Generic 2nd Best Attività Didattica LPS

Samuel ha un incarico da Software Developer presso l'azienda ACMEware. Il Software Engineer che coordina il progetto su cui sta lavorando, vorrebbe fargli utilizzare una procedura scritta in Java che, dato un array **A** di almeno 2 elementi, restituisce il *second best* dell'array, ovvero il secondo, in ordine decrescente, tra i valori contenuti nell'array. Più precisamente, in termini più formali, la procedura restituisce un valore **S** (del tipo degli elementi dell'array), tale che valgono le seguenti 3 proprietà:
1)l'array contiene un elemento **A[x]** tale che **A[x] = S**
2)l'array contiene esattamente un elemento **A[y]** tale che **A[y] > S**
3)per ogni elemento **A[z]** dell'array diverso da **A[y]**, si ha **A[z] <= S**

Il Software Engineer, in base alla sua interpretazione dei principi dell'*information hiding* e del riuso del software, fornisce a Samuel tutte le informazioni necessarie a utilizzare tale procedura, ma non il codice sorgente.
Poiché la procedura deve essere impiegata su array di lunghezze nell'ordine di 10^8, Samuel preferirebbe riscriverla in C, in modo da ottenere prestazioni decisamente migliori. Il Software Engineer è contrario, in quanto la procedura Java, grazie a caratteristiche relativamente avanzate del linguaggio, quali i tipi generici e le interfacce, è in grado di operare su array che hanno elementi di svariati tipi, tra cui `Double`, `Integer`, `String`, nonché molti tipi oggetto usati nel progetto.

Aiutate Samuel a dimostrare al suo Software Engineer che, con un po' di maestria, si può realizzare una versione C che ha le stesse caratteristiche (ed è naturalmente più veloce)!
E per far trionfare Samuel e porlo al riparo da ogni ulteriore interferenza dal Software Engineer, esagerate scrivendo persino le versioni Assembly MC68000 e MIPS32!

Procedete come segue:

Fase 1: realizzazione della versione C

Fase 1.1: size_t
C Standard definisce un tipo che nome size_t. Studiate le caratteristiche di tale tipo (per es. nella sezione 7.6 di [Ki] o nella sezione 7.17 di C Standard). Poi, per ciascuna delle seguenti affermazioni, dite se è vera o se è falsa

1. Il tipo `size_t` è un nuovo tipo introdotto in C99, diverso da ogni altro tipo
2. Il tipo `size_t` può essere un tipo intero oppure un tipo floating point
3. Può esistere un'implementazione di C Standard in cui `size_t` è un alias per il tipo `int`
4. Non è possibile che ci sia un'implementazione di C Standard in cui `size_t` è un alias per il tipo `signed long`
5. Il risultato di `sizeof( int )` può essere 4 oppure 8
6. Può esistere un'implementazione di C Standard in cui `size_t` è un alias per il tipo `void *`
7. È possibile che ci sia un'implementazione di C Standard in cui `size_t` è un alias per il tipo `unsigned short` e un'altra in cui `size_t` è un alias per il tipo `unsigned long long`

8. Il tipo `size_t` non può essere un alias per `signed int`, ma può essere un alias per `int`
9. In almeno un'implementazione di C Standard, `size_t` è un alias per il tipo `unsigned *`
10. Non è possibile che ci sia un'implementazione di C Standard in cui `size_t` è un alias per il tipo `signed long` e un'altra in cui `size_t` è un alias per il tipo `unsigned long`

Fase 1.2: stringhe in C

Fase 1.2.1

Poiché la procedura deve poter operare anche su array i cui elementi sono stringhe, è necessario rappresentare in C, in modo efficiente, tali array. In C (diversamente che in Java) non esiste un tipo di dato dedicato alle stringhe. Le stringhe vengono convenzionalmente rappresentate come array di caratteri, nei quali un carattere di valore 0 indica il termine di una stringa. Studiate i dettagli di tale modo di gestire le stringhe nelle sezioni 13.1 e 13.2 di [Ki]. Scrivete un piccolo programma dimostrativo in C, che:

- definisce (come variabile esterna) un array di caratteri **S1** inizializzato mediante uno string literal
- contiene una funzione `invstr`, che ha due parametri di tipo array di carattere e che scrive nel secondo parametro la stringa che si ottiene leggendo all'inverso (cioè dall'ultimo carattere verso il primo) la stringa passata nel primo parametro; `invstr` assume che il secondo parametro sia un array di lunghezza sufficiente a contenere il risultato, il cui contenuto (quando inizia l'esecuzione di `invstr`) è però indefinito (cioè non si deve assumere che l'array abbia dei particolari valori).
- nella funzione `main` chiama la funzione `invstr` passando come primo parametro **S1** e come secondo parametro un array allocato in modo opportuno

Fase 1.2.2

Avrete bisogno di definire un array di stringhe. Il modo più efficiente di farlo, è definire un array i cui elementi sono puntatori a stringhe (cioè un array i cui elementi sono puntatori ad array di caratteri). Studiate i dettagli di tale tecnica nella sezione 13.7 di [Ki]. Poi scrivete un piccolo programma dimostrativo in cui viene definito un array 7 di puntatori a stringhe. L'array deve essere inizializzato, ponendo un valore in ciascuno dei suoi elementi, rispettando le seguenti condizioni:

1. almeno uno dei puntatori deve essere il null pointer
2. almeno uno dei puntatori deve puntare uno string literal (si veda la sezione 13.1 di [Ki] per la definizione di string literal)
3. almeno uno dei puntatori deve puntare una variabile di tipo array di caratteri, e tale variabile deve essere inizializzata mediante uno string literal
4. almeno uno dei puntatori deve puntare una variabile di tipo array di caratteri, e tale variabile deve essere inizializzata non mediante uno string literal ma mediante un inizializzatore formato da una lista di caratteri racchiusi tra parentesi graffe
5. almeno uno dei puntatori deve puntare un array di caratteri che contiene la stringa vuota

Il programma deve usare una variabile di tipo puntatore per scorrere l'array, e per ogni elemento dell'array diverso dal null pointer, deve stampare la stringa da esso puntata

Fase 1.2.3
Avrete anche bisogno di confrontare due stringhe per stabilire (in base ad un qualche criterio) se esse sono uguali o quale delle due è minore dell'altra. C Standard Library definisce la funzione `strcmp`, che ha esattamente lo scopo di confrontare due stringhe in base ad un criterio di ordinamento detto *lessicografico*. Studiate il funzionamento di `strcmp` nella sezione 13.5 di [Ki] e scrivete un piccolo programma dimostrativo dell'uso di `strcmp`, in cui vengono fatti dei confronti tra coppie di stringhe. Si osservi che l'ordinamento lessicografico è quello comunemente usato nei dizionari, e che date due stringhe differenti, viene considerata minore la stringa che precede l'altra (ovvero "ABCD" viene considerata minore di "XY").

Fase 1.3: l'idea di Samuel
L'algoritmo che, dato un array, individua il second best, è, in linea di principio, indipendente dal tipo degli elementi dell'array. L'unica cosa necessaria, è che dati due elementi dell'array, sia possibile effettuare un confronto per stabilire se essi sono uguali oppure quale dei due è minore dell'altro.
Ovviamente il modo di fare il confronto dipende da qual'è il tipo degli elementi confrontati, ma questa è l'unica parte dell'algoritmo che varia in base al tipo degli elementi.
Come implementare un algoritmo del genere in C? Samuel ci ha pensato bene, e ha un'idea, che vi espone pregandovi di aiutarlo. Poiché l'unica parte dell'algoritmo che dipende dal tipo degli elementi è quella in cui si fa il confronto, questa parte va implementata separatamente dal resto. Ovvero, Samuel ha pensato di scrivere un'unica funzione C che implementa la parte di algoritmo indipendente dal tipo degli elementi, che si interfaccia con il codice che effettua i confronti, di cui ovviamente esisteranno tante versioni, una per ciascuno dei tipi degli elementi considerati. A tale scopo è necessario definire un'interfaccia uniforme tra la parte di codice indipendente dai tipi degli elementi e quella che fa i confronti, in modo che la parte indipendente possa attivare in uno stesso modo ciascuna delle versioni del codice che effettua i confronti. Samuel ha capito che il modo migliore di fare ciò in C è usare i puntatori a funzione. La parte di codice indipendente dai tipi va implementata in una funzione (chiamata `alg_2nd_best`) che tra i suoi parametri ha un puntatore (di nome `comparator`) ad un'altra funzione. Attraverso `comparator`, la funzione `alg_2nd_best` chiama una funzione che effettua il confronto tra due elementi dell'array: in questo modo `alg_2nd_best` può conoscere l'esito dei confronti ignorando del tutto quali sono i tipi degli elementi e qual'è la specifica funzione di confronto utilizzata.

Fase 1.3.1
Per poter realizzare l'idea di Samuel, è necessario che l'interfaccia tra e le funzioni di confronto sia uniforme, ovvero è necessario che tutte le funzioni di confronto siano compatibili con il tipo del puntatore a funzione `comparator`: devono quindi avere la stessa quantità di parametri, gli stessi tipi per i parametri e lo stesso tipo per il risultato.
Occupatevi per prima cosa del risultato (che è la parte più semplice). Il risultato restituito dalla chiamata, è il risultato del confronto tra 2 valori. I meccanismi di confronto messi direttamente a disposizione dal C, purtroppo, non sono sempre uniformi. Per valori di tipo numerico (ad esempio di tipo `int` oppure due valori di tipo `double`) possiamo usare gli operatori relazionali (si veda cap. 5 di [Ki]) ma tali

operatori non possono essere usati sulle stringhe, per le quali, come visto nella fase 1.2.3, si usa `strcmp`. Per tipi di dato definiti dal programmatore mediante la definizione di tipi struttura, è necessario che sia il programmatore stesso a definire delle funzioni che fanno confronti.
Prendendo spunto da `strcmp`, si può pensare di scrivere, per ogni tipo T usato nel progetto, una funzione che prende come parametri 2 valori di tipo T e che restituisce un risultato **r** tale che:

- **r** = 0 se i due valori sono uguali
- **r** > 0 se il primo valore è maggiore del secondo
- **r** < 0 se il primo valore è minore del secondo

In preparazione del lavoro da svolgere nelle prossime fasi, scrivete una funzione `intcmp`, che confronta due valori di tipo `int` e restituisce un risultato come appena descritto.

Fase 1.3.2
Tutte le funzioni che confrontano valori devono avere non solo lo stesso tipo di risultato, ma anche stessa quantità e stessi tipi di parametri. Naturalmente, i parametri devono essere 2. Ma come possono avere gli stessi tipi per i parametri, quando ciascuna di esse deve confrontare valori di tipo diverso? A prima vista, ogni funzione dovrebbe avere tipi diversi: ad esempio `strcmp` ha parametri di tipo array di carattere mentre `intcmp` ha due interi. La soluzione è definire i parametri con un tipo "generico", e poi convertire i parametri dal tipo generico al tipo corretto all'interno di ciascuna funzione. Il C purtroppo non ha tipi generici (diversamente da Java o da altri linguaggi più moderni) ma può ottenere gli stessi effetti grazie alla sua capacità di gestire direttamente indirizzi di memoria mediante i cosiddetti puntatori a `void`. Approfondite il funzionamento dei puntatori a `void` nella sezione 13.5 di [Ki] e usateli per scrivere due funzioni `int_comparator` e `str_comparator`, entrambe compatibili con il puntatore a funzione `comparator`:

- `int_comparator` confronta due interi
- `str_comparator` confronta due stringhe, richiamando `strcmp`

Fase 1.3.3
A questo punto non resta che scrivere la funzione C `alg_2nd_best` , che implementa la ricerca del *second best* di un array. Samuel vorrebbe spiegarvi l'algoritmo, ma in questo momento è in riunione e può parlare poco al telefono. Vi da solo un indizio: utilizzate una ricerca lineare, come per trovare il massimo, con la differenza che durante la ricerca oltre al massimo tenete traccia mediante un'altra variabile del secondo miglior valore trovato. Comunque Samuel è sicuro che l'algoritmo saprete svilupparlo da soli facilmente. Invece dovrete fare attenzione alla gestione dell'array. Infatti, la funzione deve avere come parametro un array, i cui elementi possono avere un qualunque tipo. Usate di nuovo un puntatore a `void` come "tipo generico". Quindi, per passare l'array alla funzione, dovete definire un parametro `p_ar` di tipo puntatore a void. Il parametro `p_ar` punta al primo elemento dell'array. Per puntare ai successivi elementi, dovrete usare l'aritmetica dei puntatori. Ma qui fate molta attenzione, perché i puntatori devono essere usati con più attenzione rispetto ai puntatori ad altri tipi. Come sapete, dato un puntatore `p` al primo elemento di un array che ha tipo diverso da `void`, l'espressione `p+i` punta l'`i`-esimo elemento

dell'array. Ma poiché un puntatore a `void` contiene un indirizzo di memoria, in questo caso l'espressione `p_ar+i` punta semplicemente al byte che ha indirizzo pari a quello del primo elemento aumentato di `i`. Quindi per puntare l'elemento di indice `i` di `p_ar`, non va bene l'espressione `p_ar+i`, ma dovete tenere conto del fatto che ogni elemento di `p_ar` può occupare una quantità di byte maggiore di 1. Come fare? Samuel ha solo il tempo per darvi un altro indizio: "fate come in assembly". Poi riattacca.
Dopo qualche minuto, però, vi manda un messaggio:
"per riassumere, `alg_2nd_best` deve avere i seguenti parametri:
1. punt. all'array (tipo void *)
2. lunghezza array (usate tipo size_t)
3. dimensione in byte di un elemento dell'array (tipo size_t)
4. punt. `comparator`"

Fase 1.3.4
Ce l'avete fatta, avete scritto `alg_2nd_best` . Ma sarà corretta? Dovete testarla, non vorrete far fare brutta figura a Samuel! Scrivete una funzione `main` che faccia:
1)Test della funzione, per trovare il *second best* di un array di `int`
2)Test della funzione, per trovare il *second best* di un array di stringhe
3)Test su un tipo struttura

- Definite un tipo di struttura **S**, che contiene un campo **m1** di tipo `long` e un campo **m2** di tipo `char`, e un array di elementi di tipo **S**
- Definite una funzione `stS_comparator` che faccia un confronto tra due valori **s1** ed **s2** di tipo **S**, in base al seguente criterio:
- **s1** ed **s2** sono uguali se e solo se entrambi i rispettivi campi sono uguali
- **s1** è maggiore di **s2** se e solo se vale una delle due seguenti:
- il campo **m2** di **s1** è minore del campo **m2** di **s2**
- il campo **m2** di **s1** è uguale al campo **m2** di **s2** e il campo **m1** di **s1** è maggiore del campo **m1** di **s2**
- **s1** è minore di **s2** se e solo se **s2** è maggiore di **s1**
- Usate `alg_2nd_best` per trovare il *second best* dell' array di elementi di tipo **S** definito in precedenza

Fase 2: realizzazione delle versioni Assembly
Nelle versioni Assembly, i tipi C devono essere implementati come segue.
I tipi puntatore devono essere tutti implementati:
a. con il formato `word` nella versione MIPS32
b. con il formato `long` nella versione MC68000
Il tipo `int` deve essere implementato:
a. con il formato `word` nella versione MIPS32
b. con il formato `word` nella versione MC68000
Il tipo `char` deve essere implementato:
a. con il formato `byte` nella versione MIPS32
b. con il formato `byte` nella versione MC68000
Il tipo `long` deve essere implementato:
a. con il formato `word` nella versione MIPS32
b. con il formato `long` nella versione MC68000

Fase 2.1
Si realizzino una versione MIPS32 e una versione MC68000 della funzione `int_comparator` (si veda Fase 1.3.2). In entrambe le versioni, i parametri e il risultato devono essere gli stessi usati per la versione C e devono essere passati mediante registri.

Fase 2.2
Si realizzino una versione MIPS32 e una versione MC68000 della funzione `alg_2nd_best` (si veda Fase 1.3.3). In entrambe le versioni, i parametri e il risultato devono essere gli stessi usati per la versione C e devono essere passati mediante stack.

Fase 2.3
Si scrivano una versione MIPS32 e una versione MC68000 della funzione `main` (si veda Fase 1.3.4), che possono essere lanciate per verificare la correttezza delle versioni Assembly di `alg_2nd_best`, nel trovare il *second best* di un array di `int` allocato staticamente.

Fase 2.4: per essere ancora più ambiziosi
Si implementi, sia in MIPS32 che in MC68000 il tipo di struttura **S**. Ciascuno dei campi deve essere implementato con il formato stabilito in Fase 2, ma, poiché si dovrà poi definire un array di oggetti di tipo struttura **S** allocati consecutivamente, è necessario che le implementazioni del tipo **S** abbiano dei byte aggiuntivi (inutilizzati), detti *byte di pad*, per garantire il corretto allineamento. La quantità di byte di pad da aggiungere dipende dai vincoli di allineamento di ciascuna architettura, descritti nella presentazione *Organizzazione dei Dati in Memoria 1*.
Si scrivano una versione MIPS32 e una versione MC68000 della funzione `stS_comparator` (si veda Fase 1.3.2). In entrambe le versioni, i parametri e il risultato devono essere gli stessi usati per la versione C e devono essere passati mediante registri. Poi si modifichino le routine principali per verificare la correttezza di `alg_2nd_best`, nel trovare il *second best* di un array di strutture **S** allocato staticamente.

Fase 2.4: per stupire anche Samuel
Si implementi, sia in MIPS32 che in MC68000 un array di stringhe, in modo analogo a quanto fatto in C nella Fase 1.3.4 (quindi in realtà si deve implementare un array di puntatori a stringhe), allocato staticamente (sia l'array di puntatori sia le singole stringhe devono essere allocate staticamente).
Si scrivano una versione MIPS32 e una versione MC68000 della funzione `str_comparator` (si veda Fase 1.3.2). In entrambe le versioni, i parametri e il risultato devono essere gli stessi usati per la versione C e devono essere passati mediante registri. Poi si modifichino le routine principali per verificare la correttezza di `alg_2nd_best`, nel trovare il *second best* dell'array di stringhe.

That's All Folks!