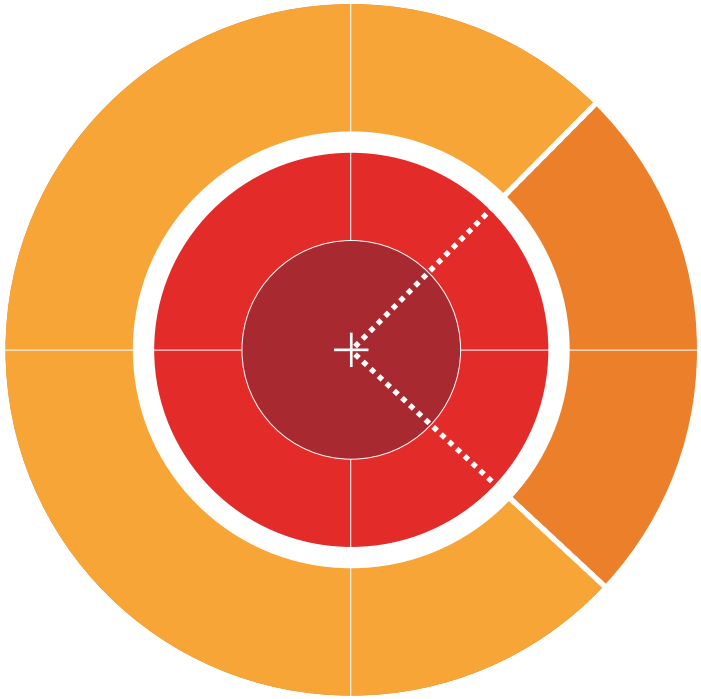


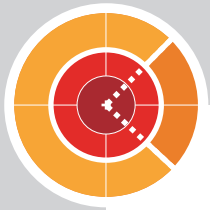
Deklarace proměnných -

- **deklarování jména** v jazyce C znamená spojení určitého identifikátoru s nějakým objektem jazyka (např. proměnnou, funkcí, typem, ...)

```
int a, b = 5;

int main(int argc, char *argv[]) {
    int a = 3, b = 5;
    float c, d;

    c = d = a * b;

    return 0;
}
```

globální proměnné

lokální proměnné (zakrývají globální proměnné)



Celočíselné datové typy

`[signed] short [int]`	-32768 .. 32767
`[signed] int` `signed`	v moderních překladačích ANSI C obvykle -2^{31} .. $2^{31}-1$
`[signed] long [int]`	-2^{31} .. $2^{31}-1$
`unsigned short [int]`	0 .. 65535
`unsigned [int]`	obvykle 0 .. $2^{32}-1$
`unsigned long [int]`	0 .. $2^{32}-1$
`[signed | unsigned] char`	-128 .. 127 | 0 .. 255

Celočíselné datové typy

Tabulka 5-1: **Hodnoty definované v limits.h (ANSI C)**

Jméno	Minimální hodnota	Význam
CHAR_BIT	8	šířka typu char, v bitech
SCHAR_MIN	-127	minimální hodnota pro signed char
SCHAR_MAX	127	maximální hodnota pro signed char
UCHAR_MAX	255	maximální hodnota pro unsigned char
SHRT_MIN	-32,767	minimální hodnota pro short int
SHRT_MAX	32,767	maximální hodnota pro short int
USHRT_MAX	65,535	maximální hodnota pro unsigned short
INT_MIN	-32,767	minimální hodnota pro int
INT_MAX	32,767	maximální hodnota pro int
UINT_MAX	65,535	maximální hodnota pro unsigned int
LONG_MIN	-2,147483647	minimální hodnota pro long int
LONG_MAX	2,147483647	maximální hodnota pro long int
ULONG_MAX	4,294967295	maximální hodnota pro unsigned long
CHAR_MIN	SCHAR_MIN nebo 0❶	minimální hodnota pro char
CHAR_MAX	SCHAR_MAX nebo UCHAR_MAX❷	maximální hodnota pro char

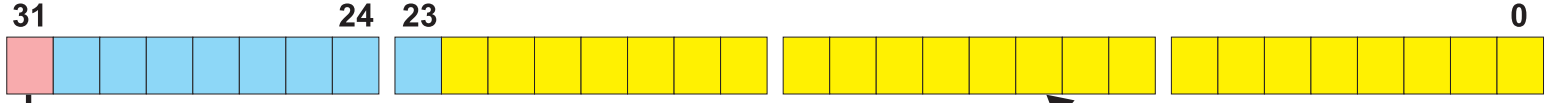
Reálné datové typy

`float` cca. $\pm 3{,}403 \times 10^{38}$

`double` cca. $\pm 1{,}798 \times 10^{308}$

`long double` obvykle jako **`double`** (někdy 80 bitů)

IEEE Standard for Binary-Point Arithmetic No. 754-1985

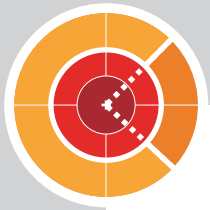


znaménko (0 = +, 1 = -)

exponent (kód s posunutou 0; 0 ~ 127)

normalizo-vaná mantisa

$$h_{float} = (-1)^{s} \times 2^{(exp\ -\ 127)} \times (1 + \textit{mantisa})$$



Datový typ ukazatel

```
int *i;
```
ukazatel na celé číslo

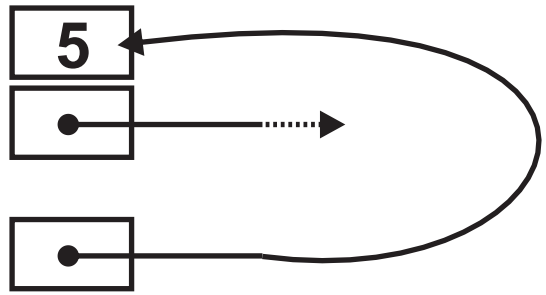
```
float *hodnota;
```
ukazatel na reálné číslo

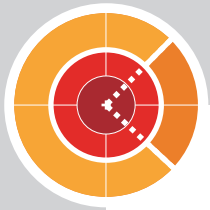
```
char *zn;
```
ukazatel na znak

```
float (*vypocet) ();
```
ukazatel na funkci, která vrací reálné číslo

```
void *uk;
```
generický ukazatel (může ukazovat na cokoli)

```
int cislo = 5;
int *uk_na_cislo;

uk_na_cislo = &cislo;
```





Datový typ pole

- pole **nelze** vytvořit z typu **`void`** a typu "funkce vracející hodnotu"
- pole je **vždy indexované** od nuly

```
int i[3];
char zn[20];
```
pole 3 intů
pole 20 znaků

```
int *uk[3];
float matice[5][5];
```
pole 3 ukazatelů na inty
pole 5x5 floatů (matice)

```
int i[20], j;
int *ip[20];

for (j = 0; j < 20; j++)
    ip[j] = &i[j];
```

Vztah datového typu pole a ukazatel

- vztah je těsný, často lze pole a ukazatel zaměnit

```
int i[20], *ip;

ip = i;
ip = &i[0];
```

} ekvivalentní operace

- vztah je dán tzv. *ukazatelovou aritmetikou*:

```
a[i] ~ *((a) + (i))
```

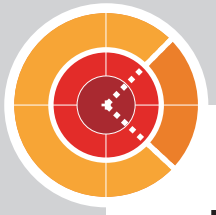
- implementačně je pole vlastně ukazatel na první prvek pole

Datový typ výčet

- kvůli zjednodušení zápisu
- množina celočíselných hodnot reprezentovaných identifikátory (tzv. **výčtové konstanty**)

```
enum ryby {kapr, uhor, stika, pstruh}
  chytil_jsem, bydli_v_rybnice;

enum ryby sezrali_jsme;
enum ryby slizky = uhor;
```

- překladač přiřadí každé výčtové konstantě hodnotu (první má hodnotu 0)
- můžeme to také udělat po svém



Datový typ výčet - vlastní určení hodnot výčtových konstant

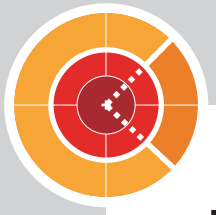
```
enum ryby {kapr = 1, uhor = 2, stika = 3,
  pstruh = 4} chytil_jsem;
```

```
enum ryby {kapr, uhor, stika = 3, lin}
  chytil_jsem;
```

0 1 4 (!!!)

- je možné přiřadit dvěma různým výčtovým konstantám stejnou hodnotu,pak ale **pozor**!

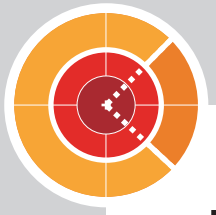
```
enum ryby {kapr = 1, uhor = 1, stika = 2,
  pstruh = 3} chytil_jsem;

    tato zjevně nesmyslná podmínka je splněna
if (kapr == uhor) { ... }
```



Datový typ struktura

```
struct complex {
    double real;
    double imag;
};

struct complex a, b;

a.real = 5.0;
a.imag = -2.0;
```

zde nemá smysl inicializovat, jedná se o abstraktní definici



Funkce pracující s typem struktura

- struktura může být parametrem funkce a může být funkcí vrácena

```
struct complex {
   double real;
   double imag;
};

struct complex mul(struct complex a,
                   struct complex b) {
   struct complex prod;

   prod.real = (a.real * b.real - a.imag * b.imag);
   prod.imag = (a.real * b.imag + a.imag * b.real);

   return prod;
}
```

Ukazatel na typ struktura (a spec. případy definice)

- někdy je třeba provést tzv. neúplnou definici

```
struct tree_node {
    struct tree_node *left_son;
    struct tree_node *right_son;
    int data;
};
```

definice struktury je úplná až zde

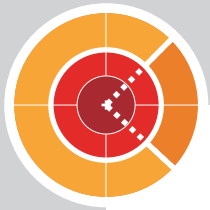
tzn. zde se odkazujeme na zatím nedodefinovanou strukturu

Ukazatel na typ struktura (a spec. případy definice)

- lze zajít ještě dál provést definici dvou vzájemně se odkazujících struktur

```
struct cell;

struct header {
    struct cell *first;
    ...
};

struct cell {
    struct header *head;
    ...
};
```

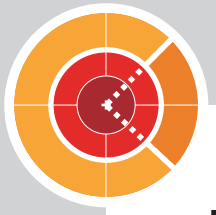
nutno uvést tuto neúplnou definici před použitím zde



Struktura jako bitové pole

- lze "ušetřit" místo tím, že pro celočíselné složky, které nezabírají místo celého datového typu, vyhradíme jen tolik bitů, kolik opravdu potřebují

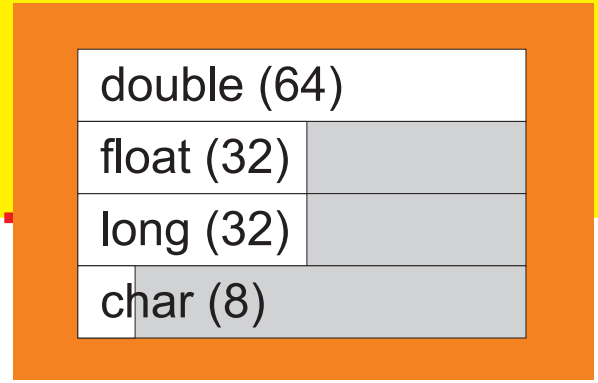
```
struct planner {
    unsigned day_of_week:3;
    unsigned day:5, month:4;
    ...
};
```



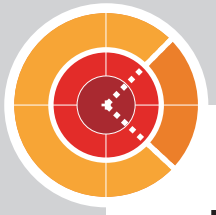
Datový typ union

- definuje se stejně jako **`struct`**

```
union value {
    double d;
    float f;
    unsigned long int i;
    char c;
};
```



- union může v daném čase obsahovat pouze jednu hodnotu
- velikost unionu je dána jeho největší složkou



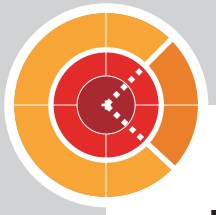
Datový typ funkce vracející T

- T může být cokoliv **kromě** **pole** a **funkce vracející T**

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

float (*func) (float x, float y);

float add(float a, float b) {
   return a + b;
}

float mul(float a, float b) {
   return a * b;
}

int main() {
   float a = 5, b = 5;

   func = &add;
   printf("add %f\n", func(a, b));

   func = &mul;
   printf("mul %f\n", func(a, b));

   return 0;
}
```

definujeme proměnnou `func` jako ukazatel na fci 2 parametrů

do proměnné typu ukazatel na funkci nyní vložíme adresu existující funkce a následně "zavoláme" ukazatel



Pojmenování definovaného typu

```
typedef int cele_cislo;

typedef struct { double re;
  double im; } complex;

typedef struct thenode {
    int key;
    struct thenode *left_son;
    struct thenode *right_son;
} node;

cele_cislo i = 5;
node uzel;
complex z;
```

Definice konstant -

- lze provést dvěma způsoby: pomocí **preprocesoru** nebo pomocí *kvalifikátoru typu*

zde NIKDY středník

```
#define PI 3.14159

const float EULER = 2.71;

int main() {
    return PI * EULER;
}
```

direktiva preprocesoru

kvalifikátor typu

Explicitní určení typu celočíselné konstanty

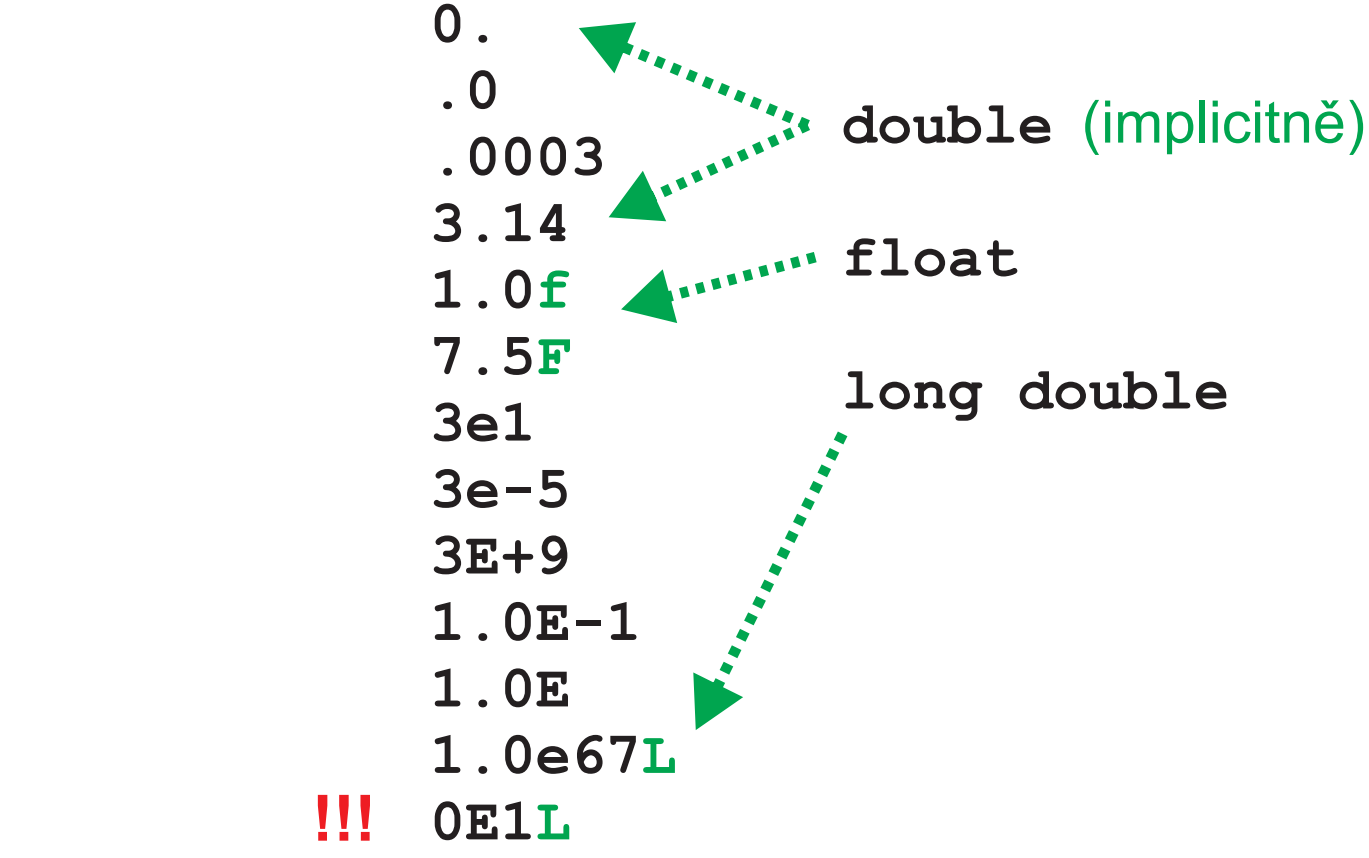
desítková

osmičková

šestnáctková

```
12345678      int
01777777      unsigned int
0X8FDD2A      long int
0x8FDD2A      unsigned long int

12345678U     unsigned int
01777777U     unsigned long int
0x8FDD2AU

12345678L     long int
01777777L     unsigned long int
0x8FDD2AL

12345678UL    unsigned long int
01777777UL
0x8FDD2AUL
```

Explicitní určení typu reálné konstanty



- lze použít i l (malé L), ale snadno se splete s 1, takže raději nepoužívat.

Definice funkcí -

```
float addition(float a, float b) {
    float c;

    c = a + b;

    return c;
}
```

- vnořené funkce nejsou dovoleny
- rekurze je dovolena

Přiřazení -

```
a = b = c = 5;
a = b;
```

- nezaměňovat (hlavně v **`if`**) s porovnáním
- přiřazovací příkaz "vrací" hodnotu (tu přiřazenou)

- v kombinaci s podmíněnou hodnotou výrazu

```
a = (volba == 'p') ? PI : EULER;
return (a > 0) ? 1 : -1;
```

V/V: printf a scanf -

- knihovní fce (`#include <stdio.h>`), nejsou součástí syntaxe jazyka C, ale jsou tradičně v rámci syntaxe probírány

ukazatel na místo v paměti (tedy adresa - op **&**)

```
int cislo;

scanf("%d", &cislo);
printf("Bylo zadano %d\n.", cislo);
```

řídicí řetězec (konverzní specifikace)

Konverzní specifikace -

Tabulka 15-2: **Výstupní konverzní specifikace**

Konverze	Definované příznaky - + # 0 *mezera*	Určení velikosti	Typ parametru	Implicitní přesnost❶	Výstup
d,i❷	- + 0 *mezera*	*žádné*	`int`	1	dd...d
		h	`short`		-dd...d
		l	`long`		+dd...d
u	- + 0 *mezera*	*žádné*	`unsigned int`	1	dd...d
		h	`unsigned short`		
		l	`unsigned long`		
o	- + # 0 *mezera*	*žádné*	`unsigned int`	1	oo...o
		h	`unsigned short`		0oo...o
		l	`unsigned long`		
x,X	- + # 0 *mezera*	*žádné*	`unsigned int`	1	hh...h
		h	`unsigned short`		0xhh...h
		l	`unsigned long`		0Xhh...h
f	- + # 0 *mezera*	*žádné*	`double`	6	d...d.d...d
		L	`long double`		-d...d.d...d
					+d...d.d...d
e,E	- + # 0 *mezera*	*žádné*	`double`	6	d.d...de+dd
		L	`long double`		-d.d...dE-dd
g,G	- + # 0 *mezera*	*žádné*	`double`	6	*jako* `e,E` *nebo* `f`
		L	`long double`		
c	-	*žádné*	`int`	1	c
s	-	*žádné*	`char *`	∞	cc...c
p❷	*definováno implem.*	*žádné*	`void *`	1	*definováno impl.*
n❷		*žádné*	`int *`	*neex.*	*žádný*
		h	`short *`		
		l	`long *`		
%		*žádné*	*žádný*	*neex.*	%

Ukázka možností konverze čísla - printf

Příklady konverze d

Příklad formátu	Příklad výstupu / Hodnota = 45	Příklad výstupu / Hodnota = -45
%12d	45	-45
%012d	000000000045	-00000000045
% 012d	00000000045	-00000000045
%+12d	+45	-45
%+012d	+00000000045	-00000000045
%-12d	45	-45
%- 12d	45	-45
%-+12d	+45	-45
%12.4d	0045	-0045
%-12.4d	0045	-0045

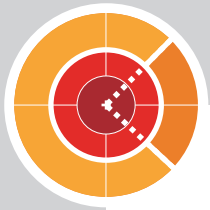
❶ Implicitní přesnost, pokud se žádná nespecifikuje
❷ Dostupné v ANSI C, jinde možná vzácné. `i` a `d` jsou při výstupu shodné.

Příkaz skoku - nepoužívat!

- používat jen když už opravdu **nezbývá nic jiného**

návěští

```
...
Restart:
while (isspace(curr))
   curr = fgetc(input);

if (curr == '`') {
    do {
        curr = fgetc(input);
    } while (curr != '`');
    goto Restart;
}
...
```



Příkaz skoku - jak určitě nepoužívat

- **za žádných okolností neskákejte:**
  (i) dovnitř *then* a *else* bloků příkazu **`if`** z vnějšku
  (ii) z bloku *then* do bloku *else* a naopak
  (iii) dovnitř těla příkazu **`switch`** nebo cyklu z vnějšku
  (iv) dovnitř složeného příkazu (bloku) z vnějšku

- příkaz **`goto`** lze **vždy nahradit** jiným mechanismem
- kdykoli je to možné, je lepší použít místo **`goto`** např. **`break`**, **`continue`** nebo **`return`**

Prázdný příkaz - ;

- prázný příkaz se skládá pouze ze **středníku**

```
char *p;
…
while (*p++)
    ;
```

nalezení konce řetězce
(typická zkratkovitá konstrukce jazyka C, kterou je třeba komentovat, jinak se stává po pár dnech **nečitelnou** i pro jejího autora)

```
if (e) {
    …
    goto L;
    …
    L:;
} else …
```

za návěštím musí následovat příkaz - nesmí následovat konec bloku }