Compilazione separata

Un compilatore C/C++ lavora su file sorgenti. La compilazione di un progetto comprende:

INPUT	FASE	OUTPUT
Source file	→ *Preprocessing* →	*translation unit*
translation unit	→ *Translation* →	*object file*
1 o più object file	→ *Linking* →	*executable (program)*

Compilazione: generazione di un file oggetto (preprocessing e traduzione) oppure generazione di un eseguibile (tutte e tre le fasi).

Preprocessing

Prima fase svolta dal preprocessore che svolge tre compiti mediante le direttive inserite nel file sorgente:

- **Inclusione di file `(#include)`**
- **Definizione/espansione di macro `(#define)`**
- **Compilazione condizionale `(#ifdef, #if)`**

Unità di traduzione (translation unit): testo risultante dell'applicazione delle direttive passato alla fase di traduzione

Macro

La direttiva `#define` permette di definire macro di due tipi:

- **Macro senza argomenti**
- **Macro con argomenti**

Corrispondo a sostituzioni testuali durante la fase di preprocessing (prima della fase di compilazione vera e propria).

Nei punti del codice dopo la direttiva `#define` in cui compare la sequenza di caratteri ID viene sostituita con DEF.

```
#define ID DEF
```

Si possono inserire anche codici più complessi.

```
#define INC(a) (a)++
```

Si possono definire macro con più di un argomento. Una macro va scritta tutta sulla stessa riga. Per migliorare la leggibilità si può andare a capo, ma a patto di mettere un backslash subito prima del newline

Vantaggio rispetto alla funzione: non si esegue una chiamata a funzione e questo fa risparmiare tutto il processo annesso risparmiando anche la copia per valore in caso che a e b siano due variabili con dimensioni consistenti.

```
#define PRINT_MAX(a, b) if ((a) > (b)) \
                            cout<<(a) ; \
                        else \
                            cout<<(b);
```

Macro vuota → dal punto nel codice in cui compare la direttiva #define in poi, si elimina dal sorgente la sequenza di caratteri ID

```
#define ID
```

Compilazione condizionale

Direttiva al pre-processore #ifdef (if defined): controlla se un identificatore è stato definito come una macro. Si tratta di una direttiva per la **compilazione condizionale.** Si possono definire parti alternative di codice da compilare all'interno di un file sorgente.

SINTASSI

```
#ifdef identificatore
     sequenza istruzioni
[#elif identificatore
         sequenza istruzioni]*
[ #else
     sequenza istruzioni ]
#endif
```

Esempio

```
#define UGO 0

int main() {
  cout << "Hello" << endl;
#ifdef UGO
  cout << "UGO defined" << endl;
#ifdef ADA
  cout << "ADA defined" << endl;
#endif
#endif
}
```

Direttiva **#define**
NOTA 1: il valore con cui è definito l'identificatore è ininfluente!
NOTA 2: la macro UGO potrebbe anche essere vuota*!*

Debugging:

```
#ifdef DEBUG
cout<<«valore di a: <<a;
#endif
```

Aggiungere codice a programmi esistenti:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#else
cin>>dim;
int* table = new(dim);
#endif
```

Direttiva #ifndef

Se la macro ID non è definita si compila la sequenza di righe 1 altrimenti la sequenza di righe 2.

```
#ifndef ID
<sequenza di righe 1>
[#else
<sequenza di righe 2>]
#endif
```

La funzione main deve essere definita in un unico file e affinchè l'output sia unico il file contenente la funzione main deve usare almeno un identificatore (variabile, costante, funzione) di almeno un altro dei file sorgente; pertanto **ogni altro file sorgente deve essere raggiungibile indirettamente o direttamente dalla funzione main.** Un file sorgente che non rispetta questa caratteristica è ininfluente alla generazione dell'eseguibile

Fase di traduzione

Dopo il preprocessing c'è una fase di traduzione in cui **ogni singola translation unit viene compilata e tradotta in modo completamente separato dalle altre. L'esguibile completo non potrà mai essere generato a partire da un singolo file.**

Il risultato della fase di traduzione è un file oggetto.

File oggetto: file binario (stesso linguaggio dell'eseguibile); file incompleto perchè fa usi di oggetti definiti altrove. Nel file oggetto c'è solo indicato che tali oggetti sono definiti in qualche altro file oggetto senza essere specificato dove di preciso.

Simboli: identificatore degli oggetti presenti nei file oggetti. Quindi nell'unico file oggetto contenente il simbolo main c'è almeno un simbolo definito in uno degli altri file oggetto.

Fase di collegamento

Fase effettuata dal linker che si occupa di collegare i file oggetto per formare un unico file eseguibile completo. **Avviene una sostituzione di tutti i simboli presenti nei file oggetto con dei riferimenti al punto in cui il simbolo è definito.** Se il riferimento non esiste il linker si arresta segnalando errore.

I collegamenti tra i diversi file oggetto possono avvenire dinamicamente o staticamente. Quello che avviene nel nostro programma è il **collegamento statico**.

Dal sorgente all'eseguibile con il compilatore gcc (analizzazione della compilazione separata)

Il comando g++ è un front-end (interfaccia) che invoca automaticamente tutte le componenti del compilatore gcc; a sua volta **gcc è costituito da moduli distinti dedicati ognuno a una delle fasi della compilazione.**

- **cc1plus:** pre-processore e traduttore in assembly per il C++ che genera un file assembly a partire dal corrispondente file-sorgente.
- **as**: traduttore in linguaggio macchina che genera un file oggetto a partire da un file assembly
- **collect2**: invocatore del linker che genera un file eseguibile collegando assieme (linker) tutti i file oggetto passati in ingresso

Con l'opzione -c il compilatore si ferma alla generazione del file oggetto (nomefilesorgente.o).

```
g++ -c -Wall ciao_mondo.cc
```

Serve per file di grosse dimensioni in modo da evitare la ricompilazione in caso di un errore di sintassi.

Con l'opzione -v si compila il programma in modalità verbose in modo da vedere chiaramente tutte le fasi della compilazione.

Input/output: gli oggetti cin, cout, <<, >>, vista la compilazione separata, sono definiti e dichiarati nella libreria iostream. L'operatore di uscita e di ingresso (<<, >>) non sono altro che funzioni.

Il compilatore g++, vista la sintassi con le parentesi angolate, cerca il file iostream nelle directory predefinite.

Per utilizzare entità definite in una qualche libreria standard del C++, quali ad esempio l'operatore di uscita <<, si includono degli header file standard, che contengono solo l'interfaccia della libreria con la direttiva #include.

Lo schema è quindi identico a quello che abbiamo utilizzato per scrivere i nostri programmi su più file.

La differenza sta soltanto nel tipo di collegamento.

In questo caso il linker esegue un collegamento dinamico in quanto non unisce il file alla libreria standard ma mette semplicemente tutte le informazioni per rintracciare nelle librerie i simboli non definiti.

E' possibile forzare il linker ad effettuare un collegamento statico con l'opzione -static.

```
g++ -Wall -static ciao_mondo.cc
```

oppure

```
g++ -static ciao_mondo.o
```

Eseguibile dinamico → eseguibile prodotto mediante collegamento dinamico alle librerie	**Eseguibile statico** → eseguibile prodotto per mezzo di un collegamento statico alle librerie
Vantaggi: ● Dimensioni minori ● Non si spreca spazio replicando le entità contenute nelle librerie in tutti gli eseguibili (da qui le librerie dinamiche prendono il nome di librerie condivise, in microsoft DDL)	**Vantaggi:** ● Un eseguibile collegato dinamicamente ad una data libreria condivisa potrà essere eseguito solo su sistemi che contengono tale libreria. Quindi un eseguibile statico è più portabile. ● Si crea una dipendenza tra eseguibile e librerie

Compilazione separata e ricompilazione

Compilazione separata
(l'ordine in cui sono riportati i file oggetto non conta)

```
g++ -c file1.cc file2.cc ... fileN.cc
g++ file1.o file2.o ... fileN.o
```

Cosa fa il linker:

1) collega staticamente gli N file oggetto in un unico file oggetto
2) siccome non è specificata l'opzione -static collega dinamicamente tale file alle librerie ottenendo un eseguibile

Un eseguibile va cambiato solo se qualcuno dei file oggetto è cambiato, ossia se il file sorgente (o un file incluso) ha subito delle modifiche. Se è cambiato solo un file sorgente non ha senso tradurre anche tutti gli altri.

Vantaggi:

- L'operazione di collegamento dei file oggetto è estremamente più semplice e veloce della traduzione dei corrispondenti file sorgente. Quindi grazie alla compilazione separata, la ricompilazione in seguito a cambiamenti è più rapida.

- La semplicità di implementazione di programmi scritti su file sorgenti con diversi linguaggi. Ciascuno dei file sorgenti viene tradotto separatamente mediante il compilatore opportuno.

Le specifiche di collegamento permettono a file oggetto risultanti dalla traduzione di file in C++ di accedere a funzioni scritte in altri linguaggi perché, magari, più efficienti. È un'estensione della dichiarazione extern.

Sintassi `extern "nome_linguaggio"`

Esempio di funzione fun definita in un file sorgente C

```
extern "C" int fun(int)
```

Pertanto *fun* verrà tradotta secondo il linguaggio C.

Compilazione automatica

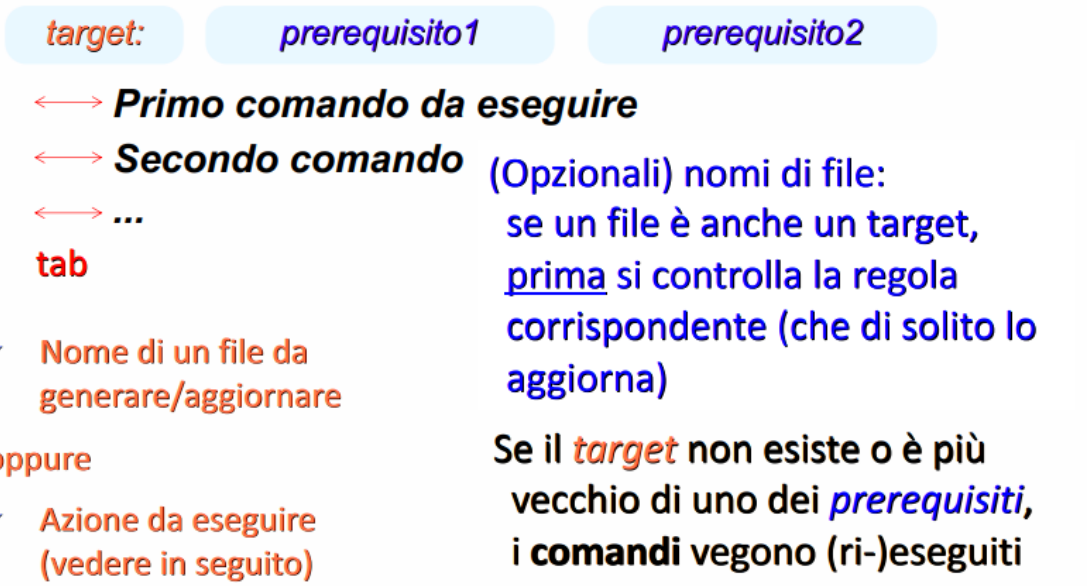
Esistono tool automatici per evitare di invocare il compilatore manualmente (operazione error-prone) che hanno una funzionalità di compilazione "embedded" in IDE.

Make: software che permette all'utente di costruire e installare software senza conoscerne i dettagli. Capisce in automatico quali sono i file da compilare rendendo, di conseguenza, auotomatico il processo di creazione di file che dipendendono da altri file.

GNU make:

1) Prende in input un file di testo chiamato Makefile che contiene un file di testo contenente un set di regole scritte usando un linguaggio speciale (make scripting language)
2) Prende in input un programma target
3) Segue le regole scritte in make file per ottenere il target desiderato

Regole



make [-f makefile_name] [target(s)]

- Se non specificata l'opzione -f make cerca in automatico nella directory corrente un file chiamato "Makefile"
- Se nessun target è specificato esegue le regole per il primo target specificato in Makefile

Vantaggi:

- Controllo approfondito di ciò che viene eseguito
- Permette di staccare programmi e compilatori da ambienti integrati

Dettagli del makefile:

- le linee di comando di una regola vengono passate alla shell esattamente come sono scritte (i commenti iniziano con #)
- Una linea troppo lunga può essere spezzata con un backslash '\' (senza blanck dopo)

Phony target:

- Comando che ripulisce da tutti i file oggetto⇒ **clean:**
  rm *.o
- Comando che ripulisce da tutti i file oggetto e l'eseguibile ⇒ **cleanall:**
  rm Web-list *.o
- È un target che non è il nome di un file da creare/aggiornare ma un'azione da eseguire
- I comandi associato sono eseguiti solo quando esplicitamente richiesto
- Non dovrebbe esistere alcun file con lo stesso nome di un phony target

Ogni macchina genera un eseguibile diverso; è pertanto una buona norma rimuovere tutti i file oggetto prima di condividere la cartella del progetto.

Definizione di una variabile nel makefile **variable_name = string**

Sostituzione di una variabile **$(variable_name)**

Make, tramite il suos et di regole implicite, permette di omettere i comandi per un file target name.o in quanto make andrà a cercare un file di nome name.c o name.cc; se lo trova invoca g++ -c sul file sorgente per generare/aggiornare name.o. Questo meccanismo consente di omettere la descrizione delle regole ovvie.

Grazie alle regole implicite, molte delle regole che necessitano di essere scritte in un makefile spesso si riducono a dire che un determinato file oggetto dipende da qualche header file; esistono modi perper generare le dipendenze in modo automatico.