JAZYK C
VÝPISKY

00_Obsah

01_základy

komentáře

```
// komentář na jeden řádek

/* komentář na
více řádků */
```

identifikátory

identifikátor může být podle normy ANSI dlouhý 31 znaků, může obsahovat pouze písmena anglické abecedy, číslice a podtržítko a nesmí začínat číslicí, jazyk C je case sensitive

klíčová slova jazyka C

asm, auto, break, case, cdecl, char, const, continue, default, do, double, else, enum, extern, far, float, for, goto, huge, if, int, interrupt, long, near, pascal, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while

první program

```
#include <stdio.h>                  // vložení hlavičkového souboru

int main()                          // hlavní funkce
{
      printf("Nazdar chlape!\n");   // výstup na obrazovku
      return 0;                     // návratová hodnota funkce
}
```

kompilace v gcc

gcc -std=c99 -Wall -pedantic -W -g -o vystupni_soubor vstupni_soubor

parametry: -std=c99: překlad dle normy ISO C99, -Wall: varovné zprávy, -pedantic: donutí striktně dodržovat danou normu, -W: částečná sémantická kontrola, -g: ladící informace, -o soubor: název výstupního souboru

02_datové typy a proměnné

prázdný datový typ void

datový typ void označuje „žádnou hodnotu“, používá se u funkcí, které nevracejí žádnou hodnotu nebo u ukazatelů

celočíselné datové typy

název typu	velikost	rozsah
int signed signed int	4 Byte	-2 147 483 648 až 2 147 483 647
unsigned int unsigned	4 Byte	0 až 4 294 967 295
signed short int short int short signed	2 Byte	-32 768 až 32 767
unsigned short int short unsigned	2 Byte	0 až 65 535
char signed char	1 Byte	-128 až 127
unsigned char	1 Byte	0 až 255

reálné datové typy

název typu	velikost	rozsah	počet des. míst
float	4 Byte	3.4E-38 až 3.4E38	7
double	8 Byte	1.7E-308 až 1.7E308	15
long double	10 Byte	3.4E-493 až 1.1E493	18

operátor sizeof

vrací počet Byte, který zabírá daný datový typ v paměti, na různých architekturách může být odlišný

```
velikost = sizeof(int);
velikost = sizeof(prom);
velikost = sizeof(*pointer);
```

operátor typedef

operátor typedef vytváří nový datový typ

```
typedef char * P_ZNAK;        // definice nového datového typu ukazatel na znak
P_ZNAK p1, p2, p3;            // stejné jako char *p1, *p2, *p3;
```

```
typedef int POLE[10];         // definice nového datového typu pole o 10 prvcích
POLE moje_pole;
```

definice a přiřazování hodnot proměnným

deklarace je přidělení identifikátoru proměnné, definice je přidělení identifikátoru a paměti proměnné

```
int prom;               // definice proměnné
char x, y;              // definice více proměnných
float z = 1.5;          // definice a inicializace proměnné
x = 'a'                 // přiřazení hodnoty proměnné
prom = 5;               // přiřazění hodnoty proměnné
prom = prom + 5;        // zvýší hodnotu prom o pět
prom = faktorial(5)     // přiřazení hodnoty výsledku funkce
```

lokální a globální proměnné

lokální proměnné jsou definovány a jsou platné uvnitř funkce / bloku, globální proměnné jsou definovány vně funkce a existujou po celou dobu běhu programu, mohou je používat všechny funkce, u dvou proměnných se stejným identifikátorem má vyšší prioritu více vnořená proměnná, která zastíní tu druhou

celočíselné konstanty

dekadické: 0, 1, 2, 10, 20
oktalové: 0, 01, 02, 012, 024
hexadecimální: 0x0, 0x1, 0x2, 0xA, 0x14

záporná konstanta: znaménko – na začátku
long konstanta: příznak L na konci
unsigned konstanta: příznak U na konci

reálné konstanty

přímý tvar: 1.25, .0, -1.
semilogaritmický tvar: 24.68e6, 12e-9, .009e30

float konstanta: příznak F na konci
long konstanta: příznak L na konci

znakové konstanty

znakové konstanty jsou typu int, zápis znaku: 'J', '\112', 'x4a'

escape znaky:

escape sekvence	název	hexadec. hodnota	označení	význam
\a	alert	0x07	BELL	zvukový signál
\b	backspace	0x08	BS	zpětná mezera
\f	form feed	0x0C	FF	nová stránka
\n	line feed	0x0A	LF	nový řádek
\r	carriage return	0x0D	CR	návrat vozíku
\t	horizontal tab	0x09	HT	tabulátor
\v	vertical tab	0x0B	VT	vertikální tabulátor
\\	backslash	0x5C	\	zpětné lomítko
\'	single quote	0x2C	'	apostrof
\"	double quote	0x22	"	uvozovky
\0	null character	0x00	NUL	nulový znak

řetězcové konstanty

"Nazdar chlape, jak se máš?"

"Nazdar chlape,\
jak se máš?"

"Nazdar chlape,"
"jak se máš?"

"Nazdar" "chlape, jak se máš?"

implicitní přetypování

datový typ char a short int se konvertují na typ int, je-li jeden z operandů typu long double / double / float / unsigned long / long / unsigned int, druhý operand se konvertuje také na tento typ a výsledkem bude rovněž tento typ, jinak musí být oba operandy typu int

explicitní přetypování

```
(int) 'A'               // převod znaku na ordinální číslo
(double) 7              // převod celého čísla  na reálné
(int) -2.56456          // oříznutí desetinné části
```

```
int main()
{
      int a = 10, b = 3;                 // celočíselné operandy
      int vysl_c;                        // celočíselný výsledek
      float vysl_r;                      // reálný výsledek

      vysl_c = a / b;                    // vysl_c == 3
      vysl_r = a / b;                    // vysl_r == 3

      vysl_r = (float) a / b;            // vysl_r = 3.3333
      vysl_c = (float) a / b;            // vysl_c = 3
      vysl_r = (float) a / (float) b;    // správný zápis
}
```

přetypovaná proměnná nemůže být L-hodnotou

paměťové třídy

auto: implicitní třída lokálních proměnných, není automaticky inicializována, platí jen uvnitř dané funkce

extern: proměnná je definována v jiném zdrojovém souboru, používá se při odděleném překladu

static: proměnná existuje i po dokončení funkce - je pouze nedostupná a ponechává si svoji hodnotu, při dalším volání ji lze znovu použít

register: proměnná bude uložena v registru procesoru

```
#include <stdio.h>

void fce()
{
      static int i = 0;
      printf("Volani c.%d\n", ++i);
      return;
}

int main()
{
      int a;
      for(a=0; a<5; a++) fce();
      return 0;
}
```

typové modifikátory

const: konstanta, tuto proměnnou nelze během její platnosti měnit

volatile: proměnná může být kdykoliv asynchronně změněna z vyšší moci (přerušením apod.), není optimalizována, ani ukládána do vyrovnávací paměti

03_operátory a výrazy

rozdělení operátorů

dle funkce: aritmetické, operátory přiřazení, logické, bitové, relační
dle počtu operandů: unární, binární, ternární

aritmetické operátory: + - * / %
operátory inkrementace a dekrementace: postfixové: a++, a--, prefixové: ++a, --a

```
cislo = 5;
x = cislo++;
// cislo = 6, x = 5

cislo = 5;
y = ++cislo;
// cislo = 6, y = 6
```

přiřazovací operátory: = += -= *= /=
logické operátory: == != && || ! < <= > >=

```
k = (i && j) || (i = 5); // pokud je (i && j) == 1, (i = 5) už se neprovede!
```

ternární operátor: ? :, např. abs_h = (a >= 0) ? a : -a
operátor čárky: ,

```
k = (i--, i + j); // je stejné jako:
i--; k = i + j;
```

výraz: x = (y + 10) / 2
příkaz: x = (y + 10) / 2;

priority operátorů

priorita	operátory	směr vyhodnocení
1	() [] -> .	->
2	! ~ ++ -- - * & sizeof *přetypování*	<-
3	* / %	->
4	+ -	->
5	<< >>	->
6	< <= > >=	->
7	== !=	->
8	&	->
9	^	->
10	|	->
11	&&	->
12	||	->
13	? :	<-
14	= += -= *= /=	<-
15	,	->

bitové pole a bitové operace

```
typedef struct                  // definice bitového pole
{
      unsigned sec : 5;         // uloženo v 0 - 4 bitech
      unsigned min : 6;         // uloženo v 5 - 10 bitech
      unsigned hod : 5;         // uloženo v 11 - 15 bitech
} CAS;

CAS time;                       // proměnná time typu CAS
time.sec = 20;                  // práce s položkou bitového pole
time.min = 13;
time.hod = 7;
```

operátor	bitová operace
&	součin
|	součet
^	exklusivní součet
<<	posun doleva
>>	posun doprava
~	negace

04_řízení běhu programu

podmínka if-else

```
if (x == 0)       // když x je rovno 0
{                 // začátek bloku
      x = 1;      // příkaz
}                 // konec bloku
else if (x == 1)  // jinak když x je rovno 1
{
      x = 0;
}
else              // jinak
{
      x = -1;
}
```

```
if (x)            // když x není rovno 0
      x = 0;
```

```
if (!x) x = 1;    // když je x rovno 0
```

for (cyklus se známým počtem průchodů)

```
for (int i = 0; i < 10; i++)  // provede 10 cyklů
      printf("%d \n", i);
```

```
for (;;)                      // nekonečný cyklus
{
      x++;
      printf("%d \n", x);
}
```

while (podmíněný cyklus s neznámým počtem průchodů)

```
while ((znak = getchar()) < '0' || znak > '9') // cyklus
      i++;
```

do-while (cyklus s podmínkou na konci s neznámým počtem průchodů)

```
do                      // provede příkazy
{
      i++;              // příkaz
}
while (i < 10);         // dokud i je menší než 10
```

příkazy break a continue

příkaz break ukončí daný cyklus, příkaz continue přeruší běh cyklu a pokračuje od začátku cyklu

přepínač switch

```
switch (prom)           // přepínač
{
      case 1:           // pokud je prom rovna 1
            x = 1;      // příkaz
            break;      // ukončí switch (jinak by se vykonávaly další příkazy)
      case 2:           // pokud je prom rovna 2
            x = 2;
            break;
      case 3:           // pokud je prom rovna 3
            x = 3;
            break;
      default:          // jiné hodnoty
            x = 0;
            break;
}
```

goto (nepodmíněný skok)

```
goto navesti;
// kód
navesti:
```

05_funkce

funkce main

```
int main(int argc, char *argv[])    // hlavní funkce main s parametry
{
      // příkazy
      return 0;                     // program proběhl úspěšně (vrací 0)
}
```

vlastní funkce

```
int funkce(int prom1, int prom2)    // definice funkce
{
      // příkazy
      return hodnota;         // návratová hodnota funkce
}

vysledek = funkce(x, y);      // volání funkce a přiřazení hodnoty do proměnné
```

```
void funkce(float promenna)         // definice funkce typu void
{
      // příkazy
      return;
}

funkce(promenna);                   // volání funkce
funkce(dalsi_funkce(promenna));     // funkce jako parametr jiné funkce
```

rekurze

```
#include <stdio.h>

int faktorial(int i)
{
      if (i == 1) return 1;
      else return i * faktorial(i-1);     // funkce volá sama sebe
}

int main()
{
      int i = faktorial(5);
      printf("%d\n", i);
      return 0;
}
```

06_preprocesor

standardní knihovna jazyka C

assert.h, ctype.h, errno.h, float.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h, time.h

direktivy preprocesoru

#define: definice symbolických konstant, dle programátorské konvence se konstanty píší velkými písmeny

```
#define MIN 25
#define MAX (2 * MIN)
#define POZDRAV "Nazdar chlape!\n"
```

#include: vložení zdrojového souboru

```
#include "soubor.h"     // uživatelský hlavičkový soubor
#include <stdio.h>      // standardní hlavičkový soubor
```

direktivy podmíněného překladu

#if, #elif, #else, #endif: podmínka

```
#define B 1
#if (A)
printf("Prvni sekce\n");
#elif (B)                    // přeloží se pouze tato sekce
printf("Druha sekce\n");
#else
printf("Posledni sekce\n");
#endif
```

#error: přeruší kompilaci a generuje chybu

#ifdef, #ifndef: testuje, zda daná konstanta byla/nebyla definována

```
#define B
#ifdef A
printf("Prvni sekce\n");
#else
      #error Konstanta A není definována!
#endif
```

#undef: zruší definici makra

```
#define PLAT 10000
...
#undef PLAT
```

07_oddělený překlad

praktický příklad odděleného překladu

```
// hlavni.c

#include "std_def.h"          // uživatelské definice
#include "knihovna1.h"        // hlavičky z knihovny 1
#include "knihovna2.h"        // hlavičky z knihovny 2

int main()
{
      INTEGER n;
      n = nacti();            // volání funkce z knihovny 1
      vypis(n);               // volání funkce z knihovny 2
      scanf("%d", &n);
      return 0;
}
```

```
// stddef.h

#ifndef N               // podmíněný překlad (zabrání vícenásobnému vtažení)
      #define N
      #define INTEGER int
#endif
```

```
// knihovna1.h

INTEGER nacti();              // načte celé číslo z klávesnice
```

```
// knihovna1.c

#include <stdio.h>            // standardní hlavičkový soubor
#include "std_def.h"          // uživatelské definice
#include "knihovna1.h"        // hlavičky z knihovny 1
#include "knihovna2.h"        // hlavičky z knihovny 2

INTEGER nacti()               // načte a vrátí celé číslo z klávesnice
{
      INTEGER i;
      scanf("%d", &i);
      printf("Provedeme kontrolni vypis: ");
      vypis(i);
      return i;
}
```

```
// knihovna2.h

void vypis(INTEGER i);        // vypíše i na obrazovku
```

```
// knihovna2.c

#include <stdio.h>            // standardní hlavičkový soubor
#include "std_def.h"          // uživatelské definice
#include "knihovna2.h"        // hlavičky z knihovny 2

void vypis(INTEGER i)         // vypíše dvojnásobek i na obrazovku
{
      printf("%d \n", 2 * i);
      return;
}
```

08_ukazatele

referenční a dereferenční operátor

&: referenční operátor (vrací adresu proměnné)

*: dereferenční operátor (vrací obsah dané adresy, umožňuje zápis údaje na danou adresu), smí být použit jen na proměnné typu ukazatel

příklad:

adresa v paměti	1000	1002	1004	1006
její obsah	1004		7	
proměnná	pointer_i		i	

pointer_i - obsah ukazatele, adresa, na kterou ukazatel ukazuje (1004)
*pointer_i - hodnota, na níž ukazatel ukazuje (7)
&pointer_i - adresa ukazatele, adresa, na níž je v paměti uložena proměnná typu ukazatel na integer (1000)
i - hodnota proměnné (7)
*i - nemá význam! operátor * smí být použit jen na proměnné typu ukazatel
&i - adresa proměnné i (1004)

práce s ukazateli

```
int i = 7;          // definice a inicializace celočíselné proměnné
int *p_i;           // definice ukazatele na celé číslo
p_i = &i;           // přiřazení adresy proměnné i ukazateli p_i
*p_i = 8;           // změníme hodnotu na kterou ukazuje p_i
```

```
int i = 7;          // definice a inicializace celočíselné proměnné
int *p_i = &i;      // definice a inicializace ukazatele
```

znak * v definici ukazatele nevystupuje jako dereferenční operátor!

příklady práce s ukazateli a chyby

```
int i = 10;         // definice a inicializace celočíselné proměnné
int *p_i, *p_j;     // definice ukazatelů na celé číslo

p_i = &i;           // ukazatel p_i ukazuje na i
*p_i = 20;          // stejné jako i = 20;
*p_i = i;           // do hodnoty p_i se přiřadí obsah i
*p_j = 30;          // cesta do pekel! p_j ukazuje na náhodnou adresu v paměti
p_j = p_i;          // ukazatel p_j ukazuje tamtéž co p_i
i = *p_j;           // do proměnné i se uloží hodnota na níž ukazuje p_j
*p_i = *p_j;        // do hod. na níž ukaz. p_i se přiřadí hod. na níž ukazuje p_j
i = *p_i + 2;       // do proměnné i se uloží hodnota ukazatele p_i zvýšená o 2
*(p_i + 5) = 30;    // na 5. adresu za obsahem p_i uložíme 30 (využití u polí)
p_i = 20;           // podezřelé! do ukazatele uložíme absolutní adresu 20
i = p_i;            // podezřelé! do proměnné i se uloží obsah ukazatele (adresa)
i = &p_i;           // podezřelé! do proměnné i se uloží adresa ukazatele
*p_i = &i;          // podezřelé! do hod. na níž ukaz. p_i se uloží adresa i
p_i = &20;          // chyba při překladu! konstanta 20 nemá adresu
p_i = &(i + 5);     // chyba při překladu! výraz (i + 5) nemá adresu
```

nulový ukazatel

nulový ukazatel je konstanta NULL z knihovny <stdio.h>, která neukazuje nikam

```
int *p_j = NULL
if (p_j != NULL) *p_j = 20;
```

přetypování ukazatelů

```
#include <stdio.h>
int main()
{
      int i = 65, *p_i = &i;
      char znak = 'B', *p_znak = &znak;

      p_znak = p_i;                              // nevhodné
      p_znak = (char *) p_i;                     // přetypování p_i

      printf("*p_znak == %c\n", *p_znak);
      return 0;
}
```

obecný (generický) ukazatel

```
#include <stdio.h>
int main()
{
      void *p_ukaz;                        // definice generického ukazatele
      int a = 7;
      char znak = 'B';

      p_ukaz = &a;                         // p_ukaz ukazuje na a
      * (int *) p_ukaz = 7;
      printf("a == %d\n", a);

      p_ukaz = &znak;                      // p_ukaz ukazuje na znak
      * (char *) p_ukaz = 'B';
      printf("znak == %c\n", znak);

      return 0;
}
```

generický ukazatel (void) může ukazovat na proměnnou jakéhokoliv datového typu

aritmetika ukazatelů

```
int i, *p_i, *p_j;
p_j = p_i + 2;
```

ukazatel bude ukazovat o n údajů dále, např. bude-li ukazatel ukazovat na prvek struktury o velikosti 20 bajtů, pak po přičtení 1 k ukazateli bude nový ukazatel ukazovat o 20 bajtů dále, aritmetika ukazatelů je stejná jako práce s proměnnými

ukazatele jako parametry funkcí

```
// záměna dvou proměnných

#include <stdio.h>

void zamen(int *p_i, int *p_j)
{
	int temp;
	temp = *p_i;
	*p_i = *p_j;
	*p_j = temp;
	return;
}

int main()
{
	int i = 7, j = 3;
	printf("i == %d, j == %d\n", i, j);
	zamen(&i, &j);
	printf("i == %d, j == %d\n", i, j);
	return 0;
}
```

```
// řazení čísel

#include <stdio.h>

void zamen(int *prvni, int *druhy, int *count)
{
	int temp;
	temp = *prvni;
	*prvni = *druhy;
	*druhy = temp;
	(*count)++;
	return;
}

int serad(int *a, int *b, int *c, int *d)
{
	int count = 0;
	while (!(*a <= *b && *b <= *c && *c <= *d))
	{
		if (!(*a <= *b)) zamen(a, b, &count);
		if (!(*b <= *c)) zamen(b, c, &count);
		if (!(*c <= *d)) zamen(c, d, &count);
	}
	return count;
}

int main()
{
	int a = 5, b = 8, c = 2, d = 6, count;

	printf("a=%d b=%d c=%d d=%d\n", a, b, c, d);
	count = serad(&a, &b, &c, &d);
	printf("a=%d b=%d c=%d d=%d count=%d\n", a, b, c, d, count);

	return 0;
}
```

ukazatel na funkci

```
int funkce(int arg1, int arg2);     // hlavička funkce

int (*p_funkce)();                  // definice ukaz. na funkci vracející int
p_funkce = funkce;                  // přiřazení adresy funkce do ukazatele

prom = funkce(x,y);                 // volání funkce
prom = (*p_funkce)(x,y);            // volání funkce pomocí ukazatele

int (*p_funkce[10])();              // definice pole ukazatelů na funkce
```

09_dynamické přidělování paměti

dynamická alokace

```
#include <stdio.h>                  // NULL, printf()
#include <stdlib.h>                 // malloc(), free()

int main()
{
      int *p_ukaz;                                          // definice ukazatele
      if((p_ukaz = (int *) malloc(sizeof(int))) == NULL)    // alokace paměti
      {
            fprintf(stderr, "Chyba pri alokaci pameti!\n");
            return -1;
      }

      *p_ukaz = 7;                  // práce s dynamicky alokovanou proměnnou
      printf("p_ukaz obsahuje adresu %p, kde je hodnota %d\n", p_ukaz, *p_ukaz);

      free(p_ukaz);                 // uvolnění paměti
      p_ukaz = NULL;                // nastavení ukazatele na NULL
      return 0;
}
```

funkce malloc() z knihovny <stdlib.h> alokuje paměť, parametrem je počet bajtů, které chceme alokovat (používá se operátor sizeof()), funkce vrací ukazatel typu void na začátek alokovaného prostoru v paměti, při každém volání bychom tedy měli přetypovat návratovou hodnotu na ukazatel na ten datový typ, jenž alokujeme, pokud se přidělení paměti nepovede, funkce vrací hodnotu NULL, při každé alokaci paměti je nutné testovat zda se alokace podařila

funkce free() z knihovny <stdlib.h> uvolňuje alokovanou paměť, paměť kterou jsme ručně alokovali musíme také ručně uvolnit, parametr funkce free() je typovaný jako ukazatel na void, proto bychom správně měli psát free((void *) p_ukaz);

ztráta odkazu na alokovanou paměť

```
void funkce()
{
      int *p_ukaz;
      p_ukaz = (int *) malloc(sizeof(int));
      *p_ukaz = 7;
      return;                 // po skončení funkce() ztratíme odkaz na p_ukaz
}

int main()
{
      funkce();
      return 0;
}
```

```
int *p_ukaz1, *p_ukaz2;
p_ukaz1 = (int *) malloc(sizeof(int));
p_ukaz2 = (int *) malloc(sizeof(int));

*p_ukaz1 = 7;
*p_ukaz2 = *p_ukaz1;

p_ukaz2 = p_ukaz1;            // ztráta odkazu na p_ukaz2
```

funkce calloc() a realloc()

funkce calloc() z knihovny <stdlib.h> alokuje paměť pro větší množství údajů, parametrem je počet prvků, pro které chceme alokovat paměť a počet bajtů, které chceme alokovat (používá se operátor sizeof()), funkce vyplní paměť nulami, funkce vrací ukazatel typu void na začátek alokovaného prostoru v paměti, při každém volání bychom tedy měli přetypovat návratovou hodnotu na ukazatel na ten datový typ, jenž alokujeme, pokud se přidělení paměti nepovede, funkce vrací hodnotu NULL, při každé alokaci paměti je nutné testovat zda se alokace podařila

```
int *p_ukaz;                                // definice ukazatele
p_ukaz = (int *) calloc(2, sizeof(int)); // alokace paměti pro p_ukaz
*p_ukaz = 7;
*(p_ukaz + 1) = 8;
```

funkce realloc() z knihovny <stdlib.h> naalokuje další místo pro již rezervovanou paměť, parametrem je ukazatel na blok paměti který chceme realokovat a nová velikost v bajtech (používá se operátor sizeof()), funkce vrací ukazatel typu void na začátek alokovaného prostoru v paměti, při každém volání bychom tedy měli přetypovat návratovou hodnotu na ukazatel na ten datový typ, jenž alokujeme, pokud se přidělení paměti nepovede, funkce vrací hodnotu NULL, při každé alokaci paměti je nutné testovat zda se alokace podařila

```
p_ukaz = realloc(p_ukaz, 3);
*(p_ukaz + 2) = 9;
```

10_pole a textové řetězce

statické pole

pole je datová struktura obsahující více prvků téhož datového typu, k prvkům pole (statického i dynamického) se přistupuje pomocí indexů nebo pomocí ukazatelů na pole (pomocí aritmetiky ukazatelů), v jazyce C se indexuje od 0, počet prvků statického pole musí být znám v okamžiku překladu a určuje se při definici pole

```
int pole[10];                 // definice pole 10 celých čísel
prom = pole[0];               // přiřazení hodnoty prvního prvku pole do prom
prom = pole[9];               // přiřazení hodnoty posledního prvku pole do prom
prom = pole[i];
prom = pole[5-3] * 2;
pole[0] = 7;                  // přiřazení hodnoty do prvního prvku pole
int pole[4] = {3,6,9,12};     // inicializace prvků pole
int pole[] = {3,6,9,12};      // inicializace prvků pole
int pole[4] = {3};            // inicializace 1 prvku pole
```

dynamické pole

```
int *pole;                                        // deklarace dynamického pole
pole = (int *) malloc(pocet * sizeof(int));       // definice dynamického pole

pole = (int *) realloc(pole, (pocet += 10) * sizeof(int)); // změna velikosti

prom = *pole;                                     // první prvek pole
prom = *(pole + 1);                               // druhý prvek pole
prom = *(pole + pocet - 1);                       // poslední prvek pole

free(pole);                                       // uvolnění paměti
pole = NULL;                                      // nastavení ukazatele na NULL
```

možné přístupy ke statickému a dynamickému poli

```
int pole_s[10];   // statické pole
int *pole_d;      // dynamické pole

pole_d            // adresa prvního prvku vytvořeného dynamického pole
                  // použití: pole_d = malloc(), free(pole_d)

pole_s            // adresa prvního prvku statického pole (konstantní ukazatel)
                  // použití: nepoužívá se

*pole_d           // hodnota prvního prvku dynamického pole
                  // použití: pro získání/nastavení prvku pole

pole_s[0]         // hodnota prvního prvku statického pole
                  // použití: pro získání/nastavení prvku pole

*(pole_d + 5)     // hodnota šestého prvku dynamického pole
                  // použití: pro získání/nastavení prvku pole

pole_s[5]         // hodnota šestého prvku statického pole
                  // použití: pro získání/nastavení prvku pole

pole_d[0]         // hodnota prvního prvku dynamického pole
                  // použití: práce s dynamickým polem pomocí indexů

*pole_s           // hodnota prvního prvku statického pole
                  // použití: práce se statickým polem pomocí ukazatelů

pole_d[5]         // hodnota šestého prvku dynamického pole
                  // použití: práce s dynamickým polem pomocí indexů

*(pole_s + 5)     // hodnota šestého prvku statického pole
                  // použití: práce se statickým polem pomocí ukazatelů

&pole_s[0]        // adresa prvního prvku statického pole
                  // použití: nepoužívá se (stejné jako pole_s)
```

pole jako parametr funkce

pole předáváme do funkce pomocí ukazatele na první prvek pole

```
#define POCET 10                                  // velikost pole

void funkce1(int pole[]);                         // hlavičky funkcí
void funkce2(int pole[], int velikost_pole);

int main()
{
      int pole1[10];                              // definice pole
      int pole2[POCET];

      funkce1(pole1);                             // volání funkcí
      funkce2(pole2, POCET);

      return 0;
}
```

statické vícerozměrné pole a matice

dvourozměrné pole (matice) je v operační paměti ukládáno po řádcích

```
int matice[4][3];             // definice dvourozměrného pole
prom = matice[0][0];          // přiřazení hodnoty prvního prvku matice do prom
prom = matice[3][2];          // přiřazení hodnoty posled. prvku matice do prom
matice[radky][sloupce] = prom;
int pole[][3] = {{3,6,9},{2,4,6}};       // inicializace dvourozměrného pole
```

při předávání vícerozměrného statického pole do funkce je určení počtu prvků pole povinné kromě první dimenze

```
#define RADKY 4                                   // rozměry pole
#define SLOUPCE 3

void funkce1(int matice[RADKY][SLOUPCE]);      // hlavičky funkcí
void funkce2(int matice[][SLOUPCE]);

int main()
{
      int pole[RADKY][SLOUPCE];                   // definice pole

      funkce1(pole);                              // volání funkcí
      funkce2(pole);

      return 0;
}
```

dynamické vícerozměrné pole

```
int *pole[4];                                   // semidynamické pole o 4 řádcích
pole[0] = (int *) malloc(7 * sizeof(int));
pole[1] = (int *) malloc(5 * sizeof(int));
pole[2] = (int *) malloc(7 * sizeof(int));
pole[3] = (int *) malloc(2 * sizeof(int));
```

k semidynamické matici se přistupuje jen pomocí indexů, protože jednotlivé řádky v paměti nemusí ležet bezprostředně za sebou

```
int **pole;                                     // dynamické pole
pole = (int **) malloc(4 * sizeof(int *));
pole[0] = (int *) malloc(7 * sizeof(int));
pole[1] = (int *) malloc(5 * sizeof(int));
pole[2] = (int *) malloc(7 * sizeof(int));
pole[3] = (int *) malloc(2 * sizeof(int));
```

```
#define RADKY 2                                         // rozměry pole
#define SLOUPCE 4

void funkce(int **pole);                                // hlavička funkce

int main()
{
      int **pole;                                       // definice pole
      pole = (int **) malloc(RADKY * sizeof(int *));
      pole[0] = (int *) malloc(SLOUPCE * sizeof(int));
      pole[1] = (int *) malloc(SLOUPCE * sizeof(int));

      funkce(pole);                                     // volání funkce

      free(pole[1]);                                    // uvolnění paměti
      free(pole[0]);
      free(pole);
      return 0;
}
```

práce s řetězci

v jazyce C je řetězec pole znaků typu char, každý řetězec končí nulovým znakem '\0', tzn. že řetězec "ahoj" zabírá v paměti 5 bajtů: {'a','h','o','j','\0'}, funkce jazyka C pro práci s řetězci pracují s nulovým znakem automaticky, ale na nulový znak je třeba myslet při definici řetězce a při práci s řetězcem jakožto polem - znak po znaku

```
char retezec[20];           // definice řetězce o maximální délce 19 znaků

char retezec[5] = "ahoj";   // definice a inicializace řetězce o 5 znacích
char retezec[] = "ahoj";    // definice a inicializace řetězce o 5 znacích
char retezec[8] = "ahoj";   // definice a inicializace řetězce o 8 znacích

char *retezec;                                  // deklarace dynamického řetězce
retezec = (char *) malloc(20 * sizeof(char));
```

do řetězce (statického ani dynamického) nemůžeme přiřadit text pomocí operátoru =, tzn. retezec = "ahoj"; je cesta do pekel! u statického řetězce měníme obsah statického ukazatele a u dynamického řetězce ztratíme odkaz na alokovanou paměť, do řetězce přiřazujeme text pomocí funkce strcpy() z knihovny <string.h>

```
strcpy(retezec, "ahoj");    // přiřazení textu do řetězce
```

pole řetězců

```
char pole[3][10];                   // definice statického pole řetězců

strcpy(pole[0], "ahoj");            // nastavení hodnoty prvního řetězce
strcpy(pole[1], "cau");             // nastavení hodnoty druhého řetězce
strcpy(pole[2], "nazdar");          // nastavení hodnoty třetího řetězce

pole[0][3] = 'y';                   // nastavení čtvrtého znaku prvního řetězce

for(int i = 0; i < 3; i++)          // výpis celého pole pomocí indexů
      printf("%s\n", pole[i]);
```

```
char *pole[3];                      // deklarace semidynamického pole řetězců

pole[0] = (char *) malloc(10 * sizeof(char));
pole[1] = (char *) malloc(20 * sizeof(char));
pole[2] = (char *) malloc(15 * sizeof(char));

strcpy(pole[0], "ahoj");            // nastavení hodnoty prvního řetězce
strcpy(pole[1], "cau");             // nastavení hodnoty druhého řetězce
strcpy(pole[2], "nazdar");          // nastavení hodnoty třetího řetězce

pole[0][3] = 'y';                   // nastavení čtvrtého znaku prvního řetězce

for(int i = 0; i < 3; i++)          // výpis celého pole pomocí aritmetiky ukaz.
      printf("%s\n", *(pole + i));

free(pole);                         // uvolnění paměti
```

```
char **pole;                        // deklarace dynamického pole řetězců

pole = (char **) malloc(3 * sizeof(char *));
pole[0] = (char *) malloc(10 * sizeof(char));
pole[1] = (char *) malloc(20 * sizeof(char));
pole[2] = (char *) malloc(15 * sizeof(char));

strcpy(pole[0], "ahoj");            // nastavení hodnoty prvního řetězce
strcpy(pole[1], "cau");             // nastavení hodnoty druhého řetězce
strcpy(pole[2], "nazdar");          // nastavení hodnoty třetího řetězce

pole[0][3] = 'y';                   // nastavení čtvrtého znaku prvního řetězce

for(int i = 0; i < 3; i++)          // výpis celého pole pomocí indexů
      printf("%s\n", pole[i]);

for(int i = 0; i < 3; i++)          // uvolnění paměti
      free(pole[i]);
free(pole);
```

11_datový typ struktura, unie, výčet

definice struktury

struktura uchovává pod jedním identifikátorem více údajů různého datového typu, vztahujících se k jednomu objektu

```
struct osoba                        // deklarace nové struktury
{
      char jmeno[30];
      char prijmeni[30];
      unsigned vek;
};

struct osoba moje_struct;           // definice vlastní struktury typu osoba
```

```
typedef struct                      // deklarace nového datového typu struktura
{
      char jmeno[30];
      char prijmeni[30];
      unsigned vek;
} OSOBA;

OSOBA moje_struct;                  // definice vlastní struktury
```

```
typedef struct osoba                // deklarace nové struktury a
{                                   // nového datového typu struktura
      char jmeno[30];
      char prijmeni[30];
      unsigned vek;
} OSOBA;

OSOBA moje_struct;                  // definice vlastní struktury
```

práce se strukturou

k položkám staticky definované struktury přistupujeme pomocí operátoru .

```
OSOBA moje_struct = {"Petr","Novak",20};  // inicializace prvků pole

strcpy(moje_struct.jmeno, "Pavel");       // naplnění struktury daty
strcpy(moje_struct.prijmeni, "Svoboda");
moje_struct.vek = 35;
```

```
OSOBA o1, o2;                             // definice vlastních struktur
o1 = o2;                                  // zkopírování dat z o2 do o1
```

vnořená struktura

```
typedef struct tmaturita            // deklarace struktury
{
      unsigned short rok;
      float prumer;
} TMATURITA;

typedef struct student              // deklarace struktury
{
      char jmeno[30];
      unsigned short vek;
      TMATURITA maturita;           // vnořená struktura
} STUDENT;

STUDENT st;                         // definice vlastní struktury

strcpy(st.jmeno, "Petr Novak");     // naplnění struktury daty
st.vek = 20;
st.maturita.rok = 2006;
st.maturita.prumer = 2.1;
```

pole struktur

```
OSOBA moje_struct[30];                    // definice pole 30 struktur

for(int i = 0; i < 30; i++)               // naplnění pole struktur daty
{
      strcpy(moje_struct[i].jmeno, "Petr");
      strcpy(moje_struct[i].prijmeni, "Novak");
      moje_struct[i].vek = 20;
}
```

struktura a dynamická alokace

k položkám dynamicky definované struktury přistupujeme pomocí operátoru -> nebo pomocí operátoru . spolu s dereferenčním operátorem

```
typedef struct osoba
{
      char *jmeno;
      char *prijmeni;
      unsigned vek;
} OSOBA;

OSOBA *os;                                      // definice struktury
os = (OSOBA *) malloc(sizeof(OSOBA));           // alokace paměti pro strukturu

os->jmeno = (char *) malloc(30 * sizeof(char)); // alokace paměti pro řetězec
os->prijmeni = (char *) malloc(30 * sizeof(char));

strcpy(os->jmeno, "Pavel");                     // naplnění struktury daty
strcpy(os->prijmeni, "Svoboda");
os->vek = 35;

free(os->jmeno);                                // uvolnění paměti
free(os->prijmeni);
free(os);
```

spojový seznam

dynamický spojový seznam je struktura ukazující sama na sebe, obsahuje ukazatel na další prvek

```
typedef struct tslovo               // definice struktury spojového seznamu
{
      char *text;
      unsigned cetnost;
      struct tslovo *p_dalsi;
} TSLOVO;
```

datový typ unie

unie je podobná struktuře s tím rozdílem, že v unii může být v jednom okamžiku jen jedna položka, velikost unie je rovna nejdelší položce, unie se používá když údaj nabývá právě jedné hodnoty přičemž není jasný datový typ této hodnoty, unie šetří paměť

```
typedef union                       // deklarace unie
{
      char znak;
      int cislo;
      float real;
} TUNION;

TUNION u;                           // definice vlastní unie

u.znak = 'A';                       // práce s položkou unie
u.real = 2.4;
printf("%c", u.znak);               // chyba! v unii je uložena položka real
```

výčtový datový typ enum

enum se používá, máme-li předem definovanou množinu hodnot, jichž může údaj nabývat

```
typedef enum                        // deklarace výčtového typu
{
      PONDELI, UTERY, STREDA, CTVRTEK, PATEK, SOBOTA, NEDELE
} DNY;

DNY d1, d2;                         // definice vlastního výčtového typu

d2 = UTERY;
if(d2 == UTERY) printf("utery");
```

12_práce se soubory

otevření a uzavření souboru

k souborům se přistupuje pomocí proměnné datového typu FILE*, která se většinou definuje jako globální, funkce fopen() z knihovny <stdio.h> otevírá soubor (textový i binární), prvním parametrem funkce je název/cesta k souboru, druhým parametrem je označení módu pro otevření souboru, funkce vrací ukazatel typu FILE* na daný soubor, pokud se otevření souboru nepovede, funkce vrací hodnotu NULL, při každém otevírání souboru je nutné testovat zda se otevření podařilo, funkce fclose() z knihovny <stdio.h> uzavírá otevřený soubor, parametrem je ukazatel typu FILE* na daný soubor, pokud se uzavření souboru povede, funkce vrací hodnotu 0, pokud se nepovede, funkce vrací symbolickou konstantu EOF (end of file) z knihovny <stdio.h>

mód pro otevření souboru	**popis**
"r"	textový soubor pro čtení
"w"	textový soubor pro zápis resp. přepis
"a"	textový soubor pro zápis na konec resp. přípis
příznak b ("rb","wb","ab")	binární soubor
příznak + ("r+","w+","a+","rb+","wb+","ab+")	soubor pro zápis i čtení

```
FILE *soubor;                                   // definice odkazu na soubor

if((soubor = fopen("c:\\data\\readme.txt", "r")) == NULL)  // otevření souboru
{
      fprintf(stderr, "Chyba pri otevirani souboru!\n");
      return -1;
}

if(fclose(soubor) == EOF)                                  // zavření souboru
{
      fprintf(stderr, "Chyba pri zavirani souboru!\n");
      return -1;
}
```

13_literatura

[1] KADLEC, Václav. *Učíme se programovat v jazyce C* . [s.l.] : Computer Press, 2002. 294 s. ISBN 80-7226-715-9.