Università di Napoli Federico II - Facoltà di Ingegneria - Corso di Laurea in Ingegneria Informatica

Corso di Programmazione I

Programmazione Modulare:
Meccanismi e Strumenti a supporto in C/C++

Modularizzazione in C/C++

- L'uso disciplinato di alcuni meccanismi del linguaggio C/C++ consente una corretta strutturazione di un programma in moduli.
- Tra i principali meccanismi vi sono:
  - la compilazione separata,
  - l'inclusione testuale,
  - le dichiarazioni *extern,*
  - l'uso dei prototipi di funzioni.

Specifica e implementazione (1/3)

- E' buona norma tenere separata la *specifica* di un modulo dalla sua *implementazione*.
- Un programma utente di un modulo A deve conoscerne la specifica, ma disinteressarsi dei dettagli della sua implementazione.
- Ciò può essere realizzato scrivendo un file di intestazione *o header file* (con l'estensione ***.h***) contenente le dichiarazioni che costituiscono l'**interfaccia** di A, ed un file separato per l'implementazione di A.
- Siccome ogni modulo deve essere *autoconsistente*, ovvero deve contenere tutte le informazioni necessarie per la compilazione, l'*header file* deve essere incluso (mediante la direttiva al preprocessore #include) nella implementazione di ogni modulo utente

Main. C

```
// Utilizzatore del modulo A
#include "A.h"
```

A.h

```
// Interfaccia di A
```

A.cpp

```
// Implementazione del modulo A
#include "A.h"
```

Specifica e implementazione (2/3)

- Esempio:
  - Un programma C++ consiste di più file sorgente che sono individualmente compilati in file oggetto
  - Questi sono poi collegati insieme per produrre la forma eseguibile del programma

```
// File: func.h:
int somma(int, int);
int prodotto(int, int);
```

```
// File: somma.cpp:

int somma(int a, int b)
 {
  return a+b;
 }
```

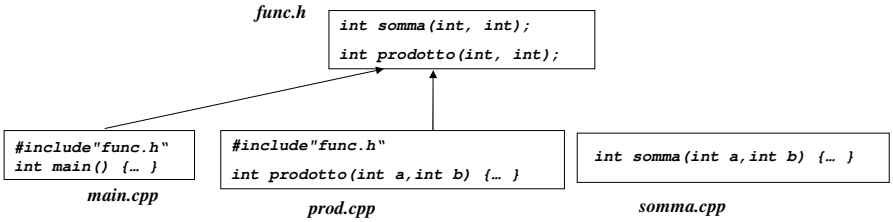
```
// File: prod.cpp:

#include "func.h"

int prodotto(int a, int b) {
  int prod=0;
  for (int i=b; i>=1; i--) {
    prod=somma(prod,a);
  }
  return prod;
}
```

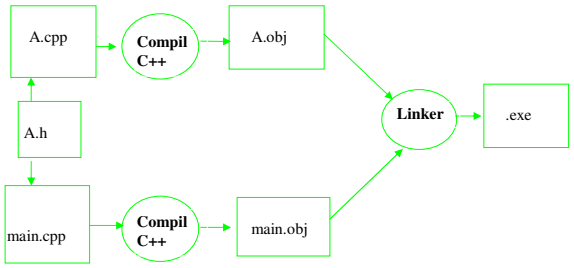
```
// File: main.cpp:
#include "func.h"
#include <iostream.h>  // pre-compilatore
#include <stdlib.h>    // pre-compilatore

int main(){
  int m,n;
  cout << "inserire due numeri interi positivi:";
  cin >> m >> n;
  cout << "somma:" << somma(m,n);
  cout << "prodotto:" << prodotto(m,n);
  system("PAUSE");
  return 0;
}
```



Specifica e implementazione (3/3)

- Si osservi che, fintanto che l'interfaccia resta inalterata, l'implementazione può essere modificata senza dover ricompilare il modulo utente (ma naturalmente occorre ricollegare i moduli oggetto).



- La specifica, contenuta nel file di intestazione, può essere riguardata come una sorta di *contratto* sottoscritto tra l'implementatore e l'utente.
- Quando più programmatori lavorano simultaneamente ad un progetto di grandi dimensioni, una volta accordatisi sulla specifica dei vari moduli, possono procedere all'implementazione dei rispettivi moduli indipendentemente l'uno dagli altri.

Librerie di moduli software

- Queste tecniche di sviluppo modulare consentono lo sviluppo su base professionale di librerie di moduli software.
- Il produttore di una libreria distribuisce:
  - i file di intestazione (che devono essere inclusi dall'utilizzatore nel codice sorgente) dei moduli che fanno parte della libreria;
  - i moduli di libreria in formato oggetto (già compilati), che l'utilizzatore deve collegare assieme ai propri moduli oggetto.
- Tale scelta è tipicamente motivata da esigenze di tutela della proprietà, ed inoltre evita di dover ricompilare i moduli di libreria.

Preprocessore C

- Il **preprocessore** e' un programma che viene attivato dal compilatore nella fase precedente alla compilazione, detta di **precompilazione**.
- Il **preprocessore** legge un sorgente C e produce in output un altro sorgente C, dopo avere sostituito i commenti con spazi bianchi, unito le linee che terminano con '\', espanso in linea le macro, incluso i file e valutato le compilazioni condizionali o eseguito altre direttive.
- Una direttiva inizia sempre con il carattere '#' eventualmente preceduto e/o seguito da spazi.
- I token seguenti '#' definiscono la direttiva ed il suo comportamento.
- Una direttiva al preprocessore puo' comparire in qualsiasi punto del sorgente in compilazione ed il suo effetto permane fino alla fine del file.

Macro

- **Definire una macro** significa *associare un frammento di codice ad un identificatore*.
- Ogni volta che il preprocessore C incontra l'identificatore cosi' definito, esegue la sua sostituzione in linea con il frammento di codice ad esso associato.
- La definizione delle macro avviene per mezzo della direttiva #define

```
#define MAX 100
#define STRING_ERR "Rilevato errore !\n"
```

- Il preprocessore elabora l'input in maniera sequenziale:

```
foo = X;
 #define X  4
bar = X;
```

```
foo = X;
bar = 4;
```

Macro

- Le macro possono essere definite anche in forma paramentrica; in tal caso la sostituzione dei parametri *formali* con quelli *attuali* avviene in modo testuale durante la fase di espansione della macro.

```
#define min(X,Y)  ((X) < (Y) ? (X) : (Y))
```

```
x = min(a,b);
```

```
x = ((a) < (b) ? (a) : (b));
```

Inclusione di file

- Il preprocessore C, tramite la direttiva **#include**, puo' ricercare il file indicato in alcune directory standard o definite al momento della compilazione ed espanderlo testualmente in sostituzione della direttiva.
- La direttiva #include puo' essere impiegata in due forme:

#include <nomefile>
#include "nomefile"

- Nel 1° caso il ***nomefile*** viene ricercato in un insieme di directory standard definite dall'implementazione ed in altre che sono specificate al momento della compilazione.
- Nel 2° caso il ***nomefile*** viene ricercato nella directory corrente e poi, se non e' stato trovato, la ricerca continua nelle directory standard e in quelle specificate al momento della compilazione come nel 1° caso.
- **N.B. -** Nel caso che un header venga modificato, e' necessario ricompilare tutti i sorgenti che lo includono.

Compilazione Condizionale

- Il preprocessore C può testare espressioni aritmetiche o controllare se un nome è stato definito come macro e decidere di includere o escludere parti del codice sorgente dalla compilazione.

- Le direttive:

  #if #ifdef #ifndef #elif #else #endif

  consentono di associare la compilazione di alcune parti di codice alla valutazione di alcune condizioni **in fase di compilazione**

Compilazione Condizionale: Esempio 1

```
#if <espressione_costante>
<statement_1>
#else
<statement_2>
#endif
```

- Se l'espressione costante specificata, valutata in compilazione, ritorna TRUE, allora verranno compilati gli statement_1, altrimenti verranno compilati gli statement_2
- Esempi di utilizzo
  - Il programma può richiedere codice diverso su sistemi operativi diversi
  - Vogliamo compilare una versione di "debug" del programma

Compilazione Condizionale: Esempio 2

- Queste direttive possono essere utilizzate per ***impedire che uno stesso header file venga incluso più di una volta*** nello stesso file sorgente
  - Ciò può causare errori (definizione multipla) e aumenta il tempo di compilazione

```
#ifndef _NOMEHEADER_H
#define _NOMEHEADER_H
    Corpo dell'header
#endif
```

#ifndef considera superata la condizione se non e' definito l'identificatore.

La direttiva #endif chiude la #if, #ifdef, #ifndef

Compilazione Condizionale: Esempio 2

- In questo modo il primo file che presenta la direttiva #include “header.h” valuta l’espressione #ifndef, trova che _HEADER_H non è stato definito, lo definisce e include il codice compreso tra #ifndef e #endif
- I files successivi all’atto dell’inclusione valutano l’espressione, trovano che _HEADER_H è stato già definito, e saltano alla riga successiva (#endif) senza includere il file header.h

```
#ifndef _NOMEHEADER_H  // sta per #if !define _NOMEHEADER_H
#define _NOMEHEADER_H
  Corpo dell’header
#endif
```

L'utility make

- Il codice di un programma di medie dimensioni è tipicamente costituito da più moduli (file sorgenti e file header)
- Ogni sorgente modificato deve essere ricompilato
- Se un file header viene modificato, ogni sorgente che lo include deve essere ricompilato
- Si deve ripetere la fase di link per collegare i diversi file oggetto e produrre il nuovo file eseguibile
- L'utility ***make*** automaticamente determina quali parti del programma devono essere ricompilate ed esegue i comandi per ricompilarle
- **make** può essere usato con ogni linguaggio di programmazione il cui compilatore può essere invocato con un comando di shell
- **make** può anche essere usato ogni volta che alcuni file devono essere aggiornati automaticamente a partire da altri quando questi ultimi cambiano

Il Makefile

- make ha bisogno di un file (chiamato makefile o Makefile) per sapere cosa fare
- Un makefile è costituito da una serie di **regole** :

```
target ...  :  prerequisiti ...
        comando
        ...
```

- Un *target* è il nome di **un file** che deve essere generato (es. file oggetto o eseguibile) oppure un nome che identifica **un'azione da compiere** (in questo caso prende il nome di *phony target*)
- Un *prerequisito* è un file usato come input per creare il target
- Un *comando* è un'azione che make esegue

Il Makefile

- *Tipicamente, un comando serve a creare il file target se uno dei prerequisiti cambia*
- Una regola può avere più comandi, uno per riga oppure sulla stessa riga e separati da un ';'
- **NOTA**: occorre inserire un carattere 'Tab' all'inizio di ogni riga che contiene comandi
- I comandi possono anche essere inseriti sulla riga dei prerequisiti, purchè ci sia un ';' tra i prerequisiti e i comandi
- Una linea lunga può essere spezzata in più linee utilizzando un backslash '\' alla fine di ogni riga
- Una regola può anche non avere prerequisiti
- Il carattere '#' in una linea inizia un commento

Un semplice esempio

- Quando make incontra un target, controlla se esiste un file avente lo stesso nome
- In caso negativo si tratta di un phony target, quindi i comandi associati vanno **in ogni caso** eseguiti
- Nota che i phony target possono avere prerequisiti
- Esempio: abbiamo un file sorgente main.cpp, un header file defs.h e vogliamo creare l'eseguibile myprog

Un semplice esempio (ok)

- Usiamo il seguente makefile:

```
myprog.exe :  defs.h
    g++ -o myprog.exe main.cpp
```

- La prima volta che eseguiamo make, non esiste un file di nome myprog.exe e quindi myprog.exe è un phony target
- Il comando 'g++ -o myprog.exe main.cpp' viene eseguito e produce il file myprog.exe
- Se eseguiamo di nuovo make, viene trovato il file myprog.exe. Il prerequisito defs.h non è cambiato quindi non occorre aggiornare il file myprog.exe. Risposta:

```
make: `myprog.exe' is up to date.
```

- Questo è il comportamento corretto

Un semplice esempio (ko)

- Usiamo il seguente makefile:

```
all :  defs.h
    g++ -o myprog.exe main.cpp
```

- La prima volta che eseguiamo make, non esiste un file di nome all e quindi all è un phony target
- Il comando 'g++ -o myprog.exe main.cpp' viene eseguito e produce il file myprog.exe
- Se eseguiamo di nuovo make, il file all comunque non viene trovato. Dunque all è ancora un phony target e il comando viene di nuovo eseguito. Risposta:

  g++ -o myprog.exe main.cpp

- Questo comportamento non è l'obiettivo di un Makefile

Un semplice esempio

- Quando un sorgente include un header file è importante specificare l'header file tra i prerequisiti. Perchè?
- Consideriamo il seguente sorgente main.cpp:

```
#include <iostream>
#include "defs.h"
using namespace std;

int main() {
   cout << VALUE;
}
```

e il seguente header file defs.h:

```
#define VALUE 10
```

Un semplice esempio

- Se usiamo il makefile:

```
myprog.exe :
    g++ -o myprog.exe main.cpp
```

- Eseguiamo make e poi myprog. Il valore mostrato è 10
- Modifichiamo defs.h:

  #define VALUE 20
- Eseguiamo make:

  make: `myprog.exe' is up to date.
- E poi myprog. Il valore mostrato è ancora 10
- Se invece defs.h è un prerequisito di myprog.exe, il programma viene ricompilato e il valore mostrato è quello corretto

Un esempio di makefile

```
prova.exe: prova.o funzprova.o
    C:\Programmi\Dev-Cpp\bin\g++ -o prova.exe prova.o funzprova.o

prova.o: prova.cpp prova.h
    C:\Programmi\Dev-Cpp\bin\g++  -c prova.cpp

funzprova.o: funzprova.cpp prova.h
    C:\Programmi\Dev-Cpp\bin\g++  -c funzprova.cpp

clean:
    rm prova.exe prova.o funzprova.o
```

Un esempio di makefile

- Per creare l'eseguibile chiamato prova basta digitare make
- Per eliminare il file eseguibile e tutti i file oggetto, digitare ***make clean***
- In entrambi i casi, make fa riferimento al file 'makefile' (file utilizzato per default)
- Il target 'clean' non è un file ma specifica un'azione. Siccome non compare tra i prerequisiti di nessuna altra regola, make lo ignora, a meno che non glielo diciamo specificamente (con 'make clean')

Come make elabora un makefile

- make inizia dal primo target (che non comincia con '.'), che prende il nome di ***default goal*** (nel nostro esempio è l'eseguibile *prova.exe*)
- I prerequisiti di prova.exe sono 2 file oggetto, per cui make deve elaborare le regole ad essi corrispondenti
- Ogni file oggetto viene (ri)compilato se:
  - Il file oggetto non esiste
  - Il file sorgente o uno dei file header da cui dipende sono più recenti del file oggetto
- Dopodichè, il file prova.exe viene (ri)collegato se:
  - Il file prova.exe non esiste
  - Esiste un file oggetto più recente di prova.exe

Come eseguire make

- Gli argomenti passati al comando make sono i target che make si occupa di aggiornare
  - es. 'make main.o' aggiorna solo il file main.o

  se non specificato, make aggiorna il primo target
- Opzioni:

  -f *file* utilizza il file *file* come makefile

  -C *dir* entra nella directory *dir* prima di leggere il makefile

Utilizzare "variabili"

```
CPP  = g++
OBJ  = prova.o funzprova.o
EXE  = prova.exe
BIN  = "C:\Programmi\Dev-Cpp\bin\"
RM   = rm –f

$(EXE): $(OBJ)
    $(BIN)$(CPP) -o $(EXE) $(OBJ)
prova.o: prova.cpp prova.h
    $(BIN)$(CPP) -c prova.cpp
funzprova.o: funzprova.cpp prova.h
    $(BIN)$(CPP) -c funzprova.cpp
clean:
    ${RM} $(OBJ) $(EXE)
```

Variabili

- Vantaggi:
  - Minore probabilità di errori
  - Scrittura semplificata del makefile

Phony target

- Un *phony target* è un target che non corrisponde ad un file da creare. È giusto un nome per alcuni comandi da eseguire dietro esplicita richiesta
- Ci sono due ragioni per usare phony target
  - Evitare conflitti con file aventi lo stesso nome
  - Migliorare le prestazioni
- Esempio:

```
clean:
    rm *.o a.out
```

- Quando si digita 'make clean', make non trova nessun file di nome clean e quindi interpreta clean come un phony target ed esegue il comando rm
- Cosa succede se viene creato un file di nome 'clean' e si digita make clean ?

Phony target

- Siccome non ha prerequisiti, il file 'clean' viene inevitabilmente considerato aggiornato e quindi il comando rm NON sarà eseguito
- Per evitare questo problema, si può esplicitamente dichiarare il target 'clean' come phony, utilizzando il target speciale .PHONY
- I prerequisiti del target speciale .PHONY sono considerati phony target

```
.PHONY : clean
clean :
    rm *.o a.out
```

- In questo modo, make non cerca un file di nome clean (e questo è un risparmio di tempo) ed esegue il comando rm

Introduzione alle regole implicite

```
main.o : main.cpp defs.h
    g++ -c main.cpp
```

- Non è necessario specificare il comando in questi casi. Se non lo si specifica, make utilizza il comando:

  g++ -c main.cpp -o main.o

- Se si sfrutta questa *regola implicita* è anche superfluo specificare main.c tra i prerequisiti
- Per cui è sufficiente:

  main.o : defs.h

Regole implicite

- Alcuni modi standard per creare certi tipi di file sono usati molto spesso (es. creare un .o a partire da un .cpp)
- make utilizza regole implicite per effettuare queste operazioni, evitando all'utente di esplicitarle nel makefile
- Il nome del file determina quale regola implicita usare
- Le regole implicite utilizzano diverse variabili predefinite, in modo che è possibile cambiare il modo in cui operano
- Per consentire a make di cercare una regola implicita per aggiornare un file target, occorre non specificare alcun comando:
  - Scrivere una regola senza comandi
  - Non scrivere affatto la regola

Regole implicite

- Una regola implicita può anche fornire sia i comandi che i prerequisiti per aggiornare il file target
  - es. la regola implicita per aggiornare main.o includerà tra i prerequisiti il sorgente main.cpp
- Tipicamente, si scrive una regola senza comandi quando occorre specificare prerequisiti che la regola implicita non può fornire
- In generale, make cerca una regola implicita per ogni target che non ha comandi e per ogni prerequisito per cui non è specificata una regola

Riferimenti

- http://www.gnu.org/software/make/manual/
- Da C++ a UML
  - Capitolo 12, par.12.13