INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

PROGRAMOVÁNÍ ŘÍDÍCÍCH SYSTÉMŮ

Pokročilejší programovací techniky

Ing. Ivo Špička, Ph.D.

Ostrava 2013

ISBN 978-80-248-3041-4

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

2 POKROČILEJŠÍ PROGRAMOVACÍ TECHNIKY

OBSAH KAPITOLY:

Co je a k čemu slouží ukazatel

Jaký je rozdíl mezi staticky a dynamicky alokovanou paměti

MOTIVACE:

Po prostudování této přednášky budete schopni popsat smysl použití databází a databázových systémů, identifikovat základní problémy, které databázový přístup řeší a definovat základní databázové pojmy.

CÍL:

Po prostudování tohoto odstavce budete umět:

- Pracovat s ukazateli
- Používat adresovou aritmetiku a operátor sizeof()
- Ukazatel na funkce
- Dynamickou alokaci paměti
- Struktury a výčtový typ

2.1 UKAZATELE

Mimo základní datové typy v jazyce c existují typy odvozené. Jedním z těchto typů jsou ukazatele. Ukazatel je v podstatě abstraktnější ztvárnění adresy. Jde o techniku, jak nepřímo přistupovat k proměnným a datům v paměti. V tomto článku se seznámíme s ukazateli, přetypováním ukazatelů, ukazatelům na funkce a ukazatelům na ukazatele. Dále bude vysvětlena adresová aritmetika a některé další detaily týkající se ukazatelů.

2.1.1 Motivace

Proč vůbec zavádět ukazatele, když doteď jsme se bez nich obešli. Ukazatele jsou potřebné ze dvou důvodů, O prvním důvodu jsme se zmínili, když jsme rozebírali způsob předávání parametrů funkcím. Druhým důvodem je použití dynamicky (tedy proměnně) vytvářených proměnných.

2.1.2 Deklarace ukazatele

Obecná deklarace ukazatele (pointeru) se od deklarace proměnné liší pouze použitím hvězdičky před jménem proměnné.

```
<domenovy_typ> *<jméno>;
```

Konkrétně pak například

```
int a = 3;
int *pa = &a;
```

Zde jsme definovali proměnnou a s hodnotou tři a poté jsme definovali ukazatel na typ int, do kterého jsme uložili adresu proměnné a. Všechna data leží někde v paměti, kterou si můžeme představit jako obrovské pole. Každá buňka tohoto pole (bajt) má svoji adresu, což je pořadové číslo buňky v paměti. Ukazatel tak není nic jiného než proměnná uchovávající toto číslo.

K získání adresy dat slouží operátor získání adresy (reference na proměnnou) &.

Pokud chceme přistupovat k datům, na která ukazuje ukazatel, musíme použít operátor dereference *

Výhoda tohoto přístupu spočívá v tom, že se můžeme odkazovat na velké množství dat jen s pomocí čísla uloženého na čtyřech byte, což je velikost proměnné typu ukazatel, a nemusíme data kopírovat, například když je potřebuje zpřístupniv jako parametr funkce.

Parametr nelze nikdy inicializovat přímo číselnou hodnotou.

```
pa = 2; // chyba
*pa = 2;
```

V první řádce našeho příkladu je sémantická chyba, kde místo hodnoty proměnné a přepisujeme adresu uchovanou v ukazateli.

2.1.3 Ukazatel bez doménového typu a neplatná adresa

Ukazatele mohou ukazovat na libovolná data. Existuje však speciální forma ukazatele, tzv. ukazatel bez doménového typu.

```
void *ukazatel;
```

Tento ukazatel může ukazovat na libovolná data a jakýkoliv ukazatel lze převést na ukazatel bez doménového typu. Takovýto ukazatel ale logicky nelze dereferencovat, neboť nevíme, na jaká data ukazuje, a tudíž nevíme, jak máme odkazovaná data reprezentovat. Před přístupem k datům tak ukazatel musíme explicitně přetypovat na ukazatel s konkrétním doménovým typem. Ukazatel bez doménového typu se hojně využívá v místech, která mají být dostatečně obecná a musí umět pracovat s různým typem dat.

2.1.4 Prázdný ukazatel

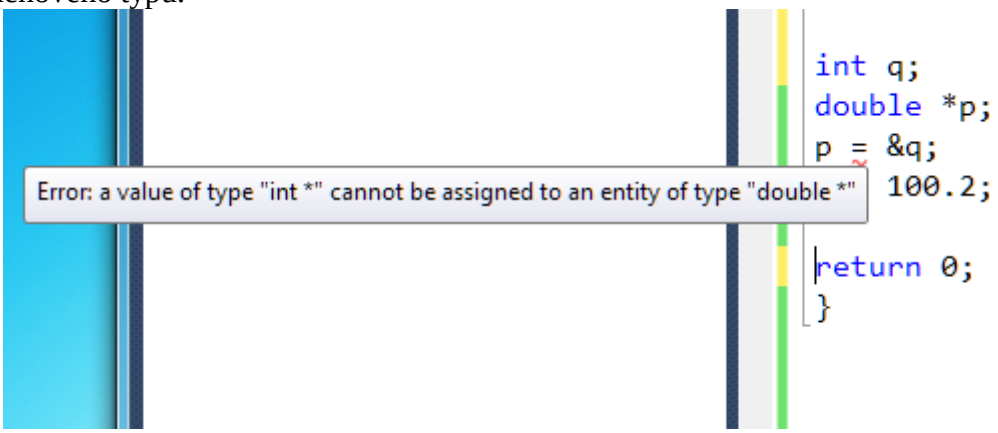
Ukazatel může také ukazovat "nikam". Jeho hodnota je potom dána standardním makrem NULL (obvykle je to nula). Ukazatel NULL nelze dereferencovat. Makro je standardně definováno takto:

```
#define NULL ((void*)0)
```

Jinými slovy neplatný ukazatel je na adrese nula. Proto můžeme testovat ukazatele na platnost v podmínkách jako každé jiné číslo. Ukazatel NULL se vyhodnotí jako nepravda. Snažte se dodržovat pravidlo, že pokud ukazatel nikam neukazuje, má mít hodnotu NULL.
Nyní si ukážeme, jak lze jednoduše udělat chybu.

```
int q;
double *p;
p = &q;
p = 100.2;
```

Ukazateli p je přiřazena adresa čísla int. Tato adresa je pak použita na levé straně přiřazovacího příkazu pro přiřazení hodnoty s pohyblivou řádovou čárkou. Jelikož je však číslo typu int obvykle kratší než double, způsobil by přiřazovací příkaz přepsání paměti sousedící s q. Proto lze přiřazovat ukazatele sobě navzájem, jen pokud jsou shodného doménového typu.



Další možný chybný zápis vzniká tehdy, pokud se snažíme použít ukazatel dříve než je do něj přiřazena adresa proměnné. V takovém případě program pravděpodobně zhavaruje. Je důležité si uvědomit, že deklarace ukazatele pouze vytvoří proměnnou schopnou uchovávat adresu paměti. Nedá ji však žádnou smysluplnou hodnotu.

```
int *p;
*p = 10; //chyba - ukazatel p na nic neukazuje!
```

Další chyba může vzniknout následujícím způsobem.

```
int q,r;
int *p;
p = &q;
r = *p;
```

Jaká bude hodnota proměnné r? V příkladu jsme nejprve deklarovali dvě proměnné, q a r. Deklarovali jsme ukazatel na int p. Inicializovali jsme ukazatel p adresou proměnné q. Zatím je vše v pořádku. Chybný je poslední řádek kódu. Dereferencovali jsme ukazatel p. To znamená, že pomocí ukazatele p nepřímo čteme obsah proměnné q. Obsah této proměnné, protože jsme proměnnou zatím nikde nepřiřadili hodnotu, neinicializovali ji, je zcela náhodný. Tato chyba vede k neočekávanému chování programu či jeho zhroucení.
Místa, na která se ukazatel odkazuje (jejichž adresu obsahuje) se mohou lišit. Takže můžeme například psát cykly, v nichž pomocí jediného ukazatele postupně přistupujeme ke všem prvkům pole. Další možnosti použití ukazatelů se otevírají při dynamickém přidělování paměti.

2.1.5 Konstantní ukazatel a ukazatel na konstantu

V deklaraci ukazatele lze přidat klíčové slovo const. Podle toho, kam ho vložíme, vytvoříme konstantní ukazatel, nebo ukazatel na konstantu.

```
const int *p; // ukazatel na konstantu
int *const p; // konstantní ukazate
```

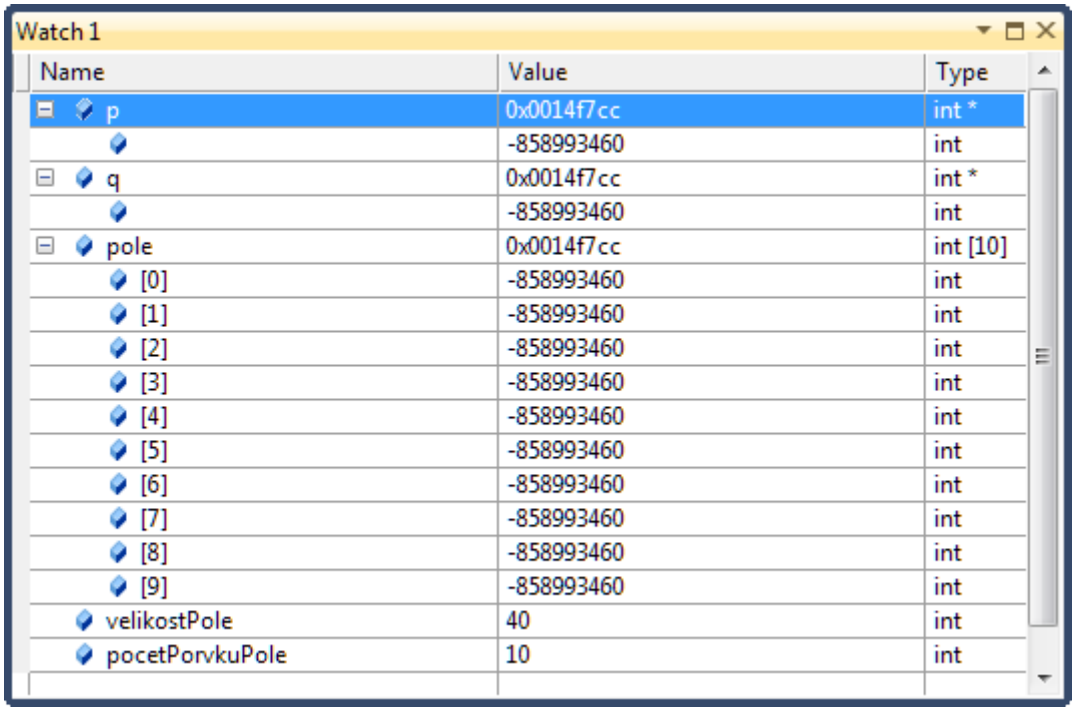
První řádek deklaruje ukazatel na konstantu. Takový ukazatel můžeme měnit, odkazovaná data jsou ale konstantní a tedy neměnná. Při zápisu do dereferencovaného ukazatele obdržíme chybové hlášení o přístupu k datům pouze pro čtení.
Druhá řádka deklaruje konstantní ukazatel. Takový ukazatel je pevně spojen s obsahem a jeho hodnotu nelze měnit. Lze ale měnit odkazovaná data.

2.1.6 Vztah pole a ukazatele

Pole a ukazatele v jazyce C spolu souvisí. Pole, jejich identifikátor bez operátoru indexování, představují totiž konstantní ukazatele na první prvek pole (s indexem nula). Toto chování se liší pouze při použití operátoru sizeof(), který vrátí velikost pole a nikoliv velikost ukazatele, a při získávání adresy pole, kde vrácený ukazatel představuje jiný typ než ukazatel na první prvek (číselná hodnota je ale stejná).
Pokud se pokusíme přiřadit ukazateli na typ p adresu pole, získanou operátorem získání adresy a identifikátorem pole, obdržíme chybové hlášení, že ukazateli na int nelze přiřadit ukazatel typu int(*)[10].


```
int main (int argc, char **argv)
{
int pole[10];
int velikostPole;
int pocetPorvkuPole;
int *p, *q;
p = pole;
// p = &pole; nelze
q = &pole[0];
velikostPole = sizeof(pole);
pocetPorvkuPole = sizeof(pole)/sizeof(int);
}
```

Hodnoty ukazatelů pole, p a q budou shodné.



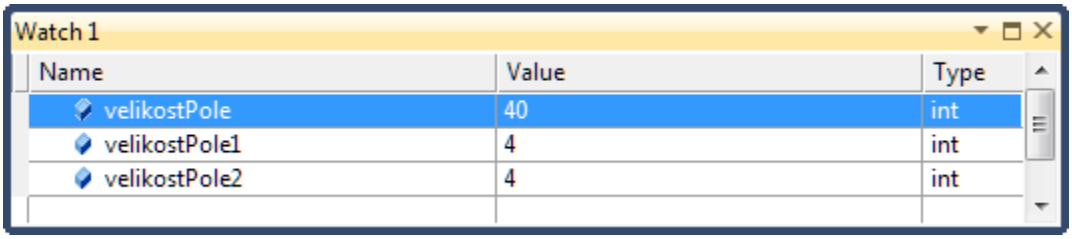
Proměnná velikostPole bude obsahovat hodnotu 40, což je výsledek výrazu ***sizeof***(pole) a jelikož velikost datového typu int je čtyři byte, pak počet prvků pole je 10, což odpovídá hodnotě proměnné pocetPrvkuPole.

```
void funkce (int pole[], int velikost)
{
int pocet;
pocet = sizeof(pole)/sizeof(int);
}
```

Ukažme si princip na příkladu.Pozor ale v okamžiku, kdy pole budeme předávat jako parametr funkce, pak je toto pole předáno hodnotou adresy prvého prvku pole, která je uložena do proměnné odpovídající parametru funkce, v následujícím příkladu parametru int pole[]. Uvnitř funkce již není dostupná informace o velikosti původního pole a proto se proměnná počet nastaví na hodnotu jedna, ať je původní pole jakkoli velké.

```
int main (int argc, char **argv)
{
int pole[10];
int velikostPole, velikostPole1, velikostPole2;
int *p, *q;
p = pole;
// p = &pole; nelze
q = &pole[0];
velikostPole = sizeof(pole);
velikostPole1 = sizeof(p);
= sizeof(q);
}
```

Proměnné p a q jsou ukazatele na int a je jim přiřazena adresa ukazatele pole a adresa ukazatele na první prvek pole. Do proměnné se uloží zjištěná velikost pole. Do proměnné velikostPole1 se uloží velikost proměnné p a do proměnné velikostPole2 pak velikost proměnné q. Mohli bychom mylně předpokládat, že pokud ukazatele p a q ukazují na pole o deseti prvcích typu int, pak operátorem sizeof získáme onoho velikost pole. Není tomu tak. Obě zjištěné velikosti budou jen čtyři, což odpovídá počtu byte, na kterém je implementován typ ukazatel, jak vidíme z hodnot proměnných velikostPole1 a velikostPole2.



Proto nebude fungovat funkce, jelikož skutečný parametr předaného pole se vyčíslí jako hodnota ukazatele pole, stejně jako je tomu v případě ukazatele p v našem příkladu.

2.2 ADRESOVÁ ARITMETIKA A OPERÁTOR SIZEOF()

Jelikož je ukazatel číslo, můžeme na něj aplikovat určité matematické operátory. Tyto operace byly navrženy pro efektivní implementaci indexace v poli. K ukazateli lze přičíst konstantu a lze spočítat rozdíl dvou ukazatelů.

```
int pole[3] = {1, 2, 3};
int a = pole[0]; // v a bude 1
a = *(pole + 1); // v a bude 2
```

Pozor na výraz pole + 1, mohli bychom očekávat, že výraz bude o jedničku větší než hodnota pole. Tak tomu ale není. Hodnota ukazatele se zde změní o velikost jeho typu, zde hodnotu čtyři. Ukazatel doménového typu nemá žádný typ, tudíž je podle standardu nelze s ním počítat pomocí ukazatelové aritmetiky. Nelze určit, o kolik by se měla hodnota ukazatele změnit, pokud bychom k němu přičetli jedničku. Abychom jej mohli použít v adresové aritmetice, je nutné ho nejprve přetypovat na nějaký typový ukazatel.

Na třetí řádce výpisu tak kód pole + 1 způsobí posunutí ukazatele na druhý prvek pole. Následná dereference tedy vrátí hodnotu dvě. Je tudíž ekvivalentní zápis pole[1] a * (pole + 1);

Přesně takto se interně převádí indexace polí v jazyce C – přičtení konstanty a následná dereference. Proto je také první prvek pole má index nula. Závorky kolem vlastního výrazu pole + 1 jsou nutné, neboť priorita operátoru dereferencování je vyšší než priorita operátoru přičtení. Proto následující kód do proměnné a zapíše hodnotu čtyři.

```
int pole[3] = {3, 2, 1};
a = *pole + 1; // v a bude 4
```

Ukazatele lze také odčítat. Musí jít o ukazatele stejného typu a měly by ukazovat na stejné pole. Výsledkem této operace je počet prvků pole mezi těmito ukazateli. U ukazatelů bez doménového typu platí to samé, co bylo uvedeno výše u přičítání konstanty. I zde je nutné ukazatel nejprve přetypovat.

K získání velikosti typu lze použít operátor sizeof(). Tento operátor přejímá název typu a vrací jeho velikost v bajtech. Vyhodnocení tohoto operátoru se provádí při překladu, protože překladač ví, jak jsou jednotlivé typy veliké.

```
int velikost = sizeof(int);
int velikost2 = sizeof(int*);
```

Hodnoty velikost a velikost2s e budou lišit podle toho, kde kód přeložíme. Na 64 bitovém systému budou hodnoty nejpravděpodobněji čtyři a osm. Není proto vhodné převádět ukazatel na číslo typu int, protože se na určitých architekturách nemusí hodnota ukazatele do rozsahu čísla int vejít.

2.3 UKAZATELE NA FUNKCE

V jazyce C se lze odkazovat na funkce. Deklarace ukazatele na funkci vypadá následovně:

```
int (*funkce)(int a, int b); // je deklarace ukazatele
int *funkce(int a, int b); // je deklarace funkce
```

Zde jsme vytvořili ukazatel na funkci, která vrací hodnotu typu int a přejímá dva celočíselné parametry. Od deklarace funkce se liší pouze přidáním hvězdičky a závorek. Závorky jsou povinné, neboť jinak bychom deklarovali funkci, která vrací ukazatel na int. Jak ale získat adresu funkce? Na funkci nelze použít operátor adresace &. Adresu funkce získáme přímo zápisem jméno funkce bez závorek.

```
int funkce(void);
// ...
funkce (); // je volání funkce
funkce; // ukazatel na funkci
```

Proto je při volání funkce bez parametrů důležité psát prázdné kulaté závorky. Poslední řádek výpisu ve skutečnosti nic nedělá. Takový výraz se při překladu odstraní, protože je zbytečný. Ukazatel na funkci lze použít pro předání určité konkrétní implementace funkce do obecného algoritmu. Jako příklad poslouží funkce qsort()ze souboru stdlib.h.

```
void qsort(void *base, size_t n, size_t size,
int(*compar)(const void *, const void *));
```

Tato funkce vyžaduje tzv. komparátor, tedy funkci, která implementuje srovnání dvou prvků ve struktuře, na níž ukazuje ukazatel base.

2.4 VÍCENÁSOBNÉ UKAZATELE ANEB UKAZATELE NA UKAZATELE

Jelikož ukazatel je proměnná jako každá jiná, lze získat adresu ukazatele. Mluvíme pak o ukazateli na ukazatel. Jak takový vícenásobný ukazatel vypadá?

```
int a = 2;
int *uka = &a;
int **ukuka = &uka;
int b = *(*ukuka);
```

Ukazatel na ukazatel int je ukazatel typu int*. Proto se nám v deklaraci objeví dvě hvězdičky. S podobnou konstrukcí jsme se setkali u funkce main (), jejíž druhý parametr je typu char**. Je to tedy ukazatel na ukazatel na char, zde se jedná o pole ukazatelů na řetězce znaků. Vícenásobné ukazatele se používají nejčastěji u vícerozměrných datových struktur, jako jsou matice nebo obecně pole polí. Z pohledu práce se nijak neliší od běžných ukazatelů, jen musíme provést vícenásobnou dereferenci, abychom se dostali k uloženým datům.

2.4.1 Pole ukazatelů.

Potřebujeme-li pole konstantních řetězců, je možno deklarovat pole ukazatelů na char a inicializovat ho:

```
const char *tab[]={"nula","jedna","dva","tri","ctyri"};
```

tab[0] nyní představuje řetězec "nula" apod. Výhodou proti dvourozměrnému poli char je to, že jednotlivé řetězce mohou mít různé délky.

2.5 DYNAMICKÁ ALOKACE PAMĚTI

V jazyce C se rozlišují dva typy dat. Jsou to staticky alokovaná data a dynamicky alokovaná data. V tomto článku si vysvětlíme rozdíl mezi statickou a dynamickou alokací a též si uvedeme funkce pro práci s pamětí.

2.5.1 Motivace

Až doteď jsme byli schopni pracovat s daty, o kterých jsme dopředu věděli, jak jsou rozsáhlá. Pokud jsme chtěli zpracovávat text po řádkách, museli jsme mít a priori znalost maximální délky řádku. Důvodem bylo, že jsme museli řádku uchovat v paměti a vytvořené pole mělo

velikost danou přímo v programu (byla pevně daná už před překladem). Tento problém částečně řeší lokální nekonstantní pole zavedené ve standardu C99. Nicméně takto vytvořené pole nelze měnit v době běhu programu, jeho délka je neměnná.
Proměnná či pole vytvořená v době překladu se nazývá staticky alokovaná proměnná. Její velikost zná překladač předem. Výhoda takového přístupu je prvně jeho rychlost a též bez obslužnost – při definování proměnné se vyhradí data a při opuštění příslušného bloku kódu se paměť automaticky uvolní. Tím, že se místo vyhrazuje pouze posunem ukazatele na zásobníku programu, máme zaručenu konstantní složitost takové alokace.
Dále v textu se budeme zabývat tzv. dynamicky alokovanými daty, tedy daty, jejichž místo v paměti je vyhrazeno až při běhu programu. Výhodou tohoto přístupu je velká flexibilita – máme kompletní kontrolu nad délkou alokace a můžeme ji dodatečně ovlivnit. Nevýhodou je relativní náročnost tohoto typu alokace, kde se musí projít halda (kde se dynamicky alokované proměnné ukládají) a vyhledat v ní příslušně dlouhý úsek spojité paměti. Správa paměti je přenechaná plně na programátorovi, který musí zajistit správné zacházení se zdroji v paměti, jinak může dojít k pádům programu nebo k vyčerpání volné paměti.

2.5.2 Vyhrazení a uvolnění paměti

Pro alokaci úseku paměti máme k dispozici několik funkcí z hlavičkového souboru stdlib.h:

void* malloc(size_t velikost)	**Vyhradí velikost bajtů paměti a vrátí ukazatel na první prvek. Ukazatel je bez doménového typu, je tedy třeba ho přetypovat na typ, který požadujeme.**
void* calloc(size_t nclenu, size_t velikost)	Vyhradí pole o nclenu členech, kde každý má velikost velikost bajtů. Alokované pole je vyplněno binárními nulami. Tato funkce je pomalejší než malloc(). Funkce vrátí ukazatel na první prvek a stejně jako u malloc() je vhodné přetypování.
void* realloc(void *pole, size_t velikost)	Realokuje pole pole. Funkce vytvoří nové pole o velikosti velikost bajtů a překopíruje do něj obsah původního pole, které muselo být dříve vytvořeno pomocí jedné z funkcí malloc(), calloc()nebo realloc(). Funkce vrátí ukazatel na první prvek nového pole. Nové pole může ležet na jiném místě v paměti než původní pole, původní ukazatel by se tedy již po zavolání neměl používat (zdrojové pole bude automaticky uvolněno). Je-li velikost menší, než je délka původního pole, bude v novém poli pouze fragment původního obsahu. Bude-li větší, nově vyhrazené místo nebude inicializováno. Nastavíme-li velikost na nulu, bude zdrojové pole uvolněno.

Velikost je udávaná v bajtech. Pokud chceme vytvořit pole typu int o deseti položkách, můžeme použít operátor sizeof()vysvětlený v sekci o ukazatelích. Výsledná velikost bude 10*sizeof(int).
Každá z výše uvedených funkcí vrátí ukazatel na první prvek alokovaného pole. Pokud ukazatel ppřepíšeme nebo pokud pozbude platnosti, zůstane místo vyhrazené, ale k vyhrazené paměti se již nedostaneme. Mluvíme pak o úniku paměti (memory leak), kdy se neuvolní alokované místo a zabírá donekonečna zdroje. Takové chování může u programů, které běží dlouho, způsobit vyčerpání volných zdrojů nebo nadměrné využívání odkládacího prostoru a

tedy zpomalení reakcí počítače. Řádné uvolnění paměti se provede voláním následující funkce:

Funkce	Popis
void free(void *pole)	Bezpečně uvolní zabrané zdroje. Pole pole muselo být dříve vytvořeno voláním jedné z výše uvedených funkcí. Předáme-li parametr NULL, funkce nic neprovede.

Pokusíme-li se uvolnit paměť dvakrát, obdržíme chybu segmentace a program bude typicky ukončen.

Ke každé alokaci by správně mělo příslušet i volání free(). Toto bývá ve složitějších programech často velmi náročné, a proto existují nástroje, které odhalí neuvolněnou paměť.

Pro vyplnění vyhrazené paměti určitou hodnotou slouží následující funkce:

Funkce	Popis
void *memset(void *zdroj, int vzor, size_t pocet)	Vyplní počet prvků pole zdroj vzorem uloženým v parametru vzor. Vrátí ukazatel na vyplněné pole zdroj.

2.6 ALOKACE VÍCEROZMĚRNÉHO POLE

Dynamicky alokované vícerozměrné pole lze vytvořit dvěma různými způsoby. Prvním a jednodušším postupem je vytvoření serializované verze vícerozměrného pole. Alokaci si vysvětlíme na příkladu vytvoření matice 3x3:

```
int i, j;
// jednorozměrné pole velikosti
int *matice3x3 = (int*)malloc(3*3*sizeof(int));
for (i = 0; i < 3; i++)
{
for (j = 0; j < 3; j++)
{
// vyplnění matice daty
// indexuje se jedním indexem
matice3x3[i*3+j] = i + j;
}
}
// ...
free(matice3x3);
```

Tento kód je jednoduchý, jeho jedinou nevýhodou je nutnost uchovávat velikost řádky pole, abychom mohli prvním indexem skákat přes celé řádky. Navíc tento přístup umožní operačnímu systému před načítat data, která jsou uchována v řadě za sebou, tudíž je přístup do paměti rychlejší. Příkaz

```
matice3x3[i*3+j] = i + j
```

nahrazuje indexový přístup do pole, číselně by odpovídal výrazu matice3x3[i][j] pokud by se jednalo o pole deklarován jako

```
int matice3x3[3][3];
```

Díky tomu, že ale nelze zaměňovat ukazatele int* int** nelze tento zápis použít.

Druhý přístup se velmi často vyskytuje v učebnicích o programování. Přesto se jedná o horší z obou přístupů. To proti, že konstrukce a uvolnění pole je komplikovanější, a navíc vlivem nesouvislosti dat je přístup do pole pomalejší. Výhodou je, že můžeme místo pointerové aritmetiky použít indexování.

```
int i, j;
// vytvoření přístupového vektoru
int **matice3x3 = (int**)malloc(3*sizeof(int*));
for (i = 0; i < 3; i++)
{
// vytváření jednotlivých řádek
matice3x3[i] = (int*)malloc(3*sizeof(int));
for (j = 0; j < 3; j++)
{
// vyplnění matice daty
// indexuje se klasicky dvěma indexy
matice3x3[i][j] = i + j;
}
}
// ...
for (i = 0; i < 3: i++)
{
//uvolnění řádek matice
free(matice3x3[i]);
// uvolnění přístupového vektoru
free(matice3x3);
```

Základem tohoto postupu je vytvoření tzv. přístupového vektoru, tedy pole ukazatelů na jednotlivé řádky. V cyklu pak naalokujeme jednotlivé řádky, které jsou již klasická jednorozměrná celočíselná pole. Indexace pak probíhá tak, jak jsme zvyklí u staticky alokovaných polí přes dva indexy (nemusíme si pamatovat velikost řádky). Uvolňování pole probíhá v opačném pořadí – nejprve uvolníme řádky a poté přístupový vektor.

2.7 STRUKTURY A VÝČTOVÝ TYP

Pro uchování množiny různorodých dat slouží struktury. Pro zacházení s množinou identifikátorů slouží enumerace. Nízko úrovňový kód zase využije unie. V tomto článku se zmíníme o všech těchto konstrukcích.

2.7.1 Výčtový typ enum

V programování se často stává, že máme proměnnou, která může nabývat jen přesně daného množství hodnot. Příkladem může být vykreslování, kde proměnná barva bude nabývat hodnot modra, cervena nebo zelena. Jednou z možností je používat číslo typu int a v dokumentaci vyjmenovat povolené hodnoty proměnné barva. Tento způsob není příliš praktický, neboť když pak narazíme na řádek kódu barva = 2, nemáme ponětí, která barva odpovídá kódu dva, dokud neprostudujeme dokumentaci. Můžeme místo toho vytvořit množinu konstant celočíselných konstant:

```
const int modra = 0;
const int cervena = 1;
const int zelena = 2;
const int blika = 3;
const int blika_rychle = 4;
// ...
barva = modra;
rychlost = rychle;
```

Tento zápis je už srozumitelnější, nicméně stále můžeme napsat:

```
barva = rychle;
```

Abychom se těmto problémům vyhnuli, lze využít výčtový typ enum.

2.7.2 Deklarace enum

Deklarace má následující syntaxi:

```
enum <[jméno typu]> {<výčet hodnot>}<[deklarace]>;
```

Začínáme klíčovým slovem enum, za kterým následuje nepovinné jméno nového výčtového typu. Ve složených závorkách je výčet možných hodnot oddělených čárkami. Jako poslední položka před středníkem je nepovinný seznam deklarací oddělený čárkami. V našem ovladači bychom tedy měli následující dva výčtové typy:

```
enum barva { modra, cervena, zelena} barvy;
enum blik
{blika = 3, blika_rychle} blikani;
```

Vytvořili jsme dvě proměnné: barvy typu enum barva a blikani typu enum blik.
Ve druhé deklaraci je zajímavé použití přiřazení hodnoty tři. Jelikož jsou výčty číselné množiny, každý člen je interně reprezentován číslem z nejmenšího rozsahu, který pokryje všechny prvky výčtu. Pokud žádné číslo nezadáme, jako je tomu v případě výčtu barva, začne překladač přiřazovat jednotlivým konstantám výčtového typu přiřazovat hodnoty od nuly a každá další položka dostane o jedničku vyšší hodnotu. V případě výčtu blik se začnou přiřazovat hodnoty tři a čtyři (překladač inkrementuje poslední zadanou hodnotu). Číselné hodnoty se ve výčtu mohou opakovat.
Pokud budeme chtít deklarovat proměnnou výčtového typu později, provedeme to následovně:

```
enum barva moje_baryvy;
```

Pokud se chceme opisování enum vyhnout, můžeme využít definici vlastního datového typu pomocí klíčového slova typedef.

2.7.3 Deklarace s použitím typedef

```
typedef <starý typ> <nový typ>;
```

Klíčové slovo typedef umožňuje pojmenovat nový typ, který se skládá z již existujících typů. Například je možné vytvořit typ ukazatel_na_int, který bude ve skutečnosti ukazatelem na hodnotu typu int.

```
typedef int* ukazatel_na_int;
//...
int a = 3;
ukazatel_na_int pa = &a;
```

Pojmenovaný výčet můžeme vytvořit tímto zápisem:

```
typedef enum {modra, cervena, zelena } barvy;
barvy moje_barva;
```

2.7.4 Přetypování na int

Jelikož jsou prvky výčtu ve skutečnosti čísla, můžeme je implicitně přetypovat na číselnou hodnotu. V jazyce C je možné i opačné přetypování a proměnné tak lze přiřadit i hodnotu mimo rozsah výčtu.

```
typedef enum {modra, cervena, zelena } barvy;
barvy moje_barva;
int a = modra; //
moje_barva = 4; // ok v C, chybně v C++
moje_barva = ( barvy)2; // explicitní přetypování
```

2.7.5 Porovnávání výčtových proměnných

Výčtové proměnné se při porovnávání chovají jako číselné hodnoty. V jazyce C jde bez problému porovnávat hodnoty z jiných výčtů.

2.8 STRUKTURA

Struktury jsou kontejnery pro heterogenní data. Srozumitelněji řečeno, jedná se společné pojmenování jednotlivých položek různých datových typů. Tak jako pole obsahuje prvky stejného typu, struktury se mohou skládat z položek různých datových typů. Typickým příkladem použití struktury je zápis adresy konkrétní osoby, jako je jméno, město, ulice, číslo, PSC.

2.8.1 Deklarace struktury

Deklarace struktury má následující syntaxi:

```
struct <[jméno typu]> {<výčet členů>}<[deklarace]>;
```

Od deklarace výčtu se liší pouze použitým klíčovým slovem a také výčtem členů. Výše uvedený záznam
adresáře by se uchoval ve struktuře:

```
truct adresa
{
char jmeno[30];
char mesto[20];
char ulice[20];
int cislo;
int PSC;
} Pepa;
```

Pro deklaraci s klíčovým slovem typedef a bez něj platí stejná pravidla jako u výčtového typu. Uvnitř
struktury může být jakýkoliv typ včetně ukazatelů na strukturu samotnou nebo jiné vnořené struktury.

2.8.2 Inicializace struktury

Struktura se inicializuje podobně jako pole, tedy výčtem hodnot oddělených čárkami ve složených závorkách. V případě vnořených struktur inicializujeme vnitřní strukturu dalším vnořeným párem složených závorek. Pokud vynecháme některé hodnoty, překladač zbytek doplní nulami stejně jako u polí.

```
struct adresa Pepa = {"Pepik","Nova Paka","Zahumenni", 28, 30100};
struct adresa Tonda = {"Tonda ","Pha " };
struct adresa Nikdo = {};
```

2.8.3 Přístup ke členům struktury

Pro přístup ke členům struktury slouží operátor tečka. Máme-li našeho zaměstnance Pepu, změníme mu věk následovně:

```
Pepa.cislo = 25;
```

Pokud máme pouze ukazatel na strukturu, máme dvě možnosti, jak se dostat ke členským proměnným. Buď ukazatel nejprve dereferencujeme a poté použijeme operátor tečka, nebo použijeme přímo operátor nepřímého přístupu, který provede prvně zmíněné kroky interně.

```
struct adresa *pk = &Pepa;
(*pk).cislo = 20;  // použití dereference
pk->PSC = 3100; //  pohodlnější,
//ekvivalentní dereferenci a použití operátoru přímého přístupu
```

```
// ke složce struktury tečka
```

2.8.4 Kopírování a porovnávání

Proměnné struktury stejného typu lze jednoduše kopírovat (na rozdíl od polí). Stačí pouze přiřadit jednu proměnnou do druhé a automaticky dojde ke zkopírování všech členů struktury.

```
// Franta a Tonda se stanou jednou osobou se stejnou adresou
Pepa = Tonda;
```

Problém nastává, když máme ve struktuře dynamicky alokovaná data. Při kopírování se zkopírují pouze ukazatele, ale data zůstanou sdílená (jde o tzv. mělkou kopii). V takovém případě se o zkopírování dat musíme postarat sami.
Pokud chceme proměnné struktur porovnávat, musíme tak učinit člen po členu, nelze jednoduše napsat:

```
if (Pepa == Franta) //...
```

2.9 UNIE

Unie se používají ve velmi specifických případech v nízko úrovňovém kódu a běžně se s nimi nesetkáme.
Syntaxí se velmi podobají strukturám, jen místo klíčového slova struct se použije union. Jediným rozdílem od struktur je to, že v daný okamžik existuje v paměti pouze jeden člen unie. Unie má tak velikost odpovídající prostorově největšímu typu v unii, mění se pouze interpretace dat.

```
union ZnakCislo
{
int cislo;
char znak;
} unie;
// unie bude mít velikost rovnou sizeof(int)
// ...
// nyní bude v unii číslo
unie.cislo = 65;

/*
Dotazujeme se na data v unii jako na znak. Hodnota ve znaku závisí na rozložení dat
v paměti (little nebo big endian). Dostaneme buď nejvýznamnější, nebo nejméně významný
bajt čísla. Vyhodnocení podmínky tak závisí na architektuře stroje.
*/
if (unie.znak == 'A') //...
```

2.10 SOUBORY

Soubory patří k základním datovým prvkům v počítači. Převážná většina programovacích jazyků má podporu určité formy práce se soubory. V jazyce C k tomuto účelu slouží funkce fopen(), fclose(), fscanf(), fprintf(),getc(), putc()a další.

2.10.1 Otevření souboru

Se soubory se v každém systému pracuje trochu jinak. Například v systému Windows jsou cesty oddělovány zpětnými lomítky a disk se značí velkým písmenem. V systémech odvozených od UNIXu se adresáře oddělují obyčejnými lomítky a pojmenování diskových oddílů písmeny neexistuje. Každý systém má také jinou filozofii oprávnění. Standard jazyka

C poskytuje jistou formu abstrakce v podobě funkce fopen()z hlavičkového souboru stdio.h, která otevře soubor pro manipulaci.
Funkce má následující syntaxi:

```
FILE *fopen(const char *cesta, const char *způsob);
```

Funkce přejímá dva parametry, oba jsou řetězce. První řetězec je cesta k souboru, Funkce přejímá dva parametry, oba jsou řetězce. První řetězec je cesta k souboru, ke kterému si přejeme přistoupit. Formát cesty je závislý na cílové platformě. Způsob je kód operace, kterou si přejeme se souborem provést. Dostupné způsoby práce jsou:

Způsob	Popis
"r"	Otevře soubor pro čtení. Zápis do souboru skončí chybou. Soubor musí existovat.
"w"	Vytvoří prázdný soubor pro zápis. Pokud soubor existuje, bude jeho obsah nejprve vymazán.
"a"	Otevře soubor pro zápis na konec. Soubor je vytvořen, pokud neexistuje. Veškeré zápisy probíhají na konci existujícího obsahu
"r+"	Otevře soubor pro čtení i zápis. Soubor musí existovat.
"w+"	Vytvoří prázdný soubor pro čtení i zápis. Pokud soubor existuje, bude jeho obsah nejprve vymazán.
"a+"	Otevře soubor pro čtení kdekoliv a zápis na konec. Soubor je vytvořen, pokud neexistuje. Veškeré zápisy probíhají na konci existujícího obsahu, i když předtím čteme na jiném místě souboru.

Kromě výše uvedených způsobů manipulace existuje ještě varianta s písmenem b, které lze umístit na konec řetězce nebo před znak +. Takový soubor bude chápán jako binární soubor, kdežto výše uvedené způsoby otevírají soubor jako textový. Na systémech odvozených od UNIXu není ve způsobu zacházení s textovými a binárními soubory žádný rozdíl, přesto se písmeno b doporučuje přidat kvůli přenositelnosti.
Funkce vrací ukazatel na strukturu FILE. Pokud dojde k chybě, vrátí funkce NULL, jinak obdržíme ukazatel na strukturu, v níž jsou uchované některé důležité údaje nutné pro práci se souborem. Tyto informace jsou ovšem před programátorem skryty, neboť se mohou systém od systému lišit.

```
FILE *f, *g;
// otevření konfiguračního souboru pro čtení UNIXu
f = fopen("/etc/X11/xorg.conf", "r");
/* otevření konfiguračního souboru pro čtení v systému Windows je nutné psát dvě zpětná lomítka, jinak bude následující znak chápán jako řídící sekvence
*/
g = fopen("C:\\boot.ini", "r");
```

2.10.2 Ostatní funkce pro práci se souborem

Abychom mohli pracovat se souborem, je nejprve nutné ho otevřít pomocí výše uvedené funkce fopen() a získat ukazatel na strukturu FILE. Poté se se souborem pracuje podobně jako se standardním vstupem a výstupem (což není náhoda). Následuje seznam několika důležitých funkcí pro manipulaci se souborem:

Funkce	Popis
int fclose(FILE *f)	Bezpečně uzavře předem otevřený soubor. Každý otevřený soubor by měl být řádně ukončen. Funkce vrací EOF(End Of File), nastane-li chyba.
int fscanf(FILE *f, const char *maska, ...)	Tato funkce funguje naprosto stejně jako funkce scanf()s tím rozdílem, že zdrojem dat je otevřený soubor f.

int fprintf(FILE *f, const char *maska, ...)	Tato funkce funguje naprosto stejně jako funkce printf()s tím rozdílem, že cílem je otevřený soubor f.
int fgetc(FILE *f)	Funkce přečte ze souboru právě jeden znak a přesune se na další. Pokud nastane chyba, vrátí funkce EOF.
int fputc(int c, FILE *f)	Funkce vypíše právě jeden znak do souboru. Pokud nastane chyba, vrátí funkce EOF.
long ftell(FILE *f)	Vrátí aktuální pozici v souboru v bajtech.
int fseek(FILE *f, long posun, int start)	Přesune se na pozici v souboru. Cíl je určen relativním posunem o posun bajtů vůči startovní pozici start. Startovní pozice může být SEEK_SET (začátek souboru), SEEK_CUR (aktuální pozice), SEEK_END (konec souboru). Funkce vrací mínus jedna, pokud nastane chyba.
int fflush(FILE *f)	Vyprázdní vyrovnávací paměť a zapíše veškerý dosud nezapsaný obsah do souboru. Tato funkce by se měla volat mezi po sobě jdoucím čtením a zápisem nebo zápisem a čtením do souboru. Pokud nastane chyba, vrátí funkce EOF.
FILE *tmpfile(void)	Vytvoří soubor s unikátním jménem tak, aby nedošlo ke jmenné kolizi. Soubor bude automaticky smazán po uzavření.

Jednoduchý program, který vypíše obsah předaného souboru, bude tedy vypadat takto:

```
#include <stdio.h>
int main (int argc, char **argv)
{
// byl předán název souboru?
if (argc < 2) return -1;
// otevřeme soubor, relativní cesta se vyhodnocuje k pozici
// spuštěného programu
FILE *f = fopen(argv[1], "r");
// povedlo se otevřít?
if (f == NULL) return -1;
int c;
// přečti obsah znak po znaku a vypiš
while ((c = fgetc(f)) != EOF)
{
putc(c);
}
// uzavřeme soubor
fclose(f);
return 0;
}
```

Všimněme si nyní, že se pro manipulaci se znakem používá typ int a nikoliv char, jak bychom očekávali. Je to z toho důvodu, že funkce fgetc()signalizuje chybu pomocí zvláštní návratové hodnoty EOF(End Of File). Kdyby funkce vracela znaky, nezbyl by pro tuto konstantu žádný kód. Proto je nutné rozšířit návratový typ na int, kde už EOF existovat může.

2.11 STANDARDNÍ VSTUP A VÝSTUP JAKO SOUBORY

Ač se to může zdát zvláštní, standardní vstup a standardní výstup nejsou nic jiného než soubory a jazyk C s nimi ani jinak nepracuje. Standard pouze poskytuje funkce, které práci s nimi usnadňují, můžeme s nimi ale pracovat pomocí souborových funkcí. Před spuštěním našeho kódu dojde k automatickému otevření tří souborů – stdin, stdout a stderr. Tyto soubory jsou na konci programu automaticky uzavřeny.
Soubor stdin je otevřen pro čtení a obsahuje vstup programu, soubor stdout je otevřen pro zápis a obsahuje výstup programu. Konečně soubor stderr je soubor otevřený pro zápis a obsahuje chybová hlášení programu. Výstup a výpis chyb tak lze oddělit. Pokud chceme vypsat hlášení na standardní chybový výstup, můžeme použít funkci fprintf()s prvním parametrem stderr.

2.12 PREPROCESOR

Preprocesor umožňuje ovlivnit kód před jeho vlastním překladem. Nachází uplatnění v multiplatformním kódu a při spojování větších projektů.

2.12.1 První fáze překladu

Direktivy preprocesoru jsou vždy uvozeny dvojkřížkem (#). Vše od tohoto znaku až do konce řádky je chápáno jako příkaz preprocesoru. Pokud potřebujeme uvést delší příkaz, který by zabral několik řádek, můžeme příkaz zalomit použitím zpětného lomítka na konci řádky. Slovo preprocessing znamená předzpracování. Tak tomu ve skutečnosti opravdu je. Překladač totiž nejprve zpracuje veškeré příkazy preprocesoru a až poté začne s fází překladu. Preprocesorem se dá ovlivnit kód několika způsoby: vyjmutím části kódu, nahrazení části kódu a vložení kódu z externího souboru.

2.12.2 Připojení souboru

Už v prvním programu jsme využili preprocesor. Příkazem

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

jsme preprocesoru řekli, že na toto místo v kódu si přejeme vložit obsah souboru stdio.h. Obecně můžeme pomocí příkazu include vložit na jakékoliv místo v programu obsah jiného souboru. Nicméně musíme být opatrní, abychom některý soubor nevložili víckrát, protože preprocesor toto nekontroluje a provede jen vložení. Při následném překladu překladač vyhodnotí, že kód obsahuje duplikátní deklarace.
V jazyce C se vkládají pouze hlavičkové soubory. Soubory se zdrojovým kódem je nesmyslné vkládat, neboť překladač potřebuje znát pouze deklarace použitých funkcí, nikoliv jejich těla. Příkaz include má dvě varianty.
Ta první používá špičaté závorky okolo jména souboru, ta druhá používá uvozovky. Význam se liší podle toho, v jakých adresářích se hledají příslušné soubory. Špičaté závorky jsou vyhrazeny pro systémové hlavičky a hledají se běžně v systémových adresářích (nebo v adresářích uvedených v parametrech překladače).
Naproti tomu soubory uvedené v uvozovkách se hledají relativně k umístění aktuálně překládaného souboru.

```
#include <stdio.h>
// vloží obsah hlavičky projektu
#include "mojehlavicka.h"
// ...
/*
opětovné vložení by mohlo způsobit problémy
v tomto případě se ale nic nestane z důvodu
vysvětleného níže
*/
#include <stdio.h>
```

Máme-li rozsáhlejší projekt, není mnohdy snadné uhlídat pořadí vkládání jednotlivých hlavičkových souborů. Velmi snadno se nám může stát, že se nám vkládání zacyklí či že vložíme jednu hlavičku dvakrát. Abychom se tomuto vyhnuli, používá se standardní konstrukce, která zabrání vícenásobnému vložení.

2.12.3 Podmíněný překlad

Podmíněný překlad je mocná technika, která umožňuje adaptovat kód v závislosti na prostředí, v němž je kód překládán. Nejde o nic jiného než o podmínky preprocesoru, které umí vypouštět bloky kódu.
Struktura příkazu se podobá příkazu if

```
#ifdef <makro> nebo #ifndef <makro> nebo #if <podmínka>
// ..
#else nebo #elif <podmínka>
// ...
#endif
```

Příkaz funguje následovně. Nejprve se vyhodnotí podmínka: v prvních dvou případech se ptáme, zdali byla nebo nebyla dříve definovaná proměnná preprocesoru (makro), ve třetím případě se vyhodnocuje podmínka.
Podmínka musí být vyhodnotitelná v době překladu a smí obsahovat pouze makra, nikoliv konstanty nebo proměnné z kódu. Je-li podmínka vyhodnocena jako pravda, vloží se do kódu blok programu až do else, elif nebo endif, pokud není uvedeno větvení. Je-li podmínka vyhodnocena jako nepravda, vloží se druhý blok kódu (je-li uveden).
Makro můžeme definovat několika způsoby. Tím nejběžnějším je použít příkaz define, který zruší předem definované makro.

```
/*
makra se běžně pojmenovávají velkými písmeny
tento příkaz vytvoří nové makro PROMENNA bez přiřazené
hodnoty
*/
#define PROMENNA
// nové makro PI s hodnotou 3.1415, středník se nepíše!!
#define PI 3.1415
// definice funkčního makra
#define abs(x) (((x)>=0)?(x):-(x))
Opakem příkazu define je příkaz
#undef <existujici-makro>,
```

Makra se používají jako substituční členy v programu a jako řídící proměnné. Poslední uvedená definice definuje funkční makro, které se chová jako funkce (za jménem makra jsou kulaté závorky). Zde definujeme absolutní hodnotu. Důvod, proč používáme tolik závorek v těle funkčního makra, je ten, že ve fázi předzpracování se neřeší priorita operátorů, což znamená, že bez závorek se může výraz vyhodnotit chybně.

V příkladu výše jsme vytvořili makro PI. Kdekoliv v kódu ho tak můžeme použít jako obyčejnou konstantu.

```
float y = sin(PI*x);
```

Před vlastním překladem se totiž všechna makra rozvinou, což znamená, že se všechny výskyty maker nahradí jejich hodnotami. Substituce probíhá na úrovni textu kódu, nedochází k žádné kontrole typů.

Proto je možné napsat následující chybný kód.

```
#define MAKRO "abc"
int a = MAKRO;
```

Zde si bude překladač stěžovat na nekompatibilitu typů, která se schovává v použití makra. Proto se doporučuje pro konstantní hodnoty využívat konstant namísto maker. Obecně chyby způsobené špatným rozvinutím makra jsou někdy obtížně dohledatelné, protože překladač si bude stěžovat až u substituovaného kódu.

Zabránění vícenásobnému vložení hlavičky

Jedním z možných řešení výše popsaného problému je použití následující konstrukce.

```
#ifndef SPECIFICKE_JMENO_PRO_HLAVICKU
#define SPECIFICKE_JMENO_PRO_HLAVICKU
// kód hlavičky
#endif
```

Při prvním vložení ještě neexistuje uvedené makro, a tak se hlavička vloží. Při dalším pokusu už makro existuje, a tudíž se celý kód hlavičky přeskočí.