

Programiranje 2

6. predavanje

Saša Singer

`singer@math.hr`

`web.math.pmf.unizg.hr/~singer`

PMF – Matematički odsjek, Zagreb

Sadržaj predavanja

- **Stringovi** (kraj od prošli puta):
 - Primjeri obrade stringova.
 - Korisne funkcije za pretvorbu iz `<stdlib.h>`.
- **Pokazivači** (drugi dio):
 - Polje pokazivača — deklaracija. Polje stringova.
 - Polje pokazivača i dvodimenzionalno polje.
 - Sortiranje rječnika zamjenana pokazivača.
 - Argumenti komandne linije.
 - **Razlika** — dvodimenzionalno polje i niz nizova.
 - Pokazivač na funkciju.

Informacije — kolokviji

Programiranje 2 je u kolokvijskom razredu C3.

Službeni termini svih kolokvija su:

- Prvi kolokvij: petak, 22. 4. 2016., u 15 sati.
- Drugi kolokvij: četvrtak, 23. 6. 2016., u 15 sati.
- Popravni kolokvij: ponedjeljak, 5. 9. 2016., u 15 sati.

Uputa: “izbjegnite” popravni — obavite to ranije!

Informacije

Konzultacije (službeno):

🕒 petak, 12–14 sati, ili — po dogovoru.

Ne zaboravite, “žive” su i domaće zadaće na adresi

<http://degiorgi.math.hr/prog2/ku/>

Dodatni bodovi “čekaju na vas”.

Stringovi (nastavak)

Sadržaj

- Stringovi (nastavak):
 - Primjeri obrade stringova.
 - Korisne funkcije za pretvorbu iz `<stdlib.h>`.
 - Implementacija funkcije `atoi`.
 - Obratna funkcija `itoa`.

Broj riječi u stringu

Primjer. Napisati funkciju koja **broji riječi** u stringu koji je stigao kao argument. **Riječ** je niz znakova **bez praznina**, a riječi su odvojene **bar jednom prazninom** ili znakom **\t**.

U **prolasku** kroz string, koristimo **logičku** varijablu **razmak** koja pamti jesmo li:

- u **razmaku** (“između” riječi) i tad je **TRUE (1)**, ili
- u **riječi** i tad je **FALSE (0)**.

Brojač riječi **povećavamo** na **izlasku** iz riječi.

```
int broj_rijeci(char *str)
{
    int brojac = 0;
    int razmak = TRUE; /* Ispred prve rijeci. */
}
```

Broj riječi u stringu (nastavak)

```
while (*str != '\0') {
    if ((*str == ' ') || (*str == '\t')) {
        if (!razmak) {
            ++brojac;
            razmak = TRUE;
        }
    }
    else
        razmak = FALSE;
    ++str;
}
if (!razmak) ++brojac; /* Zadnja rijec */
return brojac;
}
```


Broj riječi u stringu — glavni program

Primjer (nastavak). Glavni program (v. `br_rij_1.c`).

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

#define TRUE 1
#define FALSE 0

int broj_rijeci(char *str)
    ...
```

Broj riječi u stringu — glavni i izlaz

```
int main(void) {  
    char s[] = "Ja sam mala Ruza, mamina sam kci."  
    printf("String:\n");  
    printf("%s\n", s);  
    printf("Broj rijeci: %d\n", broj_rijeci(s));  
    return 0;  
}
```

Izlaz:

String:

Ja sam mala Ruza, mamina sam kci.

Broj rijeci: 7

Zadatak. Napišite funkciju `broj_rijeci` koristeći **indekse**.

Broj riječi u stringu (nastavak)

Ovaj primjer možemo napraviti i tako da brojač povećavamo

na **ulasku** u riječ (što je **prirodnije**).

Umjesto **razmak**, zgodnije je koristiti **logičku** varijablu **rijec** s obratnim značenjem:

ako smo **u riječi**, vrijednost je **TRUE** (**1**),

ako smo **u razmaku** (“između” riječi), vrijednost je **FALSE** (**0**).

```
int broj_rijeci(char *str)
{
    int brojac = 0;
    int rijec = FALSE;  /* Ispred prve rijeci. */
```

Broj riječi u stringu (nastavak)

```
for ( ; *str != '\0'; ++str)
    if ((*str == ' ') || (*str == '\t')) {
        if (rijec)
            rijec = FALSE ; /* Moze BEZ if */
    }
    else
        if (!rijec) {
            ++brojac;
            rijec = TRUE;
        }
return brojac;
}
```

Umjesto **while** petlje, iskoristili smo **for** petlju.

Broj riječi u stringu — komentar

Napomena: Funkcija za brojanje riječi radi na principu konačnog automata. To je “jednostavniji” stroj od Turingovog — traka je samo ulaz, ne treba pisati na traku (v. Prog1).

Pripadni konačni automat ima samo 2 moguća stanja:

- unutar riječi,
- izvan riječi, tj. razmak “između” riječi.

Ta stanja smo pamtili u varijabli razmak ili rijec, koristeći samo 2 vrijednosti — 0 i 1.

Prijelaz iz stanja u stanje (ili ostanak u stanju) određen je samo trenutnim stanjem i sljedećim znakom na ulazu.

Povećanje brojača je dodatna akcija — koju radimo kod odgovarajućeg prijelaza iz jednog stanja u drugo.

Broj riječi u stringu — varijacije na temu

U prethodnoj funkciji, “separatori” riječi su samo 2 znaka:

- praznina (znak ' ') i horizontalni tabulator (znak '\t').

Svi ostali znakovi su dio riječi.

Varijacije. Promijenite funkciju broj_rijeci tako da “separatori” riječi budu i sljedeći znakovi:

- “bjeline” (kao u C-u za funkciju scanf), tj. znakovi:

- ' ', '\t', '\v', '\n', '\r', '\f',
(možete koristiti funkciju isspace),

- još i “interpunkcijski” znakovi:

- ', ', ':', ';', '.', '!', '?'.

Kao i prije, svi ostali znakovi su dio riječi.

Broj riječi u stringu — zadaci

Zadaci. Napišite funkciju koja ima `string` kao argument i radi sljedeće:

- **Broji** riječi s određenim svojstvom:
 - samo “prave” riječi, sastavljene od slova (`alpha`),
 - samo “brojeve”, sastavljene od dekadskih znamenki (`digit`),
 - samo `operatore` — po pravilima C-a.
 - Ovo nije lako: neki operatori su sastavljeni iz 2 znaka — poput `||`, `<<`, `<=`, `+=` (ima ih još).
- Nalazi najdulju riječ s određenim svojstvom i vraća pokazivač na početak te riječi.
 - Oprez: provjeru je li riječ najdulja (do sada) moramo napraviti na izlazu iz riječi.

Primjer — implementacija funkcije `atoi`

Primjer. Napišimo implementaciju funkcije `atoi` iz standardne biblioteke (zaglavlje `<stdlib.h>`) koja pretvara niz znakova (`string`), koji predstavlja cijeli broj, u njegovu numeričku vrijednost. Funkcija treba:

- preskočiti sve početne bjeline (kao u funkciji `scanf`),
 - učitati predznak (ako ga ima),
 - i redom pročitati sve znamenke broja,
- te ih pretvoriti u broj tipa `int` (v. `f_atoi.c`).

```
#include <ctype.h>
```

```
int f_atoi(const char s[])  
{
```


Primjer — implementacija atoi (nastavak)

```
int i, n, sign;

    /* preskace sve bjeline, prazan for */
for (i = 0; isspace(s[i]); ++i) ;
sign = (s[i] == '-') ? -1 : 1;    /* predznak */
    /* preskoci predznak */
if (s[i] == '+' || s[i] == '-') ++i;
    /* Hornerov algoritam za broj */
for (n = 0; isdigit(s[i]); ++i)
    n = 10 * n + (s[i] - '0');
return sign * n;
}
```

Ovdje koristimo funkcije `isspace` i `isdigit` iz `<ctype.h>`.

Na primjer: `f_atoi(" -1234") = -1234` (broj tipa `int`).

Funkcije iz <stdlib.h>

U standardnoj biblioteci <stdlib.h> već postoje neke funkcije konverzije za pretvaranje stringa u broj odgovarajućeg tipa.

| | |
|----------------------------|----------------------|
| double atof(const char *s) | pretvara s u double, |
| int atoi(const char *s) | pretvara s u int, |
| long atol(const char *s) | pretvara s u long. |

Ove funkcije rade slično kao čitanje broja odgovarajućeg tipa:

- ➊ preskaču se bjeline na početku stringa,
- ➋ a zatim se “čita” najdulji niz znakova koji odgovara pravilu za pisanje broja tog tipa,
- ➌ i pretvara u broj.

Funkcije konverzije iz <stdlib.h> — *primjer*

Primjer: pretvaranje “početka” stringa u broj.

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

int main(void) {
    printf("%d\n", atoi(" 123"));          /* 123 */
    printf("%d\n", atoi(" 12 3"));         /* 12 */
    printf("%d\n", atoi(" 12abc3"));       /* 12 */
    printf("%d\n", atoi("abc12 3"));       /* 0 */

    printf("%g\n", atof(" 12.5a4c 3"));    /* 12.5 */
    printf("%g\n", atof(" 12.5e4c 3"));    /* 125000 */
    return 0;}

```

Još neke funkcije iz <stdlib.h>

Napomena. U datoteci zaglavlja <stdlib.h> deklarirane su i funkcije za apsolutnu vrijednost cijelih brojeva.

| | |
|-------------------|-----------------------|
| int abs(int n) | apsolutna vrijednost, |
| long labs(long n) | apsolutna vrijednost. |

Funkcija double fabs(double d) nije tu, već u matematičkoj biblioteci <math.h>.

Zadatak: itoa — ispis cijelog broja u string

Zadatak. Napišite implementaciju funkcije `itoa` koja pretvara cijeli broj u string (kao kod ispisa).

Napomena: Funkcija `itoa` ne postoji u `<stdlib.h>`. Jedna moguća implementacija dana je u knjizi KR2, str. 64 (kôd je na sljedećoj stranici, v. `f_itoa_1.c`).

☛ Međutim, ta funkcija ne radi za najmanji prikazivi cijeli broj — na primjer, -2^{31} na 32 bita.

Probajte ju popraviti, tako da radi za sve prikazive cijele brojeve iz tipa `int`!

Funkcije `atoi` i `itoa` mogu se realizirati i čitanjem/pisanjem u string pomoću funkcija `sscanf`, `sprintf` (v. kasnije).

Implementacija funkcije itoa

```
void f_itoa(int n, char s[])
{
    int i, sign;

    /* pamti predznak u sign i
       pretvara n u nenegativan broj */
    if ((sign = n) < 0)
        n = -n;
    i = 0;
    /* generiraj znamenke u obratnom poretku */
    do {
        s[i++] = n % 10 + '0'; /* znamenka */
    } while ((n /= 10) > 0);    /* obrisi ju */
}
```

Implementacija funkcije itoa (nastavak)

```
    if (sign < 0)
        s[i++] = '-';    /* predznak minus na kraj */
    s[i] = '\0';         /* kraj stringa */

    invertiraj(s);

    return;
}
```

Funkciju `invertiraj` smo napravili ranije!

Zadatak. Objasnite (bez testiranja) zašto ova funkcija **ne radi** za **najmanji** prikazivi cijeli broj (na pr. -2^{31} na 32 bita).

Probajte ju **popraviti** (nije teško)!

Zadatak: funkcije `atof` i `ftoa`

Zadatak. Napišite implementaciju funkcije `atof` (`<stdlib.h>`) koja **prevodi niz znakova** (`string`), koji predstavlja **realni broj**, u njegovu **numeričku** vrijednost (tipa `double`).

- **Realni broj** smije biti napisan po **svim pravilima C-a** za pisanje realnih konstanti (**točka**, **e**)!

Zadatak. Razmislite kako biste napravili implementaciju funkcije `ftoa` koja **pretvara realni broj** (tipa `double`) u `string` (kao kod ispisa u `printf`).

- Što sve treba **zadati**?

Izazov. Probajte realizirati **obje** funkcije tako da

- rade za **sve prikazive double** brojeve (**nije lako**).

Dodatni primjeri

Domaća zadaća. Progledajte **pažljivo** primjere u **prvom poglavlju** knjige **KR2**.

● Vrlo su **instruktivni** u smislu **tehnika programiranja**.

Posebno se isplati pogledati **odjeljak 1.5** (i nadalje), gdje je

● niz primjera o **obradi “teksta”** (ulaz i izlaz znakova, brojanje znakova, riječi i linija u tekstu).

Pokazivači (nastavak)

Sadržaj

- Pokazivači (drugi dio):
 - Polje pokazivača — deklaracija. Polje stringova.
 - Polje pokazivača i dvodimenzionalno polje.
 - Sortiranje rječnika zamjenana pokazivača.
 - Argumenti komandne linije.
 - Razlika — dvodimenzionalno polje i niz nizova.
 - Pokazivač na funkciju.

Polje pokazivača

Polje pokazivača ima deklaraciju:

```
tip_pod *ime[izraz];
```

Napomena: **Primarni** operator `[]` ima viši prioritet od **unarnog** operatora `*`.

Primjer. Razlikujte **polje** pokazivača (ovdje, 10 pokazivača):

```
int *ppi[10];
```

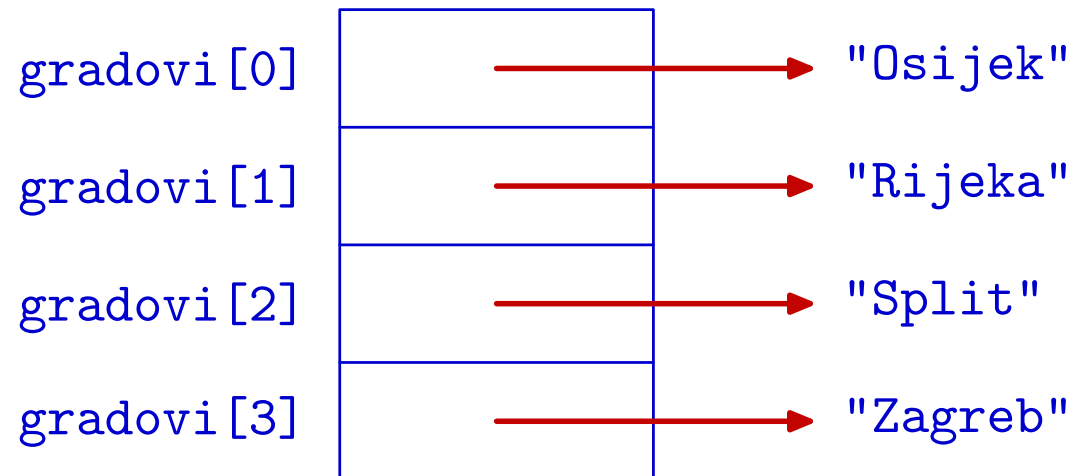
od **pokazivača** na **polje** (ovdje, od 10 elemenata):

```
int (*ppi)[10];
```

Polje pokazivača (nastavak)

Pokazivač na `char` možemo inicijalizirati `stringom`. Isto vrijedi i za `polje` takvih `pokazivača` — dobivamo “`polje stringova`”.

```
static char *gradovi[] = { "Osijek", "Rijeka",  
                           "Split", "Zagreb"};
```



Ovo se često koristi za `fiksna imena` objekata. Na primjer, `dani` u tjednu, `mjeseci` u godini (v. [KR2](#), str. 113–114).

Polje pokazivača i dvodimenzionalno polje

Postoji bitna **razlika** između **polja pokazivača** (na znak, odnosno, string):

```
char *mjesto[] = { "Hum", "Kanfanar",  
                  "Sveti Vincenat" };
```

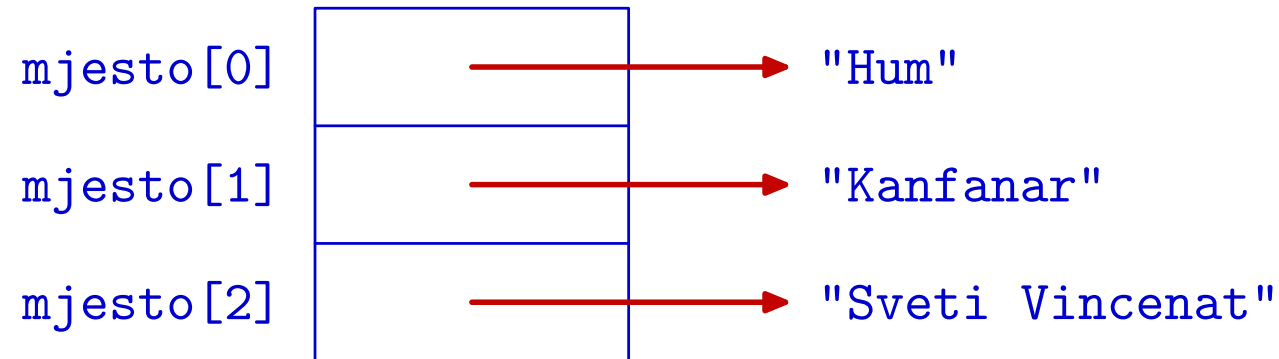
i **dvodimenzionalnog polja** znakova:

```
char amjesto[][16] = { "Hum", "Kanfanar",  
                      "Sveti Vincenat" };
```

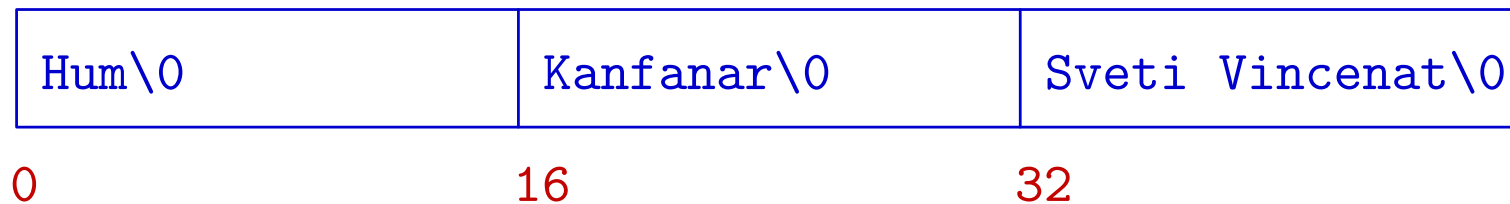
Razlika se najbolje **vidi** na sljedećim slikama.

Polje pokazivača i dvodimenzionalno polje

Polje pokazivača mjesto:



Dvodimenzionalno polje amjesto:



Sortiranje rječnika

Problem. Imamo **hrpu riječi**, raznih duljina, koje treba:

- učitati sa standardnog ulaza,
- leksikografski **sortirati**,
- ispisati u sortiranom poretku.

Očito je da **pojedine riječi** treba spremati kao **stringove**. Da ne kompliciramo, pretpostavit ćemo da

- svaka pojedina **riječ** stane u **string** od **80** znakova.

(Dinamičko spremanje stringova **proizvoljne** duljine — sami.)

Dogovor: ulaz sadrži po **jednu** riječ u svakom redu (**gets**).

Kako ćemo **spremiti** našu “**hrpu**” riječi?

Sortiranje rječnika (nastavak)

Ako ih spremimo u dvodimenzionalno polje, onda je sve relativno lako — sortiranje niza “redaka = riječi” (v. Prog1). Međutim, to je loše:

- spremanje je memorijski rastrošno (v. prošlu sliku),
- sortiranje je vrlo sporo — zbog zamjena stringova (polja).

Zato ćemo program realizirati preko polja pokazivača na stringove (riječi). U tom polju, nazovimo ga **p**,

- **i**-ti element sadrži pokazivač na **i**-tu riječ.

Sortiranje riječi realiziramo

- zamjenama pokazivača u polju **p**, a ne stringova!

Problem: učitavanje i stvarno spremanje riječi.

Sortiranje rječnika — čitanje riječi

Primjer. Učitavanje rječnika — ne ovako!

```
int main() {  
    char *p[2], q[80];  
  
    gets(q);    /* plava */  
    p[0] = q;  
  
    gets(q);    /* crvena */  
    p[1] = q;  
  
    puts(p[0]);    /* Sto ce se ispisati? */  
    puts(p[1]);    /* Sto ce se ispisati? */  
    return 0; }
```

Sortiranje rječnika — čitanje riječi (nastavak)

Izlaz:

crvena

crvena

Razlog: `p[0]` i `p[1]` sadrže istu adresu = onu od `q`, pa “vide”
🕒 isti string = zadnji učitani.

Funkciji `gets` treba poslati pravo mjesto za spremanje stringa!

Jedno moguće rješenje za stvarno spremanje riječi:

🕒 koristimo jedno “veliko” jednodimenzionalno polje znakova, u koje redom “trpamo” riječi, jednu za drugom.

(Dinamičko alociranje stringova za riječi — sami, v. vježbe.)

Sortiranje rječnika — spremanje riječi

Dakle, imamo dva jednodimenzionalna polja:

- polje znakova w — koje sadrži sve riječi (kao stringove), jednu za drugom, bez “rupa”,
- polje pokazivača na znakove p — tako da $p[i]$ sadrži pokazivač na početak $i+1$ -e riječi u polju w .

Radi jednostavnosti, pretpostavit ćemo još da

- imamo najviše 100 riječi.

To je maksimalni broj elemenata — pokazivača u polju p .

Uz ove pretpostavke, polja w i p onda možemo definirati

- kao polja fiksne dimenzije.

(Dinamičko alociranje ovih polja — sami, v. vježbe.)

Sortiranje rječnika — spremanje riječi (nast.)

Izgled polja w i p :

- Prva riječ počinje na početku polja w , tj. na mjestu $w[0]$, pa je $p[0] = \&w[0] = w$.
- Uzmimo da prva riječ zauzima ukupno n_1 znakova.
 - Oprez: $n_1 = \text{strlen}$ od prve riječi $+ 1$, zbog nul-znaka na kraju prve riječi.
- Druga riječ ide odmah iza prve. Dakle, počinje na mjestu $w[n_1]$, pa je $p[1] = \&w[n_1] = w + n_1$.

I tako redom.

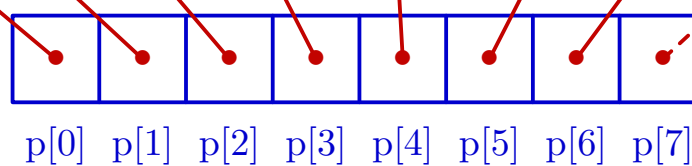
Sad bi došla slika kako to izgleda (v. sljedeća stranica).

Spremanje riječi u rječniku

polje w

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|----|---|---|---|----|---|---|---|---|----|---|---|---|---|---|----|---|---|---|---|---|---|----|---|---|---|----|---|---|---|---|----|----|---|
| | | | | | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 |
| J | a | \0 | s | a | m | \0 | m | a | l | a | \0 | R | u | z | a | , | \0 | m | a | m | i | n | a | \0 | s | a | m | \0 | k | c | i | . | \0 | \0 | |

polje p



Sortiranje rječnika — detalji implementacije

Radi jednostavnosti, **oba** polja **w** i **p** definiramo kao **globalna** (statička). Stvarno,

- samo **w** **koristimo** kao **globalno** polje,
- dok je **p** “uredno” **argument** svih funkcija.

Dogovor: da ne bude “prejednostavno”,

- **stvarni** broj riječi **nije** unaprijed zadan, već
- čitamo riječi **sve dok** ne učitamo **prazan red** (praznu riječ, tj. string duljine **0** — poput ***p[7]** na prošloj slici).

Za **sortiranje** koristimo:

- sortiranje **izborom ekstrema** (v. **Prog1**).

U nastavku je cijeli program **dicts_1.c**.

Sortiranje rječnika — početak programa

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* Sortiranje rjecnika. */

#define MAXBROJ 100
#define MAXDULJ 80

/* MAXBROJ = max. broj rijeci,
   MAXDULJ = max. duljina pojedine rijeci. */
```


Sortiranje rječnika — globalne definicije

```
/* Globalno polje znakova za rjecnik. */  
  
char w[MAXBROJ * MAXDULJ];  
  
/* Globalno polje pokazivaca na  
   pojedine rijeci - stringove. */  
  
char *p[MAXBROJ];  
  
/* Stvarni broj rijeci u rjecniku. */  
  
int broj_rijeci;
```

Sortiranje rječnika — učitavanje rječnika

```
/* Ucitava rijeci i vraca broj rijeci.  
   Kraj ucitavanja je prazna rijec. */  
int unos(char *p[])  
{  
    char *q = w;  
    int radi = 1, broj = 0, dulj;  
    while (radi) {  
        if (broj >= MAXBROJ) return -1;  
        p[broj] = gets(q);  
        if ((dulj = strlen(q)) == 0) break;  
        q += dulj + 1;  
        ++broj;  
    }  
    return broj;  
}
```

Sortiranje rječnika — sortiranje

```
/* Sortiranje rječnika izborom ekstrema.  
   Dovodimo najmanju riječ na početak.  
   Koristimo samo zamjene pokazivaca. */
```

```
/* Polje pokazivaca na znakove-stringove. */
```

```
void sort(char *p[], int n)  
{  
    int i, j, ind_min;  
    char *temp;
```

Sortiranje rječnika — sortiranje (nastavak)

```
for (i = 0; i < n - 1; ++i) {
    ind_min = i;
    for (j = i + 1; j < n; ++j)
        if (strcmp(p[j], p[ind_min]) < 0)
            ind_min = j;
    if (i != ind_min) {
        temp = p[i];
        p[i] = p[ind_min];
        p[ind_min] = temp;
    }
}

return;
}
```

Sortiranje rječnika — ispis rječnika

```
/* Ispisuje sve rijeci u rjecniku. */  
  
void ispis(char *p[])  
{  
    int i;  
  
    for (i = 0; i < broj_rijeci; ++i)  
        puts(p[i]);  
  
    return;  
}
```

Sortiranje rječnika — glavni program

```
/* Glavni program. */

int main(void)
{
    if ((broj_rijeci = unos(p)) >= 0) {
        printf("Broj rijeci = %d\n", broj_rijeci);
        sort(p, broj_rijeci);
        ispis(p);
    }
    else
        printf("Previše rijeci na ulazu.\n");

    return 0;
}
```

Sortiranje rječnika — primjer ulaza

Ulaz:

Ja
sam
mala
Ruza,
mamina
sam
kci.

Sortiranje rječnika — izlaz

Izlaz:

Broj rijeci = 7

Ja

Ruza,

kci.

mala

mamina

sam

sam

Sortiranje rječnika — poboljšanje unosa

Funkcija `unos` strogo očekuje praznu riječ kao kraj ulaza i javlja grešku ako pokušamo učitati preveliki broj riječi.

Modificirana verzija funkcije `unos` kontrolira sve što može.

- Uredno prekida čitanje ako učitamo maksimalni broj riječi (prije prazne) i to nije greška. U ostatku programa radimo samo s riječima koje su “stale” u rječnik (polje `p`).
- Provjerava je li `gets` uspješno učitao string, ili je vratio `NULL` (što je signal greške), i samo tad javlja grešku.

Jedina modifikacija u glavnom programu je poruka o grešci.

Još bolje bi bilo koristiti funkciju `fgets` za kontrolu broja učitanih znakova u string (v. kasnije, kod datoteka).

Sortiranje rječnika — modificirani unos

```
/* Ucitava rijeci i vraca broj rijeci.  
   Kraj ucitavanja je prazna rijec. */  
int unos(char *p[])  
{  
    char *q = w;  
    int radi = 1, broj = 0, dulj;  
    while (radi) {  
        if (broj >= MAXBROJ) break; // polja p, w!  
        if (gets(q) == NULL) return -1;  
        if ((dulj = strlen(q)) == 0) break;  
        p[broj++] = q;  
        q += dulj + 1;  
    }  
    return broj;  
}
```

Sortiranje rječnika — pripadni glavni program

```
/* Glavni program. */

int main(void)
{
    if ((broj_rijeci = unos(p)) >= 0) {
        printf("Broj rijeci = %d\n", broj_rijeci);
        sort(p, broj_rijeci);
        ispis(p);
    }
    else
        printf("Greska u citanju rijeci.\n");

    return 0;
}
```

Sortiranje rječnika — varijacije na temu

U prethodnom programu, polja **w** i **p** imaju **fiksnu** dimenziju (zadali smo **maksimalni broj** riječi i **maksimalnu duljinu** riječi).

Varijacija 1. Promijenite program tako da se polja **w** i **p**

- **dinamički alociraju** i, po potrebi, **realociraju** (ako treba povećati “duljinu” nekog polja).

Varijacija 2. Umjesto polja **w**, koristite **dinamičku alokaciju** za **svaku** učitano **riječ**:

- ako se zna **maksimalna duljina** riječi,
- isto, s tim da maksimalna **duljina** riječi **nije** zadana (ili ograničena).

Pažljivo **kontrolirajte** da ne “**gazite**” po memoriji!

Sortiranje rječnika — zadatak

Zadatak. Prilagodite i iskoristite **QuickSort** algoritam za **sortiranje** rječnika.

Oprez: funkcija **swap** treba zamijeniti vrijednosti **pokazivača** na **znakove** (stringove).

🕒 **Argumenti** funkcije **swap** moraju biti **pokazivači** na te **pokazivače**, tj. tip argumenata je **char ****.

Zadatak. Za **sortiranje** rječnika iskoristite funkciju **qsort** iz standardne biblioteke **<stdlib.h>** (treba znati pokazivač na funkciju). **Uputa:** Pogledajte primjer na zadnjem predavanju i **pazite** na **tipove**, kao za **swap**!

Razmislite **kako** biste iskoristili **qsort** da radimo **zamjene** cijelih **stringova**, a **ne** pripadnih **pokazivača**.

Argumenti komandne linije

Argumenti komandne linije

Programi vrlo često koriste parametre koji se zadaju zajedno s imenom programa i onda se učitavaju iz tog programa. Takvi parametri zovu se argumenti komandne linije.

```
cp ime1 ime2
```

Ako želimo koristiti argumente komandne linije, onda moramo funkciju `main` deklarirati s dva formalna argumenta (ne `void`). Standardna imena za te argumente su (prema KR2):

• `argc` — tipa `int`, i

• `argv` — polje pokazivača na `char` (= polje stringova).

```
int main(int argc, char *argv[])  
{ ... }
```

Argumenti komandne linije (nastavak)

Značenje `argc` (“argument count”):

- Broj `argc - 1` je broj argumenata komandne linije. Ako ih nema, onda je `argc = 1`.

Značenje `argv` (“argument value”):

- U `argv` je polje pokazivača na argumente komandne linije.
- `argv[0]` uvijek pokazuje na string koji sadrži ime programa, onako kako je pozvan na komandnoj liniji.
- Ostali parametri smješteni su redom kojim su upisani.
- Iza svega je `argv[argc] = NULL` (“za svaki slučaj”).

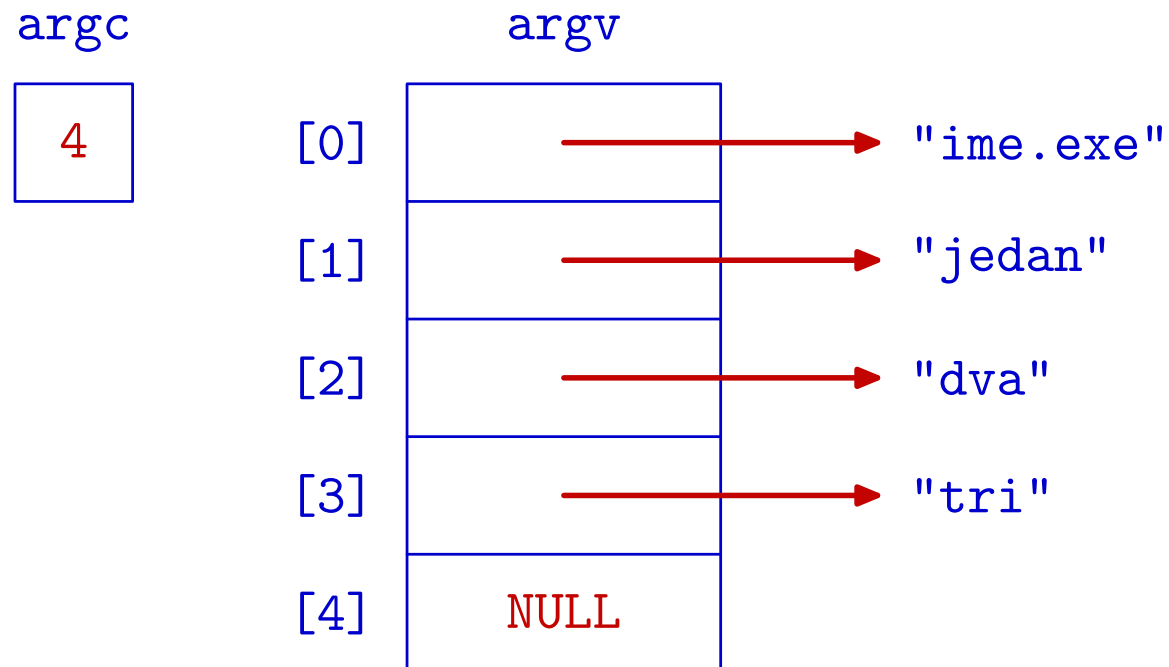
Argumenti se “čitaju” kao stringovi u funkciji `scanf`, tj. bjeline su separatori (osim kad je argument u navodnicima).

Argumenti komandne linije (nastavak)

Primjer. Ako program pozovemo s:

```
ime.exe jedan dva tri
```

onda je:



Argumenti komandne linije (nastavak)

Primjer. Program koji ispisuje argumente komandne linije:

```
#include <stdio.h>  /* program arg_1 */

int main(int argc, char *argv[])
{
    int i;

    printf("argc = %d\n", argc);
    for (i = 0; i < argc; ++i)
        printf("argv[%d]: %s\n", i, argv[i]);

    return 0;
}
```

Argumenti komandne linije (nastavak)

Komandna linija (izvršavanje u komandnom prozoru!):

```
arg_1 Ja sam mala Ruza, mamina sam kci.
```

Izlaz:

```
argc = 8  
argv[0]: arg_1  
argv[1]: Ja  
argv[2]: sam  
argv[3]: mala  
argv[4]: Ruza,  
argv[5]: mamina  
argv[6]: sam  
argv[7]: kci.
```

Čitanje brojeva s komandne linije

Ranije smo napravili program koji s **ulaza** čita broj **n**, a zatim **dinamički** kreira **polje a** s **n** elemenata.

Primjer. Broj **n** možemo **učitati** i s **komandne linije**, samo ga treba **pretvoriti** iz **stringa** u tip **int** (funkcija **atoi**).

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

int main(int argc, char *argv[])
{
    ...
    n = atoi(argv[1]);
    ...
}
```

Provjera argumenata komandne linije

Napomena. Ako program **očekuje** neke **argumente** na komandnoj liniji, onda **uvijek** treba **provjeriti**:

- jesu li zaista **upisani** pri pozivu programa — test **argc**,
- imaju li svi **argv[i]** **korektan** (očekivani) oblik.

U prošlom primjeru to bi izgledalo ovako:

```
if (argc < 2) {  
    printf("Broj elemenata nije zadan.\n");  
    exit(-1); }  
n = atoi(argv[1]);  
if (n <= 0) {  
    printf("Broj elemenata nije pozitivan.\n");  
    exit(-1); }
```

Razlika:

2D polje i niz nizova

Razlika — dvodimenzionalno polje i niz nizova

Prijenos **dvodimenzionalnog** polja (matrice) kao argumenta funkcije odgovara **deklaraciji** s **pokazivačem**

```
tip_pod (*ime)[izraz_2]
```

Ovdje je **ime** = **pokazivač** na (prvi) **redak**, tj.

🕒 **pokazivač** na **polje** od **izraz_2** elemenata tipa **tip_pod**, odnosno (ekvivalencija pokazivač — polje),

🕒 **ime** = **polje redaka** matrice.

Razlika — dvodimenzionalno polje i niz nizova

Nažalost, to se često “miješa” s deklaracijama oblika

```
tip_pod **ime  
tip_pod *ime[]
```

Ovdje je `ime` = pokazivač (ili polje pokazivača) na objekt(e)

- tipa `pokazivač` na `tip_pod`, odnosno,

- tipa “neki `niz` (ili neko `polje`) objekata tipa `tip_pod`”.

Međutim, ovo `nije` matrica, nego `polje nizova` ili `niz nizova`!

Razlika: kod sortiranja rječnika i argumenata komandne linije

- radimo s `nizom nizova` znakova, a ne s matricom.

Pogledajte `vježbe` — poglavlje 4.3.

Razlika — dvodimenzionalno polje i niz nizova

Ključna razlika je u tome što se stvarno pamti:

- ☛ Za 2D polje (matricu), pamti se samo jedan pokazivač — onaj na prvi redak, tj. na cijelo polje redaka matrice. Matrica je spremljena “u bloku” — redak za retkom.
- ☛ Za polje nizova, pamti se cijelo polje pokazivača — na svaki pojedini niz u polju nizova.

Pojedini nizovi ne moraju imati iste duljine i mogu biti “razbacani” po memoriji — svaki niz u svom bloku.

Razlog za zabunu = pristup elementima se jednako piše:

- ☛ `ime[i][j]` = j-ti element u i-tom retku matrice,
- ☛ `ime[i][j]` = j-ti element u i-tom nizu.

Međutim, računanje stvarnih adresa je bitno drugačije!

Razlika — dvodimenzionalno polje i niz nizova

Ako “pobrkamo” matrice i nizove nizova, to može dovesti do

- teško uočljivih **grešaka** i “gaženja” memorije.

Naime, neku **matricu** *A* (na primjer, brojeva),

- zaista **možemo** spremiti u strukturu “**niza nizova**”, recimo kao “**niz redaka**” (tj. nizova iste duljine).

S tom strukturom **možemo** raditi kao s “matricom”, u smislu

- da pristup elementu pišemo kao *A*[*i*][*j*], s dva indeksa.

Međutim, tako **spremljena** matrica *A* **nije** 2D polje i

- ne smije** se poslati kao argument funkciji koja očekuje “pravu” matricu — **spremljenu** u obliku 2D polja!

Zato **oprez**, kad koristite pakete programa za matrice.

Pokazivač na funkciju

Pokazivač na funkciju

Pokazivač na funkciju deklarira se kao:

```
tip_pod (*ime)(tip_1 arg_1, ..., tip_n arg_n);
```

Ovdje je `ime` varijabla tipa — `pokazivač` na `funkciju`, koja

- uzima `n` argumenata tipa `tip_1` do `tip_n`

- i vraća vrijednost tipa `tip_pod`.

Slično kao i u `prototipu` funkcije, `ne treba` pisati `imena` argumenata `arg_1` do `arg_n`.

Primjer:

```
int (*pf)(char c, double a);  
int (*pf)(char, double);
```

Pokazivač na funkciju (nastavak)

U deklaraciji **pokazivača** na **funkciju** — **zagrade** su nužne.

- **Primarni** operator **()** — “poziva” ili argumenata funkcije, ima viši prioritet od **unarnog** operatora *****.

Primjer. Razlikujte **funkciju** koja vraća **pokazivač** na neki tip (ovdje, na **double**):

```
double *pf(double, double);  
double *(pf(double, double));  /* Isto */
```

od **pokazivača** na **funkciju** koja vraća **vrijednost** nekog tipa (ovdje, tipa **double**):

```
double (*pf)(double, double);
```

Pokazivač na funkciju — svrha

Pokazivač na funkciju omogućava da jedna funkcija prima neku drugu funkciju kao argument. Realizacija ide tako da

- prva funkcija dobiva pokazivač na drugu funkciju.

```
int prva(int, int (*druga)(int));
```

U pozivu prve funkcije navodimo samo stvarno ime druge funkcije (koja negdje mora biti deklarirana s tim imenom), tj.

- ime funkcije je sinonim za pokazivač na tu funkciju.

```
prva(n, stvarna_druga);
```

Cijela stvar je vrlo slična onoj za polja!

Pokazivač na funkciju — primjer

Primjer. Treba izračunati vrijednost **integrala** zadane (realne) funkcije f na **segmentu** $[a, b]$

$$I = \int_a^b f(x) dx.$$

Za **računanje** integrala obično se koriste **približne** (numeričke) formule. Slično Riemannovim sumama, te formule koriste

• **vrijednosti** funkcije f u određenim točkama iz $[a, b]$.

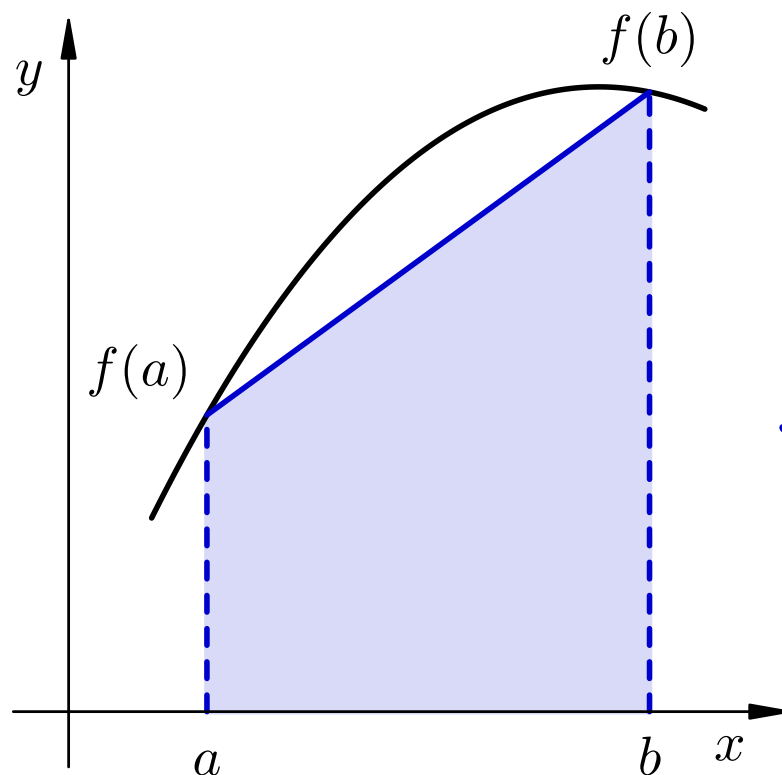
Funkcija za (približno) računanje integrala I onda **mora** imati (barem) **3** argumenta:

• **granice** integracije a , b , i

• **funkciju** koja računa vrijednost $f(x)$ u zadanoj točki x .

Primjer — trapezna formula

Primjer. Napisati funkciju koja po **trapeznoj** formuli približno računa **integral** zadane funkcije. **Trapezna** formula ima oblik:



$$\int_a^b f(x) dx \approx \frac{f(a) + f(b)}{2} (b - a).$$

Trapezna formula — program

```
#include <stdio.h>
#include <math.h>
double integracija(double, double,
                   double (*)(double));

int main(void) {
    printf("Sin: %f\n", integracija(0, 1, sin));
    printf("Cos: %f\n", integracija(0, 1, cos));
    return 0; }

double integracija(double a, double b,
                   double (*f)(double)) {
    return 0.5 * (b - a) * ( (*f)(a) + (*f)(b) );
}
```

Trapezna formula — rezultati i komentar

Uočite da u **pozivima** funkcije **integracija** navodimo

📌 **samo imena** funkcija **sin** i **cos**

iz matematičke biblioteke **<math.h>**.

Rezultati:

Sin: 0.420735

Cos: 0.770151

Točnost ovih približnih vrijednosti i “**nije nešto**”. Provjerite!

Puno **bolju točnost** možemo postići korištenjem tzv.

📌 **produljene trapezne** formule.

Produljena trapezna formula — ideja

Produljena trapezna formula za približnu integraciju.

- Izaberemo prirodni broj $n \in \mathbb{N}$.
- Segment $[a, b]$ podijelimo na n podintervala tačkama x_i , za $i = 0, \dots, n$, tako da je

$$a = x_0 < x_1 < \dots < x_{n-1} < x_n = b.$$

Pripadni podintervali su $[x_{i-1}, x_i]$, za $i = 1, \dots, n$.

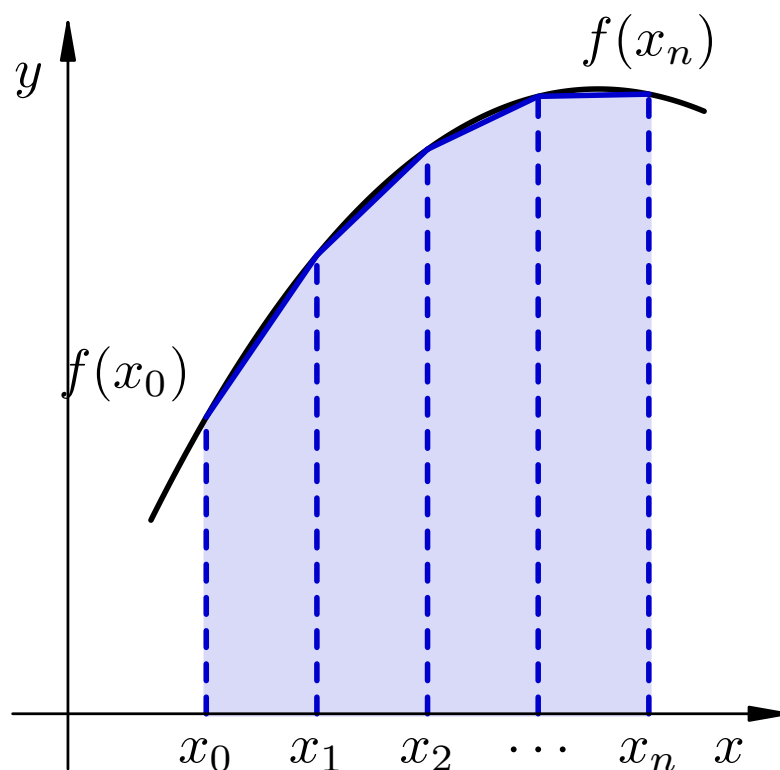
- Zato vrijedi

$$I = \int_a^b f(x) dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x) dx.$$

- Na svakom podintervalu $[x_{i-1}, x_i]$, za $i = 1, \dots, n$, iskoristimo trapeznu formulu i sve zbrojimo.

Produljena trapezna formula

Obično se točke x_i uzimaju **ekvidistantno**, tako da svi **podintervali** imaju **jednaku** duljinu — označimo ju s h .



Onda je **duljina** podintervala

$$h = \frac{b - a}{n},$$

a **točke** su

$$x_i = a + i \cdot h, \quad i = 0, \dots, n.$$

Produljena trapezna formula — formula

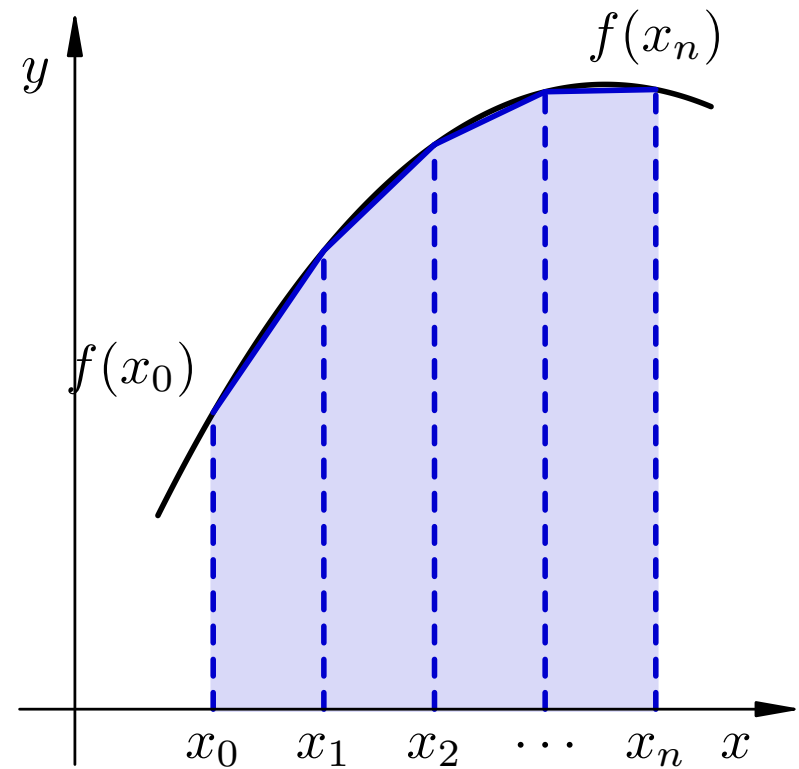
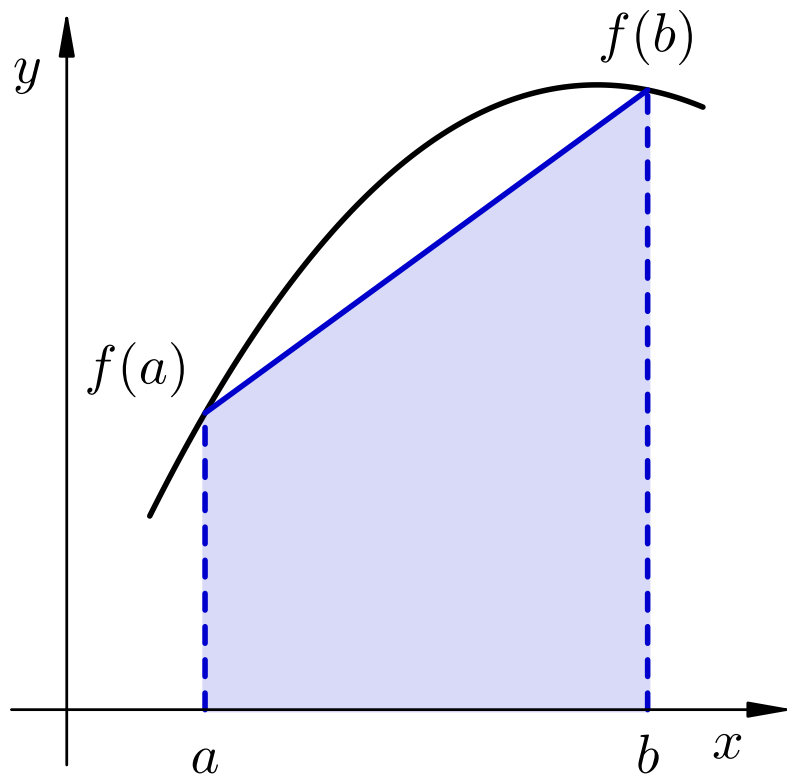
Obična trapezna formula na podintervalu $[x_{i-1}, x_i]$ daje

$$\int_{x_{i-1}}^{x_i} f(x) dx \approx \frac{f(x_{i-1}) + f(x_i)}{2} (x_i - x_{i-1}) = \frac{h}{2} (f(x_{i-1}) + f(x_i)).$$

Kad zbrojimo za sve podintervale, izlazi

$$\begin{aligned} I &\approx \sum_{i=1}^n \frac{h}{2} (f(x_{i-1}) + f(x_i)) \\ &= \frac{h}{2} (f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)) \\ &= h \left(\frac{1}{2} (f(a) + f(b)) + \sum_{i=1}^{n-1} f(a + ih) \right). \end{aligned}$$

Obična i produljena trapezna formula — slike



Povećavanjem n dobivamo sve bolju aproksimaciju integrala.

Produljena trapezna formula — deklaracije

Primjer. Program za **produljenu** trapeznu formulu, koji povećava broj podintervala n (v. `integ_2.c`):

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

```
#include <math.h>
```

```
double integracija(double, double, int,  
                   double (*)(double));
```

Produljena trapezna formula — funkcija

```
double integracija(double a, double b, int n,  
                   double (*f)(double))  
{  
    double suma, h = (b - a) / n;  
    int i;  
  
    suma = 0.5 * ( (*f)(a) + (*f)(b) );  
    for (i = 1; i < n; ++i)  
        suma += (*f)(a + i * h);  
    return h * suma;  
}
```


Produljena trapezna formula — glavni program

```
int main(void)
{
    double a = 0.0, b = 2.0 * atan(1.0);  /* pi/2 */
    int n = 1;

    printf("Integral sinusa od 0 do pi/2:\n");
    while (n <= 100000) {
        printf(" [n = %6d]: %13.10f\n", n,
               integracija(a, b, n, sin));
        n *= 10;
    }
    return 0;
}
```

Produljena trapezna formula — rezultati

Izlaz:

Integral sinusa od 0 do $\pi/2$:

[n = 1]: 0.7853981634

[n = 10]: 0.9979429864

[n = 100]: 0.9999794382

[n = 1000]: 0.9999997944

[n = 10000]: 0.9999999979

[n = 100000]: 1.0000000000

Ha! Nije loše! Zadnja vrijednost je točna na svih 10 decimala.

Više o metodama za numeričku integraciju u Numeričkoj matematici.

Ime funkcije = pokazivač na tu funkciju

Uputa: kako koristiti **pokazivač** na funkciju — kod **stvarnog** i **formalnog** argumenta neke funkcije (v. KR2, str. 118–120).

Za propisno **deklariranu** funkciju,

📌 **ime** funkcije je **sinonim** za **pokazivač** na **tu funkciju**.

Zato, na primjer, u **pozivu** funkcije **integracija** navodimo **samo ime** funkcije **sin** — kao **stvarni** argument:

```
i_sin = integracija(0, 1, sin);
```

Adresni operator **&** ispred **sin** **nije potreban** (kao ni ispred imena polja), iako ga je **dozvoljeno** napisati:

```
i_sin = integracija(0, 1, &sin);
```

Poziv funkcije zadane pokazivačem

Isto **smijemo** napraviti i kod **poziva** funkcije koja je **formalni** argument — zadan **pokazivačem**, unutar neke druge funkcije.

🚫 Međutim, **nemojte** to raditi!

Na primjer, **f** je **formalni** argument funkcije **integracija**:

```
double integracija(double a, double b,  
                  double (*f)(double)) {  
    return 0.5 * (b - a) * ( (*f)(a) + (*f)(b) );  
}
```

Strogo govoreći, tj. pazeći na **tipove** objekata,

🚫 **f** je **pokazivač** na funkciju,

🚫 a ***f** je **funkcija**, nakon **dereferenciranja**!

Poziv funkcije zadane pokazivačem (nastavak)

U tom smislu, prošli primjer je potpuno **korektno** napisan.

Naprotiv, iako je **dozvoljeno**, **nije** sasvim **korektno** napisati

```
double integracija(double a, double b,  
                  double (*f)(double)) {  
    return 0.5 * (b - a) * ( f(a) + f(b) ); }  

```

Analogno, **dozvoljeno** napisati i ovo (ekvivalentno gornjem):

```
double integracija(double a, double b,  
                  double (*pf)(double)) {  
    double (*f)(double) = pf;  
    return 0.5 * (b - a) * ( f(a) + f(b) ); }  

```

Pokazivač na funkciju i obrada znakova (zadaci)

Pokazivač na funkciju — dodatni primjeri

Pokazivač na funkciju može se **zgodno** iskoristiti kod obrade **znakova** u stringu (v. prošlo predavanje).

Vrsta znakova koju želimo obraditi (na pr. **samoglasnici**) zadana je odgovarajućom **funkcijom** oblika

```
int vrsta(int c);
```

Takva funkcija **provjerava** pripada li zadani **znak c** određenoj **vrsti znakova** ili ne (kao funkcije iz **<ctype.h>**). Nekoliko takvih funkcija smo već napisali.

Funkcija za **obradu takvih znakova** u zadanom **stringu** dobiva (osim stringa) i **pokazivač** na **funkciju** za provjeru znakova:

```
int (*vrsta)(int c)
```

Pokazivač na funkciju — zadaci

Zadaci. Napišite funkciju koja, kao argumente, prima `string` (tj. pokazivač na `char`) i `pokazivač` na funkciju za provjeru znakova, i `radi` sljedeće:

1. vraća `broj` takvih znakova u stringu,
2. to isto, a kroz `varijabilni` argument vraća `prvi` takav znak u stringu, ako ga ima (u protivnom, ne mijenja taj argument),
3. vraća `pokazivač` na `prvi` takav znak u stringu, ako ga ima (u protivnom, vraća `NULL`),
4. to isto, a kroz `varijabilni` argument vraća `broj` takvih znakova u stringu,
5. vraća `pokazivač` na `zadnji` takav znak u stringu, ako ga ima (u protivnom, vraća `NULL`).