A (max. 4b): Nezáporné celé číslo N (N < 1 000 000) v desiatkovej sústave je zapísané v reťazci znakov (**char**-ov). Napíšte funkciu v jazyku C, ktorá prevedie ľubovoľné celé číslo zapísané v reťazci na **int**. Ako príklad uveďte reprezentáciu reťazca v pamäti pre číslo 12345. Okrem správnej implementácie tela funkcie je potrebné napísať aj hlavičku funkcie – správne uviesť vstupné a výstupné argumenty.

Riešenie:

Najskôr si uvedomíme ako vyzerá číslo zo zadania v reťazci. Reťazec obsahuje znaky zakódované v ASCII.

'1'	'2'	'3'	'4'	'5'	0

Teda, obsahuje nasledovné číselné (ASCII) kódy:

49	50	51	52	53	0

Úlohou je napísať funkciu, ktorá takéto číselné kódy prevedie do jednej premennej typu **int**. Zadanie týmto jasne určuje čo bude vstupom do funkcie (**reťazec znakov**) a čo bude výstupom funkcie (**hodnota int**). Hlavička funkcie bude preto vyzerať nasledovne:

```
int prevod(char *str);
```

V ďalšom kroku sa musím rozhodnúť akým postupom (algoritmom) budem takéto reťazce znakov prevádzať do jedného čísla **int**. Keďže programovací jazyk C umožňuje pracovať len s jednoduchými dátovými typmi, nie je možné urobiť prevod nejako „čarovne“ jednou operáciou. Budem to teda musieť vykonávať postupne. Jednoduché dátové typy, ktoré v tomto probléme vystupujú sú znaky (**char**), obsahujúce číslice (nula až deväť – teda ASCII kódy 48 až 57), a výsledok ako **int**. Algoritmus musí teda tieto znaky postupne pripočítavať do premennej, v ktorej bude vznikať výsledné číslo.

Reťazec môžem spracúvať od začiatku alebo od konca. Zvolím si napríklad spracovanie od začiatku, v prípade ukážkového reťazca "12345" je prvý znak '1', druhý znak '2'. Teda po spracovaní prvého znaku musím mať vo výslednej premennej 1, po spracovaní druhého znaku už 12.

Algoritmus bude preto veľmi jednoduchý, spracovanie každého ďalšieho znaku Z bude znamenať vynásobiť doterajší výsledok V hodnotou 10 a pripočítať k nemu číslicu zo znaku Z. Teda napr. pre doterajší výsledok V=12 a tretí znak Z='3' (ASCII kód 51) to bude: V*10+Z-'0' = 12*10+(51-48) = 120+3 = 123.

Požadovaná funkcia bude teda vyzerať nasledovne:

```
int prevod(char *str)
{
  int i, k = 0;
  for (i = 0; str[i] > 0; i++)
    k = 10 * k + (str[i] - '0');
  return k;
}
```

Ukážkové použitie v hlavnej funkcii môže byť nasledovné:

```
int main(void)
{
  char *cislo = "12345";
  int x = prevod(cislo);
  printf("%d\n", x);
  return 0;
}
```

B (max. 1b): Cyklus je typ príkazu, ktorý sa vykonáva opakovane. Existuje taký cyklus, ktorého telo sa nevykoná ani raz? Ak áno, napíšte príklad v jazyku C. Existuje taký cyklus, ktorý sa bude vykonávať donekonečna? Ak áno, napíšte príklad v jazyku C.

Riešenie:

Samozrejme existujú oba typy cyklov, závisí to od podmienky s ktorou cyklus napíšeme.

Najjednoduchší cyklus, ktorý sa nevykoná ani raz je tvaru **while** s podmienkou ktorá je neplatná, a najjednoduchšia neplatná hodnota je číslo 0:

```
while(0)
  printf("Nevykona sa.\n");
```

Alternatívne, cyklus tvaru **for**, ktorý sa nikdy nevykoná, musí mať podmienku, ktorú nespĺňa pri prvom volaní cyklu pri behu programu. Napr.:

```
int i;
for(i = 0; i < 0; i++)
  printf("Nevykona sa.\n");
```

V prípade nekonečných cyklov to je už komplikovanejšie. Musíme zabezpečiť, aby podmienka bola vždy platná, pričom v rámci tela by nemalo byť prerušenie break, return, exit. Najjednoduchšia vždy platná hodnota je 1 alebo hocijaká iná NEnulová (napr. -47). Cyklus tvaru **while**:

```
while(1)
  printf("Vypisuje do nekonecna!\n");
```

alebo tvaru **for** bez podmienky, teda sa aj telo cyklu opakuje bez obmedzenia:

```
for(;;)
  printf("Vypisuje do nekonecna!\n");
```

viaceré nesprávne odpovede obsahovali podmienku obsahujúcu nejaký výraz s premennou, ktorá sa v rámci cyklu menila (napr. i++). V takom prípade však treba dbať na obmedzený rozsah dátového typu priebehovej premennej, ktorej hrozí po veľmi vysokom počte zvýšení pretečenie. Napr. nasledujúce cykly NEBUDÚ vykonávať telo donekonečna ako by sa mohlo zdať:

```
for(i = 0; i >= 0; i++) // nespravne
  printf("NEvypisuje do nekonecna!\n");
```

v prípade, že i je typu **int**, tak inkrementovaním hodnoty 2,147,483,647 sa i dostane na hodnotu −2,147,483,648, ktorá je však už záporná a cyklus skončí.

```
for(i = 0; i < i+1; i++) // nespravne
  printf("NEvypisuje do nekonecna!\n");
```

podobne, ako v predchádzajúcom prípade, ak i= 2,147,483,647 tak i+1 je záporné číslo, a podmienka i < i+1 prestane platiť.

```
for(i = 1; i = i; i++) // nespravne
  printf("NEvypisuje do nekonecna!\n");
```

V tomto prípade pretečenie nevadí, ale vznikne tzv. obtočenie, teda po pretečení premennej i do záporných čísel sa opäť dostane na hodnotu 0, čím sa podmienka vyhodnotí ako nesplnená (resp. priradenie bude mať hodnotu 0 – teda nepravda ak to interpretujeme ako podmienku) a cyklus skončí.

C (max. 2b): Čo vypíše nasledujúci program v jazyku C? Odpoveď zdôvodnite.

```
char *hraj(char *s)
{
      if (s == "kamen")
            return "papier";
      if (s == "noznice")
            return "kamen";
      if (s == "papier")
            return "noznice";
      return "?";
}
```

```
int main(void)
{
      char *x = "kamen";
      int i;
      for (i = 0; i < 47; i++)
            x = hraj(x);
      printf("%s\n", x);
}
```

Riešenie:

V tejto úlohe sa jedná o jednoduchý prechod medzi stavmi: kamen, papier, noznice. Premenná **x** môže nadobúdať viaceré tri možné hodnoty. Na začiatku je nastavená na hodnotu "kamen". Volanie: x = hraj(x) premennú upraví podľa návratovej hodnoty funkcie hraj. Je to klasická hra kameň-papier-nožnice. Po prvom volaní hraj (keď i je 0) teda bude mať x hodnotu "papier", po druhom "noznice" a po treťom opäť "kamen". Takto sa to bude ďalej opakovať, a keďže cyklus sa vykoná celkovo 47 krát (pre i=0,1,...,46) tak konečná hodnota x bude "noznice", pretože zvyšok po delení 47 číslom 3 je 2.

Viacerých z vás splietlo porovnávanie reťazcov, ktoré bolo v úlohe použité, keďže zvyčajne sa reťazce porovnávajú po znakoch (napr. funkciou strcmp). Situáciu ste potom vyhodnotili tak, že žiadne porovnanie nemohlo úspešne prebehnúť a preto x bude mať na konci výpočtu hodnotu "?", čo však nie je správne. Pri takýchto (tzv. statických) reťazcoch, ktoré sú v zdrojovom kóde napevno zadefinované kompilátor pri výpočte priradí rovnaké bloky pamäte, a teda smerník (resp. jeho hodnota = celé číslo) na reťazec "kamen" vo funkcii hraj je rovnaký ako smerník vo funkcii main, a rovnaký ako aj návratová hodnota return "kamen", a preto je možné tieto smerníky (celé čísla) porovnávať obyčajných operátorom porovnania ==. Vyskúšajte si to, naozaj to tak funguje.

D (max. 3b): Zakrúžkujte správnu možnosť v každom z nasledujúcich výrokov o programovaní v jazyku C. Vždy je správna práve jedna možnosť. Hodnotenie: 0.5b za každú správnu odpoveď, -0.5b za nesprávnu odpoveď, 0 bodov ak neodpoviete. Ak bude celkový súčet záporný, za úlohu sa udelí 0 bodov.

Riešenie:

~~Platí~~ / Neplatí ... Rekurzia znamená, že tvoj program môže bežať donekonečna.

Tvrdenie neplatí, pretože rekurzia označuje situáciu, keď sa v tele funkcie vyskytuje volanie tej istej funkcie; týmto sa môžu volania funkcií vnárať. Pojem rekurzia nesúvisí s nekonečnými volaniami.

Platí / ~~Neplatí~~... Rozsah platnosti identifikátora nám presne určuje, kde v zdrojovom kóde môžeme identifikátor používať, a s ktorou pamäťou je previazaný.

Rozsah platnosti identifikátora z definície určuje rozsah miest, kde môžeme identifikátor použiť, a k tomu je nutné vedieť s akou pamäťou je previazaný (lebo inak by sme jeho hodnotu nevedeli prečítať a teda ani použiť).

Platí / ~~Neplatí~~ ... Každý cyklus typu **for** je možné prepísať na cyklus typu **while**, a platí to aj opačne**.**

Tvrdenie platí, **for** alebo **while** majú rovnakú vyjadrovaciu silu, použitie konkrétneho tvaru cyklu závisí od vhodnosti alebo preferencií programátora.

Každý **while** cyklus v tvare

```
while (podmienka)
{ telo; }
```

prepíšeme na cyklus v tvare **for** nasledovne

```
for (; podmienka; )
{ telo; }
```
Podobne každý **for** cyklus v tvare

```
for (inicializacia; podmienka; krok)
{ telo; }
```

vieme prepísať na ekvivalentný **while** cyklus

```
{
  inicializacia;
  while (podmienka)
  { telo; krok; }
}
```
Všimnime si v tomto prepise použitím **while** cyklu, že inicializácia a celý **while** je obalené vonkajším blokom (zložené zátvorky {}), ktorý ohraničuje rozsah platnosti premenných vytvorených pri inicializácii (napr. **int i**) v rámci bloku, aby nepokračovali existovať po skončení prepísaného **while** cyklu, pretože neexistovali ani po skončení pôvodného **for** cyklu.

Platí / Neplatí ... Dátový typ **double** sa používa na prácu s reálnymi číslami.

Tvrdenie je nejasné, a teda obe možnosti sme akceptovali ako správne. Je pravda, že sa **double** používa keď chceme pracovať s reálnymi číslami, ale na druhej strane ak požadujeme absolútnu presnosť reálnych čísel, nie je možné **double** použiť, pretože má len obmedzenú presnosť, ktorá je vhodná len pre DESATINNÉ čísla s určitou presnosťou. Reálne čísla sú matematicky príliš perfektné, a v počítači ich nie je možné efektívne reprezentovať (pretože určitý typ reálnych čísel tzv. transcendentné čísla majú nekonečný desatinný rozvoj, s ktorým súčasné počítače ani teoreticky nemôžu vedieť pracovať efektívne).

Platí / ~~Neplatí~~... Smerník je vo svojej podstate celé číslo.

Tvrdenie platí, smerník je adresa do pamäte a adresuje sa po celých byte-och, teda smerník je celé číslo – poradie od začiatku pamäte.

~~Platí~~ / Neplatí ... Zložený dátový typ (pole, štruktúra) zaberá v pamäti toľko byte-ov ako je súčet veľkostí dátových typov, z ktorých sa skladá.

Tvrdenie neplatí, pretože napr. štruktúry sú pre efektívnosť operácií s nimi zarovnané na veľkosť slova, a tiež jednoduché dátové typy v rámci štruktúry sú zarovnané podľa veľkosti dátového typu v byte-och. Napr. štruktúra postupne obsahujúca typy **int**, **char**, **short**[2], **int**, **char** zaberá v pamäti 20 byte-ov, napriek tomu, že samotne jednoduché dátové štruktúry zaberajú 4+1+2*2+4+1=14 byte-ov.

E (max. 3b): Dané je celé číslo N (N < 1 000 000), ktoré nekončí nulou. Napíšte funkciu v jazyku C, ktorá ako vstupný argument dostane N a vráti otočené číslo N, teda napr. pre N=123 vráti 321. Okrem implementácie tela funkcie je potrebné napísať aj hlavičku funkcie – správne uviesť vstupné a výstupné argumenty.

Riešenie:

Úloha spočíva v prevode celého čísla (ako **int**) na otočené číslo (tiež ako **int**). Túto operáciu je potrebné vykonať ako funkciu, bez výpisu, ktorý ste viacerí chybne realizovali. Budeme uvažovať podobne ako v úlohe A. Najskôr určíme hlavička funkcie:

```
int otoc(int x);
```

Rovnako ako v prípade úlohy A operáciu otočenia nie je možné realizovať čarovne v jednom kroku, ale musíme ju realizovať postupne po jednotlivých číslicach. Využitím delenia 10 a zvyšku po delení 10 vieme z čísla **int** určiť poslednú cifru. Algoritmus teda bude z jedného čísla postupne zisťovať číslice v postupnosti

od najnižšieho rádu (poslednej), pričom rovnakým spôsobom ako v úlohe A tieto číslice budeme skladaťdo jedného čísla, a teda prevedieme pôvodné číslo na nový číslo s obráteným poradím číslic. Napr. v prípade čísla 123 sú číslice od konca 3,2,1 a keď tieto číslice v tomto poradí postupne podľa algoritmu z úlohy A pripočítame spolu, dostaneme postupne výsledné čísla: 3, 32, 321. Teda z pôvodného čísla 123 dostaneme výsledné 321. Program vyzerá nasledovne:

```
int otoc(int x)
{
  int y = 0;
  while(x > 0)
  {
    y = y*10 + x % 10;
    x /= 10;
  }
  return y;
}
```

Číslice od konca dostaneme operáciou zvyšok po delení 10, pričom následne pôvodné číslo (x) predelíme 10, aby sme poslednú číslicu odstránili, a ďalej pokračujeme pre ďalšiu číslicu, atď.

Ukážka použitia v hlavnej funkcii:

```
int main(void)
{
	int x = 123;
	int y = otoc(x);
	printf("%d\n", y);
	return 0;
}
```

F (max. 2b): Čo vypíše nasledujúci program v jazyku C? Odpoveď zdôvodnite.

```
int i = -1, j = 0, k = 1;
if (i++ && !j || k++)
	j++;
if (i++ || j && !i || k++)
	j++;
printf("%d %d %d\n", i, j, k);
```

Riešenie:

Úloha vyžaduje odkrokovať výpočet krátkeho programu a zistiť, ktoré operácie inkrementovanie (zvýšenie hodnoty o 1) sa vykonajú pre určité nastavenie hodnôt premenných. Operátor ++ keď je uvedené ZA premennou znamená, že sa najskôr vyhodnotí (vráti) hodnota výrazu a až potom sa hodnota zvýši. Tiež, všimnime si, že podmienky obsahujú rozličné typy zloženej podmienky (logický súčin AND &&) a (logický súčet OR ||), pričom pri vyhodnocovaní sa uplatňuje tzv. skrátené vyhodnocovanie, keď vyhodnocovanie prerušuje hneď ako je jednoznačný celkový výsledok.

V prípade prvej podmienky:

```
if (i++ && !j || k++)
```

podmienka je vlastne jeden (veľký) OR dvoch menších výrazov (i++ AND !j) a (k++), vyhodnocujú sa postupne zľava doprava; ak niektorý z nich je TRUE, vyhodnocovanie celého OR sa končí ako TRUE, inak (ak je FALSE) pokračuje sa s vyhodnocovaním ďalšieho prvku.

V prvom výraze (i++ AND !j) do vyhodnotenie vstupuje hodnota i (-1, teda TRUE), ktorá sa následne zvýši na 0 a !j (negácia j, ktoré je 0) čo je TRUE, a teda celkovo TRUE AND TRUE je TRUE a teda sa vyhodnotenie OR končí a príkaz

k++ nebude vykonaný. Následne sa vykoná j++. Po prvej podmienke budú teda hodnoty premenných nasledovné: i=0, j=1, k=1.

V druhej podmienke:

```
if (i++ || j && !i || k++)
```

je tiež (veľký) OR s dokonca tromi podvýrazmi: (i++), (j AND !i) a (k++), ktoré sa vyhodnocujú zľava doprava, kým niektorý z nich bude TRUE. Prvý sa vyhodnocuje (i++), ktorého hodnota je 0 (FALSE) pričom sa premenná i hneď následne inkrementuje na hodnotu 1, pokračuje sa s vyhodnotením výrazu (j AND !i), ktorý má teraz hodnotu (1 AND !1) keďže sa už použije zvýšená hodnota i=1, teda (TRUE AND FALSE) a teda celkovo FALSE, pokračuje sa teda s vyhodnotením posledného výrazu (k++), ktorý sa vyhodnotí ako TRUE, pričom sa k zvýši na 2, a následne ak j sa zvýši na 2.

Výsledné hodnoty sú teda i=1, j=2, k=2, pričom jediné zvýšenie ++, ktoré sa nevykonalo je k++ v prvom **if** príkaze.

G (max 5b): Doplňte chýbajúce príkazy v implementácii číselného hadíka. Program by mal pre nezáporné celé číslo N na vstupe vykresliť číselného hadíka z čísel 1 až N^2 v dvojrozmernej matici podľa ukážky nižšie. Zo vstupu program spracuje postupne všetky čísla.

Ukážka vstupu:

```
3
4
```

Výstup pre ukážkový vstup:

```
1 2 3
6 5 4
7 8 9

1 2 3 4
8 7 6 5
9 10 11 12
16 15 14 13
```

```
v          int n)
{
  int i, j;
                          (1)
    for (j = 0; j < n; j++)
    {
      if (i % 2 == 0)
                                (2)
      else
                                (3)
    }
}

int main(void)
{
  int i, j, **a, n;
  while (                       ) (4)
  {
    a =                                   ; (5)
    for (i = 0; i < n; i++)
                                          (6)

    hadik(a, n);
```

```
    for (i = 0; i < n; i++)
    {
      for (j = 0; j < n; j++)
                                          (7)
      printf("\n");
    }
  }
}
```

Riešenie:

Pre názornejšie vysvetlenie sme označili príkazy na doplnenie číslami. Nasleduje úvaha ako tieto príkazy doplniť, aby výsledný program spĺňal zadanie.

Analyzujeme príkazy v poradí v akom sa budú vykonávať pri behu programu.

Hlavná funkcia **main** obsahuje cyklus **while**, v ktorom sa zdá byť spracovanie jedného čísla zo vstupu. Keďže program musí podľa zadania byť schopný spracovať všetky čísla na vstupe, každé rovnako a postupne po jednom, tak príkaz (4) nám umožňuje hlavnú slučku opakovane spustiť. Príkaz (4) bude teda obsahovať načítanie jedného čísla, a telo cyklu predstavuje spracovanie celé čísla.

Príkaz (4) preto doplníme ako načítanie jedného čísla až pokým nie je koniec vstupu.

(4) `while (scanf("%d", &n) == 1) // nacitania vsetkych cisel zo vstupu`

Následne si všimneme telo cyklu, a spôsob spracovania. Spracovanie je rozdelené na tri časti:

- Inicializácia premennej a príkazy na doplnenie (5) a (6)
- Samotné vyplnenie hadíka – volanie funkcie hadik s príkazmi na doplnenie (1), (2) a (3)
- Výpis hadíka, treba doplniť príkaz (7).

Všimnime si premennú a, je typu **int**** teda je to dvojrozmerná matica; a tiež si všimneme volanie funkcie **hadik**, v ktorom sú vstupné parametre a (dvojrozmerná matica) a číslo zo vstupu – predpokladáme že to bude veľkosť matice.

Príkazy (5) a (6) teda zjavne musia vytvoriť maticu príslušnej veľkosti, keďže program musí počítať s rôznymi veľkosťami nie je možné túto veľkosť vopred staticky v programe zadefinovať. Dynamická alokácia dvojrozmerného poľa prebieha dvojkrokovo, najskôr sa alokuje príslušný počet riadkov:

(5): `a = (int**)malloc(sizeof(int*)*n);`

a potom pre každý riadok (i) sa alokuje príslušný počet stĺpcov:

```
   for (i = 0; i < n; i++)
(6):  a[i] = (int*) malloc(sizeof(int)*n);
```

Výpis matice prebieha v príkaze (7), nie je možné aby sa matica vypisovala na inom mieste (napr. vo funkcii **hadik**) pretože inde v programe nie je výpis nového riadku (\n) ktorý je potrebný na správne odriadkovanie čísel v matici (aby neboli všetky v jednom riadku). Výpis prebieha nutne po prvkoch:

(7): `printf("%d ", a[i][j]);`

Zostáva určiť chýbajúce príkazy vo funkcii **hadik**. Vo funkcii si prvé všimneme použitie premennej i, ktorá sa používa hneď v tele cyklu j, ale dovtedy zatiaľ nebola nastavená jej hodnota, a teda nutne príkaz (1) musí zabezpečiť nastavenie hodnoty premennej i. Vzhladom na vyplnenie dvojrozmernej matice môžeme predpokladať, že chýbajúci

príkaz (1) je štandardný cyklus pre i, podobne ako v prípade pre j, a nasledujúce príkazy (2) a (3) určujú hodnoty prvkov v párnych a nepárnych riadkoch matice.

(1): `for (i = 0; i < n; i++)`

Sú rôzne možnosti ako čísla 1,2, ... N^2 do matice vyplniť. Jedna možnosť je, že príkazy (2) a (3) budú priraďovať vždy hodnoty prvku v i-tom riadku a j-tom stĺpci, a teda celé priradenie bude vyzerať nasledovne:

```
void hadik(int **a, int n)
{
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
      if (i % 2 == 0)
        a[i][j] = n*i+j+1;
      else
        a[i][j] = n*(i+1)-j;
    }
}
```

Priradenie správneho čísla je založené natom, že v párnych (počítame od 0) riadkoch k počiatočného čísla v riadku n*i pripočítame hodnotu stĺpcu (j) – hodnoty v riadku stúpajú, avšak v nepárnych riadkoch od počiatočnej hodnoty (n*(i+1)) odpočítame hodnotu stĺpca – hodnoty v riadku klesajú.

Alternatíva je, že vždy priraďujeme postupne rastúce čísla 1, 2, … N^2 (pre postupne rastúce i a j to je hodnota n*i+j+1) a správne určime stĺpec v matici, do ktorého príslušné číslo patrí, ako v nasledovnom programe:

```
void hadik(int **a, int n)
{
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
      if (i % 2 == 0)
        a[i][j] = n*i+j+1;
      else
        a[i][n-j-1] = n*i+j+1;
    }
}
```

Hlavná funkcia s doplnenými príkazmi vyzerá nasledovne:

```
int main(void)
{
	int i, j, **a, n;
	while (scanf("%d", &n) == 1)
	{
		a = (int**)malloc(sizeof(int*)*n);
		for (i = 0; i < n; i++)
			a[i] = (int*) malloc(sizeof(int)*n);

		hadik(a, n);

		for (i = 0; i < n; i++)
		{
			for (j = 0; j < n; j++)
				printf("%d ", a[i][j]);
			printf("\n");
		}
	}
	return 0;
}
```