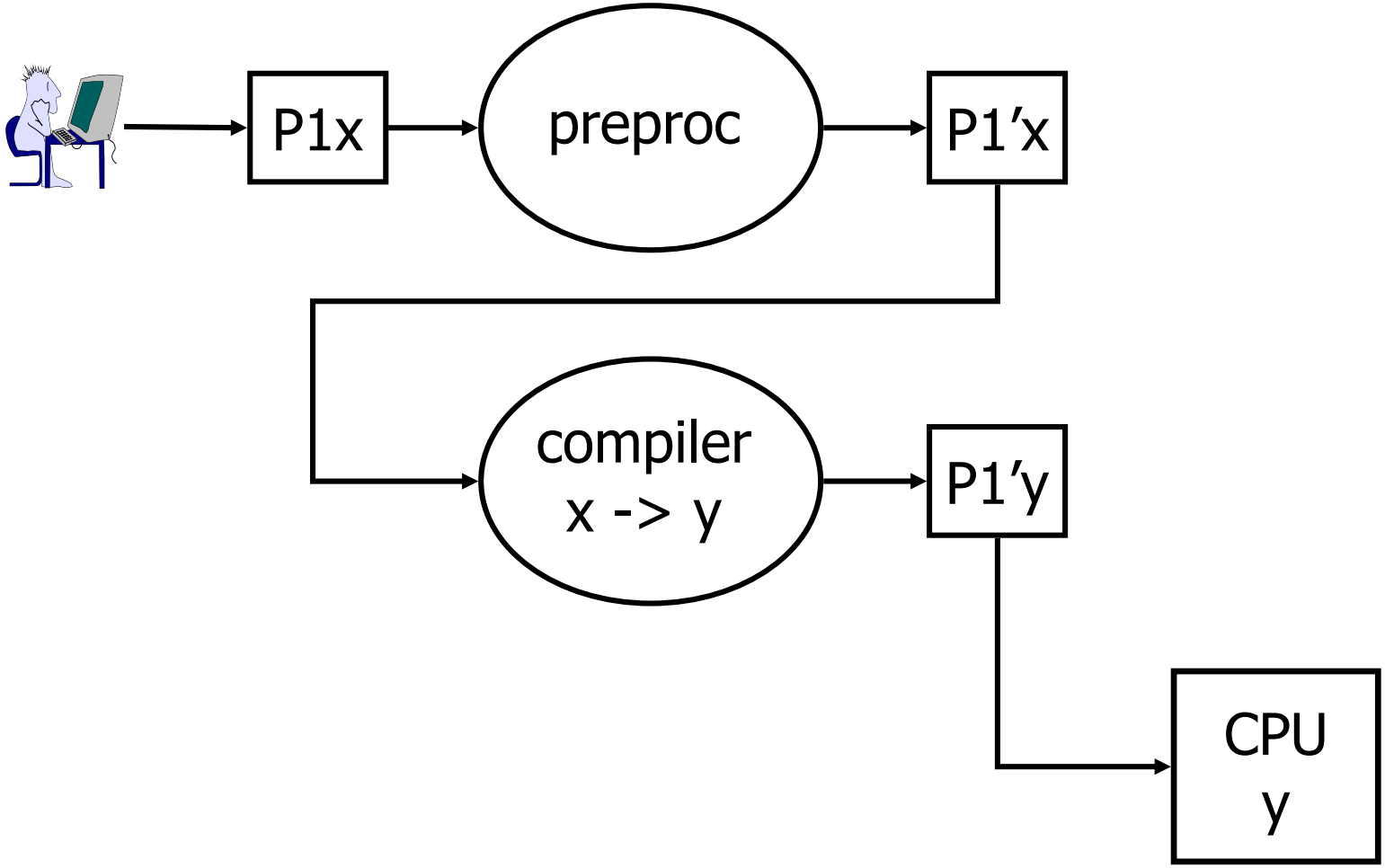
Προεπεξεργαστής C

Βασική ιδέα

- Ο προεπεξεργαστής (pre-proccesor) της C είναι ένα πρόγραμμα που εκτελείται και **μετασχηματίζει** τον πηγαίο κώδικα πριν αυτός δοθεί στον μεταγλωτιστή για μετάφραση (`gcc -E`).
- Πραγματοποιεί αλλαγές σε επίπεδο **κειμένου**, με την μορφή «απλών» αντικαταστάσεων χαρακτήρων, **χωρίς** γνώση ή ερμηνεία του συντακτικού της C.
- Οι πιο χαρακτηριστικές χρήσεις:
  - μακροεντολές
  - εισαγωγή κειμένου υπό συνθήκη
  - αντιγραφή κειμένου από άλλα αρχεία


P1x
preproc
P1'x
compiler
x -> y
P1'y
CPU
y

Μακροεντολές

- `#define <macro> <expr>`
- Ορίζει μια μακροεντολή `<macro>` προς χρήση στο κείμενο του προγράμματος, που θα αντικατασταθεί από τον προεπεξεργαστή με την έκφραση `<expr>`.
- Η επεξεργασία των macros γίνεται με **απλή αντικατάσταση** σε επίπεδο κειμένου.
- Σημείωση: χρειάζεται προσοχή (παρενθέσεις) έτσι ώστε να αποφεύγονται λάθη με τελεστές που ακολουθούν στο κείμενο του προγράμματος.
- Σημείωση2: χρειάζεται προσοχή με «παραμέτρους» των macros καθώς αυτές αποτιμώνται **εκ νέου**, **όσες φορές** εμφανίζονται στην έκφραση.

```
/* ορισμός συμβολικών σταθερών */

#define N 100
…
int t[N];
…
for (i=0; i < N; i++ ) {
  …
}
…
if (i == N) {
  …
}
…
```

```
/* ορισμός συμβολικών σταθερών */

#define N 10*10
…
int t[N];
…
for (i=0; i < N; i++ ) {
  …
}
…
if (i == N) {
  …
}
…
```

```
/* ορισμός συμβολικών σταθερών */

#define A 10

#define B 20

#define C (A+B)

#define D (C*B)
```

```
/* ορισμός μακροεντολών */

#define LOWERCASE(a) (a- 'A' + 'a')

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

  do {
    c=getchar();
    if ((c >= 'A') && (c <= 'Z')) { c=LOWERCASE(c); }
    putchar(c);
  } while (c != '/n');

}
```

```
/* ορισμός μακροεντολών */

#define LOWERCASE(a) (((a>='A') && (a<='Z')) ? a-'A'+'a' : a)

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

  do {
    c=getchar();
    putchar(LOWERCASE(c));
  } while (c != '/n');

}
```

```
/* μη χρήση παρενθέσεων … */

#define PLUSONE(x) x+1

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

  scanf("%d",&i);
  j=PLUSONE(i)*2;
  printf("%d\n",j);

}
```

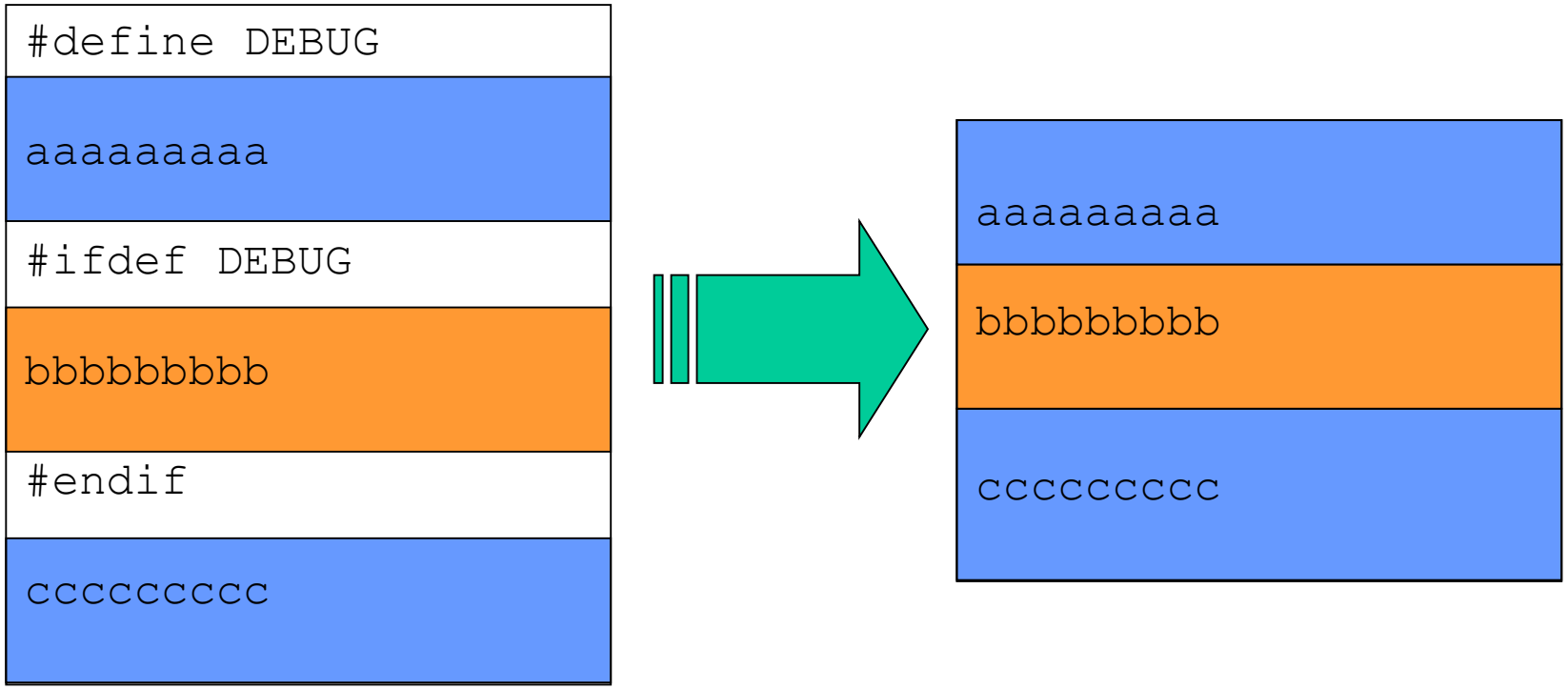
```
/* και πιο σωστά … */

#define PLUSONE(x) (x+1)

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

  scanf("%d",&i);
  j=PLUSONE(i)*2;
  printf("%d\n",j);
}
```

```
/* επανειλημμένη αποτίμηση παραμέτρων ... */

#define SQUARE(a) ((a)*(a))

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

  scanf("%d",&i);
  j=SQUARE(++i);
  printf("%d\n",j);
}
```

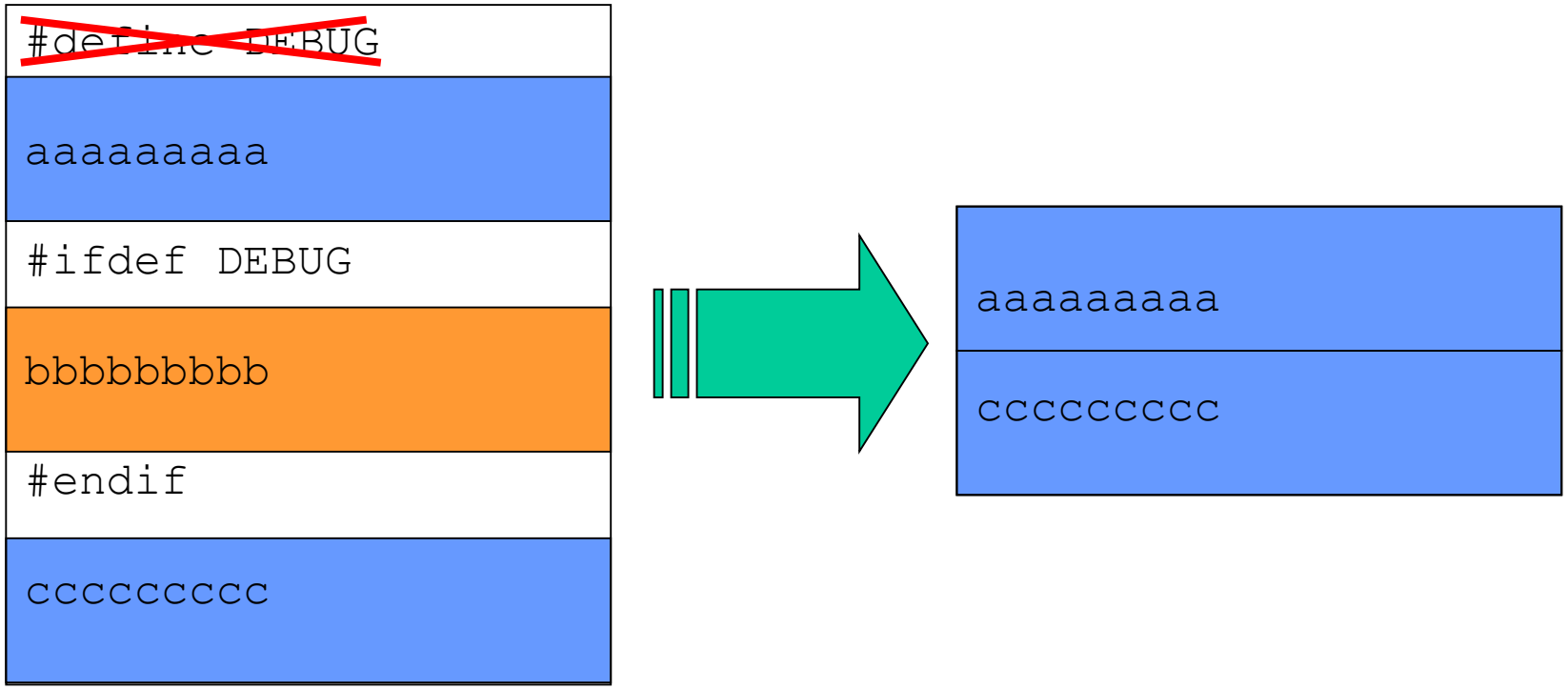
Εισαγωγή υπό συνθήκη

- `#if, #else, #elif, #endif`
- `#ifdef, #ifndef`
- Συνθήκες πάνω σε συμβολικά ονόματα.
- Αν μια συνθήκη ισχύει τότε το αντίστοιχο κείμενο συμπεριλαμβάνεται για μετάφραση, διαφορετικά όχι.
- Το κείμενο που συμπεριλαμβάνεται / αφαιρείται μπορεί να είναι οτιδήποτε (δεν χρειάζεται να αντιστοιχεί σε κάποια συντακτική ενότητα).
- Χρήσιμο για επιλογή ανάμεσα σε διαφορετικές εκδόσεις του κώδικα (κώδικας αποσφαλμάτωσης, υποστήριξη για διαφορετικές πλατφόρμες κλπ).
- Χρησιμοποιείται και για την αποφυγή διπλής εισαγωγής κειμένου (βλέπε `#include`).

```
#define DEBUG

…

#ifdef DEBUG   /* ή #if defined(DEBUG) */

… /* my debugging code */

#endif

…
```


#define DEBUG
aaaaaaaaa
#ifdef DEBUG
bbbbbbbbb
#endif
ccccccccc
aaaaaaaaa
bbbbbbbbb
ccccccccc

```
#define DEBUG

…

#ifdef DEBUG   /* ή #if defined(DEBUG) */

… /* my debugging code */

#endif

…
```


#define DEBUG
aaaaaaaaa
#ifdef DEBUG
bbbbbbbbb
#endif
ccccccccc
aaaaaaaaa
ccccccccc

```
#ifdef OLDVERSION

… /* old  code */

#else

… /* new code */

#endif
```

```
#define LINUX_VERSION 5

#if LINUX_VERSION == 6

… /* code for this platform */

#else

… /* code for default platform */

#endif
```

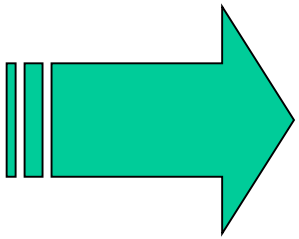
```
#ifndef INCL_X    /* ή #if !defined(INCL_X) */

#define INCL_X   /* avoid recursive inclusion */

… /* my code */

#endif
```

Εισαγωγή αρχείων

- `#include "filename"`
- Τα περιεχόμενα του αρχείου `filename` εισάγονται **ακριβώς** στο σημείο που εμφανίζεται η εντολή.
- Αν το αρχείο προς εισαγωγή δεν βρίσκεται στον κατάλογο εργασίας, τότε ο αντίστοιχος κατάλογος πρέπει να προσδιορίζεται με το όνομα του αρχείου.
- Αν το όνομα του αρχείου δίνεται σε `< >` αντί `" "`, τότε ο προεπεξεργαστής θεωρεί ότι το αρχείο βρίσκεται στον κατάλογο όπου έχουν τοποθετηθεί τα header files της C, π.χ. `/usr/include`.
- Αν το αρχείο που εισάγεται περιέχει και αυτό με την σειρά του εντολές `#include` τότε γίνεται εισαγωγή και αυτών των αρχείων μέσα στο αρχείο.

```
/* αρχείο stdio.h */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
```

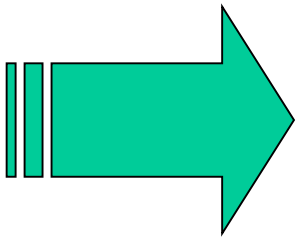
```
/* αρχείο myprog */

#include <stdio.h>

bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb
```



```
/* αρχείο myprog */

/* αρχείο stadio.h */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa

bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb
```

```
/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
```

```
/* αρχείο file2 */

#include "file1"

bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb
```

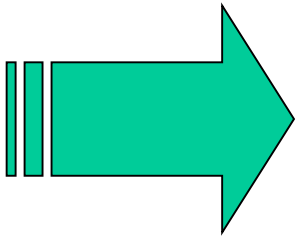


```
/* αρχείο file2 */

/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa

bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb
```

```
/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
```

```
/* αρχείο file2 */

#include "file1"

bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb
```

```
/* αρχείο file3 */

#include "file2"

cccccccccc
cccccccccc
cccccccccc
```



```
/* αρχείο file3 */

/* αρχείο file2 */

/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa


bbbbbbbbbb
bbbbbbbbbb
bbbbbbbbbb

cccccccccc
cccccccccc
cccccccccc
```

```c
/* myprog_min.c */

int min(int a, int b) {
  if (a < b) { return(a); } else { return(b); }
}
```

```c
/* myprog_max.c */

int max(int a, int b) {
  if (a < b) { return(b); } else { return(a); }
}
```

```c
/* myprog_main.c */

#include "myprog_min.c"
#include "myprog_max.c"

int main(int argc, char *argv[])) {
  int a,b;
  scanf("%d %d",&a,&b);
  printf("%d %d\n",min(a,b),max(a.b));
}
```

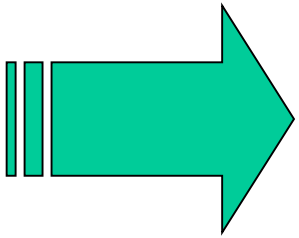
Αποφυγή επανειλημμένης εισαγωγής

- Ένα αρχείο μπορεί να εισάγει με την σειρά του (έμμεσα) επιπλέον αρχεία, τα οποία μπορεί να εισάγονται **ξανά** από το κυρίως πρόγραμμα.
- Η διπλή εισαγωγή μπορεί να είναι ανεπιθύμητη.
- Π.χ. διπλή εισαγωγή μεταβλητών και συναρτήσεων που οδηγεί (στη μετάφραση) σε συντακτικά λάθη.
- **Λύση 1:** ο προγραμματιστής πρέπει να γνωρίζει ποιά αρχεία εισάγουν (έμμεσα) τα αρχεία που εισάγει στο πρόγραμμα του (άμεσα) – ιδιαίτερα άκομψο.
- **Λύση 2:** το ίδιο το αρχείο φροντίζει να μην μπορεί να εισαχθεί δύο φορές στο ίδιο κείμενο – κλασική περίπτωση χρήσης του `#ifndef`.

```
/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
```

```
/* αρχείο file2 */

#include "file1"

bbbbbbbbbb
bbbbbbbbbb
```

```
/* αρχείο file3 */

#include "file2"
#include "file1"

cccccccccc
cccccccccc
```

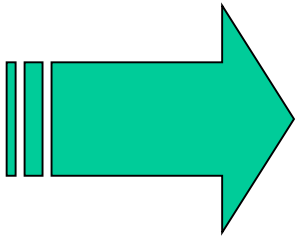


```
/* αρχείο file3 */

/* αρχείο file2 */

/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa

bbbbbbbbbb
bbbbbbbbbb

/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa

cccccccccc
cccccccccc
```

```
#ifndef INC_FILE1
#define INC_FILE1
/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa
#endif
```

```
/* αρχείο file2 */

#include "file1"

bbbbbbbbbb
bbbbbbbbbb
```

```
/* αρχείο file3 */

#include "file2"
#include "file1"

cccccccccc
cccccccccc
```



```
/* αρχείο file3 */
/* αρχείο file2 */
/* αρχείο file1 */

aaaaaaaaaa
aaaaaaaaaa

bbbbbbbbbb
bbbbbbbbbb

cccccccccc
cccccccccc
```

```c
/* myprog_min.c */
#ifndef INCL_MYPROGMIN
#define INCL_MYPROGMIN
int min(int a, int b) { … }
#endif
```

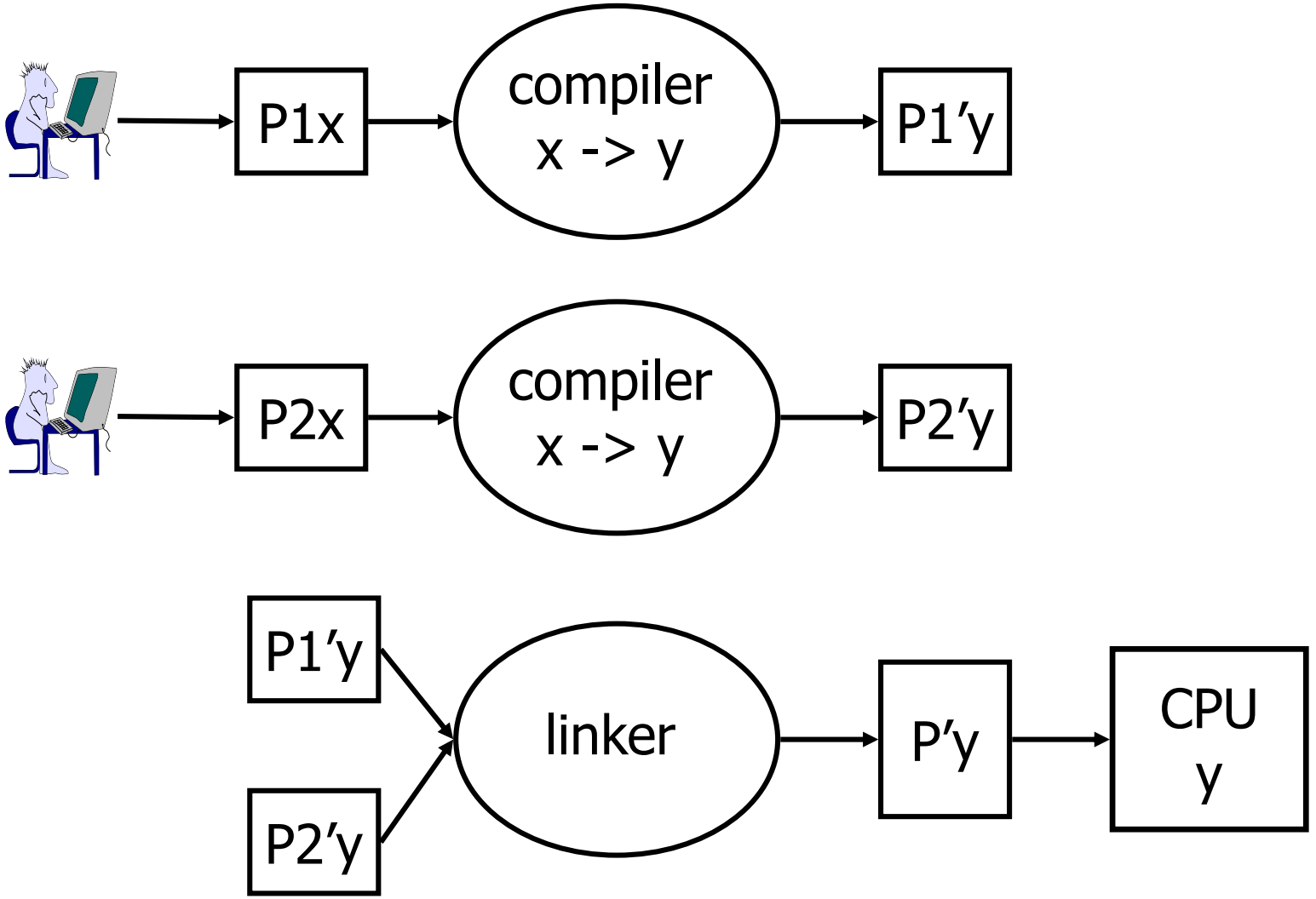
```c
/* myprog_max.c */
#include "myprog_min.c"
int max(int a, int b) {
  if (a == min(a,b)) { return(b); } else { return(a); }
}
```

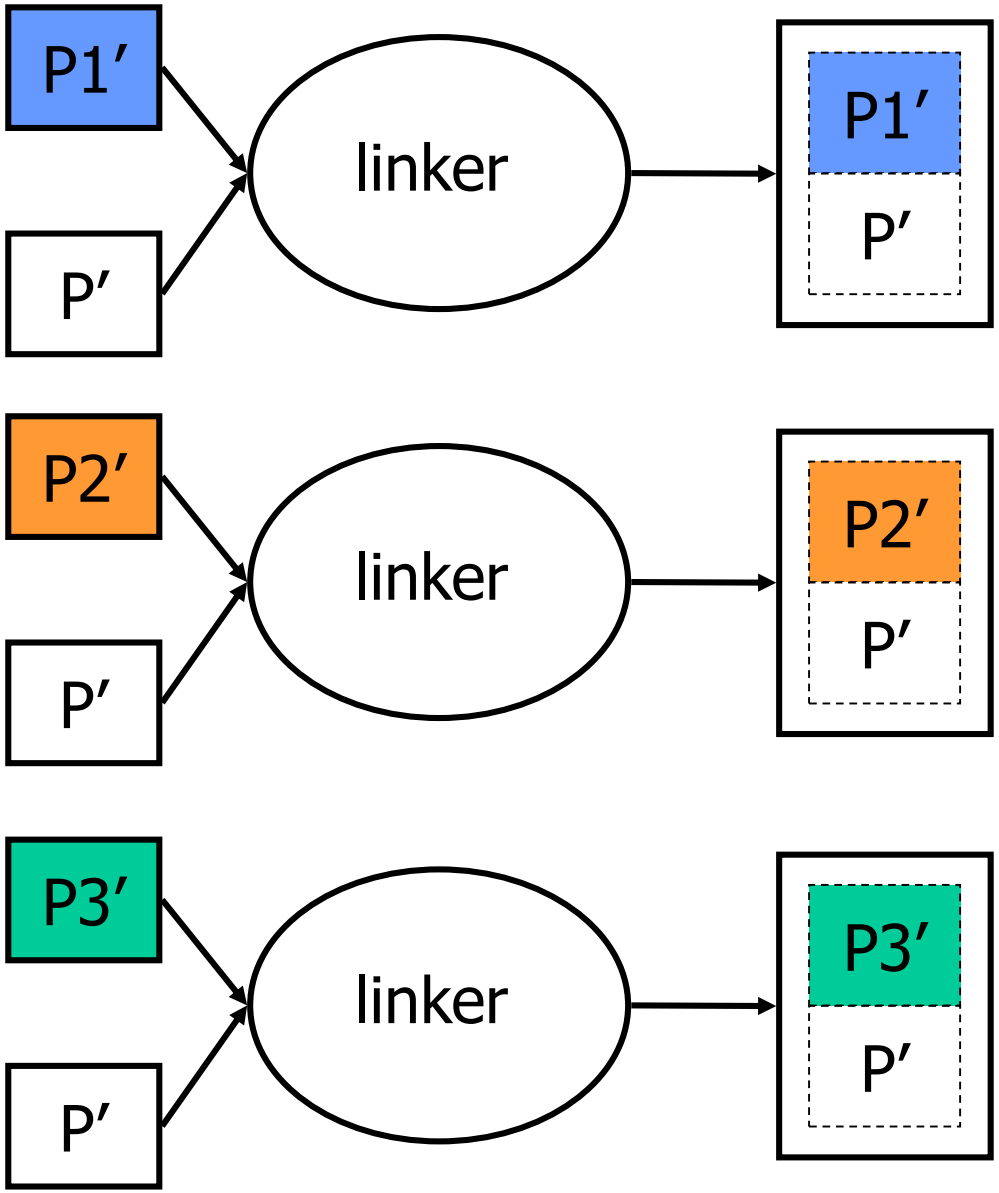
```c
/* myprog_main.c */

#include "myprog_min.c"
#include "myprog_max.c"

int main(int argc, char *argv[])) {
  int a,b;
  scanf("%d %d",&a,&b);
  printf("%d %d\n",min(a,b),max(a.b));
}
```

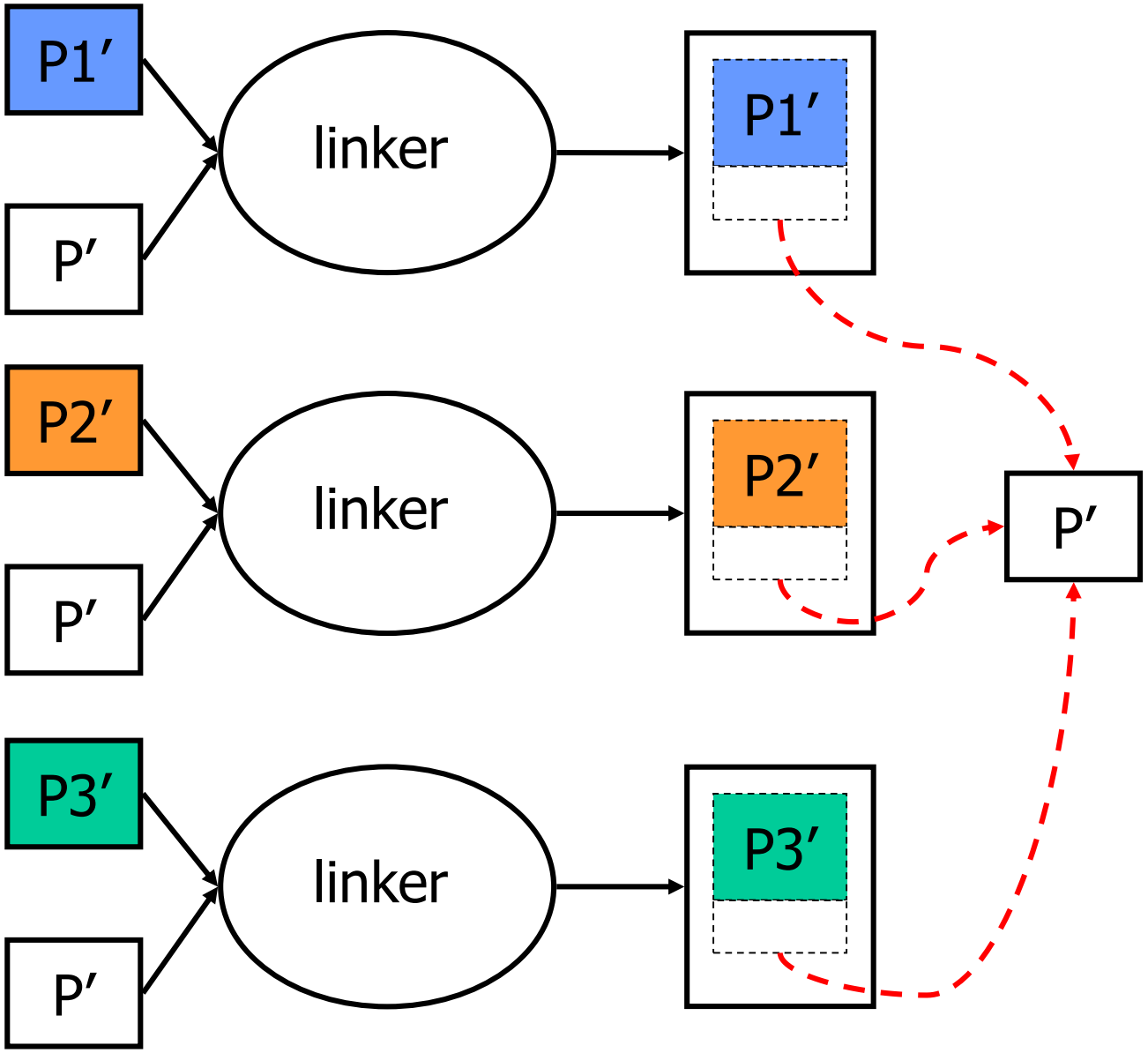
Ξεχωριστή μετάφραση

Επαναχρησιμοποίηση κώδικα

- Το μεγαλύτερο ίσως «στοίχημα» στην βιομηχανία λογισμικού είναι η **επαναχρησιμοποίηση** κώδικα.
- Ιδανικά, δεν χρειάζεται να ξαναγράψουμε κώδικα που έχει ήδη γραφτεί (από κάποιους άλλους).
- **Πως** χρησιμοπούμε κώδικα που υπάρχει;
- Προσέγγιση Α: **αντιγράφουμε** τον πηγαίο κώδικα, κάνοντας ίσως προσαρμογές για τις ανάγκες μας.
- Προσέγγιση Β: **καλούμε** τον εκτελέσιμο κώδικα, περνώντας τις παραμέτρους που χρειάζονται σύμφωνα με τις οδηγίες χρήσης / περιγραφή λειτουργικότητας του.
- Το Β απαιτεί μηχανισμό **σύνδεσης** κώδικα που έχει ήδη μεταφραστεί με τον κώδικα που γράφουμε.


P1x
compiler
x -> y
P1'y
P2x
compiler
x -> y
P2'y
P1'y
P2'y
linker
P'y
CPU
y


P1′
P′
linker
P1′
P′
P2′
P′
linker
P2′
P′
P3′
P′
linker
P3′
P′


P1′
P′
linker
P1′
P2′
P′
linker
P2′
P′
P3′
P′
linker
P3′

Ανάπτυξη αυτόνομων τμημάτων λογισμικού

- **Σχεδίαση προγραμματιστικής διεπαφής:** αποφασίζουμε το πως θα χρησιμοποιηθεί (προγραμματιστικά) η λειτουργικότητα.
- **Κατασκευή header file:** ορισμός της διεπαφής με την μορφή ενός header file που θα χρησιμοποιηθεί (κυρίως) από τα προγράμματα «πελάτες».
- **Κατασκευή υλοποίησης:** σε ξεχωριστό αρχείο υλοποιούμε τη λειτουργικότητα, όπως ορίζεται στο header file, που μεταφράζεται για να δημιουργηθεί το τμήμα κώδικα που θα συνδεθεί (αργότερα) με τον κώδικα από τα προγράμματα «πελάτες».

Header files στην C

- Η διεπαφή προγραμματισμού στην C ορίζεται μέσω ενός header file, που περιέχει (α) ορισμούς τύπων δεδομένων, (β) δηλώσεις σταθερών και καθολικών μεταβλητών, και (γ) δηλώσεις συναρτήσεων.
- Οι δηλώσεις μεταβλητών και συναρτήσεων πρέπει να έχουν τον προσδιορισμό `extern`, υποδεικνύοντας στον μεταγλωτιστή ότι οι αντίστοιχες υλοποιήσεις τους **δεν** βρίσκονται απαραίτητα στο ίδιο αρχείο.
- Κάνοντας `#include` ένα header file, το πρόγραμμα μπορεί να **χρησιμοποιεί** τύπους, μεταβλητές και συναρτήσεις που υλοποιούνται σε **ξεχωριστό** τμήμα λογισμικού (τον πηγαίο κώδικα του οποίου μπορεί να μην γνωρίζουμε) και να **μεταφραστεί** επιτυχώς.

Σύνδεση

- Όταν ένα πρόγραμμα μεταφράζεται με βάση τις δηλώσεις που περιέχει ένα header file, ο κώδικας που παράγεται από τον μεταφραστή περιέχει αναφορές σε **εξωτερικές** μεταβλητές και συναρτήσεις.
- Για να παραχθεί ο τελικός εκτελέσιμος κώδικας, απαιτείται μια επιπλέον διαδικασία **ενσωμάτωσης** των ορισμών και υλοποιήσεων τους, που βρίσκονται σε κάποιο άλλο αρχείο που έχει ήδη μεταφραστεί.
- Αυτή η διαδικασία ονομάζεται **σύνδεση** (linking).
- Αφού ο μεταφρασμένος κώδικας συνδεθεί με όλα τα υπόλοιπα μεταφρασμένα τμήματα που παρέχουν τους ορισμούς / υλοποιούν των εξωτερικών μεταβλητών / συναρτήσεων, δημιουργείται το τελικό εκτελέσιμο.

```
/* foo.h */

extern int i;

extern void boo();
```

υλοποιείται από

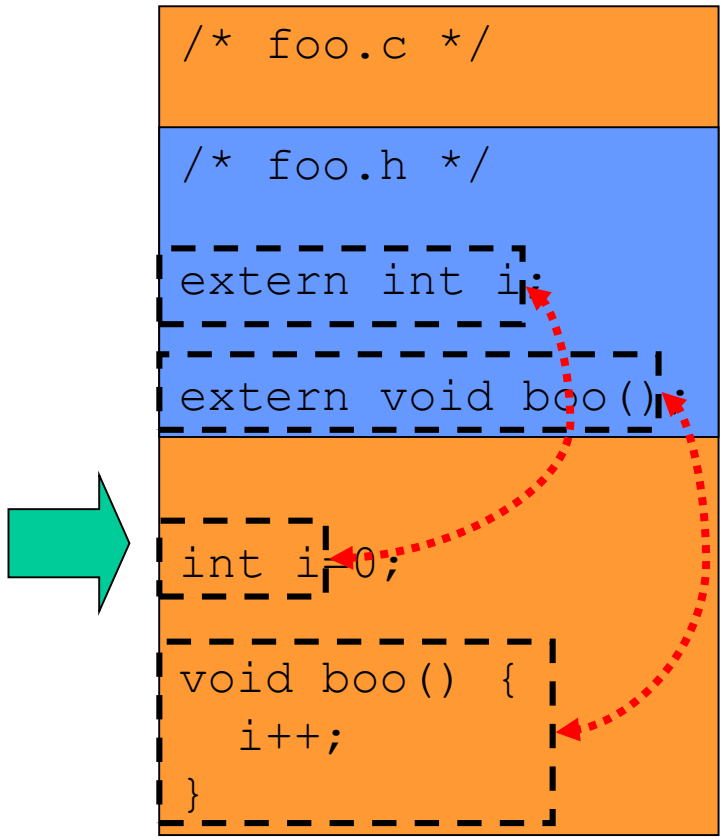
χρησιμοποιείται από

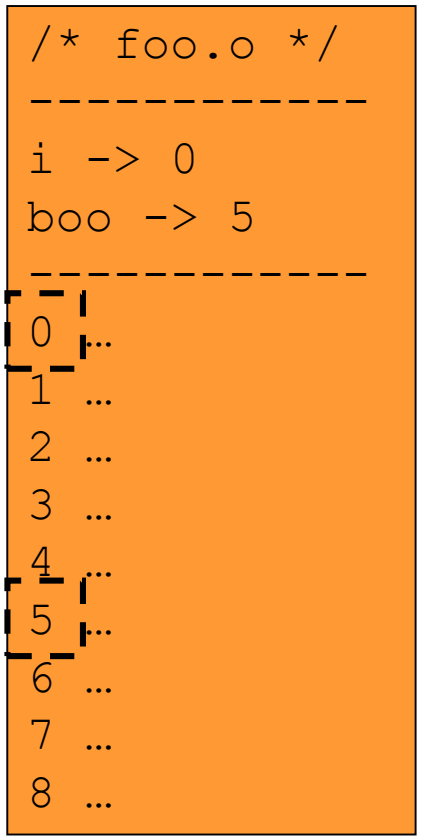
```
/* foo.c */

#include "foo.h"

int i=0;

void boo() {
  i++;
}
```

```
/* main.c */

#include "foo.h"

int main(int argc, int *argv[]) {

  boo();
  i++;
  boo();
}
```

```
/* foo.h */

extern int i;

extern void boo();
```

```
/* foo.c */

#include "foo.h"

int i=0;

void boo() {
  i++;
}
```

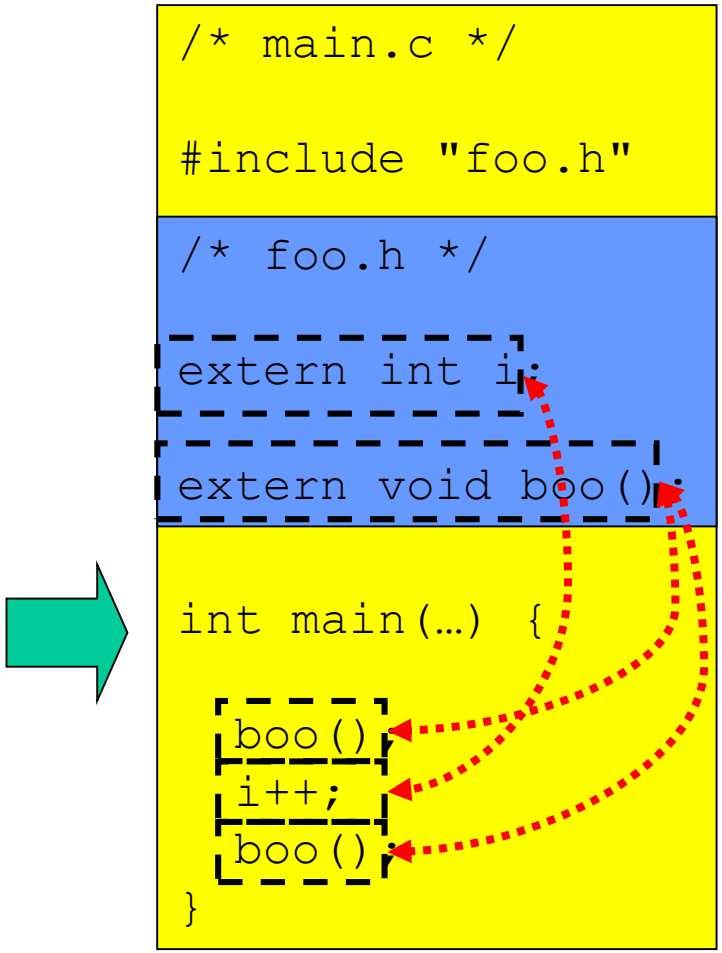





```
> gcc foo.c -c -o foo.o
>
```

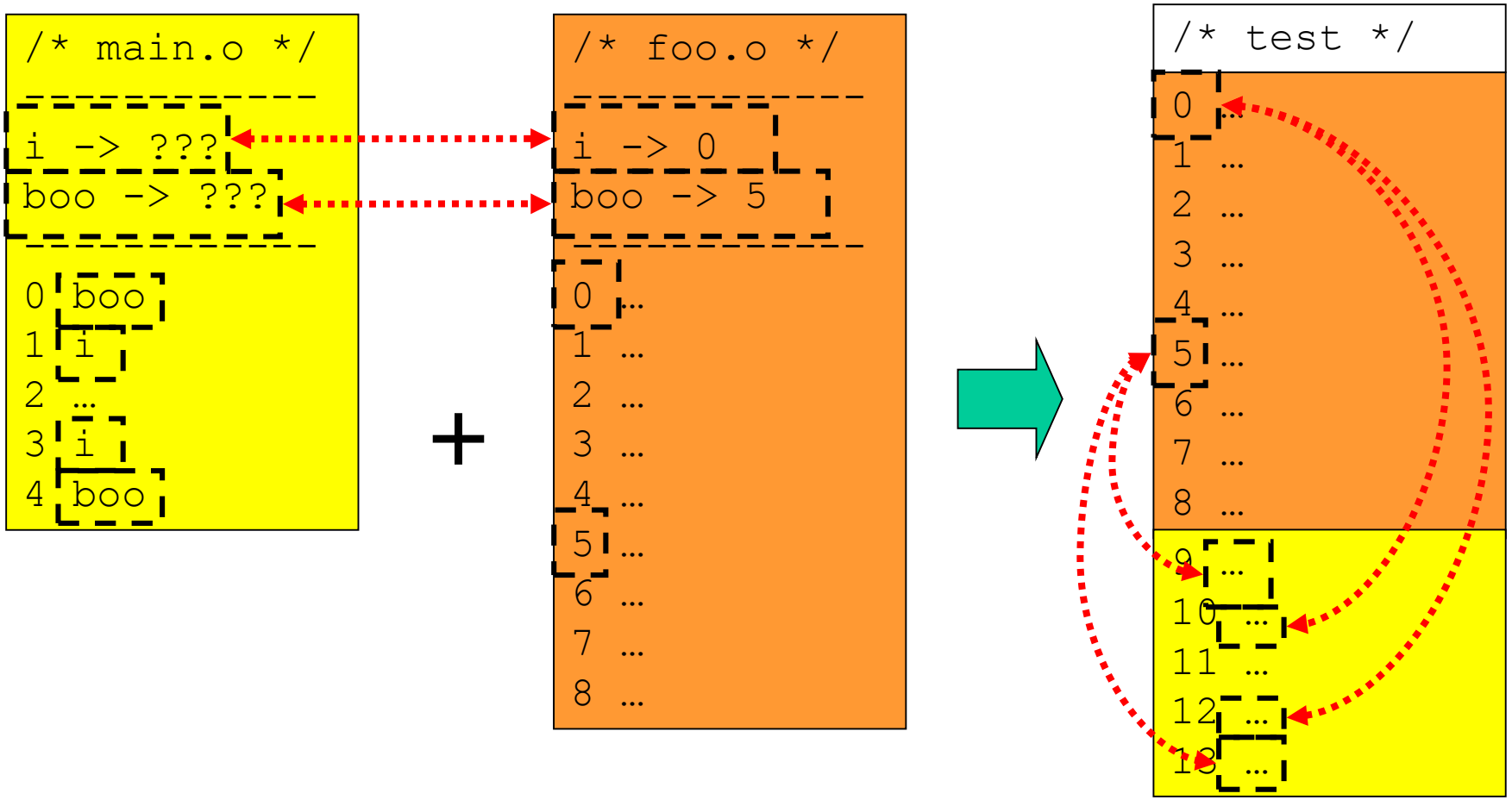
```
/* foo.h */

extern int i;

extern void boo();
```

```
/* main.c */

#include "foo.h"

int main(…) {

  boo();
  i++;
  boo();
}
```




```
/* main.o */
------------
i -> ???
boo -> ???
------------
0 boo
1 i
2 …
3 i
4 boo
```

```
> gcc main.c -c -o main.o
>
```




```
> gcc main.o foo.o -o test
>
```

Περισσότερα για το `extern`

- Ο προσδιορισμός `extern` χρησιμοποιείται για τις δηλώσεις **εξωτερικών** μεταβλητών / συναρτήσεων, που ορίζονται / υλοποιούνται σε ένα άλλο αρχείο.
- Η παράλειψη του `extern` σε δήλωση συνάρτησης (που δεν υλοποιείται τοπικά) οδηγεί σε **αναζήτηση** για **«ταιριαστή»** υλοποίηση της συνάρτησης στα αρχεία με τα οποία γίνεται σύνδεση του κώδικα.
- Η παράλειψη του `extern` σε δήλωση (καθολικής) μεταβλητής **δεν** εγγυάται τοπικότητα – αν μια άλλη καθολική μεταβλητή δηλώνεται σε άλλο αρχείο με το ίδιο όνομα, κατά την διασύνδεση οι αναφορές σε αυτό το όνομα θα αφορούν την **ίδια** θέση μνήμης.

```
/* a.c */

int i;

void incval() {
  i++;
}
```

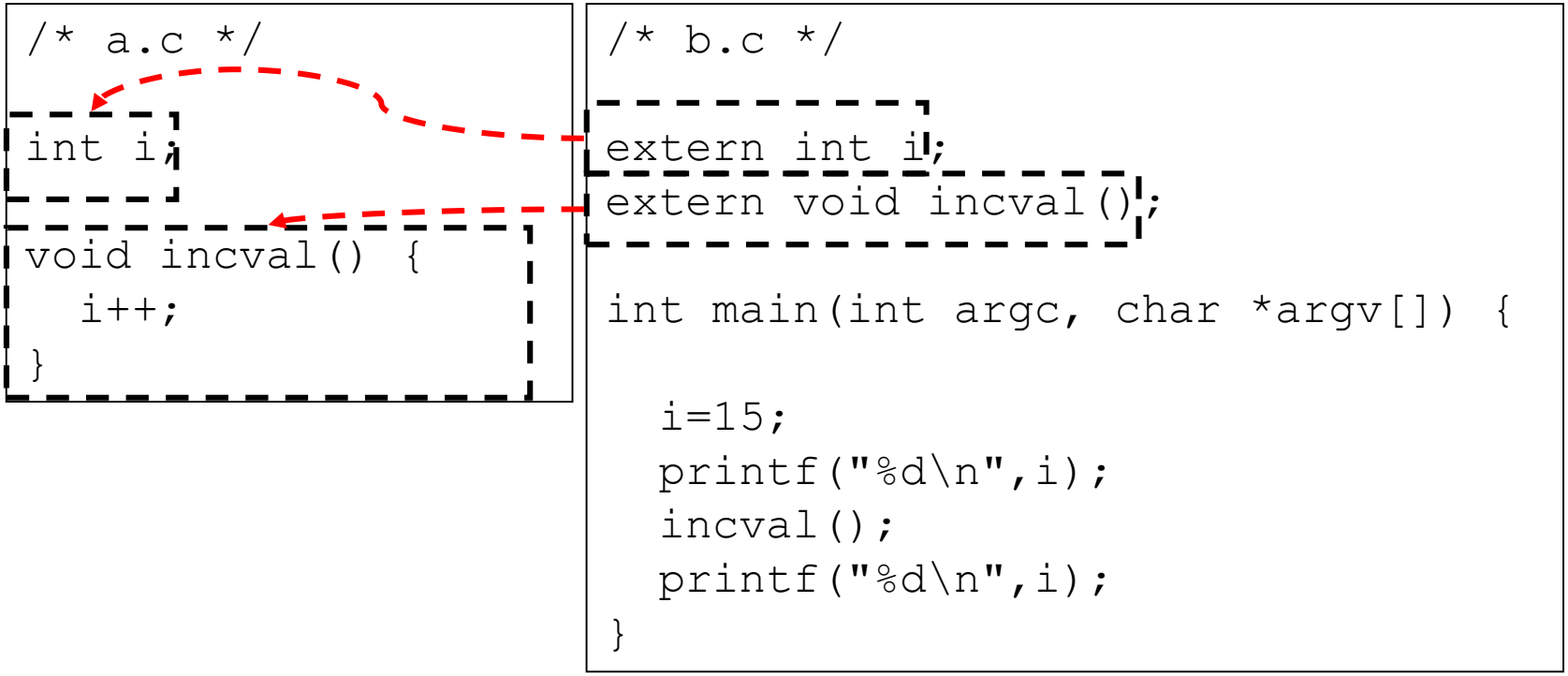
```
/* b.c */

extern void incval();

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

  i=15;
  printf("%d\n",i);
  incval();
  printf("%d\n",i);
}
```

```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
16
>
```

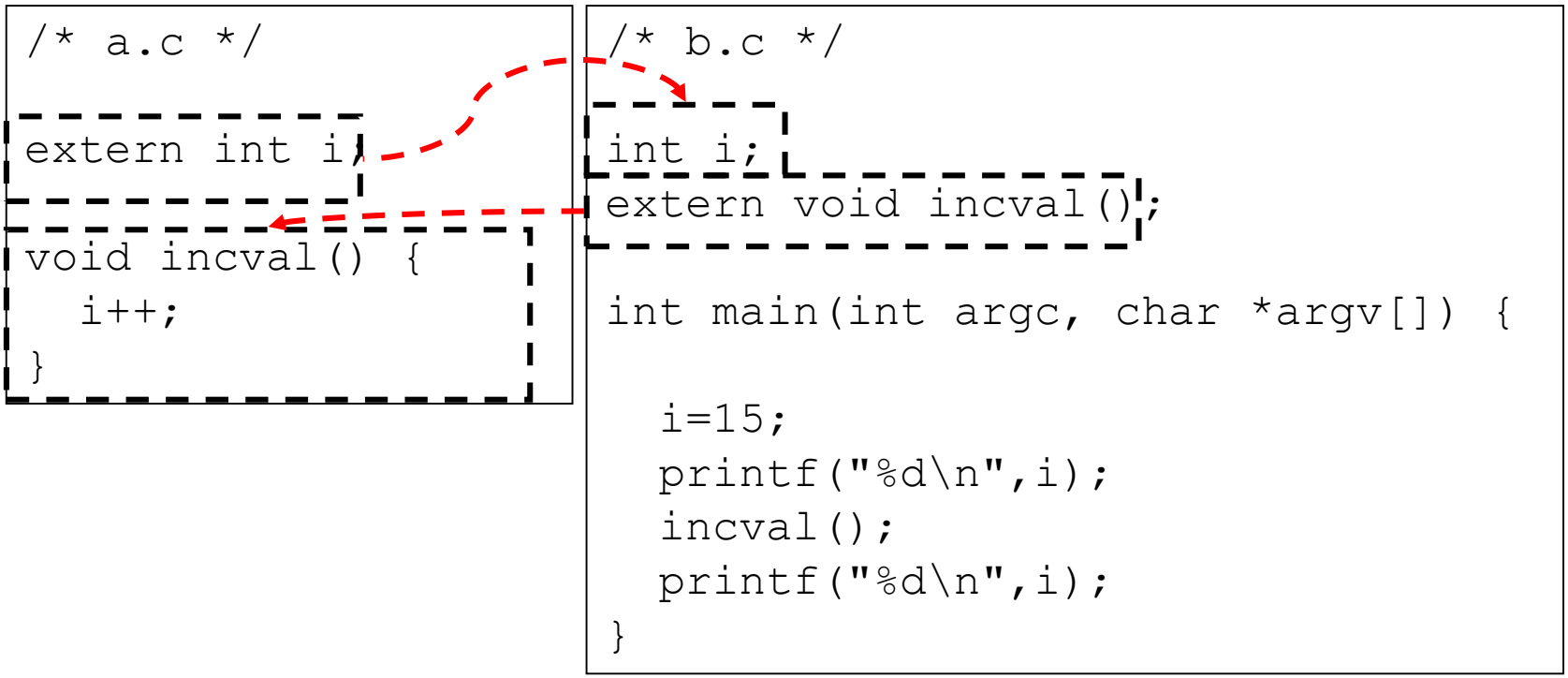
```c
/* a.c */

int i;

void incval() {
  i++;
}
```

```c
/* b.c */

extern void incval();

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

  i=15;
  printf("%d\n",i);
  incval();
  printf("%d\n",i);
}
```

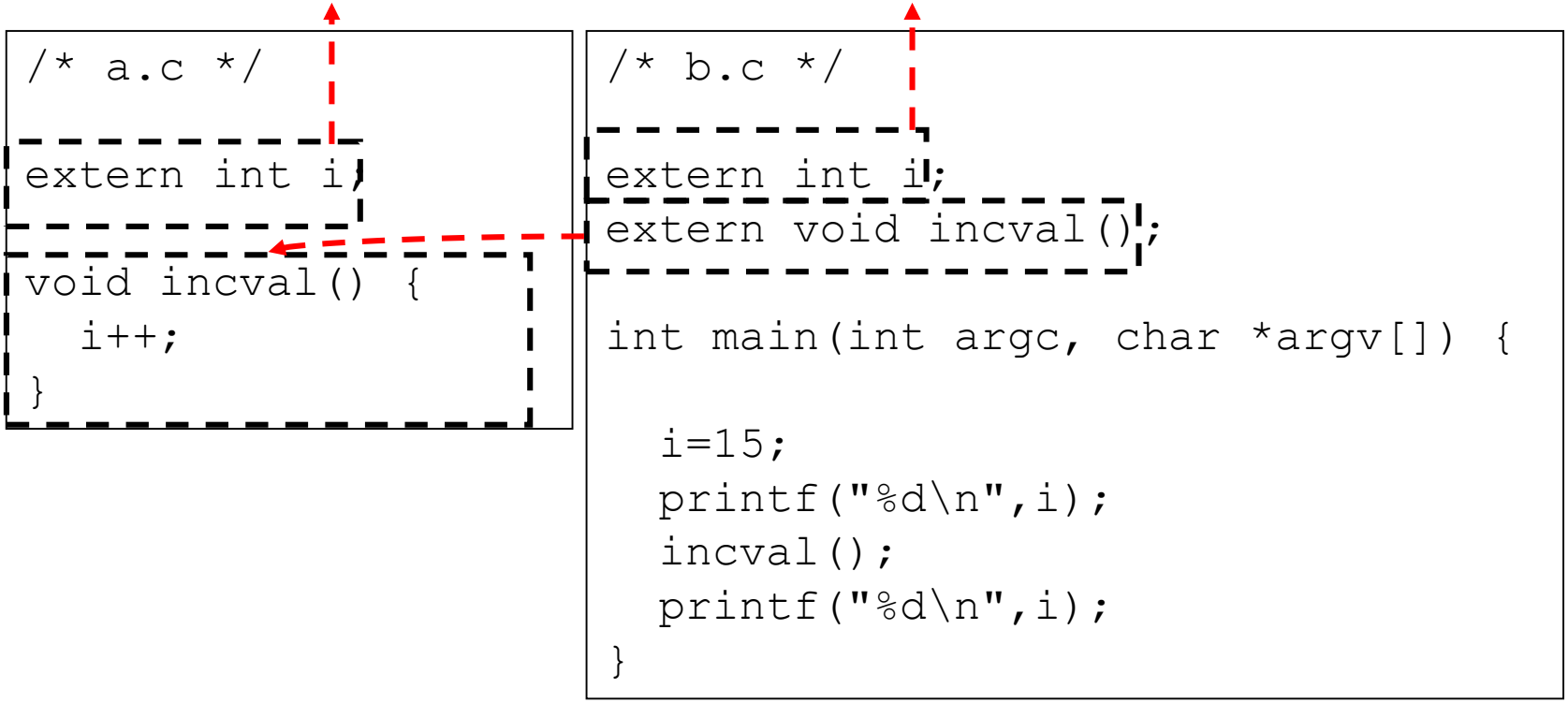
```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
15
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
16
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
16
>
```



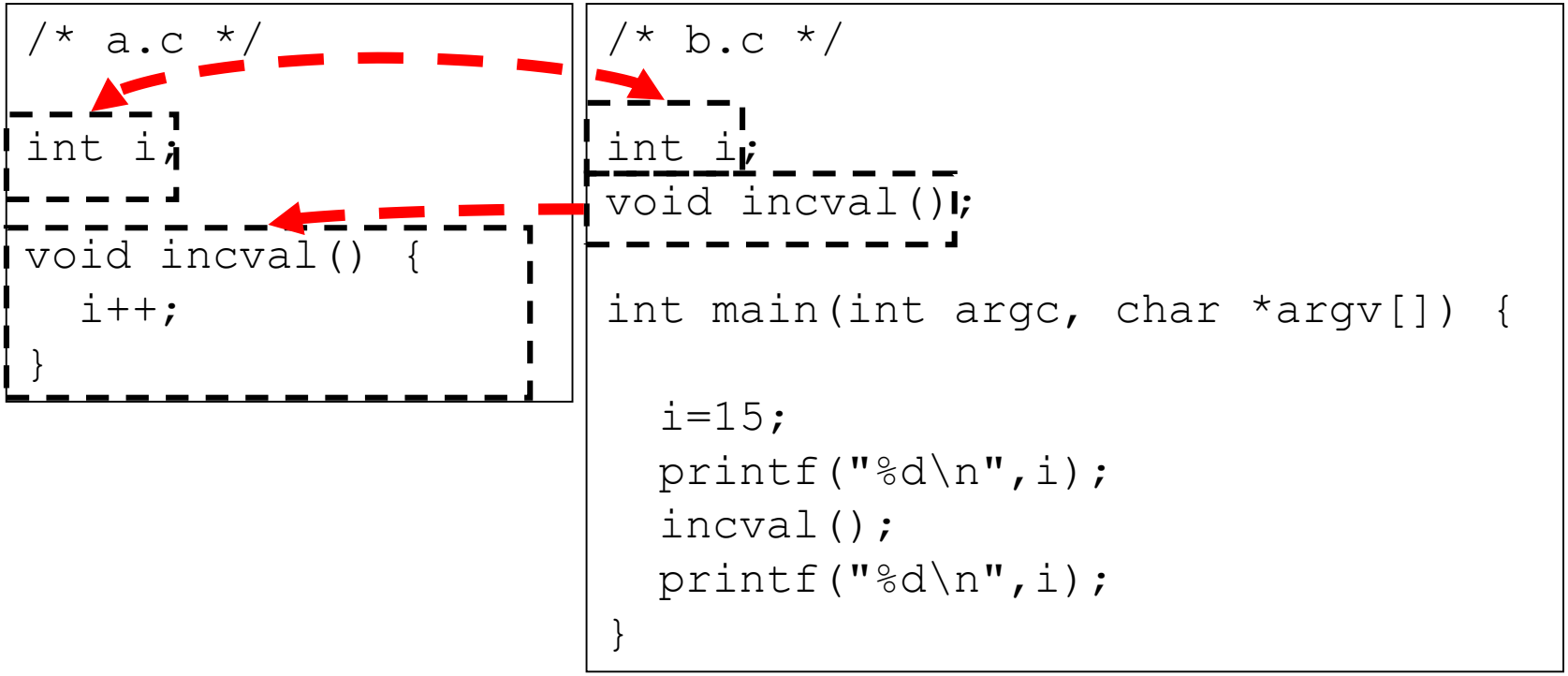

```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
undefined reference to _i
>
```

ὁμως ...

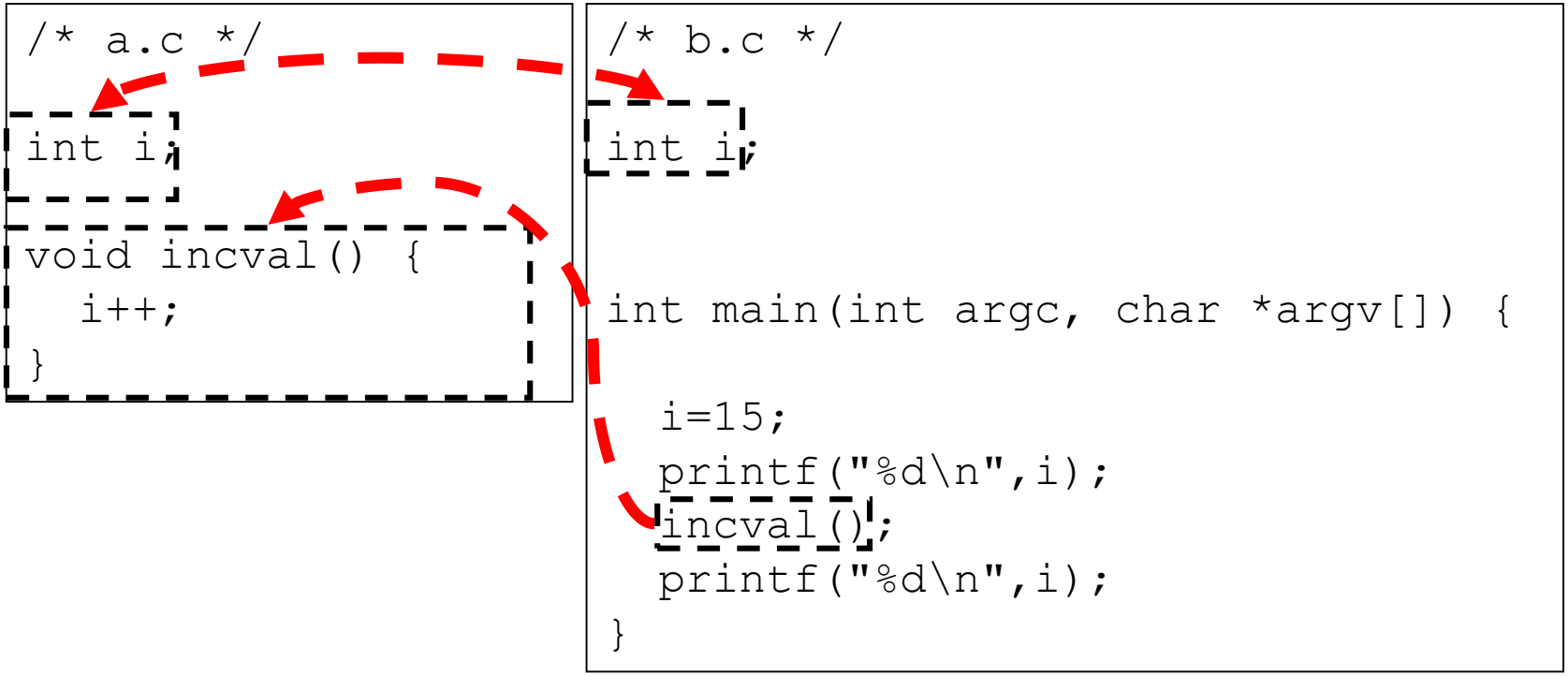
```c
/* a.c */

int i;

void incval() {
  i++;
}
```

```c
/* b.c */

int i;
extern void incval();

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

  i=15;
  printf("%d\n",i);
  incval();
  printf("%d\n",i);
}
```

```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
15
16
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
15
16
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>./test
15
16
>
```

Περισσότερα για το `static`

- Πως αποφεύγουμε μια καθολική μεταβλητή ή/και συνάρτηση που υλοποιούμε **αποκλειστικά για τους σκοπούς του τοπικού κώδικα** σε ένα αρχείο να «προσπελασθεί» (κατά λάθος) μέσα από κώδικα που βρίσκεται σε άλλο αρχείο;
- Με τον προσδιορισμό `static` επιτυγχάνεται η επιθυμητή τοπικότητα, δηλαδή η εμβέλεια των καθολικών μεταβλητών / συναρτήσεων **περιορίζεται** στο αρχείο όπου αυτές δηλώνονται / υλοποιούνται.
- Είναι αδύνατο να γίνει αναφορά σε αυτές (είτε επίτηδες είτε κατά λάθος) μέσα από κώδικα που βρίσκεται σε ένα άλλο αρχείο.

```c
/* a.c */

int i;

void incval() {
  i++;
}
```

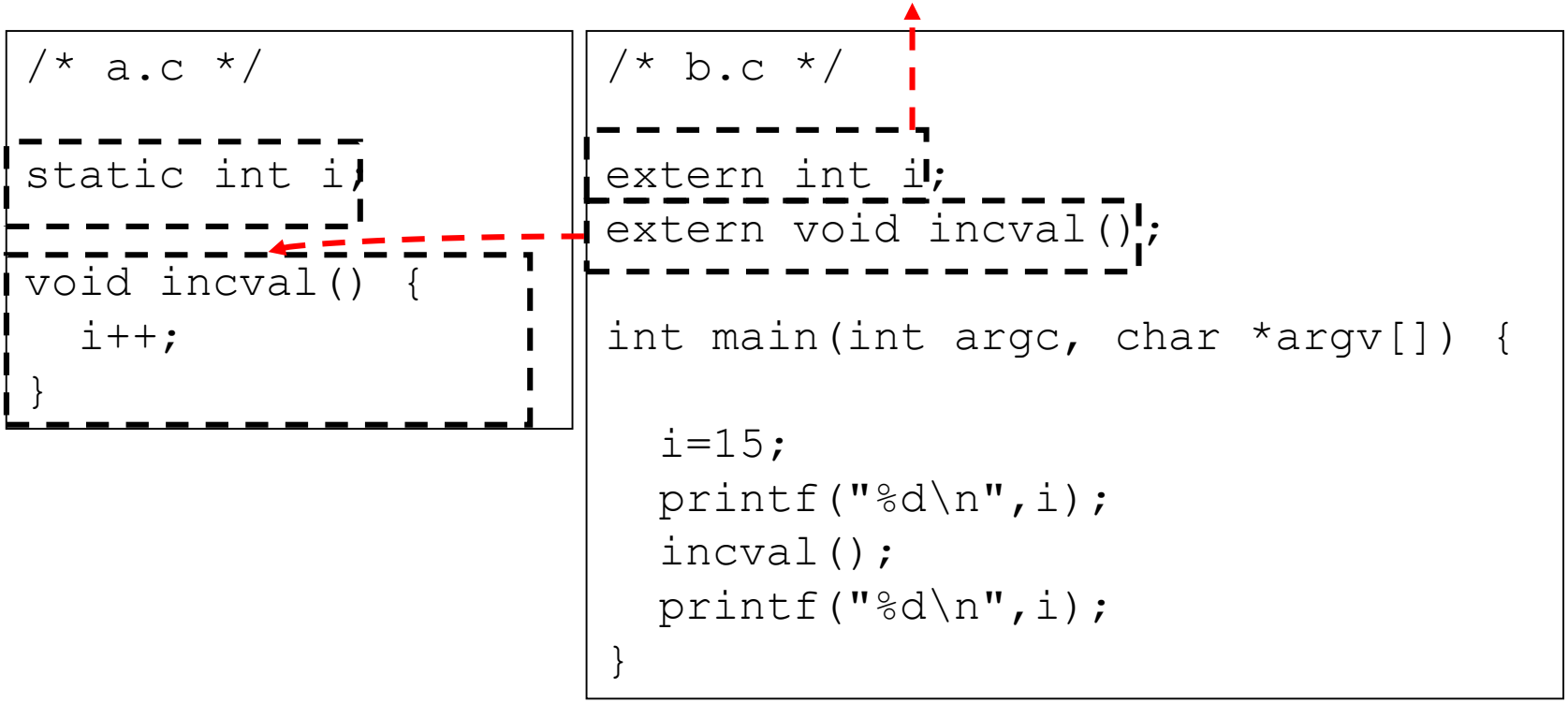
```c
/* b.c */

static int i;
extern void incval();

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

  i=15;
  printf("%d\n",i);
  incval();
  printf("%d\n",i);
}
```

```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
15
>
```

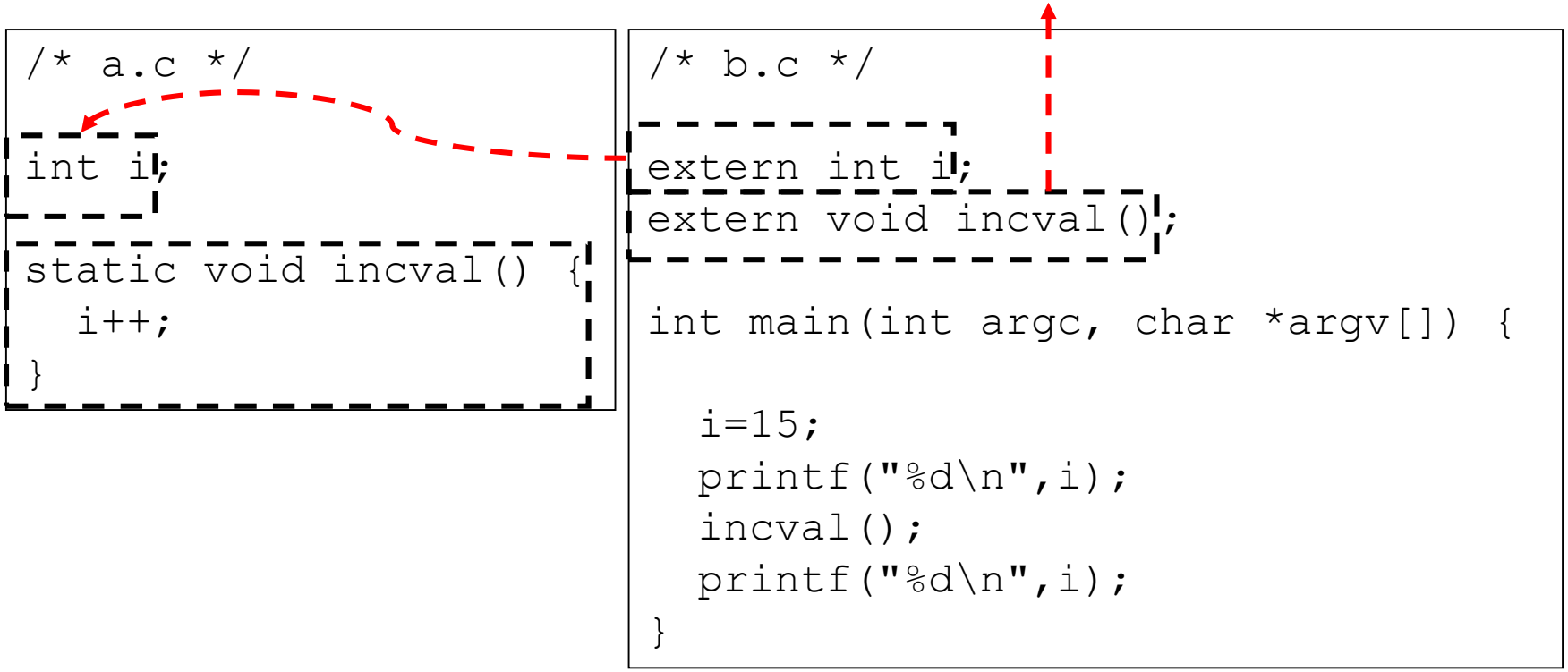
```
/* a.c */

static int i;

void incval() {
  i++;
}
```

```
/* b.c */

int i;
extern void incval();

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

  i=15;
  printf("%d\n",i);
  incval();
  printf("%d\n",i);
}
```

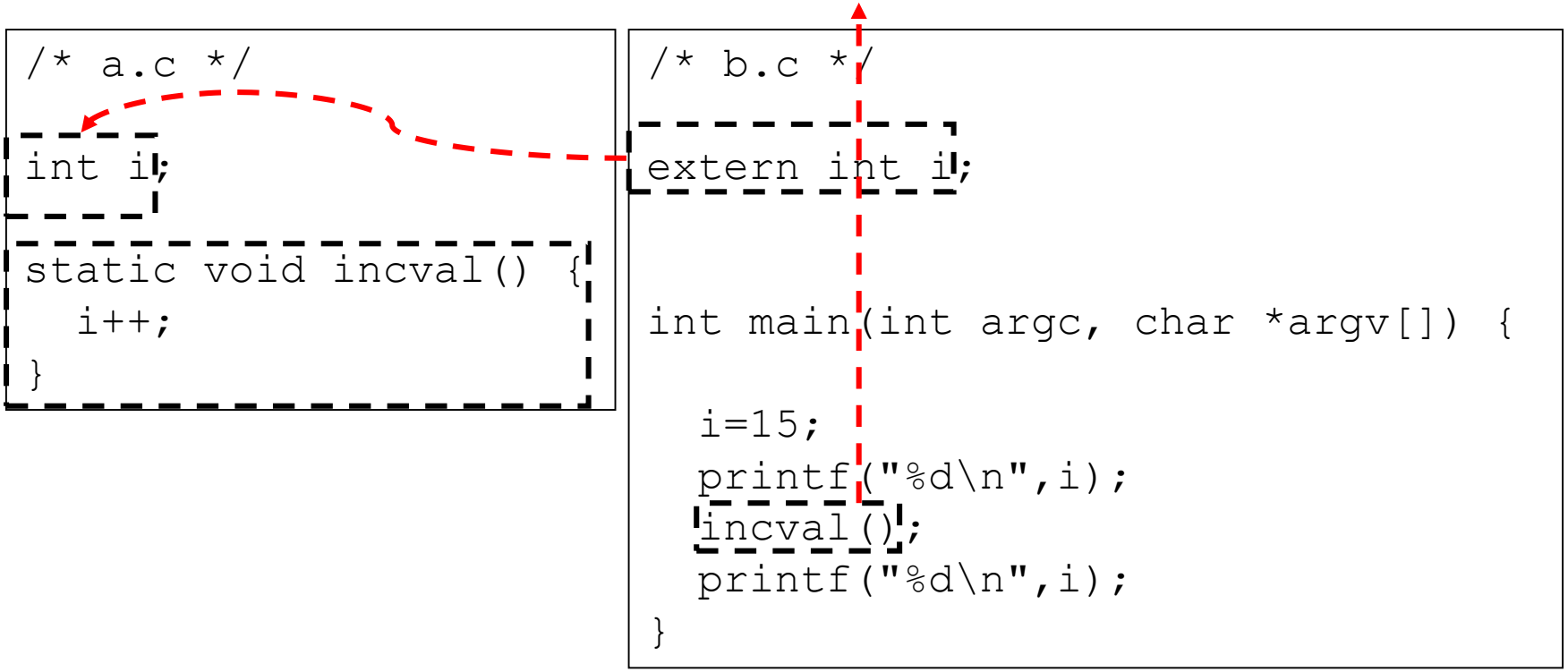
```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
>
>./test
15
15
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
undefined reference to _i
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
undefined reference to _incval
>
```




```
> gcc a.c -c -o a.o
> gcc b.c -c -o b.o
> gcc a.o b.o -o test
undefined reference to _incval
>
```

Ένα «μικρό» πρόβλημα

- Όταν ένας κώδικας Α ορίζει μεταβλητές και υλοποιεί συναρτήσεις με σκοπό αυτές να χρησιμοποιηθούν μέσα από οποιοδήποτε άλλο κώδικα, τότε αυτές δεν μπορεί να δηλωθούν ως `static`.
- Όταν ο κώδικας Α συνδεθεί με ένα άλλο κώδικα Β, οι μεταβλητές / συναρτήσεις του Α είναι διαθέσιμες για σύνδεση με αντίστοιχες (άμεσες ή έμμεσες) εξωτερικές δηλώσεις που υπάρχουν στον Β.
- Αν οι εξωτερικές δηλώσεις του Β έχουν προκύψει από **λάθος** (π.χ. παράλειψη προσδιορισμού `static` σε δήλωση καθολικής μεταβλητής / συνάρτησης ή παράλειψη δήλωσης και υλοποίησης συνάρτησης), η διασύνδεση θα οδηγήσει σε λάθος αποτέλεσμα.

Η ρίζα του προβλήματος

- Αναφορά σε εξωτερικές μεταβλητές / συναρτήσεις γίνεται με βάση έναν **επίπεδο** χώρο ονομάτων, όπου δεν μπορεί να αποκλεισθεί η **τυχαία** χρήση του ίδιου ονόματος από διαφορετικά τμήματα κώδικα.
- Ο προγραμματιστής δεν έχει τη δυνατότητα να προσδιορίσει το **τμήμα λογισμικού** το οποίο θα πρέπει να παρέχει τον ορισμό / υλοποίηση μιας εξωτερικής μεταβλητής / συνάρτησης.
- Άλλες γλώσσες λύνουν το πρόβλημα δίνοντας σε κάθε τμήμα λογισμικού ένα **μοναδικό όνομα** με βάση το οποίο άλλα τμήματα κώδικα μπορεί να αναφέρονται (χωρίς πιθανότητα λάθους) σε μεταβλητές και συναρτήσεις που αυτό προσφέρει.