OLTRE LA PROGRAMMAZIONE STRUTTURATA

Non bastano i concetti di strutturazione tipo blocchi e/o funzioni, la programmazione sia ***"in-the-large"*** (in grande) che quella ***"in-the-small"*** (in piccolo) richiede nuovi concetti:

1) MODULARITÀ

===> **programmare per parti**

Nasce come metodologia di programmazione *"in the large"*: un team di programmatori può suddividere il problema in sottoproblemi e, quindi, ogni programmatore si può concentrare sulla soluzione di un sottoproblema ===> evoluzione della programmazione TOP-DOWN

2) ASTRAZIONI

2a) ASTRAZIONE DI DATO (o DATO ASTRATTO)

2b) TIPO DI DATO ASTRATTO

===> **programmare per astrazioni**

Nasce come metodologia di programmazione *"in the small"*: ogni programmatore può individuare le astrazioni che risolvono un sottoproblema e, quindi, il team può risolvere il problema combinando le astrazioni sviluppate ===> evoluzione della programmazione BOTTOM-UP

1) MODULARITÀ

La **PROGRAMMAZIONE MODULARE** consiste nella costruzione di programmi suddividendoli in **PARTI** dette ***MODULI***

Caratteristiche di un MODULO:

- parte **INDIPENDENTE** dal resto del programma
- sviluppabile **SEPARATAMENTE** (compilazione e testing separata)
- con relazioni di interazione con le altre parti ben definite (tramite la definizione di una **INTERFACCIA**)

Queste caratteristiche conferiscono alla programmazione MODULARE le proprietà di:

☺ **RIUSABILITÀ**

☺ **ESTENDIBILITÀ**

☺ **LEGGIBILITÀ**

La MODULARITÀ si ottiene (*generalmente*) introducendo in un *linguaggio di programmazione* dei **COSTRUTTI** che consentono la DECOMPOSIZIONE in MODULI e la specifica della loro interazione

Linguaggi con costrutti che consentono la modularità:
MODULA, MODULA2, MESA, ADA, Turbo Pascal, Assembler 8086/88 – Linguaggi a Oggetti

PROGETTO MODULARE

La **PROGRAMMAZIONE MODULARE** richiede:

1) la definizione di **MODULI**: generalmente, tramite un costrutto base del linguaggio usato

2) garanzia di **LOCALITÀ**: le informazioni di un modulo devono essere **PRIVATE**, a meno che il modulo non le dichiari esplicitamente pubbliche e quindi globali

3) ogni MODULO deve definire in modo preciso la sua **INTERFACCIA**, che consente il collegamento con altri moduli

Per ottenere un ***buon programma MODULARE*** devono essere soddisfatte le seguenti **PROPRIETÀ:**

a) Un MODULO deve SCAMBIARE il **minor numero** possibile **di informazioni** con altri moduli, per garantire la **RIUSABILITÀ** e la **ESTENDIBILITÀ**

b) Ogni MODULO deve interfacciarsi con il **minor numero** possibile di altri moduli, per garantire la **RIUSABILITÀ** e la **ESTENDIBILITÀ**

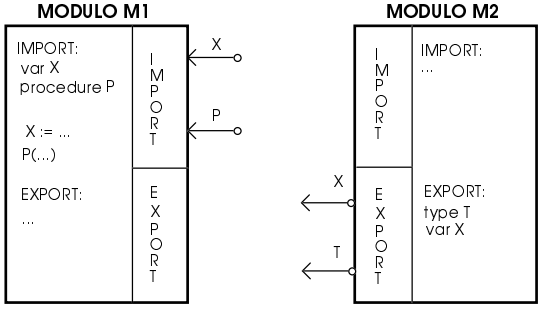
c) La **interazione** fra due MODULI deve essere **esplicita ed evidente**, per garantire la **LEGGIBILITÀ**

CONCETTO DI MODULO

Un **MODULO** può definire al suo interno diverse informazioni:
tipi, costanti, variabili, procedure e funzioni

La proprietà di **LOCALITÀ** garantisce che:
tutto ciò che è definito all'interno di un MODULO è visibile (cioè usabile) ***SOLO*** da quel modulo, a meno che non sia specificatamente reso visibile anche all'esterno e quindi ad altri moduli

Infatti, la proprietà di una precisa **INTERFACCIA** di un MODULO stabilisce che:
SOLO ciò che è **ESPLICITAMENTE esportato** è reso visibile all'esterno del MODULO
e dualmente
SOLO ciò che è **ESPLICITAMENTE importato** è reso visibile all'interno del MODULO
Quindi in una sola frase si può dire che un MODULO rappresenta un **AMBIENTE DI VISIBILITÀ**

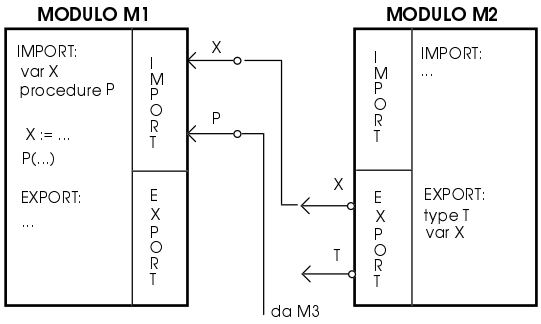


(**segue CONCETTO DI MODULO**)

Ogni **MODULO** viene definito separatamente e quindi può essere **COMPILATO separatamente** (o indipendentemente)

Le **correlazioni fra MODULI sono risolte staticamente** durante la fase di **collegamento** (linking dei moduli per produrre il programma eseguibile)

===> nel programma eseguibile non si distinguono più le parti separate



DIFFERENZA FRA MODULO E PROCEDURA:
Una procedura (o funzione), *in genere*, stabilisce per le informazioni in essa definite (ad esempio, variabili locali di una procedura Pascal) sia la **VISIBILITÀ** che il **TEMPO DI VITA**

Un MODULO, invece, ne stabilisce solo la VISIBILITÀ
e delega alla procedura il compito di stabilire il TEMPO DI VITA

2) ASTRAZIONI

2a)ASTRAZIONE DI DATO (o DATO ASTRATTO)

La **PROGRAMMAZIONE per ASTRAZIONI** consiste nella costruzione di programmi mettendo insieme delle **ASTRAZIONI DI DATO**

Una **ASTRAZIONE DI DATO** viene definita da
- ✯ un insieme di informazioni (cioè di dati)

unite a

- ✯ le operazioni (procedure e funzioni) che agiscono su di esse

Caratteristiche di una ASTRAZIONE DI DATO:
- **PROTEZIONE** delle informazioni: i dati non sono accessibili dall'esterno e quindi non sono manipolabili
- **INFORMATION HIDING**: la rappresentazione dei dati non è visibile e quindi abbiamo una completa indipendenza del resto del programma dalle scelte fatte

Queste caratteristiche conferiscono alla programmazione per ASTRAZIONI le proprietà di:

☺ **RIUSABILITÀ**

☺ **ESTENDIBILITÀ**

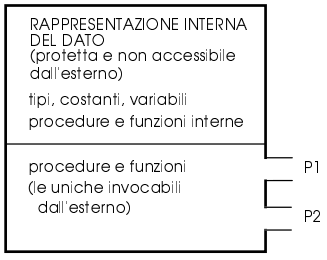
☺ **LEGGIBILITÀ**

La programmazione per ASTRAZIONI DI DATO si ottiene (*generalmente*) introducendo in un *linguaggio di programmazione* dei **COSTRUTTI** appositi
Linguaggi con costrutti di questo tipo: **CLU**

PROGETTO PER ASTRAZIONI

La **PROGRAMMAZIONE PER ASTRAZIONI** richiede:

1) la definizione di **ASTRAZIONI DI DATO**: generalmente, tramite un costrutto base del linguaggio usato
2) garanzia di **PROTEZIONE** e di **INFORMATION HIDING**: le informazioni di una ASTRAZIONE DI DATO devono essere **PRIVATE** e la loro **RAPPRESENTAZIONE INTERNA** deve essere **invisibile all'esterno**
3) ogni ASTRAZIONE DI DATO deve definire le OPERAZIONI che sono le uniche invocabili dall'esterno e che ne rappresentano l'***INTERFACCIA***



La proprietà di INFORMATION HIDING assicura che modifiche nella RAPPRESENTAZIONE INTERNA del dato astratto **non influenzano** il resto del programma

a patto di non produrre modifiche sulla interfaccia delle operazioni

ESEMPIO DI ASTRAZIONE DI DATO: UNO STACK

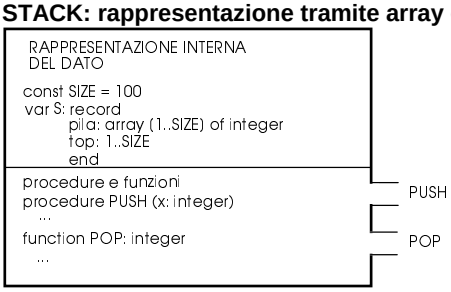
Uno **STACK** (o PILA) rappresenta un tipico esempio di dato astratto: viene definito come un contenitore di elementi che vengono inseriti ed estratti secondo una politica LIFO (last in - first out)

La **rappresentazione interna** dello STACK non è visibile a chi lo usa. Essa può essere realizzata in vari modi: tramite un array ed un indice che rappresenta la posizione dell'ultimo elemento inserito (il top dello stack); tramite una lista e un puntatore all'ultimo elemento inserito; e così via.

Le **operazioni** fornite all'esterno e che ne rappresentano l'interfaccia consentono di:
- inserire un elemento in cima allo STACK ===> **PUSH**
- estrarre un elemento dalla cima dello STACK ===> **POP**

Possono essere definite ulteriori operazioni, come ad esempio una operazione che consente di sapere se lo STACK è vuoto (cioè privo di elementi).

STACK: rappresentazione tramite array ed indice



2b) TIPO DI DATO ASTRATTO

La necessità che porta al passaggio dall'ASTRAZIONE DI DATO al **TIPO DI DATO ASTRATTO** è la stessa che in un linguaggio Pascal-like porta dalla specifica di un singolo dato (tramite la definizione di una variabile) alla specifica di un insieme di dati (tramite la definizione di un tipo)

Un linguaggio di programmazione che consente la definizione da parte del programmatore di tipi di dati astratti consente l'**ampliamento dell'insieme dei tipi** utilizzabili aventi le stesse caratteristiche dei tipi primitivi del linguaggio.

Per un **tipo primitivo** di un linguaggio non è infatti nota (in genere) la rappresentazione interna e la realizzazione delle sue operazioni.

Quindi, così come un linguaggio di programmazione che definisce, ad esempio, il ***tipo primitivo INTERO*** consente:
a) la istanziazione di variabili di tipo INTERO;
b) l'uso delle operazioni definite per il tipo INTERO (operazioni aritmetiche)

e garantisce che
c) su una istanza di tipo INTERO possano essere effettuate solo le operazioni consentite

un linguaggio di programmazione che fornisce al programmatore la possibilità di definire TIPI DI DATI ASTRATTI consente di ampliare l'insieme di base di tipi del linguaggio stesso
===> AUMENTO DELLA CAPACITÀ ESPRESSIVA

NOTA BENE: I costruttori di tipo (come ad esempio il costruttore ARRAY del Pascal) definisce sì un nuovo tipo, ma non allo stesso livello di un tipo primitivo poiché le operazioni consentite sono solo quelle definite dal linguaggio (come la selezione di un elemento dell'array)

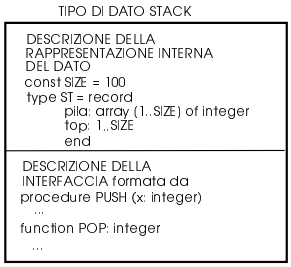
Un TIPO DI DATO ASTRATTO presenta tutte le caratteristiche di una ASTRAZIONE DI DATO ed in più, definendo un TIPO, aggiunge quella di essere **ISTANZIABILE**

Quindi, un **TIPO DI DATO ASTRATTO** è una **entità descrittiva** che specifica le caratteristiche (rappresentazione dei dati ed operazioni) di ogni istanza generabile da esso

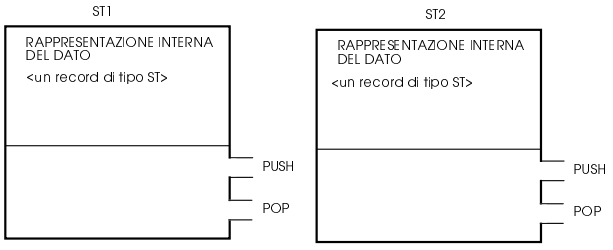
La programmazione per TIPI DI DATI ASTRATTI si ottiene (*generalmente*) introducendo in un *linguaggio di programmazione* dei **COSTRUTTI** appositi
Linguaggi con costrutti di questo tipo sono: **SIMULA67, CONCURRENT PASCAL, ALPHARD, ARGUS**

NOTA BENE: Le operazioni che un tipo di dato definisce sono riferite alle sue istanze

ESEMPIO DI TIPO DI DATO ASTRATTO: GLI STACK



Dal TIPO DI DATO ASTRATTO STACK si possono creare il numero di **istanze** necessarie, ad esempio **ST1** ed **ST2**

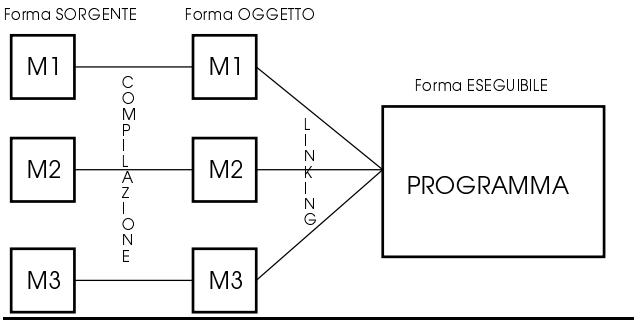


NOTA BENE: Ogni istanza del TIPO DI DATO STACK ha una sua rappresentazione interna separata

CONCETTI DI STRUTTURAZIONE APPLICATI AD UN PROGRAMMA C

1) MODULO ===> AMBIENTE DI VISIBILITÀ

Un modulo è un insieme di DATI (costanti, tipi e variabili) e CODICE (funzioni e procedure)
Un programma MODULARE è costituito da più moduli



Nel LINGUAGGIO C è possibile suddividere un programma su più MODULI

NOTA BENE: Non esiste però un COSTRUTTO, ma semplicemente

MODULO in C <===> file sorgente C
con la differenza che in un file C di default tutto è visibile
===> per rendere invisibili le entità, si deve usare la CLASSE DI MEMORIA static
Mentre per la fase di IMPORT/EXPORT, si deve usare la CLASSE DI MEMORIA extern

ESEMPIO DI MODULO C

```
/* FILE STACK.C ===> MODULO C */
#include <stdio.h>
#define SIZE 20

/* questo MODULO rappresenta un ambiente di visibilita' per la variabile S
(che quindi non e' visibile dall'esterno); per ottenere cio' in C si deve definire
esplicitamente S come STATIC  */

static int S [SIZE];

/* FASE DI ESPORTAZIONE la variabile TOP e le funzioni PUSH e POP
sono rese accessibili all'esterno: sono definite implicitamente (di default)
EXTERN*/

int TOP = 0;       /* inizializzazione */

void PUSH (int x)
{       S [TOP] = x;       TOP ++;    };
int POP ()
{       TOP --;       return  S [TOP];  };

/* FILE MAIN.C ===> MODULO che usa il MODULO STACK.C */
#include <stdio.h>
#define SIZE 20 /* E' necessario definirla anche in questo modulo:
                in C, non e' possibile esportare/importare le costanti (MACRO) */

/* FASE DI IMPORTAZIONE: per poter usare TOP e le funzioni PUSH e
POP, devono essere dichiarate esplicitamente EXTERN (secondo la
convenzione)*/
extern int TOP;
extern void PUSH (int x);
extern int POP (void);
void main (void   /* main */
{       if (TOP == SIZE) printf ("lo stack e' saturato");
        else  PUSH(10);
        if (TOP == SIZE) printf ("lo stack e' saturato");
        else  PUSH(100);
        if (TOP == 0) printf ("lo stack e' vuoto");
        else printf ("primo elemento estratto %d\n", POP());
        if (TOP == 0) printf ("lo stack e' vuoto");
        else printf ("secondo elemento estratto %d\n", POP());    };
```


2) ASTRAZIONE DI DATO

Una astrazione di dato è l'insieme di una struttura dati e delle uniche operazioni in grado di agire su di essa

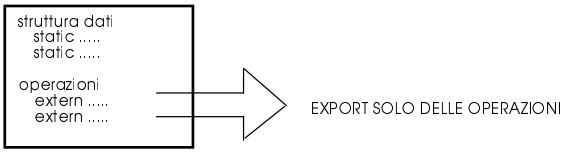
REALIZZAZIONE:

a) tramite un costrutto dedicato

IN C: NON ESISTE

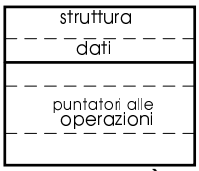
b) tramite uso "DISCIPLINATO" di un MODULO

IN C: un modulo/file sorgente realizza una astrazione di dato se e solo se la struttura dati è **STATIC** e un insieme di funzioni (le operazioni) sono **EXTERN**



c) tramite l'ampliamento del concetto di struttura dati

IN C: una STRUCT in cui un sottoinsieme dei campi è costituito da puntatori a funzioni (cioè referenziano le operazioni)



===> NON C'È GARANZIA DI PROTEZIONE

ESEMPIO DI ASTRAZIONE DI DATO IN C: UNO STACK REALIZZATO CON UN MODULO C

```
/* FILE STACK.C ===> MODULO CHE IMPLEMENTA L'ASTRAZIONE DI
DATO STACK */
#include <stdio.h>
#define SIZE 20

static struct { int TOP;  int S [SIZE]; } stato;
/* RAPPRESENTAZIONE INTERNA: array + indice */
/* la classe di memoria STATIC garantisce che la rappresentazione interna
e' PROTETTA e quindi non accessibile dall'esterno */

/* OPERAZIONI ESPORTATE: le uniche in grado di agire sullo STACK */
/* Non essendo specificata nessuna classe di memoria, sono di default
definite come funzioni EXTERN e quindi visibili ed invocabili dall'esterno */

void INIT ()
{       /* La funzione INIT serve per inizializzare correttamente lo STACK */
        stato.TOP = 0;  };

void PUSH (int x)
{       if (stato.TOP == SIZE) printf ("lo stack e' saturato");
/* il controllo sullo STACK pieno DEVE essere effettuato all'interno della
funzione PUSH non essendo visibile all'esterno la rappresentazione interna */
        else  {       int top;
                      top = stato.TOP;
                      stato.S [top] = x;
                      stato.TOP ++;  };
};

int POP ()
{       if (stato.TOP == 0) { printf ("lo stack e' vuoto"); return -100000; }
/* il controllo sullo STACK vuoto DEVE essere effettuato all'interno della
funzione POP non essendo visibile all'esterno la rappresentazione interna */
        else  {       int x, top;
                      stato.TOP --;
                      top = stato.TOP;
                      x = stato.S [top];
                      return x; };
};
```



```
/* FILE MAIN.C ===> MODULO che usa l'astrazione di dato STACK */
#include <stdio.h>

/* OPERAZIONI IMPORTATE: per poter usare lo STACK */
/* Viene specificata in queste dichiarazioni/prototipi come classe di memoria
quella EXTERN (secondo la convenzione) */

extern void INIT (void);
extern void PUSH (int x);
extern int POP (void);

void main ()
{ /* main */

/* prima di poter usare lo STACK bisogna operare l'inizializzazione */
INIT();

/* uso dello STACK */

PUSH(10);
PUSH(100);

printf ("primo elemento estratto %d\n", POP());
printf ("secondo elemento estratto %d\n", POP());
};
```

ESEMPIO DI ASTRAZIONE DI DATO IN C: UNO STACK REALIZZATO CON UNA STRUTTURA

```
#include <stdio.h>
#define SIZE 20

struct  {     struct { int TOP;  int S [SIZE]; } stato;
              struct {     void (*push)(int);
                           int (*pop)(void);  } operaz;
      } STACK;
/* STRUTTURA CHE IMPLEMENTA L'ASTRAZIONE DI DATO STACK: oltre alla rappresentazione interna, la struttura mantiene anche i puntatori alle OPERAZIONI invocabili ===> MANCA PROTEZIONE sulla rappresentazione interna*/

void INIT ();

void main ()       /* main */
{ INIT();     /* prima di poter usare lo STACK bisogna l'inizializzarlo */
/* uso dello STACK */
STACK.operaz.push(10);     STACK.operaz.push(100);
printf ("primo elemento estratto %d\n", STACK.operaz.pop());
printf ("secondo elemento estratto %d\n", STACK.operaz.pop());      };

static void stack_PUSH (int x)
{     if (STACK.stato.TOP == SIZE) printf ("lo stack e' saturato");
      else {      int top;
                  top = STACK.stato.TOP;              STACK.stato.S [top] = x;
                  STACK.stato.TOP ++;  };     };
static int stack_POP ()
{     if (STACK.stato.TOP == 0) { printf ("lo stack e' vuoto"); return -100000;
}
else  {     int x, top;
            STACK.stato.TOP --;          top = STACK.stato.TOP;
            x = STACK.stato.S [top];     return x; };  };

void INIT ()
/* Inizializzazione dell'astrazione di dato: devono essere inizializzati non solo il TOP, ma anche i puntatori a funzione */
{     STACK.stato.TOP = 0;
      STACK.operaz.push = &stack_PUSH;
      STACK.operaz.pop = &stack_POP;
/* oppure      STACK.operaz.push = stack_PUSH;
                STACK.operaz.pop = stack_POP;*/      };
```


3) TIPO DI DATO ASTRATTO

Un tipo di dato astratto consente di descrivere dati astratti dello stesso tipo (sue istanze)

REALIZZAZIONE:

a) tramite un costrutto dedicato
IN C: NON ESISTE

b) tramite uso "DISCIPLINATO" di un MODULO

IN C: un modulo/file sorgente realizza un tipo di dato astratto se dichiara il tipo della struttura dati e un insieme di funzioni (le operazioni) sono definite EXTERN
===> NON C'È GARANZIA DI PROTEZIONE

c) tramite l'ampliamento del concetto di TIPO di struttura dati

IN C: un tipo STRUCT in cui alcuni dei sottocampi sono puntatori a funzioni (cioè referenziano le operazioni) con uso di casting (ritipaggio) per realizzare **PROTEZIONE**
Necessità di una funzione di CREAZIONE (istanziazione)

d) tramite la definizione di un GESTORE

IN C: un modulo/file sorgente realizza un tipo di dato astratto se e solo se definisce un insieme (statico o dinamico) di strutture dati, un insieme di funzioni (le operazioni) EXTERN, fra cui anche una funzione di CREAZIONE (o istanziazione)
In questo caso, la funzione di creazione torna una chiave che **deve** essere utilizzata per identificare l'istanza

ESEMPIO DI TIPO DI DATO ASTRATTO IN C: STACK REALIZZATI CON UN MODULO C

```
/
* FILE STACK.C ===> MODULO CHE IMPLEMENTA IL TIPO DI DATO ASTRATTO STACK */
#include <stdio.h>
#define SIZE 20

typedef struct { int TOP;  int S [SIZE]; } STACK_TYPE;
/* definizione del TIPO che descrive la rappresentazione interna di una generica istanza di STACK ===> MANCA PROTEZIONE */

/* OPERAZIONI ESPORTATE: le uniche in grado di agire sulle istanze */
/* Diversamente da prima, adesso è necessario passare un parametro che identifica l'istanza su cui devono agire: ATTENZIONE IL PARAMETRO DEVE ESSERE PASSATO PER RIFERIMENTO */

void INIT (STACK_TYPE *sp)
{  sp->TOP = 0; };

void PUSH (STACK_TYPE *sp, int x)
{     if (sp->TOP == SIZE) printf ("lo stack e' saturato");
      else {      int top;
                  top = sp->TOP;
                  sp->S [top] = x;
                  sp->TOP ++;  };
};

int POP (STACK_TYPE *sp)
{     if (sp->TOP == 0) { printf ("lo stack e' vuoto"); return -1000000; }
else  {     int x, top;
            sp->TOP --;
            top = sp->TOP;
            x = sp->S [top];
            return x; };
};
```



```
/* FILE MAIN.C ===> MODULO che usa il tipo di dato astratto STACK */
#include <stdio.h>
#define SIZE 20

typedef struct { int TOP;  int S [SIZE]; } STACK_TYPE;
/* BISOGNA RIDEFINIRE IL TIPO (e la costante SIZE) PER POTER DEFINIRE VARIABILI E QUINDI ISTANZE DI STACK ===> MANCA PROTEZIONE */

/* OPERAZIONI IMPORTATE: per poter usare ISTANZE di STACK */
/* Viene specificata in queste dichiarazioni come classe di memoria
quella EXTERN secondo la convenzione */

extern void INIT (STACK_TYPE *);
extern void PUSH (STACK_TYPE *, int);
extern int POP (STACK_TYPE *);

void main ()
{ /* main */

STACK_TYPE s1, s2;
/* definizione di due variabili: CREAZIONE IMPLICITA DI DUE ISTANZE di STACK */

/* prima di poterle usare bisogna operare l'inizializzazione */
INIT(&s1);
INIT(&s2);

/* uso degli STACK */
PUSH(&s1, 10);  PUSH(&s1, 100);
PUSH(&s2, 30);  PUSH(&s2, 300);

printf ("primo elemento estratto da s1 %d\n", POP(&s1));
printf ("secondo elemento estratto da s1 %d\n", POP(&s1));

printf ("primo elemento estratto da s2 %d\n", POP(&s2));
printf ("secondo elemento estratto da s2 %d\n", POP(&s2));
};
```

ESEMPIO DI TIPO DI DATO ASTRATTO IN C: STACK REALIZZATO CON UN TIPO STRUTTURA

In questo caso è necessario usare PUNTATORI per poter ampliare/restringere la visibilità e quindi garantire **PROTEZIONE**

```
/* FILE STACK.C ===> modulo che definisce IL TIPO DI DATO ASTRATTO STACK */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20

typedef     struct {      void (* push)();
                          int (* pop)();
                                    } operaztype;
typedef operaztype * ptrinterface;
/* questo tipo serve esternamente per invocare le operazioni su una istanza di STACK */

typedef struct
{       operaztype operaz;
        struct { int TOP;  int S [SIZE]; } stato;
} STACKOBJ;
/* TIPO CHE DEFINISCE UNA STRUTTURA CHE IMPLEMENTA IL TIPO DI DATO ASTRATTO STACK: oltre alla rappresentazione interna, la struttura mantiene anche i puntatori alle OPERAZIONI invocabili ===> NOTA BENE l'ordine di definizione: prima i puntatori alle operazioni e poi la rappresentazione interna */

typedef STACKOBJ *ptr;
/* questo tipo serve internamente alle operazioni per ampliare la visibilita': tramite esso, risulta visibile anche la rappresentazione dei dati */
```



```
/* OPERAZIONI */

static void stack_PUSH (ptrinterface temp, int x)
{ ptr obj = (ptr) temp;
/* si amplia il tipo ptrinterface, tramite casting (ritipaggio), in modo da poter operare sull'istanza */

if (obj->stato.TOP == SIZE) printf ("lo stack e' saturato");
else  obj->stato.S [obj->stato.TOP ++] = x;
}

static int stack_POP (ptrinterface temp)
{ ptr obj = (ptr) temp;
/* si amplia il tipo ptrinterface, tramite casting (ritipaggio), in modo da poter operare sull'istanza */

if (obj->stato.TOP == 0) { printf ("lo stack e' vuoto"); return -100000; }
else  return obj->stato.S [-- obj->stato.TOP ];
};

/* In questo caso e' necessario definire una FUNZIONE DI CREAZIONE esplicita */
ptrinterface stackCREATE ()
/* La funzione di creazione (istanziazione) alloca spazio per l'intera struttura e la inizializza in modo corretto */
{ ptr obj = (ptr) malloc (sizeof (STACKOBJ));

obj->operaz.push = stack_PUSH;
obj->operaz.pop = stack_POP;
obj->stato.TOP = 0;
return (ptrinterface) obj;
/* viene ritornato un puntatore con visibilita' parziale */
};
```



```
/* FILE MAIN.C ===> MODULO che usa il tipo di dato astratto STACK */
#include <stdio.h>

typedef     struct {      void (* push)();
                          int (* pop)();
                                    } operaztype;
typedef operaztype * ptrinterface;
/* bisogna ridefinire il tipo per poter definire variabili e quindi istanze di STACK ===> questa volta pero' la PROTEZIONE viene GARANTITA */

/* BISOGNA IMPORTARE LA FUNZIONE DI CREAZIONE */
extern ptrinterface stackCREATE ();

main ()  { /* main */
ptrinterface OBJ1, OBJ2;
/* definizione di due variabili puntatori: serviranno per referenziare le due ISTANZE di STACK che si vogliono usare */

/* prima dell'uso delle istanze e' necessaria la loro creazione ed inizializzazione */
OBJ1 = stackCREATE();
OBJ2 = stackCREATE();

/* uso degli stack creati: sono visibili le sole operazioni e non la rappresentazione*/
OBJ1->push(OBJ1, 10);
OBJ1->push(OBJ1, 100);
OBJ2->push(OBJ2, 30);
OBJ2->push(OBJ2, 300);

printf ("primo elemento estratto da OBJ1 %d\n", OBJ1->pop(OBJ1));
printf ("secondo elemento estratto da OBJ1 %d\n", OBJ1->pop(OBJ1));

printf ("primo elemento estratto da OBJ2 %d\n", OBJ2->pop(OBJ2));
printf ("secondo elemento estratto da OBJ2 %d\n", OBJ2->pop(OBJ2));
};
```

CONSIDERAZIONE: Questa soluzione consentirebbe anche una fase di distruzione degli STACK realizzata tramite una funzione che utilizza la funzione primitiva free().

ESEMPIO DI TIPO DI DATO ASTRATTO IN C: STACK REALIZZATO CON UN GESTORE

```
/* FILE STACK.C ===> modulo che definisce IL TIPO DI DATO ASTRATTO STACK */
#include <stdio.h>
#define SIZE 20
#define DIM 10

static struct { int TOP;  int S [SIZE]; } SS[DIM];
/* ARRAY ===> questo GESTORE puo', al massimo, fornire 10 STACK */

void stack_push (int key, int x)
/* il parametro key serve per individuare quale STACK deve essere interessato all'operazione richiesta */
{       if (SS[key].TOP == SIZE) printf ("lo stack e' saturato");
        else  SS[key].S [SS[key].TOP ++] = x;
}

int stack_pop (int key)
/* il parametro key serve per individuare quale STACK deve essere interessato all'operazione richiesta */
{       if (SS[key].TOP == 0) { printf ("lo stack e' vuoto"); return -100000; }
        else  return SS[key].S [-- SS[key].TOP ];
};

static int KEY; /* valore iniziale 0 */
/* questa variabile serve per sapere se ci sono ancora STACK liberi */

int stackCREATE ()
{ int key;
/* questa funzione serve per acquisire uno STACK ed inizializzarlo */
if (KEY == DIM) { printf("Istanze di stack esaurite\n"); return -1; }
else {      key = KEY;
            SS[key].TOP = 0;
            KEY++;
            return key;
      }
};
```

```
/* FILE MAIN.C ===> MODULO che usa il tipo di dato astratto STACK */
#include <stdio.h>

/* BISOGNA IMPORTARE LA FUNZIONE DI CREAZIONE/ACQUISIZIONE oltre che le funzioni che agiscono sulle istanze */
extern int stackCREATE (void);
extern void stack_push (int, int x)
extern int stack_pop (int)

main ()
{ /* main */
int k1, k2;
/* definizione di due variabili intere: serviranno per mantenere le chiavi alle due ISTANZE di STACK che si vogliono usare */

/* prima dell'uso delle istanze e' necessaria la loro acquisizione ed inizializzazione */
k1 = stackCREATE();
k2 = stackCREATE();

/* uso degli stack creati: sono visibili le sole operazioni e non la rappresentazione*/

stack_push(k1, 10);
stack_push(k1, 100);
stack_push(k2, 30);
stack_push(k2, 300);

printf ("primo elemento estratto da k1 %d\n", stack_pop(k1));
printf ("secondo elemento estratto da k1 %d\n", stack_pop(k1));

printf ("primo elemento estratto da k2 %d\n", stack_pop(k2));
printf ("secondo elemento estratto da k2 %d\n", stack_pop(k2));
};
```

CONSIDERAZIONE: Questa soluzione si basa su uno schema di acquisizione di STACK: sarebbe possibile pensare anche ad una fase di RILASCIO?

ESEMPIO DI TIPO DI DATO ASTRATTO IN C: STACK REALIZZATO CON UN TIPO STRUTTURA (SECONDA VERSIONE)

In questo caso, si usa un puntatore ad una struttura (comune a tutte le istanze) che mantiene i puntatori alle operazioni ⇨ cfr. classe della programmazione ad oggetti

```
/* FILE STACK.C ===> modulo che definisce IL TIPO DI DATO ASTRATTO STACK */
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20

typedef  struct {  void (*push)();
                   int (*pop)();
                   }       operaztype;
typedef operaztype **ptrinterface;
/* questo tipo (puntatore di puntatore) serve esternamente per invocare le operazioni su una istanza di STACK */

typedef struct
{       operaztype *operaz;
        struct { int TOP;  int S [SIZE]; } stato;
} STACKOBJ;
/* TIPO CHE DEFINISCE UNA STRUTTURA CHE IMPLEMENTA IL TIPO DI DATO ASTRATTO STACK: oltre alla rappresentazione interna, la struttura mantiene anche il puntatore alla struttura che conterra' le OPERAZIONI invocabili ===> NOTA BENE l'ordine di definizione: prima il puntatore alla struttura con le operazioni e poi la rappresentazione interna */

typedef STACKOBJ *ptr;
/* questo tipo serve internamente alle operazioni per ampliare la visibilita': tramite esso, risulta visibile anche la rappresentazione dei dati */
```



```
/* OPERAZIONI */

static void stack_PUSH (ptrinterface temp, int x)
{ ptr obj = (ptr) temp;
/* si amplia il tipo ptrinterface, tramite casting (ritipaggio), in modo da poter operare sull'istanza */

if (obj->stato.TOP == SIZE) printf ("lo stack e' saturato");
else obj->stato.S [obj->stato.TOP ++] = x;
}

static int stack_POP (ptrinterface temp)
{ ptr obj = (ptr) temp;
/* si amplia il tipo ptrinterface, tramite casting (ritipaggio), in modo da poter operare sull'istanza */

if (obj->stato.TOP == 0) { printf ("lo stack e' vuoto"); return -100000; }
else return obj->stato.S [-- obj->stato.TOP ];
}

static operaztype *CLASSE = NULL;
/* NUOVA VARIABILE STATICA ===> rappresenta il puntatore alla struttura con le operazioni che sara' comune a tutte le istanze di stack */

/* In questo caso e' necessario definire una FUNZIONE DI CREAZIONE esplicita */
ptrinterface stackCREATE ()
/* La funzione di creazione (istanziazione) alloca spazio per l'intera struttura e la inizializza in modo corretto */
{ ptr obj = (ptr) malloc (sizeof (STACKOBJ));
if (CLASSE == NULL)
/* se stiamo creando la prima istanza, allora creiamo anche la struttura con le operazioni */
        {       CLASSE = (operaztype *) malloc(sizeof(operaztype));
                CLASSE->push = stack_PUSH;
                CLASSE->pop = stack_POP;
                }
/* in ogni modo, inizializziamo i campi della istanza */
obj->operaz = CLASSE;
obj->stato.TOP = 0;
return (ptrinterface) obj;
/* viene ritornato un puntatore con visibilita' parziale */
};
```

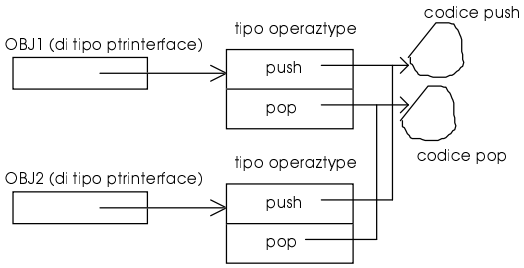


```
/* FILE MAIN.C ===> MODULO che usa il tipo di dato astratto STACK */
#include <stdio.h>

typedef  struct {  void (*push)();
                   int (*pop)();
                   }       operaztype;
typedef operaztype **ptrinterface;
/* bisogna ridefinire il tipo per poter definire variabili e quindi istanze di STACK ===> questa volta pero' la PROTEZIONE viene GARANTITA */

/* BISOGNA IMPORTARE LA FUNZIONE DI CREAZIONE */
extern ptrinterface stackCREATE ();


main ()  { /* main */
ptrinterface OBJ1, OBJ2;
/* definizione di due variabili: serviranno per referenziare le due ISTANZE di STACK che si vogliono usare */

/* prima dell'uso delle istanze e' necessaria la loro creazione ed inizializzazione */
OBJ1 = stackCREATE();           OBJ2 = stackCREATE();

/* uso degli stack creati: sono visibili le sole operazioni e non la rappresentazione*/
(*OBJ1)->push(OBJ1, 10);
(*OBJ1)->push(OBJ1, 100);
(*OBJ2)->push(OBJ2, 30);
(*OBJ2)->push(OBJ2, 300);

printf ("primo elemento estratto da OBJ1 %d\n", (*OBJ1)->pop(OBJ1));
printf ("secondo elemento da OBJ1 %d\n", (*OBJ1)->pop(OBJ1));

printf ("primo elemento estratto da OBJ2 %d\n", (*OBJ2)->pop(OBJ2));
printf ("secondo elemento da OBJ2 %d\n", (*OBJ2)->pop(OBJ2));
};
```

DIFFERENZE DELLE DUE SOLUZIONI CON TIPO STRUTTURA

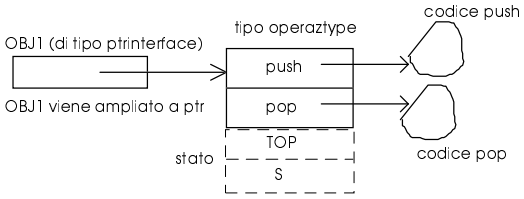
1) SOLUZIONE

L'utente usa il tipo ptrinterface



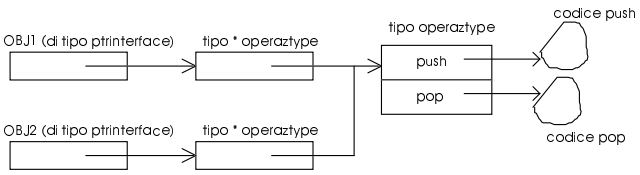
Ogni istanza ha il proprio puntatore alle funzioni proprie del tipo di dato astratto

All'interno delle funzioni collegate a push e pop viene ampliata la visibilità ⇨ da ptrinterface a ptr



2) SOLUZIONE

L'utente usa il tipo ptrinterface ma questa volta è



Ogni istanza ha un puntatore ad una struttura che contiene puntatori alle funzioni proprie del tipo di dato astratto

All'interno delle funzioni collegate a push e pop viene ampliata la visibilità ⇨ da ptrinterface a ptr