

1. Minimální program

– do souboru main.c uložíme následující kód a pomocí „F9“ ho zkompilujeme a spustíme:

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

```
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

Vypíše na obrazovku „Hello world!“ bez uvozovek a odřádkuje.

Ukončí program. Nula znamená, že vše proběhlo bez chyby.

2. Výpis do konzole

```
printf("Text");
```

– vypíše na obrazovku slovo Text (bez uvozovek)

– pokud chceme vypsát některý speciální znak nebo znak se speciálním významem, musíme ho tzv. oescapovat tak, že před něj napíšeme zpětné lomítko (\):

\n	Nový řádek (new line)
\t	Tabulátor (několik mezer za sebou)
\a	Pípnutí (alert)
\"	Dvojitá uvozovka
\\	Zpětné lomítko

```
printf("Text %d text %d ...", číslo1, číslo2, ...);
```

– vypíše text v uvozovkách, ale znaky „%d“ nahradí postupně číslem 1, 2 atd.

– znak za procentem určuje, v jakém formátu se má číslo vypsát:

%d	Vypíše číslo v desítkové soustavě.
%o	Vypíše číslo v osmičkové soustavě.
%x	Vypíše číslo v šestnáctkové soustavě malými písmeny.
%X	Vypíše číslo v šestnáctkové soustavě velkými písmeny.
%%	Vypíše znak procenta.

3. Komentáře

= texty ignorované překladačem

– používají se pro psaní poznámek nebo zakomentování části kódu, který nechceme vykonat

Řádkový komentář

– začíná dvěma lomítky a končí koncem řádku

– lze ho napsat na prázdný řádek i za příkaz

```
// komentář na prázdném řádku
```

```
printf("Hello world!"); // komentář za příkazem
```

Blokový komentář

- začíná znaky „/*“ a končí znaky „*/“
- mezi hvězdičkami může být libovolný počet řádků
- může obsahovat i řádkové komentáře

```
/*Komentář*/  
/*  
    Několika řádkový  
    Komentář obsahující  
    // řádkový komentář  
*/
```

2. lekce

4. Celočíselné proměnné

Definování proměnné

```
int i, cislo1;  
int j;
```

- po definování proměnné je v ní náhodná hodnota, před prvním použitím např. ve výpočtu nebo výpisu je potřeba do ní uložit nějakou hodnotu

Uložení hodnoty do proměnné

```
i = 5; // v proměnné i je 5  
cislo1 = i * 2 - 6; // v proměnné cislo1 je 5 (5 * 2 - 6)
```

Načtení hodnoty do proměnné z klávesnice

```
scanf("%d", &i);  
scanf("%d %d", &i, &cislo);
```

- před názvy proměnných se musí psát ampersand (&) !!!
- jednotlivé proměnné ve formátovacím řetězci oddělovat vždy jen mezerou, ačkoliv je možné používat i jiné znaky, ale pokud je pak uživatel přesně nezadá, tak se program začne chovat neočekávaně
- při čtení znaků z klávesnice se jako mezera interpretuje jakýkoliv **bílý znak** (mezera, konec řádku, tabelátor)
- znaky se nečtou přímo z klávesnice, ale z **klávesnicového bufferu**, takže pokud jsme při předchozím psaní na klávesnici toho napsali více, tak se text navíc použije pro následující volání funkce scanf

Adresa proměnné

- proměnné se ukládají do operační paměti (RAM)
- při programování se na proměnné odkazujeme jejich názvy, ale počítač používá jejich adresy
- aktuálně použitou adresu proměnné zjistíme přidáním ampersandu před název proměnné, např. &i
- adresa proměnné může být při každém spuštění programu jiná
- adresa proměnné je pořadové číslo prvního bytu proměnné v paměti
- byte (česky bajt) = 8 bitů; značení 1 B = 8 b
- do jednoho bytu se vejde pouze číslo od 0 do 255
- např. proměnná int má 4 byty, takže se do ní vejde číslo od 0 do 4 294 967 295 (cca 4.3 miliardy)

```
int i;  
i = 5;  
printf("Hodnota proměnné i: %d\n", i); // vypíše 5  
printf("Adresa proměnné i: %d\n", &i); // náhodné číslo, např. 23624
```

5. Celočíselné dělení

- celočíselné dělení je dělení ve stylu „13 děleno 5 je 2 zbytek 3“
- celočíselné dělení se dělá pomocí znaku lomítka (/), zbytek po dělení se nazývá **dělení modulo** a značí se procentem (%)

```
i = 13 / 5; // do proměnné i se uloží 2
i = 13 % 5; // do proměnné i se uloží 3; čteme 13 děleno modulo 5
```

6. Desetinná čísla

- na počítači se zapisují vždy s desetinnou tečkou, místo české čárky (např. 3.52 místo 3,52)
- definují se pomocí klíčového slova **float** (používá se ve většině programovacích jazyků)
- desetinná čísla se na počítači ukládají v různých formátech; slovo float znamená, že se číslo uloží tzv. s plovoucí (float) desetinnou tečkou
- v printf a scanf se místo %d použije %f

```
float cislo; // definování proměnné cislo
cislo = 3.52; // uložení čísla 3,52 do proměnné cislo
printf("%f", cislo); // vypsání čísla
scanf("%f", &cislo); // načtení desetinného čísla z klávesnice
```

- pokud chceme určit, kolik má funkce printf vypsát číslic, používáme modifikovanou syntaxi %f:

```
printf("%7.3f", cislo);
```

- vypíše vždy alespoň sedm znaků (započítává se i desetinná tečka)

- zokrouhlí číslo tak, aby mělo tři desetinná místa

- pokud je číslo příliš krátké, vypíšou se před něj mezery tak, aby mělo sedm znaků; pokud je naopak číslo delší než sedm znaků, tak se vypíše delší, ale pořad bude mít tři desetinná místa

Například

```
printf("%6.2f\n", 123.45);
printf("%6.2f\n", 1.5);
printf("%6.2f\n", 2.47612);
printf("%6.2f\n", 17);
printf("%6.2f\n", 111222.6666);
```

vypíše

```
123.45      ... číslo má přesně 6 znaků
  1.50      ... číslu se doplnila nula a přidaly se před něj dvě mezery
  2.48      ... číslo se zokrouhlilo
 17.00      ... číslu se doplnila desetinná místa
111222.67   ... číslo bylo příliš dlouhé, tak se vypsalo celé (říkáme, že přeteklo)
```

- pokud bychom chtěli před kratší čísla vypsát místo mezer nuly, tak se místo "%6.2f" zadá "%06.2f"

```
printf("%06.2f", 1.37); // vypíše 001.37
```

- podobně lze tyto zápisy použít i pro celá čísla, jen se vynechá počet desetinných míst, např.

```
printf("%5d\n", 12345);
printf("%5d\n", 12);
printf("%05d\n", 12);
```

vypíše

```
12345
  12
00012
```

– pokud bychom chtěli pouze specifikovat počet desetinných míst, tak se místo "%6.2f" zadá jen "%.2f"
`printf("%.2f", 123.4); // vypíše 123.40`

3. lekce

7. Logické (pravdivostní) operátory

Pravdivostní hodnoty

- pravda (true) = jakékoliv nenulové číslo (typicky 1)
- nepravda (false) = nula

Pravdivostní operátory

- $A \parallel B$ = „A nebo B“ = výsledek je nepravdivý, pouze pokud je nepravdivá hodnota A i B
- $A \&\& B$ = „A a současně B“ = výsledek je pravdivý, pouze pokud je pravdivé A i B
- $!A$ = „negace A“ = výsledek má opačnou pravdivost, než A

8. Priorita operátorů

– udává, které operátory mají přednost před kterými

* / %	(nejvyšší priorita)
+ -	
< > <= >= == !=	
&&	(nejnižší priorita)

- operátory s vyšší prioritou se vyhodnocují dříve než operátory s nižší prioritou (patrné z pozice * a +)
- pokud mají operátory stejnou prioritu, tak se vyhodnocují zleva doprava tak, jak jsou zapsané v kódu
- pokud si nejsem jistý prioritou operátoru nebo pokud jí chci změnit, tak stačí výraz ozávkovat
- tabulka je zjednodušená

9. Větvení programu

Syntaxe 1

```
if (podmínka) {  
    ... příkazy ...  
}
```

– příkazy ve složených závorkách se provedou, pouze pokud je podmínka splněná (je true)

Syntaxe 2

```
if (podmínka) {  
    ... příkazy pro true ...  
}  
else {  
    ... příkazy pro false ...  
}
```

- pokud je podmínka splněna (je true), provedou se pouze příkazy v prvních složených závorkách
- pokud podmínka není splněna (je false), provedou se pouze příkazy z druhých složených závorek

- příkazy ve složených závorkách se nazývají **blok**
- příkazy v bloku je kvůli přehlednosti kódu dobré odsazovat o dvě mezery vpravo

10. Zkrácené zapisování bloků

- prázdný blok `{ }` lze zapsat, jako samotný středník
- pokud je v bloku jen jeden příkaz, lze vynechat složené závorky

Příklad. Delší verze kódu:

```
if (i > 0) {
    ;
}
else {
    printf("Cislo neni kladne.");
}
```

Zkrácená verze kódu:

```
if (i > 0)
    ;
else
    printf("Cislo neni kladne.");
```

11. Konstrukce else if

```
if (i == 1) {
    // i je 1
}
else if (i == 2) {
    // i je 2
}
else if (i == 3) {
    // i je 3
}
else {
    // i je cokoliv jiného než 1, 2 a 3
}
```

12. Podmíněný výraz

- jediný **ternární operátor** v C (= operátor se třemi operandy)
- syntaxe: *(podmínka) ? splněno : nesplněno*
- výsledkem operace je *splněno*, pokud je *podmínka* true, jinak je výsledkem *nesplněno*
- tento operátor má jednu z nejnižších priorit vůbec, takže jednotlivé části jsou vyhodnoceny vždy jako první a proto je není třeba závorkovat; závorku kolem podmínky doporučuji psát jen kvůli přehlednosti

Příklad. Následující dva kódy jsou rovnocenné:

```
// Dlouhá verze
if (i > 0)
    i = 10;
else
    i = 20;

// Krátká verze
i = (i > 0) ? 10 : 20;
```

13. Zkrácené zápisy aritmetických operací

Dlouhý zápis	Zkrácený zápis	Poznámka
<code>i = i + 1</code>	<code>i++</code>	„inkrementace proměnné“
<code>i = i - 1</code>	<code>i--</code>	„dekrementace proměnné“
<code>i = i + 5</code>	<code>i += 5</code>	
<code>i = i - 5</code>	<code>i -= 5</code>	
<code>i = i * 5</code>	<code>i *= 5</code>	

... atd. pro většinu operátorů (viz tabulka priorit operátorů)

14. While cyklus

```
while (podmínka) {  
    ... příkazy ...  
}
```

- příkazy se opakovaně vykonávají tak dlouho, dokud je splněná podmínka za slovem while
- pokud podmínka není splněná ani na začátku, tak se příkazy nevykonají vůbec
- závorka kolem podmínky je povinná

Příklad.

```
int i = 5;  
while (i <= 10) {  
    if (i > 5)  
        printf(", ");  
    printf("%d", i);  
    i++;  
}  
printf(";%d", i);
```

- program vypíše čárkou oddělený seznam čísel od 5 do 10 včetně; pak vypíše středník a číslo 11

5. lekce

15. Řízení běhu cyklu

- běh každého cyklu lze řídit pomocí těchto příkazů:
 - break** – okamžitě ukončí cyklus tím, že skočí za ukončovací složenou závorku těla cyklu
 - continue** – ukončí se současná iterace cyklu tak, že se skočí za poslední příkaz cyklu před ukončovací složenou závorkou; cyklus pokračuje dál

Příklady.

```
int i = 5;  
while (i <= 10) {  
    if (i == 7)  
        continue;  
    printf("%d ", i);  
}
```

- vypíše čísla od 5 do 10 včetně, ale vynechá číslo 7

```
int i = 5;  
while (i <= 10) {  
    if (i == 7)  
        break;  
    printf("%d ", i);  
}
```

- vypíše pouze čísla 5 a 6

16. Cyklus do-while

```
do {  
    ... příkazy ...  
} while (podmínka);
```

- opakovaně provádí příkazy ve složených závorkách a to tak dlouho, dokud je splněná podmínka
- pořadí je takové, že se nejdřív provedou příkazy a pak se teprve testuje splnění podmínky; pokud je splněná, tak cyklus opět provede příkazy a pak opět testuje podmínku; pokud není splněná, cyklus skončí a běh programu pokračuje za cyklem
- cyklus běží vždy alespoň jednou
- alternativní zápis pomocí while:

```
... příkazy ...  
while (podmínka) {  
    ... stejné příkazy ...  
}
```

Příklad: srovnání while cyklu a cyklu do-while

```
1) int i = 5;  
   while (i < 2) {  
       printf("%d ", i);  
       i++;  
   }  
   printf(":%d", i);
```

```
2) int i = 5;  
   do {  
       printf("%d ", i);  
       i++;  
   } while (i < 2);  
   printf("%d", i);
```

- první program vypíše pouze číslo pět
- druhý program vypíše číslo pět mezeru a číslo šest

16. For cyklus

- velmi časté použití while cyklu:

```
int i; // iterační proměnná  
i = 1; // inicializace proměnné  
while (i <= 10) { // podmínka  
    ... příkazy ...  
    i++; // inkrementace / změna iterační proměnné  
}
```

- zkrácený zápis pomocí syntaxe cyklu for:

```
int i;  
for (i = 1; i <= 10; i++) { // inicializace ; podmínka ; inkrementace  
    ... příkazy ...  
}
```

- oba zápisy fungují naprosto rovnocenně
- for má v závorce středníkem oddělené tři části: inicializaci, podmínku a inkrementaci
- část inicializace se provádí pouze jednou a to hned na začátku ještě před testováním podmínky
- po vykonání inicializace se provádějí jednotlivé iterace cyklu
- iterace cyklu se skládá z testování podmínky, provedení příkazů a inkrementace proměnné
- pokud je podmínka pravdivá, tak se provedou příkazy a následně inkrementace proměnné

- pokud je podmínka nepravdivá, tak se nic neprovádí a cyklus okamžitě skončí
- pokud je podmínka nepravdivá hned na začátku, tak se před jejím testováním provede pouze inicializace, ale příkazy cyklu a část inkrementace se vůbec neprovádí
 - pokud se použije v příkazech příkaz **break**, tak se cyklus okamžitě ukončí; podmínka se již netestuje a neprovádí se inkrementace
 - pokud se použije v příkazech **continue**, tak se ukončí současné vykonávání příkazů a skočí se na inkrementaci a testování proměnné a cyklus běží dál

Příklady.

```
int i;
for (i = 0; i < 5; i++) {
    printf("%d ", i);
}
```

- vypíše čísla 0 až 4; v proměnné „i“ je po skončení cyklu číslo 5

```
int i;
for (i = 0; i < 5; i++) {
    if (i == 3)
        break;
    printf("%d", i);
}
```

- vypíše čísla 0 až 2; v proměnné „i“ je po skončení cyklu číslo 3

```
int i;
for (i = 0; i < 5; i++) {
    if (i == 3)
        continue;
    printf("%d", i);
}
```

- vypíše čísla 0, 1, 2 a 4; v proměnné „i“ je po skončení cyklu číslo 5

– ačkoliv má každá část za slovem for svůj logický význam, tak může být programátorem vyplněna libovolným výrazem, který vůbec nemusí souviset s faktickým významem této části a dokonce může být zcela vynechána:

<code>for (i = 1; j < 10; k--)</code>	... každá část se provádí s jinou proměnnou
<code>for (; i < 10; i++)</code>	... neprovádí inicializaci
<code>for (i = 1; i < 10;)</code>	... po vykonání příkazů se neprovádí inkrementace
<code>for (i = 1; ; i++)</code>	... při vynechání podmínky se podmínka bere jako splněná
<code>for (; ;)</code>	... nekonečný cyklus

- do části inkrementace se obvykle dává `i++`; `i--`; `i += 2` apod.

17. Funkce

Příklad definice funkce:

```
double mocnina(double zaklad, int exponent)
{
    int i;
    double vysledek = 1.0;

    for (i = 0; i < exponent; i++)
        vysledek *= zaklad;

    return vysledek;
}
```

Poznámky k funkci z příkladu:

- funkce se jmenuje „mocnina“
- návratový typ funkce je double, tj. funkce vrací výsledek typu double
- má dva argumenty: základ a exponent
- volání funkce a uložení výsledku do proměnné x:


```
x = mocnina(2.5, 3);
```
- funkční prototyp funkce (viz dále):


```
double mocnina(double zaklad, int exponent);
```
- první řádek se nazývá **hlavička** funkce
- příkazy mezi složenými závorkami se nazývají **tělo** funkce
- celý zápis hlavička + tělo se nazývá **definice** funkce
- pokud potřebujeme funkci volat ještě před její definicí (například pokud funkci voláme v main(), ale máme jí definovanou až za main()) tak na začátek souboru, vně kterékoliv funkce, zapíšeme **deklaraci** funkce
- **deklarace** funkce = sdělení překladači, jak se funkce jmenuje, jaký má návratový typ a argumenty; dělá se pomocí **funkčního prototypu** funkce = hlavička bez těla ukončená středníkem
- jakékoliv proměnné definované uvnitř těla funkce jsou přístupné pouze uvnitř této funkce; takové proměnné nazýváme jako **lokální**
- jakékoliv proměnné definované vně funkce ještě před její definicí jsou uvnitř funkce přístupné také; takové proměnné nazýváme jako **globální**
- ve funkci nejsou přístupné proměnné z funkce, která funkci zavolala; např. pokud ve funkci main() nadefinujeme proměnnou cislo a pak z ní zavoláme funkci test(), tak ve funkci test() nebude proměnná cislo existovat
- pokud ve funkci nadefinujeme proměnnou, jejíž název se shoduje s názvem již existující globální proměnné, tak lokální definice překryje globální definici, čili ve funkci nebude přístupná globální proměnná, ale pouze ta nová lokální
- lokální proměnné funkce vznikají při jejím zavolání a přestanou existovat s jejím ukončením; to znamená, že když funkci voláme znovu, tak její proměnné neobsahují hodnoty z předchozího volání, ale náhodné hodnoty
- **procedura** = funkce, která nevrací žádnou hodnotu; v hlavičce funkce místo návratového typu použijeme klíčové slovo **void**, např.:


```
void tisknihvezdy(int pocet)
```
- pokud funkce nemá argumenty, tak se v deklaraci funkce do závorky napíše klíčové slovo **void** a funkce se pak volá s prázdnými závorkami:


```
int nahodnecislo(void); // deklarace
x = nahodnecislo(); // volání
```
- za klíčové slovo **return** se napíše výraz, jehož hodnota má být funkcí vrácena jako výsledek její činnosti; současně s tím se ukončí její běh (podobně jako po break v cyklech)

– main() je rezervovaný název funkce, která musí být vždy přítomná v kódu a volá se s jeho spuštěním; vykonání **return** způsobí ukončení celého programu a předávaná hodnota je předána operačnímu systému; nula znamená, že vše proběhlo bez chyb, jiné číslo znamená chybu

18. Preprocesor

– textově zpracovává kód ještě před jeho kompilací

Makra

– definování **symbolické konstanty** (konstanty; makra bez parametrů):

```
#define název hodnota
#define M_PI 3.14
```

– definování **makra** (makra s parametry):

```
#define nadruhou(x) ((x)*(x))
#define obvodobdelnika(a, b) (2 * (a) + 2 * (b))
```

– oddefinování makra

```
#undef název_makra
```

– symbolická konstanta může být definována s vynecháním hodnoty; využívá se např. ve spojení s #ifdef, viz níže

– při definování makra s parametry je žádoucí dát kód makra do závorek a současně dávat závorky i kolem všech argumentů, aby nedošlo k nečekané interpretaci:

```
#define nadruhou(x) x * x
printf("%d", nadruhou(10-1)); // vypíše -1, protože se preprocesorem rozvine do:
printf("%d", 10 - 1 * 10 - 1);
```

Soubory

– vložení obsahu souboru na místo zadání příkazu:

```
#include "cesta k souboru"
```

– zahrnutí knihovny do souboru:

```
#include <název souboru knihovny>
#include <stdio.h>
```

Podmíněný překlad

1) #ifndef název_makra

```
// tato část se zahrne do finálního souboru, pokud makro existuje
#endif
```

2) #ifndef název_makra

```
// tato část se zahrne, pokud makro neexistuje
#endif
```

3) #if výraz

```
// tato část se zahrne, pokud je výraz pravdivý
// výraz smí obsahovat jen operátory, konstanty, další makra
#endif
```

4) #ifdef název_makra // identická syntaxe je pro #ifndef

```
// tato část se zahrne do finálního souboru, pokud makro existuje
#else
// tato část se zahrne do finálního souboru, pokud makro neexistuje
#endif
```

```

5) #if výraz
    // tato část se zahrne, pokud je výraz pravdivý
#elif výraz
    // pokud je první nepravdivý a druhý pravdivý
    // #elif může být použito několikrát, popřípadě zcela vynecháno
#else
    // pokud byly všechny výrazy nepravdivé
#endif

```

Speciální příkazy

`__LINE__` = současný řádek
`__FILE__` = současný soubor
`#error` zpráva = ukončí preprocessing a vypíše chybovou hlášku

19. Číselné soustavy

Převod z dvojkové do desítkové soustavy

– všimni si, jak lze z číslic 2, 5, 7, 3 a 4 vypočítat číslo 25 734:

2	5	7	3	4
---	---	---	---	---

$$\begin{aligned}
 &= 2 \cdot 10\,000 + 5 \cdot 1\,000 + 7 \cdot 100 + 3 \cdot 10 + 4 = 25\,734 \\
 &= 2 \cdot 10^4 + 5 \cdot 10^3 + 7 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 = 25\,734
 \end{aligned}$$

– podobně můžeme „vyčíslit“ (převést do desítkové soustavy) i čísla z ostatních soustav; máme-li číslo ve dvojkové soustavě, tak se používají mocniny dvou:

1	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

$$\begin{aligned}
 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
 &= 2^7 + 2^5 + 2^2 + 2^1 + 2^0 \\
 &= 128 + 32 + 4 + 2 + 1 = 167
 \end{aligned}$$

– podobně jako v desítkové soustavě říkáme, že máme číslici v řádu jednotek, desítek, stovek, tisíců atd., tak ve dvojkové soustavě říkáme, že máme číslici na **n-tém bitu**; např. číslo 1010 0111 má jedničku v 0., 1., 2., 5. a 7. bitu

číslo ve dvojkové soustavě:	1	0	1	0	0	1	1	1
bit:	7.	6.	5.	4.	3.	2.	1.	0.
mocnina:	2^7		2^5			2^2	2^1	2^0
vypočtená mocnina:	128		32			4	2	1
součet:	128 + 32 + 4 + 2 + 1							
číslo v desítkové soustavě:	167							

– pro snadný převod je dobré si zapamatovat tabulku základních mocnin:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

– když převádíme, tak si v hlavě odpočítáváme 1, 2, 4, 8, 16, 32, atd., ale píšeme jen ty čísla, které mají jedničku v odpovídajícím bitu:

$$\begin{aligned}
 0001\ 1010 &\equiv 2 + 8 + 16 = 26 \\
 1000\ 0111 &\equiv 1 + 2 + 4 + 128 = 135 \\
 0110\ 0100 &\equiv 4 + 32 + 64 = 100
 \end{aligned}$$

Převod dvojkových čísel skládajících se pouze z jedniček na desítková

- zapíšeme si číslo o jedna větší, které je ve dvojkové soustavě ve tvaru jedna jednička a za ní samé nuly; toto číslo převedeme do desítkové soustavy jako 2^n a konečný výsledek o jedničku snížíme, takže dostaneme $2^n - 1$, např.:

$$11111 = 100000 - 1 \hat{=} 2^5 - 1 = 32 - 1 = 31$$

$$11111111 = 10000000 - 1 \hat{=} 2^8 - 1 = 256 - 1 = 255$$

Převod malých čísel z desítkové soustavy do dvojkové

- pro počítání z paměti s malými čísly (typicky do 8 bitů) lze použít trik, ve kterém rozložíme číslo v desítkové soustavě na součet druhých mocnin tak, že se každá mocnina dvojky vyskytuje v součtu pouze jednou:

$$173 = 128 + 32 + 8 + 4 + 1$$

- příklad jsem si v hlavě počítal následovně:

$$173 = 128 + \text{zbytek } 45$$

$$45 = 32 + \text{zbytek } 13$$

$$13 = 8 + \text{zbytek } 5$$

$$5 = 4 + \text{zbytek } 1$$

- když mám číslo rozložené, tak zapisuji jedničku pouze na těch bitech, které mají svou mocninu v součtu; bity, které nemají svou mocninu v součtu zapisuji jako nulu:

$$173 = 128 + 32 + 8 + 4 + 1 \equiv 1010\ 1101 \text{ (128 ano, 64 ne, 32 ano, 16 ne, 8 ano, 4 ano, 2 ne, 1 ano)}$$

- další příklady:

$$25 = 16 + 8 + 1 \equiv 11001$$

$$130 = 128 + 2 \equiv 10000010$$

$$62 = 32 + 16 + 8 + 4 + 2 = 111110$$

Převod velkých čísel z desítkové soustavy do dvojkové – metoda I

- všimněme si, jak by šlo z čísla 25 734 výpočtem získat cifry 2, 5, 7, 3 a 4:

$$25\ 723 / 10 = 2\ 572 \text{ zbytek } 3$$

$$2\ 572 / 10 = 257 \text{ zbytek } 2$$

$$257 / 10 = 25 \text{ zbytek } 7$$

$$25 / 10 = 2 \text{ zbytek } 5$$

$$2 / 10 = 0 \text{ zbytek } 2$$

- výpočet děláme tak dlouho, dokud nevyjde výsledek dělení nula
- sepíšeme zbytky pozpátku od posledního k prvnímu: 25 723

- totéž teď zkusíme pro číslo 167 (1010 0111):

$$167 / 2 = 83 \text{ zbytek } 1$$

$$83 / 2 = 41 \text{ zbytek } 1$$

$$41 / 2 = 20 \text{ zbytek } 1$$

$$20 / 2 = 10 \text{ zbytek } 0$$

$$10 / 2 = 5 \text{ zbytek } 0$$

$$5 / 2 = 2 \text{ zbytek } 1$$

$$2 / 2 = 1 \text{ zbytek } 0$$

$$1 / 2 = 0 \text{ zbytek } 1$$

- sepsáním zbytků pozpátku opravdu dostáváme 1010 0111
- nevýhodou této metody je, že číslice vycházejí v opačném pořadí, než se pak zapisují; následující metoda tuto nevýhodu nemá

Převod velkých čísel z desítkové soustavy do dvojkové – metoda II

- zkusíme opět z čísla 25 734 získat cifry 2, 5, 7, 3 a 4, ale tentokrát trochu jinak:

$$25\ 735 / 10\ 000 = 2 \text{ zbytek } 5\ 735$$

$$5\ 735 / 1\ 000 = 5 \text{ zbytek } 735$$

$$735 / 100 = 7 \text{ zbytek } 35$$

$$35 / 10 = 3 \text{ zbytek } 5$$

$$5 / 1 = 5 \text{ zbytek } 0$$
- výpočet končí, když je zbytek 0
- sepsáním výsledků dělení od prvního k poslednímu dostáváme 25 735
- nevýhodou této metody je, že musíme na začátku určit, kterým číslem začneme dělit; nevadí ale, pokud zvolíme číslo příliš velké, protože to bude mít za následek jen to, že nám vyjdou nuly před číslem, které stačí ve výpise vynechat

$$25\ 735 / 1\ 000\ 000 = 0 \text{ zbytek } 25\ 735$$

$$25\ 735 / 100\ 000 = 0 \text{ zbytek } 25\ 735$$

$$25\ 735 / 10\ 000 = 2 \text{ zbytek } 5\ 735$$

$$5\ 735 / 1\ 000 = 5 \text{ zbytek } 735$$

...

Šestnáctková soustava

- máme číslice 0, 1, ..., 9, A, B, C, D, E a F (celkem 16 číslic); písmeno A odpovídá desítkě, písmeno B jedenáctce atd.; pro rychlé převádění si stačí uvědomit, že písmena jsou číslována od nuly, tj. A je pro 1-0, B je pro 1-1, C je pro 1-2 atd.
- v programovacích jazycích se obvykle před šestnáctkové číslo píše předpona 0x, např. 0x3F; pokud bychom měli číslo v osmičkové soustavě, tak se obvykle píše s nulou na začátku (proto nesmíme u běžných desítkových čísel psát před ně nuly!); např.:

$$52 = \text{číslo } 52 \text{ v desítkové soustavě}$$

$$052 = \text{číslo } 42 \text{ v osmičkové soustavě}$$

$$0x52 = \text{číslo } 82 \text{ v šestnáctkové soustavě}$$
- pro převod čísla z šestnáctkové soustavy do desítkové používáme stejný algoritmus, jako u dvojkové soustavy:

$$0x5C0B \hat{=} 11 \cdot 16^0 + 0 \cdot 16^1 + 12 \cdot 16^2 + 5 \cdot 16^3 = 23\ 563$$
- podobně můžeme stejně jako u dvojkové pomocí dělení převádět z desítkové do šestnáctkové:

$$23\ 563 / 16 = 1\ 472 \text{ zbytek } 11 = \mathbf{B}$$

$$1\ 472 / 16 = 92 \text{ zbytek } 0$$

$$92 / 16 = 5 \text{ zbytek } 12 = \mathbf{C}$$

$$5 / 16 = 0 \text{ zbytek } 5$$
- nejčastěji se ale převádí mezi šestnáctkovou a dvojkovou soustavou; jak to efektivně dělat ukáže následující kapitola

DEC	BIN	HEX
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Převod mezi šestnáctkovou a dvojkovou soustavou

- chceme-li převést číslo z dvojkové soustavy do šestnáctkové, stačí si ho rozdělit po čtveřicích a každou čtveřici převést samostatně do desítkové soustavy (0 až 15) a zapsat jí pomocí jedné šestnáctkové číslice (0 až F):

Dvojková soustava:	0101	1100	0000	1011
Desítková soustava:	1+4	4+8	0	1+2+8
	5	12	0	11
Šestnáctková soustava:	5	C	0	B

- dostáváme, že $0101\ 1100\ 0000\ 1011 \hat{=} 0x5C0B$
- skutečnost, že šestnáctkové číslice 0 až F lze reprezentovat dvojkově jako 0000 až 1111 je důvod, proč se šestnáctková soustava používá
- opačná konverze je stejně jednoduchá:

Šestnáctková soustava:	5	C	0	B
Desítková soustava:	5	12	0	11
	4+1	8+4	0	8+2+1
Dvojková soustava:	0101	1100	0000	1011

Shrnutí

Základ	Česká název	Počestěný anglický název	Značka	Číslice	Prefix
2	Dvojková	Binární	BIN	0, 1	0b (nefunguje v C)
8	Osmičková		OCT	0 – 7	0
10	Desítková		DEC	0 – 9	žádný
16	Šestnáctková	Hexadecimální	HEX	0 – 9, A – F	0x

20. Bajty

- anglicky **byte**, česky bajt
- značí se B, např. $1024\ B = 1024$ bajtů (bit se oproti tomu značí malým b, např. $8\ b = 8$ bitů)
- bajtem rozumíme osmici bitů; protože v raných začátcích počítačů mohl mít na některých strojích bajt i jiný počet bitů než osm, tak se v některých starších specifikacích (např. internetových protokolů) uvádí místo slova bajt pojem **oktet**
- řekneme-li např. že máme 2 bajtové číslo, myslíme tím, že máme číslo, které má 16 bitů nebo-li 16 číslic ve dvojkové soustavě
- 0. bitu v bajtu (bitu nejvíce vpravo) říkáme nejméně významný bit, anglicky **least significant bit** (LSB)
- 7. bitu v bajtu (bitu nejvíce vlevo) říkáme nejvýznamnější bit, anglicky **most significant bit** (MSB)
- příklad pro číslo 10100111:

7.	4.	3.	0.
1	0	1	0
0	1	1	1
MSB		LSB	

Převody jednotek

- kvůli efektivnímu ukládání bajtů v paměti počítače se typicky paměť rozděluje na bloky o velikostech mocnin čísla dvě; kvůli tomuto faktu se zavádí trochu odlišný způsob chápání předpon jednotek
- ze soustavy SI jsme zvyklí na to, že předpona kilo znamená 1 000, takže např. $4\ km = 4\ 000\ m$, $1\ kN = 1\ 000\ N$ apod., ale 1 000 není mocnina dvou; nejbližší mocnina dvou je $2^{10} = 1024$, takže na počítači se obvykle rozumí, že $1\ kB = 1024\ B$; aby se tato rozdílná interpretace předpony kilo odlišila, tak se často píše s velkým k: $1\ KB = 1024\ B$; problém je v tom, že další předpony jako mega, giga, tera atd. se už velkým písmenem píší, takže např. ze samotného zápisu MB nejde vyčíst, o kterou z následujících dvou situací se jedná a musíme to určit z kontextu:
 - 1 MB = 1 000 kB = 1000 0000 B nebo
 - 1 MB = 1024 KB = 1 048 576 B

- kvůli nejednoznačnosti byla formálně zavedena soustava předpon, která označuje fakt, že se nejedná o násobky 1 000, ale násobky 1 024, ale v praxi se příliš nepoužívá:

Desítková soustava			Dvojková soustava		
k	kilo	$10^3 = 1000$	Ki	kibi	$2^{10} = 1024$
M	mega	$10^6 = \text{milión}$	Mi	mebi	$2^{20} = 1024 \text{ Ki}$
G	giga	$10^9 = \text{miliarda}$	Gi	gibi	$2^{30} = 1024 \text{ Mi}$
T	tera	10^{12}	Ti	tebi	$2^{40} = 1024 \text{ Gi}$

- pak lze jednoznačně psát, že např. 1 MiB (mebibajt) = 1 024 KiB (kibibajtů) = 1 048 576 B (bajtů)
- uvedený systém se používá i pro bity, takže typicky např. 1 Mb (megabit) = 1 024 Kb (kilobitů) = 1 048 576 b (bitů)

8. lekce

21. Rozsah neznaménkových celých čísel

- máme několikabitové číslo a chceme zjistit, jaká všechna čísla v desítkové soustavě může reprezentovat
- pro jednoduchost si nejdřív vše ukážeme např. na 5 bitovém čísle:

Počet bitů:	5
Rozsah čísel binárně:	00000 až 11111
Největší číslo desítkově:	$11111 = 100000 - 1 \hat{=} 2^5 - 1 = 32 - 1 = 31$ (tento trik si zapamatuj)
Rozsah čísel desítkově:	0 až 31, tj. 0 až $2^5 - 1$
Celkem čísel:	32, tj. 2^5

- podobně lze určit rozsahy několikabajtových čísel:

Bajtů	Bitů	Hodnot	Rozsah	Vyčíslený rozsah
1 B	8 b	$2^8 = 256$	0 až $2^8 - 1$	0 – 255
2 B	16 b	$2^{16} = 65\,536$	0 až $2^{16} - 1$	0 – 65 535
4 B	32 b	$2^{32} = 4\,294\,967\,296 = 4 \text{ Gi}$	0 až $2^{32} - 1$	0 – cca $4.3 \cdot 10^9$
8 B	64 b	$2^{64} = 18\,446\,744\,073\,709\,551\,616 \hat{=} 1.8 \cdot 10^{19}$	0 až $2^{64} - 1$	0 – cca $1.8 \cdot 10^{19}$

22. Celé číslo se znaménkem

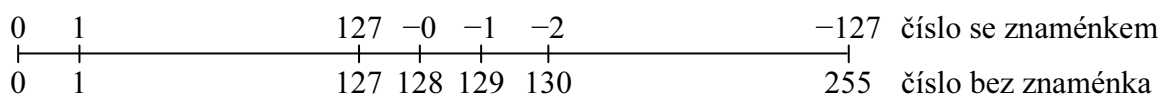
Kódování znaménka pomocí nejvýznamnějšího bitu

- znaménko ukládáme do nejvýznamnějšího bitu čísla (0 = plus, 1 = mínus)

Příklad 1

Máme 1 bajtovou proměnnou, která reprezentuje celé číslo se znaménkem uloženým v nejvýznamnějším bitu. Jaká všechna čísla může tato proměnná reprezentovat?

Kladná čísla:	0000 0000 – 0111 1111 (0 – 127)	7. 6.	0.
Záporná čísla:	$-0 \hat{=} 1000\,0000 \hat{=} 128$	Z číslo bez znaménka	
	$-1 \hat{=} 1000\,0001 \hat{=} 129$		
	$-2 \hat{=} 1000\,0010 \hat{=} 130$		
...	$-127 \hat{=} 1111\,1111 \hat{=} 255$		



Příklad 2

Máme 6-bitovou proměnnou. Jaké číslo do ní musíme uložit, aby reprezentovala číslo -11 se znaménkem uloženým v nejvýznamnějším bitu?

6-bitová proměnná: ??? ???
Číslo 11: 001 011 (8 + 2 + 1)
Znaménko: 100 000
Číslo -11: 101 011 $\hat{=} 1 + 2 + 8 + 32 = 43$

Příklad 3

Jaké číslo bude reprezentovat 2-bajtová proměnná, uložíme-li do ní hodnotu 0xC0A7?

Číslo 0xC0A7 ve dvojkové soustavě: C:0:A:7 $\hat{=} 12:0:10:7 \hat{=} 8+4:0:8+2:4+2+1$
 1100 0000 1010 0111
Znaménko: mínus (15. bit je 1)
Číslo bez znaménka: 0100 0000 1010 0111 $\hat{=} 2^{14} + 2^7 + 2^5 + 4 + 2 + 1 \hat{=} 16\ 551$
Číslo reprezentované proměnnou: -16 551

Výhody této metody kódování čísel:

- číselný obor je symetrický (má stejný počet kladných i záporných čísel; každé číslo má opačné číslo reprezentovatelné stejnou proměnnou)
- snadno se zjišťuje znaménko čísla (stačí zjistit hodnotu nejvýznamnějšího bitu)
- snadno se zjistí absolutní hodnota čísla (vynulováním nejvýznamnějšího bitu)

Nevýhody:

- dvě nuly (kladná a záporná)
- komplikovanější aritmetika s těmito čísly: aby se dala čísla např. sčítat, tak se nejdřív musí odstranit znaménko, pak podle původních znamének a velikosti čísel bez znamének rozhodnout, zda budeme čísla sčítat nebo odčítat a udělat to a nakonec do výsledku opět znaménko doplnit; celá operace tak bude několikrát pomalejší než obyčejné sečtení nebo odečtení čísel bez znaménka

Kódování čísel se znaménkem pomocí dvojkového doplňku