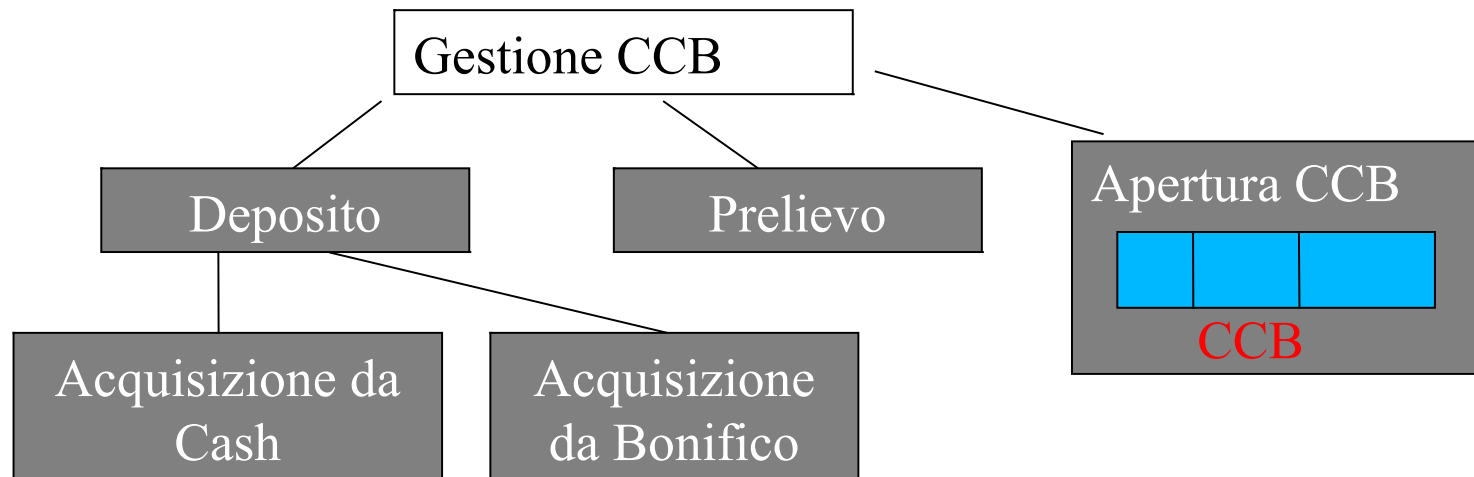


# Programmazione Modulare

# Moduli

- Secondo la metodologia stepwise refinement, un sistema deve essere progettato sulla base di sotto-programmi, detti **moduli**.
- Ciascun modulo può essere sviluppato, esteso e mantenuto separatamente dagli altri ed ispezionato facilmente in caso di errori.

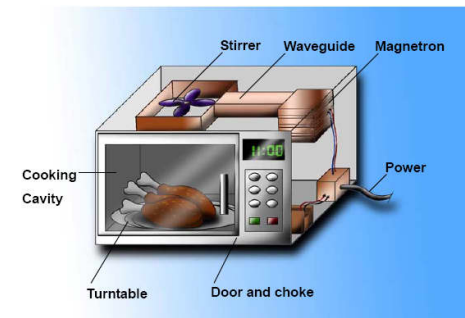


# Moduli

- Questo consente di lavorare per moduli che risolvono sotto-problemi.
- I moduli corrispondono a file in codice sorgente, distinti dal programma principale (che contiene il `main()`).
- I moduli sono implementati attraverso **funzioni** e **procedure**.
- Ogni linguaggio di programmazione consente di definire moduli, funzioni e procedure e di usarli e invocarli.

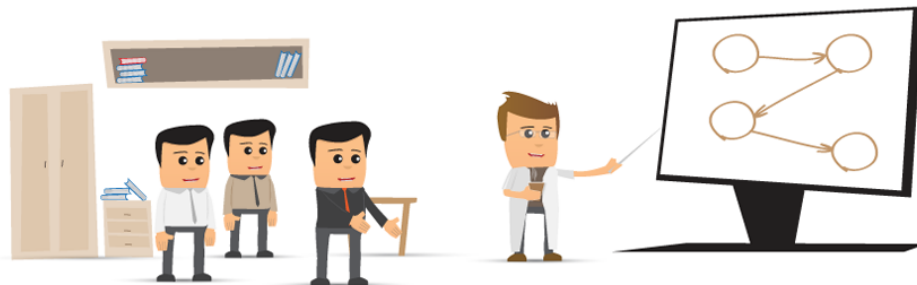
# Astrazione

- Funzioni e procedure supportano la **astrazione**, che è un processo logico che consente di risolvere problemi **eliminando dettagli e considerando aspetti essenziali**.
- Ad esempio: nel problema di trovare il percorso “più breve” tra due località la distanza è un aspetto essenziale, mentre la presenza di distributori di carburanti è un aspetto irrilevante.
- Nel quotidiano il principio di astrazione è costantemente applicato ogni qualvolta utilizziamo uno strumento senza per questo sapere come è realizzato. Ad esempio il forno a microonde:



# Astrazione

- Similmente, nella **programmazione** l'astrazione allude alla distinzione che si fa tra:
  - cosa (*what*) fa un modulo software
  - come (*how*) esso è implementato
- In una team aziendale di una software house, la prospettiva del *cosa* fa un modulo astratto è quella dell'architect software, mentre la prospettiva del *come* implementare un modulo astratto è quella dello sviluppatore.



# Moduli

- Sia funzioni che procedura fanno astrazione sulla funzionalità.
- Le funzioni sono astrazione sugli operatori e devono computare espressioni al fine di produrre un valore.

$x * f(a)$

- Le procedure sono astrazione sulle istruzioni e devono eseguire comandi al fine di cambiare lo stato dei dati.

`salva_dati_conto_corrente(numero_conto);`

# Moduli

- La programmazione per moduli consente
  - Astrazione
  - Riutilizzabilità
  - Leggibilità
  - Scomposizione del problema

# Astrazione

```
int main() {  
    int peso = 90;  
    int altezza= 198;  
    float bmi= calcolaBMI(peso,altezza);  
    printf("Il tuoBMI è %.2f",bmi);  
}
```

- Permette di programmare un sistema assumendo la disponibilità di moduli e considerando **cosa fanno** (aspetto rilevante) e non **come** sono implementati (dettaglio)



# Riusabilità

```
int main() {  
    int peso = 90;  
    int altezza= 198;  
    float bmi= calcolaBMI(peso,altezza);  
    printf("Il tuoBMI è %.2f",bmi);  
}
```

- Uno stesso modulo può essere riusato in diversi sistemi. Ad esempio calcolaBMI() può essere usato in un programma per la gestione di atleti di una squadra di calcio o in un programmazione per la gestione di pazienti in un ospedale.

# Scomposizione

```
int main() {  
    int altezza= leggiAltezza();  
    int peso= leggiPeso();  
    float bmi= calcolaBMI(peso,altezza);  
    printf("Il tuoBMI è %.2f",bmi);  
}
```

*Acquisizione dati*

*Elaborazione*

*Visualizzazione*

- Un programma può essere progettato e implementato associando a ciascun modulo (o più semplicemente a funzioni e procedure) delle proprie funzionalità o comportamenti, che messi assieme realizzano la funzionalità complessiva del programma.

# Leggibilità

```
int main() {  
    int altezza= 90;  
    int peso=    80;  
    float bmi= calcolaBMI(peso,altezza);  
    printf("Il tuoBMI è %.2f",bmi);  
}
```

è più leggibile del seguente

```
int main() {  
    int altezza= 90;  
    int peso=    80;  
    float bmi= (float) peso / ((altezza/100)*(altezza/100) )  
    printf("Il tuoBMI è %.2f",bmi);  
}
```

# Programmazione modulare

Per quanto visto, quindi possiamo dire che

- un sistema sw deve essere suddiviso in moduli
- ciascuno modulo deve risolvere un sotto-problema
- quindi deve essere implementato e testato singolarmente ed **autonomamente**
- deve essere per quanto possibile **indipendente** dagli altri
- deve avere una convenzione chiara (interfaccia, segnatura) per poter essere usato

La modularizzazione richiede uno sforzo di progettazione (prima della implementazione) volto a identificare quali possono essere i moduli di un sistema.

# Programmazione modulare in C

- Nel linguaggio C la programmazione modulare è realizzata attraverso la definizione di funzioni, procedure e....
- librerie, che racchiudono collezioni di funzioni e procedure
- Infatti, quando si usa la direttiva `#include`, stiamo integrando e riutilizzando funzione/procedure scritte da altri.
- Una funzione/procedura è caratterizzata da
  - identificatore, lista di parametri, valore di ritorno, implementazione.
  - identificatore, lista di parametri, valore di ritorno definiscono la segnatura/prototipo/firma:

```
<valore_di_ritorno> <nome_della_funzione>(<parametri>) {  
<implementazione della funzione>  
}
```

# Programmazione modulare in C

Ad esempio, questo programma non è modularizzato:

```
int main() {  
    int x = 0;  
    printf("Inserisci voto:");  
  
    scanf("%d",&x);  
  
    if(x>18)  
        printf("Promosso");  
    else  
        printf("Non promosso");  
}
```

perchè vi sono tre sotto-problemi (acquisizione, elaborazione e presentazione) risolti da un unico programma

# Programmazione modulare in C

Si può pensare a definire una funzione che si occupi della elaborazione:

```
int main() {  
    int x = 0;  
    printf("Inserisci voto:");  
  
    scanf("%d",&x);  
  
    if(promosso(x))  
        printf("Promosso");  
    else  
        printf(«Non promosso»);  
}
```

```
int promosso(int voto) {  
    if(voto>18)  
        return 1;  
    else  
        return 0;  
}
```

ed integrarla nella funzione main() principale.

# Programmazione modulare in C

Così, una volta acquisito il dato da parte del `main()`, il controllo della esecuzione passa a `promosso(x)` che prima **lega** il parametro formale a quello attuale e poi ne verifica il valore. Al termine, il controllo ripassa a `main`

→ `main()`

→ `int promosso(x)`      *x ,parametro formale*  
                                 *voto, parametro attuale*  
                                 *voto=x // eseguita ma non*  
                                 *codificata*

→ `main() }`

Immaginando che `main()` e `promosso()` siano implementati da due differenti programmatori, quello di `promosso()` può fornirne una realizzazione indipendentemente da quello di `main()`, a cui sarà necessario sapere solo come usare la funzione, cioè la segnatura.



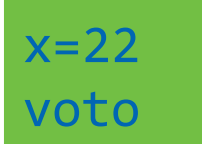
# Passaggio di parametri

- Il meccanismo che lega valori di parametri formali ai valori di parametri attuali si chiama **passaggio di parametri**. E' una assegnazione eseguita senza che il programmatore la implementi.
- In C esistono due meccanismi per il passaggio di parametri
  - per **valore**
    - copia il valore del parametro attuale nel parametro formale, così la funzione lavora su un' altra locazione di memoria senza modificare la vecchia.
  - per **riferimento**

# Passaggio di parametri: copia

Quando una funzione viene invocata, lavora su una area dati diversa dal `main()`. Durante il passaggio, si copia il valore del parametro attuale nel parametro formale, cioè in una altra locazione di memoria, quella associata al parametro formale.

```
int main() {  
    int x = 22;  
    printf("Inserisci voto:");
```

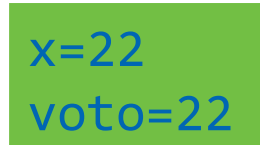


A green rectangular box representing a memory location. It contains the text 'x=22' on the top line and 'voto' on the bottom line.

x=22  
voto

```
    scanf("%d",&x);  
    if(promosso(x))
```

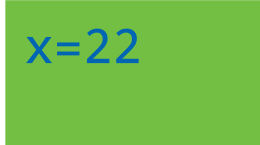
```
int promosso(int voto) {  
    if(voto>18)  
        return 1;  
    else  
        return 0;  
}
```



A green rectangular box representing a memory location. It contains the text 'x=22' on the top line and 'voto=22' on the bottom line.

x=22  
voto=22

```
        printf("Promosso");  
    else  
        printf("Non promosso");
```



A green rectangular box representing a memory location. It contains the text 'x=22'.

x=22

```
}
```

# Passaggio di parametri: copia

Nella copia, eventuali modifiche al parametro formale, effettuate durante la invocazione della funzione, non sono propagate al parametro attuale. Così, terminata la funzione, il valore del parametro formale viene cancellato e la locazione associata distrutta.

```
int main() {  
    int x = 5;  
    printf("Fuori: %d", x);  
    quadrato(x);
```

x=5  
y

**Fuori: 5**

```
void quadrato(int y) {  
    y = y*y  
    printf("Dentro: %d", y);  
}
```

y=5  
y=25

**Dentro: 25**

```
    printf("Fuori: %d", x);
```

x=5

**Fuori: 5**

# Passaggio di parametri: riferimento

Il passaggio di parametri standard è quello per copia. Fanno eccezione gli array, che vengono passati per **riferimento**, (vedremo in seguito cosa significa) dove il dato passato alla funzione è l'indirizzo dell'array.

```
int main() {
    int x[3] = {0,1,2}; // array di int
    for(int i=0; i<3; i++) { // stampa
        printf(«%d\t», x[i]);
    }

    quadrato(x, 3); // funzione
    for(int i=0; i<3; i++) { // stampa
        printf(«%d\t», x[i]);
    }
}

void quadrato(int x[], int n) { // stampa
    for(int i=0; i<n; i++) {
        x[i] = x[i]*x[i];
    }
}
```

# Passaggio di parametri: riferimento

Le modifiche vengono apportate sulla stesso array e quindi sono mantenute anche dopo il completamento della funzione.

```
int main() {
    int x[3] = {0,1,2};
    for(int i=0; i<3; i++) {
        printf(«%d\t», x[i]);
    }

    quadrato(x, 3); // funzione
    for(int i=0; i<3; i++) { // stampa
        printf(«%d\t», x[i]);
    }
}

void quadrato(int x[], int n) {
    for(int i=0; i<n; i++) {
        x[i] = x[i]*x[i];
    }
}
```

stampa:



1 2 3



1 4 9

# Passaggio di parametri

- Il meccanismo che lega valori di parametri formali ai valori di parametri attuali si chiama **passaggio di parametri**. E' una assegnazione eseguita senza che il programmatore la implementi.
- In C esistono due meccanismi per il passaggio di parametri
  - per **valore**
    - copia il valore del parametro attuale nel parametro formale, così la funzione lavora su un' altra locazione di memoria senza modificare la vecchia.
  - per **riferimento**
    - copia l'indirizzo del parametro attuale nel parametro formale, così entrambi si riferiscono alla stessa locazione di memoria. Le modifiche non vengono perse ma vengono propagate anche fuori della funzione.

# Passaggio di parametri: riferimento

Le modifiche quindi non vengono perse, anzi, vengono ereditate dal programma chiamante.

```
int main() {  
    int x = 5;  
    printf("Fuori: %d", x);  
    quadrato(&x);
```

**Fuori: 5**

x=5

```
void quadrato(int y) {  
    y = y*y  
    printf("Dentro: %d", y);  
}
```

**Dentro: 25**

y  
x=25

```
    printf("Fuori: %d", x);  
}
```

x=25

**Fuori: 25**

# Passaggio di parametri:riferimento

- Quando si passa un array ad una funzione, bisogna passare anche la sua dimensione.
- Quando si passa una variabile di tipo primitive, bisogna specificare che si sta copiando l'indirizzo tramite l'operatore **&**.

```
int main() {  
    int x[3] = {0,1,2};  
    for(int i=0; i<3; i++) {  
        printf(«%d\t», x[i]);  
    }  
  
    quadrato(x, 3); // funzione  
    for(int i=0; i<3; i++) { // stampa  
        printf(«%d\t», x[i]);  
    }  
}
```

```
void quadrato(int x[], int n) {  
    for(int i=0; i<n; i++) {  
        x[i] = x[i]*x[i];  
    }  
}
```


```
int main() {  
    int x = 5;  
    printf(“Fuori: %d”, x);  
    quadrato(&x);  
}
```



# Programmazione modulare in C

Definizione della segnatura e implementazione della funzione devono essere compatibili. I tipi di dati dei parametri formali devono coincidere con quelli dei parametri attuali

```
int main() {  
    int x_vet = {0};  
    printf("Inserisci voto:");  
  
    scanf("%d",&x[0]);  
  
    if(promosso(x))  
        printf("Promosso");  
    else  
        printf("Non promosso");  
}  
  
int promosso(int voto) {  
    if(voto>18)  
        return 1;  
    else  
        return 0;  
}
```



*non  
compatibili*

Errore di  
compilazione!

# Programmazione modulare in C

Tuttavia, possiamo pensare che uno stesso sotto-problema possa avere diverse soluzioni e che il sotto-problema può essere caratterizzato da diversi dati di input. Queste possono essere distinte sulla base della lista di parametri. Così invocheremo quella che è più adeguata al caso specifico:

```
int promosso(int voto) {  
    if(voto>18) return 1;  
    else return 0;}
```

```
int promosso(int scritto, int laboratorio) {  
    if((scritto+laboratorio)/2 >18)  
        return 1;  
    else  
        return 0;  
}
```

# Programmazione modulare in C

Tuttavia, possiamo pensare che uno stesso sotto-problema possa avere diverse soluzioni e che il sotto-problema può essere caratterizzato da diversi dati di input. Queste possono essere distinte sulla base della lista di parametri. Così invocheremo quella che è più adeguata al caso specifico:

```
int main() {
    int x;  int y;  int z;
    printf("Inserisci voto:");
    scanf("%d",x);

    printf("Inserisci voto scritto:");
    scanf("%d",y);

    printf("Inserisci voto laboratorio:");
    scanf("%d",z);

    if(promosso(x))
        printf("Promosso ad analisi matematica");
    else
        printf("Non promosso ad analisi matematica");

    if(promosso(y,z))
        printf("Promosso a laboratorio");
    else
        printf("Non promosso a laboratorio");
}
```

# Programmazione modulare in C

- Nel file principale di un programma, in C, la segnatura di una funzione precede il main(), mentre la implementazione lo segue.
- Questo consente al compilatore di verificare se la funzione promosso() venga invocata correttamente.

```
int promosso(int voto)

int main() {
    int x = 0;
    printf("Inserisci voto:");

    scanf("%d",&x);

    if(promosso(x))
        printf("Promosso");
    else
        printf("«Non promosso»");
}

int promosso(int voto) {
    if(voto>18)
        return 1;
    else
        return 0;    }
```

# Funzione e Procedura

- Nella programmazione, le funzioni devono computare espressioni al fine di produrre un valore, mentre le procedure sono astrazione sulle istruzioni e devono eseguire comandi al fine di cambiare lo stato dei dati, **senza restituire alcun valore**.
- In C una procedura è una funzione che non restituisce valori ed ha come tipo di valore di ritorno **void**.

```
int main() {  
    x = 20;  
    stampaNumero(x); //procedura  
}
```

```
void stampaNumero(int numero) {  
    printf("%d", numero);  
    // non ha istruzione return  
}
```

# Esercizio

- *Facendo uso di funzioni/procedure, scrivere un programma che, usando una matrice, acquisisca le calorie assunte in una settimana da un insieme di 5 individui e ne identifichi il massimo per individuo e complessivo.*

# Esercizio

- *Scrivere un programma che acquisisca e visualizzi i dati di una collezione di 5 studenti, ciascuno descritto da <nome, cognome, data di nascita, voti di esami>*
- *Considerare casi limite per data di nascita e voti esami.*
- *Calcolare e visualizzare la media dei voti di esami per ciascuno studente*
- *Visualizzare lo studente con la media esami più alta*

# Esercizio

*Facendo uso di funzioni/procedure, scrivere un programma che permette di*

- caricare un vettore di conti correnti bancari: ogni conto è descritto in termini del numero del conto, il titolare del conto (una variabile di tipo Persona) e del saldo;*
- stampare l'intero archivio;*
- dato il numero di conto effettuare il prelievo da tale conto*
- dato il numero di conto effettuare l'accredito su tale conto*
- dato un numero di conto stampare le informazioni del conto*

*Considerare casi limite per prelievo, accredito e numero di conto in input.*



# Programmazione modulare in C: Limiti

Analizziamo la dichiarazione di tipo che C consente di definire.

```
typedef struct {  
    char nome;  
    int numero;  
    float saldo;  
    int carte[10];  
} conto_corrente;
```

consente di

- Stabilire che **conto\_corrente** è un identificatore di tipo
- Associare una rappresentazione a **conto\_corrente** espressa mediante tipi concreti già disponibili nel linguaggio.

Tuttavia, le operazioni associate al nuovo tipo

**conto\_corrente** sono tutte quelle che il linguaggio ha già previsto per il tipo struct (e.g., assegnazione **:=** e selezione di campi **c.nome**).

# Programmazione modulare in C: Limiti

→ Il programmatore **non** può definire nuovi operatori specifici da associare al tipo.

```
conto_corrente a,b;  
a+b  Cosa sommerà? Quale risultato  
      restituirà?
```

- Supponendo che i due conti correnti siano intestati alla stessa persona, egli ha intenzione di unificare i due in un unico conto. L'operatore **+** applicato sui due campi nome (a,b) potrebbe restituire un valore inatteso anzichè il nome della medesima persona (a.nome=b.nome).
- In C, gli operatori applicabili ad un tipo strutturato sono quelli comuni per tutte le variabili struct e per i loro campi.

# Programmazione modulare in C: Limiti

- Il programmatore **non** può definire nuovi operatori specifici da associare al tipo.

`conto_corrente a,b;`

`a+b` Cosa sommerà? Quale risultato  
restituirà?

- Inoltre, gli operatori che agiscono a livello dei singoli campi violano la astrazione. Infatti, l'utilizzatore, che dovrebbe conoscere solo il “**cosa**”, è consapevole anche della **rappresentazione** del tipo `conto_corrente` – “**come**” (sa che è un record e quali sono i campi) ed è in grado di operare mediante operatori non specifici del dato.

# Programmazione modulare in C: Limiti

- Questo significa che in C **non** possiamo fare modularizzazione sui tipi strutturati rispetto agli operatori. Conseguentemente, la funzionalità di alcuni operatori sarà affidata ai costrutti di C dove è possibile fare modularizzazione, cioè funzioni e procedure.

```
typedef struct {  
    char nome;    int numero;  
    float saldo;  
    int carte[10];  
}conto_corrente;
```

```
conto_corrente somma (conto_corrente a,conto_corrente b) {  
    conto_corrente c;  
    c.nome=a.nome;  
    c.numero=a.numero;  
    c.saldo=a.saldo+b.saldo  
    return c;  
}
```

# Programmazione modulare in C

- Quindi avremo:

```
typedef struct {
    char nome;    int numero;
    float saldo;
    int carte[10];
}conto_corrente;

conto_corrente somma (conto_corrente a,conto_corrente b);

int main() {
    conto_corrente a={'a',20,20.5};
    conto_corrente b={'b',30,30.5};
    conto_corrente c=somma(a,b);
    printf("%f",c.saldo);

    conto_corrente somma (conto_corrente a,conto_corrente b) {
        conto_corrente c;
        c.nome=a.nome;
        c.numero=a.numero;
        c.saldo=a.saldo+b.saldo;
        return c;}
```

# Programmazione modulare in C

- Ma anche:

```
typedef struct {  
    char nome;    int numero;  
    float saldo;  
    int carte[10];  
}conto_corrente;
```

```
conto_corrente somma (conto_corrente a,conto_corrente b);
```

```
int main() {  
    conto_corrente a={'a',20,20.5};  
    conto_corrente b={'b',30,30.5};  
    printf("%f",somma(a,b).saldo);
```

Cosa cambia?  
Ci ritorniamo

```
conto_corrente somma (conto_corrente a,conto_corrente b) {  
    conto_corrente c;  
    c.nome=a.nome;  
    c.numero=a.numero;  
    c.saldo=a.saldo+b.saldo;  
    return c;}
```

# Programmazione modulare in C

- Più generalmente, possiamo pensare di definire un set di funzioni e procedure che interpretano la funzionalità da offrire nel *dominio del problema* da dove definiamo il tipo strutturato :

```
typedef struct {  
    char nome;    int numero;  
    float saldo;  
    int carte[10];  
}conto_corrente;
```

```
conto_corrente somma (conto_corrente a,conto_corrente b);  
conto_corrente differenza (conto_corrente a,conto_corrente b);  
conto_corrente deposito (conto_corrente a, float ammontare);  
conto_corrente prelievo (conto_corrente a, float ammontare);  
void stampa_saldo (conto_corrente a);  
int main() {  
    ...  
}
```

# Programmazione modulare in C

- Tuttavia, molti di queste funzioni (in questo esempio, *stampa\_saldo()* e *deposito()*) hanno necessità di conoscere la rappresentazione e la organizzazione dei campi.

```
typedef struct {
    char nome;   int numero;
    float saldo;
    int carte[10];
}conto_corrente;

int main() {
    ...
}

void stampa_saldo (conto_corrente a){
    printf("%f",a.saldo);
}

conto_corrente deposito (conto_corrente a, float ammontare){
    a.saldo=a.saldo+ammontare;
}

}
```

- ...il che potrebbe introdurre problemi



# Programmazione modulare in C: Limiti

- Supponiamo di voler estendere il *conto\_corrente* in modo da associare a ciascuna carta di credito il *saldo* relativo, anzichè avere un unico saldo:

→ Dovremmo modificare la struct di *conto\_corrente* ed conseguentemente gli operatori che accedevano alla rappresentazione, come *deposito()*, *stampa\_saldo()*, che per loro natura potrebbero non risentire della estensione.

```
• typedef struct {  
    int id;  
    int saldo;;  
}carta;  
  
typedef struct {  
    char nome;    int numero;  
    carta carte[10];  
}conto_corrente;
```

- Questo rende difficoltosa la manutenzione delle soluzioni progettate. È inappropriata per lo sviluppo di soluzioni a problemi complessi.

# Programmazione modulare in C

- Soluzione: non si deve accedere direttamente alla rappresentazione di un dato, qualunque esso sia, ma solo attraverso un insieme di **operazioni** considerate **lecite** che **agiscono al livello di campo** e che sono differenti dalle funzionalità del dominio.

# Programmazione modulare in C

- Un cambiamento nella rappresentazione del tipo si ripercuoterà solo sulle operazioni lecite, che potrebbero subire delle modifiche, mentre non inficierà il codice che utilizza il tipo strutturato. Così si potrebbe pensare ad un'operazione lecita sarà *get\_saldo()* applicata su *conto\_corrente* con una sola struct, mentre *stampa\_saldo()* invocherà l'operazione lecita *get\_saldo()*:

```
typedef struct {
    char nome;    int numero;
                float saldo;
                int carte[10];
}conto_corrente;

float get_saldo(conto_corrente a){
    return a.saldo;}

void stampa_saldo (conto_corrente a){
    printf("%f",get_saldo());
}
```

# Programmazione modulare in C

- Quando l'estensione prevederà due struct, modificheremo solo l'operazione lecita *get\_saldo()*, mentre la funzione *stampa\_saldo()* resta immutata:

```
typedef struct {  
    int id;  
    int saldo;  
}carta;
```

```
typedef struct {  
    char nome;    int numero;  
    carta carte[10];  
}conto_corrente;
```

```
float get_saldo(conto_corrente a){  
    float tmp_saldo=0;  
    for(int i=0;i++;i<10)  
        tmp_saldo+=a.carte[i].saldo;  
    return a.saldo;}  

```

```
void stampa_saldo (conto_corrente a){  
    printf("%f",get_saldo());  
}
```

# Programmazione modulare in C

- Nella rappresentazione di `conto_corrente` a due struct, modificheremo solo l'operazione lecita `get_saldo()`, mentre la funzione `stampa_saldo()` resta immutata:

```
typedef struct {  
    int id;  
    int saldo;  
}carta;
```

```
typedef struct {  
    char nome;    int numero;  
    carta carte[10];  
}conto_corrente;
```

```
float get_saldo(conto_corrente a){  
    float tmp_saldo=0;  
    for(int i=0;i++;i<10)  
        tmp_saldo+=a.carte[i].saldo;  
    return a.saldo;}  

```

```
void stampa_saldo (conto_corrente a){  
    printf("%f",get_saldo());  
}
```

# Programmazione modulare in C

Ricordiamo:

```
typedef struct {  
    char nome;    int numero;  
    float saldo;  
    int carte[10];  
}conto_corrente;
```

```
conto_corrente somma (conto_corrente a,conto_corrente b);
```

```
int main() {  
    conto_corrente a={'a',20,20.5};  
    conto_corrente b={'b',30,30.5};  
    printf("%f",somma(a,b).saldo); Cosa cambia?
```

```
conto_corrente somma (conto_corrente a,conto_corrente b) {  
    conto_corrente c;  
    c.nome=a.nome;  
    c.numero=a.numero;  
    c.saldo=a.saldo+b.saldo;  
    return c;}
```

# Programmazione modulare in C

- Le funzioni, in quanto astrazioni di espressioni, possono comparire ovunque occorra un'espressione.
- Pertanto, esse compaiono alla destra di operazioni di **assegnazioni**, ma anche al **posto di parametri effettivi** nelle **chiamate** di altre funzioni (o procedure), laddove un argomento potrebbe essere calcolato mediante una espressione.

$$y := f(y, \text{power}(x, 2))$$

La funzione  $f$  ha due parametri passati per valore.

# Cittadini di prima classe

- In generale, in programmazione una qualunque entità si dice **cittadino di prima classe** quando non è soggetta a restrizioni nel suo utilizzo.



- I valori di un **tipo primitivo** sono un esempio emblematico di cittadini di prima classe in tutti i linguaggi



# Cittadini di prima classe

- In C, ed in tutti i linguaggi, possiamo avere, **cittadini** di
- ♦ **terza classe**, se possono essere solo **chiamate** (invocazione)
  - ♦ **seconda classe**, se possono essere passate come **argomenti**
  - ♦ **prima classe**, se, oltre agli usi di terza e seconda classe, possono essere anche restituite come **risultato della chiamata** di altre funzioni o possono essere assegnate come **valore** a una variabile

L'uso di una funzione al posto di un parametro attuale è consentito perchè in C le funzioni sono (almeno) **cittadini di terza classe**. Vedremo che le funzioni sono cittadini di seconda classe in C.

# Programmazione modulare in C

- In C, oltre alla modularizzazione con funzioni e procedure, è possibile modularizzare con collezioni di funzioni e procedure, realizzando i cosiddetti **moduli**.
- Un modulo corrisponde ad un file sorgente, che è parte costituente di un sistema.
- Definito attraverso **header files** e **librerie statiche**.

# Header files

- Contengono le signature di funzioni e procedure, la cui implementazione è riportata in un file gemellare
- *file.c* implementazione di signature dichiarate nell'header *file.h*
- Le funzioni sono collezionate nei moduli in base al loro scopo: ad esempio, funzioni per input/output, funzioni per operazioni matematiche
- Conseguentemente, la struttura dei file sorgenti cambia:

# Header files

//main.c

```
void f(int x); // segnatura
```

```
int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

```
void f(int x) { // implementazione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

//function.h

```
void f(int x); // segnatura
```

//function.c

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

```
void f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

```
#include "function.h" // include
int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

# Header files

```
//function.h
```

```
void f(int x); // segnatura
```

```
//function.c
```

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

```
void f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

```
#include "function.h" // include
int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

- Nel main includiamo il nuovo header file, così come se fosse una delle librerie standard del C, e possiamo utilizzarne le funzioni.
- L'implementazione può includere header file predefiniti in C.
- L'inclusione di librerie fatte dal programmatore (non quelle standard) deve essere fatta tramite virgolette, non parentesi angolari!
- Gli header files **non** possono includersi reciprocamente.

# Header files

Ogni header file è diviso in sette parti:

- prologo
- guardia
- direttive di inclusione
- costanti
- tipi di dato
- variabili globali
- prototipi di funzione

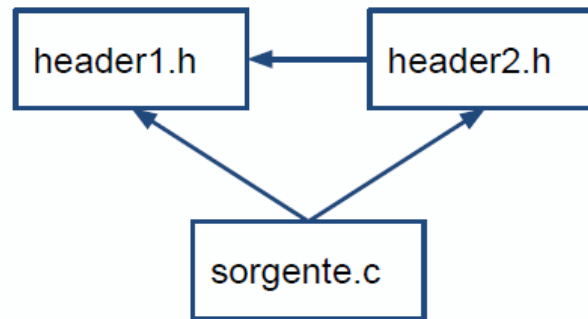
# Prologo

- Si tratta di un “commento” e descrive a cosa serve il file e che tipo di informazioni contiene.
- Le informazioni sono Autori, Data, Numero di versione, Riferimenti e Link esterni, Copyright

```
/*  
 * CUnit - A Unit testing  
 * framework library for C.  
 * Copyright (C) 2004-2006  Jerry  
 * St.Clair  
 *  
 * This library is free software; ...  
 * License as published by the  
 * Free Software Foundation;  
 * ...  
 * 11-Aug-2004  Initial  
 * implementation of basic test  
 * runner interface. (JDS)  
 */
```

# Guardia

- Il vincolo per cui gli header non possono includersi reciprocamente non esclude che un header possa includere diversi altri header, quindi ci possono essere dipendenze multiple.



- La guardia serve a inserire condizioni al fine di escludere dipendenze multiple. Ad esempio, nel seguito si specifica di continuare a compilare a CONDIZIONE che il file CUNIT\_BASIC\_H\_SEEN non sia stato già definito.

```
#ifndef CUNIT_BASIC_H_SEEN
#define CUNIT_BASIC_H_SEEN
... codice
#endif
```



# Inclusioni

- Un file header+implementazione può invocare una funzione di librerie standard C
- Di seguito, si include <stdio.h> perché si invoca printf().

```
/* System includes */  
#include <stdio.h>  
#include <stdlib.h>
```

```
/* Local includes */  
// Unit test framework  
#include "CUnit/Basic.h"  
// libreria di manipolazione  
matrici  
#include "math/matrix.h"
```

# Definizioni

- Si possono anche includere definizioni di costanti e macro.
- E' possibile usare la macro **#ifndef** per evitare ridefinizioni di costanti già definite nel programma.

```
#define CU_VERSION "2.1-2"

#define
CU_MAX_TEST_NAME_LENGTH 256

#ifndef CU_TRUE
    #define CU_TRUE 1
#endif

#  define CU_MAX(a,b)
    (((a) >= (b)) ? (a) : (b))
```

# Variabili

- Le variabili definite negli header hanno visibilità globale e sono visibili da tutte le funzioni o da tutti i moduli.
- Questo spesso può essere uno svantaggio.

```
int global_variable;  
// visibile a livello  
// globale. EVITARE
```

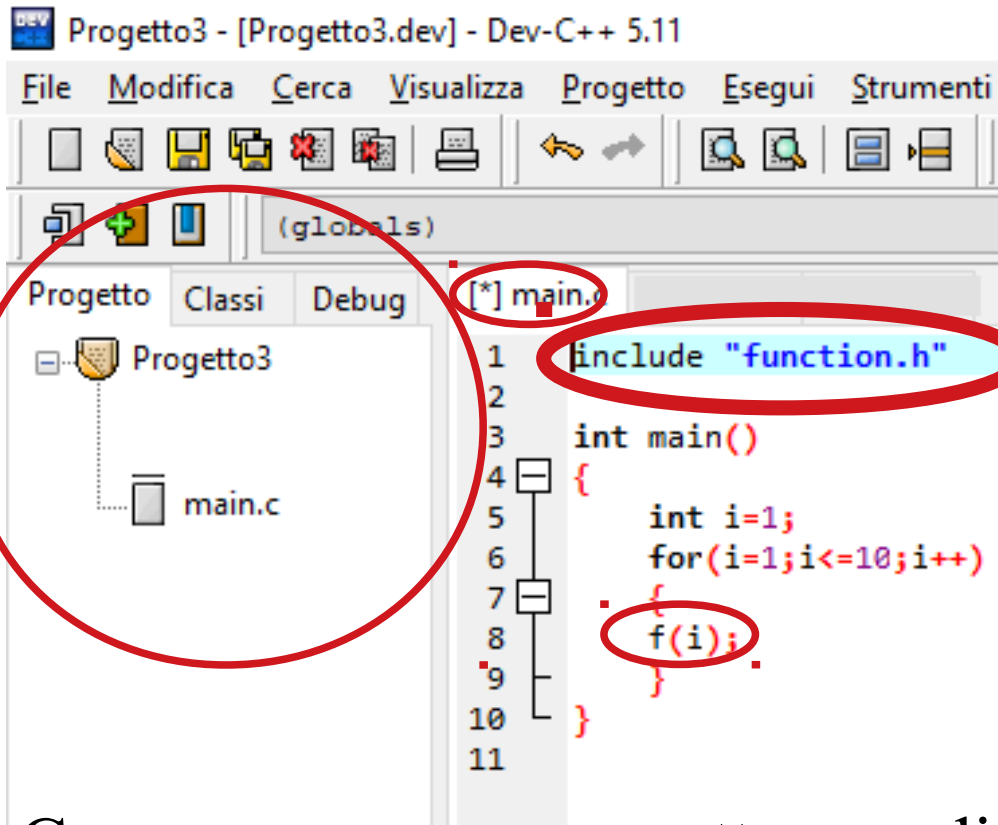
```
static int mod_var;  
// visibile solo  
// a livello di  
// modulo
```



# Prototipi

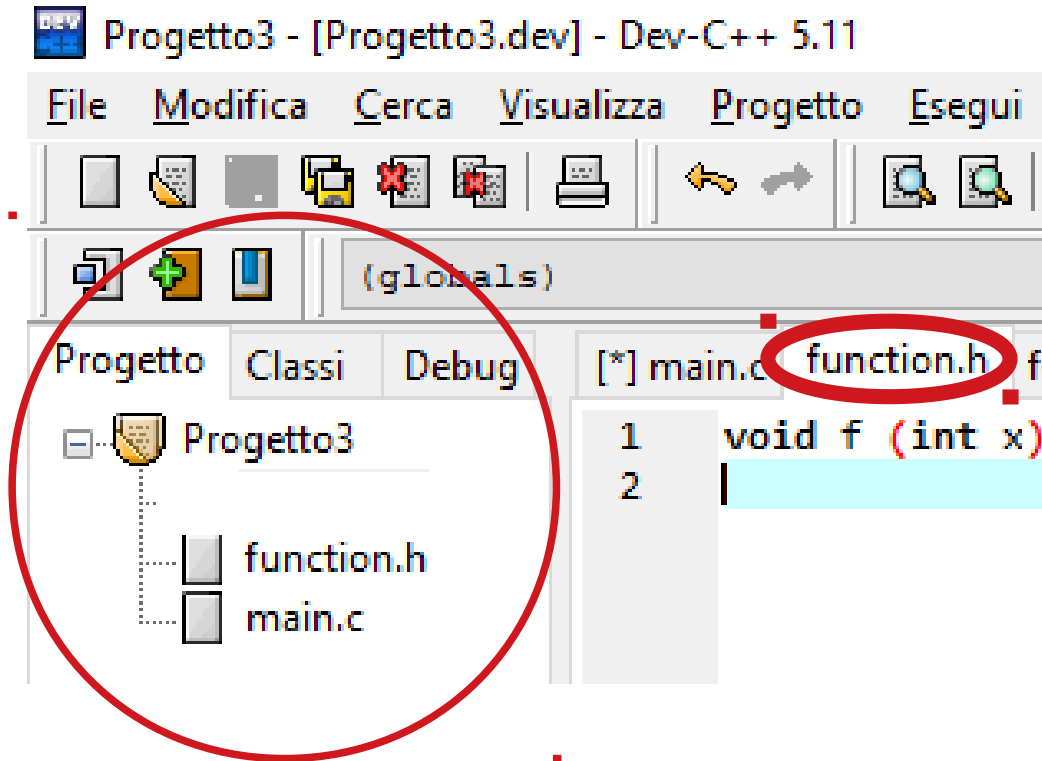
- L'elemento minimo che un header debba contenere è il prototipo di una funzione.

# Creazione degli Header files: IDE Dev



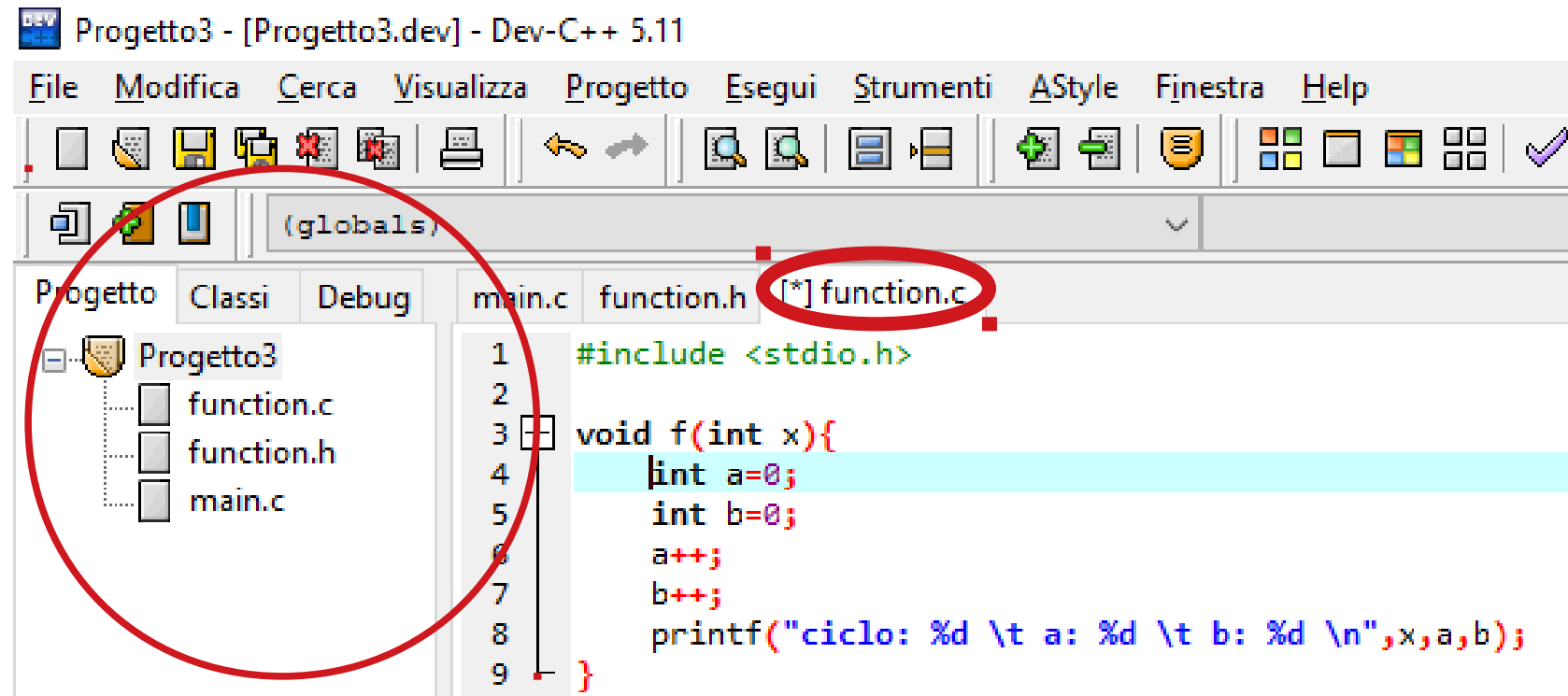
Creare un nuovo progetto e codificare il modulo principale, quello della funzione *main()*. Questo modulo includerà l'header file delle funzioni da includere e le invocherà nel corpo di codice.

# Creazione degli Header files: IDE Dev



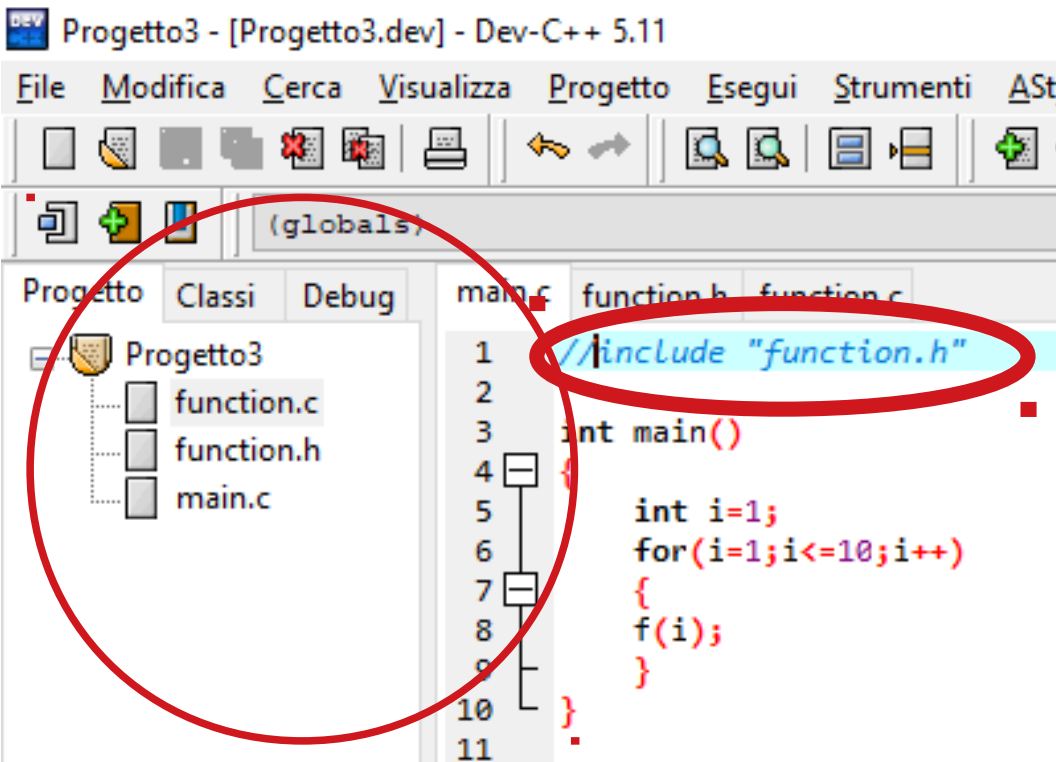
Nel medesimo progetto, creare un nuovo file con estensione *.h* in cui dichiararemo le signature delle funzioni da invocare.

# Creazione degli Header files: IDE Dev



Nel medesimo progetto, creare un nuovo file con estensione *.c* (con medesimo nome del file *.h*) in cui codificheremo le funzioni da invocare.

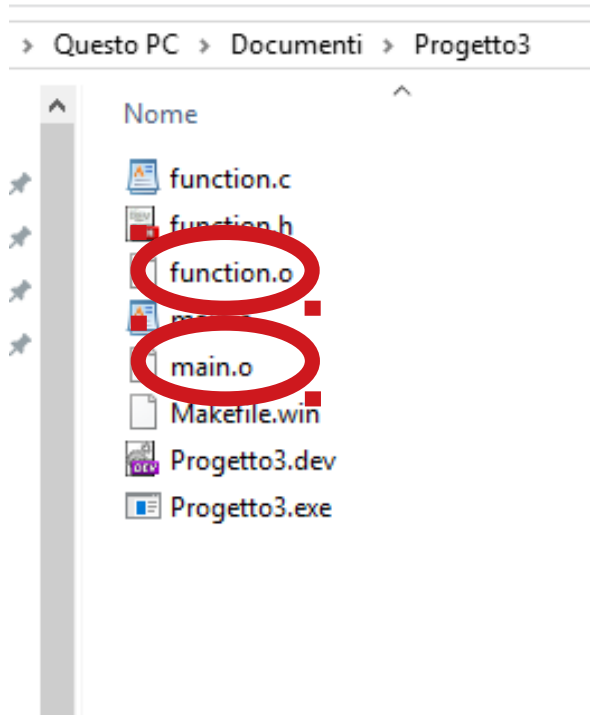
# Creazione degli Header files: IDE Dev



In alcuni IDE (ad esempio DEV), la ricerca degli header files da includere in un progetto parte automaticamente dalla cartella del progetto. Quindi, specificare di includere un header file dello sviluppatore, con lo stesso nome, fornire lo stesso nome per due header: **errore!**

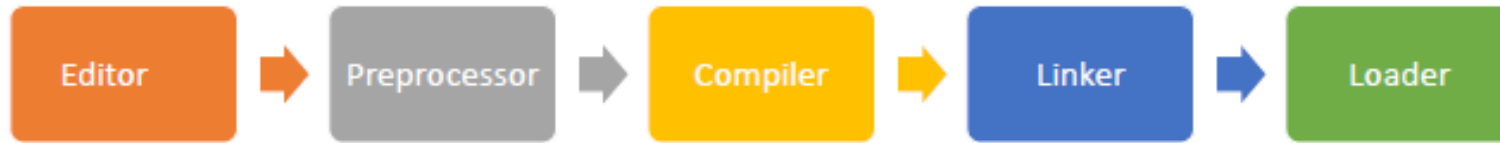


# Creazione degli Header files: IDE Dev



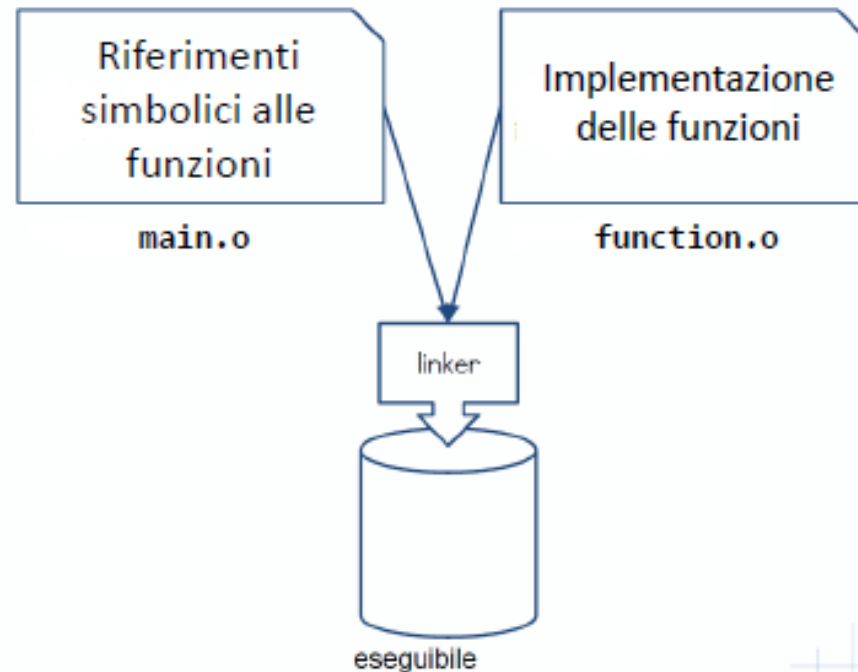
Si noti che quando si compila un intero progetto (opzione *Riassembla Tutto*), si genera il codice oggetto del modulo principale e dei file di implementazione.

# Creazione degli Header files



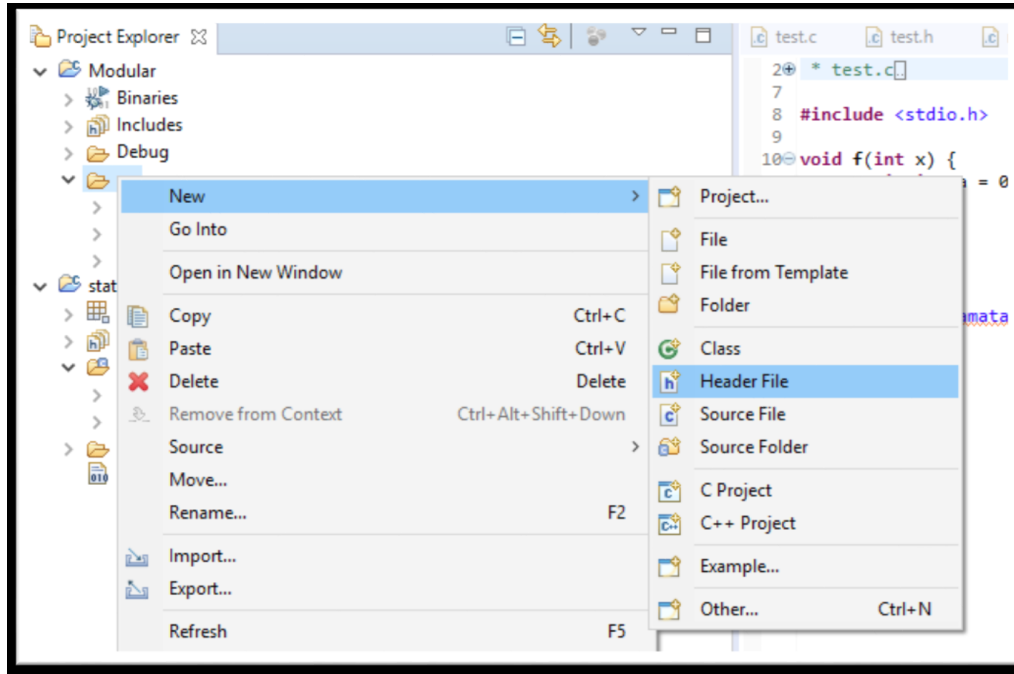
Difatti, il preprocessore, risolve la direttiva `#include`, includendo l'header file dello sviluppatore. Quindi, genererà codice oggetto anche per moduli esterni e per il modulo principale imporrà codice esterno.

# Creazione degli Header files



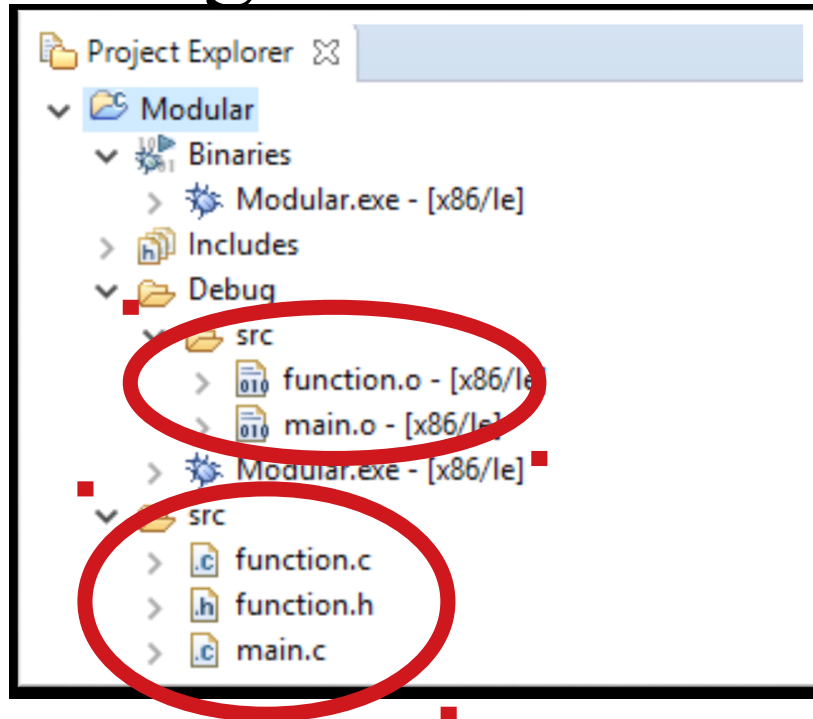
Infine, il linker risolve i riferimenti simbolici: il file *main.o* “sa” che esiste una funzione  $f$  perché è indicato nel file header. La funzione però è implementata nel file *function.o*. Quindi, il linker si occupa di linkare i riferimenti simbolici con la reale implementazione della funzione. Infine, produrrà l'eseguibile aggregando i diversi file oggetto.

# Creazione degli Header files: IDE Eclipse



Anche in Eclipse, l'header file ed il file di implementazione possono essere creati nello stesso progetto.

# Creazione degli Header files: IDE Eclipse



Conseguentemente, saranno generati codici oggetto per ogni file di implementazione.

# Scope e Ambito d'azione

- Lo scope o visibilità è il frammento di codice in cui un dato o funzione è visibile, cioè l'insieme di blocchi di codice in cui quel dato/funzione può essere vista ed in cui ha effetto.
- Ad esempio, una variabile ha scope nel blocco A in cui viene dichiarata ed in tutti i blocchi interni ad A, a meno di trovare nuove dichiarazioni che usano l'identificatore della variabile.
- Ci sono 4 possibili scope
  - File scope
  - Function scope
  - Block scope
  - Function Prototype scope

# Scope e Ambito d'azione

- File scope:
  - identificatori definiti fuori dalle funzioni sono visibili in **tutto** il file di codice
  - usato per dichiarare **prototipi di funzione** (richiamati ad ogni invocazione della medesima funzione) e **variabili globali** (usati in tutto il codice sorgente)
- Function scope
  - identificatori definiti nel corpo di una funzione visibili solo nella medesima implementazione della funzione, salvo ridefinizioni.
  - usato per **variabili locali**
- Block scope
  - identificatori dichiarati in un blocco visibili solo in quel blocco
  - usato per **contatori** e **condizioni** su costrutti iterativi
- Prototype scope
  - identificatori dichiarati in un prototipo sono visibili nella segnatura medesima

# Scope e Ambito d'azione

```
void function(int var1); // var1 = visibilità nel prototipo
```

```
int var2;//var2 = visibilità globale
```

```
int main() {  
    int var3=0; // var3 = visibilità locale  
    if(var3==0) {  
        int var4 = 1; // var4 = visibilità nel blocco  
    }  
}
```



# Scope e Ambito d'azione

```
void function(int var1); // var1 = visibilità nel prototipo
```

```
int var2;//var2 = visibilità globale
```

```
int main() {  
    int var3=0; // var3 = visibilità locale  
    var2=1    //ok  
    if(var3==0) {  
        int var4 = 1; // var4 = visibilità nel blocco  
    }  
}
```

# Scope e Ambito d'azione

```
void function(int var1); // var1 = visibilità nel prototipo
```

```
int var2;//var2 = visibilità globale
```

```
int main() {  
    int var3=0; // var3 = visibilità locale  
    if(var3==0) {  
        int var4 = 1; // var4 = visibilità nel blocco  
        var3=2 //ok  
    }  
}
```

# Scope e Ambito d'azione

```
void function(int var1); // var1 = visibilità nel prototipo
```

```
int var2; // var2 = visibilità globale
```

```
int main() {  
    int var3=0; // var3 = visibilità locale  
    if(var3==0) {  
        int var4 = 1; // var4 = visibilità nel blocco  
    }  
    var1=3; //NO  
    var4=4; //NO  
}
```

# Scope e Ambito d'azione

```
void function(int var1); // var1 = visibilità nel prototipo
```

```
int var2;//var2 = visibilità globale
```

```
int main() {  
    int var3=0; // var3 = visibilità locale  
    if(var3==0) {  
        int var4 = 1; // var4 = visibilità nel blocco  
    }  
    var1=3; //NO  
    int var4=4; //ok, ridefinizione. var4 del blocco  
                precedente non è più visibile  
}
```

# Permanenza in memoria

- In C esistono specificatore per determinare la permanenza in memoria di variabili:
  - Auto
  - Extern
  - Static
  - Register

# Permanenza in memoria

- Auto:
  - durata in memoria pari alla esecuzione del blocco in cui la variabile è usata
  - usata per variabili locali
- Extern
  - durata in memoria pari all'intera esecuzione del file
  - usata per variabili globali
- Static
  - durata in memoria pari alla esecuzione di una funzione, ma il valore non è distrutto al termine della funzione
- Register
  - durata breve in memoria, tipicamente per variabili usate molto spesso per un breve tempo. Ad esempio, contatori di iterazioni memorizzati in registri ad alta velocità. Non più usato.

# Permanenza in memoria

```
int main() {  
    for(register int i=1; i<10; i++)  
    {  
        f(i);  
    }  
}
```

```
int f(int x){  
    static int a=0;  
    int b=0;  
    a++;  
    b++;  
    printf("n. %d  a=  %d  b= %d ",x,a,b);  
}  
. . .
```

# Permanenza in memoria

n.1	a=1	b=1
n.2	a=2	b=1
n.3	a=3	b=1
n.4	a=4	b=1
n.5	a=5	b=1
n.6	a=6	b=1
n.7	a=7	b=1
n.8	a=8	b=1
n.9	a=9	b=1
n.10	a=10	b=1

- La variabile statica *a* resta in memoria, quindi ad ogni invocazione della funzione il valore continua a essere incrementato.
- La variabile locale *b* viene azzerata (e inizializzata) ad ogni invocazione della funzione *f*



# Esercizio

Scrivere la libreria Matematica che include gli operatori per il calcolo del 1-minimo tra due interi, 2-massimo tra due interi, 3-media tra due interi, 4-valore assoluto di un numero.

Realizzare un prototipo di macchina calcolatrice che permetta all'utente di scegliere l'operazione da effettuare (le opzioni possibili includono le funzioni incluse in Matematica e l'opzione Esci). In base all'operazione scelta (diversa da Esci) si richiede l'inserimento degli input, si visualizza il risultato e si mostra nuovamente il menu delle scelte.

# Esercizio

- Scrivere la libreria Studente che includa gli operatori per 1-verbalizzazione di un esame 2- calcolo media esami, 3-calcolo voto massimo, 4-calcolo esami restanti
- Lo studente è descritto da Matricola, Elenco esami piano di studi, Elenco esami sostenuti, Elenco voti di esami sostenuti.