v.2.03 6.9.2012

Organizace

- Přednášející: Ing. Petr Petyovký, Ing. Miloslav Richter, Ph.D.
  Ústav automatizační a měřící techniky FEKT, VUT Brno

- úvod do jazyka C s jednoduchými příklady, dotazy
- seznámení s programovacím prostředím MSVC
- složitější příklady – rozbor úlohy a její řešení
- dotazy - průběžně

Literatura

(jazyk C se mírně liší od ekvivalentních prvků jazyka C++, proto je lépe se soustředit na knihy o jazyku C a ne C++)

- Kernigham, Ritchie – Programovací jazyk C (ANSI C99)
- Herout Pavel – Učebnice jazyka C (především první díl)
- www.uamt.feec.vutbr.cz/~richter
- ...

Probírané součásti programování a jazyka

- historie jazyka, srovnání s ostatními jazyky
- algoritmy, jejich tvorba, nástroje
- program v jazyce C
- proměnné – typy, práce s nimi
- operace s proměnnými, bitové operace
- příkazy pro větvení programu
- cykly
- funkce
- pole
- formátované vstupy a výstupy
- práce se soubory
- preprocesor makra
- větvení programu – switch, case
- výčtový typ
- znaky a řetězce

1) Historie jazyka – vývoj norem v čase. Programovací jazyky - Základní členění programovacích jazyků (interpretované, nativní, fuze obou přístupů).

Historie a standardy (normy) jazyka

- norma jazyka popisuje – klíčová slova, mechanizmy, rozšiřující knihovny
- začátek 1969 Dennis Ritchie (AT&T Bell lab) – navržen pro tvorbu aplikačního SW
- K&R C – (1978) autoři Kernigham a Richie – mírně odlišný od současného C
- ANSI C (C89) – norma odvozená úpravami z K&R.
- C90 – jazyk C89 přijatý normami ISO s drobnými změnami (prakticky C89)
- C99 – přidány inline funkce, nové typy (long long, complex, podpora IEEE 754 float čísel), nový komentář, v některých případech je přísnější
- C11 – nové knihovny, multi threading, kontrola mezí, podpora UNICODE, zlepšuje kompatibilitu s C++
- embedded C (2008) – rozšíření C, fixed point operace, práce s (více) paměťovými bankami (pojmenované paměťové prostory), I/O adresace a operace,

Který ze standardů byste zvolili pro programování v jazyce C?
embedded, K&R, C89, C99, C11, C90?

V případě programování uprocesorů by se použil výrobcem doporučený překladač s integrovaným příslušným embedded C.
C11 – současná verze, přísnost překladu zaručuje lepší přenositelnost (mezi C překladači i do C++)
C99 – tato verze bude patrně implementována ve všech solidních překladačích
C89 (C90) - základ na který se lze spolehnout takřka ve všech překladačích
K&R – základ jazyka, ale dnes již nepoužívaný díky zastaralým konvencím

Jazyk C++

- jazyk C++ svými možnostmi je podstatně výše než C. Umožňuje objektové programování včetně vícenásobného dědění, implementuje mechanizmy výjimek, šablon,
- součástí C++ není odkaz na normu jazyka C, ale většina normy C je „zopakována“ v normě C++. Program v C tedy nemusí být přeložitelný v C++ (a naopak).
- jelikož normy C a C++ nejsou vydávány ve stejných okamžicích, liší se nově přijímanými vlastnostmi.
- C++ 98
- C++ 03
- C++ 09

Programovací jazyky a jejich členění

- Programy (a programovací jazyky) můžeme rozdělit i podle způsobu jakým se provádějí. Z hlediska hierarchie jsou na počátku mikroinstrukce procesoru,ve které jsou rozloženy instrukce strojového kodu (binární číslo určité délky jehož bity informují procesor o tom jakou činnost program požaduje). Jelikož psaní programu pomocí binárních či hexa sekvencí čísel není pohodlné, je možné použít jazyk symbolických adres (=asembler) ve kterém jsou tyto čísla reprezentována řetězci (jsou pojmenovány jmény
- Jak instrukce, tak adresy) - důležité je, že jméno je reprezentováno přesně danou instrukcí. Takto se programují především časově důležité části kódu. Následují vyšší jazyky, které již mají instrukce, které přesně neodpovídají instrukcím procesoru. Jejich výhodou je, že nejsou závislé na instrukční sadě procesoru. Lze je spustit na takovém, pro který existuje překladač. Například jazyk c se "snaží" o to aby vazba mezi ním a spustitelným kódem byla co nejbližší.

- Dalším druhem jsou tzv. interpretované jazyky, kdy se program nepřevádí do strojového jazyka, ale tzv. interpret si "čte" program tak jak je napsán a postupně ho provádí. Někde mezi nimi jsou způsoby, kdy je program přeložen "na půl cesty" do tzv. bytecodu, který je následně spuštěn v běhovém prostředí - spojení mezi programem a systémem - program je vykonáván pomaleji, ale v prostředí jsou implementovány složitější funkce (čtení a přehrávání obrázků a videa). Program lze spustit tam, kde je implementováno běhové prostředí.
- program převádí „algoritmy“ do instrukcí srozumitelných počítači (přímo procesoru, nebo nějakému rozhraní (virtuální stroj))
- nativní, kompilované – výsledkem je přímo v HW spustitelný kód, překlad jedenkrát, velmi rychlé provádění
- interpretované – interpret „čte“ programový text ve vyšším jazyku (instrukci po instrukci) a průběžně se jím řídí (převádí ho na instrukce srozumitelné HW)
- fuze (kombinace) – například překlad po řádku či bloku, just in time kompilace (JIT) – postupný převod jako u interpretu, ale výsledek si pamatuje, takže příště už přímo provádí instrukce, nebo převede před prvním spuštěním (source code -> bytecode-> machine code) – přizpůsobení se aktuálnímu prostředí.

Seřaďte podle „blízkosti“ k procesoru

- Strojový jazyk
- Asembler
- C
- Matlab
- C#
- Java

Strojový jazyk – respektuje instrukční sadu procesoru – binárně vyjádřeno
Asembler – respektuje instrukční sadu procesoru – vyjádřeno pomocí symbolických adres
C – vyšší jazyk, blízký strojovému jazyku, překládá se do strojového jazyka
C#, Java – vyjádření v bytecode
Matlab - interpret

2) Algoritmus, vývojové diagramy. Strukturované programování. Struktura programu v jazyce C, funkce main (). Pojmy prolog, epilog. Pojem proměnná. Jednoduché číselné proměnné (celočíselné, s pohyblivou řádovou čárkou) a jejich vyjádření v jazyce C.

Program se skládá z dat a kódu (rozděleného do funkcí), který s daty pracuje. Podle úrovně jazyka (překladače) a HW (procesoru) jsou k dispozici různé datové typy různé přesnosti a různé instrukce, v případě jazyka instrukční sada (interně realizovaná instrukcemi procesoru - jednotlivými, nebo sekvencí. Například pokud nemá procesor instrukci pro násobení, je potřeba násobení složit s instrukcí dostupných - například sčítání - toto řešení je ovšem pomalejší).
Kód programu se snažíme rozdělit na menší významově spojené části

Procesor má pouze operace s celými čísly. Je možné na něm počítat s reálnými čísly (čísla s pohyblivou desetinnou čárkou/tečkou)?

- řešením je například to, že část celého čísla bude desetinná (např. že číslo 1 bude v této reprezentaci reprezentováno číslem 1000 - poslední tři místa čísla tak budou reprezentovat desetinnou část). Druhým řešením by bylo reprezentovat číslo pomocí dvou hodnot - čísla a exponentu

Algoritmus

- je zápis vykonávané činnosti po jednotlivých krocích
- jedná se o popis řady povelů, která přesně popisuje prováděné operace směřující k danému cíli
- obvykle sestává z částí: vstupní bod (popis vstupních parametrů), výstup(ní) bod(y) (popis výstupních parametrů), vlastní činnost (prováděná činnost, větvení/rozhodování).

slovně popište algoritmus pro výpočet obsahu čtverce

Vstup funkce: výpočtu obsahu čtverce. Vstupní parametr: délka hrany
je-li délka hrany záporná -> vrať hodnotu -1 (signalizující chybu)
obsah = délka hrany * délka hrany
pokud došlo k přetečení (délka hrany příliš velká pro výpočet) -> vrať hodnotu -2
vrať vypočtený obsah

návratových hodnot může být více – jedna určuje typ chyby, druhá vypočtenou hodnotu

Vývojové diagramy

- grafické zobrazení algoritmu
- existují různé typy
- nejčastější pomocí šipek (určujících kudy se program „šíří“) a bloků (I/O, operace/výpočet, rozhodovací/větvící blok …

Nakreslete vývojový diagram pro výpočet obsahu čtverce

…

Strukturované programování

- procedurální, imperativní, funkční programování – založeno na volání podprogramů. Program plní jednotlivé kroky, aby se dostal k cíli.
- strukturované programování využívá podprogramů a bloků programu (například cykly) ohraničujících logické celky pro zlepšení čitelnosti, použitelnosti a udržovatelnosti psaného kódu. Nedoporučuje „chaotické“ skákání z místa na místo (příkaz goto, nejhůře křížem).

Struktura programu v jazyce C

- Program je předpis co má procesor dělat/vykonat
- Skládá se z částí - některé se nemusí vyskytnout, jiné mohou být navíc. Záleží to na typu OS, (u)procesoru… Prolog - kód/vlastní program (main) - epilog
- Obsah prologu a epilogu závisí na cílovém operačním systému
- Spustí se s parametry, vstupy a výstupy je možné přesměrovat
- Jelikož program je většinou spouštěn jiným programem (či OS, proces spouští proces) je slušné oznámit, jak jeho činnost dopadla pomocí návratové hodnoty (OK, neotevřen soubor, nedostatek paměti….

Seřaďte, jak jsou postupně použity:

- preved.exe /d
- Epilog
- Zpracování návratové hodnoty programu
- Připojení k síti
- Prolog
- Vlastní program

- preved.exe /d - spuštění programu s předáním parametru /d
- Prolog – zavedení programu, přichystání proměnných, po ukončení zavolá funkci main
- Vlastní program - napsaný programátorem, začíná funkcí main
- Epilog – následuje po ukončení funkce main, ukončení programu, úklid, vrácení zdrojů
- Zpracování návratové hodnoty programu – je možné provést v procesu, který program zavolal
- Připojení k síti – není součástí standardního procesu volání, program ho může použít

Program a paměť

Program je situován do paměti. V paměti zabírá většinou tyto bloky

- vlastní kód programu – blok, ve kterém jsou prováděné instrukce
- zásobník - ukládání lokálních proměnných, režijní data při volání podprogramů
- paměť dat - globální data programu
- Ostatní paměť systému – program má možnost o ni požádat (dynamická paměť, alokace)

příkaz

- Příkaz v jazyce C je ukončen středníkem
- Středník říká „tady je konec celku“, nemá funkci pouhého oddělovače výrazů ale je „spouštěčem“ vykonání
- Jelikož jediný středník má tuto funkci, je jeho uvádění povinné

funkce main

- Musí být v programu přítomna (u některých překladačů má upravené jméno) je to funkce, které překladač předá řízení
- Funkce main musí vracet celočíselnou hodnotu.
- V seznamu parametrů jsou jí předána data z programové řádky při volání (tato část může zůstat prázdná)

Plný tvar

```
int main(int argc, char *argv[],  char *env[])
{ return 0;}
```

Zkrácený tvar

```
int main( )     // hlavička funkce: návratový typ, jméno funkce, seznam parametrů
{               // začátek programového bloku – tělo funkce musí být blok
  return 0;     // zapsání návratové hodnoty funkce pro další použití
}               // konec bloku – těla funkce
```

založte v programovém prostředí projekt, v něm vytvořte soubor s01_main.c, a napište funkci main.

Prostředí pro programování – GUI (Graphical User Interface) v sobě integruje nástroje pro tvorbu kódu a jeho ladění. Z hlediska vytvoření programu využíváme:

- Překlad (compile) – typ překladače (C, C++) je dán rozlišením pomocí koncovky souboru (c, cpp). Provede kontrolu „správnosti“ jazyka (syntaxe - správný sled znaků a tvar slov; a sémantika - správnost použití proměnných a klíčových slov).
- Složení programových modulů (link) – program se skládá z několika souborů vlastních a cizích (knihovny). Tyto je nutné spojit do jednoho celku. Zde se kontroluje duplicita názvů funkcí a proměnné, nebo jejich nepřítomnost. Zároveň se provádí optimalizace (na rychlost, na paměť) a vytvoření výsledného kódu
- Celkové vytvoření programu (Build) – provede zkompilování a linkování všech částí programu
- Částečné vytvoření (Rebuild) – provede zkompilování a linkování jen těch částí, které se změnily od minulého sestavení (rychlejší, někdy může dojít ke zmatení překladače, zvláště při přenosu souborů odjinud)
- Vyčištění starých souborů (clean/ručně) – používá se na odstranění souborů vytvářených při překladu (vhodné před přenosem či archivací souborů).
  V případě větších změn (nebo po více Build preventivně) je dobré provést Clean+Rebuild.

Zkuste napsat funkci main a provést její překlad

komentáře

- Komentáře slouží k popisu činnosti programu
- Měly by popisovat filozofii, důvod proč to tak je a ne co tam je – to je vidět z kódu
- Na začátku modulu by měly být data k souboru (autor, verze, název, komentář …)
- Některé programy jsou schopny na základě úvodní sekvence komentáře vytvořit dokumentaci (najdou specifické komentáře a jejich „okolí“ – např. následující definici funkce).

které z následujících komentářů jsou správně
// funkce vraci nahodne cele cislo z intervalu 0 - 9
// dělení dvěma
/********+ scitani ******
**********************/
/********/   deleni ******/

Jednořádkový komentář – v pořádku
// funkce vraci nahodne cele cislo z intervalu 0 - 9
Jednořádkový komentář – špatně popis, popisuje to co je zřejmé z kódu
// dělení dvěma
Víceřádkový komentář – v pořádku
/********+ znak scitani *****
**********************/
Víceřádkový komentář – nemusí být na více řádcích, chyba je v tom, že spojení „vyplňující *“ a znaku pro dělení komentář uzavře a zbytek je brán jako kód
/********/ znak deleni *****/

Napište na začátek souboru s funkcí main pomocí komentáře hlavičku souboru.

```
/* soubor s01_main.c
Testovaní základních vlastnosti jazyka C
Autor:
Datum zalozeni:
Datum poslední upravy:
*/
```

Proměnné (identifikátory)

- Datové proměnné jsou charakterizovány typem (jménem typu) a jménem (proměnné)
- Proměnná leží v paměti, jejíž pozice je reprezentována jménem a datový typ určuje (normu) jak se bude s číslem pracovat (jak bude interpretováno v instrukcích) a jak velké množství paměti zabere.
- Podle typu je možné proměnné rozdělit na celočíselné, reálné a složené. Dalším typem je typ pro vyjádření adresy, na které leží proměnná.
- jelikož je nutné v určitých situacích (návratová hodnota nebo parametry funkce) uvést typ i když je nevyužíváme, existuje typ void, (značí, že se nic nevrací či nic nepředává v poli parametrů).

Celočíselné proměnné

- jsou znaménkové a bezznaménkové (kladné).
- jsou užívány převážně k indexaci.
- Bezznaménkové (zabírající stejný počet bytů) mají větší rozsah než znaménkové. U některých kombinací znaménkových a bezznaménkových typů může dojít k problémům (srovnávání, odečítání).
- Bezznaménkové je nutné používat pro bitové operace.
- Celočíselné proměnné odvozují svoji velikost od paměti (byte = 8bitů) a od velikosti registru či spojených registrů (hovoříme pak o slovech nebo dvojitých slovech).

Jaký rozsah hodnot signed a unsigned má jedno a dvoubajtové číslo?

Jednobajtové: -128 až 127; 0 až 255 (0xff)
Dvoubajtové: -32768 až 32767; 0 až 65535 (0xffff)

Velikost typů v jazyce C

- Není přesně dána, je určena hodnota, kterou musí umět reprezentovat, ale může být větší.
- Mezi jednotlivými typy platí pouze relace – „větší“ typ musí beze ztráty pojmout typ menší -> musí být stejný nebo větší
- Celočíselné typy jsou: char <= short int <= int <= long int <= long long int
- Velikosti pro daný překladač jsou dány v knihovnách <limits.h> (někdy i <ctype.h>)

Jaké jsou maximální možné hodnoty čísel vyjádřitelné celočíselnými typy?
(načtěte zmíněné hlavičkové soubory do souboru s main a otevřete je pomocí menu, které se otevře, pokud na názvu souboru zmáčknete pravé tlačítko myši).

Hlavičkové soubory

Překladač k tomu, aby mohl kód správně přeložit, potřebuje znát typy proměnných nebo hodnoty konstant. Tyto typy je ovšem nutné uvést do každého souboru s příponou c. K tomuto uvedení jsou určeny soubory s příponou h, které v C slouží pro zveřejnění typů a hodnot proměnných. Mechanizmus je takový, že definici napíšeme pouze jednou do souboru *.h, a tento vložíme („nainkludujeme) do všech zdrojových souborů, kde definice v něm obsažené potřebujeme. Mechanizmus funguje tak, že pokud napíšeme (příkaz preprocesoru)
#include <math.h> potom se do daného místa při překladu vloží daný soubor (jako by tam byl skutečně přítomen (skopírován). Takže soubor .h napíšeme jednou a „vkopírujeme“ ho tam kde je potřeba pomocí include. Při vkládání rozlišujeme hlavičkové soubory „systémové“, které leží v adresářích do kterých byl instalován překladač - #include <math.h> a zápis pro soubory „naše“ ležící v adresářích, ve kterých je zdrojový text - #include “mojemath.h“.

Reálná čísla (proměnné s pohyblivou řádovou částkou)

- skládají se z exponentu a mantisy (ve většině případů jsou obě části znaménkové). Opět se vychází z bytové reprezentace. Reálné číslo zabírající stejný rozsah v paměti jako číslo celé může pojmout větší číslo, ale v menší přesnosti.
- Reálné typy jsou float <= double <= long double
- Parametry/rozměry reálných čísel pro daný překladač jsou uvedeny v knihovně <float.h>.

Reálné číslo zabírá dva byty. První byte je mantisa, druhý je exponent. Jaká jsou největší a nejmenší kladná a záporná čísla, která lze vyjádřit?

Lze převést dvoubajtové celé číslo 32001 beze ztráty na tento reálný typ?

Jaká jsou maximální čísla vyjádřitelná pomocí reálných typů?

Jelikož se velikost zabrané paměti daným typem v C liší (pro jednotlivé platformy či překladače), existuje klíčové slovo sizeof umožňující zjistit velikost typu (v bytech).

```
unsigned int velikost;
velikost = sizeof(velikost);          // velikost typu, který je dán názvem proměnné
velikost = sizeof(unsigned int);      // velikost typu, který je přímo uveden
```

Zjistěte pomocí sizeof velikost celočíselných typů a porovnejte s údaji získanými v <limits.h>

Pro případ, že je nutné zachovat rozměr při přenosu na jiné platformy, existují definované znaménkové typy __int8, __int16, __int32, __int64, __int128 (u Microsoftu build in, nebo v intsafe.h).
Jinak v knihovnách <inttypes.h> nebo <stdint.h> - int8_t, uint8_t, …

Zjistěte definici int_32 (int32_t).

Definice proměnných, inicializace, konstanty

- Definovat proměnné je možné pouze na začátku programového bloku {…}, nesmí se mezi nimi objevit příkaz
- Pokud potřebujeme nadefinovat proměnnou později, musíme pro ni „vytvořit“ blok
- Při definici napíšeme požadovaný typ a názvy jedné nebo více proměnných
- Definice stejného typu se může opakovat, stejné jméno proměnné se opakovat nesmí
- Při definování je možné proměnné dát inicializační hodnotu (inicializovat ji). Tento postup se doporučuje – práce s neinicializovanou proměnnou je častá chyba.
- Pro každý typ existuje vyjádření konstantní hodnoty, kterým inicializaci můžeme provést
- Každá proměnná leží v paměti - toto místo je popsáno adresou.

```
int main()
{
signed char ca, cb = 10, cc = -20;  // znaménkový char bez inicializace
// hodnota nedefinovaná, kladné a záporné číslo desítkové
unsigned char  cd = 0xFEu, ce = 054u, ce = ´a´;
// bezznaménkový char – hexadecimálně a oktalově
// ´a´ je znaková proměnná – je konvertována na int ASCII hodnotu znaku
int i=0xc2, j=6, k=-3, m=0; // obdobně pro typ int, celočíselná nula
double da=3.4, db = 4.2e3, dc=-5e-5, dd = 0.0; // reálné varianty, reálná nula…
```

Nadefinujte a inicializujte proměnné a zjistěte (pomocí debugeru) jakou mají hodnotu a jak jsou umístěny v paměti. Zjistěte, zda jim rezervované místo odpovídá jejich velikosti – sizeof.

2) Přiřazovací příkaz, L-hodnota. Aritmetické a logické operace. Příkazy větvení `if, else`. Ternární operátor.

Konverze – převody proměnných různých typů mezi sebou

- často se stane, že se v přiřazení, ve výpočtu, nebo na místě parametrů, či ve funkcích, vyskytnou proměnné různých typů. Aby bylo možné pracovat, je nutné typy mezi sebou převádět.
- Jazyk C je v převodech typů velice benevolentní – pokud existuje nějaká cesta jak převod uskutečnit, pak je využita
- existuje konverze implicitní (tu provádí překladač) a explicitní (vynucená programátorem). Při konverzi může dojít i ke ztrátě (double=>char)
- Při vyčíslování příkazu je uskutečněn převod na typ proměnné, která má maximální přesnost
- celočíselné typy (menší než int) se automaticky převádí ve výpočtech na int
- při přiřazení je pravá strana převedena na typ levé strany
- při předání parametrů do a z funkce jsou proměnné převedeny na typ daný v hlavičce funkce
- explicitně je možné převést přetypování pomocí uvedení typu v závorkách, před proměnnou

```
int funkce (double xx, char yy)
 {
   float v;
   v = xx * yy; // počítáno v max. přesnosti – double, konertováno na float
   return v; // konvertováno na výstupní typ -> int
 }

int i;
double d;
d = i;  // implicitní konverze int-> double
d = i / 5; // výpočet v int, následně konverze
d = (double)i / 5; // programátorem vyvolaná konverze zapříčiní, že se pravá strana
// vypočte v maximální přesnosti - double
d = d/i; // výpočet v maximální přesnosti, i je konvertováno na double, pak výpočet

d = funkce(i, d); // i je konvertováno na typ double, d je konvertováno na char
// (ztráta přesnosti). Výsledek je konvertován z int na double
```

Přiřazovací příkaz

- Slouží k zapsání hodnoty do proměnné. Hodnotou může být výsledek výpočtu, konstanta, nebo hodnota jiné proměnné.
- Pokud je na pravé straně hodnota jiného typu než je proměnné, v rámci přiřazení se provede konverze (převod) na výsledný typ (je-li to možné. Jazyk C je však při těchto převodech velice benevolentní, a pokud najde cestu, pak hodnoty převede).
- Při přiřazení větší hodnoty než je maximum daného typu dojde k „ořezání“ hodnoty (toto nebezpečí je signalizováno upozorněním (warning) při překladu).
- Přiřazení je realizováno pomocí operátoru “=“. Neplést s “=“ v definici proměnné, kde je to inicializace a ne přiřazení.
- všechny přiřazovací příkazy mají vlastnost „zřetězení“. To znamená, že operátor = má návratovou hodnotu – proměnnou, které bylo přiřazeno

zřetězení: a = b = c = 0;

Co se stane při následujícím kódu

```
{
 int ia, ib=2;                  // není přiřazení ale inicializace
double da, db=3.14;
// proměnné stejného typu, hodnota se přepíše
ia = ib;
da = db;
// proměnné různého typu, nejdříve se provede konverze na typ výsledku, potom
přiřazení
// konverze je implicitní, vyvolaná překladačem
ia = db; // je-li velká mantisa, dojde ke ztrátě
da = ib; // je-li velké číslo a typy mají obsazenu stejně velkou paměť, dojde ke ztrátě

}
```

Zkuste upravit program tak aby ve dvou posledních řádcích skutečně došlo ke ztrátě.
Musíte upravit inicializační hodnoty a možná i změnit typ proměnných.
Možná nepůjde splnit najednou.

Zkrácené verze přiřazení

- Jelikož se často výsledek jednoduchých algebraických výpočtů ukládá do jednoho z operandů, existuje zjednodušená forma přiřazení
- Operátor pro zjednodušené přiřazení se skládá z operátoru požadované operace a operátoru rovná se (jedná se o dvojznak – proto musí být psány bezprostředně za sebou)
- Postup je takový, že se nejdříve vyčíslí pravá strana a potom se provede (případná konverze a) přiřazení společně s požadovanou operací

```
a += 10;
b *= 20 + 4; // ekvivalent b = b * (20+4)
```

L-hodnota, R-hodnota

- Často používané pojmy v souvislosti s přiřazením
- L-hodnota znamená, že může stát na levé straně přiřazení. Jinými slovy reprezentuje sebou něco, kam je možné uložit výsledek. Může to tedy například být proměnná, která je vlastně místem pro uložení hodnoty. Jako nevhodné pro L-hodnotu mohou být například konstanty, nebo výpočty. Z levých stran 13 = L; k*b = 10; je těžké určit kam s výsledkem. (nalevo je hodnota, která nemá určené místo v paměti).
- R – hodnota je hodnota, která může stát pouze napravo od rovná se (může být parametrem, nereprezentuje ale místo v paměti pro uložení výsledku) například 8+2
- R-hodnota reprezentuje pouze „obsah“ zatímco L-hodnota i „umístění“

Které z následujících výrazů jsou R-hodnota a které L-hodnota (proměnné jsou typu int)

a
a * b
a + 2
3
a3

L-hodnota : a, a3
R-hodnota: a, a3, a *b, a+2,3

Rozdělení operátorů

- Operátory lze dělit na matematické, boolovské a ostatní (speciální - například sizeof)
- Matematické - pracují s proměnnou po bitech a mohou být algebraické nebo logické.
- Boolovské - pracují s proměnnou jako celkem a znají dva stavy (na které si proměnnou převedou (konvertují)) - nula a nenula = false a true. Většinou je poznáme podle "zdvojení" operátoru.
-

Které operátory jsou algebraické a které logické?

+ & = / | % ~ ^ && ! >>=

matematické:
unární (+-),~(negace bit po bitu)
a = +b ; a = -b; a = ~b;

binární (+-*/%(modulo-zbytek po dělení)&|(OR)^(exclusiveOR) << >> (bitové posuny vlevo, vpravo)

a = b / e +c *d;
a = b % e;
a = (b & c) | (d ^ f);
a = (b << 3) | (c >> a);

logické
== < > <= >= && || !(NOT) !=

m = ((n==3) && (o > 5))|| (!(p >= 13) && (r != 10))

Existují speciální operátory zvýšení a snížení o jedničku (inkrementace a dekrementace - lze použít jednodušších instrukcí než pro sčítání/odečítání), které se používají při procházení cyklů, polí ...
Existují dvě verze pre a post. V prvním případě se proměnná změní a potom použije, ve druhém případě se proměnná použije a potom změní. Vícenásobné použití pro jednu proměnnou v jednom příkazu je zakázáno (protože není definován postup vyčíslení)

```
int i=10, j=10, k, l ,m ;
k = ++i + ++j ;
l = i-- + j -- ;
m = ++i + i++ ;
```

Jaké hodnoty budou mít jednotlivé proměnné z předchozího příkladu po vykonání daného řádku.

```
int i=10, j=10, k, l ,m ;
k = ++i + ++j ;  // i = 11, j=11, k = 22
l = i-- + j -- ;     // i = 10, j = 10, l = 22
m = ++i + i++ ; // nedefinováno
```

Aritmetické a logické operace

Typ výsledku

- je dán nejpřesnějším zúčastněným typem. Nejdříve dojde k výpočtu a potom k přiřazení s použitím konverze

Jaké budou výsledky?

```
double a=5 , b =2 ,c,d,e,f ;
int i=5, j=2 , k,l ,m, g,h;
c = a/b ;
d = i/j ;
k=a/b ;
l=i/j ;
e = i/b;
f= (double)i/j ;
m = i%j;
g = 2.7;
h =  -2.7;
```

c = 2.5 (a/b => 2.5)
d = 2.0 (i/j => 2)
k = 2 (a/b => 2.5)
l = 2 (i/j => 2)
e = 2.5 (i/b => 2.5)
f = 2.5 ((double)i/j => 2.5)
m = 1 (musí platit (i/j) * j + m = i => (2) *2 +1 = 5)
g = 2; h = -2 použití konverze pro „zaokrouhlení“. Ořeže desetinnou část a vrátí celek – „stahuje“ k nule. Lépe použít knihovní funkce – round, ceil, floor.

Bitové operace

napište kód, který nastaví n-tý bit v proměnné typu int. Následně napište kód, který ho vynuluje.

unsigned int vysledek = 0u; // pro bitové operace používáme bezznaménkový typ
unsigned int pom = 1u; // pouze pro demonstraci jednotlivých kroků – lze udělat v celku
int n=4; // který bit budeme nastavovat – bity jsou číslovány zprava a začínají bitem jedna

pom = pom << (n-1); // pomocnou jedničku posuneme o daný počet míst na požadovanou pozici
vysledek = vysledek | pom; // nastaveni bitu pomocí operace bitový OR
pom = ~pom; // bitová negace – všude budou jedničky, pouze na jednom místě nula
vysledek &= pom; // vynulovaní bitu pomocí operace bitový AND – „zkrácené“ přiřazení

Podmínky vyčíslení

- Pro složitější výrazy je nutné uvažovat i prioritu operátorů a jejich asociativitu.
- Pokud si nejsme jisti, pomůžeme si závorkami, které mají prioritu vyšší (program je i čitelnější)
- Priorita – říká, která operace se provede dřív. U operací se stejnou prioritou o pořadí rozhodne „překladač“.
- U logických operací se počítá do „určení výsledku“ – máme-li řadu podmínek spojených pomocí OR, končí výpočet při prvním TRUE, při řadě spojené pomocí AND, končí vyčíslování při prvním FALSE – proto by se nemělo používat v těchto výrazech příkazů pro inkrementaci/ dekrementaci – nemuselo by dojít k jejímu použití=vyčíslení.
- asociativita určuje, jak se vyčíslí výraz, ve kterém jsou operátory stejné priority

a * b * c + d * e

priorita určí, že se bude nejdříve násobit a potom sčítat (násobení má vyšší prioritu)
asociativita určí, zda se nejprve vypočte a * b nebo b * c – neurčuje, zda se první vyčíslí a*b*c nebo d*e – o tom rozhodne překladač

příkazy větvení programu

- slouží k podmíněnému rozdělení toku programu do dvou větví na základě podmínky. Při jejím splnění je použita jedna větev, při nesplnění se provede větev druhá – nepovinná.
- ve větvi může být jeden příkaz, nebo blok
- prvním příkazem pro větvení je dvojice if/else, druhým je ternární operátor

Příkaz if – else

if (podmínka)
  při podmínce splněné
else
  není-li podmínka splněna
	// do b zapsat abs(a)
	if (a > 0)
	  b = a;
	else
	  b = -a;

ternární operátor

- náhrada za dvojici if – else pro jednodušší výrazy
- na rozdíl od if else je výsledkem výraz, který se dá dále použit v jiném výrazu nebo jako parametr funkce
- má tři parametry – podmínku a výrazy které se provádí při jejím splnění / nesplnění

```
    (podmínka) ? při splnění : při nesplnění;
b = (a > 0)       ?      a          :       -a;
    (a > 0)       ?   b = a        :    b = -a;   // alternativní zápis
```

Pomocí příkazu if – else vyřešte minimum ze tří proměnných
Pomocí ternárního operátoru najděte minimum ze tří proměnných.

```
double a, b, c,v;

if  ((a < b) && (a < c))
  {
    v = a;
  }
else
  {
    if (b < c ) v = b;
    else v = c;
  } // výsledek je v proměnné v


v = (a < b && a <c ) ? a : ((b<c) ? b : c) ;
```

*Postup **sestavení programu** v prostředí MS Visual C (preprocesor, kompilátor, linker, zdrojové a hlavičkové soubory, spustitelné a knihovní soubory.*

*Základní **orientace v prostředí** – MS Visual C++*

Příklad na násobení dvou čísel.

3) Příkazy cyklů: `for, while`. Funkce, Volání funkcí – předávání parametrů do těla funkce – formální parametry, návratová hodnota. Globální a lokální proměnné, zásobník.

Cykly

- cykly se používají v případě, že je potřeba nějaký blok (sekvenci příkazů) několikrát opakovat na základě nějaké podmínky
- v C existují tři typy cyklů. Cyklus for se používá především tam, kde je znám počet opakování, nebo iterační krok. Cyklus while se používá tam kde je podmínka na počátku. Cyklus do – while tam kde je podmínka na konci, vždy se vykoná alespoň jednou.
- tělo cyklu se provádí, je-li podmínka splněna
- u for proběhne na začátku jednou inicializace, následně po vyhodnocení podmínky tělo cyklu a po něm iterace

```
//for(init;podminka;iterace)
int i;

for (i=0;i<10;i++)
 { //tělo cyklu
   // opakující se příkazy
   // tělo proběhne 10x
 }
```

```
// totéž jako for s while
int i = 0; // inicializace
while (i<10) //podmínka
{ //tělo cyklu
   // při splnění podmínky
  // opakující se příkazy
  i++;
  //změna řídící proměnné
}
```

```
// proběhne alespoň jednou
int i = 0; // řídící proměnná

do { // začátek těla cyklu
// opakující se příkazy

i++
// změna řídící proměnné
} while (i<10);
// je-li splněno,pokračujeme
```

pomocí příkazu cyklu (zkuste všechny tři typy) a bitového posunu zjistěte počet bitů v typu int. Na základě definice proměnných

unsigned int ui= ~0u; //pomocná proměnná pro výpočet–všechny bity nastaveny na 1
int i; // proměnná cyklu

```
// bloky cyklu jsou nezávislé, mohou v nich vystupovat různé proměnné
for (i=0; ui != 0;i++) // dokud jsou v pomocné proměnné jedničky, pokračujeme.
   ui >>= 1;
// v každém cyklu „vysuneme“ jednu jedničku a proměnná i nám ji započítá.
// Zde je použit posun doprava o daný počet (zde o jednu pozici).
// Do „prázdných“ míst se (zleva) doplňuje  hodnota 0.
// v proměnné i je počet bitů daného typu , zde (unsigned) int.
```

Příkazy řízení cyklu - continue, break

- s příkazy cyklů souvisí příkaz ukončení cyklu – break, a příkaz přeskočení zbytku cyklu continue
- příkaz break způsobí, že se ukončí nejvnitřnější (nejbližší) cyklus (ne blok)
- příkaz continue způsobí, že se přeskočí zbylá (následující) část těla cyklu a provede se (iterační část u for) podmínka a na jejím základě, se buď končí nebo pokračuje v daném cyklu
- pro opuštění cyklu (a také celé funkce) je možné použít i příkaz return
- v případě několikanásobně vnořených cyklů je možné použít příkaz goto k jejich opuštění (jedno z málo odůvodnitelných použití)

```
for (
    for (
        for (
            for (
                if (chyba) goto pryc;
}}}}}
pryc:
```

```
// program, který kopíruje písmena, čísla přeskočí, jinak končí
char PoleDat[50], Vysledek[50];
for (i = 0,j=0;i<50;i++)
 {
  if (isnum(PoleD[i]) // kde je definováno? Co dělá?
     continue; // je-li aktuální znak číslice, přeskočíme zbytek těla cyklu
  if (!isalfa(PoleD[i]))  // kde je definováno? Co dělá?
   {
     break; // není-li to písmeno, končíme
   } // break neskáče sem, je to konec bloku ne cyklu
  Vysledek[j++] = PoleD[i]; // písmeno zkopírujeme

// konec těla cyklu, sem skočí po continue, pokračuje iterací i++ a podmínkou ve for
 }// do tohoto místa se skáče po break (konec těla cyklu nejbližší k break)
```

Napište pomocí cyklu while program, který v poli sečte kladná čísla, záporná přeskočí, při nule ukončí cyklus. Procvičte si příkazy – continue a break;

funkce

- jsou základem programu, zpřehledňují ho,
- reprezentují opakující se kód, prováděný na základě parametrů
- funkce se skládá ze dvou částí - prototyp, hlavička funkce říká, jak se funkce jmenuje a s jakými parametry (počet a typ) pracuje. Druhou částí je tělo funkce, ve kterém je vlastní kód
- hlavička má tři části – typ návratové hodnoty, jméno funkce, seznam parametrů (čárkami oddělený seznam typ+jméno proměnné)

```
návratový_typ jméno_funkce(seznam_parametrů)
{
  // tělo funkce
}
```

```
double secti(double p1, double p2) // hlavička
{ // tělo
  double p = p1 + p2;  // výpočet
  return p; // pomocí return se výsledek uloží k dalšímu zpracování
}
```

volání funkce

```
double a;
a = secti(3.1, 8); // druhý parametr je typu int, proto musí překladač funkci „znát“
// provede se konverze (změna) předávaného typu int na typ požadovaný (double)
// tak aby s ním funkce uměla pracovat
// výsledek je uložen do určitého místa, ze kterého je použit – zde přiřazen do proměnné a
a = secti(3.1, 8) * 10; // s výsledkem je možné i pracovat
```

napište funkci, která vrátí minimum ze tří parametrů typu int. ukažte její volání

```
int MinInt(int p1, int p2, int p3)
{
 int pom;
pom = ...;
return pom;
}

ii = MinInt( 10, 12.1, 9);
```

Životnost/viditelnost proměnných

- Z hlediska životnosti (kdy proměnná fyzicky existuje) a viditelnosti (lze ji použít v programu) rozlišujeme proměnné na globální a lokální.
- Globální proměnná je definována mimo funkce a leží v globální paměti. Existuje (žije) po celou dobu programu a je dosažitelná ze všech modulů. Je možné ji použít od deklarace dále.
- Použití globálních proměnných se doporučuje jen v nejnutnějších případech (kód s těmito proměnnými se nedá sdílet- pro více řešení, mezi vlákny ...)
- funkce je vždy globální
- Lokální proměnné jsou proměnné, které existují pouze v bloku, ve kterém byly definovány. Jejich viditelnost/možnost použití je stejná – od místa deklarace do konce bloku ve kterém byly deklarovány. Proměnná (v C) může být definována pouze na začátku bloku. Lokální proměnné se tvoří na zásobníku. Lokálními parametry jsou i parametry funkce.

Rozdělte lokální a globální proměnné. Kde začíná a končí jejich platnost

```
extern double bb ;
int pocet ;
extern int cc ;

int main(int argc, char *argv[])
{
  int a ;
// příkazy
{
  double b ;
// příkazy
}
// příkazy
return 0 ;
}
```

```
extern double bb ; // deklarace – oznamuji existenci proměnné, fyzicky je v globální paměti
int pocet ;  // definice – oznamuji jméno, podle typu se zabere paměť v globální paměti
extern int cc ; // oznámení jména cc a jeho typu int

int main(int argc, char *argv[])
{  // začátek bloku – lokální proměnné– oznámení jmen a jejich typů– fyzicky vzniknou
  int aa ; // zde existují lokální proměnné argc, argv, aa (+ globální)
// příkazy
{
// začátek bloku– nová lokální proměnná b
// (+ nadřazené lokální argc,argv,a (+globální))
  double b ;
// příkazy
}
// zaniká s koncem bloku proměnná b – jelikož zanikne fyzicky, nelze použít ani její jméno
// příkazy
return 0 ;
} // zanikají lokální proměnné – argc, argv, aa
// konec souboru – konec viditelnosti bb, pocet, cc. V globální paměti jsou ale stále přítomny
// je možné je používat v celém programu
```

deklarace, definice

- Při použití názvu proměnné (i funkce) je nutné, aby překladač věděl, s jakým typem pracuje - aby vybral správnou instrukci. Proto musí v době kdy je proměnná uvedena, znát její typ.
- Toto lze realizovat deklarací (především u globálních proměnných mezi moduly) nebo definicí (globální i lokální).
- Deklarace oznamuje, že daná proměnná existuje a uvádí její název a typ - oznámení se provádí klíčovým slovem extern.
- Při definici dochází k uvedení typu a jména, navíc je ovšem proměnné přiděleno místo v paměti kde se bude hodnota proměnné fyzicky vyskytovat.
- Pokud je proměnné přiřazena i hodnota, hovoříme navíc o inicializaci proměnné.

Co jsou definice a co deklarace? Kdy dochází k inicializaci?

```
int i=0 ;
extern double a ;
long Int, j,k,l ;
extern char b,c,d ;
long m=2 ;
```

```
int i=0 ;        // definice (obsahuje i deklaraci) s inicializací
extern double a ; // deklarace
long int, j,k,l ;  // definice
extern char b,c,d ; // deklarace
long m=2 ; // definice s inicializací
```

4) Datový typ pole. Velikost pole. Operátory přístupu k prvku: [].: jednorozměrné pole, dvourozměrné pole,

Datový typ pole.

- zde je prezentován jeden z pohledů na pole a manipulaci s ním. Druhý (alternativní) pohled bude prezentován po probrání ukazatelů
- pole reprezentuje řadu po sobě následujících proměnných stejného typu
- indexace prvků probíhá od nuly. Pole o deseti prvcích má prvky s indexy 0-9
- při definici musí být uveden počet prvků, musí to být konstanta
- pro přístup k prvku pole slouží operátor [int]. Parametr značí pozici prvku od počátku pole

```
int Pole[20]; //dvacet prvků typu int,

int i, suma = 0;
for (i = 0;i<20;i++)
    suma += Pole[i];  // výpočet součtu prvků pole
```

Proveďte/naprogramujte spojení dvou polí do třetího

```
int i,j;
double p1[20], p2[10], p3[30];
// výsledné pole musí být připraveno tak aby se do něj vešly všechny prvky

// vlastní spojení
for (i=0;i<20;i++) // kopie prvního pole
   p3[i] = p1[i];
for (j=0;j<10;j++,i++) // přidání druhého pole.
// operátor čárka umožňuje, aby byly dva výrazy „na stejné úrovni“.
// výrazy oddělené čárkou se vykonají „zaráz“, může jich být libovolně mnoho
   p3[i] = p2[j];
```

Pole a funkce

- předávání pole do funkcí je možné, je však odlišné v tom, že ve funkci se netvoří lokální kopie, ale pracuje se s originálním polem
- předání/vrácení pole z funkce pomocí návratové hodnoty není možné

```
int soucet(int Pole[20], int delka)
// rozsah pole (20) je možné uvést, není ovšem využit,
// počet prvků je nutné dodat zvlášť

int soucet(int Pole[], int delka) // z hlediska funkce ekvivalentní zápis k předchozímu
{
 int suma = 0;
if (delka < 0) return 0;
for (delka--; delka >=0;delka--)
   suma += Pole[delka];
 return suma;
}
```

volání

```
int vektor[20]={1,2,4,5}; // inicializace pole, zbylé prvky doplněny nulou
int vysledek;
vysledek = soucet(vektor,20);
```

napište funkci, která setřídí prvky pole podle velikosti

```
void Setrid(int Pole[], int delka)
{
  int i,j,pom;

 for (i=0;i<delka-1;i++)
   for (j=i+1;j< delka;j++)        // rychlejší verze: for (j=delka-1;j>i;j--)
      if (Pole[j] < Pole[i])
       {
          pom = Pole[j];
          Pole[j] = Pole[i];
          Pole[i] = pom;
       }
}
```

volání

```
int vektor[30];
Setrid(vektor, 30); // výsledné setříděné pole je v prostoru pole původního
// původní pole tedy „zanikne“
```

Jak funguje princip třídění bublesort?

Jak funguje princip třídění quicksort?

Napište funkci PorovnejIndexem, která provede setřídění pomocí indexového pole. První parametr bude pole, které je nutné „setřídit“ od nejmenšího prvku, ale máme požadavek, aby zůstalo nezměněné. Proto předáme do funkce ještě jedno pole, ve kterém budou indexy prvků tak, aby určovaly prvky podle velikosti.
Použijte následující program, kde je i testování správnosti setřídění.

```
#define POCET 20
int PoleDat[POCET], Indexy[POCET],i;
// naplňte PoleDat náhodnými hodnotami 0-30
PorovnejIndexem(PoleDat,Indexy, POCET);
for (i = 0;i<POCET-1;i++)
    if (PoleDat[Index[i]]>PoleDat[Index[i+1]])
        printf(“chybne serazeno na indexu %d“, i);
```

vstup a výstup na konzolu

- vstupním zařízením je klávesnice, výstupním monitor (přístup jako k sériovým zařízením)
- spojení s konzolami je nastaveno automaticky na začátku programu
- ke konzole je možné přistupovat po jednotlivých znacích, nebo pomocí formátovaných verzí
- pro formátované přístupy slouží funkce printf a scanf
- skládají se ze dvou částí – formátovacího řetězce a seznamu parametrů
- formátovací řetězec se skládá ze tří prvků – prostého textu (který se tiskne), escape sekvencí (slouží k formátování - /n nový řádek), oznámení proměnných a formy jejich tisku
- seznam parametrů musí odpovídat oznámeným proměnným v řetězci

printf(“hodnota pozice %i je %f“, pozice, hodnota);
řídící znaky:
c char
i, d – int zobrazený dekadicky x int - zobrazený v hexa
f,g,e – float různé typy tisku mantisa exponent lf – double

načítání hodnot

- znaky uvedené se musí přesně vyskytovat v načítaném proudu
- u proměnných je nutné uvést znak & (adresa)

scanf(“(%d,%d,%f)“, &x,&y,&v)

ve vstupním řetězci musí být znak „(“ následovaný celým číslem čárkou celým číslem čárkou reálným číslem a znakem „)“
například (10,14,34.2)
v případě jiného textu dojde k chybě
návratovou hodnotou je počet načtených proměnných

soubory

- práce se soubory je podobná práci se standardním vstupem a výstupem
- soubor (stream) je základem práce se seriovými zařízeními, které mají stejný přístup (můžeme tak pracovat s diskem, klávesnicí, monitorem, pamětí, seriovou linkou …). Pro různé typy zařízení je používán stejný přístup.
- i stdin a stdout jsou vlastně typu FILE* (připravené prostředím). Jejich funkce jsou vlastně funkce pracující s FILE, které se ovšem nepoužívá, protože stdin a stdout se dodá překladačem
- u ostatních přístupů se funkce liší tím, že navíc je nutné uvést otevřený FILE
- je nutné si uvědomit, že máme ukazatel na řídící strukturu přístupu k seriové lince. Jejím pouhým přiřazením získáváme druhý přístup k téže řídící struktuře a tedy použití obou je ekvivalentní. Chceme-li mít dva "skutečné"=samostatné přístupy k jednomu souboru, musíme otevřít dvakrát soubor (pokud nám to operační systém dovolí).

```
FILE * vstup; // proměnná typu soubor nese data s informací o souboru
FILE * vystup;

vstup = fopen(“soubor.txt“,“rt“); // otevření souboru s daným jménem
// pro čtení (=r) v textovém modu (=t)
if (vstup == NULL) // signalizace chyby otevření
   printf(“chyba otevření“);
výstup = fopen(“vysledek.txt“,“wt“); // výstupní soubor pro zápis
if (vystup == NULL) // signalizace chyby otevření
   printf(“chyba otevření vystupu“);

...

fclose(vstup); // odevzdání přístupu k souboru, uzavření
fclose(vystup);
```

formátovaný vstup a výstup se soubory

- stejný jako při konzole
- rozdíl je v tom, že jako první parametr je odkaz na otevřený soubor

```
fprintf(vystup, “hodnota pozice %i je %f“, pozice, hodnota);
fscanf(vstup, “(%d,%d,%f)“, &x,&y,&v);
```

kopírování souborů

```
int znak; // i když pracujeme s char, funkce vrací i chyby typu int,
// které by se přiřazením do char ztratily (splynuly by s "legálním" kódem znaku)
znak = fgetc(vstup); // načtení jednoho znaku ze vstupu
while (!feof(vstup)) // funkce feof vrací hodnotu, zda bylo čteno mimo soubor
 {
   putc(vystup, znak); // přepsání znaku do výstupního souboru
   znak = fgetc(vstup);
 }
```

Pozn. Mimo funkce feof je možné pro zjištění konce souboru použít test na proměnnou EOF. Tato je vrácena při načtení prvního znaku, který nepatří k souboru. Je použitelná pouze u textového přístupu k souboru. Aby bylo možné tuto hodnotu odlišit od načítaného znaku (char hodnota 0-255), má hodnotu typu int. Proto při manipulací se souborem přednostně pracujeme s typem int.
Používat funkci feof je univerzálnější (bezpečnější).

Funkce pro práci se soubory

- fopen, fclose – otevření, uzavření souboru
- fgetc, fputc – načtení a výstup jednoho znaku
- fprintf, fscanf – výstup a načtení formátovaných řetězců, načte po bílý znak
- fgets – načtení řetězce (načte po konec řádku)
- feof – test na konec řádku
- ftell – zjištění aktuální pozice vůči počátku
- fseek(file, (long)offset,odkud) – nastavení na pozici ve streamu. Odkud – začátek/SEEK_SET, konec/SEEK_END, aktuální pozice/SEEK_CUR.

Práce s binárními soubory – přenáší se paměťové bloky (byty jak jsou v paměti)

- open, close – otevření, zavření souboru
- fread, fwrite – čtení a zápis bloku paměti

Příkazy preprocesoru

- Před překladem kódu překladačem mu „předpřipraví“ kód preprocesor
- Preprocesor vypouští nadbytečné (prázdné) mezery a řádky
- Preprocesor je možné ovládat pomocí příkazů - řádky začínající znakem #
- Nejčastěji používané příkazy – include, define, ifdef, ifndef, if (+defined), endif, asm
- Příkaz include – slouží k vložení souboru do daného místa (preprocesor „odbočí“ do jiného souboru a posílá ho do překladače – překladači se zdá soubor jako jeden celek
- Příkaz define slouží k definici proměnných k podmíněnému překladu, definici konstant, k náhradě složitých kódů (makra s funkčním voláním) – všechny definice platí od místa zveřejnění do konce souboru
- Příkazy define je možné „zrušit“ pomocí undefine

Definice konstant

- existují dva typy definic: „pro překladač“ a definice konstant
- příkaz s jedním parametrem #define EXISTUJE12 slouží k tomu, že překladač si zařadí daný řetězec (proměnnou EXISTUJE12) do tabulky a můžeme se dále ptát na její existenci (především ve spojení s příkazy preprocesoru #if...)
- příkaz se dvěma parametry #define MOJE_PI 3.1415 slouží k definici textových řetězců. Preprocesor si „zařadí“ první řetězec (od mezery za define po první další mezeru) jako referenční a přiřadí k němu řetězec druhý (od druhé mezery do konce řádku). Pokud preprocesor v dalším textu najde první řetězec, nahradí ho řetězcem druhým (který potom „vidí“ překladač).
  Tento mechanizmus slouží k lepší čitelnosti kódu (člověk vidí MOJE_PI, překladač „dostane“ 3.1215 – každý má to čemu rozumí lépe).
- jelikož se proměnná fyzicky nevytváří, bývají definice umístěny do hlavičkových souborů. Hodnota Pí má potom v celém programu stejnou přesnost, a je-li potřeba přesnost změnit, učiníme tak na jednom místě a změní se v celém programu.
- názvy za define je zvykem psát celé velkými písmeny
- řídící proměnné (náhradu za #define XXX) je možné zadat i jako parametry prostředí – platí potom pro všechny soubory

Zjištění „přítomnosti“ konstant

- pokud je podmínka splněna, potom preprocesor „pošle“ následující blok překladači, jinak ho vynechá
- #ifdef XXX – vrací true, je-li XXX definováno (preprocesor ho má v tabulce)
- #ifndef – vrací true, není-li definováno
- #else, #elif – větvení a větvení s další podmínkou
- #endif – ukončení bloku (začátek dán #if....)
- #if + výraz (pomocí define se můžeme ptát na přítomnost proměnných a skládat je pomocí logických operátorů (jazyka C) - && || !).

Realizujte tzv. „podmíněný překlad“ části textu. Pokud je zapnutý přepínač LADENI vytiskněte text „ladění“. Pokud je zapnutý LADENI a HODNOTA vytiskněte text „ladění“ a hodnotou proměnné xx dekadicky a hexa. Jinak proveďte tisk pouze dekadicky

#define LADENI
#define _HODNOTA

- definice proměnných. Proměnná HODNOTA je „vypnuta“ pomocí úvodního podtržítka¨

#ifdef LADENI

- test na přítomnost proměnné, je-li přítomna, pak se tato část překládá, jinak ne

printf(“ Ladeni “);
#endif

- konec podmíněného překladu

#if defined(LADENI) && defined (HODNOTA)
printf(“ Ladeni %d %x“, xx, xx);
#else
printf(“ Vysledek %d“, xx);

- tisk při nesplnění podmínkách („normální“ stav)

#endif

Pomocí define nadefinujte konstantu 2*PI

#define 2*PI 6.28

- nelze, jelikož první není regulérní řetězec

#define 2PI 6.28

- nelze, protože název nemůže začínat číslem

#define MOJE_PI 3.1415
#define DVE_PI MOJE_PI+MOJE_PI

- v pořádku (?). Pokud se nalezne text DVE_PI, nahradí se zbytkem řádku.

Co se stane v následujícím případě?

obvod = DVE_PI * polomer;

dojde k postupnému nahrazení řetězců

obvod = MOJE_PI+MOJE_PI * polomer;
obvod = 3.1415+3.1415 * polomer;

- z obou výrazů je patrné, že nedošlo k tomu, co jsme požadovali, poloměr násobí pouze druhé „Pi“.
- řešením je využití vysoké priority operátoru (), takže to co je mezi nimi se nejdříve vyčíslí

Opravte předchozí příklad

změna definice

#define DVE_PI (MOJE_PI+MOJE_PI)

- při použití již dělá očekávané

obvod = (MOJE_PI+MOJE_PI) * polomer;
obvod = (3.1415+3.1415) * polomer;

Pozn.

- předchozí mechanizmus definice konstant by měl být postupně nahrazen definicí proměnné s klíčovým slovem const.
- „klasická“ definice s const umožní zadat konstantě přesný typ.
- proměnná se nemusí fyzicky vytvořit.

```
int const MAX_POCET = 50;
double const PI = 3.1415;
```

- inicializace je jediné místo, kdy je možné přiřadit hodnotu, dále již manipulaci hlídá (a zakáže) překladač.

Použití:

```
if (i < MAX_POCET) {...}

obvod = 2 * PI * polomer;
```

Vkládání (hlavičkových) souborů - include

- umožňuje rozdělení souborů do modulů
- jsou v ní uvedeny deklarace proměnných, funkcí, maker, nových datových typů (enum, struct, union ...)
- jelikož se počítá s vícenásobných vložením (do různých cpp souborů), nesmí žádná část hlavičkového souboru tvořit (explicitně) kód – linker by našel více stejných proměnných a zahlásil by chybu.

Proveďte ošetření hlavičky proti vícenásobnému načtení. Stane-li se, že je hlavičkový soubor načten vícekrát (například v různých vkládaných souborech) je vhodné, aby se nezpracovával vícekrát než jednou (ztráta času, ...). Dalším problémem je, jsou –li dva soubory volány „do kříže“ – pak dojde k zacyklení. Zabraňte těmto chybám. uvnitř hlavičkového souboru vytvoříme následující konstrukci

```
// dotaz zda už jsme tu byli
#ifndef JEDINECNY_IDENTIFIKATOR_H_SOUBORU
// ještě ne (jinak bychom tento blok přeskočili)
#define JEDINECNY_IDENTIFIKATOR_H_SOUBORU
// ihned jako první si poznačíme, že už jsme tu byli

// tady bude vše potřebné – include jiných souborů, deklarace proměnných, funkcí, typů ...

#endif
// ukončení bloku
```

Pozn.: tento mechanizmus by byl vhodnější použít již při „volání“ souboru pomocí include – ušetřilo by se otevírání souboru. Ale výše uvedenou konstrukci by musel psát uživatel při každém volání, což by bylo pracné a při více načítaných souborech i nepřehledné

Makra s funkčním voláním – define s parametry

- podobné jako definice konstant. V prvním řetězci jsou však proměnné, kterými jsou při vkládání do textu nahrazeny proměnné z druhého řetězce¨
- většinou se snažíme, aby makro fungovalo jako funkce
- dosti často používané v systémových knihovnách (*.h)
- vhodné pro zpřehlednění textu, pro zápisem dlouhé ale kódem krátké bloky programu

```
// při definici
#define VYPOCET(a,b) a*a+b
```

je při použití nahrazení:

```
cc = VYPOCET (dd+ee,ff) * 6; // VYPOCET=> a*a+b      a => dd+ee       b=>ff
cc = a*a + b * 6 = dd+ee*dd+ee+ff * 6
```

opět vidíme, že to není to co bychom chtěli. Opravte definici a rozepište výsledné vyčíslení.

#define VYPOCET(a,b) ((a)*(a)+(b))

závorky kolem „parametru“ zajistí vyčíslení před použitím „parametru“
závorky kolem celého výrazu zajistí vyčíslení výrazu před dalším použitím

je při použití nahrazení:

cc = VYPOCET (dd+ee,ff) * 6; // VYPOCET=> a*a+b a => dd+ee b=>ff
cc = ((a)*(a) + (b))* 6 = ((dd+ee)*(dd+ee)+(ff)) * 6

nyní se makro chová jako funkce.

Jsou zde však stále rozdíly mezi použitím makra a funkce. Jaké?

Funkce provádí konverze na typy dané v definici.
Předávaný parametr se vyčíslí a změní typ na typ parametru, který je uveden v hlavičce funkce. Obdobně se konvertuje návratová hodnota.
Makro počítá v typech použitých proměnných (v největší přesnosti z použitých). Při „volání“ s proměnnými typu int je výpočet prováděn v int, při volání s double je výpočet prováděn nad typem double.

Funkce je v celém programu pouze jednou. Při jejím použití je volána jako podprogram. Nevýhodou je časová režie spojená s předáváním (a úklidem) proměnných a skokem do podprogramu. Makro se vkopíruje pokaždé do textu zdroje a v každém místě použití se znovu přeloží. Takový program je rychlejší. Pokud je ale makro delší než kód režie (jednotky až desítky instrukcí), dochází k prodlužování kódu programu.

Zápis pomocí makra by měl být nahrazen pomocí tzv. inline funkcí. Jejich překlad je proveden jako u makra přímo v kódu bez volání, ale zároveň se provádí přetypování na základě typů z hlavičky funkce.

```
inline int Vypocet(double a, long  b) {return a*a+b;}
```

při použití

```
double a;
a = Vypocet (‘c’,3.245);
```

se v daném místě přeloží

```
a = (int) ((double)( ‘c’) * (double)( ‘c’)+(long)( 3.245) )
```

definice vlastních maker. Bitové operace a operátory.

Maskování je příklad logických operací pro jednotlivé bity. Pro tyto operace se přednostně používají bezznaménkové celočíselné (!) typy. Pokud potřebujeme proměnnou obsahující samé jedničky, potom je dobré si ji "vyrobit" pomocí bitové negace - pouze tak si můžeme být jisti, že výpočet na jiné platformě nebude ovlivněn změnou velikosti typu.

```
#define cidlo1 0x4
#define cidlo2 0x8
#define motor3 0x10
#define motor4 0x2
unsigned int stav = 0xF ;
unsigned int rizení = ~0u ;
//zjistěte, jsou-li sepnutá čidla 1 a 2 v proměnné stav
// v případě, že ano, vypněte (vynulujte) bity pro motory 2a 3 v proměnné řízení
```

Enum – výčtový typ

- definuje „nový“ typ, jehož hodnoty je možné určit výčtem (uvedením) v definici
- po definici lze použít jako kterýkoli jiný typ
- vlastní reprezentace symbolických konstant (hodnot typu) je celočíselná (začíná nulou, lze i explicitně určit, další je o jedna větší než předchozí)
- lze konvertovat z/na int v rámci definovaných hodnot enum (používat opatrně)
-

```
enum Barva {Cervena, Zelena, Modra, Bila}; // interně přiřazeny hodnoty 0,1,2,3

enum Barva kruh, ctverec = Cervena; // definice proměnných typu enum
kruh = ctverec;
if (kruh != Bila)
    ctverec = Zelena;
```

Nadefinujte pomocí výčtového typu nový typ pro logické proměnné. Ukažte příklad definice, inicializace a použití tohoto typu.

```
Enum BOOL {False = 0, True = 1};
// pro jistotu definujeme i hodnoty konstant pro správnou konverzi


BOOL ba , bb = False;
int i=4,j=5;

ba = i == j; // vyčíslí se porovnání (i a j) a výsledek se konverzí přiřadí do ba
if (ba)  // konverze na int
  bb = True;
```

Přepínač (switch – case – break)

- slouží k vícenásobnému větvení
- příkaz je switch, za kterým je uvedena celočíselná hodnota
- následuje tělo switche, ve kterém jsou „návěstí“ case hodnota:
- pokud v těle (za case) existuje hodnota přesně odpovídající předané hodnotě (ve switch), potom program pokračuje touto větví (od tohoto místa)
- program pokračuje stále dál, dokud nedojde na konec těla switche, nebo nenarazí na příkaz break (nekončí tedy na dalším uvedení hodnoty pomocí case).
- může existovat i (jediná) sekce označená default pro všechny ostatní (neuvedené) hodnoty parametru.
- příkazy switch jdou vnořovat, i když při složitější konstrukce jsou nepřehledné

```
switch (hodnota) {
case 0: ... ; break;
case 1: ...;
case 2: ...; break;
default: ... ;break;
case 3: ...;break;
}
```

vytvořte pomocí příkazu switch funkci, která dostane jako parametr znak a vrátí předdefinovanou proměnnou YES pro malé a velké „a“ a číslo „1“; pro malé a velké „n“ a číslo „0“ vrátí proměnnou NO. V ostatních případech vrátí ERROR.

```
#define YES 1
#define NO 0
#define ERROR -1
int KeyDetect(int znak)
{
	int ret=ERROR;
	switch (znak) { // znak musí být celočíselný typ
		case ’a’: // hodnota znak musí přesně odpovídat  vyjmenované hodnotě v case
		case ’A’:	// všechny tři znaky mají stejnou část programu.
		case ’1’: ret = YES;break; // od označeného místa program stále pokračuje,
// pokud není příkaz break, který zapříčiní skok za tělo příkazu switch
		case ’n’:
		case ’N’:
		case ’0’: ret = NO;break;
		default: ret = ERROR;break;//default větev pro nevyjmenované hodnoty
	}
	return ret;
}
```

Ukazatel (Pointer) jako datový typ

- proměnné jsou umístěny v paměti na určitém místě (adrese) a zabírají určitý prostor (počet bytů), který je daný typem proměnné
- ukazatel je tedy adresa společně s typem, který je zde uložen. To umožňuje snadno pracovat s daty uloženými na dané adrese – reprezentované ukazatelem
- hodnota ukazatele je reprezentována podle paměťového modelu (celé číslo reprezentující paměť, celé číslo reprezentující offset vůči hodnotě registru, dvojice celých čísel (segment/báze a offset vůči ní), ...), typ je s ukazatelem spojován v překladači.

Proměnné jsou uloženy v paměti za sebou v pořadí definice, od adresy 200H a směrem k vyšším hodnotám. Adresa je reprezentována celým číslem.

Zakreslete paměťovou mapu, jestliže definice proměnných je:

```
int a,b;
double c;
```

a po průběhu programu:

```
a = sizeof(a);
b = sizeof(double);
```

nabývají proměnné hodnoty a = 4 a b = 8;

adresa	proměnná a uložená hodnota
200	a 4
204	b 8
208	c nedefinováno
20F	

získání adresy (reference)

- definice ukazatele je realizována pomocí typu a hvězdičky (která platí pouze pro jednu proměnnou)
- ukazatel by měl obsahovat smysluplnou adresu – je ho třeba inicializovat tak aby obsahoval adresu, na které leží stejný typ, jakého je ukazatel
- operátor pro získání adresy je & (znak logické and – má nyní dva významy)
- pro neinicializovaný ukazatel, nebo jako označení chybového stavu se používá konstanta NULL (či nullptr)

```
int *pii, ii; // pii je ukazatel na int, ii už je typu int
// v pii i ii jsou „nesmysly“ – obě proměnné dosud nebyly inicializovány

ii = 0; // inicializace konstantou
pii = NULL; // dále lze testovat, zda obsahuje smysluplnou adresu ( != NULL)
pii = &ii;  // inicializace pomocí adresy existující proměnné
// operátor zjistí adresu, na které leží proměnná ii a přiřadí ji do pii.
// hodnota pii tedy udává místo v paměti, na kterém leží hodnota typu int (původně ii)
```

Nakreslete paměťovou mapu pro následující příklad, platí-li pravidla z minulého příkladu a adresa je reprezentována osmi byty.

```
int i = 5;
int *pi = &i; //definice s inicializací. Udává místo v paměti, kde leží int
int **pii = &pi; // ukazatel na ukazatel. Udává místo v paměti, na kterém leží adresa
// na níž leží int
int ***piii = &pii; // Udává adresu, na níž leží adresa, na níž leží adresa, na níž je int
```

adresa	počet bytů	název / hodnota
200	4	i / 5
204	8	pi / 200
20C	8	pii / 204
214	8	piii / 20C

přístup k prvku na adrese (dereference)

- je realizován pomocí operátoru přístupu (dereference) „hvězdička“, který je aplikován na ukazatel
- ukazatel obsahuje adresu a překladač „ví“ jakého je ukazatel typu, takže umí s obsahem pracovat
- operátor hvězdička má nyní tři významy (plynoucí z kontextu)
  jako matematický operátor krát,
  v definici značí, že se jedná o ukazatel,
  před ukazatelem znamená, že se pracuje s obsahem na dané adrese

```
int *pii, ii; // pii je ukazatel na int, ii už je typu int

ii = 0; // inicializace konstantou
pii = &ii;  // inicializace pomocí adresy existující proměnné

*pii = 10; // v pii je adresa prvku ii. Hvězdička říká, že se pracuje s obsahem
// na této adrese. Tento příkaz je tedy ekvivalentní zápisu ii = 10, a zapíše
// na adresu (na které leží obsah/hodnota ii) hodnotu 10.
```

Použijeme-li minulý příklad

```
int i = 5;
int *pi = &i; //definice s inicializací. Udává místo v paměti, kde leží int
int **pii = &pi; // ukazatel na ukazatel. Udává místo v paměti, na kterém leží adresa
// na níž leží int
int ***piii = &pii; // Udává adresu, na níž leží adresa, na níž leží adresa, na níž je int
```

potom následující zápisy v řádcích jsou ekvivalentní

manipulace s i	i = 5	*pi = 5	**pii = 5	***piii = 5
manipulace s pi		pi = 200	*pi = 200	**piii = 200
manipulace s pii			pii = 204	*piii = 204
manipulace s piii				piii = 20C

podobně se mohou vyskytovat i na pravé straně rovná se, nebo jako parametry funkcí

Datový typ void

- je úzce spojen s ukazateli
- používá se jako „univerzální“ datový typ, ke kterému mají všechny ostatní typy stejně blízko (či daleko)
- pro převod je vhodné použít přetypování

```
int * pi;
void *pv = (void*)pi;
pi = (int *) pv;
```

Ukazatel jako parametr funkce

- slouží k předání výsledku mimo funkci. Funkce má možnost předat jednu návratovou hodnotu. Další návratové hodnoty je možné předat pomocí ukazatelů v poli parametrů. Ukazatel funkci odkáže na místo, kam má uložit výsledek.
-

Napište funkci, která načítá data z klávesnice / souboru a vrací minimální a maximální hodnotu. Návratová hodnota obsahuje počet hodnot, ze kterých byly určeny.

```
//hlavička říká, že předáváme ukazatele/adresy, na kterých jsou hodnoty typu double
int MinMax(double *aMax, double *aMin)
{int Minimum, Maximum;
 ...
 *aMax = Maximum; // na předanou adresu se zapíše nalezená hodnota
*aMin = Minimum; // před přiřazením dojde ke konverzi
 return Pocet; // vrátí se počet hodnot
}
```

volání:

```
double mi, ma;
long kolik;

kolik = MinMax( &ma, &mi); //ukazatele musí být na stejný typ jako v prototypu fce
// návratová hodnota je pro přirazení konvertována – nemusí být stejný typ
```

Ukazatel jako návratová hodnota

Který z následujících předávání ukazatele jako návratové hodnoty je možný? A proč?

```
double * Funkce(double hodnota, double adresa*, double *adresa 2)
{
 double vysledek;

// možné výsledky

return *adresa;
return adresa;
return hodnota;
return &hodnota;
return &adresa;
return vysledek;
return &vysledek;
}
```

```
double * Funkce(double hodnota, double *adresa, double *adresa2)
{
 double vysledek;

// možné výsledky

return *adresa; // nesouhlasí typ - double
return adresa; //typ souhlasí, adresa je předána „z venku“->existuje, může být vrácena
return hodnota; // nesouhlasí typ
return &hodnota;//nelze předat „ven“ adresu lokální proměnné, která zanikne nakonci
return &adresa; // nesouhlasí typ – double **
return vysledek; // nesouhlasí typ - double
return &vysledek; // nelze předat adresu lokální proměnné
}
```

Napište funkci tak, aby vrátila minimum ze tří hodnot typu int. Zároveň je dán požadavek, aby bylo možné tuto proměnnou nahradit daným čísle – tj. aby mohl být nalevo od znaménka rovná se.

```
int *Min3(int *a1, int *a2, int *a3) // předávají se adresy
{
  if  ((*a1 < *a2) && (*a1 < *a3)) // porovnávají se hodnoty
     return a1; //  vrací se ukazatel (adresa)
 if (*a2 < *a3)
     return a2;
return a3;
}
```

volání:

```
int x1=10, x2=20, x3=-10, r;

r = *Min3(&x1,&x2,&x3); // předávají se adresy, vrací se adresa. Přístup dereferencí
// pomocí dereference se dostaneme k hodnotě na vrácené adrese
*Min3(&x2,&x3,&x1) = 10; // parametry jinak umístěny, výsledek musí být stejný
// ve funkci Min3 bude ale jiný průběh při krokování
// návratovou adresu je možné použít i k zápisu na dané místo
r = *Min3(&x1,&x1,&x1); // důležitá „ladící dovednost“ – všechny parametry stejné
```

pole a/versus pointer

- z uvedených vlastností ukazatelů a typů je výjimka – proměnná typu pole
- pole a ukazatel mají natolik podobný přístup, že pokud je to možné, pole se konvertuje na ukazatel a dále se s nimi manipuluje stejně
- základní vlastností je, že název pole se konvertuje na ukazatel na první prvek pole („ztrácí“ se tím pojem o délce (?), kontrola opuštění mezí není ani v jednom případě)
- další vlastností je možnost „oindexování“ ukazatele (k adrese se přičte hodnota indexregistru – dojde k posunu na jinou adresu).
- díky znalosti typu ukazatele a tím i velikosti daného typu je krokem posunu v poli (i adresy ukazatele) vzdálenost rovná rozměru daného typu -> posunujeme se po prvcích pole
- první prvek má offset nula, druhý jedna ...
- index může být i záporný
- ukazatel je možné měnit (posunout), „hodnota“ pole je konstanta

Přístup k prvkům pole

- pro přístup k prvkům pole slouží unární operátor [ ] – parametrem je celočíselná hodnota udávající prvek od počátku (vztažná adresa)
- další možností je využití tzv. ukazatelové aritmetiky. Přičte-li (odečte-li) se k ukazateli celé číslo, výsledkem je adresa tolikátého prvku pole.

```
int Pole[10]={0,1,2,3,4,5,6,7,8,9}; // pole o deseti prvcích, indexy 0-9
int *pi = NULL; // pomocný ukazatel na int

pi = Pole; // díky konverzi ukazuje nyní pi na první (nultý) prvek Pole
Pole[2] = -2;
// přístup ke třetímu prvku pomocí operátoru [ ] – pracuje s hodnotou na dané pozici
*(Pole + 2) = -2; // ekvivalentní zápis – posun ukazatele o dvě „délky“ typu int
// a přístup k dané adrese. Závorky () jsou nutné kvůli větší prioritě * před +

*(pi + 2) = -2;
pi[2]=-2;
// přístup pomocí ukazatele je stejný. Nedochází ke konverzi z typu pole
```

S ukazateli je spojen pojem *ukazatelová aritmetika*

- přičteme-li (odečteme) k ukazateli celé číslo, „posune“ se adresa o daný počet prvků typu, na který ukazatel ukazuje (tj. N * sizeof(typ pole) bytů)
- přičtením (odečtením) jedničky se dostáváme na další (předchozí) prvek
- odečteme-li dva ukazatele (musí patřit ke stejnému poli/paměťovému celku) získáme počet prvků, které mezi nimi leží
- další matematické operace nejsou pro ukazatele definovány

```
int Pole[20]={0};
int *první, *aktualni,*poslední,i,pocet;// tři ukazatele na int a dva inty
int suma=0;
for (aktualni =první=Pole, i=0;i<20;i++,aktualni++) //posun ukazatele na další prvek
      suma += *aktualni;  // součet prvků v poli – nahraďte pro Pole a prvni
pocet = aktualni – první; //počet prvků mezi ukazateli.
// Prvky jsou typu, na který ukazují ukazatele (zde je to int)
```

přístup k poli přes ukazatel pro Pole a prvni

suma += Pole[i];

suma += *(první + i);

pole – předávání do funkce a z funkce – může být problém při kombinacích [] a * například pole[] na zásobníku – krátký relativní od zásobníku, ukazatel – dlouhý od počátku, nemusí dojít ke správné konverzi – dlouhý ukazatel na krátký od zásobníku

Zjištění velikosti pole

- pro zjištění velikosti typu se používá sizeof
- pokud je parametrem sizeof pole, vrací velikost pole, pokud je parametrem ukazatel, vrací velikost ukazatele (tj. adresy)

Co se stane v následujícím případě? Předává se pole jako pole nebo pomocí ukazatele?:

```
int Test(int pole[ ]) // popřípadě (int pole[10]) či (int *pole)
{
  int i;
  pole[0] = 10; // dojde ke změně hodnoty i mimo funkci?
  i = sizeof(pole);  // i nabývá hodnoty  ???
  return i;
}
```

```
int  Pole[20]= {1},i;
i = sizeof(Pole); // i nabývá hodnoty ??? pole[0] nabývá hodnoty ???
i = Test(Pole); // i nabývá hodnoty ??? pole[0] nabývá hodnoty ???
```

Řetězec – string

- řetězcem je míněno pole typu char, jehož posledním znakem je znak ´\0´ (nula)
- koncový znak je důležitý, protože s ním pracují všechny funkce pro řetězce
- pro práci s řetězci existuje knihovna funkcí (existuje také knihovna pro string jako struktura, třída – těm se zde nevěnujeme)
- s řetězcem se pracuje jako s běžným polem

definice pole dané délky s inicializací pomocí znaků (použita pouze část pole):
char Ret1[10]= {´Z´,´d´,´a´,´r´,´\0´};

definice pole pomocí řetězce. Znak ´\0´ je přidán automaticky. Délku určí překladač tak aby se řetězec (včetně ukončovacího znaku) vešel:
char Ret2[ ]= “Zdar“;

Ret2 = “Zdar“; // nelze - mimo definici již musíme s polem pracovat po prvcích

Ret2[0]=´Z´; Ret2[1]=´d´; Ret2[2]=´a´; Ret2[3]=´r´; Ret2[0]=´\0´;

Knihovní funkce pro práci s řetězci

- jsou v knihovně string.h
- většina funkcí nekontroluje, zda je v cílovém řetězci dostatek místa (!!!)
- strlen(ret) vrací délku **bez** ukončovacího znaku (!!!)
- strcpy(cil, zdroj) – kopíruje řetězec včetně ukončení
- strcat(cil,zdroj) – připojí zdroj za původní řetězec
- strcmp(ret1,ter2) – porovná dva řetězce podle „velikosti“ (výsledek -1,0,1)
- strchr – hledá první výskyt znaku v řetězci
- strstr – vyhledá podřetězec v řetězci
- strncpy(cíl, zdroj, počet), strncat, strncmp – pracuje jen s uvedeným počtem znaků

Předchozí funkce pracují s typem char, který je dán hodnotou 0-255. Znaky definuje tzv. ASCII tabulka.

Pro UNICODE knihovna uchar.h, wchar.h.
Rozšířený bitový typ char16_t, char32_t.

Alokace paměti – dynamické proměnné - motivace

- standardní automatické lokální proměnné se vytváří na zásobníku. Jelikož zásobník mívá často omezenou velikost (délku), může (zvláště při použití polí) dojít k jeho přetečení
- proměnné, se kterými se pracuje, mívají různou délku pro různé okamžiky (texty, vyhodnocení dat …)
- z předchozího vyplývá nutnost (vhodnost) práce s daty, které dynamicky mění svou velikost. Především u polí se tento přístup jeví jako nutnost.
- Pro práci potřebujeme mít možnost požádat o paměť (alokace zdroje), a vrátit paměť (dealokace, odalokování zdroje). Každá alokace by měla mít svou odalokaci – v C není automatický mechanizmus, musí zařídit programátor.

Alokace paměti – dynamické proměnné

- je možné alokovat paměť o dané velikosti bytů – která nemusí být konstantní. Velikost se určuje až za chodu programu.
- velikost typu je nutné "zjistit" pomocí klíčového slova sizeof(typ)
- jelikož se alokuje "univerzální" blok paměti, je návratová hodnota typu *void a je nutné ji přetypovat
- paměťový blok je nutné uvolnit
- kontrola mezí není prováděna

```
#include <stdlib.h>

int *pi = NULL;

pi = (int *) malloc (počet * sizeof(int)); // vrátí ukazatel na počátek bloku paměti

if (pi != NULL) free(pi);  // odalokuje paměťový blok daný ukazatelem
```

Napište funkci, která vrátí první větu (kopii) textu pomocí dynamického řetězce. Statické pole nelze vracet.

```
char Text[]="Bylo jaro. Ale venku byla stále zima.";
char *Vysledek ;

Vysledek = VratPrvniVetu(Text);
```

```
char Text[]="Bylo jaro. Ale venku byla stále zima.";
// délka a ukončovací znak se doplní překladačem
char *Vysledek  = NULL; // proměnná pro uložení výsledku - inicializovat

Vysledek = VratPrvniVetu(Text); // volání funkce
// – parametrem je ukazatel na počátek textu
// - vrací se ukazatel na řetězec, který obsahuje kopii textu první věty

if  (Vysledek) // != NULL => byl naplněn => byl alokován
 {
   free(Vysledek) ; // vrátí se paměť, která již nebude využívána
Vysledek = NULL; // signál pro případné další použití  => nealokováno
}
```

```
// funkce vracející první větu z textu předaného jako parametr
// vrací ukazatel na nově naalokovaný blok paměti
char * VratPrvniVetu(char text[])
{
  int delka=0,i;
 char *vyslret=NULL;

for(delka=0;text[delka]!=´\0´;delka++) // zjištění délky první věty
  if(text[delka]==´.´)
   {
      delka++; break;
   }
vyslret=(char*) malloc((delka+1) * sizeof(char)); // alokace paměti pro kopii textu
for(i=0;i<delka;i++)    // kopírování první věty
   *(vyslret+i) = text[i];
*(vyslret+delka) =´\0´;
 return vyslret;
}
```

Životnost proměnných v čase

- lokální proměnné existují a je možné je použít pouze v rámci bloku, kde jsou definovány
- statické (lokální) proměnné, je možné použít pouze v rámci bloku, kde jsou definovány. Jsou však uloženy v globální části paměti, existují stále.
- globální proměnné – jsou přístupné v rámci celého programu. Jsou uloženy v globální části paměti, existují stále.
- alokované zdroje (vracené přes ukazatel jako otevřené soubory, alokovaná paměť …) existují, dokud je nevrátíme (zavření souboru, vrácení paměti …). Přístupné jsou pouze do doby, dokud si pamatujeme jejich adresu – ztratíme-li adresu, nezanikne získaný zdroj (soubor je stále otevřen, paměť zabraná …) ale pouze vazba na ni
- Připomenutí: (stejně jako u jednorozměrných polí) pokud provedeme přiřazení polí, nepřiřazuje se obsah, ale přiřazují se ukazatele (pokud přiřazujeme do lokální proměnné, oznámí překladač chybu, protože je to konstantní hodnota. Pokud přiřadíme dynamické proměnné, potom oba ukazatele ukazují do stejného prostoru. Pokud ukazatel vlevo byl jediný, který ukazoval na platná data, potom k nim po přiřazení ztratíme přístup.

Čtení složitých definicí

- začínáme od názvu proměnné
- pokud je to možné postupujeme doprava, jinak doleva
- ( ) mají dvojí význam. První vyznačuje prioritu (na pravé závorce končíme a čteme doleva od názvu proměnné). Druhý význam je, že značí oblast parametrů funkce -> čteme "je funkce s parametry"
- * (v definici proměnné) čteme jako "ukazatel"
- [] čteme jako "pole"

Vícerozměrná pole

- díky rozdílné definici/interpretaci je nevhodné jednotlivé typy při práci „míchat“. Jednotlivé typy se mezi sebou nekonvertují (lépe kopírovat ve funkcích)
- realizována jako pole polí
  int xxx[2][10] xxx je pole obsahující, dvě pole obsahující 10 prvků typu int
- pomocí ukazatelů
  int **yyy ; yyy je ukazatel, (který ukazuje) na ukazatel, (který ukazuje) na int
- jako kombinace polí a ukazatelů
  int *zzz[2]; zzz je pole obsahující dva ukazatele na int
  int (*www)[2]; www je ukazatel na pole obsahující dva inty

Pozn.: jelikož každý ukazatel se dá "oindexovat", je možné každý ukazatel chápat i jako "pole"
Pozn.: díky rozdílné definici/interpretaci jednotlivých polí je nevhodné s nimi pracovat "společně" – například je různě přiřazovat, předávat do funkcí na místě jednoho parametru …

Realizace pomocí pole polí

double Pole2a[2][3];

- v paměti zabírá lineární prostor.
- "posun" o řádek realizován překladačem, který "zná" délku řádku.
- pole nelze modifikovat – pevné rozměry, obdélníkový tvar

rozložení pole v paměti

[0][0]	[0][1]	[0][2]	[1][0]	[1][1]	[1][2]

grafické znázornění pole

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]

double Pole2a[2][3]={{1,2,3},{4,5,6}}; // inicializace pole v definici
Pole2a[0][0] = Pole2a[1][2]; // přístup k prvkům pomocí pole
double *radek = &Pole2a[1][0]; // zjištění adresy řádku
double *radek1 = &Pole2a[1]; // zjištění adresy řádku
*(Pole2a + delka_radku *i +j) = 3; // ekvivalent Pole2a[i][j]=3; nutný lineární přístup

Realizace pomocí pole ukazatelů

double *Pole2b[2];

- v paměti zabírá lineární prostor ukazatelů (na pole doublů, které je nutné vytvořit (definovat pole, naalokovat) a přiřadit – tato pole zabírají další lineární prostory v paměti, které na sebe obecně nenavazují)
- první index vybere ukazatel, který je oindexován druhým indexem (vybere prvek v poli)
- pole má pevný počet řádků. Počet prvků v řádku je libovolný, pro jednotlivé řádky může být různý

```
double Pole1[3] = {1,2,3}, *Pole2;
double *Pole2b[2];
Pole2 = (double*) malloc(5*sizeof(double));
Pole2[0]=4;      Pole2[1]=5;      Pole2[2]=6;      Pole2[3]=7;      Pole2[4]=8;
Pole2b[0] = Pole1;       Pole2b[1]= Pole2;
Pole2b[0][0] = *(*(Pole2b+1)+1); // oindexovat ukazatel a přistoupit. První výsledek
// je ukazatel, druhý double
if (Pole2) free(Pole2);
```

Nakreslete paměťovou mapu.

Realizace pomocí ukazatelů na pole

double (*Pole2c)[2];

- v paměti zabírá prostor na jeden ukazatel. Do toho je nutné naalokovat lineární prostor pro celé pole
- první index vybírá řádek (vypočte ho překladač díky znalosti délky řádku plynoucí z druhého indexu), druhý index vybere prvek v řádku.
- pole má pevný počet prvků v řádku. Může mít libovolný počet řádků.

```
double pole [2][3]={1,2,3,4,5,6};
double (*pp)[2];
double (*pp)[3]=(double(*)[3])pole; // použití existujícího pole
pp = (double(*)[3]) malloc (2*3*sizeof(double)); // alokace pole nového
pp[0][0] = 0;pp[0][1] = 01;pp[0][2] = 2;pp[1][0] = 10;pp[1][1] = 11;pp[1][2] = 12;
*(*(pp+1)+1) = 11;
if (pp) free (pp);
```

Nakreslete paměťovou mapu.

Realizace pomocí ukazatele na ukazatel

double **Pole2d

- v paměti zabírá místo na jeden ukazatel. Je nutné naalokovat (a sem uložit) pole ukazatelů (lineární prostor – lze si ho představit jako vertikální pole, každý ukazatel pak ukazuje na "řádek" matice). Do každého ukazatele je nutné naalokovat řádek prvků (double)
- první index vybere ukazatel v poli ukazatelů (ukazující na řádek), druhý index oindexuje řádek (vybere prvek v daném řádku)
- počet řádků může být proměnný. Řádky mohou být různě dlouhé, lze je měnit.

double **Pole2d=NULL;
Pole2d = (double **) malloc(2 * sizeof(double*));
if(Pole2d == NULL) -> chyba
Pole2d[0]=(double*) malloc(3*sizeof(double*));
Pole2d[1]=(double*) malloc(6*sizeof(double*));
if(Pole2d [0]== NULL) {free(Pole2d); -> chyba }
if(Pole2d [1]== NULL) { free(Pole2d[0]);free(Pole2d); -> chyba}
Pole2d[1] [0]=4; Pole2d[1] [1]=5;Pole2d[1] [2]=6; Pole2d[1] [3]=7; Pole2d[1] [4]=8;
Pole2d[0] [0]=*(*(Pole2d+1)+1); Pole2d[0] [1]=2; Pole2d[0] [2]=3;
free(Pole2d[1]); free(Pole2d[0]); free(Pole2d); Pole2d = NULL;

Na základě předešlého příkladu (Nakreslete si jeho paměťovou mapu.) napište kód (nejdříve v main, potom přesuňte do funkcí) na alokaci a uvolnění pole daných rozměrů (s rozměry jako parametrem a použitím v cyklech). Proměnné pole inicializujte na nulu.
Nakreslete paměťovou mapu.

Struktura (union)

- struktura a union jsou složené typy, které "v sobě" mohou obsahovat více proměnných
- struktura obsahuje v každém okamžiku všechny své proměnné, union obsahuje (může být "aktivní") pouze jednu. Ve struktuře jsou proměnné "za sebou" a tudíž má rozměr rovnající se (minimálně) součtu rozměrů jednotlivých proměnných (někdy se vkládají mezi proměnné mezery, aby byly proměnné zarovnány na hranici – například adresa dělitelná 4,8,16…). V unionu jsou proměnné "přes sebe" a má rozměr největší z nich. Je možné přistupovat ke všem proměnných, pouze jedna má však smysl pro čtení (ta naposled naplněná – která to je se musí pamatovat jinde).

struktura

adresa	typ proměnné	název proměnné
200	int	ii
204	char [16]	cc
214-21c	double	dd

union

adresa / proměnná	int ii	char [16] cc	double dd
200-204	xxx	xxx	xxx
205-208	N	xxx	xxx
208-216	N	xxx	N

N = nevyužito

Definice struktury

- je možné rozdělit na popis struktury a definici proměnné
- popis struktury je možný dvěma způsoby – oznámení jména a kompletní popis složek struktury – netvoří kód -> umísťujeme do hlavičkového souboru
- kód se tvoří podle popisu struktury při definice proměnné
- celý/používaný název typu se skládá z klíčového slova struct a jména typu

```
extern struct Jmeno; // oznámení jména, lze použít pouze ukazatel na tento typ

struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err;
};                // konec definice (nezapomínat na středník)
//  zde lze definovat proměnnou

struct SKomplex aa, *paa; // definice proměnné (vyhradí paměť minimálně na
// 2x double a 1x int); definice ukazatele (vyhradí paměť na adresu)
struct Jmeno *pjj; // pouze ukazatel, proměnná není možná – není známa velikost
```

Operátory přístupu . a -> (*).

- pro přístup k vnitřním proměnným se používá operátor tečka
- pokud máme ukazatel na strukturu, používáme k přístupu operátor ->

```
struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err;
};

struct SKomplex aa, *paa; // definice proměnné a ukazatele

paa = &aa; // ukazatel je nutné inicializovat

if (!aa.err) // není-li proměnná v chybovém stavu
    aa.Re = aa.Im * aa.Im;
if (! (*paa).err)  // díky nižší prioritě „.“ před * je nutné použít ( )
    paa-> Im = paa->Re; // přístup k prvkům pomocí ukazatele na strukturu
```

V komplexní proměnné podle předchozí definice zaměňte Re a Im složky.
Předveďte pro proměnnou i ukazatel na ni.

```
struct SKomplex { // jméno struktury
double Re, Im;        // seznam proměnných
int err; };

struct SKomplex aa, *paa; // definice proměnné a ukazatele
double pom;
paa = &aa; // ukazatel je nutné inicializovat
if (!aa.err) // výměna pro proměnnou typu struktura
 {
   pom = aa.Re;
    aa.Re = aa.Im;
   aa.Im = pom;
 }
if (! (*paa).err)  // výměna pro ukazatel
{
   pom = paa->Re;
    aa->Re = aa->Im;
   aa->Im = pom;
 }
```

Mechanizmus přiřazení struktur a definice s inicializací struktury strukturou

- pro přiřazení struktury do struktury se používá mechanizmus kopie paměťového bloku. Toto je výhodné, pokud nejsou vnitřní proměnné typu ukazatel. Pro ukazatel dojde k tomu, že nyní dvě struktury mají stejný ukazatel a tedy „vlastní“ společnou paměť (aniž by o tom měly nějaký signál).
- tento typ se nazývá mělká kopie, protože nejde do hloubky (nedělá kopii hodnot, na které se ukazuje)
- při přiřazení dvou ukazatelů se kopíruje adresa a ukazují na stejný objekt

```
struct SKomplex aa, *paa,*pbb;
struct SKomplex bb = aa; // provede se kopie paměťového bloku aa do místa bb
paa = (struct SKomplex*) malloc(sizeof(struct SKomplex));
pbb = &pb;
aa = bb; // provede se kopie paměťového bloku bb do místa aa

paa = pbb;  // kopíruje se hodnota, kterou je ukazatel, struktura se nekopíruje
// naalokovaná proměnná se „ztratila“ a již na ni není odkaz
free(paa); // neruší alokovanou proměnnou, ale proměnnou pb => chyba
```

Struktura jako návratová hodnota a parametr fce

- platí stejná pravidla jako pro ostatní typy
- buď se předávají hodnotou (nevhodné – velká paměťová náročnost) nebo ukazatelem -> pokud můžeme, dáváme přednost ukazateli. Nejde-li ukazatel, použijeme hodnotu.
- struktura jako návratová hodnota je nutná když se předává nová proměnná. Pokud předáváme zpět proměnnou, která byla parametrem, můžeme předat ukazatel.

Napište funkci pro součet dvou komplexních čísel.
Napište funkci pro návrat komplexního čísla s větší vzdáleností od počátku.

```
// součet dvou komplexních čísel – vzniká nová proměnná – návratem je  hodnota
struct SKomplex Soucet(struct SKomplex *p1, struct  SKomplex const *const p2)
//ukazatele u argumentů místo hodnot šetří místo (rozměr struct proti ukazateli)
{ // aby nedošlo k nechtěné změně struct ve funkci -> modifikátor const
 struct SKomplex ret; // vytvoří se pomocná proměnná
ret.Re  = p1->Re + p2->Re;
ret.Im  = p1->Im + p2->Im;
 return ret; // na základě hodnoty pomocné proměnné se naplní návratová hodnota
}

// vrácení „většího“ – vrací se jeden z parametrů => ukazatel
struct SKomplex * Delsi(struct SKomplex *p1, struct SKomplex *p2)
{double d1,d2;
  struct SKomplex *pret; // pomocná proměnná, hodnotu lze vrátit i bez ní
  d1 = sqrt(p1->Re* p1->Re+ p1->Im* p1->Im); //  výpočet délky
  d2 = sqrt(p2->Re* p2->Re+ p2->Im* p2->Im);
  pret = d1>d2 ? p1:p2; // zápis příslušné adresy do výsledné hodnoty
  return pret; // vrácení adresy většího prvku
}
```

nedynamické a dynamické proměnné typu struktura

- proměnnou typu struktura je možné získat i dynamicky (musí se chovat jako standardní typy)
- Je možné také „požádat“ o pole struktur

```
struct SKomplex aa,*paa, *pap;

paa = (struct SKomplex*) malloc(sizeof(struct SKomplex)); // jedna struktura

pap = (struct SKomplex*) malloc(20 * sizeof(struct SKomplex)); // pole 20-ti struktur
// struktury jsou lineárně za sebou v paměti

paa->Re = aa.Im; // „klasický“ přístup k vnitřním proměnným u jednoho prvku
pap[10].Re = 8; // přístup k prvku pole –
// výsledek pap[X] není ukazatel, ale prvek <=> *(pap+X)
```

Práce se strukturou obsahující dynamickou proměnnou.
Hlavičkový soubor:

```
struct SText {char *txt;int delka;}; // delka neni nutna, ale zrychlí program
// txt je ukazatel, po definici proměnné neinicializovaný

#define INIT_TEXT(struc, text,del) struc.txt = malloc((del+1)*sizeof(char)); \
strncpy(struc.txt,del+1,text);struc.del = del+1;
#define DESTR_TEXT(struc) if (struc.txt) free(struc.text); struc.text=NULL; \
struc.delka=0;

#define INIT_UKTEXT(pstruc, text,del) pstruc= malloc(sizeof(SText)); \
INIT_TEXT(*pstruc,text,del);
#define DESTR_UKTEXT(pstruc) if (pstruc) { DESTR_TEXT(*pstruc); \
free(pstruc); }; pstruc = NULL;
```

Zdrojový soubor:

```
struct SText ee,*pee;
INIT_TEXT(ee,10,“ahoj“);          INIT_UKTEXT(pee,10,“zdar“);
DESTR_TEXT(ee);                   DESTR_UKTEX (pee);
```

Využití struktur

- s výhodou se využívají při tvorbě lineárních seznamů a stromů (a typů odvozených)
- základní vlastností těchto typů je, že kromě dat obsahují i ukazatel(e) na stejný typ jako jsou sami.
- lineární seznam obsahuje ukazatel na další prvek (poslední prvek má NULL). Modifikacemi jsou cyklické seznamy, které nemají poslední prvek. Obousměrně vázaný seznam má vazbu nejen na následující, ale i na minulý prvek. S jejich pomocí lze realizovat Frontu (FIFO), Zásobník (stack, LIFO), pole ...
- při realizaci stromu struktura s daty tvoří uzel a obsahuje dva (nebo více) ukazatele, které ukazují na uzly následující (binární (vážené) stromy, příkladem může být třeba strom pro morseovku – v uzlu je aktuálně dosažený znak, ukazatele ukazují na znak, na který se přesuneme, přišla-li tečka resp. čárka).

Union

- přístup je stejný jako u struktury
- rozdíl je v tom, že může obsahovat pouze jednu proměnnou

```
union UBarva {
long barva; // long má 4 byty
char slozky[4];
}

union UBarva uu;

uu.barva = 0x11223344; // po přiřazení se zároveň vyplní i pole char slozky[4]
// v závislosti na uložení long v paměti budou složky nabývat hodnot:
```

složka	little endian	big endian
uu.slozky[0]	44	11
uu.slozky[1]	33	22
uu.slozky[2]	22	33
uu.slozky[3]	11	44

Bitové pole

- podobá se struktuře, obsahuje pouze celočíselné proměnné, u kterých je možné určit jejich počet bitů
- lze použít i k „překrytí“ bitů jmény

```
struct DATUM {
    unsigned den : 5; // počet bitů dostačující pro uložení čísla 31 (max počet dnů)
    unsigned mesic : 4;
    unsigned rok : 7;
};

    struct DATUM dnes, zitra;
    dnes.den = 6;
    dnes.mesic = 1;
    dnes.rok = 1995 - 1980;
    zitra.den = dnes.den + 1;
```

využití ke čtení bitů v registru

```
struct FLAG_REGISTR {
unsigned NULA:1;
unsigned PRENOS:1;
unsigned PRENOS2:1;
unsigned NEVYUZITO:1;
unsigned VETSI:1;
unsigned MINUS:1;
unsigned NEVYUZITO:1;
unsigned NEVYUZITO:1;
}

union FlagReg{
char hodnota;
struct FLAG_REGISTR reg;};

union FlagReg;
FlagReg. hodnota = LoadRegA();
if (FlagReg.reg.NULA) ....
```

Ukazatel na funkci

- jméno funkce je zároveň ukazatelem na vstupní bod funkce – adresa pro volání
- ukazatelů na funkci se využívá při předávání do funkcí, které si naleznou parametry a mají s nimi něco vykonat, kdy nalezení je složité a akce se mění
- dále je možné je používat pro callback funkce
- využívají se i ve složitějších strukturách pro volání funkcí, které jsou s nimi spojeny

```
// využití v hlavičce funkce a ukázka použití v těle funkce
double Vypocet(int cc, float dd, float (*pf)(int , double)){return  (*pf)(cc ,dd); }
float secti(int a, double b) {...}
float odecti(int a, double b) {...}

float (*pff)(int , double); // viz. čtení složitých definicí
// pff je ukazatel, na funkci s dvěma parametry – int a double, vracející float
if (input == ´+´)
    pff = secti;
else
   pff = odecti; // nebo lze vysl = Vypocet(xx,yy,odecti);
vysl = Vypocet(xx,yy, pff);
```

Typedef

- umožňuje nadefinovat nový typ
- nový typ je odvozen od typu stávajícího a je z/na něj konvertovatelný
- používá se v případě, že chceme některý typ měnit, například máme řadu funkcí, které mají pracovat pouze s jedním typem, ale až při překladu se chceme rozhodnout s jakým
- dále je výhodné je použít při složitých definicích jako například při použití ukazatelů na funkce

typedef NOVY_TYP int; // vytvořili jsme nový typ NOVY_TYP odvozením z int

NOVY_TYP funkce(NOVY_TYP aa, NOVY_TYP *bb) // použití jako u jiných typů
{...; return aa;}

Volatile

- modifikátor, který v definici proměnné upozorňuje překladač, že proměnná může být asynchronně (na pozadí) měněna (například v přerušení)
- takováto proměnná nemůže být optimalizována/ukládána do registru, protože druhý proces ji mění v paměti a tak musí být vždy používána z paměti

```
volatile int flag = 1; // definice proměnné

// uvnitř přerušovací rutiny
flag = 0; // nastavení, že přišla událost na kterou se čeká

ve vlastním programu

while(flag); // nekonečný cyklus.
// Ukončí ho přerušení, které změní hodnotu proměnné.
//pokud bude proměnná v registru, přerušení ji změní v paměti, k ukončení nedojde
```

Restrict

- modifikátor spojený s definicí ukazatele
- tento modifikátor říká, že daný ukazatel je jediný „aktivní“ přístup k daným datům v daném okamžiku.
- prototyp funkce informuje uživatele, jaké relace mají mít paměťové bloky. Je-li uveden restrict, potom to například znamená, že se nekontrolují kolize zapříčiněné překrytím zdrojového a cílového prostoru a s tím spojeným přepisováním dat.
- zároveň se jedná se o informaci, že přístup k datům může překladač provádět libovolně (optimalizovat na rychlost a bez kontroly kolizí)
- splnění podmínky je věcí programátora a není kontrolováno

int memory_move(restrict char *sour, restrict char *dest)
definice říká, že sour a dest jsou jediné, které ukazují na své data. Jinými slovy ale také, že blok dat daný sour a dest se nepřekrývá a není tedy nutné řešit přesun tak, aby nedošlo k překrytí dat, která budeme ještě potřebovat, daty přesouvanými.

_Bool

- knihovnou definovaný typ pro logické proměnné společně s konverzemi
- knihovna stdbool.h

_Complex

- knihovna pro práci s komplexními čísly včetně nejčastěji používaných funkcí
- complex.h