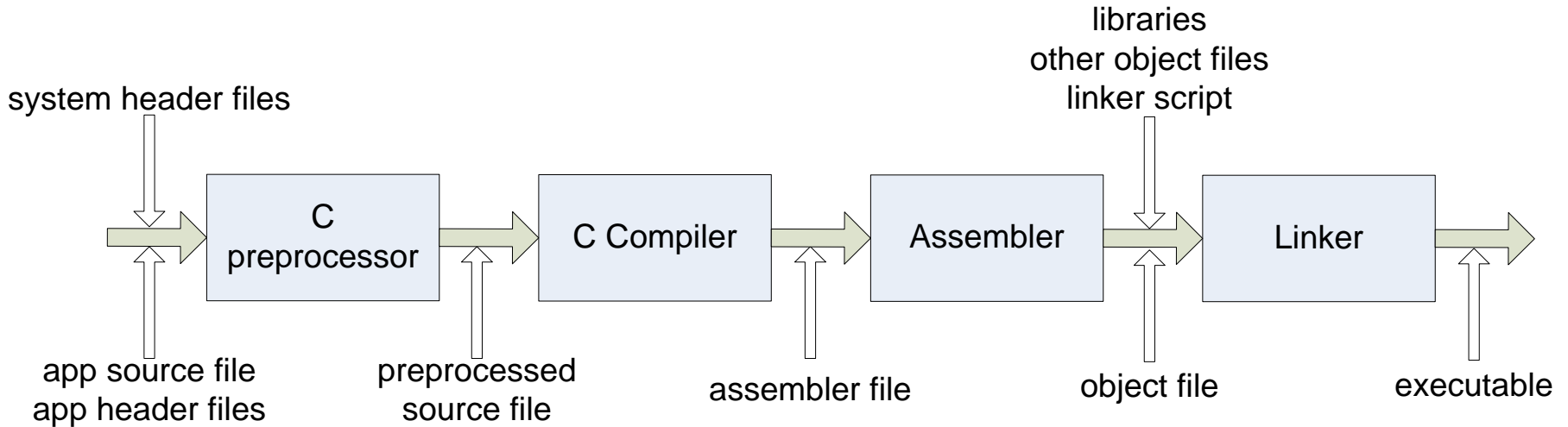




Билд процес

Ток билдовања



name.c – изворне датотеке

name.i – претпроцесиране изворне датотеке

name.s – асемблерске датотеке

name.o – објектне датотеке

name - извршна датотека



Претпроцесор

- Први корак у превођењу
- Претпроцесирање мења изворани код пре него што почне превођење
- Представља моћан алат. Практично језик у језику.
- Може се користити за различите ствари, али увек се треба трудити да се не претерује, јер код пре свега треба да буде јасан!
- Неке од ствари за које се претпроцесорске директиве најчешће користе:
 - Укључивање заглавља
 - Условно превођење
 - Именовање литерала
 - Брзи „позиви“ кратких функција



Излаз из претпроцесирања

- Код многих компајлера могуће је излистати у датотеку изглед изворног кода након претпроцесирања
- То је некада корисно за правилно тумачење кода и дебаговање претпроцесирања
- Код GCC-а опција `-E` наређује компајлеру да излиста претпроцесиран код:

```
gcc -E file_name.c -o file_name.i
```

- Претпроцесирани код ће бити уписан у датотеку `file_name.i`

```
app.h :  
  
int extfunc(int a);
```

```
app.c:  
  
#include "app.h"  
#define MAX 30  
  
int func(int a)  
{  
    if (a < extfunc(a>>2))  
        return -1;  
#ifdef LIMIT  
    if (a > MAX)  
        return 1;  
#endif  
    return MAX;  
}
```

```
# 1 "app.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "app.c"  
  
# 1 "app.h" 1  
int extfunc(int a);  
# 3 "app.c" 2  
  
int func(int a)  
{  
    if (a < extfunc(a>>2))  
        return -1;  
  
    return 30;  
}
```



#include 1/2

- Можда и најважнија команда претпроцесирања. Без ње би се тешко могао замислити било који сложенији пројекат.
- Употреба #include директиве:

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

- На месту инклюд директиве након претпроцесирања биће уписан комплетан садржај датотеке која се наводи
- Датотека уопште не мора бити заглавље, већ било шта. Заглавље као концепт је пре свега програмерска конвенција, а не нешто наметнуто од стране претпроцесора.
- Ако је путања датотеке дата у <> заградама онда ће компајлер прво гледати у системским директоријумима. На овај начин се спречава да корисничка датотека истог имена замаскира системску.
- GCC подразумевано гледа у следећим путањама:

```
/usr/local/include  
libdir/gcc/target/version/include  
/usr/target/include  
/usr/include
```



#include 2/2

- Уколико је путања дата под наводницима, онда компајлер прво гледа по корисничким путањама, а затим по системским.
- Компајлери обично омогућују додавање корисничких путања до заглавља. У GCC-у, а и у многим другим компајлерима, ово је команда којом се саопштавају путање до заглавља: **-Ifolder_name**

```
gcc -c -I../inc -I../libinc ../src/app.c -o app.o
```

- Све путање задате -I командом се у случају наводника претражују пре системских путања
- Чак је могуће коришћењем **-nostdinc** опције натерати GCC да не претражује системске путање уопште

```
gcc -nostdinc ../src/app.c
```

- Дакле, у случају употребе наводника ово је редослед претраге:
 - Путања на којој се налази .c датотека
 - Путање задате -I опцијом
 - Стандардне путање (уколико -nostdinc није наведено)



#define - Макрои

- Две врсте макроа:
 - Макро-објекти
 - Макро-функције
- Ситнакса:

Макро-објекти:

```
#define MACRO_NAME [replacement token list]
```

Макро-функције:

```
#define MACRO_NAME(arg1, arg2, ...) [replacement token list]
```

- Где год се MACRO_NAME појави у коду биће замењено од стране претпроцесора са „replacement token list”.
- „Replacement token list” може бити празна
- Употреба макро-функција мора имати заграде
- Уобичајено је да се макрои пишу великим словима да би одмах било јасно да се ради о макроу, а не о редовном идентификатору



Макро-објекти

- Макро-објекти се користе најчешће за именовање литерала
- Пример:

```
#define DELAY 50
```

- Именовање литерала је погодно из више разлога:
 - Име може далеко боље описати смисао литерала/константе
 - Уколико настане потреба за променом литерала, а он се користи на више места, лакше је променити вредност само на једном месту
 - Јасније истиче употребу литерала

```
for (i = 0; i < 5000; i++)  
{  
    /* loop body */  
    usleep(5000);  
}
```

```
for (i = 0; i < NUM_ITER; i++)  
{  
    /* loop body */  
    usleep(DELAY);  
}
```

- У C99 стандарду је уведен `const` квалификатор и обично је боље њега користити

Макро-функције

- Омогућавају да макрои буду параметризовани

```
#define UPOZNAJ(X) Ja sam X.  
UPOZNAJ(Djura)  
Ja sam Djura.
```

```
#define RADTODEG(X) ((X) * 57.29578)
```

- Макро-функције нису праве функције и не троше се инструкције на њихов позив
- За разлику од редовних функција, код макро-функција не сме постојати размак између имена и отворене заграде

```
#define RADTODEG (X) ((X) * 57.29578)  
RADTODEG(2)
```

Биће сведено на:

$(X) ((X) * 57.29578) (2)$

не на:

$((2) * 57.29578)$

- Ово важи само за дефиницију, не за употребу

`RADTODEG (90)` је исто што и `RADTODEG(90)`



Макори и приоритет операција



- Макро-функције нису праве функције!
- Оне само дефинишу трансформације текста
- Ево једног проблема који се јавља ако примењујемо логику редовне функције на макро-функције

```
#define RADTODEG(X) X * 57.29578
```

```
C / RADTODEG(A+B)
```

Биће сведено на:

```
C / A+B * 57.29578
```

- Оператор множења има предност у односу на оператор сабирања, па у овом случају долази до нежељеног резултата
- Решење је да се сваки параметар макро-функције стави у заграду, као и цела макро-функција

```
#define RADTODEG(X) ((X) * 57.29578)
```

```
C / RADTODEG(A+B)
```

Биће сведено на:

```
C / ((A+B) * 57.29578)
```



Макро-функције и бочни ефекти



- Евалуација бочних ефеката аргумената може бити врло другачија код макро-функција него код правих функција
- Пример:

```
#define MIN(a, b) ((a)>(b) ? (b) : (a))
```

- Употребимо ову макро-функцију на следећи начин:

```
MIN(++x, y)
```

Биће сведено на:

```
((++x)>(y) ? (y) : (++x))
```

- Уколико је променљива **x** једнака или већа од променљиве **y**, тада ће она приликом евалуације овог израза бити увећана два пута!
- При позиву редовне функције повећање ће се обавити само једном.
- За овакве употребе макроа можда је боље употребити `inline` функције које постоје од C99 стандарда. Уколико дефинишемо `static inline` функцију биће уметнута на сваком месту на којем се зове, а дефиниција неће бити генерисана. Дакле, као макро, само мало боље и безбедније.



; и макрои

- Обично не ваља стављати ; на крај дефиниције макроа
- Рецимо:

```
#define MIN(a,b) ((a)>(b) ? (b) : (a));
```

```
if (MIN(x,y) > 0)
```

Биће сведено на:

```
if ((x)>(y) ? (y) : (x)); > 0) /* SYNTAX ERROR */
```

- Или:

```
#define DEBUG(msg) printf(msg);
```

```
if (ret == 0)
    DEBUG("Success");
else
    DEBUG("Failed");
```

Биће сведено на:

```
if (ret == 0)
    printf("Success"); /* two statements */
else
    printf("Failed"); /* else not associated with if */
```

#if 1/3

- Омогућава различито превођење кода у зависности од неких параметара.
- Следеће директиве то омогућавају:
 - #if, #ifdef, #ifndef, #else, #elif
- Сви блокови започети са #if, #ifdef или #ifndef морају бити завршени са одговарајућим #endif

syntax:

```
#if expression  
    text  
#endif
```

- Израз је Цеоовски израз али са следећим ограничењима:
 - Само интеџерске и знаковне константе/литерали - јер резултат израза мора бити срачунљив током компајлирања
 - Оператори за: сабирање/одузимање, множење/делење, битски оператори, шифтовања, поређења, логички оператори
 - Макрои



#if 2/3

- Пример

```
#if DEBUG > 2
    printf("Debug message\n");
#endif
```

- Само интеџери (и знаковни литерали, који се свде на интеџере):

```
#if DEBUG == "on" /* ERROR */
    printf("Debug message 1\n");
#endif

#if DEBUG > 2.0 /* ERROR */
    printf("Debug message 2\n");
#endif
```

- Може се користи и за искључивање дела кода из процеса превођења

```
#if 0
    /* code */
#endif
```

- Само као привремено решење. Ако постоји могућност да ће искључени код бити потребан у будућности, боље је ослонити се на систем за контролу верзија.
- Обично постоји могућност и да се симболи дефинишу споља, кроз параметре компајлера.

```
gcc -DDJURA -DDEBUG -DPERA=14 -DMILE=(PERA-2) file_name.c
```



#if 3/3

- Још једна употреба условног превођења је да се избегне вишеструко укључивање истих датотека
- Укључена заглавља могу укључивати друга заглавља, тако да је тешко строго контролисати шта ће и колико пута бити укључено
- За ту стврху су корисне директиве `#ifndef` и `#define`

```
#ifndef _FILE_NAME_H
#define _FILE_NAME_H

/* code */

#endif
```

- Слична конструкција се може употребити да се спречи вишеструко дефинисање макроа

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



Спајање токена/симбола

- Коришћењем оператора **##** могуће је спојити два токена/симбола
- Пример:

```
#define DESCRIPTOR_FIELD(field) struct1.union1.m_##field
```

```
DESCRIPTOR_FIELD(polje)
```

Биће сведено на:

```
struct1.union1.m_polje
```

```
#define STR_AB djura##pera
```

```
x = STR_AB; -> x = djurapera;
```

али

```
x = djura##pera -> error
```




Третирање токена као знаковних низова

- Могуће је неки токен/симбол тертирати као знаковни низ
- Корисно када желимо исписати симбол
- Користи се # као унарни оператор
- Пример:

```
#define TOKEN(token) printf("#token " = %d\n", token)
```

```
TOKEN (x+y);
```

Биће сведено на:

```
printf("x+y" " = %d\n", x+y)
```

```
#define STR_T #djura
```

```
printf(STR_T); -> printf("djura");
```

али

```
printf(#djura); -> error!
```

Предефинисани макрои

- Листа макроа који су предефинисани:

| Назив макроа | Опис |
|--------------|------------------------------|
| __DATE__ | Датум компајлирања |
| __LINE__ | Линија у коду |
| __FILE__ | Назив датотеке |
| __TIME__ | Време компајлирања |
| __STDC__ | Подаци о подржаном стандарду |



#error и #warning

- Коришћењем ових директива могуће је наредити компајлеру да пријави грешку или упозорење при превођењу

```
#error "Error message"
```

- Исписује текст грешке и прекида превођење

```
#warning "Warning message"
```

- Исписује текст упозорења и наставља превођење. Ова директива није део стандарда, али већина компајлера је подржава.
- Пример:

```
#ifdef WIN32
/* WIN32 specific code */
#elif defined ( linux )
/* linux specific code */
#warning "Linux version not fully supported"
#else
#error "Not supported OS"
#endif
```



Компајлер у билд процесу



- Преводи Це код у асемблерски код
- Пример позива GCC-а

```
gcc -S file_name.c -o file_name.s
```

- Опција -S наређује компајлеру да генерише асемблерски код, у супротном биће генерисана објектна датотека.
- GCC не генерише сам објектну датотеку, већ зове асемблер.

```
int main()  
{  
    printf ("Hello World!\n");  
}
```

```
.file 1 "hello.c"  
.section .mdebug.abi32  
.previous  
.rdata  
.align 2  
.LC0:  
.ascii "Hello World!\000"  
.text  
.align 2  
.globl main  
.set nomips16  
.ent main  
.type main, @function  
main:  
.frame $sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0  
.mask 0x00000000,0  
.fmask 0x00000000,0  
la $4,.LC0  
j puts  
.end main  
.size main,.-main  
.ident "GCC: (GNU) 4.5.0"
```



Асемблер у билд процесу

- Преводи асемблерски код у објектну датотеку
- Позив асемблера из командне линије:
може директно, а може и кроз GCC

```
as file_name.s -o file_name.o  
gcc file_name.s -o file_name.o
```

- Објектна датотека садржи следеће информације у бинарном облику:
 - Машински код
 - Иницијализоване податке
 - Табелу симбола
 - Релокационе информације
 - Дебаг информације
 - везе између Це идентификатора и физичких ресурса
 - везе између Це линија и инструкција
- Користећи objdump алат (или нешто слично) могуће је излистати садржај објектне датотеке у текстуалном облику



Објектна датотека

- Objdump -t

SYMBOL TABLE:

```
| df *ABS*      00000000 link.c
| d .text      00000000 .text
| d .data      00000000 .data
| d .bss       00000000 .bss
| d .comment   00000000 .comment
g F .text      0000001d foo
*UND*         00000000 func
*UND*         00000000 var
```

- Objdump -d

00000000 <foo>:

```
0: 55          push  %ebp
1: 89 e5       mov   %esp,%ebp
3: 83 ec 18    sub   $0x18,%esp
6: 8b 45 08    mov   0x8(%ebp),%eax
9: 89 04 24    mov   %eax,(%esp)
c: e8 fc ff ff call  d <foo+0xd>
11: a3 00 00 00 mov   %eax,0x0
16: a1 00 00 00 mov   0x0,%eax
1b: c9         leave
1c: c3         ret
```

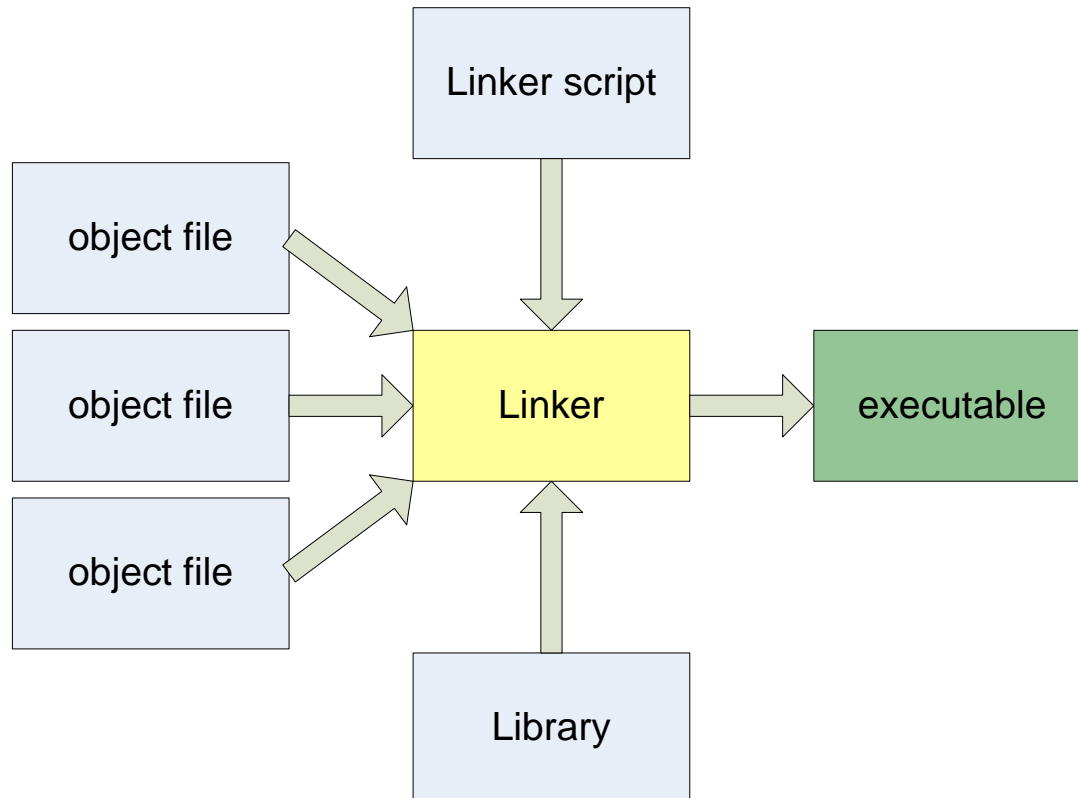
- Objdump -h

Sections:

| Idx | Name | Size | VMA | LMA | File off | Algn |
|-----|--|----------|----------|----------|----------|------|
| 0 | .text | 0000001d | 00000000 | 00000000 | 00000034 | 2**2 |
| | CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE | | | | | |
| 1 | .data | 00000000 | 00000000 | 00000000 | 00000054 | 2**2 |
| | CONTENTS, ALLOC, LOAD, DATA | | | | | |
| 2 | .bss | 00000000 | 00000000 | 00000000 | 00000054 | 2**2 |
| | ALLOC | | | | | |
| 3 | .comment | 00000024 | 00000000 | 00000000 | 00000054 | 2**0 |
| | CONTENTS, READONLY | | | | | |
| 4 | .note.GNU-stack | 00000000 | 00000000 | 00000000 | 00000078 | 2**0 |
| | CONTENTS, READONLY | | | | | |

Повезивач 1/2

- Последња фаза у билд процесу
- Повезује више објектних датотека у јединствену извршну датотеку





Повезивач 2/2

- Главни задатак повезивача је да разреши међусобне зависности датотека:
 - Позиви спољних функција
 - Приступи спољним променљивама

```
extern int var;  
extern int func(int);  
  
int foo(int a)  
{  
    var = func(a);  
    return var;  
}
```

```
foo:  
    addiu    $sp, $sp, -24  
    sw       $31, 20($sp)  
    jal      func  
    lw       $31, 20($sp)  
    sw       $2, var  
    j        $31  
    addiu    $sp, $sp, 24
```

- Симболи који су означени црвеном бојом морају бити разрешени током процеса превођења



Пуњач

- Пуњач је често део оперативног система. Чак и када постоји као засебан алат (да ли на платформи домаћину или на циљној платформи) обично је у тесној вези са оперативним системом.
- Пуњење подразумева преношење машинских инструкција са спољне меморије у адресни простор процесора.
- Осим за машинске инструкције у процесорској меморији се мора обезбедити место и за глобалне променљиве (променљиве статичке трајности). Почетне вредности за иницијализоване променљиве се већ налазе у извршној датотеци. Те вредности морају бити пренете у процесорску меморију.
- За неиницијализоване глобалне променљиве обично нема потребе преносити вредности, већ само заузети место за њих у посебном сегменту меморије. Тај сегмент меморије се назива **bss** сегмент и подразумевано је постављен на 0.



Пуњач - локалне променљиве и динамички заузета меморија



- Локалне променљиве и динамички заузета меморија обично се не тичу ни повезивача ни пуњача:
 - **Локалне променљиве** се обично налазе у регистрима или на програмском стеку. Дакле, све адресе су релативне у односу на почетак стека, а то је динамичка категорија.
 - **Динамички заузета меморија** се обично заузима у меморијском простору који се назива „хрпа” (енг. heap). Адреса динамички заузете меморије се не зна током превођења, повезивања и пуњења. За доделу динамички заузете меморије најчешће је одговоран оперативни систем.



Начини повезивања

- Постоје три начина повезивања:

1) Статичко повезивање (током повезивања)

- Код сваке спољне функције на крају завршава у извршној датотеци.
- Позиви функција се обављају директно преко адресе, која се разрешава током повезивања

2) Динамичко повезивање током пуњења

- Спољне функције се налазе на једном, физички одвојеном месту.
- Оперативни систем пресликава адресе спољних функција на адресни простор програма који се извршава.
- Позиви се обављају кроз Procedure Linkage Table (PLT) која се делом генерише током повезивања, али се довршава тек током пуњења (сваки спољни симбол има поље у PLT)

3) Динамичко повезивање током извршавања

- Нема правих спољних функција.
- Програм садржи експлицитан код који пресликава адресе неких функција на свој адресни простор.
- Дакле, показивачи на функције се експлицитно добављају током извршавања.



Статичка библиотека

- Пример:

avglib.h:

```
double avg(double a, double b);
```

avglib.c:

```
double avg(double a, double b)
{
    return (a + b) / 2.0;
}
```

- Билдовање библиотеке:

```
gcc -c avglib.c
ar rs libavg.a avglib.o
```

- Прва линија генерише објектну датотеку
- Друга линија позива архивер алат (**ar**) који се користи за прављење библиотека
- По конвенцији назив библиотеке треба да почиње са „lib” а да му екстензија буде „.a”
- Статичка библиотека је у суштини само скуп објектних датотека



Употреба статичке библиотеке

- Пример:

```
app.c:  
#include "avglib.h"  
  
int main()  
{  
    printf("average value %lf\n",  
        avg(3.7, 4.6));  
    return 0;  
}
```

- Билдовање програма:

```
gcc --static -I../include -L../lib -o app app.c -lavg
```

- `--static` наређује повезивачу (који се позива из GCC-а) да користи статичку верзију библиотеке (у супротном, биће коришћена динамичка верзија исте библиотеке - уколико постоји)
- `-lavg` наређује повезивачу да повеже програм са `libavg.a` библиотekom (видимо да је почетак имена библиотеке подразумевано „lib“)



Динамичка (дељена) библиотека



- Пример:

avglib.h:

```
double avg(double a, double b);
```

avglib.c:

```
double avg(double a, double b)
{
    return (a + b) / 2.0;
}
```

- Билдовање библиотеке:

```
gcc -c -fpic avglib.c
gcc -shared -o libavg.so avglib.o
```

- Прва линија генерише објектну датотку, али коришћењем наредбе **-fpic**. Та наредба говори компајлеру да генерише код који не зависи од позиције, тзв. **position independent code**. Ово је важно зато што тај код треба да се преслика на различите меморијске адресе.
- У другој линији наредба -shared говори компајлеру да генерише дељену библиотеку („дељени објекат“, „shared object“ у Линукс терминологији)
- По конвенцији назив библиотеке треба да почиње са „lib“ а да му екстензија буде „.so“



Употреба динамичке (дељене) библиотеке



- Пример:

```
app.c:  
#include "avglib.h"  
  
int main()  
{  
    printf("average value %lf\n",  
        avg(3.7, 4.6));  
    return 0;  
}
```

- Билдовање програма:

```
gcc -I../include -L../lib -o app app.c -lavg
```

- -lavg наређује повезивачу да повеже програм са libavg.so библиотekom (видимо да је почетак имена библиотеке подразумевано „lib”)
- Динамичка (дељена) библиотека је подразумевани облик библиотеке
- Библиотека мора бити на истој путањи као и извршна датотека. Ако није, путања до библиотеке мора бити додата на следећи начин:

```
export LD_LIBRARY_PATH=/library/path:$LD_LIBRARY_PATH
```



Динамичко повезивање током извршавања



- Користи се иста динамичка (дељена) библиотека
- Пример:

```
#include <dlfcn.h>
#include "avglib.h"

int main()
{
    void* handle;
    double (*avg)(double,double);
    char* error;

    handle = dlopen("libavg.so", RTLD_LAZY);
    if (handle == NULL)
    {
        fputs(dlerror(), stderr);
        exit(1);
    }

    avg = dlsym(handle, "avg");
    if ((error = dlerror()) != NULL)
    {
        fputs(error, stderr);
        exit(1);
    }

    printf("%f\n", (*avg)(3.7, 4.6));
    dlclose(handle);
    return 0;
}
```

- Мора постојати код који ће:
 - Отворити динамичку библиотеку
`void* dlopen(const char* filename, int flag);`
 - Додати адресе симбола (променљиве или функције) које ће бити коришћене
`void* dlsym(void* handle, const char* symbol);`
 - Затворити динамичку библиотеку
`int dlclose(void* handle);`
- Библиотека се нигде не помиње током билдовања програма

`gcc -I../include -L../lib -o app app.c`



Које повезивање користити?



- Предности статичког повезивања:
 - Позиви функција су бржи (нема индирекције)
 - Покретање програма је брже, јер не мора се трошити време на повезивање
 - Све је у једној извршној датотеци - дистрибуција програма је једноставнија
- Предности динамичког повезивања током пуњења:
 - Мања извршна датотека
 - Нема дуплирања кода уколико више различитих програма користе исту библиотеку (постоји само један примерак библиотеке, а различити процеси/програми је користе – „дељена”)
 - Нове верзије библиотека (унапређене на разне начине) могу бити коришћене у програму без његовог поновног превођења (докле год спрега није промењена)
- Предности динамичког повезивања током извршавања:
 - Време које је потребно за повезивање (и пуњење библиотеке) троши се само ако се током извршавања јави потреба за функционалношћу те библиотеке
 - Програм се може другачије понашати у зависности од тога да ли је нека библиотека присутна или не (нпр. неке функционалности постају доступне кориснику само ако је библиотека присутна, итд.)