Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria

CORSO DI

FONDAMENTI DI INFORMATICA C
Verso gli Oggetti: Moduli e Tipi di Dato Astratti

Prof. Franco Zambonelli – Ing. Giacomo Cabri

Lucidi realizzati in collaborazione con
Prof. Letizia Leonardi - DII, Univ. Modena

Anno Accademico 2001/2002

VERSO UNA TECNOLOGIA OBJECT-ORIENTED

MOTIVAZIONI

- progettare sistemi software di grandi dimensioni richiede *adeguati supporti*
- "crisi del software": i costi di gestione diventano *preponderanti* su quelli di produzione
- il software dovrebbe essere **protetto, riusabile, documentato, modulare, incrementalmente estendibile**

IL PUNTO CHIAVE

- i linguaggi di programmazione devono fornire
  - *non solo* un modo per esprimere *computazioni*
  - ma anche un modo per **dare struttura** alla descrizione
  - e un supporto per **organizzare bene il processo produttivo del software**

L'OBIETTIVO

- costruzione ***modulare* e *incrementale*** del software
- ottenuta per **estensione** / **specializzazione** di ***componenti***
  ➔ tecnologia ***component-based***
- Le strutture dati e le strutture di controllo (programmazione strutturata) *non bastano*
- Le funzioni e le procedure *non bastano*

LE NECESSITA'

- servono strumenti per la costruzione ***modulare* e *incrementale*** del software

CRISI DIMENSIONALE

Programmi di piccole dimensioni

- accento sull'algoritmo
- diagrammi di flusso
- programmazione strutturata

Non appena il programma cresce in dimensioni, non si riesce più a gestirlo: bisogna diminuire la complessità!

Programmi di Medie Dimensioni

- funzioni e procedure come astrazioni di istruzioni complesse
- decomposizione degli algoritmi in blocchi funzionali (anche su file diversi, per permettere lo sviluppo di equipe)

Alla base rimane il concetto di "algoritmo": tutto il programma è un grosso algoritmo spezzato su più funzioni/file

Programmi di Grandi Dimensioni

- *tipicamente trattano grandi moli di dati **MA***
  - la decomposizione funzionale non è adeguata
  - non c'è accoppiamento dati-funzioni che li elaborano: dati elaborati globalmente da tutto il programma
- *tipicamente devono essere sviluppati da team **MA***
  - la decomposizione funzionale e il disaccoppiamento dati-funzioni non permette la decomposizione del lavoro
- *tipicamente trattano ed elaborano dati relativi ad entità del mondo reale (persone, oggetti, grafici, documenti), e interagiscono con entità del modo reale MA*
  - le entità del mondo reale non sono "dati" su cui operano delle funzioni, ma sono entità con attributi e comportamenti, e devono essere trattate in modo coerente alla loro essenza

CRISI GESTIONALE

Il costo maggiore nel processo di produzione del software è dovuto al suo mantenimento:

- correttivo
- adattativo

Programmi di piccole dimensioni

- non difficilissimo trovare gli errori
- propagazione degli effetti delle modifiche limitate intrinsecamente dalle dimensioni del programma

Programmi di Medie Dimensioni

- gestione basata sulle procedure per l'individuazione degli errori
- gli effetti delle modifiche non sono limitati alle procedure in cui tale modifiche sono fatte, ma si propagano, a causa del non-accoppiamento dati funzioni

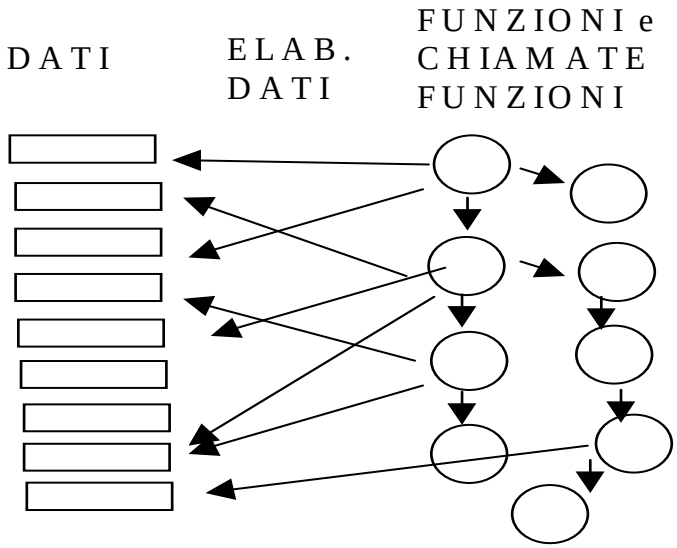
Programmi di Grandi Dimensioni

- quasi impossibile trovare gli errori
- se le modifiche si possono propagare non è possibile fare delle modifiche senza coinvolgere tutto il team di sviluppo

è necessario cambiare RADICALMENTE il modo di concepire, progettare e programmare il software!

IL CAMBIO DI PARADIGMA

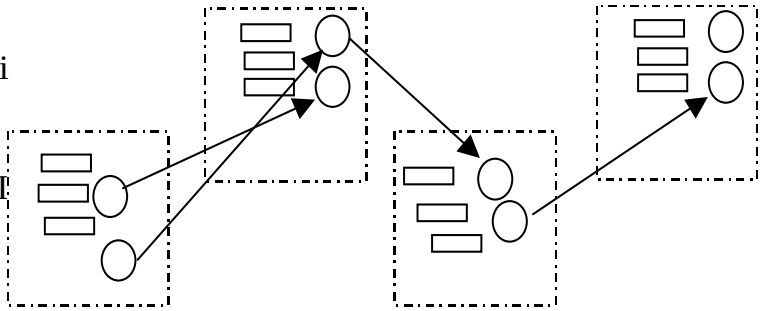
Tradizionale:



A Moduli:

MODULI =
File che raggruppano logicamente i dati e le funzioni che operano con esse

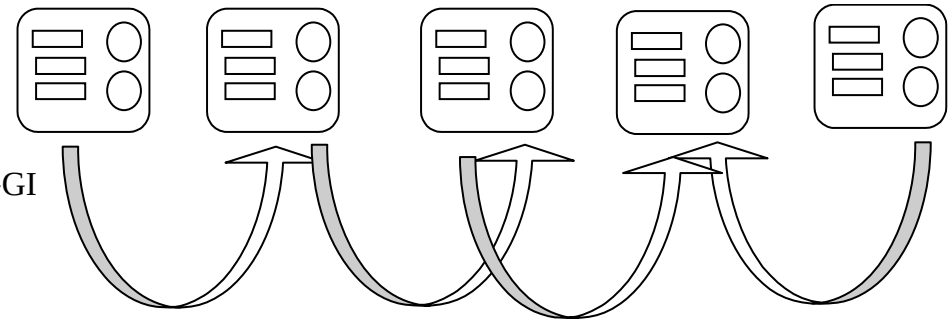
LIMITAZIONI SULL'USO DEI DATI DA PARTE DELLE FUNZIONI



Ad Oggetti:

OGGETTI =
dati + funzioni che vi operano

SCAMBIO DI MESSAGGI (richieste di servizio) TRA OGGETTI



CONCETTI CHIAVE

- Astrazione di Dato e Tipi di dato astratto (ADT)
- Modello Cliente / Servitore (*Client* / *Server*)
- Oggetti e Classi
- Moduli

ASTRAZIONE DI DATO

- Si focalizzare sulle *categorie concettuali* delle applicazioni
- Si costruisce un programma in termini di "entità", con propri attributi e comportamenti:
  - *attributi*: una struttura dati interna NASCOSTA (o non facilmente accessibile)
  - *comportamenti*: operazioni che possono AGIRE SUI DATI, che interagiscono tra loro

VANTAGGI

- agevola la modularità accoppiando dati e funzioni che su essi operano (diminuzione di complessità, certi dati è corretto che possono essere trattati solo tramite certe funzioni e viceversa).
- favorisce il controllo sull'integrità dei dati (solo certe funzioni possono agire sui dati e non funzioni arbitrarie che potrebbero fare dei danni o cose errate!). Se un dato risulta errato, si localizza facilmente in quale operazione è l'errore
- favorisce la creazione di componenti autonomi e "validati"
- minimizza la distanza concettuale tra problema e sua risoluzione

TIPO DI DATO ASTRATTO

Entità simili possono essere considerate "dello stesso tipo" (o della stessa classe), anche se con attributi specifici diversi.

ASTRAZIONE DI DATO: DEFINIZIONE

DATO CHE DEFINISCE UNA “ENTITA’” ASTRATTA (p.e. COME VIRTUALIZZAZIONE SOFTWARE DI UNA ENTITA’ DEL MONDO REALE:

- i dati (o “attributi”) che rappresentano tale entità – tipicamente non accessibili dall’esterno ma solo attraverso
- le operazioni (o “comportamenti” o “metodi”) che su tali entità si possono effettuare

TIPO DI DATO ASTRATTO (ADT): tipo da cui è possibile definire (“**istanziare**”) variabili come astrazioni di dato.

ESEMPI:

CONTATORE. Una entità che virtualizza qualcosa o qualcuno il cui compito è contare degli eventi

- Attributo numerico indica gli eventi che ha contato fino a quel momento
- Metodi per incrementare e per sapere fino a quanto ha contato fino a quel momento

STUDENTE. Una entità che rappresenta in software uno studente universitario

- attributi quali nome, cognome, numero di matricola, anno di corso, numero di esami dati, etc. etc.
- metodi per aggiornare l’anno di corso e gli esami dati

PROBLEMA: I linguaggi di programmazione tradizionali non permettono la definizione di astrazioni di dato e di tipi di dato astratto

ESEMPIO: il contatore

Approccio classico:

```
main()
{
…
int cont;

cont++;
cont--;

if (cont == MAX) {….}

cont = cont*cont;
/* puó avere senso per un intero ma non ha alcun senso
per una entità contatore */
}
```

PROBLEMI

- dov'è l'entità, il tipo di dato astratto, contatore?
- dove sono le operazioni ammesse sui dati di tipo contatore?
- possiamo fare cose NON SENSATE sul contatore
- stiamo usando le mosse elementari della macchina C (integer e operatori di incremento), non stiamo usando delle categorie concettuali di tipo contatore! non possiamo ri-usare il concetto di contatore!

ESEMPIO: studenti

Approccio Classico:

```
typedef struct Studente {
        char nome[20];
        char cognome[20];
        int matricola;
        int num_esami_dati;
        struct esami[29] {char nome[20]; int voto;}
                    }

void nuovo_esame_dato(Studente s, char *nome_esame, int
voto_preso)
{
    strcpy(s.esami[s.num_esami_dati].nome, nome_esame);
    s.esami[s.num_esami_dati++].voto = voto_preso;
}
/* e altre funzioni varie…..*/

main()
{
Studente s1, s2;


s1.nome = "pippo"; s2.cognome = "rossi";
/* NON BELLO: COSI' ACCEDO AI DATI INTERNI DI STUDENTE*/

/* POI POSSO FARE */
nuovo_esame_dato(s1, "Sistemi Operativi", 18);

/* MA ANCHE */
s1.esami[4].voto++; /* NON DOVREI! NON HA SENSO! Non e'
una operazione ammissibile sul tipo di dato*/

                              }
```

COME FARE:

- Cercare di costruire “astrazioni di dato” nei linguaggi tradizionali
  - Scomposizione di un programma su piu’ file
  - Associare all’interno di uno stesso file i dati e le funzioni che operano su essi
  - Limitare l’uso dei dati in un file da parte di altri funzioni in altri file che potrebbero usarli in modo non corretto

- Oppure:
  - Usare un nuovo linguaggio di programmazione, basato su principi diversi e dove:
  - PER DEFINIZIONE, i dati sono accoppiati alle funzioni a formare astrazioni di dato
  - SI PERDA il dualimo dati-funzioni → OGGETTI

Proviamo per ora la prima strada:

- scomposizione in piu’ file (moduli) di un programma C
- accoppiare in un file i dati e le funzioni che su di essi operano
- confinando l’uso dei dati a quelle funzioni
- vietando l’uso di certi dati da parte delle altre funzioni

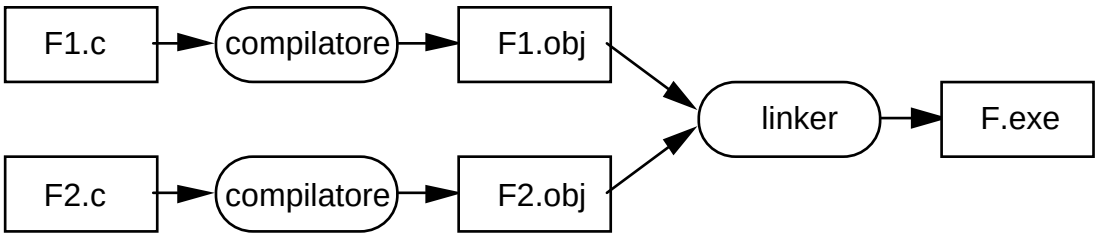
APPLICAZIONI C SU PIU' FILE

Serve poter sviluppare applicazioni su piú file:
- **alcune funzioni e alcune definizioni di dati in un file**
- **altre funzioni e dati in file diversi**
- **confinare l'uso di certe variabili a certe funzioni**

Perché??

1. Se il programma è di dimensioni notevoli:
- piú facile scrivere e aggiornare file piccoli
- accoppiamento dati funzioni
- divisione logica tra le varie parti del programma
- modifiche chiaramente localizzate
- migliora i tempi di compilazione (quando si fanno delle modifiche si ri-compilano solo i file modificati e poi si rifa il link) → concetto di progetto!
- accoppiamento dati funzioni → MODULI, quasi ADT

2. Se il programma è scritto da un team:
- ognuno scrive su propri file e poi si collega il tutto



Cosa serve?
- I programmi devono poter usare dati e funzioni definiti altrove (da altre persone e, quindi, in altri file)!
- Una metodologia per scomporre le cose su piú file

Vediamo quindi questi due punti partendo da:
- gestione variabili per poterle proteggere
- metodologia di scomposizione

CLASSI DI MEMORIZZAZIONE
TEMPO di VITA — VISIBILITÀ

In C, ogni **entità (variabile o funzione)** usata in un programma è caratterizzata da

- **Nome**, identificatore unico nel programma (o in una porzione)
- **Tipo**, per indicare l'insieme dei valori
- **Valore**, tra quelli ammessi dal tipo
- **Indirizzo**, riferimento alla memoria che la contiene
- **Tempo di vita**, durata di esistenza nel programma
- **Visibilità** (scope) del nome nel programma

Tempo di vita e visibilità sono specificati mediante la **CLASSE di MEMORIZZAZIONE** ⇨ indica il tipo di area di memoria in cui una entità viene memorizzata

NOTA BENE: In altri linguaggi, tempo di vita e visibilità di una entità non sono concetti indipendenti uno dall'altro

Le classi di memorizzazione sono 4:

1. **auto** ⇨ *automatica*
2. **register** ⇨ registro (*caso particolare di auto)*
3. **static** ⇨ *statica*
4. **extern** ⇨ *esterna*

IMPORTANTE: La classe di memorizzazione può essere applicata alla **definizione** sia di **variabili** che di **funzioni**

PERÒ ... per **variabili** sono applicabili tutte e 4
per **funzioni** sono applicabili solo **static** e **extern**

Alle **dichiarazioni** si applica**, in genere,** solo la classe di memorizzazione **extern**

CLASSI DI MEMORIZZAZIONE PER LE VARIABILI

VISIBILITÀ
possibilità di riferire la variabile
TEMPO di VITA
durata della variabile all'interno del programma

1. CLASSE di MEMORIZZAZIONE auto

- **default** per **variabili locali a un blocco o funzione**
  NOTA BENE: non si applica alle funzioni
- **VISIBILITÀ**
  La variabile è **locale** e quindi, è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di dichiarazione in poi
- **TEMPO DI VITA**
  la variabile è **temporanea** cioé esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- **ALLOCAZIONE:**
  su **STACK** (valore iniziale indefinito di default)

ESEMPIO:

```
somma(int v[ ],int n)
{
int k,sum = 0;   /*  Quanto vale k ?  */
/* è come scrivere auto int k,sum = 0; */
for (k = 0; k < n; k++) sum += v[k];
return sum;
}
```

2. CLASSE di MEMORIZZAZIONE register

- **come le auto**
  e quindi
  NOTA BENE: non si applica alle funzioni

  VISIBILITÀ
  La variabile è **locale** e quindi, è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di dichiarazione in poi
  TEMPO DI VITA
  la variabile è **temporanea** cioé esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- **ALLOCAZIONE:**
  su **REGISTRO MACCHINA** (valore iniziale indefinito di default)
  Solo se possibile cioè se:
  - registri disponibili
  - dimensione variabile compatibile con quella dei registri

ESEMPIO:

```
somma(int v[ ],int n)
{
register int k,sum = 0;
for (k = 0; k < n; k++) sum += v[k];
return sum;
}
```

NOTA:
La classe di memorizzazione register può essere usata anche per i parametri di una funzione

3. CLASSE di MEMORIZZAZIONE static

- **TEMPO DI VITA**
  la variabile è **permanente** per tutto il programma: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine

La definizione di una variabile statica può essere:
1. globale cioé esterna ad ogni funzione *oppure*
2. locale cioè all'interno di una funzione o blocco

- QUESTO INFLUENZA LA **VISIBILITÀ**
  1. la variabile è visibile ovunque, dal punto di dichiarazione in poi, ma **solo all'interno del file che la contiene**
  2. visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di dichiarazione in poi
- **ALLOCAZIONE:**
  nei DATI GLOBALI (***valore iniziale di default 0***)
  Per il microprocessore 8086/88 l'allocazione è nel DATA SEGMENT

ESEMPIO:

File "CCC.c"

```
funA(...);
funB(void);

static int ncall = 0;
...
funA(...)
  {  ncall++;  ...  }

static funB(void)
  {  return ncall;  }
```

File "DDD.c"

```
void fun1(...);
extern funA(void);
/* OK! */
extern funB(void);
/*ERRATO*/
...
void fun1(...)
  {  ncall++;  /*ERRATO!*/
funA();    /* OK! */
funB(void); /* ERRATO!*/
 }
```

ESEMPI: VARIABILI AUTOMATICHE E STATICHE

ESEMPIO 1: Variabile statica locale

```
#include <stdio.h>
void static_demo (void);  /* dichiarazione funzione */
main()
{   int i;

for( i= 0; i < 10; ++i)
    static_demo();
/* ...- static_variable ... ERRORE!!! */
}

void static_demo(void)
{   int variable = 0;
    static int static_variable;
printf("automatic = %d, static = %d\n",
    ++variable, ++static_variable);
}
```

`variable` è una ***variable automatica***

visibile solo nella funzione `static_demo()` e con
tempo di vita pari alla singola invocazione
allocata nella parte di STACK e
inizializzata esplicitamente sempre a 0 ad ogni invocazione

`static_variable` è una ***variabile statica locale***

visibile solo nella funzione `static_demo()`, ma con
tempo di vita pari a tutto il programma
allocata nella parte di DATI GLOBALI e
inizializzata implicitamente a 0 solo all'inizio dell'esecuzione del programma

Quindi il valore della variabile `variable` è sempre uguale ad 1, mentre il valore della variabile `static_variable` viene incrementato ad ogni chiamata

ESEMPIO 2: Variabile statica globale

```
/* file static1.c */
#include <stdio.h>

static int static_var;
void st_demo (void); /* dichiarazione funzione */

void main()
{   int i;
for( i= 0; i < 10; ++i) st_demo();

static_var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("static globale = %d\n", static_var);
}

void st_demo(void)
{   int variable = 0;

printf("automatic = %d, static globale = %d\n",
    ++variable, ++static_var);
}
```

`variable` è una ***variable automatica***

⇨ *come prima*

`static_var` è una ***variabile statica globale***

visibile solo nel file `static1.c`

tempo di vita pari a tutto il programma

allocata nella parte di DATI GLOBALI e

inizializzata implicitamente a 0 solo all'inizio dell'esecuzione del programma

Quindi il valore della variabile `variable` è sempre uguale ad 1, mentre il valore della variabile `static_var` viene incrementato ad ogni chiamata e poi viene posto uguale a 100 nella funzione main()

4. CLASSE di MEMORIZZAZIONE extern

- **default** per **variabili globali** cioé esterne ad ogni funzione
- vale sia per definizioni che per dichiarazioni
- **VISIBILITÀ**
  globale cioé la variabile è visibile ovunque, dal punto di dichiarazione in poi anche **al di fuori del file** che ne contiene la definizione
- **TEMPO DI VITA**
  la variabile è **permanente** per tutto il programma: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- **ALLOCAZIONE:**
  nei DATI GLOBALI (***valore iniziale di default 0***)
  Per il microprocessore 8086/88 l'allocazione è nel DATA SEGMENT

ESEMPIO:

File "AAA.c"

```
extern void fun2(...);
...
int ncall = 0;
...
fun1(...)
{
ncall++;
...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...
void fun2(...)
{
ncall++;
...
}
```

ESEMPIO: VARIABILE EXTERN

```
/* file main.c */
#include <stdio.h>
int var;
/* definizione variabile esterna: extern di default */
extern void demo (void);
/* dichiarazione funzione esterna */

void main()
{   int i;
for( i= 0; i < 10; ++i)
    demo();
var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("extern = %d\n", var);
}

/* file demo.c */
#include <stdio.h>

extern int var;  /* dichiarazione variabile esterna*/
void demo(void)
{   int variable = 0;
printf("automatic = %d, extern = %d\n",
    ++variable, ++var);
}
```

variable viene posta sullo STACK e inizializzata a 0 **ad ogni invocazione della funzione** demo

var viene posta nella parte DATI GLOBALI e inizializzata a 0 **una sola volta**

ANSI C:

`int var;` ⇨ viene considerata una **definizione** perchè **non** è stata usata esplicitamente la classe di memorizzazione **extern** (valida di default)

`extern int var;` ⇨ viene considerata una **dichiarazione** perchè è stata usata esplicitamente la classe di memorizzazione **extern** (valida di default)

CLASSI DI MEMORIZZAZIONE PER LE FUNZIONI

VISIBILITÀ
possibilità di riferire la funzione
TEMPO di VITA
durata della funzione all'interno del programma
⇨ **sempre globale** cioé pari all'intera durata del programma

ALLOCAZIONE:
sempre nella parte di CODICE
Per il microprocessore 8086/88 l'allocazione è nel CODE SEGMENT

NOTA BENE:
Le classi di memorizzazione auto, register e static locali (a blocchi o funzioni) non hanno senso poiché NON è possibile definire una funzione all'interno di un'altra funzione (o blocco)

1. CLASSE di MEMORIZZAZIONE static

La definizione di una funzione statica può essere solo globale cioé esterna ad ogni funzione

- **VISIBILITÀ**
  la funzione è visibile ovunque, dal punto di definizione in poi, ma **solo all'interno del file che la contiene**

ESEMPIO:

File "CCC.c"

```
fun1(...);
funA(void);
extern funB(void);
static int ncall = 0;
...
static fun1(...)
  {  ncall++;  ...  }
funA(void)
  {  return ncall;  }
```

File "DDD.c"

```
void fun1(...);
funB(void);
extern funA(void);
static int ncall = 0;
...
static void fun1(...)
  {  ncall++;  ...  }
funB(void)
  {  return ncall;  }
```

ESEMPIO: FUNZIONE STATICA

```
#include <stdio.h>
static void static_fun (void);
/* dichiarazione funzione */
void main()
{   int i;
for( i= 0; i < 10; ++i)
    static_fun();
}
static void static_fun(void)
{ printf("Sono una funzione statica:
        sono visibile solo in questo file\n");
}

static void static_fun(void);
```

dichiarazione/prototipo funzione

⇨ questa dichiarazione serve per poter usare questa funzione nel main(), riportando la definizione alla fine

Si può evitare, se si definisce direttamente la funzione prima del main()

La classe di memoria `static` può anche essere omessa

`static void static_fun(void) {...};` ***definizione funzione***

2. CLASSE di MEMORIZZAZIONE extern

- **default** per le **funzioni**

- vale sia per definizioni che per dichiarazioni

- **VISIBILITÀ**
  globale cioé la funzione è visibile ovunque, dal punto di definizione in poi anche **al di fuori del file** che ne contiene la definizione

ESEMPIO:

File "AAA.c"

```
extern void fun2(...);
...
int ncall = 0;
...
fun1(...)
{
ncall++;
...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...
void fun2(...)
{
ncall++;
...
}
```

ESEMPIO: FUNZIONE EXTERN

NOTA BENE: è lo stesso di prima

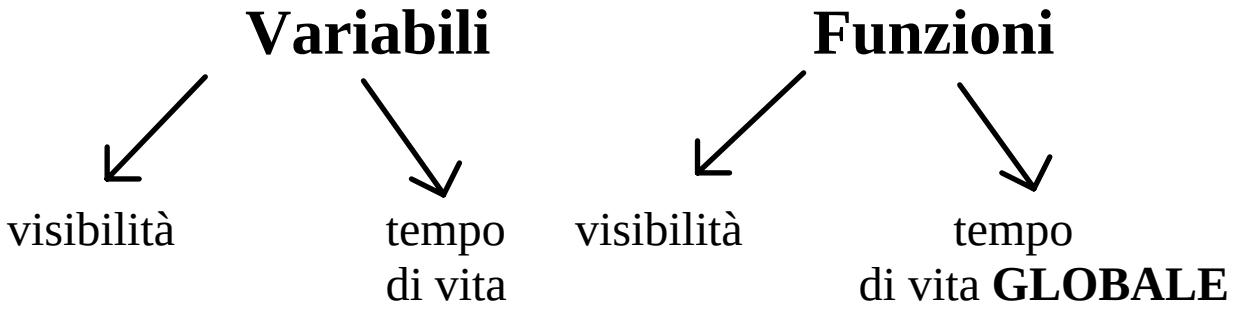
```
/* file main.c */
#include <stdio.h>
int var;
/* definizione variabile esterna: extern di default */

extern void demo (void);
/* dichiarazione funzione esterna */

void main()
{   int i;
for( i= 0; i < 10; ++i)
    demo();
var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("extern = %d\n", var);
}

/* file demo.c */
#include <stdio.h>

extern int var;  /* dichiarazione variabile esterna*/

void demo(void)
/* definizione funzione esterna: extern di default */
{   int variable = 0;
printf("automatic = %d, extern = %d\n",
    ++variable, ++var);
}
```

`extern void demo (void);` ***dichiarazione/prototipo funzione***

⇨ si usa la stessa convenzione usata per le variabili anche se per una funzione la differenza fra definizione e dichiarazione è sempre chiara

`void demo (void) { ... };` ***definizione funzione***

CLASSI DI MEMORIA DELLE ENTITÀ IN C



CLASSI DI MEMORIZZAZIONE

1) **auto** / **register** — **N.B.: solo per variabili**
variabili LOCALI a blocchi o funzioni
(default auto)
VISIBILITÀ: limitata al blocco
T. DI VITA: limitata al blocco
ALLOCAZIONE: STACK (auto)

2) static ⇨ **visibilità e tempo di vita scorrelato**

a) dentro a funzioni o blocchi
N.B.: solo per variabili
VISIBILITÀ: limitata al blocco
TEMPO DI VITA: globale

b) fuori da qualunque funzione
sia variabili che funzioni
VISIBILITÀ: limitata al file
TEMPO DI VITA: globale
ALLOCAZIONE: DATI GLOBALI (variabili)
CODICE (funzioni)

3) extern ⇨ **variabili e funzioni**
(default a livello di file)
VISIBILITÀ: globale
TEMPO DI VITA: globale
ALLOCAZIONE: DATI GLOBALI (variabili)
CODICE (funzioni)

Applicazione in C su più file

La presenza di **definizioni** e **dichiarazioni** di entità insieme con il concetto di **classe di memoria** rende possibile sviluppare una applicazione su più file

Ogni singolo file viene **compilato** in modo **INDIPENDENTE** e poi i vari file oggetto sono messi insieme al **collegamento**

In un singolo file, per poter usare entità definite negli altri file è necessario **dichiarare le entità esterne** utilizzate

- Infatti, durante la compilazione di un singolo file sorgente, il compilatore non può conoscere le entità (variabili e funzioni) definite negli altri file e quindi ha necessità delle loro dichiarazioni per poter fare gli opportuni controlli che il loro uso sia appropriato
- **è necessario dichiarare le entità esterne utilizzate**

DICHIARAZIONE: specifica le proprietà di una entità

- sia **funzione** (in ANSI C mediante il suo prototipo)
- sia **variabile**
- sia **tipo di dato**

→ in ogni modo, non viene allocato spazio in memoria

```
extern fattoriale(int n);     /* prototipo funzione */
extern float xyz;    /* dichiarazione variabile */
typedef short int Signed16;  /* dichiarazione tipo */
```

DEFINIZIONE: specifica le proprietà di una entità e la sua allocazione

- sia **funzione**
- sia **variabile**

```
fattoriale(int n) {.../* codice funzione */}
float xyz = 10.5;
```

segue Applicazione su più file

Ogni entità può essere dichiarata *più volte* (in file diversi) ma deve essere definita *una e una sola* volta

Una entità è **dichiarata nei file** che la usano
ma
definita solo ed unicamente in un file che la alloca

Sia per le dichiarazioni che per la definizione si deve usare la classe di memoria **extern**

La clausola extern quindi è usata

- **sia da chi le esporta** (cioè chi mette a disposizione l'entità),
- **sia da chi la importa** (cioè chi usa l'entità),seppure con semantica diversa

La classe **extern** è il **default** per ogni entità definita/dichiarata a livello di programma

Metodologia di Uso

(adottata dall'ANSI C)

una sola *definizione* (con eventuale inizializzazione esplicita) in cui non compare esplicitamente la clausola extern
le *dichiarazioni* riportano esplicitamente la classe *extern*

ESEMPIO:

Il file "f3.c" **mette a disposizione**
la variabile x e la funzione f() - DEFINIZIONI

I file "f1.c" e "f2.c" **utilizzano** la variabile x e la funzione f()
messa a disposizione dal file "f3.c" - DICHIARAZIONI

f1.c

```
extern  int x;

extern float f
    (char c);

/*dichiarazioni
==> IMPORT */

void prova()
{
< uso di x e f >
}
```

f2.c

```
extern  int x;

extern float f
    (char c);

/*dichiarazioni
==> IMPORT */

void main()
{
< uso di x e f >
}
```

f3.c

```
int x = 10;

float f (char c);
{  var  locali  e
codice di f >
}

/*definizioni
==> EXPORT
*/
```

COMPILAZIONE INDIPENDENTE

bisogna **compilare** f1.c, f2.c e f3.c

LINKING

bisogna fare il **linking** di f1.obj, f2.obj e f3.obj **insieme**

→ **RISOLVE I RIFERIMENTI ESTERNI**

per ottenere il programma nella sua forma eseguibile

Negli ambienti di sviluppo integrato bisogna:

- **creare un progetto**
- **aggiungere al progetto i file parte del programma completo**
- **Quindi: *compile* compila un singolo file, *make* ri-compila e linka i file che sono cambiati dall'ultima volta che si sono compilati, *link* collega i file già compilati, *buildall* fa il processo completo di costruzione**

segue ESEMPIO:

Tutte le **dichiarazioni** possono essere inserite **in un HEADER FILE** "f3.h" incluso dai file utilizzatori. Serve per:

- non riscrivere un sacco di volte le stesse dichiarazioni su più file
- per modificarle una sola volta le dichiarazioni e fare avere effetto a queste modifiche su tutti i file cui servono le dichiarazioni

"f3.h"

```
Extern int x;
extern float f(char c);
   ...
```

f1.c

```
#include "f3.h"

/*dichiarazioni
==> IMPORT */

void prova()
{
< uso di x e f >
}
```

f2.c

```
#include  "f3.h"

/*dichiarazioni
==> IMPORT */

void main()
{
< uso di x e f >
}
```

f3.c

```
int x = 10;

float f (char c);
{  var  locali  e
codice di f >
}

/*definizioni
==> EXPORT
*/
```

Un **header file** contiene *solitamente* **dichiarazioni** e non **definizioni**

→ vedi file header di libreria

Struttura di un programma (in generale)

In ogni file, possiamo avere

DICHIARAZIONE di	**DEFINIZIONE** di
Tipi Variabili Funzioni	Variabili (Dati) Funzioni (Algoritmi)

- Ogni programma, anche se suddiviso su più file, deve contenere *sempre* **una**, ed **una sola**, **funzione** di nome **main**
- L'esecuzione avviene attraverso
  funzioni che si invocano

  la visibilità da un file all'altro viene garantita dalle dichiarazioni extern di variabili/funzioni definite extern di default

  -l'esecuzione inizia dalla funzione **main**
  -il main può invocare altre funzioni (anche di altri file)
  -l'esecuzione termina quando
    - termina il flusso di esecuzione del main
    - viene chiamata una delle funzioni di sistema che fanno terminare l'esecuzione (ad es. **exit**)
- Le variabili possono essere usate (sono visibili) solo **dopo** la loro definizione o dichiarazione di tipo **extern**
- Le funzioni possono essere usate anche **prima** della loro definizione, purchè vengano dichiarate

  nel caso che siano definite in altri file, la dichiarazione deve presentare esplicitamente la classe **extern**

Struttura di un programma (ogni singolo file)

☸☸ inclusione **header file** per **librerie standard** C
#include <stdio.h> ...
☸☸ dichiarazione **tipi**
... tipo1; ..... tipoN;
☸☸ definizione **variabili globali** all'intero programma
tipoVar1 nomeVar1, ...; ...;
tipoVarJ nomeVarJ, ...;
☸☸ definizione **variabili statiche**
static tipoVarJ+1 nomeVarJ+1, ...; **static** ...;
static tipoVarK nomeVarK, ...;
☸☸ dichiarazione **variabili globali** all'intero programma
extern tipoVarK+1 nomeVarK+1, ...; **extern** ...;
extern tipoVarN nomeVarN, ...;
☸☸ dichiarazione **prototipi funzioni** (definite sotto)
tipo1 F1(parametri); ... **static** tipoK+1 FK+1(parametri); ...
tipoK FK(parametri); **static** tipoJ FJ(parametri);
☸☸ dichiarazione **prototipi funzioni** (definite altrove)
extern tipoJ+1 FJ+1(parametri); **extern**...
extern tipoN FN(parametri);
☸☸ *eventuale* definizione della **funzione main**
main(int argc, char **argv)
{
- definizione **variabili locali (auto e static)** al main
- codice del main }

☸☸ definizione della generica **funzione esterna Fy** (con y=1...K)
tipoy Fy(parametri)
{
- definizione **variabili locali (auto e static)**
- codice della funzione Fy }

☸☸ definizioni della generica **funzione statica Fx** (con x=K+1...J)
static tipox Fx(parametri)
{
- definizione **variabili locali (auto e static)**
- codice della funzione Fx }

ASTRAZIONI DI DATO e TIPI DI DATO ASTRATTI IN C

Obiettivo:

accoppiare dati e funzioni in modo tale che:

- **su certe categorie di dati**
- **si possano applicare solo le funzioni che hanno senso su quei dati**

Nei linguaggi “classici”, non a oggetti, è comunque possibile seguire una “filosofia di progettazione del software” a oggetti. Però le limitazioni del linguaggio rendono l'approccio o "sporco" o limitato dal punto di vista espressivo.

In C, due approcci possibili:

- ADT realizzati tramite ***typedef e variabili*** (e/o puntatori)
- Astrazioni di Dato realizzati tramite ***moduli*** (files)

I° Caso: ***tipo di dato astratto*** definito da `typedef`

- **gli oggetti sono variabili** del tipo definito da `typedef`
  → il cliente può *crearne tante istanze* quante desidera
- **le operazioni sono funzioni**
  fra i cui parametri figura **il nome della variabile-oggetto** (istanza) su cui devono agire

Conseguenza:
il cliente deve *trasferire esplicitamente l'oggetto* ai suoi servitori (funzioni che realizzano le operazioni dell'oggetto)

II° Caso: ***astrazione di dato*** realizzato tramite moduli o file

- il C *non fornisce un costrutto "modulo" nel linguaggio*
- si usano i **file** come **contenitori**, sfruttandoli
  - *sia* per separare interfaccia e implementazione (`.h` / `.c`)
  - *sia* come (elementari) *meccanismi di protezione*

Conseguenza:
la *separazione concettuale* interfaccia/implementazione diventa una *separazione fisica*
→ *i clienti includono lo header* per usare l'ADT

ES. 1: UN CONTATORE come ADT

counter.h...

```
typedef int Contatore;

int  getValue(Contatore c);
void setValue(Contatore* c, int v);
void inc(Contatore *c);
void dec(Contatore *c);
```

.... e counter.c

```
#include "counter.h"
int  getValue(Contatore c){return c;}
void setValue(Contatore* c, int v) {*c=v;};
void inc(Contatore *c){ (*c)++;}
void dec(Contatore *c){ (*c)--;}
```

Uso:

```
#include "counter.h"
void main(){
  Contatore c1, c2;
  setValue(&c1, 10); setValue(&c2, -31);
  printf("Valore di c1 = %d", getValue(c1));
  inc(&c2); dec(&c1);
}
```

PRO:

- Possibilità di definire e usare *più oggetti* di tipo counter

CONTRO:

- Necessità di passare puntatori
- Rischio di usare oggetti *non inizializzati* (se ci si dimentica setValue...)
- Possibilitá di fare operazioni non ammesse sul contatore, poiché c'è libero accesso alla variabile. Es.

```
Contatore *=Contatore;
```

ES. 2: UN CONTATORE come singola astrazione di dato

count.h...

```
int  getValue(void);
void setValue(int);
void inc(void);
void dec(void);
```

.... e count.c

```
static int Contatore = 0;

int  getValue(void){return Contatore;}
void setValue(int v) {Contatore=v;};
void inc(void){ Contatore++;}
void dec(void){ Contatore--;}
```

Uso:

```
#include "count.h"
void main(){
  setValue(10);
  printf("Valore = %d", getValue());
  inc();
}
```

PRO:

- Non è necessario definire variabili e passare oggetti o puntatori
- Il contatore è protetto, essendo inaccessibile dall'esterno del file count.c in modo diretto
- Solo le operazioni dentro definite dentro al file count.c possono accedere a cont, non c'è rischio di fare operazioni errate!

CONTRO:

- E' impossibile definire e usare *più oggetti* contatori: non c'e' il tipo di dato astratto! E'una *singola astrazione di dato*

C'è *un solo contatore*, già esistente, e si può usare *solo quello!*

ESERCIZI PROPOSTI

- Provare a realizzare con i due approcci sopra proposti i concetti di astrazione di “studente” e tipo di dato astratto “studente”
- Discutere vantaggi e svantaggi dei due approcci

- considerando gli esempi di programmi sviluppati nei primi moduli di Fondamenti di Informatica, pensare a quali tra i dati usati nei programmi potrebbero essere utilmente definiti in termini di astrazioni di dato e tipi di dato astratti

- pensare a entità del mondo reale e provare a definire in che modo questi si potrebbero virtualizzare in software in termini di astrazioni di dato e tipi di dato astratti
  - quali attributi li descrivono?
  - quali sono le operazioni ammesse?