Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

In [ ]:

Il concetto di **object**

> **object**: region of data storage in the execution environment, the contents of which can represent

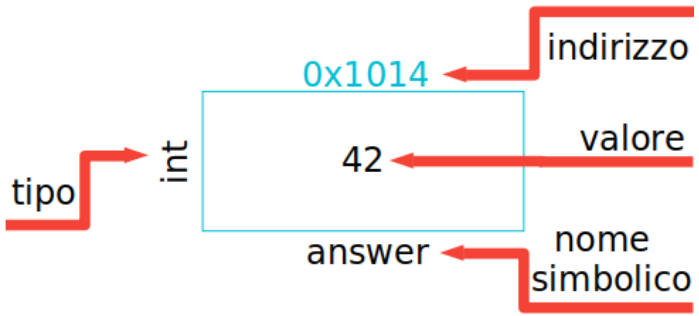
values

> (From: *C committee draft* www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)

NESSUNA relazione con la *Programmazione ad Oggetti* né con i file oggetto

Quindi in *C*, una regione di memoria contenente dati è chiamata genericamente **object** (variabili, array, etc.)

`int answer = 42;`



L'indirizzo corrisponde all'indirizzo di memoria del **primo** byte che compone l'object.

Dichiarazione Vs Definizione (object)

> **Dichiarazione**: dichiara il nome ed il tipo di un object (in sostanza dice che l'object esiste da qualche parte)

> **Definizione**: alloca lo spazio necessario per l'object (in sostanza viene materialmente creato in memoria)

In molti casi, le due fasi corrispondono. Esempio: `int a;` dichiara che esiste un object di nome `a` di tipo `int`, e contemporaneamente ne alloca lo spazio in memoria.

In generale, in uno stesso programma,un object può avere *più* dichiarazioni *ma una sola* definizione.

Esempio di un object *dichiarato* ma non *definito*:

`extern int g;`

`extern` *dichiara* l'esistenza dell'object `g`, *definito* (adesso o in futuro) da qualche altra parte. Il compilatore quindi, nel caso in cui sia richiesto l'utilizzo di `g` prima dell'effettiva *definizione*, non se ne preoccupa e continua il processo di compilazione (in quanto semplice traduzione).

Esempio:

In [11]:
```
#include <stdio.h>
int main(void)
{
    extern int first, last; /* use global vars */
    printf("i valori di first e last sono: %d %d\n", first, last);
    return 0;
}
/* global definition of first and last */
int first = 10, last = 20;
```

```
i valori di first e last sono: 10 20
```

dato che si richiede l'utilizzo di `first, last` *prima* della loro effettiva *definizione*, vanno almeno *dichiarate* in precedenza.

Inizializzazione Vs Assegnazione

> **Assegnazione**: dare valore ad un object in qualsiasi punto del programma

> **Inizializzazione**: dare valore ad un object *in fase di definizione*

```
{
    int a; // dichiarazione & definizione
    a = 3; // assegnazione
}
{
    int a = 3; // dichiarazione & definizione & inizializzazione
}
```

In C, alcune cose lecite in fase di inizializzazione **non lo sono** in fase di assegnazione.

Ad ogni modo, un object è detto **inizializzato** se gli è stato dato un valore dal programma, indipendentemente dal modo in cui ciò sia avvenuto.

Esempio:

```
int V[] = {1,2,3}; // si può fare
int Q[];
Q[] = {4,5,6}; // non si può fare
const int i = 5; // si può fare
const int i;
i = 5; // non si può fare
```

Layout della memoria di un programma C

Un programma C in esecuzione utilizza 4 regioni di memoria logicamente distinte:

1. regione dedicata a contenere il codice in esecuzione (**text/code area**)
2. regione dedicata a variabili `external`, `global` e `static`. Si divide a sua volta in:
   - **initialized data segment**: variabili `extern`, `global` e `static` i cui valori **sono esplicitamente inizializzati** in fase di dichiarazione
   - **uninitialized data segment** (*bss, block started by symbol*): variabili `extern`, `global` e `static` non inizializzate in fase di dichiarazione. Il kernel inizializza queste variabili a 0 (o `NULL` nel caso di puntatori) prima che il programma entri in esecuzione
3. l'**execution stack**, o *call stack*, o spesso chiamato comunemente *stack* (anche se fonte di ambiguità), contenete gli indirizzi di ritorno delle funzioni invocanti, argomenti e **variabili locali**. Il termine *stack* si riferisce alla *politica di accesso* (LIFO, Last In First Out).
4. l' **heap**, regione di spazio libero utilizzata per l'allocazione dinamica (NB: nessuna relazione con l'omonima struttura dati)

il comando linux `size` fornisce informazioni sull'occupazione di memoria di un programma:

`esempio.c`:

```
#include<stdio.h>

int main() {
    return 0;
}
```

`gcc esempio.c -o esempio`

`size esempio`

output:

```
   text    data     bss     dec     hex filename
   1418     544       8    1970     7b2   esempio
```

La dimensione massima dell' *execution stack* può essere recuperata con il comando `ulimit -s` (l'output è espresso in KB).

`ulimit -s`

output:

`8192`

In questo caso, lo spazio a disposizione nell' execution stack è 8192 KB.

Tipi di allocazione in C

Il C supporta due tipi di allocazione di memoria:

- *__static allocation__*: destinata alle variabili `global` o `static`. Per Ogni variabile viene definito un blocco di spazio di dimensione fissata. Lo spazio viene allocato* all'avvio del programma*.
- *__automatic allocation__*: destinata alle variabili `automatic`, come gli argomenti di funzione o le variabili locali. Tali variabili vengono allocate quando il programma entra materialmente nel blocco in cui tali variabili sono definite, e deallocate quando termina tale blocco.

Una terza tipologia di allocazione, la *__dinamic allocation__*, *non è supportata "in maniera diretta" dal C, ma è disponibile attraverso funzioni di libreria apposite. L'allocazione dinamica è necessaria quando non si conosce* a priori* quanta memoria è necessaria.

Strutture dati native del C

Una struttura dati è un insieme di dati di tipi base aggregati assieme secondo uno schema.

Il C mette a disposizione in maniera nativa le seguenti strutture dati:

- **array**:
  - monodimensionali
  - multidimensionali
- **struct**:
  - semplici
  - ricorsive