Programmazione procedurale e ad oggetti - 25/06/2018

Nome: .....
Cognome: .....
Matricola: .....

1. Illustrate le principali funzioni che il C offre per la gestione della memoria dinamica.

Sol. Si vedano (lucidi e libro di testo) le funzioni `malloc`, `calloc` e `free`.

2. A cosa serve la direttiva al preprocessore `#undef`? Mostrate un esempio d'uso significativo.

Sol. La direttiva viene utilizzata per eliminare eventuali definizioni di macro. L'uso tipico avviene per risolvere il conflitto su nomi tra macro definite dall'utente e macro presenti in file d'intestazione. Ad esempio, volendo definire una macro `DIMBUF`, se scrivessimo semplicemente `#define DIMBUF 100` e ci fosse in precedenza una definizione di macro con lo stesso nome, `#define DIMBUF 400` (ad es. presente in un include file) eviteremmo la segnalazione di ridefinizione scrivendo `#undef DIMBUF` subito prima della definizione della nostra macro. Immediatamente dopo la direttiva `#undef DIMBUF` cessano gli effetti di un'eventuale macro di nome `DIMBUF` (valida fino a quel punto), cosa che permetta la successiva definizione con successo.

3. Supponete di utilizzare la seguente struttura per implementare i nodi di alberi binari,

```
struct nodo{int c; struct nodo *sx; struct nodo *dx;}
```

Definite una procedura che riceve in ingresso un albero `p` e un intero `k`, e restituisce il numero di nodi dell'albero che si trovano sul livello `k` (il livello della radice è 0).

Sol. Una prima soluzione si ha usando una variabile statica per tener traccia del livello

```
int NnodiLiv(struct nodo* l,int k)
{static int level=0;int left=0;int right=0;
 if(k==level)return 1;
 if (l->sx!=NULL){level++;left=NnodiLiv(l->sx,k);}
 level--;
 if (l->dx!=NULL){level++;right=NnodiLiv(l->dx,k);level--;}
return left+right;}
```

Osservando che il livello da ricercare nei due sottoalberi è $k-1$ (assumendo come radice ciascun figlio), possiamo più semplicemente scrivere

```
int NnodiLiv(struct nodo* l,int k)
{int left=0;int right=0;
 if(k==level)return 1;
 if (l->sx!=NULL)left=NnodiLiv(l->sx,k-1);
 if (l->dx!=NULL)right=NnodiLiv(l->dx,k-1);
 return left+right;}
```

4. Illustrate i criteri utilizzati per individuare il metodo da eseguire quando un oggetto riceve un messaggio che richiede l'esecuzione di un metodo con selettore p. Distinguete i due casi, ereditarietà semplice ed ereditarietà multipla.

Sol. Un oggetto x che riceve un messaggio con selettore p ricerca innanzitutto nella propria classe d'appartenenza X se esiste un metodo con tale nome. Se così è allora è tale metodo ad essere eseguito, altrimenti distinguiamo i due casi:

Ereditarietà semplice. L'albero di ereditarietà fornisce un ordine totale delle superclassi di X. Il metodo che viene eseguito è quello che si trova nella prima superclasse che contiene un metodo con nome p e che si trova sul percorso che parte da X e si dirige verso la radice.

Ereditarietà multipla. Il grafo di ereditarietà non definisce in generale un ordine totale delle superclassi di X. Possono sorgere ad esempio dei conflitti tra due superclassi Y_1 e Y_2 di X che non sono confrontabili (ovvero nel grafo d'ereditarietà non esiste un cammino che parte da Y_1 e arriva a Y_2, o viceversa) e che contengono un metodo di nome p. Non esiste una strategia generale in grado di risolvere automaticamente i conflitti. Nel caso del C++, i conflitti vengono segnalati in compilazione e risolti (manualmente) attraverso l'uso dell'operatore di scope resolution (`::`).

5. Dite quale codice occorre scrivere (in aggiunta alla classe `Persona` così definita)

```
class Persona{
    char* nome; char* cognome;
    public:
    Persona(char* n,char* c);
};
Persona studente=Persona("Paolo","Rossi");s
```

in modo che `std::cout<<studente` stampi

```
Nome: Paolo
Cognome: Rossi
```

Sol. Occorre effettuare l'overloading dell'operatore `<<` per la classe `Persona`, ovvero aggiungere tra i membri di `Persona`

```
friend ostream& operator<<(ostream& os, const Persona& c)
{os<<"Nome: "<< c.nome<<' \n'<< "Cognome: "<< c.cognome<< '\n';return os;};
```

Si noti come la funzione operatore sia definita *amica* della classe `Persona`, in quanto deve poter accedere alle componenti private degli oggetti di tale classe.

6. Date due classi `T1` e `T2`, supponete che sia stato fatto l'overloading dell'operatore `+` in modo che `a+b` sia di tipo `T1` se anche `a` e `b` lo sono. Illustrate le possibili soluzioni per far sì che l'espressione `a+c` sia riconosciuta come corretta quando `c` è un'istanza di `T2`.

Sol. Se la classe `T1` è classe base pubblica di `T2` non occorre fare nulla, in quanto il compilatore può convertire automaticamente un oggetto di tipo `T2` in un oggetto di tipo `T1` (vale il principio di sostituzione). Altrimenti, oltre all'ovvia soluzione di effettuare un ulteriore overloading dell'operatore `+` (per `T1` e `T2`), una soluzione è quella di aggiungere un costruttore per `T1` con un parametro di tipo `T2`. Sarà quindi il compilatore a chiamare automaticamente tale costruttore per effettuare la conversione di `c`. In alternativa, si può definire un operatore di conversione di tipo nella classe `T2`, ovvero una funzione membro `T2::operator T1(){...}` in grado di convertire un oggetto di tipo `T2` in un oggetto di tipo `T1`.