

strutturare dati e codice

Puntatori e passaggio parametri

† Tipo di dati	<code>int * Pi ;</code>
† Op. dereferenziazione	<code>j = *Pi ;</code>
† Op. indirizzo	<code>Pi = &i ;</code>
† By value	<code>int f(int i) ;</code> <code>a = f(b) ;</code>
† By address	<code>int f(int * Pi) ;</code> <code>a = f(&b) ;</code>
† By reference	<code>int f(int & Pi) ;</code> <code>a = f(b) ;</code>

Ambito di validità (*scope*)

- † ... dalla definizione alla chiusura del blocco
 - in C++ definizione ovunque nel blocco
- † variabile locale - def. interna al corpo di una function
- † variabile globale - def. esterna ai corpi fun
 - extern - globale in gruppo di file (external linkage)
 - static - globale in un singolo file (internal linkage)

typedef

- † non definizione di un nuovo tipo ma *alias*

```
typedef oldname newname
```

- † talvolta usato per puntatori

```
typedef int* IntPtr ;  
IntPtr Pi, Pj ;
```

- † ma soprattutto per le strutture (in C)

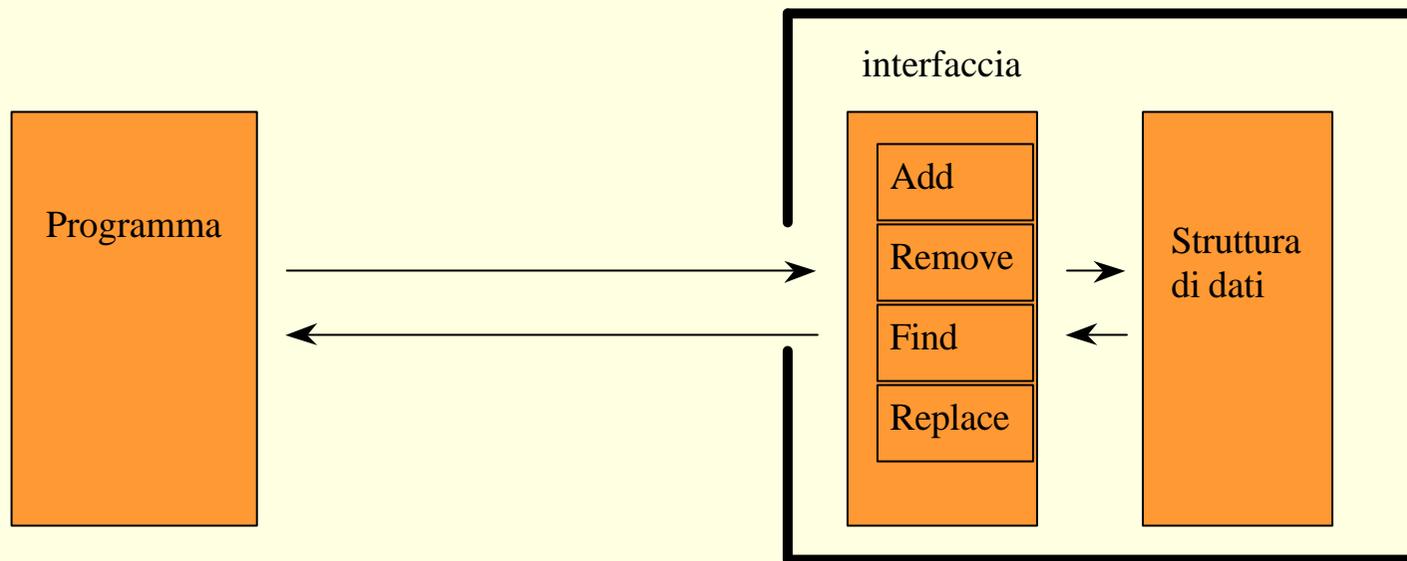
Strutturare i dati

- † Esigenza nata dalle applicazioni gestionali
 - in campo scientifico bastano gli array (Fortran)
- † Raccogliere un gruppo di variabili in una struttura comune
- † Definizione di nuovi tipi di dati

Tipi di dati astratti (ADT)

† Per potere scrivere del codice efficiente e modulare occorre per ciascun modulo/oggetto/componente

- concentrarsi su **cosa** deve fare e non su **come** lo fa
- identificare i dettagli che si possono nascondere nel modulo (*information hiding*)
- definire una interfaccia che isoli la struttura dei dati dal programma che la usa



Array, Union, Struct

- il C offre vari ausili per la strutturazione dei dati
 - array, union e **struct**
- il C++ oltre alle **strutture** del C, anche le **classi**

- `array` : collezione di oggetti dello stesso tipo
- `union` : sovrapposizione di dati
- `struct` : collezione di oggetti di tipi arbitrari

esempio di struct

- consideriamo i caratteri di un word processor. Oltre al carattere hanno un font (normale, *italico*, **grassetto**..) ed un corpo (size = 11, 12, 14 ..)
- in C è possibile memorizzare la struttura dei dati “caratteri di word processor” con una struct :

```
struct wp_char {  
    char    wp_cval;  
    int     wp_font;  
    int     wp_size;};
```

dichiarazione



```
struct wp_char  a1,a2;
```

definizione di a1 e a2



Typedef & Struct

- in C non è possibile usare direttamente il nome di una struct nella definizione di variabili di quel tipo

```
struct wp_char {char ; int ; int ;};
```

```
...
```

```
struct wp_char a1,a2;
```

- con Typedef si definisce un nuovo nome

```
Typedef struct {char ; int ; int ;} wp_char ;
```

```
...
```

```
wp_char a1,a2;
```

- in C++ non è necessario ...

SimpleStruc 1,2,3

Struct & Pointer

- Utilizzo:

```
struct wp_char a1, a2;  
a1.wp_cval = ' f';  
a1.wp_font = 1;  
a1.wp_size = 10;
```

```
a2 = a1
```

• operatore assegnazione

assegnazione:
unica operazione tra due struct

- Puntatori a struct

```
struct wp_char *wp_p;  
wp_p = &a1.wp_cval;  
(*wp_p).wp_cval = 'x';  
wp_p->wp_cval = 'x';
```

-> operatore dereferenziazione

Struct Structure

- i membri di una `struct` sono allocati in memoria nell'ordine di apparizione nella definizione
- il nome di una `struct` corrisponde al puntatore all'indirizzo del suo primo membro

```
-struct wp_char item;           // esempio precedente //  
-(char *)item == &item.wp_cval;
```

- è possibile creare array di `struct` oppure usare `struct` come membri di altre `struct`
- una `struct` non può contenere un esempio di se stessa come un suo membro (ma un puntatore si)

Definizione di ADT

- † Oltre alla struttura dei dati occorre definire delle operazioni sui dati
- † definizione della struttura in header file
- † definizione delle operazioni in library file
- † esempio - Data : gestione di date
- † esempio - Stash : un deposito generico di dati

Versione C-like: Data

```
struct data {int giorno;  
            int mese; int anno;};
```

...

```
void set_data(data*, int, int, int);
```

```
void next_data(data* );
```

```
void print_data(data* );
```

...

```
data oggi;
```

...

- non ci sono connessioni specifiche tra gli attributi della struttura ed i suoi metodi
- occorre passare ad ogni metodo il puntatore alla struttura

Versione C-like: CStash (1)

† Definizione della Struct

- size - dimensione di ogni elemento
- quantity - quanti elementi ci stanno
- next - il prossimo elemento inseribile
- storage - blocco di byte contigui
(unsigned char *) per contenere i dati

CLib.h

Versione C-like: CStash (2)

- † Definizione delle operazioni della ADT
 - initialize - valori iniziali (size)
 - add - aggiunta di elemento al deposito
 - fetch - preleva l'elemento indicato
 - count - numero di elementi (implementation hiding)
 - inflate - aumenta le dimensioni
 - cleanup - libera la memoria
- † tutte le funzioni devono ricevere un puntatore alla struct su cui operano

Versione C-like: CStash (3)

Gli elementi puntati non possono essere cambiati, il puntatore si

```
int add(CStash* s, const void* element) {
    if(s->next >= s->quantity) //Enough space left?
        inflate(s, increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = s->next * s->size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < s->size; i++)
        s->storage[startBytes + i] = e[i];
    s->next++;
    return(s->next - 1); // Index number
}
```

Indice con base 0

Versione C-like: CStash (4)

```
void inflate(CStash* s, int increase) {
    int newQuantity = s->quantity + increase;
    int newBytes = newQuantity * s->size;
    int oldBytes = s->quantity * s->size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = s->storage[i]; // Copy old to new
    delete [](s->storage); // Old storage
    s->storage = b; // Point to new memory
    s->quantity = newQuantity;
}
```

allocazione dinamica della memoria in C++

var = new type	ritorna puntatore a type
b = new char[100]	b puntatore a primo elem. array
delete var	libera memoria
delete []array	libera l'intero array

Versione C-like: commenti

- † Ogni function deve ricevere il puntatore alla struct su cui opera
- † Inizializzazione esplicita
- † Unico spazio di nomi per le function (librerie di produttori diversi ...)

member functions

† = inserire le function *nella* struttura dati

```
struct data {  
    int giorno; int mese; int anno;  
    void set (int, int, int);  
    void get (int*, int*, int*);  
    void next();  
    void print();  
};
```

member functions

† = inserire le function *nella* struttura dati

```
struct Stash {
    int size; // Size of each space
    int quantity; // Number of storage spaces
    int next; // Next empty space
    unsigned char* storage;

    void initialize(int size);
    void cleanup();
    int add(const void* element);
    void* fetch(int index);
    int count();
    void inflate(int increase);
};
```

Note sulle member functions

- † Non è più necessario il riferimento esplicito negli argomenti o nelle istruzioni (*this*)
- † Lo spazio di nomi è delimitato dalla struct
- † Copia unica del codice (non dei dati!)
Stash A, B, C ;
- † header file - definizione completa dell'ADT
incapsulare

Definizione di member function

- poiché strutture diverse possono avere metodi interni con lo stesso nome, occorre una notazione speciale per definire un metodo

```
void data::next()  
{ giorno++;  
  if (giorno > 28) {  
    // implementare l'algoritmo difficile  
  }  
}
```

in una Member Function, gli attributi della struct sono usati direttamente, senza dovere dichiarare esplicitamente l'oggetto su cui operano: è chiaro che si lavora con gli attributi dell'oggetto chiamante

Definizione di member function

```
void Stash::initialize(int sz) {  
    size = sz;  
    quantity = 0;  
    storage = 0;  
    next = 0;  
}
```

`::` operatore di risoluzione dell'ambito
(scope resolution operator) associa il tipo struct alla function

```
void initialize(CStash* s, int sz) {  
    s->size = sz;  
    s->quantity = 0;  
    s->storage = 0;  
    s->next = 0;  
}
```

Uso di member function

- † Analogia sintattica con l'uso dei dati contenuti in una struct

```
intStash.initialize(sizeof(int));  
for(int i = 0; i < 100; i++)  
    intStash.add(&i) ;
```

- † una struct con member function è (quasi) un oggetto
- † lo spazio occupato è lo stesso di una struct con solo dati

Uso di member function

- bisogna passare al metodo interno gli argomenti di tipo corretto i.e. compatibili con la definizione

```
data oggi;
```

```
data il_mio_compleanno;
```

```
void f() {  
    il_mio_compleanno.set(24,10,1979);  
    oggi.set(24,10,2001);  
    il_mio_compleanno.print();  
    oggi.next();  
}
```

Gestione di header file

- † un header file viene incluso nel punto in cui il compilatore incontra la direttiva include
- † come evitare inclusioni multiple (header incluso in header incluso in codice) ?

```
GenericHeader.h  
#ifndef FLAG_NAME  
#define FLAG_NAME  
... definizioni ...  
#endif // FLAG_NAME
```

Struct in Struct

† Se una struct contiene un sottoinsieme di dati con una struttura comune ...

```
struct Esterna {  
    int a;  
    struct Interna {  
        int b;  
    } dato;  
};
```

Sovrapposizione di nomi

† L'operatore di risoluzione dell'ambito può essere usato per indicare ambito *globale*

```
int f();  
struct S {  
    int f();  
};  
int S::f() {  
    ::f();  
};
```