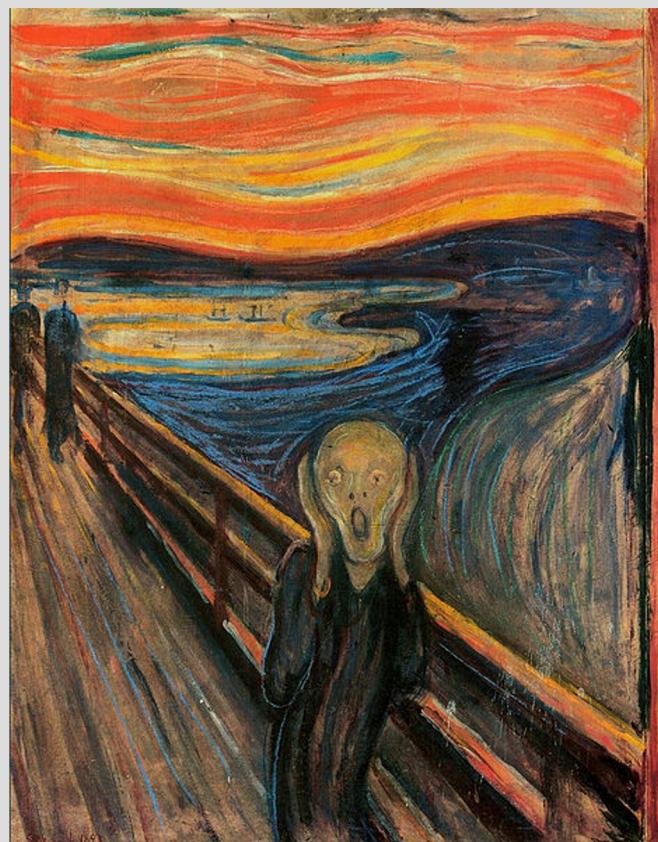


# Parte 8 – Sviluppo e testing

## Compilazione



[E. Munch – The Scream, 1893]

# Compilazione separata

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

INPUT	FASE	OUTPUT
<i>Source file</i>	$\rightarrow$ <i>Preprocessing</i>	$\rightarrow$ <i>translation unit</i>
<i>translation unit</i>	$\rightarrow$ <i>Translation</i>	$\rightarrow$ <i>object file</i>
<i>1 o più object file</i>	$\rightarrow$ <i>Linking</i>	$\rightarrow$ <i>executable (program)</i>

# Nota

In modo ambiguo, con il termine **compilazione** si indica:

- a volte la sola **generazione di un file oggetto** (preprocessing e traduzione)
- altre volte la **generazione di un eseguibile** (cioè tutte e tre le fasi)

# Pre-processing

- Come già sappiamo, dato un file sorgente, la prima passata è effettuata dal **preprocessore**
- Svolge **tre compiti**, controllati mediante le direttive inserite nel file sorgente:
  - Inclusione di file (**#include**)
  - Definizione/espansione di macro (**#define**)
  - Compilazione condizionale (**#ifdef**, **#if**)
- Il testo risultante dall'applicazione delle direttive è tipicamente chiamato **unità di traduzione** (**translation unit**), ed è questo che viene passato alla fase di traduzione

# Macro

**La direttiva per il preprocessore**

**#define**

permette di definire **macro** di due tipi

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

Le macro corrispondono a **sostituzioni testuali durante la fase di preprocessing**, cioè prima dell'inizio della compilazione vera e propria

# Macro senza argomenti

```
#define ID DEF
```

Dal punto nel codice in cui compare la direttiva **#define** in poi, si sostituisce la sequenza di caratteri **ID** con la sequenza di caratteri **DEF**

## ESEMPI

```
#define MAX 25
```

```
#define NAME "data.txt"
```

# Macro con argomenti (1)

## ESEMPIO

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

Dal punto nel codice in cui compare la **#define** in poi, si sostituisce la sequenza di caratteri **(a)++** alla sequenza **INC(a)**, dove a può essere a sua volta qualsiasi sequenza di caratteri

Perché mettere parentesi aggiuntive attorno all'argomento in **(a)++**?

# Macro con argomenti (2)

Perché senza parentesi una sostituzione lessicale può comportare risultati inattesi

**ESEMPIO** senza parentesi:

```
#define INC(a) a++  
main()  
{    int *p = new int;  
        INC(*p) ; // sostituito con *p++  
                  // che equivale a *(p++)  
                  // era quello che si voleva?  
}
```

# Macro con argomenti (3)

Si possono definire anche **macro con più di un argomento**

## ESEMPIO

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

- 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**

# Macro vuote

## ESEMPIO

`#define ID`

- Dal punto nel codice in cui compare la direttiva `#define` in poi, si elimina dal sorgente la sequenza di caratteri `ID`
- Anche una macro con argomenti si può definire come una **macro vuota**

# La direttiva #ifdef

La **direttiva al pre-processore** **#ifdef** controlla se un identificatore è stato definito come una macro (significato “if defined”)

## SINTASSI

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

Possono essere presenti un numero qualsiasi di direttive **#elif**

# Compilazione condizionale

Si tratta di una direttiva per la **compilazione condizionale**

- Gli identificatori presenti in **#ifdef** e **#elif** vengono valutati in sequenza
- Viene eseguito il codice che segue **#ifdef** o **#elif** del primo **identificatore** definito come macro, se esiste
- Se nessuno degli identificatori è definito, viene eseguito il codice che segue **#else**, se presente
- Si possono definire **parti alternative di codice da compilare** all'interno di un file sorgente
- Gli **#ifdef** possono essere **annidati** secondo le solite regole dell'if

# Utilizzo

```
#define UGO 0

int main() {
    cout << "Hello" << endl;
#ifndef UGO
    cout << "UGO defined" << endl;
#endif 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!

# Usi della compilazione condizionale (1)

*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
```

# Usi della compilazione condizionale (2)

Scrivere programmi **portabili** su diverse macchine o sistemi:

```
#ifdef __WIN64
#include <windows64.h> //codice per Windows
                         64 bit
#elif __linux__
#include <linux.h>
#elif __APPLE__
#include <apple.h>
#endif
```

Macro pre-definita dal compilatore

Compilatori diversi riconoscono versioni leggermente diverse del C/C++ (set di istruzioni diverso, estensioni del linguaggio...)

# Direttiva #ifndef

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

Se la macro **ID** *non è definita* si compila la sequenza di righe 1,  
altrimenti, se presente il ramo **else** (opzionale), la sequenza di righe 2

## ESEMPIO

```
#ifndef CLASS_H  
#define CLASS_H  
  
//blah blah blah  
  
#endif
```

# Esempio

Vedi programma `esempio_comp_condiz.cc`

1. Compilare il programma  
Cosa succede?
  
2. Togliere il commento alla definizione della  
macro **X** e ricompilare.  
Cosa succede? Perché?

# Programma su più file (1)

- Consideriamo un programma costituito da **più file sorgente**
- Come già visto, la funzione **main** deve essere definita in un solo file
- Inoltre, affinché l'output sia un unico programma i file sorgenti devono essere collegati tra loro:
  - il file contenente la funzione **main** deve usare almeno un identificatore (variabile, costante, funzione) di almeno un altro dei file sorgente che compone il programma;
  - Ogni altro file sorgente deve essere raggiungibile direttamente o indirettamente dalla funzione **main** secondo il criterio di cui sopra

# Programma su più file (2)

- Un file sorgente che non soddisfa nessuna delle precedenti proprietà non concorre a generare l'eseguibile
- Altra considerazione: dopo il **preprocessing** c'è una fase chiamata **traduzione** in cui ogni **unità di traduzione** è tradotta in isolamento
  - Non si potrà mai ottenere un **eseguibile completo** a partire da un singolo file
  - Il file con la funzione **main** non conterrebbe la definizione di almeno uno degli oggetti che usa
  - Nessuno degli altri file ha la funzione **main**

# File oggetto

- In effetti, il risultato della fase di traduzione non è un file eseguibile completo, ma un **file oggetto**
- Si tratta di un **file binario** (stesso formato di un eseguibile completo, programma in linguaggio macchina), ma è un **programma incompleto**, perché fa uso di oggetti che sono definiti altrove
- Nel file oggetto è indicato semplicemente che tali oggetti sono **definiti in un qualche altro file oggetto** (senza indicazione di quale sia questo file oggetto e dove si trovi)

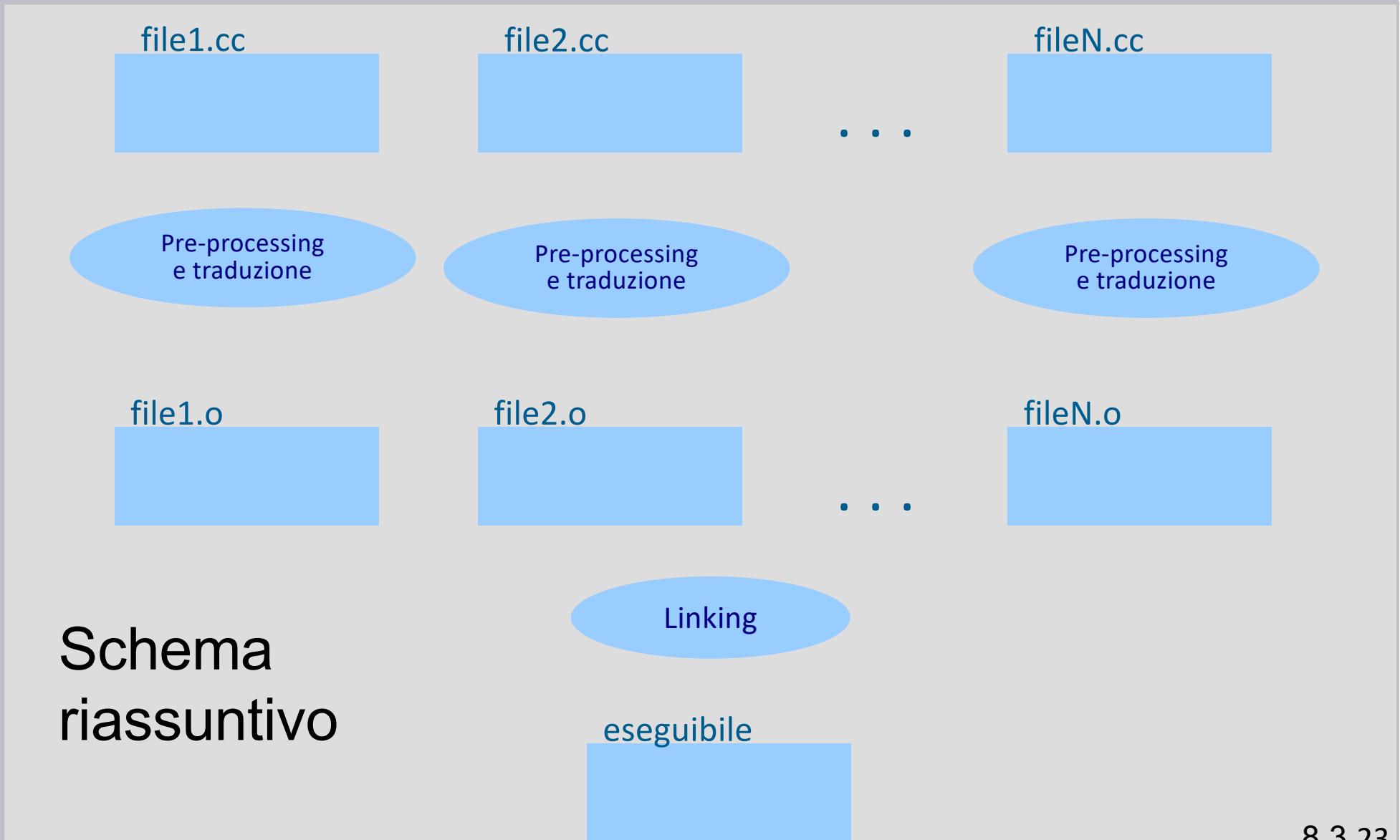
# Simboli

- Nel caso di **file oggetto** o **eseguibili** gli **identificatori** degli oggetti (variabili, costanti, funzioni) sono tipicamente chiamati **simboli**
- Quindi, dati i file oggetto ottenuti dalla **traduzione** delle unità di traduzione di un programma sviluppato su più file:
  - nell'unico file oggetto contenente il simbolo **main** c'è almeno un **simbolo** definito in uno degli altri file oggetto
  - negli altri file oggetto manca il **simbolo main**

# Collegamento

- Ecco perché si rende necessaria la fase di collegamento (**linking**) effettuata da un programma chiamato collegatore (**linker**)
- Il **linker** collega i file oggetto per formare un unico file eseguibile completo
- Inoltre, in ciascun punto di ciascun file oggetto in cui si utilizza un **simbolo** non definito nel file oggetto stesso, il **simbolo** viene sostituito da un **riferimento** al punto in cui è definito il **simbolo** in uno degli altri file oggetto
- Se il riferimento non esiste il linker si arresta e segnala l'errore al programmatore

# Compilazione



# Collegamento statico

- Quello che avviene per i simboli del nostro programma strutturato su più file è uno dei due tipi di collegamento possibile effettuati dal linker, ossia il **collegamento statico**
- L'alternativa è il **collegamento dinamico**
- Per capire bene le motivazioni di questo secondo tipo di collegamento e il modo in cui è realizzato, analizziamo più in dettaglio come si arriva da un sorgente ad un eseguibile

# Dal sorgente all'eseguibile

- Nelle slide seguenti vedremo come il compilatore **gcc** realizza le tre fasi per arrivare ad un eseguibile
- Ovvero analizzeremo la **compilazione separata**
- Considereremo prima il caso di programma costituito da un unico file sorgente, ad esempio *ciao\_mondo.cc*

# Componenti del GCC (1)

- Il compilatore **gcc** è costituito da un insieme di **moduli distinti**, ciascuno dedicato ad una delle **fasi della compilazione**
- Il comando **g++** è un **front-end** (un'interfaccia comune) per far invocare automaticamente tutte le componenti del **gcc** necessarie per compilare un programma scritto in linguaggio **C++**
- In modo simile, il comando **gcc** è solo un front-end per compilare programmi scritti in **C**

# Componenti del GCC (2)

Le 3 **componenti di base** sono dedicate ciascuna ad una delle fasi della compilazione:

**cc1plus**: pre-processore e traduttore in assembly per il C++ (cc1 per il C)

✓ Genera un file **assembly** a partire dal corrispondente file sorgente in C++ (C)

**as**: traduttore in linguaggio macchina

✓ Genera un file oggetto a partire da un file assembly

**collect2**: invocatore del linker

✓ Genera un file eseguibile collegando assieme (linker) tutti i file oggetto passati in ingresso

# Traduzione

- Come detto, nella fase di traduzione, si genera un **file oggetto** a partire dall'unità di traduzione  
*Come fermarsi alla sola fase di traduzione e generare solo un file oggetto?*
- Basta utilizzare l'opzione **-c**  
**g++ -c -Wall ciao\_mondo.cc**
- Il **file oggetto** prodotto ha tipicamente lo stesso nome del file sorgente, ma termina con il suffisso **.o**

In **Eclipse**:

L'opzione **-c** è generalmente settata di default

# Modalità verbose

- Si può osservare quello che succede durante la compilazione
- Basta eseguire il compilatore in **modalità verbose**
- Bisogna aggiungere l'opzione **-v**

**ESEMPIO** g++ -v -Wall ciao\_mondo.cc

- Compiliamo il programma ciao\_mondo.cc aggiungendo l'opzione –v

In **Eclipse**:

Projects > Property > C/C++ Build > Settings > Tool Settings:  
consente di gestire le componenti di base dell'intero processo  
di compilazione

> GCC C++ Compiler > Miscellaneous: è possibile selezionare  
l'opzione -v

# Proviamo la fase di traduzione...

- Compiliamo il programma sia con l'opzione **-c** che **-v**
- Notiamo il fatto che viene lanciato il compilatore (prima fase):
  - **cc1plus** in Linux
  - **clang** in Mac OSX
  - ...
- Controlliamo nel file system la presenza del file *ciao\_mondo.o*
- Se proviamo a visualizzarlo con un editor di testo otteniamo qualcosa di ben **poco leggibile**

# Input/output

- Noi abbiamo scritto programmi che, tra le altre operazioni, scrivevano su **stdout** (come fa *ciao\_mondo.cc*) e leggevano da **stdin**
- In particolare abbiamo usato operatori ed oggetti (**<<**, **>>**, **cout**, **cin**) che non abbiamo definito nel nostro file sorgente
- Dato il concetto di **compilazione separata**, dov'è la **definizione** e la **dichiarazione** di questi oggetti?

# Input/output

- In effetti, i nostri programmi non si sarebbero compilati correttamente se non avessimo aggiunto la direttiva  
`#include <iostream>`
- Come sappiamo tale direttiva inserisce il contenuto del file **iostream** all'interno della unità di traduzione derivante dal file sorgente *ciao\_mondo.cc*

# L'operatore di uscita

Trascurando i dettagli, l'operatore di uscita << è una funzione, la cui dichiarazione è fatta più o meno così:

```
.... operator<<(basic_ostream<...>&,  
tipo Oggetto_da_stampare) ;
```

con due argomenti:

- un oggetto di tipo **basic\_ostream (cout)**
- un oggetto che può essere di tipo **char, int, char []..** (oggetto da stampare)

# iostream

- L'operatore di uscita manda su **cout** una sequenza di caratteri che dipende dal valore e dal tipo del secondo parametro
- Potremmo aspettarci di trovare in **iostream** la **definizione** di **operator<<**
- Per vedere se le cose stanno così, dobbiamo prima scoprire dov'è il **file iostream**
- Le parentesi angolate nella direttiva  
**#include <iostream>**  
significano “cerca il file nelle **directory predefinite**”
- Per scoprire quali sono queste **directory predefinite** possiamo eseguire la traduzione in modalità verbose

## ESEMPIO

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

# Search list

Nell'output si trovano righe simili alle seguenti:

```
#include <...> search starts here:  
/usr/lib/gcc/i486-linux-  
gnu/4.1.2/.../.../.../include/c++/4.1.2  
/usr/lib/gcc/i486-linux-  
gnu/4.1.2/.../.../.../include/c++/4.1.2/i486-linux-gnu  
/usr/lib/gcc/i486-linux-  
gnu/4.1.2/.../.../.../include/c++/4.1.2/backward  
/usr/local/include  
/usr/lib/gcc/i486-linux-gnu/4.1.2/include  
/usr/include  
End of search list.
```

Questa è la lista delle directory predefinite in cui il compilatore cerca i file da includere

# Contenuto di iostream

- Controlliamo se il file **iostream** è presente in qualcuna di queste **directory**
- Ad esempio in Linux eseguiamo il comando **find /usr/include/c++ -name iostream**
- Oppure usiamo l'interfaccia del ns. Sistema Operativo per cercare il file
- Analizziamo il contenuto del file usando un editor
- Di **operator<<** non dovrebbe esserci traccia
- Ci accorgiamo però che in **iostream** c'è, tra le altre, la direttiva: **#include <iostream>**

# Contenuto di ostream

- Cerchiamo quindi il file **ostream**
- Apriamolo con un editor
- In questo file troveremo svariate dichiarazioni di **operator<<**
- Una dichiarazione per ogni tipo del secondo argomento
- Alcune sono solo **dichiarazioni**

**Dov'è il codice delle funzioni di iostream e  
in generale  
di tutte le funzioni presenti nelle librerie?**

# Libreria standard del C++

Il codice si trova nella libreria standard del C++

- Ad esempio **GLIBCXX\_3.4** ovvero **Libreria (LIB) Standard del C++ (CXX)** realizzata dalla **GNU (G)**, **versione 3.4**

Che cos'è?

- Un **file oggetto di libreria** contenente una raccolta di **definizioni** di strutture dati e funzioni
- Strutture dati quali, ad esempio, **cout** e **cin**
- Funzioni quali, ad esempio, gli **operatori di uscita e di ingresso**

# Libreria standard del C++ (2)

Dove si trova questo file?

- C'è la possibilità di risalire alle **librerie condivise** (chiariremo in seguito) da cui dipende un eseguibile
  - Il comando **ldd** in UNIX: **ldd -v a.out**
  - Il comando **otool** in MAC OSX: **otool -L a**
  - Utility tool come Dependency Walker (<http://dependencywalker.com/>) per Windows

In output dovrebbe comparire qualcosa del tipo:

```
./a.out:  
libstdc++.so.6 (GLIBCXX_3.4) => /usr/lib/i386-  
linux-gnu/libstdc++.so.6  
...
```

# Indicazione delle librerie

- E' il **linker** che collega i simboli non definiti presenti nei **file oggetto** alla loro definizione all'interno della libreria opportuna
- *Come fa il linker a sapere quali librerie collegare e dove cercarle?*
- E' necessario che queste informazioni gli vengano comunicate esplicitamente
- Il **front-end g++** è già configurato per invocare il **linker** passandogli una **serie di librerie standard**

# Collegamento

- Proviamo a invocare il **linker** del **gcc** passando semplicemente il file oggetto come argomento al comando **g++**
- Dal suffisso **.o** il **front-end** capisce che deve solo invocare il **linker**

## ESEMPIO

```
g++ ciao_mondo.o
```

# Indicazione delle librerie (2)

Aggiungiamo l'opzione `-v` al linker:

`g++ -v ciao_mondo.o`

In **Eclipse**:

Projects > Property > C/C++ Build > Settings > Tool Settings: consente di gestire le componenti di base dell'intero processo di compilazione

> ... C++ Linker > Command: aggiungere l'opzione `-v` al comando `g++`

Ad esempio, in Linux, nell'output vedremo che il comando **collect2** è invocando passandogli:

- Le **directory** in cui sono contenute le **librerie** mediante l'opzione `-L`
- Le **librerie** con cui collegare il file oggetto mediante l'opzione `-l`
- Si può verificare che, mediante l'opzione `-lstdc++`, è indicata anche la **libstdc++** (il prefisso **lib** è aggiunto automaticamente)

# Riassumendo

- I file sorgenti sono tradotti in isolamento
- 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...

# Collegamento dinamico

- Abbiamo visto che il **linker** non unisce il file di libreria standard al nostro file oggetto per creare un eseguibile contenente le definizioni necessarie
- Semplicemente mette nell'eseguibile tutte le **informazioni per rintracciare nelle librerie i simboli non definiti** nel file oggetto
- Questo *tipo di collegamento si chiama **collegamento dinamico***
- Quando nell'eseguibile si utilizza uno di questi simboli, un apposito meccanismo di supporto **run-time** va ad accedere al simbolo nella libreria

# Collegamento statico forzato

- È possibile forzare il **linker** ad effettuare il **collegamento statico** alle librerie
- Basta aggiungere l'opzione **-static**

## ESEMPIO

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

oppure

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

# Eseguibile dinamico e statico

- L'eseguibile prodotto mediante collegamento dinamico alle librerie è tipicamente chiamato **esegueibile dinamico**
- All'opposto, definiamo **esegueibile statico** un eseguibile prodotto per mezzo di un collegamento statico alle librerie
- **Vantaggi e svantaggi?**

# Vantaggio eseguibile dinamico

Il collegamento dinamico alle librerie ha i seguenti **vantaggi** rispetto a quello statico:

- Un eseguibile dinamico ha **dimensioni minori**
- **Non si spreca spazio** replicando le entità contenute nelle librerie in tutti i file eseguibili
- Le librerie collegabili dinamicamente si chiamano infatti **librerie condivise (shared libraries)**
- In ambiente **Microsoft Windows** sono invece chiamate **Librerie a collegamento dinamico (Dynamic Link Library o DLL)**

# Vantaggio eseguibile statico

- Un **eseguibile collegato dinamicamente** ad una data libreria condivisa potrà essere eseguito **solo su sistemi che contengono tale libreria**
  - Si crea una **dipendenza** tra gli eseguibili e le librerie stesse
  - Al contrario un **eseguibile statico** dovrebbe poter essere eseguito senza problemi su tutti i sistemi che ne supportano semplicemente il formato
  - Il formato di un file eseguibile si può controllare passando il nome del file come argomento al comando **file** (es: **file nome\_file**)

# Esercizio

- Compilare *ciao\_mondo.cc* con e senza l'opzione **-static**
- Controllare la differenza di **dimensioni** del file eseguibile nei due casi, utilizzando ad esempio il comando **ls -lh**
- Controllare la **dipendenza dalle librerie condivise** nei due casi col comando **ldd** o equivalente in altri sistemi operativi
- Nel caso di compilazione statica non dovrebbe esservi nessuna dipendenza

# Sorgenti multipli e librerie (1)

- Proviamo a mettere insieme quanto visto finora sul **collegamento**

- Supponiamo di compilare un programma sviluppato su più file con il comando

**g++ file1.cc file2.cc ... fileN.cc**

- Che è equivalente ai due comandi:

**g++ -c file1.cc file2.cc ... fileN.cc**

**g++ file1.o file2.o ... fileN.o**

- La seconda soluzione è chiamata

**Compilazione separata**

- Come già detto l'ordine con cui sono riportati i file sorgente o oggetto non conta

# Sorgenti multipli e librerie (2)

Il linker:

- collega prima **staticamente** gli N file **oggetto** creando un **unico file oggetto temporaneo**
- quindi, siccome non si è specificata l'opzione **-static**, **collega dinamicamente** tale file temporaneo alle librerie, ottenendo il file eseguibile

# Regole di (ri)compilazione

Quando abbiamo bisogno di **aggiornare un file eseguibile?**

- Se qualcuno dei file oggetto cambia

Quando abbiamo bisogno di **aggiornare un file oggetto?**

- Se il corrispondente file sorgente cambia
- Se un qualunque file incluso cambia

# Ricompilazione selettiva

Se è cambiato uno solo o pochi dei file sorgenti non ha senso ritradurre anche tutti gli altri

Basta ritradurre solo i file sorgente cambiati e ricollegarli agli altri file oggetto

## ESEMPIO

Se è cambiato solo file1.cc, oppure è cambiato un file incluso solo da file1.cc, e si dispone dei file oggetto corrispondenti a tutti gli altri file, si può invocare solo

**g++ -c file1.cc**

**g++ file1.o file2.o ... fileN.o**

# Velocità di compilazione (1)

- L'operazione di collegamento di N file oggetto è estremamente più **semplice e veloce** della traduzione dei corrispondenti file sorgente
- Da questo scaturisce il **secondo importante vantaggio** dello sviluppo di un programma su più file
- Grazie alla **compilazione separata**, la **ricompilazione** in seguito a cambiamenti nei sorgenti è più **rapida**

# Velocità di compilazione (2)

Per farci un'idea vediamo un esempio di **tempi di compilazione** di un **programma di medie dimensioni**

Kernel del sistema operativo Linux 2.6.21 su una macchina virtuale (vmware) su processore Pentium M a 1.5 GHz

**Tempi:**

- Con tutti i file oggetto già pronti (quasi solo collegamento): 15 sec
- Con tutti i file oggetto da tradurre: 14 minuti

**Il rapporto tra i tempi è maggiore di 50 !**

# Collegamento altri linguaggi

- Un altro **vantaggio** della compilazione separata è la **semplicità** di implementazione di programmi costituiti da file sorgente scritti con **linguaggi diversi** l'uno dall'altro
- Ciascuno file sorgente viene **tradotto separatamente** mediante il **compilatore opportuno** per il linguaggio con cui è scritto
- I **file oggetto prodotti sono collegati assieme** dal linker

# Entità globali

- Se in un file sorgente si vuole riferire una entità definita in un file sorgente scritto in un linguaggio diverso si possono avere **problemi**
- Al **variare del linguaggio**, può variare:
  - il simbolo con cui uno stesso identificatore, dichiarato nei file sorgenti, è tradotto all'interno dei corrispondenti file oggetto (i compilatori manipolano gli identificatori in modi diversi)
  - il modo in cui sono inizializzati i valori dei parametri formali ed il modo in cui sono ritornati i valori a livello di linguaggio macchina

# Specifiche di collegamento

- Nel linguaggio C++ si può informare il compilatore circa il linguaggio con cui è stato manipolato un certo identificatore esterno
- Si fa precedere la specifica  
`extern "nome_linguaggio"`  
alla dichiarazione dell'identificatore
- In pratica è una **estensione della dichiarazione extern**

# Esempi

- Per utilizzare in un file sorgente **C++** una funzione **fun** (che prende in ingresso e ritorna un intero) definita in un file sorgente **C** (tradotto utilizzando un compilatore **C**) si utilizza la dichiarazione:

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

- Si possono raggruppare più dichiarazioni tra parentesi graffe:

```
extern "C" {int fun(int) ;  
                      void f2(char) ; }
```

# Condivisione funzioni C++

- Dualmente, le **specifiche di collegamento** permettono a file oggetto risultanti dalla traduzione di file sorgente scritti in altri linguaggi di **accedere a funzioni scritte in un file sorgente C++**
- Affinché una funzione **fun definita in C++ sia condivisibile con file oggetto ottenuti dalla traduzione di file sorgente C**, basta definire (nel sorgente C++) la funzione **fun** come:

```
extern "C" int fun()  
{ . . . }
```

# Collegamento altri linguaggi

***Le specifiche di conversione disponibili dipendono dal compilatore!***

In **C++** è garantita la disponibilità in tutti i compilatori della sola specifica "C"

# Compilazione automatica

- Invocare il compilatore manualmente per un programma organizzato su più (molti) file è un'operazione lunga ed error-prone

Esistono **tool automatici**

- **Funzionalità di compilazione “embedded”** in IDE (Integrated Development Environment)
- **Make**

«Executing a set of commands according to a set of rules»

*<http://www.gnu.org/software/make/>*

# GNU make

- Consente all'utente finale di costruire e installare software senza conoscere i dettagli
- Automatizza il processo di creazione di file che dipendono da altri file
- Esplicita e risolve le dipendenze tra file, individuando l'ordine di creazione dei file e invocando comandi esterni per eseguire le operazioni necessarie
- Si basa sulla data e ora di ultima modifica dei file interessati per effettuare la ricompilazione selettiva
- Può essere usato per compilare pacchetti software scritti in altri linguaggi di programmazione come Java

# GNU make

**GNU make** - programma che:

1) prende in input

- un file di testo – il **Makefile** – contenente un **set di regole** scritte usando un **minilinguaggio speciale (make scripting language)**
  - un programma **target**
- 2) segue le regole per ottenere il target desiderato
- Altre regole intermedie possono essere prese in considerazione

# Regole

## Regole (versione semplificata)

*target:*

*prerequisito1*

*prerequisito2*

↔ **Primo comando da eseguire**

↔ **Secondo comando**

↔ ...

**tab**

- ✓ Nome di un file da generare/aggiornare

oppure

- ✓ Azione da eseguire (vedere in seguito)

(Opzionali) nomi di file:  
se un file è anche un target,  
prima si controlla la regola  
corrispondente (che di solito lo  
aggiorna)

Se il *target* non esiste o è più  
vecchio di uno dei *prerequisiti*,  
i **comandi** vengono (ri-)eseguiti

# Esempio

**Vedi file** *progetto\_multifile\_tipo/Makefile1*

- Esempio di Makefile per compilare il nostro programma di gestione delle liste
- **Web-list** è il nome del nostro eseguibile completo

*Cercate di interpretarlo ricordando le regole di (ri)compilazione di slide 52...*

# Invocare il make

**make [-f makefile\_name] [target(s)]**

- Se l'**opzione -f** non è usata, make assume che le regole siano immagazzinate in un file residente nella directory corrente col nome di **GNUmakefile**, **makefile**, o **Makefile**

Nome migliore: **Makefile**

- Make esegue le regole corrispondenti al/i *target*
- Se nessun **target** è specificato, esegue le regole per il *primo target* specificato in **makefile\_name (default target)**

# Esercizio

- Provate a invocare il comando **make** nella directory *progetto\_multifile\_tipo/* per eseguire il file Makefile1

*Attenzione a invocare il file corretto!!*

*Osserviamo l'output*

- Provate a invocarlo di nuovo
- Provate a rimuovere l'eseguibile **Web-list** ed eseguirlo di nuovo

*Proviamo a lanciare l'eseguibile*

# IDE vs. Make

## Integrated Development Environment (IDE)

Ad esempio: *Dev-C++, Code::Blocks, Eclipse, Visual studio, ...*

- Semplificano la vita
- Rendono trasparente la compilazione di programmi complessi (**progetti**)

## Make

- Comprensione e controllo approfonditi  
Gli IDE spesso generano il Makefile!!
- Permette di staccare programmi e compilatori da ambienti integrati (IDE)
- Un buon esperto IT dovrebbe padroneggiare il processo di realizzazione di programmi complessi

# Dettagli del makefile (1)

- Le **linee di comando** di una regola vengono passate alla **shell** esattamente come sono scritte (*verbatim*)
- Le righe di commento in un makefile iniziano con '#'
- E' compito del programmatore inserire correttamente i comandi in una regola

# Dettagli del makefile (2)

Un linea troppo lunga può essere spezzata  
usando un backslash '\'

Non aggiungere nessun blank dopo '\'

## ESEMPIO

`g++ -o GoTA environment.o state_handling.o GoTA_main.o`

*è equivalente a*

`g++ -o GoTA environment.o state_handling.o \  
GoTA_main.o`

# Phony target (1)

Come si fa a ripulire la directory dai file temporanei (es. file oggetto) dopo la compilazione automatica?

**clean:**

```
rm *.o
```

**cleanall:**

```
rm Web-list *.o
```

Questo è un esempio di utilizzo di **phony target**

**Phony target**

Target che non è il nome di un file da creare/aggiornare  
ma un'**azione (comando) da eseguire**

# Phony target (2)

- I comandi associati a phony target sono **eseguiti** solo quando esplicitamente richiesto (ogni volta che si passa il phony target come parametro al make)

## ESEMPIO make clean

- Non dovrebbe esistere alcun file con lo stesso nome di un phony target
- Per evitare problemi, possiamo **esplicitamente dichiarare uno o più target come phony**:  
**.PHONY: target1 target2 ...**

# Esercizio

- Aggiungiamo i phony target **clean** e **cleanall** al nostro **Makefile**

## SOLUZIONE

Vedi file *progetto\_multifile\_tipo/Makefile2*

- Proviamo ad invocarli
- Per prevenire messaggi di errore del comando **rm** in assenza dei file specificati come argomento, è possibile utilizzare l'opzione **-f**  
**rm -f \*.o**

# Variabili (1)

- I makefile sono *duplication-prone*

**Web-list: main.o fun-app.o liste.o tipo.o**

**g++ -o Web-list main.o fun-app.o liste.o tipo.o**

Questo è fastidioso ed error-prone

- Possiamo eliminare il rischio e semplificare il **Makefile**
- Le **variabili** consentono di **definire una stringa di testo** una volta e di usarla in diversi punti successivamente

# Variabili (2)

- Definizione di variabile

**variable\_name = string**

- Sostituzione di variabile

**\$(variable\_name)**

## ESEMPIO

```
OBJ = state_handling.o environment.o GoTA_main.o  
GoTA: $(OBJ)  
g++ -o GoTA $(OBJ)
```

# Esercizio

Usare le variabili per elencare i file oggetto  
del primo target nel Makefile

## SOLUZIONE

Vedi file *progetto\_multifile/Makefile3*

# Regole implicite (1)

- **make** mette a disposizione un **set di regole implicite**
- Una **regola implicita** comunemente usata è la seguente:

«Se i comandi per un file target **name.o** sono omessi, allora **make**:

  - Cerca un file di nome **name.c** o **name.cc**
  - Se lo trova, invoca **gcc/g++ -c** sul file sorgente per generare/aggiornare **name.o** (soltanto l'oggetto, **opzione -c**)»

# Regole implicite (2)

Questo meccanismo consente di omettere la descrizione delle regole “ovvie”

- ✓ Non c'è bisogno di indicare né i *comandi ovvi* né i *prerequisiti .c o .cc* ovvi nelle regole

Versione del **Makefile** con regole implicite:

**Vedi file *progetto\_multifile/Makefile4***

# Modifica regole implicite

- I comandi implicitamente invocati (e le relative opzioni) sono definiti da variabili di Make.
- Essi possono essere modificati cambiando il valore delle corrispondenti variabili.
- In questo corso non approfondiremo questo aspetto.
- Dettagli sono disponibili all'URL:  
[https://www.gnu.org/software/make/manual/html\\_node/Implicit-Rules.html](https://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html)

# Generazione dei prerequisiti (1)

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**

- Meglio della versione completa, ma ancora noioso ed error-prone
- Esistono diversi modi per **generare le dipendenze in modo automatico**
- Vediamo uno di questi metodi

# Generazione dei prerequisiti (2)

Se invocato con l'opzione **-M**, il compilatore **g++** genera le dipendenze dei file sorgenti passati come argomenti

- Provate ... (per omettere file di sistema, usare **-MM**)
- Possiamo ridirigere l'output su un file (**>**)
- Possiamo *includere* questo file nel **Makefile**

# Includere file di testo

- E' possibile includere altri file di testo in un **makefile** usando le direttive **include** nel **makefile** stesso
- Stesso comportamento della direttiva **C/C++ #include**

Sintassi: **include file1 file2 ...**

- Per far sì che make ignori i file di testo da includere nel caso non esistano, senza dare messaggi di errore, si può usare la direttiva **-include**

Sintassi: **-include file1 file2 ...**

# Test

- Rimuovere tutte le regole di dipendenza tranne quelle del **target Web-list** (con relativi comandi)
- Invocare **make**
- Dovrebbe compilarsi correttamente grazie alle regole implicite
- Modificare ora un file sorgente **.cc** e invocare di nuovo **make**
- Modificare un file header **.h** e invocare **make**
- Il programma si è ricompilato correttamente in entrambi i casi?

# Esercizio

Modificare il **Makefile** in modo che:

- Fornisca un phony target per la **generazione di un file delle dipendenze** attraverso l'opzione **-M** (o **-MM**) per il compilatore
- Usi il file delle dipendenze generato per **compilare il programma** (attenzione a cosa succede se il file non esiste!)

## SOLUZIONE

Vedi file *progetto\_multifile/Makefile5*

# Opzione utile

Per la fase di traduzione meglio aggiungere l'opzione **-Wall!!**

- Come aggiungere questa opzione nei **Makefile**?
- Dobbiamo usare la variabile per specificare i flag del comando
- La variabile per g++ è **CXXFLAGS**

**ESEMPIO CXXFLAGS=-Wall**

**Vedi file** *progetto\_multifile/Makefile6*

# Variabili automatiche

Speciali variabili espanso nella lista di comandi (non nel target o nei prerequisiti)

- **\$@** → il target della regola
- **\$<** → il primo prerequisito
- **\$?** → i prerequisiti più nuovi del target
- **\$^** → tutti i prerequisiti

## ESEMPIO

**outputfile: file1.o file2.o file3.o file4.o**

**g++ -o \$@ \$^**

# Note finali sui makefile

- E' possibile usare **istruzioni condizionali** per decidere se usare o ignorare alcune parti dei **makefile**
- Si possono usare caratteri speciali (**wildcard**)
- E' possibile specificare più di un target in una singola regola
- E' possibile specificare 'search path' per i prerequisiti usando la variabile **VPATH** o la direttiva **vpath**
- Tante altre cose....

Per approfondimenti, consultare il **manuale GNU Make**:

<https://www.gnu.org/software/make/manual/>