STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Základy procedurálneho programovania 1

Opakovanie na úvod

21. 11. 2016

zimný semester
2016/2017

Opakovanie – program

- Počítač nástroj, ktorý vykonáva programy
- **Program** je postupnosť inštrukcií, ktoré povedia počítaču, ako vykonať úlohu
- **Programovacie jazyky** sprostredkujú inštrukcie počítaču, ktorý ich vykonáva
- **Programovanie** je
  1. vymyslenie a navrhnutie postupu riešenia úlohy, a
  2. zapísanie riešenia v programovacom jazyku

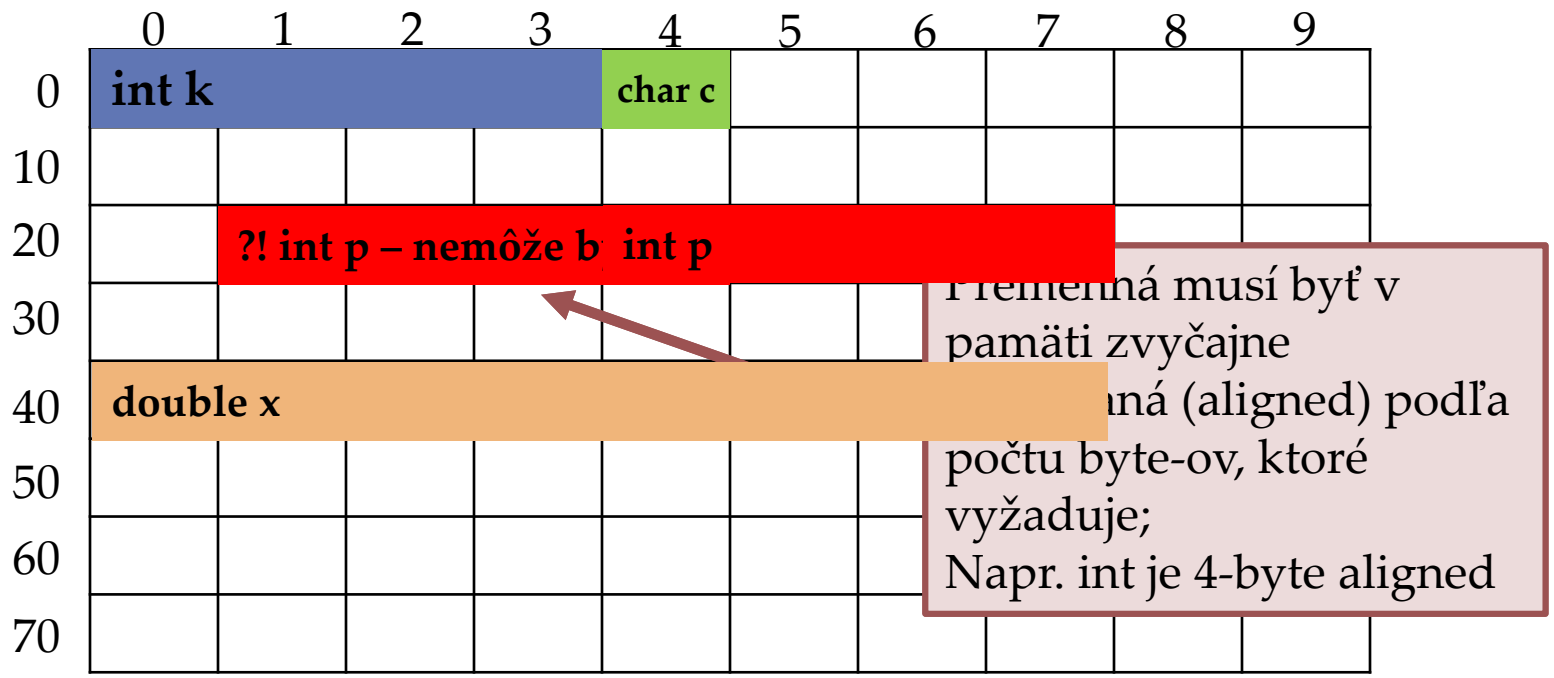
Opakovanie – premenná

- Rozličné dáta, ktoré ukladáme do pamäti zaberajú v pamäti rozlične veľa miesta
- **Adresa pamäte** je poradie (číslo) byte-u od začiatku pamäte
- **Identifikátory** odkazuju na entity (premenné, funkcie) v programe
  - Názvy premenných, funkcií, ...
- **Premenná** je pomenovaný priestor v pamäti
  - previazanie identifikátora s pamaťou
- **Priradenie** naplní hodnotu do premennej (do pamäti vyhradenej pre premennú) **vek** ← **24**

Opakovanie – premenná

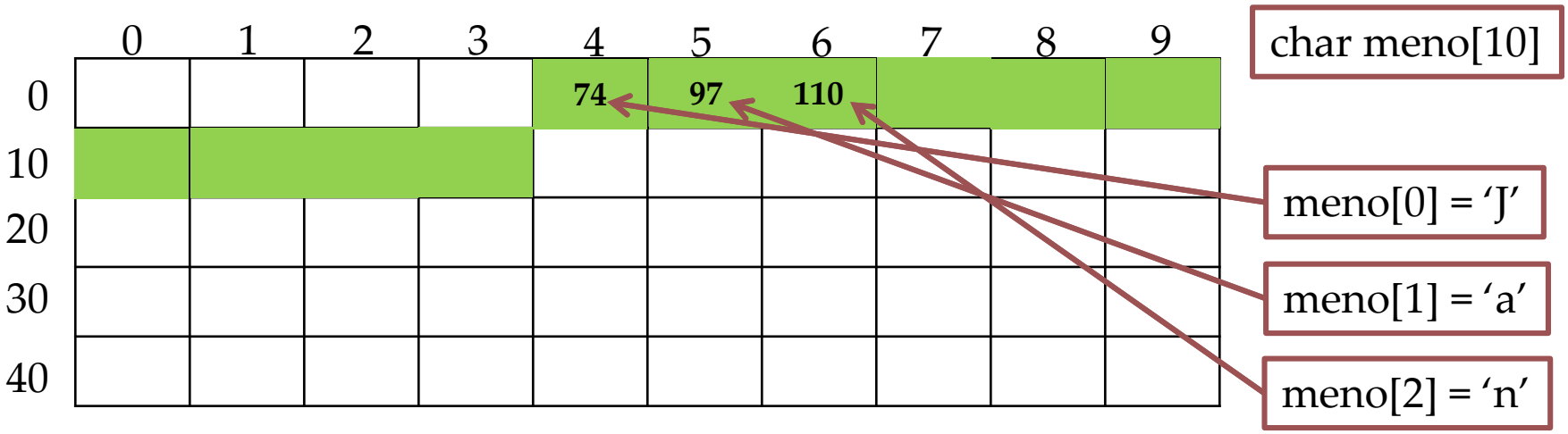
- **Premenná** je pomenovaný priestor v pamäti
  - previazanie identifikátora s pamaťou
- Začiatok (prvý byte) v pamäti, kde sú dáta pre premennú vyhradené, nazývame **adresa premennej**
  - **adresa je (obyčajné) číslo** – poradie prvého byte od začiatku (vyhradenej) pamäte
- Adresa sa označuje symbolom ampersand (&)
- Premenná x – **adresa x je &x**
- **Priradenie cez adresu** (px = &x)
  ***px = 42** je to isté ako **x = 42**

Opakovanie – premenné v pamäti



- Ako si do pamäte uložím znak? (napr. ‘x’)
- Ako si do pamäte uložím meno? (napr. „Bratislava“)

Opakovanie – Reťazec znakov v pamäti



- Ako zistím dĺžku (počet znakov hodnoty) reťazca?
- Pre premennú meno je v pamäti vyhradených 10 znakov, ak je hodnota reťazca „Jan“, tak má len 3 znaky
- Riešenie: za posledným znakom hodnoty reťazca je číslo 0
- Reťazec „Jan“ je teda:

74	97	110	0

- Reťazec „0+2=5“ je:

48	43	50	61	53	0

Opakovanie – Reťazec v jazyku C

- Reťazec je pole znakov (char)
  napr. char meno[5]
- Hodnota reťazca sú znaky až po ukončovací kód 0
- **Výpis reťazca** na obrazovku
  printf a formátovací reťazec %s:

```
char meno[5];
meno[0] = 'J';
meno[1] = 'a';
meno[2] = 'n';
meno[3] = 0;
printf("%s", meno); // vypise Jan
```

- Čo keď v reťazci nie je ukončenie kódom 0?
  - Vypisuje sa pamäť, až kým sa niekde ďalej nenarazí na kód 0
    Napr.: Jan#$TR#RF#E%9^#$@#T#TG#@T%

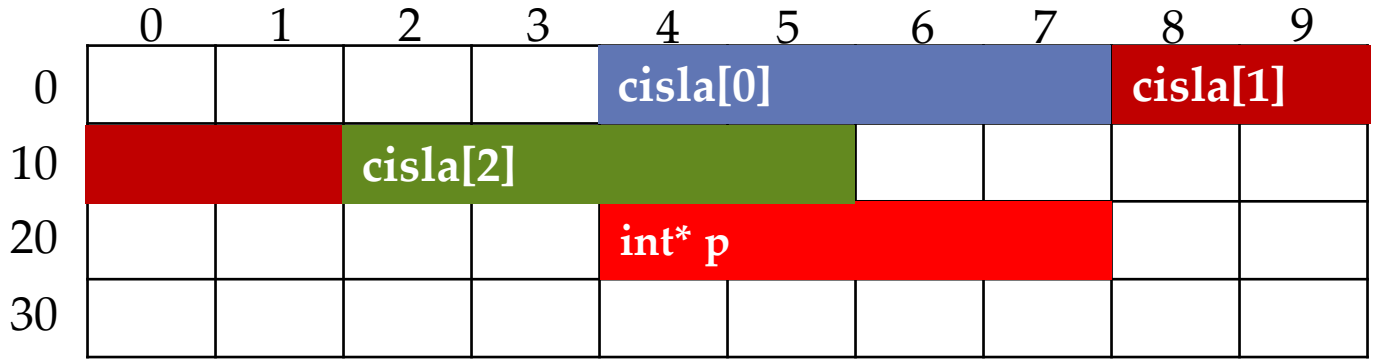
Zložené dátové typy

- Čo ak chceme, aby premenná mohla uschovať **viac** ako poskytuje základný dátový typ?
- Viac rovnakých dátových typov za sebov – **pole**
  - množstvo píšeme v hranatých zátvorkách: **int cisla[100]**
  - k prvkom pristupujeme cez hranaté zátvorky **cisla[7]**, …
  - zvyčajne prvky číslujeme od 0, prvý prvok je **cisla[0]**

- Viac rôznych dátových typov spolu – **štruktúra**
  - Kľúčové slov **struct** – definícia štruktúry
  - Pred názvom píšeme struct: **struct Osoba**
  - K prvkom pristupujeme cez bodku: napr. **osoba.vek=30;**

Opakovanie: Zložené dátové typy – pole

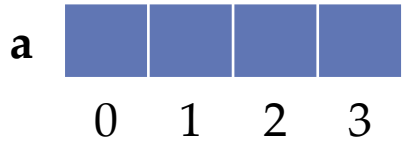
- Viac rovnakých dátových prvkov za sebou v pamäti
- Môžeme k prvkom pristupovať výpočtom presného miesta od začiatku.
- Napr. **int cisla[3]** …



- Adresa premennej **cisla**? … Prvý byte v pamäti – 4
- Adresa tretieho čísla? … **&cisla[2]** … 12
- Nech: p=**cisla**; *(p+1) = 40; … kam sa naplní 40?

Jednorozmerné pole

- Viac rovnakých dátových údajov za sebou
- Rôzne možnosti v programe:
  - Jednorozmerné pole



  - Statické:

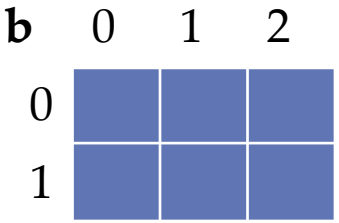
```
int a[4];
```

  - Dynamické:

```
int *a = (int*)malloc(4 * sizeof(int));
```

Viacrozmerne polia

- Dvojrozmerné polia:
  - Obdĺžnikové:



Statické:

```
int b[2][3];
```

Dynamické:

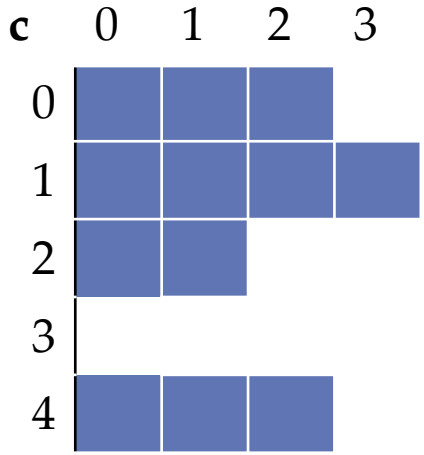
```
int **b = (int**)malloc(2 * sizeof(int*));
b[0] = (int*)malloc(3 * sizeof(int));
b[1] = (int*)malloc(3 * sizeof(int));
```

Viacrozmerne polia

- Dvojrozmerné polia:
  - Zubaté:
    (angl. jagged)

    Statické: nie je možné

    Dynamické:

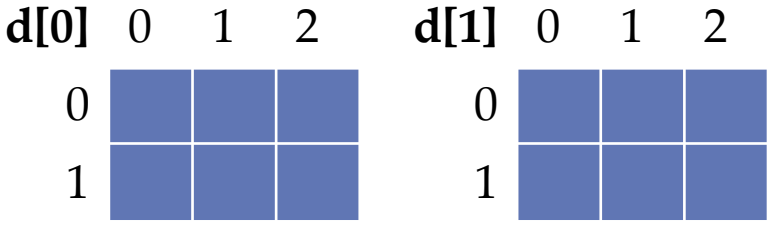



```
int **c = (int**)malloc(5 * sizeof(int*));
c[0] = (int*)malloc(3 * sizeof(int));
c[1] = (int*)malloc(4 * sizeof(int));
c[2] = (int*)malloc(2 * sizeof(int));
c[3] = (int*)malloc(0 * sizeof(int));
c[4] = (int*)malloc(3 * sizeof(int));
```

Viacrozmerne polia

- Trojrozmerné polia:
  - Statické:

```
int d[2][2][3];
```



  - Dynamické:

```
int ***d = (int***)malloc(2 * sizeof(int**));
d[0] = (int**)malloc(2 * sizeof(int *));
d[0][0] = (int*)malloc(3 * sizeof(int));
d[0][1] = (int*)malloc(3 * sizeof(int));
d[1] = (int**)malloc(2 * sizeof(int *));
d[1][0] = (int*)malloc(3 * sizeof(int));
d[1][1] = (int*)malloc(3 * sizeof(int));
```

  - Zubaté – podľa potreby ☺

Zložené dátové typy – štruktúra

- Štruktúra združuje rôzne typy do jedného, a umožňuje s nimi pracovať využitím mien (program mená položiek v štruktúre automaticky prekladá na adresy pamäte podľa toho ako sú položky v pamäti)
- **Definícia** – musíme najskôr zadefinovať ako bude vyzerať, podľa definície sa rozloží v pamäti – zarovná sa podľa veľkosti dátových typov (alignment)
- Napr. komplexné čísla:

```
struct KomplexneCislo {
  int r;      // celá časť
  int i;      // imaginárna časť
}
```

- **Použitie** – `struct KomplexneCislo x;`
  `x.r=20; x.i = 5;`

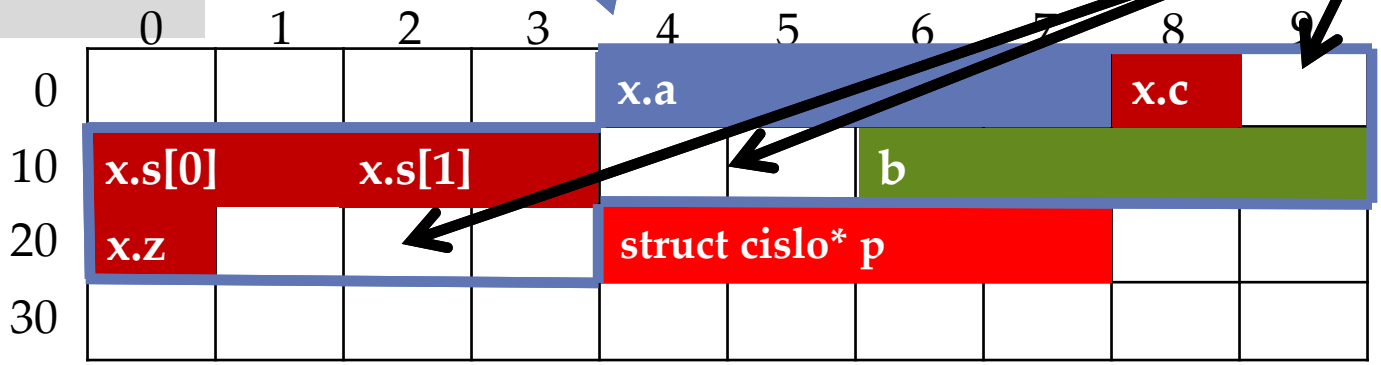
Zložené dátové typy – štruktúra

```
struct cislo {
  int a;
  char c;
  short s[2];
  int b;
  char z;
}
```

struct cislo x;

Modrý okraj: Celková veľkosť vyhradenej pamäte pre jednu štruktúru typu **struct cislo** je 20 bytes (celá musí byť zarovnaná na 4 bytes)

Nevyužité miesta, kvôli zarovnaniu



- Adresa premennej x? ... Prvý byte v pamäti – 4
- Adresa druhého čísla v poli x.s[1]? ... &x.s[1] ... 12
- Nech: p=&x; (*p).c = 'w'; Pre prístup cez smerník na štruktúru môžeme priamo použiť **šípku** (p->c = 'w')

typedef – nové pomenovanie pre typ

- Do priestoru mien (identifikátorov) zavedie nový názov pre nejaký typ
- Najčastejšie sa používa pre štruktúry, aby sa nemuselo písať „**struct**“

```
typedef struct Osoba
{
    char *meno;
    int vek;
} OSOBA;

struct Osoba *o;
OSOBA *p;
```

typedef – nové pomenovanie pre typ

- Je možné použiť aj na existujúce jednoduché typy

```
typedef int Cislo;

Cislo i;
```

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Základy procedurálneho programovania 1

Rekurzia

21. 11. 2016

zimný semester 2016/2017

Opakovanie – funkcie

- Funkcia je pomenovaná časť programu, ktorá vykonáva určitú úlohu
  - funkcia je tiež uložená v pamäti podobne ako premenné, jej dáta sú samotné inštrukcie, ktoré funkcia vykonáva)
- Funkcia je definovaná ako:

```
typ nazovFunkcie(argumenty)
{
  blok;    telo funkcie
}
```

- **argumenty** je zoznam pomenovaných dátových typov, ktoré nadobudnú platnosť v rozsahu bežiacej funkcie ako premenné.

Opakovanie – funkcie

- Funkcia je pomenovaná časť programu, ktorá vykonáva určitú úlohu
- Napr. výpočet obvodu obdĺžnika:

```
int obvod_obdlznika (int a, int b)
{
  return 2*a + 2*b;
}
```

a, b

return je príkaz, ktorý ukončí funkciu, a ako návratovú hodnotu vráti príslušnú hodnotu.

Opakovanie – rozsah platnosti (scope)

- Rozsahy platnosti sú viacerých typov
- **Blok** – rozsah platnosti v rámci bloku vo funkcií
- **Funkcia** – rozsah platnosti v rámci volania funkcie (resp. v rámci bloku tela funkcie)
- **Globálny** – rozsah platnosti vo všetkých funkciách
- Čo keď v rozličných rozsahoch platnosti je premenná rovnakého názvu?
  Ak je x v globálnom rozsahu, a aj nejaké nové x deklarované vo funkcií, tak sa vo funkcii vyhradí nová pamäť pre x v rozsahu platnosti funkcie (a do globálneho x nie je možné prostredníctvom mena 'x' zapísať') (podobne ako pri rekurzii)

Opakovanie – volanie funkcie

- Návratovú hodnotu funkcie potom môžeme použiť tam, kde funkciu voláme (tzv. **volanie funkcie**).

```
int x=10, y=20, obvod;
obvod = obvod_obdlznika(x, y);
PRINT obvod
```

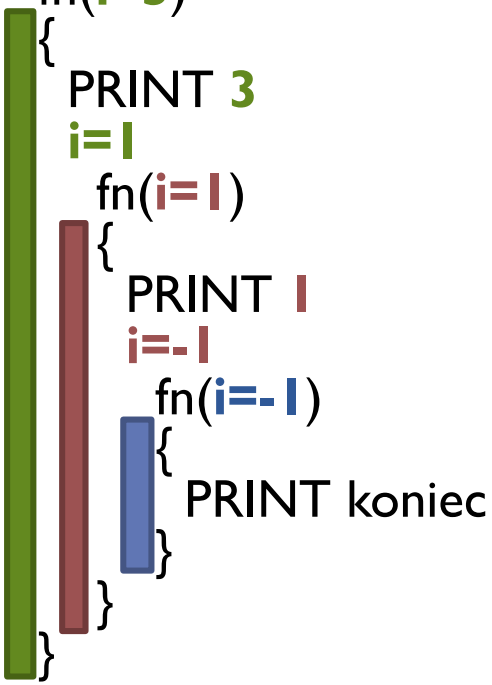
- Premenná **obvod** bude mať hodnotu 60
- Volanie funkcie vytvorí nový rozsah platnosti pre parametre funkcie
- Hlavná funkcia programu **main**

Opakovanie – rozsah platnosti, rekurzia

- Čo keď funkcia bude vo svojom tele volať funkciu s rovnakým názvom? Nazývame to **rekurzia**
- Uplatnia sa rozsahy platnosti
- **fn**(int **i**)

```
{
  if (i <= 0)
    PRINT koniec
   else
  {
    PRINT i
    i = i-2;
    fn(i);
  }
}
```

- Volanie fn(3)



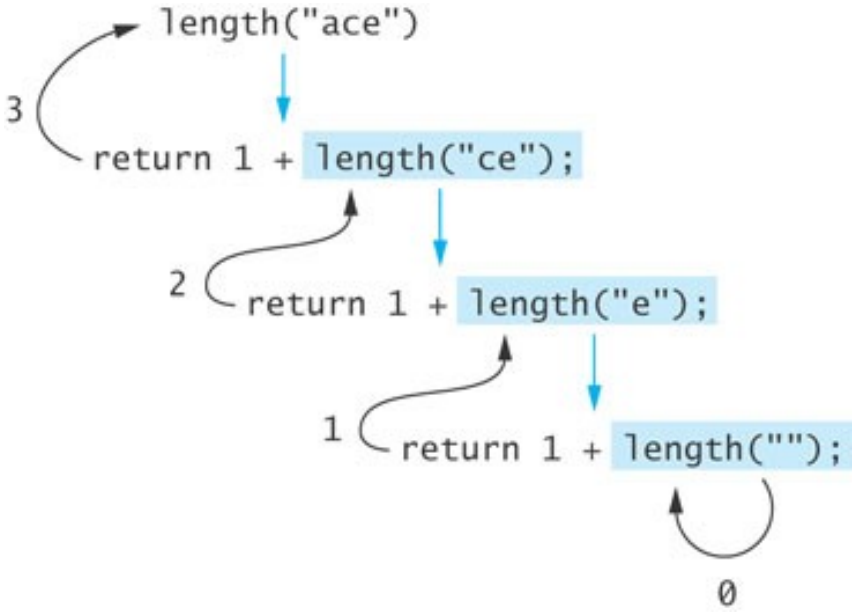
Príklad – Dĺžka reťazca

- Rekurzívny algoritmus na určenie dĺžky reťazca
  - Ak je reťazec prázdny, výsledok je 0, inak výsledok je (1 + dĺžka reťazca bez prvého znaku)

  - Zdrojový kód:

```
int length(char *s)
{
  if (!s || *s == 0)
    return 0;
  return 1 + length(s+1);
}
```

Krokovanie **`length`**(“ace”):

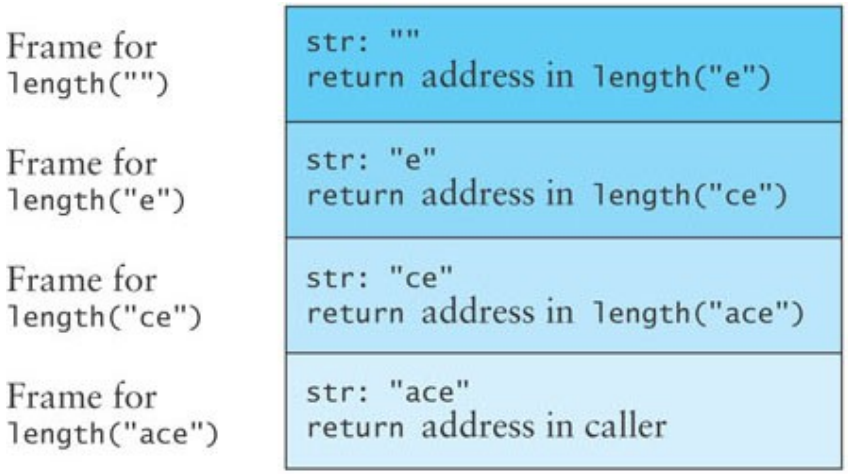


Aktivačný rámec

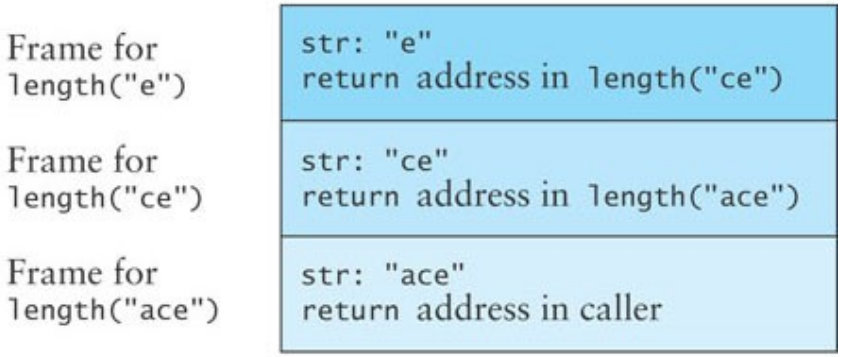
- Stavová informácia pre volanie funkcií
- Pre vykonanie volania funkcie je potrebné uchovať nasledovné informácie:
  - argumenty funkcie (hodnoty parametrov)
  - adresa, kam sa má vrátiť vykonávanie programu po ukončení volania funkcie (návratová adresa pre return)
  - lokálne premenné (hodnoty)
- Pre každé volanie funkcie sa vytvorí aktivačný rámec (stack frame) a vloží sa do zásobníka volaní (call stack)
- (Úmyselné) pretečenie zásobníka volaní predstavuje bezpečnostné riziko: **stack buffer overflow**

Zásobník volaní – ukážka

- Volanie length(“ace”):



obsah zásobníka po zavolaní length(““), vrch je hore



obsah zásobníka po návrate z vykonania length(““)

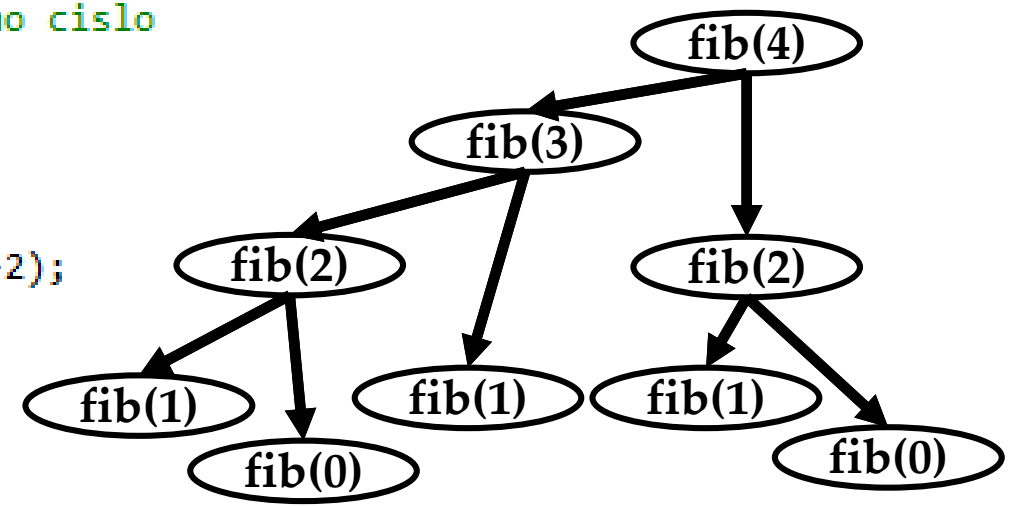
Príklad Fibonacciho čísla

- Rekurzia nám umožňuje veľmi jednoducho rozmýšľať nad problémami
- **Potrebujeme vedieť určiť len jeden krok riešenia, a zvyšok spraví rekurzia za nás ☺**

- Príklad: Urči k-te Fibonacciho číslo
- Fibonacciho postupnosť čísel: prvé dve sú 1 a 1, každé ďalšie je súčet predchádzajúcich dvoch
  1 1 2 3 5 8 13 21 34 55 …
- Ako?

Príklad Fibonacciho čísla (2)

```
// vypocitaj k-te fibonacciho cislo
int fib(int k)
{
    if (k <= 1)
        return k;
    return fib(k-1) + fib(k-2);
}
```



- Problémy?
- Pomalé – pre určenie k-teho čísla potrebujeme až 2^k volaní, pre k=10 to je 2^{10} =1024. Počet volaní určite väčší ako hodnota výsledku… čo je príliš veľa: fib(5)=5, fib(10)=55, fib(20)=6765
- Resp. veľa operácií vykonávame duplicitne
  - Opakované volania **fib**() pre rovnaké k

Priebežné pamätanie (tzv. memoizácia)

- Pri rekurzívnom prehľadávaní (stavového priestoru riešení) sa nám **veľmi často** stáva, že vykonávame duplicitné operácie
- Riešenie je priebežné pamätanie (memoizácia) medzivýsledkov, a ich znovupoužitie
- Implementácia je zvyčajne veľmi jednoduchá

```
int f[100];
// vypocitaj k-te fibonacciho cislo
int fib(int k)
{
    if (k <= 1)
        return k;
    if (f[k] > 0)
        return f[k];
    return f[k] = fib(k-1) + fib(k-2);
}
```

Ďalšia ukážka rekurzie

- Porozmýšľajte, ako by ste napísali program, ktorý vypíše všetky permutácie množiny N prvkov.
- Pre N = 1
- Pre N = 2
- Pre N = 3
- ...

Permutácie – Vnorené cykly

- Permutácie N=3 prvkov:

```
int i, j, k;
for (i = 1; i <= 3; i++)
    for (j = 1; j <= 3; j++)
        for (k = 1; k <= 3; k++)
            if (i != j && j != k && i != k)
                printf("%d %d %d\n", i, j, k);
```

- Problémy?
  - Pomalé – 3^3 operácií (27), permutácií je len 3! = 6

    Možné zrýchlenie:

```
int i, j, k;
for (i = 1; i <= 3; i++)
    for (j = 1; j <= 3; j++)
        if (i != j)
            for (k = 1; k <= 3; k++)
                if (j != k && i != k)
                    printf("%d %d %d\n", i, j, k);
```

  - Ťažko rozšíriteľné pre väčšie N

Rozšírenie pre väčšie N

```
int i, j, k, p, q;
for (i = 1; i <= 5; i++)
    for (j = 1; j <= 5; j++)
        for (k = 1; k <= 5; k++)
            for (p = 1; p <= 5; p++)
                for (q = 1; q <= 5; q++)
                    if (i != j && i != k && i != p && i != q &&
                        j != k && j != p && j != q &&
                        k != p && k != q &&
                        p != q)
                        printf("%d %d %d %d %d\n", i, j, k, p, q);
```

- Čo keď N je parameter programu? ☺
- Použijeme rekurziu ...

Rekurzívne riešenie pre väčšie N

- Pre N spravíme vnorenie do N úrovní
- Na každej úrovni spravíme jeden krok v riešení
  - Čo je jeden krok?
  - Výber ďalšieho čísla
- Začneme s prázdnym riešením (k=0)
- Rekurzívny krok: (N prvkov, a hotových k čísel)
  - Vyber k+1. číslo
  - Ako?
  - Rôzne od doteraz vybratých

Rekurzívne riešenie pre väčšie N

```
// prejdi permutacie n prvkov, ak uz mame k prvkov urcenych v poli p[0,...,k-1]
void fn(int n, int k)
{
    int i, j;
    if (k == n) // ak sme uz vsetky prvky permutacie urcili
    {
        for (i = 0; i < n; i++)
            printf("%d ", p[i]);
        printf("\n");
        return;
    }

    for (i = 1; i <= n; i++)
    {
        // skusam cislo i
        for (j = 0; j < k; j++)
            if (p[j] == i)
                break;
        if (j == k) // ak prvok i este nie je pouzity
        {
            p[j] = i;
            fn(n, k+1);
            p[j] = 0;
        }
    }
}
```

Precvičenie: Násobenie čísel v poli

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
void vynasob_dvomi(int *cisla, int n)
```

Precvičenie: Násobenie čísel v poli

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
void vynasob_dvomi(int *cisla, int n)
{
    if (n <= 0)
        return;
    if (cisla[0] % 2 == 1)
        cisla[0] *= 2;
    vynasob_dvomi(cisla + 1, n-1);
}
```

Precvičenie: Odstránenie prvého znaku z reťazca

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
void odstran_zaciatok(char *s)
```

Precvičenie: Odstránenie prvého znaku z reťazca

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
void odstran_zaciatok(char *s)
{
    if (*s)
    {
        *s = *(s+1);
        odstran_zaciatok(s+1);
    }
}
```

Precvičenie: Odstránenie čísel z reťazca.

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
int odstran_cisla(char *s)
```

Precvičenie: Odstránenie čísel z reťazca.

- Napíšte funkciu **vynasob_dvomi**, ktorá rekurzívne (bez použitia cyklu) vynásobí každé nepárne číslo dvomi. Teda napr. pole cisla[5]={1,2,3,4,5} upraví na hodnoty {2,2,6,4,10}.

```
int odstran_cisla(char *s)
{
    if (*s)
    {
        int c = 0;
        if (*s >= '0' && *s <= '9')
        {
            odstran_zaciatok(s);
            c++;
        }
        return c + odstran_cisla(s + 1);
    }
    return 0;
}
```

Ťažšia úloha – na domov :)

- Rekurzívne usporiadanie znakov v reťazci
- Napíšte rekurzívnu funkciu **usporiadaj_retazec**, ktorá abecedne usporiada znaky v reťazci od najmenšieho po najväčšie. Teda napr. pre vstupný reťazec **jano** bude výstupný reťazec **ajno**.
- Nemôžete použiť cyklus ani pomocnú funkciu!

STU FIIT

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Základy procedurálneho programovania 1

Vstup-Výstup (načítavanie-vypisovanie)

21. 11. 2016

zimný semester 2016/2017

Vstup–Výstup (Input–Output)

- Vstup – ako do programu dostaneme nejaké dáta, ktoré nie sú priamo v zdrojovom kóde programu
- Napr. načítanie zo súboru, vstup z klávesnice, parametre webového formuláru
- Výstup – ako z programu dostaneme nejakú informáciu, napr.
  - Výpis na obrazovku, zapísanie súboru, odoslanie po sieti

Načítavanie v jazyku C

- Aké poznáme spôsoby načítania vstupu?
  - Načítanie čísla – **scanf a formátovací reťazec %d**
  - Načítanie reťazca po prvú medzeru – **scanf a formátovací reťazec %s**
- Načítanie poľa čísel
  - Cyklus
- Načítanie matice (dvojrozmerné pole)
  - Dva vnorené cykly
- Funkcia **scanf** načítava z konzoly (klávesncica)
- Analogické funkcie na
  - načítanie zo súboru – **fscanf** (subor,"%d", &x);
  - načítanie z reťazca – **sscanf** (retazec,"%d", &x);

Načítavanie v jazyku C

- Ako zistíme, že sme na konci vstupu?
  - Koniec súboru
  - Koniec načítavania vstupu z klávesnice
- Platí (zvláštne) pravidlo:
  o konci vstupu sa dozvieme až keď sa pokúsime čítať za koniec vstupu

Načítavanie v jazyku C

- O konci vstupu sa dozvieme až keď sa pokúsime čítať **za** koniec vstupu
- Príklad, nech je na vstupe:
  1 2 3
- Prečítaním čísla 3 ešte nedokážeme zistiť či sme na konci vstupu, ale **až pokusom o ďalšie čítanie** dostane programu od operačného systému informáciu, že je koniec vstupu
- Ako túto informáciu program dostane?
  Návratová hodnota funkcie, ktorá sa pokúšala čítať

Načítavanie v jazyku C

- **Návratová hodnota funkcie scanf**
  - Počet prvkov, ktoré sa podarilo načítať
- Napr.
  máme vstup 5 6 7
  volanie scanf (“%d%d%d%d”, &i, &j, &k, &q) vráti 3, pretože načítali sa tri čísla a potom už bol koniec vstupu

- Pozor, treba si dávať pozor, ak načítame zo vstupu iný počet prvkov ako požadujeme, tak tie ktoré sme nenačítali zostali s pôvodnou hodnotou, a teda asi nemôžeme ďalej pokračovať
- scanf(“%d”, …) vráti alebo 1 (aj načítal číslo), alebo 0 ak nie a je koniec vstupu

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Formátovací reťazec hovorí:
  - Koľko ďalších parametrov spracovať
  - Ako ich interpretovať
- Premenné, ktoré chceme vypísať, formátujeme každú v položke začínajúcej znakom %
- Ostatné znaky sa vypíšu tak ako sú
- Ak chceme vypísať %, tak napíšeme dvakrát (%%)

```
printf("Ahoj :)");

printf("Dane na potraviny su 10%%");
```

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Položka vo formátovacom reťazci určuje typ premennej
- Štandardné typy položiek:
  - **%d, %i** – číselná hodnota so znamienkom v desiatkovej sústave, najčastejšie int:
    %u – unsigned

```
int i=33;
printf("Vek je %d\n", i);
```

  - **%c** – jeden znak (char), %s – reťazec (char*)
  - **%f** – float, **%lf** – double

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Ďalšie typy položiek:
  - **%o** – číselná hodnota v osmičkovej sústave
  - **%x** – číselná hodnota v šesnástkovej sústave

```
int i=33;
printf("Vek je %o\n", i); // Vek je 41
printf("Vek je %x\n", i); // Vek je 21
```

  - **%p** – vypíše ako pointer

```
int *ptr = &i;
printf("Pointer je %p\n", ptr); // Pointer je 0x7fff8e2ce12c
```

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Ďalšie typy položiek:
  - **%e** – desatinné číslo v tvare desiatkového exponentu
  - **%a** – desatinné číslo v tvare hexadecimálneho exponentu
  - **%g** – podľa hodnoty – ako desatinné číslo s bodkou alebo v tvare desiatkového exponentu

```
double x=0.3;
printf("%e\n", x); // 3.000000e-01
printf("%a\n", x); // 0x1.3333333333333p-2
printf("%g\n", x); // 0.3
```

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Modifikátory položiek:
  - **+ (plus)** – znamienko (plus/mínus) sa vždy pridá nazačiatok
  - **- (mínus)** – zarovnanie doľava vzhľadom na veľkosť položky
  - **0 (nula)** – doplnenie položky nulami (ak je zarovnaná doprava)
  - Voliteľne: **číselná konštantna** – minimálna dĺžka položky v znakoch (alebo * a nasledujúci parameter predstavuje dĺžku)
  - Voliteľne: **desatinná bodka a číselná konštanta – počet desatinných miest za desatinnou časťou** (alebo * a nasledujúci parameter predstavuje dĺžku položky za desatinnou časťou)

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Príklady modifikátorov položiek:

```
int i=33;
printf("<-%d>\n", i);
printf("<%-d>\n", i);
printf("<%-5d>\n", i);
printf("<%+05d>\n", i);
printf("<%+5d>\n", i);
printf("<%+0*d>\n", i/5, i);
```

Formátovacie reťazce printf

```
int printf( const char* format, ... );
```

- Príklady modifikátorov položiek:

```
int i=33;
printf("<-%d>\n", i); // <-33>
printf("<%-d>\n", i); // <33>
printf("<%-5d>\n", i); // <33   >
printf("<%+05d>\n", i); // <+0033>
printf("<%+5d>\n", i); // <  +33>
printf("<%+0*d>\n", i/5, i); // <+00033>
```

Varianty printf

- Varianty pre zápis do reťazca, prefix s:
  sprintf
- **Prvý argument je reťazec**

```
char buf[50];
sprintf(buf, "<%+*d>\n", i/5, i);
printf("%s\n", buf); // <   +33>
```

- Obmedzenie na počet zapísaných znakov (znak n v názve) – **snprintf**

```
int snprintf(char *str, size_t size, const char *format, ...);
```

Varianty printf

- Varianty pre zápis do súboru, prefix f:
  fprintf
- **Prvý argument je súbor (FILE *)**

```
FILE *f = fopen("subor.txt", "w");
fprintf(f, "<%+*d>\n", i/5, i); // <   +33>
fclose(f);
```

- Ako môžeme otvoriť súbor (fopen):
  - "r"– čítanie (read), "w" – vytvorenie a zápis (write), „a“ – otvorenie a zápis na koniec (append)
  - „rb“ binárne čítanie (fread), „wb“ binárny zápis (fwrite)

Načítavanie – Scanf a jej varianty

- scanf – načítava zo štandardného vstupu
  - Formátovací reťazec podobný ako pri printf **"%5d%s %c%lf"** načíta 5 znakov ako číslo, zvyšok do medzery ako reťazec, potom načíta jeden znak a zvyšok ako desatinné číslo 12345678 x0.9 … 12345 678 x 0.9
  - Vracia počet načítaných položiek
- fscanf – načítava zo súboru (FILE *)

```
int fscanf(FILE *stream, const char *format, ...);
```

- sscanf – načítava z reťazca (char *)

```
int sscanf(const char *str, const char *format, ...);
```

  - Pozor: opakované volania „neposúvajú načítavanie ďalej v reťazci“, stále sa načítava reťazec „od začiatku“
  - sscanf(buf+k, "%s", str); k+=strlen(str);