ФАКУЛТЕТ ЗА ЕЛЕКТРОТЕХНИКА И ИНФОРМАЦИСКИ ТЕХНОЛОГИИ

Организација на програма во повеќе датотеки

– Податочни структури и програмирање –

Организација на програма во повеќе датотеки

- Користење готови библиотеки
  - #include <stdio.h>
- Проблем: Датотеки со голем изворен код
  - Многу линии код
  - Тешко менување
- Решение:
  - Структурирана организација на кодот
  - Повторно користење на напишан код/функции
  - Свои датотеки за заглавје (header)
- Напредно:
  - Свои библиотеки

Сопствен код/функции

- Повторно користење напишан сопствен код
  - Смести во засебна датотека
  - Вклучи датотека во нови програми
- Сопствена header датотека (датотека заглавје)
- Предности
  - Подобра организација на кодот
  - Поедноставно пронаоѓање и поправање грешки
  - Поедноставно менување на кодот
    - Промена на имплементација
    - На едно место
    - Транспарентно за корисниците

Сопствен код

- Датотека заглавје (header): .h
  - svoe_ime.h
  - Декларации – прототип, аргументи
    - Функции, Променливи, Типови
  - Кои функции
  - Како се користат
    - влезни/излезни аргументи
    - што враќаат
  - интерфејс меѓу програмерот и изворниот код
- Имплементација (код): .c
  - svoe_ime.c
  - Конкретна имплементација на прототип од .h датотека

Вклучување надворешен код

- Надворешен код – во засебна датотека
- Вклучување во C програма:
  - Надворешен код се „праќа“ на компајлер во време на компајлирање
    - gcc -o nov nov.c nadvoresen.c
  - C програмата што се пишува треба да има пристап до прототипите на функциите од надворешниот код
    - #include "svoe_ime.h"

Патеки за header датотеки

- Header датотеки вклучени со #include
- Аглести загради (пр. #include <stdio.h>)
  - Пребарува на неколку „предефинирани“ (default) системски локации
  - На пр. стандарден GNU именик: /usr/include
- Двојни наводници (пр. #include "moj.h")
  - тековен именик за тековната датотека
  - Додавање именици за вклучување библиотеки
    - Надворешна библиотека која не е сместена во вообичаените системски предефинирани локации за библиотеки
    - -I pateka_imenik опција на компајлер

Пример

- Да се напише програма која од командна линија прима еден аргумент (цел број) и проверува дали тој цел број е прост број или не.
- За одредување дали бројот е прост, треба да се користи готов код од пријател, кој веќе ја има напишано функцијата во датотеките (primes.h и primes.c)

primes.h

```
int isPrime(int n); //Prototip na f-ja.  Vraka  0 ako n ne e prost, 1 ako n e prost
```

Пример (II)

primes.c

```
int isPrime(int n) {

   // vraka 0 ako ne e prost,  1 ako e prost

   if (n<2) return 0; // prviot prost broj e 2
   if (n==2) return 1; // 2 e sigurno prost
   if ((n % 2)==0) return 0; // site parni broevi >2 ne se prosti

   int i;
   for (i=3; i*i < n; i++) { // da se proveri delivot do sqrt(n)
       if ((n % i) == 0) {
          return 0;
        }
   }
   return 1;
}
```

Пример (III)

go.c

```
/*  go.c     Prima eden argument od komandna linija (cel broj) i vraka 1
 ako brojot e prost, i 0 ako ne.  Se kompajlira so: gcc -o go go.c primes.c */
#include <stdio.h>
#include <stdlib.h>
#include "primes.h"
int main(int argc, char *argv[]) {
        if (argc < 2) {
            printf("Greshka: treba da ima 1 argument (cel broj) za da se proveruva\n");
            return 1;
        }
        else {
            int n = atoi(argv[1]);
            int prime = isPrime(n);
            printf("isPrime(%d) = %d\n", n, prime);
            return 0;
        }
}
```

Вклучување „сопствена“ библиотека

Компајлирање и користење

Се компајлира со: gcc -o go go.c primes.c

```
$ gcc -o go go.c primes.c
$ ./go 1
isPrime(1) = 0
$ ./go 2
isPrime(2) = 1
$ ./go 3
isPrime(3) = 1
$ ./go 63
isPrime(63) = 0
$ ./go 67
isPrime(67) = 1
$ ./go 12347
isPrime(12347) = 1
```

./ - повикување на во тековниот директориум

Пример 2

- Да се направи сопствена библиотека со две функции `rand()` и `bubble_sort()`
- Заглавје датотека (**.h**) **util.h**.

util.h

```
/* util.h */
extern int rand();
extern void bubble_sort(int, int []);
```

- Прототипи за функции
- "extern" во C означува функции што ќе бидат поврзани подоцна

Пример 2 (II)

util.c

```
 /* util.c */
#include "util.h"
int rand_seed=10;
/* from K&R - produces a random number between 0 and 32767.*/
int rand(){
        rand_seed = rand_seed * 1103515245 +12345;
        return (unsigned int)(rand_seed / 65536) % 32768;
}
void bubble_sort(int m,int a[]){
        int x,y,t;
        for (x=0; x < m-1; x++)
                for (y=0; y < m-x-1; y++)
                        if (a[y] > a[y+1]){
                                t=a[y];
                                a[y]=a[y+1];
                                a[y+1]=t;
                        }
}
```

Пример 2 (III)

- Се вклучува #include "util.h"
- Променлива `rand_seed`
  - не е во header датотека
  - Не може да се види ниту промени од програма што ќе ја користи библиотеката
  - Криење информации
  - Уште повеќе ако е дефинирана како `static`. Целосно криење

Пример 2 (IV)

main.c

```
#include <stdio.h>
#include "util.h"
#define MAX 10
int a[MAX];
void main(){
        int i,t,x,y;
        /* fill array */
        for (i=0; i < MAX; i++){
                a[i]=rand();
                printf("%d\n",a[i]);
        }
        bubble_sort(MAX,a);
        /* print sorted array */
        printf("--------------------\n");
        for (i=0; i < MAX; i++)
                printf("%d\n",a[i]);
}
```

Компајлирање и стартување во повеќе чекори

Компајлирање библиотека

- `gcc -c util.c`
  - **`-c`** креира објект датотека .o за библиотеката (`util.o`)
  - Објект датотеката содржи машински код за библиотеката
  - не се извршува самостојно и треба да се поврзе со програма што содржи `main` функција

Компајлирање главна програма

- `gcc -c main.c`
  - креира објект датотека .o за програмата (`main.o`)
  - Објект датотеката содржи машински код за главната програма

Креирање извршна датотека

- `gcc -o main main.o util.o`
  - машински код за целата програма и поврзување на двата објекти
  - Извршна датотека `main`

Стартување извршна датотека

- `main`

Алатката `make`

- Алатка за компајлирање
  - Компајлира C програми
  - составени од повеќе датотеки
  - Повеќе чекори на компајлирање (понекогаш)
- Датотека `Makefile`
  - Текстуална датотека
  - Во ист именик со програмата
  - Треба да се вика `Makefile`
  - Рецепт за правење на програмата (поврзување и компајлирање)
- Едноставен `Makefile` за Пример1

`Makefile`

```
go: go.c primes.c
	gcc -o go go.c primes.c
```

Makefile

```
go: go.c primes.c
	gcc -o go go.c primes.c
```

- Прв збор во прв ред е името на „цел“ (target) или „рецепт“ што треба да се „произведе“
  - `go:`
- По `:` следат сите „нешта“ од кои зависи целта `go`
  - `go.c primes.c`
- Следниот ред започнува со TAB (не празни места)
  - следи команда за компајлирање `gcc -o go go.c primes.c`
  - Може да има повеќе
  - Потребни чекори за да се „направи“ („make“) рецептот „go“
- Активирање: `make` на командна линија
  - make без аргументи го активира првото правило („цел“) во `Makefile`
  - Програмата `make` знае дека правилото "go" треба да се изврши доколку некоја од датотеките од кои тоа зависи се промени

```
.../code/primes$ cp Makefile1.txt Makefile
.../code/primes$ make
gcc -o go go.c primes.c
```

`make` во повеќе чекори

Makefile

```
main: main.o util.o
      gcc -o main main.o util.o
main.o: main.c util.h
      gcc -c main.c
util.o: util.c util.h
      gcc -c util.c
```

- Организација на голема програма: повеќе библиотеки и главна програма
- Два типа линии:
  - **Линии за зависност (dependency lines)**
  - **Извршни линии (executable lines)**
- Конечна извршна датотека

- **Линии за зависност (dependency lines)**
  - Најлево напишани
  - Зависност на некоја датотека од други датотеки (**`main.o: main.c util.h`**)
  - **Експлицитно `main.o` зависи и од `util.h`** (во претходен пример->не)
  - Ако некоја од тие датотеки се промени, треба да се извршат **извршните линии кои следат** за да повторно се креира (**`main.o`**)
- **Извршни линии (executable lines)**
  - TAB пред код
  - Која било валидна UNIX/Windows команда
  - Повеќе наредби, повеќе линии
- **Конечна извршна датотека**
  - Целта за конечен резултат од `Makefile` треба да биде сместена на првата линија
  - `main` од прв ред на `Makefile`, зависи од **`main.o`** и **`util.o`**.

Пример 2 (make)

Makefile

```
all: main.o module.o
	gcc main.o module.o -o target_bin
main.o: main.c module.h
	gcc -I . -c main.c
module.o: module.c module.h
	gcc -I . -c module.c
clean:
	rm -rf *.o // del *.o //windows
	rm target_bin // del *.exe //windows
```

- 4 цели во овој `Makefile`:
- **`all:`** специјална цел. Зависи од `main.o` и `module.o`. Команда: Поврзи два `.o` објекти во конечна извршна датотека (**`target_bin`**)
- **`main.o:`** цел за објектна датотека `main.o` која зависи од `main.c` и `module.h`. Команда: Компајлирај `main.c` во `main.o`.
- **`module.o:`** цел за објектна датотека `module.o` која зависи од `module.c` и `module.h`. Команда: Компајлирај `module.c` во `module.o`.
- **`clean:`** специјална цел. Нема зависности. Чистење на излезите од компајлирањето од имениците на проектот
- `"gcc -I ."` - adds include directory of header files – in this case in the current directory "."

Пример 2 (make) (II)

- Повеќе цели во `Makefile` => аргументи на `make`
- Синтакса на командата `make`: **`make <target>`**
- Работи и без аргумент => предефинирано се повикува „првата“ цел (од првиот ред на `Makefile`)
- Во примерот: тоа е целта `all:`
  - со што се креира бараната извршна датотека target_bin!
- За стартување на целта clean потребно е да се испише:
  - make clean

Makefile

```
all: main.o module.o
	gcc main.o module.o -o target_bin
main.o: main.c module.h
	gcc -I . -c main.c
module.o: module.c module.h
	gcc -I . -c module.c
clean:
	rm -rf *.o
	rm target_bin
```

Пример 2 (make) извршување

Makefile

```
all: main.o module.o
	gcc main.o module.o -o target_bin
main.o: main.c module.h
	gcc -I . -c main.c
module.o: module.c module.h
	gcc -I . -c module.c
clean:
	rm -rf *.o
	rm target_bin
```

```
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

- Се извршува по редослед на зависности
- Дрво на зависности

Експлицитно повикување цел

Makefile

```
all: main.o module.o
	gcc main.o module.o -o target_bin
main.o: main.c module.h
	gcc -I . -c main.c
module.o: module.c module.h
	gcc -I . -c module.c
clean:
	rm -rf *.o
	rm target_bin
```

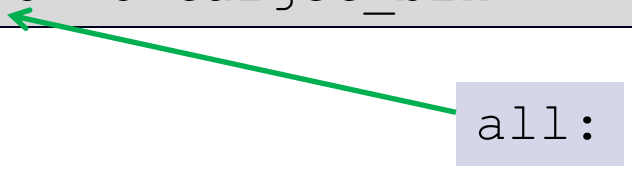
```
$ make clean
rm -rf *.o
rm target_bin
```

- Се извршува целта `clean:`
- Се бришат
  - сите `.o` датотеки како резултат на компајлирање
  - Извршна датотека `target_bin`

Автоматски зависности – `make`

- `make` автоматски прекомпајлира се што е потребно доколку некаде се случи промена.

```
$ make
gcc main.o module.o -o target_bin
```



- Нема промена на ниту една изворна датотека
- `make` ја компајлира само целта `all:`
- Целите од кои зависи `all:` не се променети
  - `make` не ги извршува
- `all:` не е име на датотека
  - Не може да се провери дали е променето
  - Затоа ја активира

Автоматски зависности – `make` (II)

Makefile

```
go: go.c primes.c
        gcc -o go go.c primes.c
```

1)

```
.../code/primes$ make
gcc -o go go.c primes.c
```

2)

```
.../code/primes$ make
make: `go' is up to date.
```

- Нема промена на ниту една изворна датотека
- `make` не компајлира ништо
- `go:` е име на извршна датотека
  - Се проверува дека не е променета
  - Затоа не се активира ништо

Автоматски зависности - make (III)

Makefile

```
all: main.o module.o
	gcc main.o module.o -o target_bin
main.o: main.c module.h
	gcc -I . -c main.c
module.o: module.c module.h
	gcc -I . -c module.c
clean:
	rm -rf *.o
	rm target_bin
```

- Ако се промени `module.c`
  - Со додавање едноставна наредба: printf("\nprva promena"); во некоја од функциите

```
$ make
gcc -I . -c module.c
gcc main.o module.o -o target bin
```