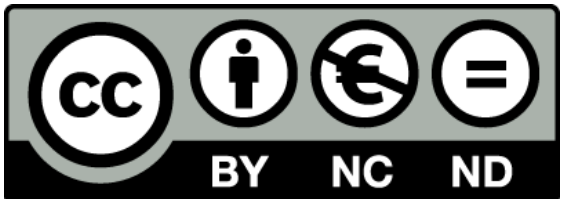
Laboratorio di Programmazione di Sistema

C Standard

Luca Forlizzi, Ph.D.

Versione 20.2



Motivazioni

- Dennis Ritchie non creò il linguaggio C partendo con un progetto ben definito
- L'evoluzione del linguaggio procedette di pari passo con la creazione di traduttori e di run-time support che consentissero di eseguire i costrutti del linguaggio
- Di conseguenza, le descrizioni delle prime versioni del C erano costituite dai manuali dei rispettivi traduttori
- In questa situazione, come conseguenza del crescente successo del C, vi era il rischio che nascessero numerosi dialetti, incompatibili tra loro

Motivazioni

- Ad arginare la proliferazione dei dialetti intervenne la prima architettura del linguaggio C, ovvero la prima edizione del testo *The C Programming Language*, pubblicato nel 1978 da Brian Kernighan e Dennis Ritchie; ci si riferisce comunemente al testo con l'abbreviazione K&R o, per sottolineare che ci si riferisce alla prima edizione del testo, K&R1
- Successivamente, il compito di definire nuove architetture del linguaggio C è stato assunto da organismi internazionali che si occupano di standardizzazione, in particolare dall'ISO/IEC
- L'esistenza di architetture di riferimento è un importante stimolo a fare in modo che i traduttori C siano implementazioni di tali architetture

C Standard

- Chiamiamo *C Standard* una famiglia di architetture, definite, oltre che in K&R, in 4 edizioni e un emendamento del documento ISO/IEC 9899

Sigla	Nome	Innovazioni principali
K&R1	C Tradizionale	• Libreria standard I/O • tipi `long int` e `unsigned int`
C89 o C90	ISO/IEC 9899:1990	• dichiarazioni e definizioni di funzioni in forma di prototipo • funzioni `void` • i tipi struttura e unione possono essere usati negli assegnamenti e come parametri e risultati di funzione
C95	ISO/IEC 9899/AMD1:1995	• supporto per set di caratteri estesi
C99	ISO/IEC 9899:1999	• tipi `long long` e `_Bool` • puntatori `restrict` • *variable-length* array • compound literals • istruzioni possono seguire dichiarazioni in un blocco • funzioni `inline`
C11	ISO/IEC 9899:2011	• supporto per esecuzione di thread multipli • espressioni con tipo generico • gestione allineamento oggetti • strutture e unioni anonime
C17 o C18	ISO/IEC 9899:2018	• nessuna

C Standard

- Ciascuno di tali documenti, tranne K&R1, definisce 2 versioni dell'architettura
  - versione *freestanding* che descrive il solo linguaggio C
  - versione *hosted* che comprende la corrispondente versione freestanding e aggiunge la descrizione della C Standard Library
- In totale C Standard comprende 11 versioni dell'architettura (ma le versioni di C11 e C18 sono quasi identiche)
- La backward-compatibility è elevata, ma non totale
- In LPS utilizzeremo prevalentemente C99, con alcuni riferimenti ad altre versioni

Scopi e principi dello Standard

- C Standard si prefigge diversi scopi
  - Definire l'architettura del C in maniera chiara e consistente
  - Favorire la portabilità del codice C, ovvero la compatibilità tra differenti implementazioni
  - Migliorare il linguaggio, anche aggiungendo nuove caratteristiche
  - Preservare i programmi C esistenti
  - Mantenere "lo spirito del C", ovvero non cambiare le caratteristiche fondamentali del linguaggio
- Tutti questi scopi sono considerati importanti
- Purtroppo, in certi casi sono in contraddizione, e allora C Standard cerca di stabilire dei compromessi

Scopi e principi dello Standard

- Gli aspetti principali dello spirito del C, vengono sintetizzati, dal Comitato ISO/IEC che redige C Standard, con le seguenti affermazioni:
  - *Trust the programmer*
  - *Don't prevent the programmer from doing what needs to be done*
  - *Keep the language small and simple*
  - *Provide only one way to do an operation*
  - *Make it fast, even if it is not guaranteed to be portable*
- Inoltre, viene riconosciuta come caratteristica fondamentale del C, la possibilità di usarlo per scrivere programmi
  - Molto efficienti
  - Che abbiano accesso a tutte le risorse di un computer

Il codice C può essere portabile, ma anche no

- Un obiettivo del C Standard è favorire la portabilità del software tra diversi computer
- Allo stesso tempo, C Standard vuole rendere possibile scrivere codice C che utilizza le risorse specifiche di un computer per ottenere una maggiore efficienza e offrire più servizi
- Quindi C Standard permette sia di scrivere programmi portabili, sia di scrivere programmi che invece usano caratteristiche non portabili
- È importante che il programmatore conosca bene C Standard per essere consapevole del fatto che il codice che scrive sia portabile o meno

C Standard è chiaro e consistente, ma anche no

- Naturalmente, il Comitato ISO/IEC si sforza di definire delle regole semantiche che siano chiare e consistenti
- Si cerca di fare in modo che le architetture del C siano concise e concettualmente semplici
- Tuttavia, viene considerato importante fare in modo che i programmi C già esistenti e realmente funzionanti, restino programmi validi in base alle regole di C Standard
- Poiché è stato scritto molto software, su computer diversi, prima che iniziasse il processo di standardizzazione, in alcuni aspetti del linguaggio, chiarezza e semplicità della semantica devono essere sacrificate per fare in modo che tale software non debba essere modificato per adeguarsi allo standard

Il linguaggio C evolve, ma anche no

- Il Comitato ISO/IEC cerca di migliorare il linguaggio, anche offrendo nuove caratteristiche, ma senza alterarne la natura fondamentale di linguaggio *piccolo* e *concettualmente semplice*
- Le caratteristiche introdotte dalle versioni più recenti di C Standard non alterano i modi fondamentali in cui si usa il C
- Inoltre, C Standard non inventa nulla
  - Nuove caratteristiche non previste da C Standard vengono introdotte da singoli traduttori C (naturalmente un programma C che usa tali caratteristiche non è conforme a C Standard e quindi non è portabile in traduttori diversi)
  - Solo dopo ampia sperimentazione, quelle caratteristiche che risultano davvero utili e prive di controindicazioni vengono incorporate in C Standard

Il linguaggio C evolve, ma anche no

- Nell'evoluzione di C Standard si cerca di tenere un buon grado di compatibilità, in particolare di backward compatibility, tra versioni diverse
- Tuttavia la backward compatibility non è totale: alcune caratteristiche vengono rimosse quando ci si rende conto che vengono utilizzate poco nel software nuovo oppure dopo essere state dichiarate *obsolete* per un lungo periodo di tempo
- Quando una caratteristica viene alterata o rimossa, si cerca di fare in modo che i programmatori ne siano consapevoli

Il linguaggio C evolve, ma anche no

- Quando possibile, si cerca di evitare incompatibilità "gratuite" con il C++
- Ovvero, si cerca di avere un sottoinsieme comune ai due linguaggi, mantenendo comunque le caratteristiche distintive di ciascuno di essi e lasciando che evolvano separatamente
- Il C++ è un linguaggio molto più grande e ambizioso del C, ed evolve in maniera molto più rapida e radicale
- Comitato ISO/IEC è ben contento di questa chiara separazione di intenti, e, sebbene alcune caratteristiche del C++ possano essere introdotte anche nel C, non ha intenzione di far diventare il C simile al C++

Le definizioni di C Standard

- Le versioni di C Standard sono testi redatti in lingua inglese che descrivono la forma e stabiliscono l'interpretazione dei programmi scritti in linguaggio C
- La forma dei programmi è descritta mediante una grammatica formale corredata da ulteriori regole e vincoli espressi in inglese
- L'interpretazione viene fornita descrivendo, in modo non formale, il comportamento di una abstract machine che esegue i programmi scritti in C

Le definizioni di C Standard

- La semantica del C è dunque espressa descrivendo, in lingua inglese, il funzionamento della abstract machine
- Si tratta quindi di una semantica
  - operazionale
  - non formale, sebbene il linguaggio usato sia piuttosto rigoroso
- La scelta di non utilizzare una semantica formale deriva dal timore che ne risultasse un documento troppo difficile da leggere per la maggior parte dei programmatori
- Ciononostante, i documenti C Standard sono testi piuttosto difficile da leggere: sono scritti con uno stile molto sintetico, che impiega convenzioni molto precise e usa i termini in modo rigoroso

Le definizioni di C Standard

- A dispetto del rigore dei testi, non sempre C Standard definisce con precisione e senza ambiguità la semantica dei costrutti del linguaggio
- In certi casi, si tratta di errori o imperfezioni nel testo
  - Per quanto rigoroso un testo in inglese non è una specifica formale, nel senso matematico del termine
  - Di tanto in tanto questi difetti vengono corretti con la pubblicazione di documenti chiamati *Technical Corrigenda* di C Standard
- In altri, invece, l'imprecisione e l'ambiguità sono deliberate e sono il risultato di una definizione parziale della semantica

Definizione Parziale della Semantica

- Il fatto che C Standard definisca volutamente la semantica di alcuni costrutti in modo parziale, molto spesso sorprende e confonde chi si sta imparando le basi del linguaggio C
- Vi sono diverse buone ragioni per definire in modo parziale la semantica di alcuni costrutti:
  - Rendere compatibili con C Standard, il maggior numero possibile di programmi realizzati prima che esso fosse adottato
  - Rendere possibile la scrittura di programmi non portabili che accedano a risorse specifiche di determinati computer (per offrire una maggior efficienza o servizi migliori)
  - Permettere alle implementazioni di fornire estensioni a C Standard pur rimanendo conformi ad esso
  - Ignorare determinati costrutti (ovvero non preoccuparsi di generare codice eseguibile corretto in risposta ad essi) che sono difficili da definire con precisione e che sono poco utili nella pratica

Definizione Parziale della Semantica

- Poiché le architetture C Standard definiscono la semantica del linguaggio in modo parziale, implementazioni diverse della stessa versione di C Standard sono compatibili tra loro in maniera parziale, ovvero possono eseguire in modo diverso un programma C
- Scrivere un programma che funziona nello stesso modo in tutte le implementazioni di una determinata versione di C Standard, richiede una conoscenza molto approfondita dell'architettura
- Scrivere un programma che funziona nello stesso modo in tutte le implementazioni di tutte le versioni di C Standard, richiede una conoscenza estremamente approfondita delle differenze tra le architetture C Standard

Definizione Parziale della Semantica

- Come detto in precedenza, C Standard descrive la semantica di un costrutto del linguaggio illustrando il comportamento (in inglese *behavior*) della abstract machine quando esegue il costrutto
- Dunque, in certi casi il behavior della abstract machine è definito in modo parziale
- Più precisamente, in relazione al grado di definitezza, C Standard individua 5 diverse tipologie di behavior
  - *well-defined* behavior
  - *locale-specific* behavior
  - *unspecified* behavior
  - *implementation-defined* behavior
  - *undefined* behavior

Well-defined Behavior

- Un well-defined behavior è un comportamento completamente definito da C Standard
- Tutti i costrutti del seguente programma, ad esempio, hanno well-defined behavior

Code1

```c
#include <stdio.h>

int main(void) {
  int x, y = 42;

  x = y;
  return x;
}
```

Locale-specific Behavior

- Lo strumento principale introdotto dall'*Amendment 1* per il controllo dell'internazionalizzazione è il concetto di *locale* (in italiano *localizzazione*)
- Il locale è un insieme di parametri, modificabili anche durante l'esecuzione di un programma, che supportano l'internazionalizzazione dei programmi in C permettendo di adattare alcuni aspetti del linguaggio alle convenzioni di uno specifico gruppo di utenti
- Esempi degli aspetti che possono dipendere dal locale sono l'insieme di caratteri usato per l'input-output e il formato di numeri, date e quantità monetarie

Locale-specific Behavior

- Un locale-specific behavior è un comportamento che dipende dalla configurazione del locale
- Ad esempio, il fatto che la funzione `islower` della libreria Standard restituisca `1` per qualche valore del parametro diverso da uno dei 26 caratteri minuscoli dell'alfabeto latino, è un locale-specific behavior
- I locale-specific behavior sono tutti legati all'esecuzione di funzioni della libreria Standard, non al linguaggio vero e proprio, pertanto non ce ne occupiamo in LPS
- Il capitolo 25 di **[Ki]** fornisce ulteriori informazioni sull'internazionalizzazione, i locale e i locale-specific behavior

Unspecified Behavior

- Un unspecified behavior è l'uso di un valore non specificato o un altro comportamento per il quale C Standard presenta due o più possibilità e nessun vincolo su quale di essa debba, ad ogni occasione, essere scelta dall'implementazione
- Uno dei casi più comuni di unspecified behavior è relativo all'ordine di valutazione delle sotto-espressioni di una stessa espressione, che usiamo come esempio per chiarire il concetto

Unspecified Behavior

- Un modo efficace di descrivere l'ordine in cui vengono valutate le sotto-espressioni di una espressione, è mostrare la traduzione dell'istruzione in una sequenza di istruzioni in stile UC equivalente
- Code2_pc è un'istruzione plain C, in cui le sotto-espressioni, in base alle regole di precedenza degli operatori, vengono valutate nell'ordine in cui compaiono nella traduzione UC dell'istruzione mostrata in Code2_uc_1

Code2_pc

```
x = 4 + z / 3;
```

Code2_uc_1

```
int t;
t = z / 3;
x = 4 + t;
```

Unspecified Behavior

- Naturalmente, una implementazione di C Standard si può dire corretta solo se nel tradurre Code2_pc produce un codice eseguibile che ha una semantica equivalente a quella di Code2_uc_1
- Se una implementazione valutasse le sotto-espressioni di Code2_pc nell'ordine in cui compaiono nella traduzione UC **non** corretta Code2_uc_2, allora l'implementazione **non** sarebbe corretta

Code2_pc

```
x = 4 + z / 3;
```

Code2_uc_2

```
int t;
t = 4 + z;
x = t / 3;
```

Unspecified Behavior

- Nel caso dell'espressione `4 + z / 3`, l'ordine di valutazione delle sotto-espressioni è completamente determinato dalle regole di precedenza degli operatori, ma ciò non è sempre vero
- Infatti, le regole di C Standard stabiliscono che per tutti gli operatori binari, ad eccezione dell'operatore di concatenazione (`,`) e degli operatori logici (`&&` e `||`), l'ordine di valutazione delle due espressioni che costituiscono gli operandi è non specificato; ovvero le implementazioni possono valutare i due operandi in un qualunque ordine
- Ad esempio, per valutare, un'espressione del tipo `E1 + E2`, dove `E1` e `E2` sono due sotto-espressioni arbitrariamente complesse, un'implementazione può scegliere di valutare prima `E1` e poi `E2`, oppure viceversa

Unspecified Behavior

- Code3_pc è un esempio concreto

Code3_pc

```
x = y * 4 + z % 3;
```

- Le due seguenti traduzioni in stile UC sono ugualmente corrette

Code3_uc_1

```
int t1, t2;
t1 = y * 4;
t2 = z % 3;
x = t1 + t2;
```

Code3_uc_2

```
int t1, t2;
t2 = z % 3;
t1 = y * 4;
x = t1 + t2;
```

Unspecified Behavior

- Dunque, qualunque espressione contenga al suo interno una sotto-espressione formata mediante un operatore binario, diverso dalle 3 eccezioni menzionate prima, ha un unspecified behavior
- A prima vista, questo fatto non sembra molto importante per un programmatore: in fin dei conti cosa importa in quale ordine vengono valutati gli operandi di un operatore binario?
- Nella maggioranza dei casi, in effetti, la cosa è irrilevante in quanto il risultato è comunque lo stesso
- Ma non sempre è così

Unspecified Behavior

- Consideriamo il programma plain C Code4_pc
- Un'implementazione potrebbe produrre codice equivalente alla traduzione UC Code4_uc_1
- **Al termine dell'esecuzione di tale codice, `j` vale 23**

Code4_pc

```
int i = 2, j;
int g( int x ) {
  i += x;
  return 3 * i; }
int main( void ) {
  j = i + g( 5 );
  return 0; }
```

Code4_uc_1

```
int i = 2, j;
int g( int x ) {
  i += x;
  return 3 * i; }
int main( void ) {
  int t;
  j = i;
  t = g( 5 );
  j += t;
  return 0; }
```

Unspecified Behavior

- Un'altra implementazione, invece potrebbe produrre codice equivalente alla traduzione UC Code4_uc_2
- **Al termine dell'esecuzione di Code4_uc_2, `j` vale 28**

Code4_pc

```
int i = 2, j;
int g( int x ) {
  i += x;
  return 3 * i; }
int main( void ) {
  j = i + g( 5 );
  return 0; }
```

Code4_uc_2

```
int i = 2, j;
int g( int x ) {
  i += x;
  return 3 * i; }
int main( void ) {
  int t;
  t = g( 5 );
  j = i + t;
  return 0; }
```

Unspecified Behavior

- Si noti che Code4_pc, non viola alcuna regola di C Standard, eppure è possibile che due diverse implementazioni producano risultati diversi eseguendolo
- Infatti C Standard dichiara esplicitamente che l'ordine di valutazione delle sotto-espressioni è non specificato e dunque lascia deliberatamente alle implementazioni la libertà di usare l'ordine che preferiscono
- È interessante provare a pensare al perché C Standard conceda tale libertà alle implementazioni e perché altri linguaggi, come Java, invece no

Unspecified Behavior

- Il motivo è che, in tal modo, ogni implementazione può utilizzare l'ordine di valutazione delle sotto-espressioni che le permette di ottenere la maggiore efficienza
- Infatti, l'ordine più efficiente per valutare sotto-espressioni può essere diverso, per implementazioni diverse, in quanto dipende dalle caratteristiche del *LM*
- Dunque siamo in presenza di una regola di C Standard, che
  - promuove la ricerca di efficienza nelle implementazioni
  - anche a prezzo di permettere che alcuni programmi C producano risultati diversi in implementazioni diverse
- Si noti la differenza rispetto alla filosofia *Write once, run anywhere* che è alla base di Java

Unspecified Behavior

- L'ordine di valutazione delle sotto-espressioni è solo un esempio di unspecified behavior: ve ne sono molti altri in C Standard
- Definiamo *portabile* (*portable* in inglese) un programma C che produce gli stessi risultati su tutte le implementazioni
- Gli esempi precedenti ci hanno mostrato che esistono molti programmi C che hanno unspecified behavior
  - molti di essi sono portabili
  - alcuni invece non sono portabili, proprio a causa dei loro unspecified behavior
- Di conseguenza definiamo
  - *portable unspecified behavior* gli unspecified behavior che non pregiudicano la portabilità
  - *non-portable unspecified behavior* gli unspecified behavior che rendono non portabile un programma

Implementation-defined Behavior

- Come abbiamo appena detto, un unspecified behavior è un comportamento per cui C Standard presenta due o più possibilità e lascia alle implementazioni libertà di scegliere una di esse
- Un implementation-defined behavior è un unspecified behavior ma con un vincolo in più per le implementazioni
- Anche nel caso di un implementation-defined behavior C Standard presenta due o più possibilità e le implementazioni possono scegliere liberamente quale di esse realizzare

Implementation-defined Behavior

- Ma nel caso degli implementation defined behavior le implementazioni hanno un vincolo aggiuntivo: documentare quale, tra le possibilità permesse da C Standard, viene scelta
- Per ogni implementation-defined behavior, la documentazione dell'implementazione deve dichiarare quale, tra le possibili semantiche indicate da C Standard, è quella che viene realizzata dall'implementazione
- Come nel caso degli unspecified behavior, distinguiamo tra
  - *portable implementation-defined behavior*: implementation-defined behavior che non pregiudicano la portabilità
  - *non-portable implementation-defined behavior*: implementation-defined behavior che rendono non portabile un programma

Undefined Behavior

- Un undefined behavior è un comportamento che C Standard non definisce, nel modo più assoluto
- Ciò significa che un'implementazione lo può realizzare in qualunque modo preferisca, senza alcun vincolo da parte di C Standard
- Alcuni esempi di possibili realizzazioni di un undefined behavior
  - provare a fornire un risultato sensato ad un'operazione
  - fornire un risultato casuale ad un'operazione
  - produrre un messaggio diagnostico
  - non fare nulla
  - terminare l'esecuzione del programma
  - bloccare l'esecuzione del programma nella ripetizione infinita di un'azione
  - *"make demons fly out of your nose"*

Undefined Behavior

- Presentiamo un primo esempio di undefined behavior nel contesto della valutazione di espressioni, in una situazione simile (ma non uguale) a quella dell'esempio con cui abbiamo introdotto il concetto di unportable unspecified behavior
- Come passo preliminare, abbiamo bisogno di introdurre un altro aspetto della valutazione delle espressioni
- Come sappiamo (cap. 4 di **[Ki]**) in C esistono degli operatori che hanno un *effetto collaterale* (in inglese *side effect*), ovvero oltre a calcolare un valore, modificano uno dei loro operandi
  - operatori di assegnamento, semplice o composti
  - operatori di incremento e decremento, prefissi o postfissi
- Ad esempio l'espressione a++ calcola un valore (il valore di a prima che avvenga l'effetto collaterale) e produce il side effect di aumentare a di 1

Undefined Behavior

- È spesso intuitivo, per chi non conosce a fondo C Standard, pensare che il calcolo del valore e il side effect di uno di tali operatori costituiscano necessariamente una sequenza indivisibile di operazioni elementari, analoghe alle istruzioni ammesse in UC
- Ma non è così: tra l'una e l'altra possono essere inserite operazioni elementari necessarie alla valutazione di altre sotto-espressioni
- Il side effect deve avvenire dopo il calcolo del valore, ma non ha alcun vincolo di precedenza rispetto a operazioni elementari di altre sotto-espressioni

Undefined Behavior

- Ad esempio, si consideri l'espressione `x = y++ * ( z + 5 )`: l'idea intuitiva è che in una traduzione in UC calcolo del valore e side effect debbano essere istruzioni consecutive, come in Code5_uc_1; ma anche la traduzione Code5_uc_2 è corretta

Code5_uc_1

```
int t1, t2;
t1 = y; // calcolo valore y++
y += 1; // side effect y++
t2 = z + 5;
x = t1 * t2;
```

Code5_uc_2

```
int t1, t2;
t1 = y; // calcolo valore y++
t2 = z + 5;
x = t1 * t2;
y += 1; // side effect y++
```

- Gli effetti finali delle due traduzioni sono comunque gli stessi, quindi `x = y++ * ( z + 5 )` ha un portable unspecified behavior

Undefined Behavior

- Vediamo adesso cosa accade combinando gli effetti della non-indivisibilità dei side effect dal calcolo dei valori dei relativi operatori, con il fatto che l'ordine di valutazione delle sotto-espressioni non è specificato
- Code6_uc_1 mostra una traduzione UC di Code6_pc, in cui l'operando sinistro di * è valutato prima del destro e i side effect sono vicini al calcolo dei rispettivi valori

Code6_pc

```
int z, x = 4, b = 1;
z = ( ++x - b ) * ( x -= 3 );
```

Code6_uc_1

```
int z, x = 4, b = 1, t;
x += 1;   t = x - b;
x -= 3;
z = t * x;
```

- Al termine di Code6_uc_1, z vale 8 e x vale 2

Undefined Behavior

- Non ci sorprende, ormai, il fatto che valutando gli operandi dell'operatore $*$ nell'altro ordine possibile, ma sempre tenendo i side effect vicini al calcolo dei rispettivi valori, si ottengano effetti diversi

Code6_pc

```
int z, x = 4, b = 1;
z = ( ++x - b ) * ( x -= 3 );
```

Code6_uc_2

```
int z, x = 4, b = 1, t1, t2;
x -= 3;   t2 = x;
x += 1;   t1 = x - b;
z = t1 * t2;
```

- Al termine di Code6_uc_2, z vale 1 e x vale 2

Undefined Behavior

- Ora separiamo il side effect dal calcolo del valore, in `++x`

Code6_pc

```
int z, x = 4, b = 1;
z = ( ++x - b ) * ( x -= 3 );
```

Code6_uc_3

```
int z, x = 4, b = 1, t1, t2;
t1 = x + 1;  t2 = t1 - b;
x -= 3;
z = t2 * x;
x = t1;
```

- Al termine di Code6_uc_3, `z` vale 4 e `x` vale 5

Undefined Behavior

- Ancora un'altra traduzione UC, ottenuta separando il side effect dal calcolo del valore in `x -= 3`

Code6_pc

```
int z, x = 4, b = 1;
z = ( ++x - b ) * ( x -= 3 );
```

Code6_uc_4

```
int z, x = 4, b = 1, t1, t2;
t2 = x - 3;
x += 1;  t1 = x - b;
z = t1 * t2;
x = t2;
```

- Al termine di Code6_uc_4, `z` vale 4 e `x` vale 1

Undefined Behavior

- Dovrebbe essere a questo punto chiaro che combinare, in una stessa espressione, operatori con side effect applicati alla stessa variabile, può portare a un numero molto grande di possibili traduzioni, persino per un'espressione relativamente semplice come quella in Code6_pc
- Servirebbe a poco individuare tutte le possibili traduzioni
- Pertanto C Standard affronta la questione in maniera più semplice, dando la seguente regola

Regola Un-D-SE

Se un'espressione che produce un side effect su una variabile V non è vincolata ad essere *eseguita interamente* prima dell'inizio o dopo del termine dell'esecuzione di altre espressioni che producono side effect su V, oppure calcolano dei valori leggendo V, allora si ha un undefined behavior

Undefined Behavior

- Nel caso di Code6_pc, la regola Un-D-SE si applica perché in `z = ( ++x - b ) * ( x -= 3 )` vi sono due operatori che producono un side effect su `x` e non vi è alcun vincolo che imponga che l'esecuzione di uno dei due avvenga interamente prima di quella dell'altro
- Pertanto Code6_pc ha un undefined behavior e dunque un'implementazione ha licenza di tradurre Code6_pc in un codice eseguibile che fa qualunque cosa
- Si osservi che diversamente dagli unspecified e implementation-defined behavior, per definizione tutti gli undefined behavior rendono non portabile un programma

Undefined Behavior

- In generale, la regola Un-D-SE si applica quando operatori con side effect si trovano immersi in espressioni più ampie in cui la variabile modificata dal side effect è presente anche in altre sotto-espressioni
- È importante imparare a individuare (e quindi a evitare) i casi in cui Un-D-SE si applica, perché tali casi producono undefined behavior
- Fortunatamente ci sono diverse regole che impongono che l'esecuzione di una espressione, e in particolare quella dei suoi side effect, termini prima dell'inizio dell'esecuzione di altre espressioni, evitando quindi l'applicazione di Un-D-SE

Undefined Behavior

- La più importante riguarda le espressioni non contenute in espressioni più grandi

Regola Comp-SE-E

Nella valutazione di un'espressione E che non è sotto-espressione di un'espressione più grande, tutti i side effect generati dalle sotto-espressioni, terminano prima che termini la valutazione di E

Undefined Behavior

- Comp-SE-E ci dice, ad esempio, che Code7_pc, pur contenendo le espressioni `++x` e `x *= 2` che modificano la stessa variabile, non ha undefined behavior

Code7_pc

```
z = ( ++x - b ) * ( a -= 3 );
x *= 2;
```

- L'espressione `z = ( ++x - b ) * ( a -= 3 )` non è contenuta in espressioni più grandi, quindi i side effect generati in essa, tra cui la modifica di `x` generata da `++x`, terminano prima che termini l'esecuzione della prima riga di Code7_pc
- Poiché il side effect generato da `++x` avviene prima che inizi la valutazione di `x *= 2`, non si applica Un-D-SE e Code7_pc non ha undefined behavior

Undefined Behavior

- Altre regole sulla terminazione dei side effects di alcune espressioni, riguardano le chiamate di funzione
- Comp-SE-F1 garantisce che Code8_pc non abbia undefined behavior, e che quando inizia l'esecuzione di `f` il parametro abbia valore pari a 2

Regola Comp-SE-F1

Tutti i side effect generati dalla valutazione degli argomenti in una chiamata di funzione, terminano prima che inizi la chiamata di funzione

Code8_pc

```
int x = 1, y;
int f( a ) {
  x *= 5;
  return x - a;
}
int main( void ) {
  y = f( ++x );
  return 0;
}
```

Undefined Behavior

- Comp-SE-F2 garantisce che Code9_pc non abbia undefined behavior, ma "solo" un non-portable unspecified behavior

Regola Comp-SE-F2

Sia `f1` una funzione nel cui corpo compare una chiamata ad un'altra funzione `f2`. Tutti i side effect eseguiti in `f1` e tutte le valutazioni di espressioni eseguite in `f1`, diverse dalla chiamata di `f2`, vengono *eseguiti interamente* prima dell'inizio o dopo del termine dell'esecuzione del corpo di `f2`

Code9_pc

```
int x = 1, y;
int f( a ) {
  x *= 2;
  return x - a;
}
int main( void ) {
  y = ++x * f( 4 );
  return 0;
}
```

Undefined Behavior

- Comp-SE-F3 garantisce che Code10_pc non abbia undefined behavior e che al termine dell'esecuzione `x` contenga il valore 6

Regola Comp-SE-F3

Tutti i side effect generati da espressioni contenute in una funzione, terminano prima che termini l'esecuzione della funzione

Code10_pc

```
int x, y = 3;
int f( a ) {
  return ( x = 2 ) * a;
}
int main( void ) {
  x += f( 2 );
  return 0;
}
```

- In future lezioni di LPS incontreremo regole relative agli operatori logici, condizionale e di concatenazione, sulla terminazione dei side effects di alcune espressioni

Esempio di Analisi dei Behavior

- I programmi CodeA_pc e CodeB_pc sono piuttosto simili

CodeA_pc

```
int x = 3, y;
int f( a ) {
  return ( x *= a ) % 2;
}
int main( void ) {
  y = 8 * x - f( 3 );
  return 0;
}
```

CodeB_pc

```
int x = 3, y;

int main( void ) {
  y = 8 * x - ( x *= 3 ) % 2;
  return 0;
}
```

- La differenza sostanziale è che `( x *= 3 ) % 2`, in CodeB_pc è una sotto-espressione dell'assegnamento che modifica `y`, mentre in CodeA_pc è incapsulata in una funzione
- I due programmi modificano `x` in un modo sospetto; è naturale chiedersi se hanno lo stesso behavior e se sono portabili

Esempio di Analisi dei Behavior

- Analisi CodeA_pc

CodeA_pc

```
int x = 3, y;
int f( a ) {
  return ( x *= a ) % 2;
}
int main( void ) {
  y = 8 * x - f( 3 );
  return 0;
}
```

- La sotto-espressione `x *= a` è contenuta in una funzione
- L'esecuzione di `f`, rispetto al calcolo di `8 * x`, termina prima o inizia dopo
- Il side effect che modifica `x` inizia dopo l'inizio di `f` e termina prima che termini `f`
- Quindi il side effect che modifica `x` termina prima oppure inizia dopo, rispetto al calcolo di `8 * x`
- Dunque Un-D-SE non si applica e CodeA_pc ha "solo" un non-portable unspecified behavior

Esempio di Analisi dei Behavior

- Analisi CodeB_pc

CodeB_pc

```
int x = 3, y;

int main( void ) {
  y = 8 * x - ( x *= 3 ) % 2;
  return 0;
}
```

- In `8 * x - ( x *= 3 ) % 2` la variabile `x` viene letta per calcolare il valore di `8 * x` e inoltre viene letta e modificata in `x *= 3`
- Il side effect che modifica `x` non è vincolato a terminare prima o a iniziare dopo rispetto al calcolo di `8 * x`
- Quindi si applica Un-D-SE, e dunque CodeB_pc ha undefined behavior
- I due programmi hanno behavior diverso ma sono entrambi non portabili

Considerazioni Generali sui Behavior

- Un programma che contenga undefined behavior potrebbe, in base a C Standard, fare qualunque cosa qualora venisse eseguito
- Quindi, a meno di casi molto particolari, bisognerebbe evitare undefined behavior in programmi C
- Più in generale, l'indagine sulle tipologie di behavior della abstract machine, ha evidenziato come esse influenzano in maniera importante la portabilità di un programma
- Chiamiamo *non-portable behavior*, i behavior che appartengono ad una delle seguenti categorie
  - non-portable unspecified behavior
  - non-portable implementation-defined behavior
  - undefined behavior
- Un programma è portabile su tutte le implementazioni di C Standard se e solo se non ha non-portable behavior

Considerazioni Generali sui Behavior

- La notevole importanza della portabilità del software, rende opportuno evitare o limitare al minimo indispensabile i non-portable behavior presenti nei programmi
- In modo particolare vanno evitati gli undefined behavior, in quanto essi sono completamente imprevedibili in base alle regole di C Standard e nella pratica sono spesso origine di malfunzionamenti
- È naturale chiedersi cosa si possa fare per riconoscere i non-portable behavior, per individuarli in programmi già esistenti ed evitarli nella scrittura di nuovi programmi
- La soluzione ideale sarebbe uno strumento per individuare in modo automatico i non-portable behavior, attraverso dei *checking*: per esplorare questa possibilità studiamo il rapporto tra regole e checking in C Standard

Regole di C Standard

- Le regole enunciate in C Standard, ricadono in una delle seguenti categorie:
  - *Regole sintattiche*, descritte mediante una grammatica formale
  - *Constraints*, descritte in inglese
  - *Regole semantiche*, descritte in inglese
- I constraints sono regole sintattiche o semantiche descritte in modo non formale
- I constraints e le regole semantiche specificano i *behavior* della abstract machine nell'eseguire i costrutti di C Standard

Violazione delle Regole in C Standard

- L'aspettativa di buona parte di coloro che si accostano alla programmazione in C, in relazione alla violazione delle regole è

  Se scrivo un programma che viola una regola del linguaggio, il compilatore lo rileva e mi avverte attraverso un messaggio di errore, che mi aiuterà a correggere il programma. Inoltre, non verrà prodotto codice eseguibile, quindi il programma scorretto non potrà essere eseguito.

- La realtà è un po' diversa

Violazione delle Regole in C Standard

- Innanzitutto, C Standard non definisce il termine *errore*
- Naturalmente considera la questione di cosa debba accadere se un programma viola una o più regole del linguaggio, ma lo fa assumendo il punto di vista delle implementazioni, non quello dei programmatori
- C Standard sancisce che se un programma C non viola le regole del linguaggio, allora una implementazione deve tradurre con successo ed eseguire il programma *come se* esso fosse eseguito dalla abstract machine
- Se invece un programma C viola una o più regole, ciò che un'implementazione è obbligata a fare dipende dalla categoria cui la regola appartiene

Violazione di Regole Sintattiche e Constraint

- In relazione alle violazioni di regole sintattiche e di constraint, C Standard impone un solo vincolo

Vincolo Violazione Regole

Se un programma contiene almeno una violazione di una regola sintattica o di un constraint, un'implementazione deve produrre, in fase di traduzione, almeno un *messaggio diagnostico*

- Si noti che
  - C Standard non descrive esattamente cosa sia un messaggio diagnostico, anzi, su questo aspetto lascia totale libertà: un'implementazione potrebbe persino stampare un messaggio senza senso o limitarsi a produrre un suono
  - C Standard non proibisce all'implementazione di produrre codice eseguibile come effetto della traduzione di un programma che viola regole sintattiche o constraint

Violazione di Regole Sintattiche e Constraint

- È evidente il vincolo imposto da C Standard alle implementazioni è piuttosto lasco
- Nel caso in cui un programma contenga almeno una violazione di una regola sintattica o di un constraint, un'implementazione ha a disposizione, ad esempio, le seguenti alternative:
  - Produrre il messaggio diagnostico e interrompere la traduzione, senza creare il codice eseguibile del programma
  - Produrre il messaggio diagnostico, continuare la traduzione fino al termine, per individuare il maggior numero possibile di errori ma senza creare il codice eseguibile del programma
  - Produrre il messaggio diagnostico, continuare la traduzione fino al termine creando il codice eseguibile del programma

Violazione di Regole Semantiche

- In relazione alle violazioni di regole semantiche, C Standard non impone nessun vincolo alle implementazioni
- Pertanto
  - Le implementazioni non sono obbligate a rilevare e segnalare al programmatore (attraverso messaggi diagnostici) le violazioni di regole semantiche
  - Le implementazioni sono libere di produrre codice eseguibile come effetto della traduzione di un programma che viola regole semantiche
- La differenza fondamentale tra i constraint e le regole semantiche è proprio negli obblighi che ha l'implementazione in caso di violazione

Violazione di Regole e Undefined Behavior

- Dunque, se un programma C viola una o più regole, C Standard non proibisce che esso venga tradotto ed eseguito, qualunque sia la categoria delle regole violate
- Cosa accade se si esegue un programma che viola delle regole?

Violazione di Regole e Undefined Behavior

- Dunque, se un programma C viola una o più regole, C Standard non proibisce che esso venga tradotto ed eseguito, qualunque sia la categoria delle regole violate
- Cosa accade se si esegue un programma che viola delle regole?
- C Standard sancisce che il programma ha undefined behavior
- Pertanto, come sappiamo, l'esecuzione di tale programma può fare qualunque cosa, sensata o meno, utile o dannosa

Individuare i Non-portable Behavior

- A questo punto possiamo specificare meglio quali sono le circostanze che possono dare luogo ad un undefined behavior
- Un costrutto C ha un undefined behavior se si verifica una delle seguenti circostanze
  1. C Standard sancisce esplicitamente che il costrutto ha undefined behavior
  2. C Standard non definisce esplicitamente la semantica del costrutto
  3. Il costrutto viola una regola semantica di C Standard

Individuare i Non-portable Behavior

- Torniamo quindi alla questione dell'individuazione dei non-portable behavior
- Come abbiamo appena detto
  - C Standard non obbliga le implementazioni a rilevare e segnalare le violazioni di regole semantiche
  - Le violazioni di regole (anche) semantiche producono undefined behavior
  - Gli undefined behavior sono casi particolari di non-portable behavior

Individuare i Non-portable Behavior

- Dunque, non ci coglie di sorpresa il fatto che C Standard non obbliga le implementazioni a rilevare e segnalare un qualunque tipo di non-portable behavior
- La risposta di C Standard alla questione di come individuare i non-portable behavior è: affidarsi al programmatore
  - Per individuare i non-portable behavior, il programmatore non può contare su meccanismi di rilevazione automatica da parte del traduttore
  - Il programmatore deve individuare (ed eventualmente evitare) i non-portable behavior conducendo un'attenta analisi del codice, basata sulla propria conoscenza del linguaggio C

Regole e Behavior nella Pratica

- Da quanto detto, emerge chiaramente il fatto che C Standard impone alle implementazioni di fornire ai programmatori solo un supporto minimo per l'individuazione e la correzione delle violazioni delle regole
- Ciò riflette la natura di C Standard, di linguaggio permissivo e rivolto a programmatori esperti
- Tuttavia, la qualità e l'estensione del supporto fornito per l'individuazione e la correzione dei non-portable behavior sono caratteristiche che possono far preferire un'implementazione rispetto ad altre

Regole e Behavior nella Pratica

- Quindi, nella pratica, le implementazioni offrono ai programmatori maggior supporto rispetto a quello che C Standard gli impone di fornire
- Ad esempio
  - messaggi diagnostici significativi
  - la traduzione non produce codice eseguibile in caso di violazione di regole sintattiche o constraint
  - messaggi diagnostici anche nel caso di (alcune) violazioni di regole semantiche, anche a tempo di esecuzione
  - messaggi diagnostici anche nel caso di (alcuni) non-portable behavior, anche a tempo di esecuzione

Regole e Behavior nella Pratica

- In ogni caso, non si deve mai dimenticare che poiché tra gli obiettivi primari delle implementazioni c'è quello di generare codice eseguibile efficiente, le implementazioni di C Standard tendono ad aiutare i programmatori molto meno di quanto non facciano le implementazioni di altri linguaggi
- In particolare è molto raro che implementazioni C facciano dei controlli sulla violazione di regole a tempo di esecuzione, e se lo fanno è quasi sempre una caratteristica opzionale
- È noto invece che in altri linguaggi le cose sono diverse: in particolare, molte violazioni delle regole in Java vengono segnalate in fase di esecuzione tramite il sollevamento di un'eccezione (ad esempio, accesso a un elemento di un array con un indice non valido)

Esempio su Regole e Behavior: operatori / e %

- Le regole relative agli operatori / e % illustrano bene le varie tipologie di categorie di regole e di behavior
- Constraints:
  - Entrambi gli operandi devono avere tipo aritmetico
  - Nel caso di %, entrambi gli operandi devono avere un tipo intero
- Poiché le regole precedenti sono constraints, se esse vengono violate l'implementazione produce un messaggio diagnostico (tipicamente segnala un errore in fase di traduzione)

Esempio su Regole e Behavior: operatori / e %

- Regole semantiche
  - Quando l'operando destro è diverso da 0, il risultato di / è il quoziente della divisione tra l'operando sinistro e il destro, mentre il risultato di % è il resto di tale divisione: well-defined behavior
  - Se l'operando destro vale 0, si ha un undefined behavior: il programma può comportarsi in qualunque modo, in particolare in modo erroneo, senza alcuna segnalazione da parte dell'implementazione

Esempio su Regole e Behavior: operatori / e %

- Regole semantiche aggiuntive, per il caso in cui i due operandi sono interi
  - In C99, C11 e C18, se entrambi gli operandi sono interi il risultato di / è ottenuto togliendo la parte frazionaria al quoziente: well-defined behavior
  - In C89, se entrambi gli operandi sono interi positivi il risultato di / è ottenuto togliendo la parte frazionaria al quoziente: well-defined behavior
  - In C89, se entrambi gli operandi sono interi ed almeno uno di essi è negativo, si ha un implementation-defined behavior: l'implementazione può restituire come risultato più grande tra gli interi che sono minori del quoziente, oppure il più piccolo tra gli interi che sono più grandi del quoziente; si noti che poiché si ha un implementation-defined behavior e non un unspecified behavior, la documentazione dell'implementazione deve descrivere quale, tra le due possibilità, si verifica

Sessione di Esercizi: C_Standard

- Svolgere il gruppo di esercizi **C_Standard**
- Per ulteriori spiegazioni sui contenuti degli esercizi, si veda il capitolo 4 di **[Ki]**