Programmazione ad oggetti

Richiami sui tipi di dati

- Tipo = insieme di valori + insieme di operazioni
- Es. **`int`**:
  - **Valori**: …, -2, -1, 0, 1, 2, …
  - **Operazioni**: +,-,*,/, <, == …
  - Le operazioni si applicano a entità (oggetti) di tipo int
- Tipi definiti dall'utente
  - **Costruttori di tipo**: definiscono nuovi tipi composti a partire da altri tipi (array, struct, ...)
  - **Operazioni** predefinite anche per i costruttori di tipo (a.y, b[i], ecc.)
  - Strutture dati complesse possono essere costruite partendo dai tipi semplici tramite i costruttori di tipo

Vantaggi del concetto di tipo

- Classificare i dati:
  - Il programmatore non deve occuparsi di celle di memoria
    - Astrazione sulla struttura di rappresentazione
- Protezione da operazioni illegali (**type checking**)
  - Data d;
  - int g;
  - d = g; $\Rightarrow$ illegale! …forse non si voleva scrivere così!
  - Quando il linguaggio di programmazione è **strongly typed**, gli errori di tipo sono rilevati dal compilatore: non è necessario eseguire il programma per scoprirli!

Un (cattivo) esempio

- Per realizzare un'applicazione gestionale, **di una certa dimensione**, vari programmatori sono incaricati di realizzarne parti diverse
- Alberto è incaricato di programmare l'interfaccia utente e di definire il tipo Data per memorizzare le date
- Bruno utilizza il tipo Data per scrivere la parte di applicazione che gestisce paghe e contributi
  - Bruno si affida alla definizione del tipo Data di Alberto

Codice C di Alberto

```
typedef struct {
   int giorno;
   int mese;
   int anno;
} Data;

void stampaData(Data d) {
   printf("%d", giorno);
   if (d.mese == 1)
   printf("Gen");
   else if (d.mese == 2)
         printf("Feb");
   …
   printf("%d", anno);
}
```

Codice C di Bruno

```
Data d;
…
d.giorno=14;
d.mese=12;
d.anno=2000;
…
if (d.giorno == 27)
    pagaStipendi();
if (d.mese == 12)
    pagaTredicesime();
  …
```

La definizione di Data cambia!

- Dopo avere rilasciato una prima versione, Alberto modifica la rappresentazione per semplificare “stampaData”

```
typedef struct {
        int giorno;
        char mese[4];
        int anno;
} Data ;

void stampaData(Data d) {
        printf("%d", d.giorno);
        printf("%s", d.mese);
        printf("%d", d.anno);
}
```

Disastro!

- Il codice scritto da Bruno non funziona più!
- Occorre modificarlo:

```
Data d;
…
if (d.giorno == 27)
      pagaStipendi();
if (strcmp(d.mese,"Dic"))
      pagaTredicesime()
      …
```

Le modifiche sono costose

- Se la struttura dati è usata in molti punti del programma, una modifica piccola della struttura comporta tante piccole modifiche
- Fare tante (piccole) modifiche è:
  - Fonte di errori
  - Lungo e costoso
  - Difficile se la documentazione è cattiva
- La realizzazione interna (**implementazione**) delle strutture dati è una delle parti più soggette a cambiamenti
  - Per maggiore efficienza, semplificazione del codice...

Soluzione

- Il problema può essere risolto **disciplinando** (o impedendo) l'accesso alla realizzazione della struttura dati
  - Bruno non deve **mai** scrivere: d.mese == 12
  - …ma anche Alberto deve dare una mano!

Come usare una Data?

- Se si impedisce a Bruno l'accesso ai campi della struttura Data, come può Bruno utilizzare le date di Alberto?

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

- Il responsabile del tipo Data (Alberto) deve fornire delle operazioni sufficienti agli utilizzatori o “clienti” del tipo Data (Bruno)

Funzioni predefinite per Data

- L'accesso alla struttura dati viene fornito unicamente tramite operazioni predefinite
- In C:

```
void inizializzaData(Data *d, int g, int m, int a) {
        d->giorno = g;
        d->mese = m;
        d->anno = a;
}

int leggiGiorno(Data d){return(d.giorno);}
int leggiMese(Data d){return(d.mese);}
int leggiAnno(Data d){return(d.anno);}
```

Uso delle operazioni predefinite

- Il codice di Bruno deve essere scritto in modo da usare solo le operazioni predefinite:

```
Data d;
…

inizializzaData(&d, 14,12,2011);

if (leggiGiorno(d) == 27)
       pagaStipendi();
if (leggiMese(d) == 12)
       pagaTredicesime();
…
```

Modifica della struttura dati

- Se Alberto cambia l'implementazione di Data, deve modificare anche le operazioni predefinite
  - …lasciandone **immutato** il prototipo!

```
int leggiMese(Data d) {
        if (strcmp(d.mese,"Gen")) return(1);
        …
        else if (strcmp(d.mese,"Dic")) return(12);
}

void inizializzaData(Data *d, int g, int m, int a)) {
        d->giorno = g;
        if (m==1) d->mese = "Gen”;
        …
}
```

Il codice di Bruno non cambia!

Incapsulamento

- Abbiamo **incapsulato** la struttura dati!
- Ogni modifica alla struttura resta confinata alla struttura stessa
- Le modifiche sono
  - …più semplici
  - …più veloci
  - …meno costose

Tipo di dato astratto (ADT)

- Astrazione sui dati che non descrive i dati in base alla loro rappresentazione (**implementazione**), ma in base al loro comportamento atteso (**specifica**)
- Il comportamento dei dati è espresso in termini di un insieme di operazioni applicabili a quei dati (**interfaccia**)

- Le operazioni dell'interfaccia sono le sole che si possono utilizzare per creare, modificare e accedere agli oggetti
  - Applicazione del concetto di **information hiding**
  - L'implementazione della struttura dati non può essere utilizzata al di fuori della specifica

Riassumendo

- Definire un tipo di dato astratto:
  - Tipo: Data
  - Valori: date (giorno mese e anno)
  - Operazioni per una data D:
    - int mese( ): restituisce il mese corrispondente a D
    - ….

codice di Alberto versione 1

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

int mese(Data d){...}
```

codice di Alberto versione 2

```
struct Data {
    int giorno;
    char mese[4];
    int anno;
};

int mese(Data d){...}
```

codice di Bruno

```
Data d;
…
if (mese(d)==12)
    pagaTredicesime()
…
```

Progettare l'interfaccia

- Interfaccia di un ADT:
  - E’ una "promessa" ai clienti del tipo che le operazioni saranno sempre valide, anche in seguito a modifiche
  - Deve includere tutte le operazioni utili…

```
void giornoDopo(Data *d) {
        d->giorno++;
        if (d->giorno > 31) {  /* ma mesi di 30, 28, 29? */
                d->giorno = 1; d->mese++;
                if (d->mese > 12) {
                        d->mese = 1; d->anno++;
                }
        }
}
```

- Se la funzione non fosse definita da Alberto, Bruno avrebbe difficoltà a scriverla senza accedere ai campi di Data

Tipo di dato astratto (ADT)

- Incapsula tutte le operazioni necessarie a manipolarne i valori
  - Offre in maniera visibile:
    - Il **nome** del tipo
    - L'**interfaccia**: tutte e sole le operazioni per manipolare i dati del tipo e la loro specifica
  - Nasconde l'implementazione del tipo e delle operazioni necessarie
- Un cliente può creare **istanze** del tipo specificato e manipolarle tramite le operazioni definite

ADT in C?

- C non ha costrutti linguistici dedicati per ADT
- Ma con preprocessore, header file, puntatori, prototipi, magia nera, e una certa disciplina si può separare interfaccia e implementazione
- C inoltre manca di molte altre astrazioni utili per programmazione in-the-large, che si ritrovano invece nei **linguaggi orientati agli oggetti**

Linguaggi orientati agli oggetti

- Esiste un certo numero di linguaggi di programmazione ("orientati agli oggetti") che:
  - Obbligano ad incapsulare le strutture dati all'interno di opportune dichiarazioni (**classi**)
  - Consentono facilmente di impedire l'accesso all'implementazione
  - Rendono possibile creare istanze concrete (**oggetti**) da una astrazione per parametrizzazione
- **Costruttori** e **metodi** sono procedure che
  - …"appartengono" agli oggetti
  - …operano in maniera implicita sull'oggetto cui appartengono

Astrazione?

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise"

-E. W. Dijkstra