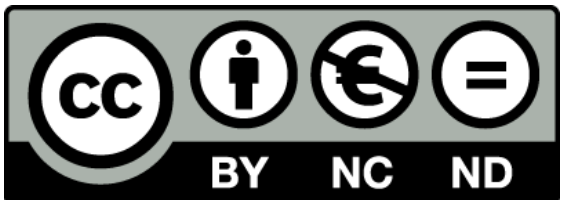
Laboratorio di Programmazione di Sistema

Tipi Base in C

Luca Forlizzi, Ph.D.

Versione 23.1



Luca Forlizzi, 2023

Tipi di Dato

- È utile che un'architettura offra molteplici possibilità per memorizzare dati, tra cui scegliere la più opportuna in funzione
  - delle esigenze del programma da realizzare
  - delle caratteristiche della memoria del sistema (quantità, efficienza, ...)
- Infatti, come sappiamo, molti *ASM-PM* permettono di memorizzare e fare operazioni aritmetiche su dati di diversi formati, riflettendo quel che accade a livello della *ISA* legata all'*ASM-PM*

Tipi di Dato

- Nella maggior parte degli *HLL*, la gestione di informazioni che possono avere natura differente, avviene attraverso la definizione un insieme di *tipi di dato*
- Un *tipo di dato* è un'entità che identifica un insieme di possibili valori che possono essere memorizzati e le operazioni che possono essere applicate a tali valori
- Adottando la terminologia di C Standard (e non quella dei linguaggi di programmazione a oggetti), chiamiamo *oggetto* un insieme di bit che permette di memorizzare uno dei valori appartenenti ad un tipo
- In particolare, un *oggetto di tipo T* è un oggetto che permette di memorizzare uno dei valori appartenenti ad un tipo T

Tipi di Dato

- Ciascun valore appartenente ad un tipo, viene memorizzato scrivendo una determinata stringa binaria, detta *rappresentazione* di tale valore, in un oggetto
- Un valore appartenente ad un tipo, può avere più di una rappresentazione, sebbene ciò non sia desiderabile
- Due diversi valori appartenenti allo stesso tipo, hanno necessariamente rappresentazioni diverse
- Può accadere che un oggetto contenga una stringa binaria s che non rappresenta nessun valore di un determinato tipo T: in tal caso, s è una *trap representation* per T
- La *rappresentazione* di un tipo T è l'insieme delle regole del linguaggio che descrivono quali sono le rappresentazioni dei valori di T

Tipi di Dato

- Spesso un *HLL* definisce più tipi di dato in grado di memorizzare informazioni che hanno la stessa natura, in modo da offrire diverse possibili scelte ai programmatori
- Ad esempio, ci possono essere diversi tipo per interi e diversi tipi per numeri floating point, che hanno insiemi di valori diversi e rappresentazioni diverse

Tipi di Dato

- Se, come accade in molti *HLL*, le definizioni dei tipi di dato sono pensate prevalentemente in funzione delle esigenze dei programmatori, si può facilitare il processo di sviluppo del software, ma si corre il rischio di non permettere un uso efficiente della memoria
- Java, per favorire la massima portabilità dei programmi, definisce in maniera molto dettagliata i propri tipi di dato
  - Ad esempio, per ciascuno dei tipi interi, l'architettura di Java definisce precisamente l'insieme di valori ammissibili e la rappresentazione
  - Tali definizioni sono state scelte anche in modo da permettere una gestione piuttosto efficiente della memoria, nelle *implementazioni basate sulle ISA comunemente usate oggi*
  - Ci sarebbero invece problemi se si volesse realizzare un'implementazione di Java per una *ISA* "esotica"

Tipi di Dato

- C Standard segue un approccio diverso sia da Java sia da altri *HLL*, che cerca di bilanciare e soddisfare molteplici esigenze
  - quelle dei programmi da realizzare
  - quella di favorire la portabilità tra implementazioni per *ISA* differenti (anche obsolete o "esotiche")
  - quella di permettere di ottenere, su tutte le implementazioni, efficienza e flessibilità nell'uso della memoria paragonabili a quelle offerte dai *LM* e dagli *ASM*
  - quelle di garantire la compatibilità con programmi scritti in passato, che utilizzano i tipi presenti in C Tradizionale
- C Standard definisce un ampio numero di tipi ma non stabilisce in modo esatto tutte le loro caratteristiche, lasciando diverse possibilità di scelta nella realizzazione delle implementazioni

Tipi di Dato

- Tecnicamente, ciò si traduce nel fatto che molte importanti caratteristiche dei tipi sono *implementation-defined*
- Di conseguenza, le implementazioni hanno la possibilità di utilizzare ciascuno dei formati di dato disponibili nella *ISA* su cui si basano, e ciascuna interpretazione di dato possibile per ognuno di essi, come rappresentazione di un diverso tipo del C
- Sulle implementazioni che colgono questa possibilità, i programmi C possono sfruttare tutte le potenzialità offerte da una *ISA* in termini di formati di dato e interpretazioni di dato
- Lo svantaggio dell'approccio di C Standard è che, poiché uno stesso tipo può avere caratteristiche diverse in implementazioni diverse, la portabilità dei programmi C non è automatica

I Tipi del Linguaggio C

- Nel linguaggio C, i *tipi base* sono i tipi di dato la cui definizione non dipende da definizioni di altri tipi di dato
- I tipi base possono essere classificati in sottogruppi
  - *tipi interi*
  - *tipi floating*
  - *tipi carattere*
- Questa presentazione introduce in modo sintetico i tipi base di C Standard e le loro caratteristiche più rilevanti per gli scopi del LPS
- Il Capitolo 7 di **[Ki]** li descrive in modo più esteso e maggiormente orientato allo sviluppo di applicazioni, che è necessario conoscere per completare la preparazione

I Tipi del Linguaggio C

- Altri tipi del linguaggio C, che verranno descritti nelle prossime presentazioni
  - *tipo* `void`
  - *tipi enumerati*
  - *tipi derivati*
    - *tipi array*
    - *tipi struttura*
    - *tipi unione*
    - *tipi puntatore*
- I tipi enumerati, che descriveremo in dettaglio in presentazioni successive, hanno valori interi
- L'insieme dei *tipi aritmetici* è l'unione dei tipi base e dei tipi enumerati
- Verranno descritte alcune regole valide per tutti i tipi aritmetici, anche prima di introdurre i tipi enumerati

I Tipi del Linguaggio C

- La rappresentazione dei tipi di dato è non specificata da C Standard, tranne che per alcune regole
- Una importante regola generale è che ogni oggetto deve essere formato da una o più parole della *ISA* usata dall'implementazione

Tipi Interi in C Standard

- C Standard mette a disposizione dei programmatori numerosi tipi di dato per numeri interi
  - una serie di tipi per rappresentare intervalli di interi positivi e negativi, chiamati collettivamente *signed integer types* (in breve tipi *signed*), tra cui
    - 5 tipi obbligatoriamente presenti in ogni implementazione, chiamati *standard signed integer types*
    - ulteriori tipi che un'implementazione può decidere di fornire, chiamati *extended signed integer types*
  - una serie di tipi per rappresentare intervalli di interi solo positivi, chiamati collettivamente *unsigned integer types* (tipi *unsigned*), tra cui
    - 6 tipi obbligatoriamente presenti in ogni implementazione, chiamati *standard unsigned integer types*
    - ulteriori tipi che un'implementazione può decidere di fornire, chiamati *extended unsigned integer types*

Tipi Interi in C Standard

- Il motivo per cui C Standard ha una tale abbondanza di tipi interi è che vuole dare alle implementazioni la possibilità di offrire, per ciascuno dei formati di dato **f** della *ISA* che esegue l'implementazione
  - un tipo di dato intero tale che ciascuno dei suoi valori è rappresentato da una parola di formato **f**, usando come interpretazione di dato una codifica *unsigned*
  - un tipo di dato intero tale che ciascuno dei suoi valori è rappresentato da una parola di formato **f**, usando come interpretazione di dato una codifica *signed*
- Un'implementazione che sfrutta questa possibilità, quindi, permette ai programmatori di utilizzare la memoria con la stessa granularità e flessibilità permesse dal *LM* della *ISA* che esegue l'implementazione

Tipi Interi in C Standard

- Come abbiamo detto, C Standard, diversamente da Java, non definisce in modo esatto tutte le caratteristiche dei tipi di dato
- In particolare, per ciascun tipo intero non vengono specificati esattamente né l'insieme dei valori rappresentabili né la rappresentazione
- C Standard stabilisce alcuni vincoli che devono essere rispettati dalle implementazioni

Tipi Interi in C Standard

- Ciascun tipo intero rappresenta un intervallo di interi
- Per tutti i tipi interi, i valori non-negativi devono essere rappresentati mediante stringhe binarie interpretate in codifica naturale
- Per tutti i tipi *signed*, i valori negativi devono essere rappresentati mediante stringhe binarie interpretate con una stessa codifica, scelta tra tre possibili che descriveremo nel seguito

Corrispondenza tra Tipi Signed e Tipi Unsigned

- Per ciascun tipo *signed S*, deve esistere un *corrispondente* tipo *unsigned U* tale che
  - i valori dei due tipi devono essere memorizzati in oggetti che hanno la stessa quantità di bit e gli stessi requisiti di allineamento
  - il sotto-intervallo dei valori non-negativi di S è contenuto nell'intervallo dei valori di U
  - ogni valore non-negativo di S, deve avere uguale rappresentazione in S e in U
- Grazie a questi vincoli, la conversione di un valore da un tipo *signed S* al corrispondente tipo *unsigned U*, o viceversa, non richiede di modificare la rappresentazione del valore

Rappresentazioni dei Tipi Interi

- Le rappresentazioni dei valori appartenenti ai tipi *unsigned* sono stringhe binarie formate da
  - una o più cifre di *valore*: ciascuna di esse rappresenta una diversa potenza di 2
  - zero o più cifre di *padding* (o di *riempimento*): ciascuna di esse non contribuisce al calcolo del valore rappresentato, ed è non specificato se valga 0 oppure 1
- È possibile (e molto frequente) che non vi siano cifre di *padding*
- Dunque se la rappresentazione di un tipo *unsigned* specifica che le cifre di valore sono N, gli oggetti di tale tipo rappresentano valori da 0 a $2^N - 1$ mediante codifica naturale

Rappresentazioni dei Tipi Interi

- Le rappresentazioni dei valori appartenenti ai tipi *signed* sono stringhe binarie formate da
  - una cifra di *segno*: se vale 0 il valore rappresentato è positivo, altrimenti negativo
  - una o più cifre di *valore*: ciascuna di esse deve rappresentare la stessa potenza di 2 della cifra che ha la stessa posizione nelle rappresentazioni dal corrispondente tipo *unsigned*
  - zero o più cifre di *padding* (o di *riempimento*): ciascuna di esse non contribuisce al calcolo del valore rappresentato, ed è non specificato se valga 0 oppure 1
- Se la rappresentazione di un tipo *unsigned* prevede N_U cifre di valore e la rappresentazione del corrispondente tipo *signed* prevede N_S cifre di valore, deve essere $N_U \geq N_S$

Codifica Numeri Interi con Segno

- Una caratteristica fondamentale delle codifiche dei numeri interi, è il modo in cui vengono rappresentati i valori negativi
- C Standard, come già detto, stabilisce che la scelta della tecnica usata per rappresentare valori interi negativi è un implementation-defined behavior
- In particolare, ciascuna implementazione può scegliere, tra tre possibili codifiche, quella più adatta alle proprie caratteristiche ed ha l'obbligo di documentare la scelta fatta
- Tipicamente, la scelta viene fatta in base alla *ISA* per cui l'implementazione produce il codice eseguibile
- Le tecniche di codifica ammesse da C Standard sono
  - Modulo e segno
  - Complemento a 1
  - Complemento a 2

Codifica Numeri Interi con Segno

- La codifica modulo e segno è nota per la sua semplicità concettuale, ma non è mai stata molto utilizzata
- La codifica in complemento a 1 è invece stata utilizzata in diverse architetture di supercomputer Cray, perché consentiva un'ottimizzazione estrema delle unità aritmetico-logiche della CPU
- Praticamente tutte le architetture attualmente in uso utilizzano la codifica in complemento a 2
- Per motivi di backward compatibility, tutte le versioni di C Standard, fino a C18 compresa, continuano ad ammettere tutte e 3 le codifiche anzidette

Signed Standard Integer Types

- Gli *standard signed integer types*, sotto elencati, devono essere obbligatoriamente definiti da tutte le implementazioni C

1	`signed char`
2	`signed short int`
3	`signed int`
4	`signed long int`
5	`signed long long int`

- In tutti i nomi, tranne che in `signed char`, le keyword `signed` e `int` sono opzionali
- Tuttavia, per usare il nome `signed int`, almeno una delle due deve essere presente (in C Tradizionale si potevano in certi casi omettere entrambe, per alcune regole di *dichiarazione implicita* che sono state rimosse in C Standard)
- Il tipo `signed long long int` non è presente in C89

Signed Standard Integer Types

- Fermo restando che molte caratteristiche degli *standard signed integer types* sono implementation-defined, C Standard stabilisce alcuni vincoli che devono essere rispettati da tutte le implementazioni
- Come detto, si deve usare utilizzare la stessa tecnica di codifica dei valori negativi per tutti i tipi *signed*
- Ogni implementazione deve definire (nell'header `limits.h`) delle costanti con nomi stabiliti da C Standard e valori pari agli estremi dei vari intervalli di valori degli *standard signed integer types*
- Ad esempio, devono essere definite le costanti `INT_MIN` e `INT_MAX` con valori pari, rispettivamente, al più piccolo e al più grande intero rappresentabile con il tipo `int`

Signed Standard Integer Types

- Per ciascuno degli *standard signed integer types*, C Standard fissa un limite minimo ai valori assoluti del più piccolo e del più grande intero rappresentabile
- Tali limiti sui valori si riflettono in un limite minimo alla quantità di cifre di valore delle rappresentazioni

Signed Standard Integer Types

- Ad esempio, per i valori minimo e massimo rappresentabili da `int`, devono valere le relazioni `INT_MIN` $\leq -(2^{15} - 1)$ e $(2^{15} - 1) \leq$ `INT_MAX`
- Una implementazione che scelga di usare la codifica modulo e segno per i valori negativi, potrebbe definire `INT_MIN` pari a $-(2^{15} - 1)$ e `INT_MAX` pari a $(2^{15} - 1)$ e rappresentare i valori di `int` con 15 cifre di valore
- Un'altra implementazione che usi la codifica in complemento a 2 per i valori negativi, potrebbe definire `INT_MIN` pari a -2^{31} e `INT_MAX` pari a $(2^{31} - 1)$ e rappresentare i valori di `int` con 31 cifre di valore

Signed Standard Integer Types

- Per ciascun tipo T appartenente agli *standard signed integer types*, l'intervallo dei valori di T deve essere sotto-insieme dell'intervallo dei valori degli *standard signed integer types* che seguono T nell'elenco
- Ovvero se T_2 segue T_1 nell'elenco degli *standard signed integer types*, l'intervallo dei valori di T_1 deve essere contenuto (al limite coincidendo) con l'intervallo dei valori di T_2
- Come conseguenza
  - il più piccolo valore rappresentabile in T_2 deve essere minore o uguale del più piccolo valore rappresentabile in T_1
  - il più grande valore rappresentabile in T_2 deve essere maggiore o uguale del più grande valore rappresentabile in T_1

Signed Standard Integer Types

- Ad esempio, si consideri `long`, i cui valori minimo e massimo rappresentabili sono indicati, rispettivamente, dalle costanti `LONG_MIN` e `LONG_MAX`
- Poiché `long` segue `int` nell'elenco degli *standard signed integer types*, devono risultare vere le relazioni $\texttt{LONG\_MIN} \leq \texttt{INT\_MIN}$ e $\texttt{INT\_MAX} \leq \texttt{LONG\_MAX}$
- Un'implementazione potrebbe definire `LONG_MIN` come -2^{31}, `INT_MIN` come -2^{15}, `INT_MAX` come $(2^{15} - 1)$, `LONG_MAX` come $(2^{31} - 1)$
- Un'altra implementazione potrebbe definire `LONG_MIN` e `INT_MIN` come -2^{31}, `INT_MAX` e `LONG_MAX` come $(2^{31} - 1)$; in questo caso `int` e `long` rappresenterebbero lo stesso intervallo di valori, pur restando due tipi distinti

Unsigned Standard Integer Types

- Gli *standard unsigned integer types*, obbligatoriamente definiti da tutte le implementazioni C, sono

0	`_Bool`
1	`unsigned char`
2	`unsigned short int`
3	`unsigned int`
4	`unsigned long int`
5	`unsigned long long int`

- In tutti i nomi in cui compare, la keyword `int` è opzionale
- I tipi `_Bool` e `unsigned long long int` non sono presenti in C89
- `_Bool` è l'unico tipo di cui C Standard definisce in modo esatto i valori rappresentabili, che sono solo 0 e 1

Unsigned Standard Integer Types

- Per $1 \leq i \leq 5$, il tipo che occupa la posizione i nell'elenco degli *standard unsigned integer types* è il tipo che corrisponde a quello che occupa la posizione i nell'elenco degli *standard signed integer types*
- Ciò è suggerito anche dai nomi, infatti si noti che il nome del tipo che occupa la posizione $i > 0$ nell'elenco degli *standard unsigned integer types* differisce da quello del tipo che occupa la stessa posizione nell'elenco degli *standard signed integer types* solo per il fatto che la keyword `unsigned` prende il posto della keyword `signed`

Unsigned Standard Integer Types

- Anche per gli *standard unsigned integer types*, molte caratteristiche sono implementation-defined, ma C Standard pone alcuni vincoli alle implementazioni
- Per tutti gli *standard unsigned integer types*, il più piccolo valore rappresentabile è `0` e, tranne che per `_Bool`, il più grande valore rappresentabile è sì implementation-defined, ma deve essere almeno pari a un valore minimo
- Ogni implementazione deve definire (nell'header `limits.h`) delle costanti con nomi stabiliti da C Standard e valori pari ai più grandi valori rappresentabili di ciascuno degli *standard unsigned integer types* diversi da `_Bool`
- Ad esempio, devono essere definite le costanti `UINT_MAX` e `ULONG_MAX` con valori pari ai più grandi interi rappresentabili, rispettivamente, con `unsigned int` e con `unsigned long`

Unsigned Standard Integer Types

- Per ciascun tipo T appartenente agli *standard unsigned integer types*, l'intervallo dei valori di T deve essere sotto-insieme dell'intervallo dei valori degli *standard unsigned integer types* che seguono T nell'elenco
- Ovvero se T_2 segue T_1 nell'elenco degli *standard unsigned integer types*, il massimo valore rappresentabile in T_1 deve essere non maggiore del massimo valore rappresentabile in T_2
- Ad esempio, `UINT_MAX` deve essere minore o uguale di `ULONG_MAX`

Unsigned Standard Integer Types

- Siano S_s uno degli *standard signed integer types* e U_s il corrispondente tipo nell'elenco degli *standard unsigned integer types*
- C Standard impone che ciascun valore non-negativo v di S_s sia contenuto anche nell'intervallo dei valori di U_s e che le rappresentazioni di v in S_s e in U_s siano identiche

Unsigned Standard Integer Types

- Inoltre i valori di S_s e di U_s devono essere rappresentati da stringhe binarie con la stessa lunghezza e memorizzati in oggetti che hanno la stessa quantità di bit e gli stessi requisiti di allineamento
- Indicando con N_U la quantità di cifre di valore della rappresentazione di U_s e con N_S la quantità di cifre di valore della rappresentazione di S_s, deve essere $N_U \geq N_S$
- Di conseguenza il numero di cifre di *padding* di U_s deve essere minore o uguale del numero di cifre di *padding* di S_s

Unsigned Standard Integer Types

- L'insieme dei vincoli precedenti permette il realizzarsi di diverse situazioni anche se le rappresentazioni dei valori di S_s e U_s hanno la stessa quantità di cifre
- Alcune possibilità sono
  - l'insieme dei valori non-negativi di S_s coincide con l'intervallo dei valori di U_s
  - i valori di U_s sono il doppio dei valori non-negativi di S_s
  - i valori di U_s sono il quadruplo dei valori non-negativi di S_s
- Nel caso più frequente, le rappresentazioni non hanno cifre di *padding* e la cifra che nelle rappresentazioni di U_s ha la stessa posizione della cifra di segno nelle rappresentazioni di S_s, è usata per esprimere una potenza di 2: quindi i valori di U_s sono esattamente il doppio dei valori non-negativi di S_s

Unsigned Standard Integer Types

- Ad esempio, in un'implementazione "esotica", potrebbe accadere che
  - sia `LONG_MAX` sia `ULONG_MAX` valgono $2^{31} - 1$
  - i valori di `long` sono rappresentati da stringhe di 32 cifre, di cui la cifra in posizione 31 è di segno e per $0 \leq i < 31$, la cifra in posizione i è di valore e rappresenta 2^i
  - i valori di `unsigned long` sono rappresentati da stringhe di 32 cifre, di cui la cifra in posizione 31 è di *padding* e per $0 \leq i < 31$, la cifra in posizione i è di valore e rappresenta 2^i

Unsigned Standard Integer Types

- Invece, in una tipica implementazione, potrebbe accadere che
  - `LONG_MAX` vale $2^{31} - 1$ e `ULONG_MAX` vale $2^{32} - 1$
  - i valori di `long` sono rappresentati da stringhe di 32 cifre, di cui la cifra in posizione 31 è di segno e per $0 \leq i < 31$, la cifra in posizione i è di valore e rappresenta 2^i
  - i valori di `unsigned long` sono rappresentati da stringhe di 32 cifre, e per $0 \leq i \leq 31$, la cifra in posizione i è di valore e rappresenta 2^i

Il Tipo `unsigned char`

- Il tipo `unsigned char` ha delle caratteristiche e un ruolo speciali
- Gli oggetti che memorizzano i valori di tipo `unsigned char` sono chiamati *byte*
- La quantità di bit che formano un byte è *implementation defined* ed è indicata dalla costante simbolica `CHAR_BIT` definita in `limits.h`
- L'unico vincolo imposto da C Standard su `CHAR_BIT` è che il suo valore deve essere almeno 8
- Nella maggioranza delle implementazioni `CHAR_BIT` è pari a 8, ma esistono casi in cui il valore di `CHAR_BIT` è maggiore di 8

Il Tipo `unsigned char`

- Le rappresentazioni dei valori di `unsigned char` non hanno cifre di *padding*
- Quindi la costante `UCHAR_MAX`, pari al più grande valore rappresentabile in `unsigned char`, vale esattamente $2^{\texttt{CHAR\_BIT}} - 1$

Il Tipo `unsigned char`

- Per qualunque tipo T (non necessariamente un tipo base), gli oggetti di tipo T sono formati da un insieme di byte
- Ovvero le rappresentazioni dei valori di T sono formate da una quantità di bit che è un multiplo di `CHAR_BIT`
- Quindi i byte, gli oggetti di tipo `unsigned char`, sono i "mattoni costitutivi" degli oggetti di un qualunque altro tipo
- Inoltre ciascun valore di un tipo T può essere copiato in un insieme di byte, con tecniche che verranno presentate nel prosieguo di LPS
- Ciò consente di leggere e manipolare valori di tipo T attraverso operazioni su valori di tipo `unsigned char`

Il Tipo `unsigned char`

- Poiché gli oggetti di ogni tipo T sono formati da un insieme di byte, il byte viene usato come unità di misura della quantità di memoria utilizzata dalla rappresentazione dei tipi
- C Standard definisce l'operatore `sizeof` che misura in termini di byte la quantità di memoria occupata dal valore di un'espressione o dalle rappresentazioni di un tipo indicato mediante il nome
  - se T è il nome di un tipo, `sizeof(`T`)` è la quantità di byte che formano un oggetto di tipo T
  - se E è un'espressione, `sizeof` E è la quantità di byte che formano un oggetto che il tipo di E
- C Standard stabilisce che il risultato di `sizeof` è un valore che appartiene ad uno dei tipi *unsigned* e che è *implementation defined* quale sia esattamente tale tipo

Il Tipo `unsigned char`

- Per definizione `sizeof(unsigned char)` è pari a 1
- Poiché `unsigned char` è il tipo *unsigned* che corrisponde a `signed char`, anche `sizeof(signed char)` è pari a 1
- Per ogni altro tipo *signed* S, `sizeof(`S`)` è un valore *implementation defined*, ed è uguale al valore di `sizeof(`U`)`, dove U è il tipo *unsigned* che corrisponde ad S
- Gli oggetti di un qualunque tipo T sono composti da una quantità di bit pari a `CHAR_BIT * sizeof(`T`)`

Aritmetica sui tipi interi in C

- Le operazioni aritmetiche sui tipi interi presentano una importante differenza, che ne rispecchia una analoga vista in precedenza nei linguaggi *ASM*, a seconda se si applichino a valori di un tipo *signed* o a valori di un tipo *unsigned*
- Nel caso in cui un'operazione aritmetica ordinaria tra valori di un tipo *unsigned* produca un risultato R che non possa essere rappresentato in tale tipo (perché negativo o perché maggiore del massimo intero rappresentabile), l'operatore aritmetico del C produce come risultato il valore di R modulo il valore del più grande intero rappresentabile dal tipo aumentato di 1

Aritmetica sui tipi interi in C

- Ad esempio, in un'implementazione per cui `UINT_MAX` vale 65535, il prodotto tra 1000 e 100 non è rappresentabile nel tipo `unsigned int`
- Il risultato di `1000U * 100U` è comunque definito ed è pari al valore $100000 \mod 65536 = 34464$ (infatti il quoziente della divisione intera di 100000 per 65536 è 1, e quindi il resto è pari a $100000 - 1 \cdot 65536$)
- Questo comportamento degli operatori aritmetici applicati a valori di un tipo *unsigned* non viene considerato un'anomalia: è esattamente questa la semantica che C Standard definisce per tali operatori

Aritmetica sui tipi interi in C

- Nel caso di operazioni aritmetiche tra valori di un tipo *signed* si vorrebbe invece che gli operatori del C producessero lo stesso risultato delle operazioni aritmetiche ordinarie
- Se un'operazione aritmetica tra valori di un tipo *signed* produce un risultato R che non può essere rappresentato in tale tipo (ovvero in caso di overflow), si ha un undefined behavior
- C Standard sceglie di rendere indefinito il comportamento, per permettere alle implementazioni C di gestire le situazioni di overflow nel modo più appropriato, in relazione alla ISA per cui l'implementazione produce il codice eseguibile
- Infatti vi sono differenze significative tra diverse architetture nel modo di gestire gli overflow, come si è visto nei casi di MIPS32 e MC68000

Altre Caratteristiche degli Interi in C Standard

- Il capitolo 7 di **[Ki]** discute altre caratteristiche dei tipi interi in C Standard
  - Dettagli sui nomi dei tipi e il loro uso nelle dichiarazioni
  - Rappresentazione nel codice sorgente di costanti intere
  - Input e output di valori interi mediante le funzioni `printf` e `scanf`
- I capitoli 7 e 23 di **[Ki]** elencano le costanti, definite nell'header `limits.h`, che descrivono i valori rappresentabili minimi e massimi degli *standard signed integer types* e degli *standard unsigned integer types* e mostrano quali sono i limiti stabiliti da C Standard nonché alcune scelte comuni a molte implementazioni, per i valori di tali costanti

Numeri Reali e Complessi in C Standard

- L'aritmetica *floating point* è un metodo che permette di rappresentare, in modo approssimato, numeri reali attraverso stringhe binarie
- Un *numero floating point* è un stringa binaria di lunghezza finita che rappresenta, in modo approssimato, un numero reale
- C Standard definisce
  - 3 tipi di dato, chiamati *real floating*, per rappresentare numeri reali attraverso numeri floating point
  - 3 tipi di dato, chiamati *complex*, per rappresentare numeri complessi attraverso coppie di numeri floating point
- I tipi complex sono stati introdotti in C99 come caratteristica obbligatoria, ma sono divenuti opzionali in C11 e in C17
- In LPS non vengono trattati i tipi complex; si rimandano gli interessati al capitolo 27 di **[Ki]**

Numeri Reali e Complessi in C Standard

- I tipi real floating sono `float`, `double`, `long double`
- Come per i tipi interi, C Standard non definisce esattamente le caratteristiche dei tipi real floating, rendendole quindi implementation-defined, ma impone alcune regole
- I capitoli 7 e 23 di **[Ki]** discutono
  - Dettagli sui nomi dei tipi e il loro uso nelle dichiarazioni
  - Informazioni su regole e scelte tipiche relative alle caratteristiche dei valori per ciascuno dei tipi floating point
  - Rappresentazione nel codice sorgente di costanti floating point
  - Input e output di valori floating point mediante le funzioni `printf` e `scanf`

Caratteri in C Standard

- Così come per i valori Booleani, C Standard rappresenta i caratteri attraverso tipi per interi
- In ciò, il linguaggio C si adegua ad una prassi che precede la nascita dell'informatica, ovvero rappresentare caratteri attraverso una *codifica* in numeri interi (ovvero una funzione iniettiva che ha come dominio l'alfabeto da rappresentare e codominio l'insieme degli interi)

Caratteri in C Standard

- C Standard prevede l'esistenza di
  - un insieme "basilare" di caratteri che vengono rappresentati ciascuno da una stringa binaria che può essere contenuta in un byte, e quindi di lunghezza pari a `CHAR_BIT` (*basic character set*)
  - ulteriori caratteri, detti *extended characters*, che possono essere rappresentati da stringhe binarie di lunghezza maggiore di `CHAR_BIT`
- L'insieme che contiene il *basic character set* e tutti gli *extended characters* è chiamato *extended character set*

Caratteri in C Standard

- Gli *extended characters* sono opzionali e sono ammessi da C Standard a partire da C95
- Possono essere rappresentati in due modi
  - sequenze di byte di lunghezza variabile
  - valori di un tipo intero la cui rappresentazione prevede un numero sufficiente di cifre binarie, definito dall'implementazione
- In LPS non viene trattata la gestione degli *extended characters*; si rimandano gli interessati al capitolo 25 di **[Ki]**

Caratteri in C Standard

- Per rappresentare il *basic character set*, C Standard fornisce (per motivi storici) un tipo di dato "dedicato" che ha nome `char` (non preceduto né da `signed` né da `unsigned`)
- In realtà `char` è comunque un tipo che memorizza numeri interi ed è per definizione un tipo diverso sia da `signed char` che da `unsigned char`
- Come per `signed char` e `unsigned char`, gli oggetti di tipo `char` sono formati da `CHAR_BIT` bit e `sizeof(char)` vale 1
- Nella maggior parte delle implementazioni, `CHAR_BIT` vale 8 e quindi il *basic character set* può avere fino a 256 caratteri
- Esistono però implementazioni in cui `CHAR_BIT` è maggiore di 8 e che quindi possono rappresentare una maggior quantità di caratteri nel *basic character set*

Caratteri in C Standard

- In base alla stessa politica adottata per gli altri tipi, C Standard non definisce esattamente tutte le caratteristiche di `char`
- In particolare sono *implementation-defined*
  - se `char` è un tipo *signed* o *unsigned*
  - l'alfabeto rappresentato da `char`
  - la codifica usata per i caratteri

Caratteri in C Standard

- L'alfabeto e la codifica dei caratteri devono rispettare alcuni vincoli
  - il byte di valore 0 rappresenta un carattere speciale detto *null character* (usato, ad esempio, come ultimo carattere delle stringhe)
  - l'alfabeto deve contenere cifre, lettere maiuscole e minuscole, alcuni simboli, un carattere spazio e alcuni caratteri di *controllo* delle operazioni di input/output
  - le cifre devono essere rappresentate, nell'ordine da 0 a 9, da interi consecutivi (ciò può essere sfruttato in modo utile grazie alla possibilità di fare operazioni aritmetiche con valori di tipo `char`)
  - i caratteri del *basic character set* devono essere rappresentati da valori positivi (quindi se `char` è un tipo *signed* nel *basic character set* ci possono essere al massimo $2^{\texttt{CHAR\_BIT}-1}$ caratteri)

Caratteri in C Standard

- Quasi tutte le codifiche esistenti sono variazioni e/o estensioni di una delle due seguenti
  - ASCII
  - EBCDIC
- ASCII e EBCDIB sono incompatibili tra di loro
- In ASCII e in tutte le codifiche da essa derivate, sia le lettere maiuscole sia le minuscole sono rappresentate, in ordine alfabetico, da interi consecutivi: ciò può essere sfruttato in modo utile grazie alla possibilità di fare operazioni aritmetiche con valori di tipo `char`
- Però in EBCDIC e in tutte le codifiche da essa derivate, questa proprietà non vale

Caratteri in C Standard

- Il capitolo 7 di **[Ki]** discute
  - Informazioni su regole e scelte tipiche relative alle caratteristiche dei valori di tipo `char`
  - Rappresentazione nel codice sorgente di costanti carattere
  - Esempi di operazioni aritmetiche su valori di tipo `char`
  - Input e output di caratteri mediante le funzioni `printf`, `scanf`, `getchar` e `putchar`

Usare i Tipi in Modo Portabile

- Lasciare alle implementazioni la libertà di definire aspetti e caratteristiche importanti dei tipi, come fa C Standard, facilita la realizzazione di implementazioni versatili ed efficienti ma pone delle difficoltà nella realizzazione di programmi portabili
- Per superare tali difficoltà, C Standard offre 3 strumenti di supporto principali
  - I "requisiti minimi" sulle caratteristiche dei tipi, che tutte le implementazioni devono soddisfare
  - Macro che descrivono le caratteristiche implementation-defined dei tipi
  - *Alias semantici* per i nomi di tipo

Usare i Tipi in Modo Portabile

- Come detto in precedenza, su alcune caratteristiche dei tipi che sono implementation-defined, C Standard impone comunque dei vincoli
- Ad esempio le implementazioni possono scegliere `INT_MIN` e `INT_MAX` (il più piccolo e il più grande valore rappresentabili in `int`) rispettando però le relazioni $\texttt{INT\_MIN} \leq -(2^{15} - 1)$ e $(2^{15} - 1) \leq \texttt{INT\_MAX}$
- Conoscendo tali vincoli, il programmatore può fare delle scelte sui tipi da utilizzare che rendano un programma corretto, qualunque siano le scelte dell'implementazione
- Tuttavia queste scelte, pur garantendo sempre la correttezza del programma, potrebbero non essere le migliori possibili dal punto di vista dell'efficienza (velocità di esecuzione e occupazione di memoria)

Usare i Tipi in Modo Portabile

- Come detto in precedenza, alcuni header di C Standard Library definiscono delle macro, tipicamente delle costanti, che descrivono alcune scelte fatte dall'implementazione relativamente alle caratteristiche dei tipi
- Ad esempio, il minimo e il massimo valore rappresentabili da ciascuno dei tipi interi *signed*
- Quindi è possibile scrivere un programma che accede a tali informazioni e che, con tecniche di meta-programmazione basate sulla *compilazione condizionata*, fa in modo che in fase di traduzione vengano scelti automaticamente i tipi più adatti da usare nelle dichiarazioni
- La compilazione condizionata e le tecniche che essa rende possibili non fanno parte dei contenuti di LPS; si rimandano gli interessati al Capitolo 14 di **[Ki]**

Usare i Tipi in Modo Portabile

- Alcuni header di C Standard Library definiscono degli *alias* per i nomi di tipo (ovvero dei nomi alternativi) che specificano delle proprietà di un tipo di interesse per il programmatore
- Questi alias possono riferirsi, in implementazioni diverse, a tipi diversi che però hanno le caratteristiche indicate dall'alias
- Alcuni esempi
  - `size_t`
  - tipi interi con numero di bit esatto
  - tipi interi con numero di bit minimo
  - i più veloci tipi interi con numero minimo di bit

Definire Alias per Nomi di Tipo

- La possibilità di definire dei nuovi nomi per i tipi di dato, aiuta a realizzare programmi facili da modificare e portabili
- A tale scopo, si può utilizzare la definizione di macro attraverso `#define`
- Tuttavia C Standard fornisce il costrutto `typedef` che è ancora più efficace
- Il capitolo 7 di **[Ki]** presenta `typedef` e ne discute l'utilità e i vantaggi rispetto all'uso di definizioni di macro