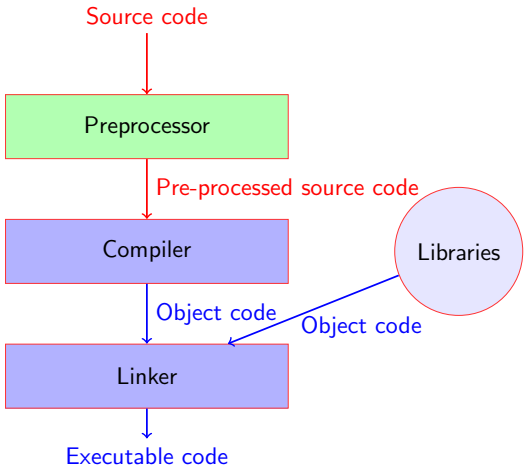
Primo passo: il preprocessor



Il preprocessore

Il preprocessore compie delle manipolazioni sul codice del programma. Precisamente:

- Elimina i commenti
- Esegue le istruzioni che iniziano con # (*direttive al preprocessore*)

Il codice sorgente prodotto verrà tradotto in codice oggetto dal compilatore.

Per compiere solo la precompilazione:

```
gcc -E file.c
```

Per salvare il contenuto in `file.i` (ai file prodotti dal preprocessore si assegna per convenzione il suffisso `.i`):

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

Il file `file.i` contiene codice sorgente (file ASCII), quindi che può essere letto con un qualunque editor.

Esempi di direttive al preprocessore: `#include`

```
#include nomeFile
```

Il prerprocessore sostituiscetale linea con il contenuto del file di nome `nomeFile`.

- Va utilizzata per inserire header file, ossia file che contengono definizioni necessarie per la compilazione del programma (hanno suffisso `.h`).
- Il nome del file va scritto:
  - Fra doppi apici se il file si trova nella directory corrente.

    ```
    #include  "f1.txt" /* f1.txt si trova nella directory corrente */
    ```

  - Fra < e > se il file non si trova nella directory corrente (deve però trovarsi una directory nota al sistema).

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


Esempi di direttive al preprocessore: `#define`

Con `#define` è possibile definire delle *macro* che permettono di usare un identificatore come abbreviazione di una espressione.
La definizione è visibile a partire dal punto in cui è posta.

Esempio

```
/* file  iter.c */
#include <stdio.h> /* serve per usare printf() */
#define  MASSIMO  4  /* definizione di una macro */

int main(){
    int n; /* definizione della variabile n di tipo int */
    n = 1; /* assegna a n  il valore 1 */
    while( n <=   MASSIMO) {
            printf(''Iterazione numero %d\n'', n);
            n++;   /* incrementa n di 1 */
        }
    return 0;
}
```

- L'istruzione

  `printf("Iterazione numero %d\n", n);`

  stampa la stringa fra apici sostituendo a `%d` il valore della variabile `n`.
- Il preprocessore sostituisce ogni occorrenza di `MASSIMO` con la sua definizione, ossia `4`.

Il programma stampa:

```
Iterazione numero 1
Iterazione numero 2
Iterazione numero 3
Iterazione numero 4
```


Esercizio

1. Dire cosa succede se nel programma precedente la definizione di `MASSIMO` è sostituita con una delle seguenti definizioni:

```
#define  MASSIMO   1+(2*5+10)

#define  MASSIMO   n+1

#define  MASSIMO   n+1;


#define  MASSIMO   x+1

#define  MASSIMO   MASSIMO+1
```

Verificare le sostituzioni effettuate dal preprocessore.

2. Cosa succede spostando la definizione di `MASSIMO` in fondo al file?

Attenzione !

- Spesso nelle definizioni è necessario l'uso delle parentesi.
  Esempio

  `#define  X   4+3`

  L'istruzione

  `n =  10 *  X;`

  dopo la precompilazione diventa

  `n =  10 *  4 + 3;`

  e, durante l'esecuzione, a `n` viene assegnato 43.

  Con

  `#define  X   (4+3)`

  la sostituzione produce la linea

  `n =  10 *  (4 + 3);`

  e a `n` viene assegnato 70.

- Si ricordi che `#define` non è un'istruzione, ma una direttiva al preprocessore, quindi non ci va il ";".

  Esempio

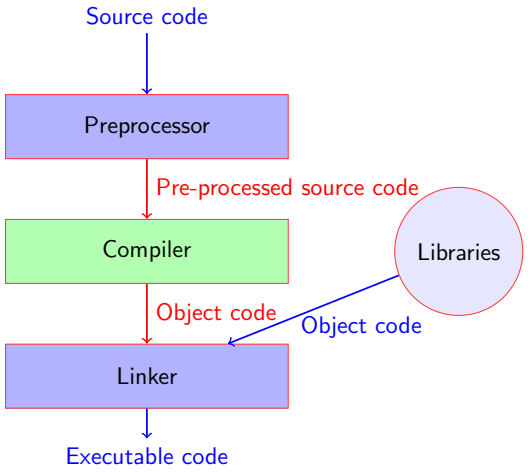
```
#define   MAX   4;
...
  if(n <  MAX)
    ...
```

  la sostituzione diventa

```
  if(n <=  4;)
```

  e produce un errore sintattico in compilazione (al passo 2).

Secondo passo: compiler



La compilazione

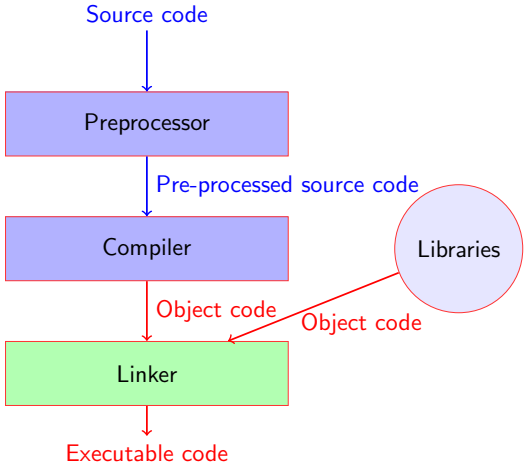
- Il secondo passo consiste nella traduzione (compiling) del codice sorgente ASCII (già trattato dal preprocessore) in codice oggetto binario.
- Il codice viene analizzato in modo sequenziale, quindi una definizione è visibile dal punto in cui è scritta scritta in avanti.
- Il codice oggetto ottenuto può non essere ancora eseguibile in quanto:
  - ci sono chiamate a funzioni di cui non è definito il codice (negli esempi precedenti, le chiamate a `printf()`);
  - la funzione `main()` non è definita.

Per eseguire solo i primi due passi di compilazione:

```
gcc -c file.c
```

Viene generato il file `file.o` che contiene il codice oggetto che traduce il codice sorgente in `file.c`.

Terzo passo: il linker



Il linker

Il compito del linker è quello di produrre un file eseguibile a partire da uno o più file contenenti codice oggetto (ossia, codice già tradotto dal compilatore).

- Se in un file `file.o` c'è la chiamata a una funzione `f()`, il linker cerca nei file oggetto a disposizione (o eventualmente nelle librerie) il codice oggetto di `f()`.
  - Se la ricerca ha esito positivo, il linker "collega" la chiamata della funzione al suo codice.
  - Se il codice di `f()` non è trovato, oppure se esiste più versioni di `f()`, il linker fallisce.
- Fra i file oggetto deve essere definita una e una sola funzione `main()`.

Nota

Se la compilazione avviene con un unico comando, i file intermedi prodotti al passo 1 e 2 non sono visibili all'utente.

Esempio 1: il programma `cerchio.c`

Il programma `cerchio.c` calcola il perimetro e l'area di un cerchio di raggio intero. Il valore del raggio è inserito dall'utente.

```
#include <stdio.h>

#define PI  3.14159

/* calcola il perimetro del cerchio di raggio intero  r */

double perimetro(int r){
  double p;
  p =  2*PI*r;
  return p;
}

/* calcola l'area del cerchio di raggio intero  r */
double area(int r){
  double a;
  a =  PI*r*r;
  return a;
}
```



```
int main(){
  int raggio;
  double p,a;
  printf("Inserire il raggio (valore intero)  ---> ");
  scanf("%d", &raggio); // raggio  contiene l'intero letto in input
  p = perimetro(raggio);
  a = area(raggio);
  printf("Perimetro = %.5f\n", p);
  printf("Area  = %.5f\n", a);
  return 0;
}
```


Definizione di una funzione

```
/* calcola il perimetro del cerchio di raggio intero  r */

double perimetro(int r){
 ...
}
```

è la definizione della funzione `perimetro()`.
Può essere vista comeè un sottoprogramma che, dato in ingresso un intero r, restituisce il valore (di tipo `double`) del perimetro del cerchio di raggio r.

- La variabile `r` di tipo `int` è il parametro formale della funzione. Il valore iniziale di `r` è stabilito a run time, quando la funzione viene chiamata.
  I parametri formali permettono di passare dei valori a una funzione.
- La variabile `r` di tipo `double` è una variabile locale della funzione, utilizzata per memorizzare risultati intermedi

- I parametri formali e le variabili locali definiti in una funzione `f()` sono risorse utilizzabili esclusivamente all'interno di `f()`. Non possono essere nominate fuori da `f()`.

  Nota
  La variabile `p` definita in `perimetro()` e la variabile `p` definita in `main()` sono due entità distinte.

- L'istruzione

  `return p`

  termina l'esecuzione della funzione `perimetro()` restituendo al chiamante il valore di tipo `double` contenuto nella variabile `p`.
- Il prototipo di `perimetro()` è:

  `double perimetro(int r);`

  oppure, equivalentemente,

  `double perimetro(int);`

In `main()`:

- L'istruzione

  ```
  scanf("%d", &raggio);
  ```

  richiede la lettura in input di un intero e pone l'intero letto nella variabile `raggio` (variabile locale a `main()`).
- L'istruzione

  ```
  p = perimetro(raggio);
  ```

  provoca l'interruzione di `main()` e la chiamata alla funzione `perimetro()` a cui, come valore iniziale del parametro `r` viene assegnato il valore della variabile `raggio`.
  Al termine dell'esecuzione di `perimetro()`, il valore restituito (perimetro del cerchio) è assegnato alla variabile `p` (locale a `main()`).
  Il controllo passa di nuovo a `main()` che continua l'esecuzione.

- L'istruzione

  ```
  printf("Perimetro = %.4f\n", p);
  ```

  stampa la scritta

  `Perimetro = ...`

  dove al posto dei puntini viene stampato il valore della variabile `p` (locale a `main()`) con 4 cifre decimali.

Uso dei prototipi

Per tradurre in codice oggetto, il compilatore dve conoscere il prototipo delle funzioni utilizzate nel codice.

L'esempio 1 è corretto in quanto le funzioni sono definite *prima* che vengano chiamate e dalla definizione di una funzione si ricava il suo protitipo.

Esempio 2

In `cerchio1.c` la definizione della funzione `area()` è spostata dopo `main()`.
Poiché in `main()` c'è una chiamata alla funzione `area()`, occorre scrivere *prima* di `main()` il prototipo:

```
double area(int r);
```



```
#include <stdio.h>

#define PI  3.14159

double area(int r);  /* definizione del prototipo */

/* calcola il perimetro del cerchio di raggio intero  r */

double perimetro(int r){
 ... COME PRIMA ...
}

int main(){
  ... COME PRIMA ...
}

/* calcola l'area del cerchio di raggio intero  r */
double area(int r){
  ... COME PRIMA ...
}
```

Compilazione su file separati

Suddividiamo il programma in due file:

- File `main.c`
  Contiene solamente la definizione della funzione `main()`
- File `calc.c`
  Contiene la definizione delle funzioni per calcolare perimetro e area.

Si osservi che:

- In `main.c` occorre definire i prototipi delle funzioni di cui non è definito il codice: `perimetro()`, `area()`, `printf()` e `scanf()`.
  Non è necessario definire il valore di `PI`, in quanto la definizione non è utilizzata.
- In `calc.c` occorre definire il valore di `PI`.
  Non è necessario includere `stdio.h` in quanto non vengono utilizzate funzioni di libreria.

main.c

```
#include <stdio.h>

double area(int r);
double perimetro(int r);

int main(){
  int raggio;
  double p,a;
  printf("Inserire il raggio (valore intero)  ---> ");
  scanf("%d", &raggio); // raggio  contiene l'intero letto in input
  p = perimetro(raggio);
  a = area(raggio);
  printf("Perimetro = %.4f\n", p);
  printf("Area  = %.4f\n", a);
  return 0;
}
```


calc.c

```
#define PI  3.14159

/* calcola l'area  del cerchio di raggio intero  r */

double area(int r){
  double a;
  a =  PI*r*r;
  return a;
}


/* calcola il perimetro del cerchio di raggio intero  r */

double perimetro(int r){
  double p;
  p =  2*PI*r;
  return p;
}
```


Compilazione di un programma suddiviso su più file

Nell'esempio precedente occorre dare i seguenti comandi di compilazione.

1. `gcc -c main.c`
   Crea il file oggetto `main.o`
2. `gcc -c calc.c`
   Crea il file oggetto `calc.o`
3. `gcc  main.o  calc.o`
   Chiama il linker sui fule oggetto `main.o` e `calc.o` e crea il file eseguibile.
   Il linker associa le chiamate alle funzioni `perimetro()` e `area()` contenute in `main.o` al codice oggetto delle corrispondenti funzioni in `calc.o`.

Si può fare in un unico passo:

```
gcc  main.c  calc.c
```

In questo caso non vengono generati esplicitamente i file oggetto `main.o` e `calc.o`.

Vantaggi

La modifica di una funzione richiede solamente la ricompilazione del file in cui la funzione è definita

Esempio 3
Se si modifica un messaggio nella funzione `main()`, occorre rigenerare solamente il file `main.o`, ma non `calc.o`

```
gcc -c main.c
gcc main.o calc.o
```

Oppure, in un unico passo:

```
gcc main.c calc.o
```

In questo caso non viene generato esplicitamente `main.o`.

Esercizio

Provare a dare il comando di compilazione

`gcc main.c`

Perché la compilazione fallisce? In quale fase è sollevato l'errore?
Fare la stessa cosa con il file `calc.c`.

Prototipi scorretti

Supponiamo di scrivere in modo scorretto il prototipo di `area()`

```
int area(int r);
```

Il compilatore, quando genera il codice oggetto di `main()`, *assume* che la funzione `area()` restituisca un intero.
Quindi, nella traduzione inserisce un cast al tipo `int` del valore restituito dalla chiamata ad `area()` prima che venga assegnato alla variabile a.

È come se nel codice ci fosse scritto:

```
a = (int) area(raggio);
```

dove `(int)` è l'operazione di "cast al tipo `int`".
La compilazione ha successo, ma il comportamento del programma è scorretto (verificarlo).

Uso di header file

Per agevolare la scrittura dei prototipi si usano gli header file (file con suffisso `.h`).

Esempio 4

- Scriviamo un header file `calc.h` contenente solamente i prototipi delle funzioni definite in `calc.c`

```
double area(int r);
double perimetro(int r);
```

- Definiamo il seguente file `main1.c`

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

int main(){
 ... COME PRIMA ...
}
```

Si assume che il file `calc.h` sia nella directory corrente.

Compilando con

```
gcc main1.c calc.c
```

si ha lo stesso effetto di prima
Infatti, il preprocessore sostituisce la linea

```
#include "calc.h"
```

di `main1.c` con il contenuto di `calc.h`.
Quindo, i file ottenuti preprocessando `main1.c` e `main.c` sono identici (verificarlo).

Nota

I file `.h` sono visti solo dal preprocessore e non dal linker.
Il linker cerca il codice oggetto delle funzioni in *tutti* i file oggetto su cui è invocato e nelle librerie (vedere il prossimo esercizio).

Esercizio

Sia `p.c` il file

```
double perimetro(int r){
  return 0;
}
```

e `a.c` il file

```
double area(int r){
  return 0;
}
```

Generare i file oggetto `main1.o`, `a.o` e `p.o`

```
gcc -c main1.c
gcc -c a.c
gcc -c p.c
```

Dire cosa succede dando al linker i seguenti comandi

```
gcc main1.o a.o p.o

gcc main1.o calc.o p.o

gcc main1.o a.o

gcc a.o p.o
```


Uso scorretto di `#include`

Il comando `#include` non va utilizzato per includere file contenente istruzioni.

Esempio 5
Consideriamo il seguente programma `main2.c`

```
#include <stdio.h>
#include "calc.c"

int main(){
... COME PRIMA ...
}
```

Il programma può essere compilato con il comando

```
gcc main2.c
```

e il compilatore non segnala alcun tipo di warning.

Tuttavia, il file ottenuto dalla precompilazione di `main2.c` contiene tutto il codice del programma (come nel programma iniziale `cerchio.c`).

Non è quindi possibile compilare separatamente una porzione di programma come avviene quando il programma è suddiviso nei file `main.c` (oppure `main1.c`) e `calc.c`.

Riassumendo:

- I file `.h` devono contenere solo definizioni (prototipi, macro, ...).
- I file `.c` non vanno mai inclusi in altri file, ma vanno intesi come moduli che possono essere compilati in modo indipendente dal resto del programma.

Uso di `makefile`

Quando si suddivide un programma su più file, è opportuno generare l'eseguibile usando il comando *make*.
Occorre definire un file di nome `makefile` contenente regole del tipo:

```
file: file1 ... fileN
      istruzioni
```

Nota
Gli spazi prima di `istruzioni` corrispondono a un carattere di tabulazione.

- La prima riga (riga di *dipendenza*) dichiara che il file `file` dipende dai file `file1`,..., `fileN`.
  Questo significa che per generare `file` occorre generare prima i file `file1`,..., `fileN`.
  Se i file `file1`,..., `fileN` non sono stati modificati (ossia, se le date dell'ultima modifica di tali file sono anterirori alla data dell'ultima modifica di `file`), non occorre rigenerare `file`
- La seconda riga (riga di *comando*) specifica quale comando occorre eseguire per ottenere `file`.

Il comando `make`

Quando `make` è eseguito, compie tutte le azioni necessarie per generare il *primo* file nella lista (vedere il libro per maggiori dettagli).

Esempio 6

Per compilare `main1.c` e `calc.c` si può usare il seguente `makefile`

```
cerchio: main1.o calc.o
gcc  main1.o calc.o -o cerchio

main1.o: main1.c calc.h
gcc  -Wall -c  main1.c

calc.o: calc.c
gcc  -Wall -c calc.c
```


Il comando `make`

Dando il comando

```
make
```

la prima volta, viene stampato

```
gcc  -Wall -c  main1.c
gcc  -Wall -c calc.c
gcc  main1.o calc.o -o cerchio
```

che indica i comandi di compilazione eseguiti per generare `cerchio` (nome dato all'eseguibile).

Se si modifica `main1.c` e si esegue di nuovo `make`, vengono eseguiti i seguenti comandi di compilazione:

```
gcc  -Wall -c  main1.c
gcc  main1.o calc.o -o cerchio
```

Infatti:

- Il file `calc.o` dipende da `calc.c` che non è stato modificato, quindi `calc.o` non richiede di essere di nuovo generato.
- Il file `main1.o` dipende da `main1.c`, che è stato modificato *dopo* la creazione di `main1.o`, dunque va ricreato con il comando
  ```
  gcc  -Wall -c  main1.c
  ```
- Siccome `cerchio` dipende da `main1.o` che è stato modificato, va ricreato con il comando
  ```
  gcc  main1.o calc.o -o cerchio
  ```

Esercizio

Provare a modificare i file in vari modi ed eseguire dopo ogni modifica il comando `make`.