble = 200; // POZOR, jen teoreticky

// dále předpoklad:
// sizeof(int) = 4
// sizeof(double) = 8

pInt++;                  // pInt bude 104
pDouble = pDouble - 10; // pDouble bude 120 = 200-10*8

```

- velmi časté použití
- rozdíl dvou pointerů
 - výsledek → celé číslo (“o kolik prvků se pointery liší”)

Datový typ pole

1D pole

Definice

```
datovy_typ identifikator[pocet_prvku];
```

- Příklady:

```
double teplota[7];
unsigned int mojePole[10000];
```

musí být známo v době překladač

- statické pole – počet prvků nelze během práce programu měnit

Inicializace

```
double teplota[7] = {1, 2, -5.35, 0.1245, 4, 43, -11};
```

- neinicializované pole → prvky nedefinovaná hodnota
- pokud:
 - počet inicializací > počet prvků → Error překladače
 - počet inicializací < počet prvků → překladač zbývající prvky inicializuje na 0 (0.0)

Práce

- s prvky – prostřednictvím indexů

```
int teplota[7];
printf("%d", teplota[3]);
teplota[2*k+1] = 10;
```

výraz / paměťové místo typu int

- indexy 0, 1, ..., pocet_prvku - 1
- meze nekontrolovány (hrozí přetečení nebo podtečení indexů) = možný pád programu
- pole v sobě nenese informaci o své délce
- příklad

```

#define PO CET_PRVKU 7
double teplota[PO CET_PRVKU];
int i;
for(i = 0; i < PO CET_PRVKU; i++) {
    printf("Zadej T pro %d.den: ", i+1);
    scanf("%lf", &teplota[i]);
}
for(i=0; i < PO CET_PRVKU; i++) {
    printf("teplota[%d] = %lf\n", i+1, teplota[i]);
}

```

Vícerozměrná pole

- = jednorozměrné pole, jehož prvky jsou pole...

Definice

```
datovy_typ id[pocetPrvku1][pocetPrvku2][pocetPrvku3]...;
```

- např. `int pole[4][5];`

Inicializace

```
int Pole[2][3] = {{11,12,13}, {21,22,23}};
```

Práce

- Příklad: Součet dvou matic

```

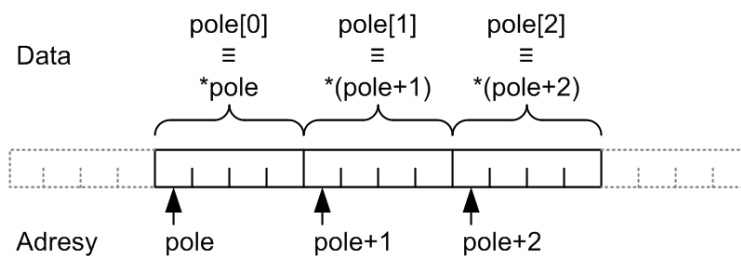
#define M 3
#define N 2
int A[M][N] = {{11,12}, {21,22}, {31,32}};
int B[M][N] = {{11,12}, {21,22}, {31,32}};
int i, j, C[M][N];
for(i = 0; i < M; i++)
{
    for(j = 0; j < N; j++)
    {
        C[i][j] = A[i][j] + B[i][j];
    }
}

```

Uložení pole v paměti a pointerová aritmetika

1D pole

```
int pole[3];
```



Locals			
Name	Value	Type	
pole	0x0012ff58	int [3]	skutečná adresa v paměti
[0]	-858993460	int	
[1]	-858993460	int	
[2]	-858993460	int	

- Pro adresy platí:

```
pole == pole+0 == &pole[0]
pole+i == &pole[i]
```

- přístup k prvkům pole

- přes indexy: `pole[i]`
- pointerly: `*(pole + i)`

rozumné využití pointerové aritmetiky

- pointer typu pole je konstantní → nelze jej změnit (fyzicky neexistuje)
- Q1: Je následující kód OK?

```
int *pInt;
int pole[3];
pInt = pole;
pInt[2] = 999;
```

výraz typu `int *`

- Q2: Je následující kód OK?

```
int *pInt;
int pole[3];
pole = pInt;
```

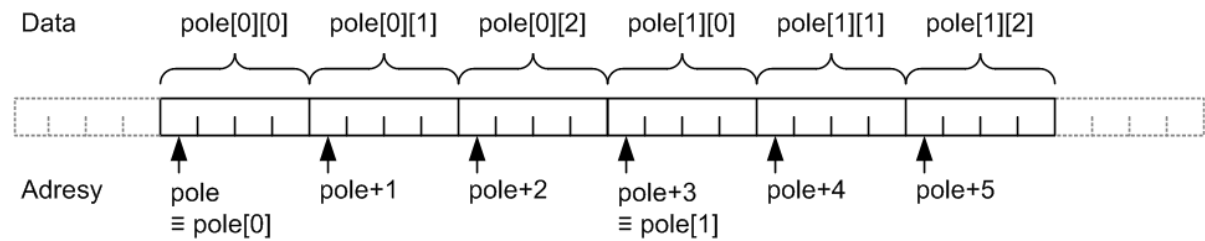
- Příklad: vynulování pole přes pointerly

```
#define N 3
int pole[N] = {10,10,10};
int *p = pole;
while (p < (pole + N))
    *p++ = 0;
```

Vícerozměrná pole

- v paměti po řádcích za sebou:

```
int pole[2][3];
```



Locals		
Name	Value	Type
Pole	0x0012ff4c	int [2][3]
[0]	0x0012ff4c	int [3]
[0]	11	int
[1]	12	int
[2]	13	int
[1]	0x0012ff58	int [3]
[0]	21	int
[1]	22	int
[2]	23	int

pointer na řádek

- příklady přístupu do pole:

```
int pole[2][3] = {0};
int *pPole;
```

```
pole[0][1] = 10;           // pres indexy
*(*(pole + 1) + 2) = 15;  // pole[1][2]
*(pole[1]+1) = 64;       // pole[1][1]
// dukaz radkoveho usporadani
pPole = pole;            // compiler warning
*(pPole+3) = -88;       // pole[1][0] - nedělat
```