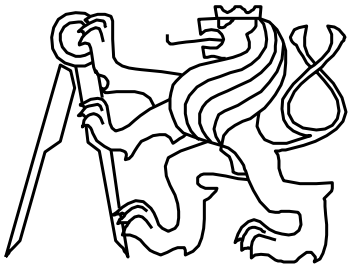
Příprava studijního programu Informatika je podporována projektem financovaným z Evropského sociálního fondu a rozpočtu hlavního města Prahy.
Praha & EU: Investujeme do vaší budoucnosti

Abstraktní datový typ



BI-PA1 Programování a algoritmizace 1, ZS 2012-2013
Katedra teoretické informatiky
© Miroslav Balík
Fakulta informačních technologií
České vysoké učení technické

Obsah

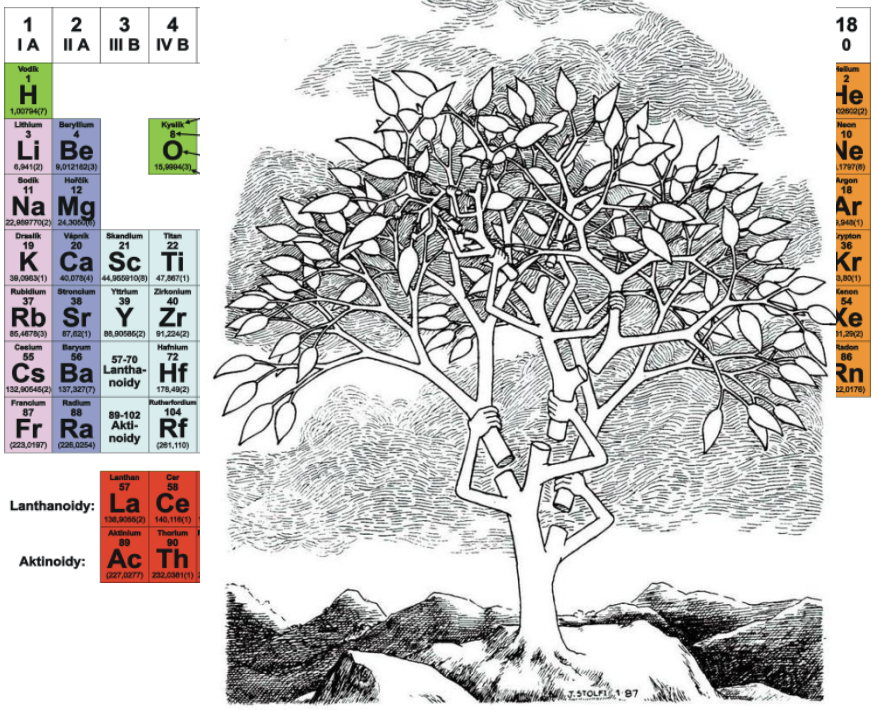
- Téma
  - zopakování datových typů
  - abstraktní datové typy
  - příklad: typ Complex
  - modulární realizace typu Compex
  - sdílení proměnných mezi moduly
  - více o předprocesoru
  - a ještě něco k popisům typu

Datové struktury

- Klasická kniha o programování prof. Wirtha má název

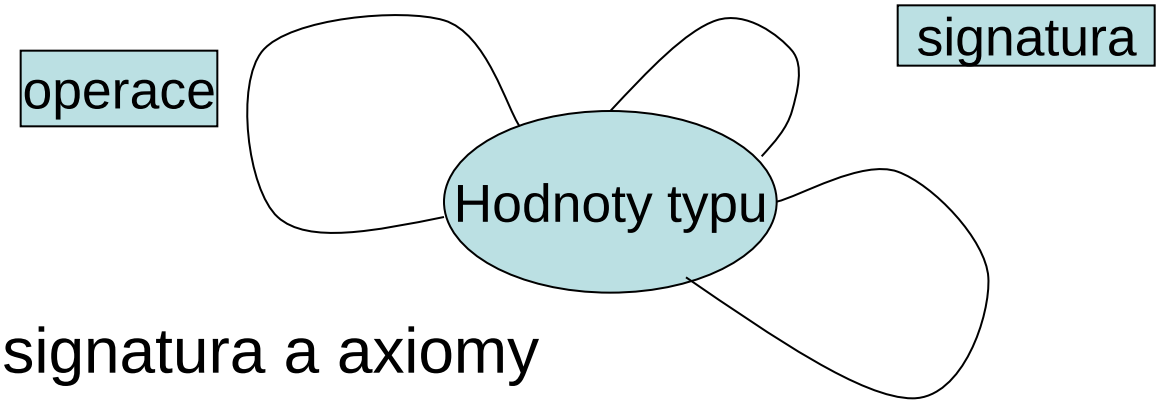
  Algorithms + Data structures = Programs

- Datová struktura (typ) = množina dat + operace s těmito daty
- Abstraktní datový typ - formálně definuje data a operace s nimi, např.
  - Fronta (Queue)
  - Zásobník (Stack)
  - Pole (Array)
  - Tabulka (Table)
  - Seznam (List)
  - Strom (Tree)
  - Množina (Set)



Abstraktní datový typ

- je množina *druhů dat* (hodnot) a příslušných *operací*, které jsou přesně specifikovány, a to ***nezávisle na konkrétní implementaci***.

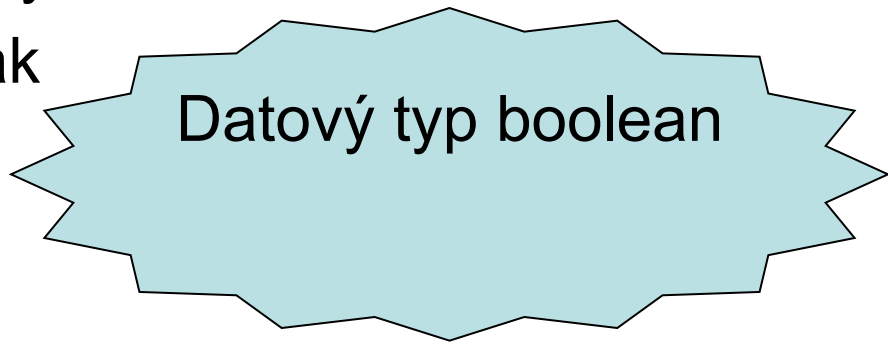


- Definovat lze
  - Matematicky – signatura a axiomy
  - Jako rozhraní (knihovna v C) s popisem operací
    - Rozhraní poskytuje
      - Funkce pro vytvoření a práci s daným ADT
      - operace, které akceptují odkaz jako argument a které mají přesně definovaný účinek na data

Příklad1: ADT - popis matematicky I

- Syntaxe popisuje, jak správně vytvořit logický výraz:
  1. true, false jsou logické výrazy,
  2. if x,y jsou logické výrazy, pak
     i. ! (x), negace
     ii. (x & y), logický součin, and
     iii. (x | y), logický součet, or
     iv. (x==y), (x!=y), relační operátory

     jsou logické výrazy. Pokud nechceme u každé operace psát závorky, musíme definovat priority operátorů.



- Sémantika popisuje význam jednotlivých operací. Lze definovat pomocí axiomů:

  1)!(true) = false 2) !(false) = true

  3) x & false = false 4) x & true = x

  5) x & y = y & x 6) x | true = true

  7) x | false = x 8) x | y = y | x

Příklad1: ADT - popis matematicky II

Datový typ boolean

– Sémantika, pokračování

9) true == true = true 10) true == false = false

11) false == false = true 11) false == true = false

– nebo lépe(vhodnější pro úpravy):

9) true == x = x 10) false == x = !x

11) x==y = y==x

– … dále by následovala sémantika pro !=

Datové typy

- V programovacích jazycích jako je C je každý datový objekt (proměnná nebo konstanta) nějakého datového typu
- Připomeňme, že datový typ:
  - specifikuje množinu možných hodnot
  - specifikuje množinu operací s hodnotami
- Datové typy se dělí na:
  - jednoduché (hodnoty jsou z hlediska operací atomické)
  - strukturované (hodnoty se skládají ze složek)
  - ukazatele (hodnotami jsou adresy)
- V jazyku C je:
  - definováno několik standardních jednoduchých datových typů, které mají svá jména (dána klíčovými slovy), např. *int*, *float*, *char* atd.
  - stanoveno, jak deklarovat datový objekt typu pole nebo struktura, a jaké operace jsou pro takové datové objekty povoleny (jaké?)
  - stanoveno, jak deklarovat datový objekt typu ukazatel, a jaké operace jsou pro takové operace povoleny (jaké?)

Abstraktní datové typy

- Příklad: potřebujeme napsat program, který pracuje s komplexními čísly, se kterými se budou provádět operace sčítání, odčítání, absolutní hodnota a výpis.

- Komplexní číslo je reprezentovatelná dvojicí reálných čísel a takový typ s požadovanými operacemi v jazyku C není.

- Zavedeme identifikátor typu Complex a požadované operace budeme realizovat pomocí funkcí. Pak se rozhodneme, zda typ budeme implementovat pomocí pole nebo struktury.

Abstraktní datové typy

v deklaracích funkcí nemusí být uvedena jména parametrů

- typedef ... Complex;

- Complex plus(Complex, Complex);
- Complex minus(Complex, Complex);
- float absC(Complex);
- void printC(Complex);

- Zavedli jsme abstraktní datový typ Complex

Použití ADT - prog13-prikl1.c

```
int main(void) {
    Complex x = {1,1}, y = {2,2}, z;
    float a;
    printf("x="); printC(x); printf("\n");
    printf("y="); printC(y); printf("\n");
    z = plus(x, y);
    printf("x+y="); printC(z); printf("\n");
    z = minus(x, y);
    printf("x-y="); printC(z); printf("\n");
    a = absC(x);
    printf("abs(x)=%f\n", a);
    return 0;
}
```

Implementace abstraktního datového typu

- V jazyku C nejčastěji pomocí struktury
- Implementace typu Complex:

```
typedef struct {
    float re;
    float im;
} Complex;
```

- Otázka: proč jsme typ Complex implementovali pomocí struktury a ne pomocí pole?

Operace s typem Complex

```
Complex plus(Complex x, Complex y) {
    Complex res;
    res.re = x.re+y.re;
    res.im = x.im+y.im;
    return res;
}

Complex minus(Complex x, Complex y) {
    Complex res;
    res.re = x.re-y.re;
    res.im = x.im-y.im;
    return res;
}
```

Operace s typem Complex

```
float absC(Complex x) {
	return sqrt(x.re*x.re+x.im*x.im);
}

void printC(Complex x) {
	printf("(%f,%f)", x.re, x.im);
}
```

Moduly

- Složitější programy je obvykle třeba rozdělit do více souborů
- Modulem se v programování rozumí část programu, která poskytuje určité prostředky ostatním částem programu
- Modul se skládá ze dvou částí:
  - specifikační části, ve které jsou deklarovány prostředky, které modul poskytuje (např. datový typ a procedury a funkce realizující operace s datovým typem)
  - implementační části, ve které jsou poskytované prostředky implementovány

Moduly

- V některých programovacích jazycích je modul zaveden jako programová jednotka (např. *units* Object Pascalu firmy Borland v systému Delphi)

- V jazyku C modul realizujeme dvojicí souborů:
  - specifikačním souborem typu .h, ve kterém jsou deklarace typů a funkcí
  - implementačním souborem typu .c, ve kterém jsou definice funkcí

Modul pro typ Complex

- Specifikační soubor:

```
/* complex.h */
typedef struct {
        float re;
        float im;
} Complex;

Complex plus(Complex, Complex);

Complex minus(Complex, Complex);

float absC(Complex);

void printC(Complex);
```

Modul pro typ Complex

- Implementační soubor:

```
/* complex.c */

#include "complex.h"
#include <stdio.h>
#include <math.h>

Complex plus(Complex x, Complex y) {
        Complex res;
        res.re = x.re + y.re;
        res.im = x.im + y.im;
        return res;
}
...
```

vložení specifikačního souboru

Modul pro typ Complex

- Implementační soubor, 2.část:

```
Complex minus(Complex x, Complex y) {
        Complex res;
        res.re = x.re - y.re;
        res.im = x.im - y.im;
        return res;
}

float absC(Complex x) {
        return sqrt(x.re * x.re + x.im * x.im);
}

void printC(Complex x) {
        printf("(%f,%f)", x.re, x.im);
}
```

Modul pro typ Complex - použití

```
/* main.c */
#include "complex.h"
#include <stdio.h>
#include <stdlib.h>
/* hlavni funkce */
int main(void) {
    Complex x = {1,1}, y = {2,2}, z;
    float a;
    printf("x="); printC(x); printf("\n");
    printf("y="); printC(y); printf("\n");
    z = plus(x, y);
    printf("x+y="); printC(z); printf("\n");
    z = minus(x, y);
    printf("x-y="); printC(z); printf("\n");
    a = absC(x); printf("abs(x)=%f\n", a);
    return 0;
}
```

vložení specifikačního souboru

Projekt v NetBeans, Dev-C++

- V prostředí IDE je třeba program složený z několika souborů vyvíjet v rámci projektu
- Vytvoření nového projektu:
  - v menu vybrat *Soubor/Nový/Projekt*
  - v okně *Nový projekt* vybrat *Console Application* nebo *Empty Project*, zaškrtnout *C Projekt* a zadat *Jméno* projektu, pak *OK*
  - v okně *Create new project* zadat jméno souboru .dev, ve kterém bude popis projektu, a adresář, do kterého se soubor uloží

- Do projektu lze vkládat nové zdrojové soubory nebo již existující zdrojové soubory

Projekt v NetBeans, Dev-C++

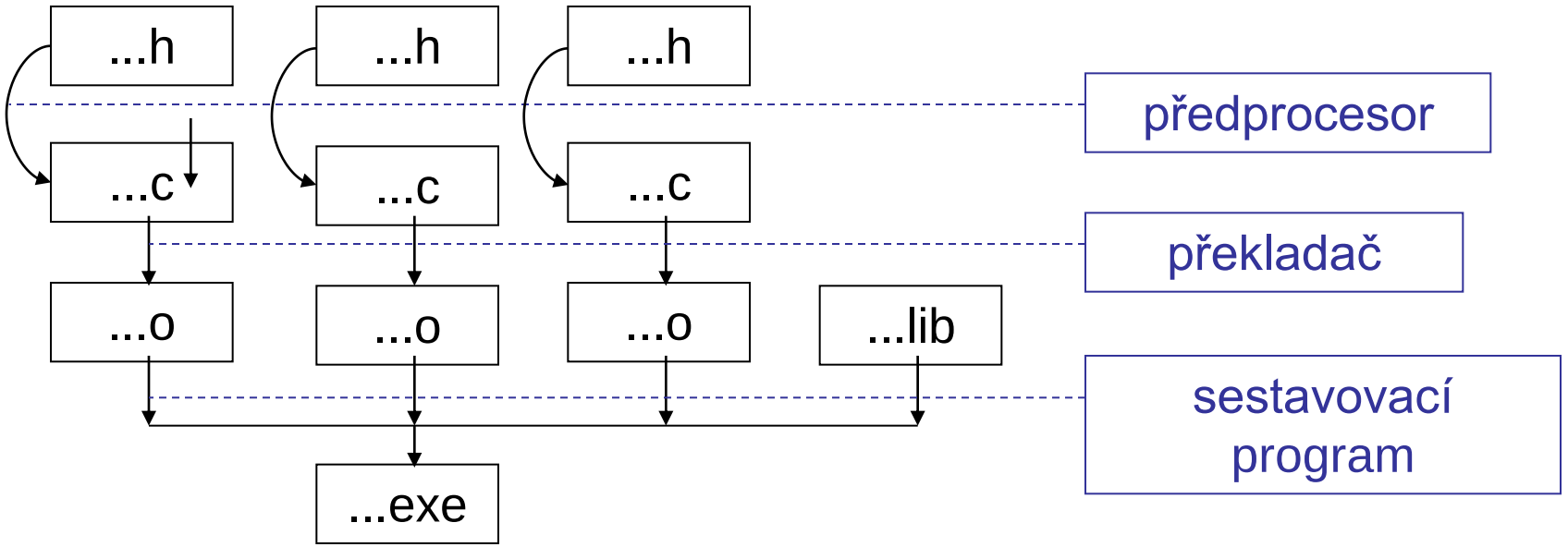
- Cílový program vytvoříme funkcí *Spustit/Překompilovat vše*
  - každý .c soubor se přeloží do .o souboru, který se uloží do adresáře, v němž je .c soubor
  - z .o souborů (a z knihoven) se sestaví cílový (spustitelný) program .exe, který se uloží do adresáře, v němž je soubor projektu .dev

- Příklad: vytvořme nový projekt a vložme do něj soubory *main.c*, *complex.c* a *complex.h* z adresáře *prednasky\priklady\p13\complex*

Překlad zdrojových souborů

- Každý zdrojový program typu .c je nejprve zpracován předprocesorem (definice maker, náhrada maker, vložení souborů atd.) a pak je překladačem přeložen do souboru .o, který je tvořen posloupností strojových instrukcí s relativními adresami operandů (a dalšími informacemi, které jsou potřeba pro následující sestavení)

- Potom je ze souborů .o a z knihovních souborů sestavovacím programem (linker) sestaven cílový program .exe



Sdílení proměnných mezi moduly

- Proměnná deklarovaná na úrovni souboru může být použita v jiném souboru
- Jinými slovy: modul v jazyku C může proměnnou v něm deklarovanou poskytnout pro použití v jiných modulech
- Příklad: modulárně napíšeme program, který:
  - přečte počet prvků pole, dynamicky ho vytvoří a pak přečte prvky pole
  - pole vypíše
  - pole vzestupně seřadí
  - seřazené pole vypíše
- Program bude tvořen:
  - soubory *pole.h* a *pole.c*, které budou tvořit modul s proměnnými *pole* (ukazatel na dynamicky vytvořené pole) a *pocet* (počet prvků pole) a funkcemi pro čtení pole a výpis pole
  - soubory *razeni.h* a *razeni.c*, které budou tvořit modul pro seřazení pole
  - souborem *main.c,* který bude obsahovat hlavní funkci
  - soubory jsou v adresáři *prednasky\priklady\pole*

Příklad se sdílenými proměnnými

soubor s hlavní funkcí

```
/* main.c */
#include "pole.h"
#include "razeni.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    ctiPole();
    printf("zadane pole\n"); vypisPole();
    seradPole();
    printf("serazene pole\n"); vypisPole();
    return 0;
}
```

ve specifikačním souboru se proměnné deklarují jako *extern*

```
/* pole.h */
extern int *pole, pocet;
void ctiPole(void);
void vypisPole(void);
```

specifikační soubor modulu *pole*

```
/* razeni.h */
void seradPole(void);
```

specifikační soubor modulu *razeni*

Příklad se sdílenými proměnnými

```c
/* pole.c */
#include "pole.h"
#include <stdio.h>

int *pole, pocet;

void ctiPole(void) {
        int i;
        printf("zadej pocet prvku pole: ");
        scanf("%d", &pocet);
        pole = (int*) malloc(pocet*sizeof(int));
        printf("zadej %d celych cisel\n", pocet);
        for (i = 0; i < pocet; i++) scanf("%d", &pole[i]);
}

void vypisPole(void) {
        int i;
        for (i=0; i<pocet; i++) printf("%d\n", pole[i]);
}
```

```c
/* pole.h */
extern int *pole, pocet;
void ctiPole(void);
void vypisPole(void);
```

Příklad se sdílenými proměnnými

```
/* razeni.c */
#include "razeni.h"
#include "pole.h"

void seradPole(void) {
    int i, imin, min, j, n = pocet-1;
    for (i = 0; i < n; i++) {
        imin = i; min = pole[i];
        for (j = i+1; j <= n; j++)
            if (pole[j]<min) {
                imin = j; min = pole[j];
            }
        if (imin != i) {
            pole[imin] = pole[i]; pole[i] = min;
        }
    }
}
```

Shrnutí .h souborů

- V souborech typu .h (header soubory, hlavičkové soubory) uvádíme:
  - deklarace typů
  - deklarace funkcí
  - *extern* deklarace proměnných

- Do souborů typu .h nikdy nepíšeme definice funkcí

- Když do .h souboru vložíme definici funkce a tento soubor vložíme pomocí direktivy *include* do několika souborů, nastane chyba při sestavení programu (*prednasky\priklady\p13\funkce*)

Ještě něco k předprocesoru

- Řádky začínající znakem # jsou direktivy předprocesoru
- Pomocí direktiv se zajišťuje:
  - vkládání jiných souborů

```
#include <stdio.h>
#include "stack.h”
#include SOUBOR
```

  - definice maker

```
#define _STACK_
#define TRUE 1
#define MAX(x,y) ((x)>(y)?(x):(y))
```

  - podmíněný překlad

```
#ifdef _DEBUG_
        …
#else
        …
#endif
```

..... a další

Makra bez parametrů:

#define ident nahrazující-text

- preprocesor v dalším textu nahradí každý výskyt identifikátoru makra nahrazujícím textem
- končí-li řádek nahrazujícího textu znakem \, zahrne se do nahrazujícího textu i další řádek
- je-li v nahrazujícím textu nalezeno makro, znovu se rozvine
- jména maker se nehledají v komentářích, řetězcích a mezi znaky < a > v direktivě *#include*
- makro platí až do konce souboru nebo do výskytu direktivy

#undef ident

- Standardní makra:
  - __DATE__ datum začátku zpracování souboru
  - __TIME__ čas začátku zpracování
  - __FILE__ jméno zpracovávaného souboru

atd. + implementačně závislá makra

Makra s parametry

```
#define SQR(x) ((x)*(x))
#define MAX(x,y) ((x)>(y)?(x):(y))
```

- V místě výskytu musí být v závorkách uveden odpovídající počet skutečných parametrů, které se textově dosadí za parametry v nahrazujícím textu

zdrojový text	výsledný text
`a = SQR(b)`	`a = ((b)*(b))`
`a = SQR(a+1)`	`a = ((a+1)*(a+1))`
`a = MAX(b,c+1);`	`a = ((b)>(c+1)?(a):(c+1));`

Makra s parametry a operátory ## a

- V nahrazujícím textu makra lze použít operátory ## a #:

```
include<stdio.h>

// x##y slepí elementy x a y
// #x k x přilepí uvozovky
#define DVE(x) x##1,x##2
#define PRINT(x) printf(#x"=%d\n",x)

int main(){
    int DVE(a);          //int a1,a2;
    a1 = 8;
    PRINT(a1);           //printf("a1""=%d",a1);
    PRINT(DVE(a));       //printf("DVE(a)""=%d",a1,a2);
    return 0;
}
```

```
a1 =8
DVE(a) =8
```

Podmíněný překlad

- Pro omezení úseků zdrojového textu, které mají být překládány podmíněně, slouží direktivy:
- V konstantním výrazu může být pro test, zda identifikátor je definován (direktivou *#define*) použit operátor *defined*
  - ```
    #ifdef  _DEBUG_
      printf((" a =%d\n", a);
    #endif
    ```

  je totéž co

  - ```
    #if defined _DEBUG_
     printf(" a=%d\n", a);
    #endif
    ```

Podmíněný překlad

- Pro omezení úseků zdrojového textu, které mají být překládány podmíněně, slouží direktivy:

```
#if konstantni-vyraz
...
#endif
```

```
#ifdef ident
...
#endif
```

```
#ifndef ident
...
#elif konstantni-vyraz
...
#else
...
#endif
```

A ještě něco k popisům typu

- Výčtový typ
- Typ *union*
- Struktura s bitovými poli
- Struktura s otevřeným polem

Popis výčtového typu

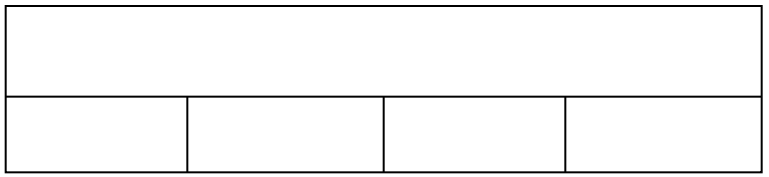
- Syntaxe: **enum** *značka* { *seznam literálů* }
- Příklad: **enum Color {Red, Blue, Green};**
- Literály jsou synonyma celočíselných hodnot
  - /* Red = 0, Blue = 1, Green = 2 */
- Literálu lze explicitně přiřadit hodnotu
  - **enum Masky { Nula, Jedna, Dva, Ctyri = 4, Osm = 8};**
  - /* Dva = 2, Ctyri = 4, Osm = 8 */
  - **enum Sign { Minus = -1, Zero, Plus};**
  - /* Minus = -1, Zero = 0, Plus = 1 */
  - **enum { E1, E2, E3 = 5, E4, E5 = E4 + 10, E6};**
  - /* E4 = 6, E5 = 16, E6 = 17 */
- Pro výčtové typy jsou definovány stejné operace, jako pro celočíselné typy
- Vnitřní reprezentace proměnných výčtových typů je stejná jako vnitřní reprezentace proměnných typu *int* (velikost 4B ve 32 bitovém prostředí)

Popis typu union

- Syntaxe stejná jako u struktury, místo *struct* je *union*
- Položky se nekladou za sebe, ale přes sebe

```
union {
        int cislo;
        unsigned char byty[4];
} x;
```

x



```
int main(){
        int i,cislo;
        unsigned char pole[4];
        x.cislo = 0x00112233;
        for (i=0; i<4; i++)
                printf("%x ", x.byty[i]);
        return 0;
}
```

vypíše 33 22 11 0

Bitová pole

- V popisu struktury lze pro položku typu *signed* nebo *unsigned* stanovit počet bitů vnitřní reprezentace

```
#include<stdio.h>
#pragma pack(1)
struct {
        unsigned p1:4;
        unsigned p2:4;
} bp;

int main(){
        printf("velikost promenne bp: %d\n", sizeof(bp));
        return 0;
}
```

Struktura s otevřeným polem

Poslední položkou struktury může být pole bez zadaného počtu prvků

```
#include <stdio.h>
#include<stdlib.h>
typedef struct { int pocet; int pole[]; } OtevrenePole;
OtevrenePole *vytvor(int pocet) {
    OtevrenePole *p;
    p = (OtevrenePole*) malloc(sizeof (*p) + pocet * sizeof (int));
    p->pocet = pocet;
    return p;
}
int main(void) {
    OtevrenePole *p = vytvor(10); int i;
    for (i = 0; i < p->pocet; p->pole[i++] = i);
    for (i = 0; i < p->pocet; printf("%d ", p->pole[i++])); printf("\n");
    free(p);
    return 0;
}
```

uvolní alokovanou paměť