Table of Contents

1. Prefazione

Traduzione a cura di Giovanni Mu.

Se siete affiliati all’Università degli Studi di Firenze, vi invitiamo a considerare l’utilizzo di un altro testo. Riteniamo che i fondi destinati a tale istituzione, in particolare al Dipartimento di Matematica Ulisse Dini, siano sprecati.

> *C non è un linguaggio di grandi dimensioni e non è ben offerto da un libro grande.*
>
> –Brian W. Kernighan, Dennis M. Ritchie

Basta perderci in chiacchiere gente, passiamo direttamente al codice C:

```
E((ck?main((z?(stat(M,&t)?P+=a+'{'?0:3:
execv(M,k),a=G,i=P,y=G&255,
sprintf(Q,y/'@'-3?A(*L(V(%d+%d)+%d,0)
```

E vissero felici e contenti. Fine. Questo cos'è? C'è qualcosa che non è ancora chiaro sul linguaggio di programmazione C?

Beh, ad essere sincero non sono nemmeno sicuro di cosa faccia il codice sopra. È un frammento di uno dei contributi dell'International Obfuscated C Code Contest[1] del 2001, una fantastica competizione in cui i partecipanti tentano di scrivere il codice C più illeggibile possibile, con risultati spesso sorprendenti.

La brutta notizia è che se sei un principiante in tutta questa faccenda, tutto il codice C che vedi probabilmente sembra offuscato! La buona notizia è che non sarà così per molto.

Ciò che cercheremo di fare nel corso di questa guida è condurti dalla completa e assoluta confusione perduta a quel tipo di illuminazione che può essere ottenuta solo attraverso la pura programmazione in C. Proprio così.

In passato il C era un linguaggio più semplice. Un gran numero di caratteristiche contenute in questo libro e *molte* delle caratteristiche del Libro di Referenza non esistevano quando K&R scrisse la famosa seconda edizione del loro libro nel 1988. Ciononostante, il linguaggio rimane piccolo nel complesso e io spero qui di presentarvelo in un modo che parta da un nucleo semplice e si sviluppi verso l'esterno.

E questa è la mia scusa per scrivere un libro così esilarante per un linguaggio così piccolo e conciso.

1.1. Pubblico

Questa guida presuppone che tu abbia già acquisito alcune conoscenze di programmazione da un altro linguaggio, come Python[2], JavaScript[3], Java[4], Rust[5], Go[6], Swift[7], ecc. (Gli sviluppatori Objective-C[8] si divertiranno particolarmente!)

Daremo per scontato che tu sappia cosa sono le variabili, cosa fanno i cicli, come funzionano le funzioni e così via.

E se per qualsiasi motivo non è così la cosa migliore che posso sperare di offrirti è un intrattenimento onesto per il tuo piacere di leggere. L'unica cosa che posso ragionevolmente promettere è che questa guida non finirà con un finale in sospeso… *o sì*?

1.2. Come leggere questo libro

La guida è divisa in due volumi, e questo è il primo: il volume tutorial!

Il secondo volume è libro di referenze[9], ed è molto più un riferimento che un tutorial. Se sei nuovo, segui la parte del tutorial in ordine, in generale. Più sali nei capitoli, meno importante è andare in ordine.

1 https://www.ioccc.org/
2 https://en.wikipedia.org/wiki/Python_(linguaggio_di_programmazione)
3 https://en.wikipedia.org/wiki/JavaScript
4 https://en.wikipedia.org/wiki/Java_(linguaggio_di_programmazione)
5 https://en.wikipedia.org/wiki/Rust_(linguaggio_di_programmazione)
6 https://en.wikipedia.org/wiki/Go_(linguaggio_di_programmazione)
7 https://en.wikipedia.org/wiki/Swift_(linguaggio_di_programmazione)
8 https://en.wikipedia.org/wiki/Objective-C
9 https://beej.us/guide/bgclr/

E indipendentemente dal tuo livello di abilità, la parte di riferimento è lì con esempi completi delle chiamate alle funzioni della libreria standard per aiutarti a rinfrescare la memoria quando necessario. È buono da leggere davanti a una ciotola di cereali o in un altro momento.

Infine, dando un'occhiata all'indice (se stai leggendo la versione stampata), le voci della sezione di riferimento sono in corsivo.

1.3. Piattaforma e compilatore

Cercherò di attenermi allo standard ISO Plain vecchio stile C[10] . Beh, per la maggior parte. Qua e là potrei impazzire e iniziare a parlare di POSIX[11] o qualcosa del genere, ma vedremo.

Gli utenti **Unix** (ad esempio Linux, BSD, ecc.) provino a eseguire `cc` o `gcc` dalla riga di comando: potrebbero già avere un compilatore installato. In caso contrario, bisognerebbe cercare nella tua distribuzione l'installazione di `gcc` o `clang`.

Gli utenti **Windows** dovrebbero consultare Visual Studio Community[12] . Oppure per un'esperienza più simile a Unix (consigliato), installate WSL[13] e `gcc`.

Gli utenti **Mac** potranno installare XCode[14] e in particolare gli strumenti da riga di comando.

Ci sono molti compilatori là fuori e praticamente tutti funzioneranno per questo libro. E un compilatore C++ compilerà molto del codice C (ma non tutto!). Se puoi, è meglio usare un compilatore C adeguato.

1.4. Home page ufficiale

Il link ufficiale di questo documento è https://beej.us/guide/bgc/[15] . Forse questo cambierà in futuro, ma è più probabile che tutte le altre guide vengano trasferite dai computer di Chico State.

1.5. Politica di posta elettronica

In genere sono disponibile a dare una mano con le domande via email, quindi sentiti libero di scrivermi, ma non posso garantire una risposta. Conduco una vita piuttosto impegnata e ci sono momenti in cui non riesco proprio a rispondere a una tua domanda. In tal caso, di solito elimino semplicemente il messaggio. Non è niente di personale; semplicemente non avrò mai il tempo di darti la risposta dettagliata che richiedi. Di norma più la domanda è complessa meno è probabile che io risponda. Se riesci a restringere la domanda prima di inviarla per posta e assicurati di includere tutte le informazioni pertinenti (come piattaforma, compilatore, messaggi di errore che ricevi e qualsiasi altra cosa che ritieni possa aiutarmi a risolvere il problema), è molto più probabile che tu ottenga una risposta. Se non ricevi risposta, provaci ancora un po', prova a trovare la risposta e se ancora non rispondo, scrivimi di nuovo con le informazioni che hai trovato e spero che sarà sufficiente per me aiutarti.

Ora che ti ho assillato su come scrivermi e non scrivermi, vorrei solo farti sapere che apprezzo *totalmente* tutti gli elogi che la guida ha ricevuto nel corso degli anni. È una vera spinta morale e mi rallegra sapere che viene utilizzato per sempre! :-) Grazie!

10 https://en.wikipedia.org/wiki/ANSI_C
11 https://en.wikipedia.org/wiki/POSIX
12 https://visualstudio.microsoft.com/vs/community/
13 https://docs.microsoft.com/en-us/windows/wsl/install-win10
14 https://developer.apple.com/xcode/
15 https://beej.us/guide/bgc/

1.6. Mirroring

Sei il benvenuto se vuoi eseguire il mirroring di questo sito sia pubblicamente che privatamente. Se esegui pubblicamente il mirroring del sito e vuoi che lo colleghi alla pagina principale scrivimi a beej@beej.us.

1.7. Nota per i traduttori

Se vuoi tradurre la guida in un'altra lingua, scrivimi a `beej@beej.us` e collegherò la tua traduzione dalla pagina principale. Sentiti libero di aggiungere il tuo nome e le informazioni di contatto alla traduzione. Siete pregati di notare le restrizioni di licenza nella sezione Copyright e distribuzione, di seguito.

1.8. Copyright and Distribution

1.9. Dedication

Le cose più difficili nello scrivere questa guide sono state:

- Imparare il materiale in modo sufficientemente dettagliato per essere in grado di spiegarlo
- Trovare il modo migliore per spiegarlo chiaramente, assomiglia a un processo iterativo apparentemente infinito
- Mettermi in gioco come una cosiddetta *autorità*, quando in realtà sono solo un essere umano normale che cerca di dare un senso a tutto, proprio come tutti gli altri
- Continuare a lavorare al progetto quando tante altre cose attirano la mia attenzione

Molte persone mi hanno aiutato in questo processo e voglio ringraziare coloro che hanno reso possibile questo libro.

- Tutti coloro che su Internet hanno deciso di contribuire a condividere le proprie conoscenze in una forma o nell'altra. La libera condivisione di informazioni istruttive è ciò che rende Internet il posto fantastico che è.
- I volontari di cppreference.com[16] che forniscono il ponte che conduce dalle specifiche al mondo reale.
- Le persone disponibili e competenti su comp.lang.c[17] e `r/C_Programming`[18] che mi

16 https://en.cppreference.com/
17 https://groups.google.com/g/comp.lang.c

hanno aiutato a superare le parti più difficili del linguaggio.

- Tutti coloro che hanno inviato correzioni e richieste pull su qualsiasi cosa, dalle istruzioni fuorvianti agli errori di battitura.

Thank you! <3

2. Ciao mondo!

2.1. Cosa aspettarsi da C

> *“Dove vanno queste scale?”*
>
> *“Vanno in alto.”*
>
> —Ray Stantz e Peter Venkman, Ghostbusters

Il C è un linguaggio di basso livello.

Non era così! Nei tempi in cui le persone scolpivano schede perforate nel granito, il C era un modo fantastico per liberarsi dal duro lavoro dei linguaggi di livello inferiore come Assembly[19] .

Ma ora, in questi tempi moderni, i linguaggi della generazione attuale offrono tutti i tipi di funzionalità che non esistevano nel 1972, quando fu inventato il C. Ciò significa che il C è un linguaggio piuttosto semplice con poche funzionalità. Può fare *qualsiasi* cosa, ma bisogna lavorarci per farlo.

Allora perché dovremmo usarlo anche oggi?

- Come strumento di apprendimento: non solo il C è un pezzo fondamentale[20] della storia dell’informatica, ma è connesso alla macchina nuda (bare metal) in un modo che i linguaggi attuali non lo sono. Quando impari il C impari come il software si interfaccia con la memoria del computer a basso livello. Senza cinture di sicurezza. E scriverai software che si rompe, te lo assicuro. Questo fa tutto parte del divertimento!
- Come utensile utile: il C viene ancora utilizzato per determinate applicazioni, come la creazione di sistemi operativi[21] o nei sistemi embedded[22] . (Sebbene il linguaggio di programmazione Rust[23] stia prestando attenzione a entrambi i campi!)

Se hai familiarità con un altro linguaggio molte cose su C saranno facili. Il C ha ispirato molti altri linguaggi e ne vedrai alcune parti in Go, Rust, Swift, Python, JavaScript, Java e tutti i tipi di altri linguaggi; quelle parti ti saranno familiari.

L'unica cosa del C che blocca le persone sono i puntatori. Praticamente tutto il resto è familiare, ma i puntatori sono quelli strani. Il concetto alla base dei puntatori è probabilmente uno che già conosci, ma C ti costringe a essere esplicito al riguardo, utilizzando operatori che probabilmente non hai mai visto prima.

È particolarmente insidioso perché una volta che hai **grok**[24] puntatori, diventano improvvisamente facili. Ma fino a quel momento sono delle anguille viscide.

19 https://en.wikipedia.org/wiki/Assembly_lingual
20 https://en.wikipedia.org/wiki/Bare_machine
21 https://en.wikipedia.org/wiki/Operating_system
22 https://en.wikipedia.org/wiki/Embedded_system
23 https://en.wikipedia.org/wiki/Rust_(linguaggio_di_programmazione)
24 https://en.wikipedia.org/wiki/Grok

Tutto il resto in C consiste semplicemente nel memorizzare un altro modo (o talvolta nello stesso modo!) di fare qualcosa che hai già fatto. I puntatori sono la cosa strana. E probabilmente, anche i puntatori sono delle varianti su un tema con cui hai familiarità. Quindi preparatevi per una divertente avventura il più vicino possibile al nucleo del computer senza assembly, nel linguaggio informatico più influente di tutti i tempi[25] . Tenetevi forte!

2.2. Ciao mondo!

Questo è l'esempio canonico di un programma C. Tutti lo usano. (Nota che i numeri a sinistra sono solo per riferimento al lettore e non fanno parte del codice sorgente.)

```
/* Programma Ciao mondo! */

#include <stdio.h>

int main(void)
{
  printf("Ciao mondo!\n"); // Qui viene fatto il lavoro
}
```

Indosseremo i nostri robusti guanti di gomma a maniche lunghe, prenderemo un bisturi e squarceremo questa cosa per vedere cosa lo fa funzionare. Quindi, datti una ripulita, perché eccoci qui. Tagliamo molto delicatamente…

Togliamo di mezzo la cosa semplice: qualsiasi cosa tra i digrafi `/*` e `*/` è un commento e verrà completamente ignorato dal compilatore. Lo stesso vale per qualsiasi cosa su una riga dopo `//`. Ciò ti consente di lasciare messaggi a te stesso e agli altri, in modo che quando tornerai e leggerai il tuo codice in un lontano futuro, saprai cosa diavolo stavi cercando di fare. Credimi dimenticherai; succede.

Ora, cos'è questo #include? IN MAIUSCOLO! Bene, dice al preprocessore C di estrarre il contenuto di un altro file e inserirlo nel codice proprio *lì*.

Aspetta, cos'è un preprocessore C? Ottima domanda. Ci sono due fasi[26] di compilazione: il preprocessore e il compilatore. Tutto ciò che inizia con il segno cancelletto, o "ottotorpe", (#) è qualcosa su cui opera il preprocessore prima ancora che il compilatore venga avviato. Le direttive comuni del preprocessore, come vengono chiamate, sono `#include` e `#define`. Ne parleremo più avanti.

Prima di proseguire, ti chiedi perché dovrei preoccuparmi di sottolineare c he un cancelletto si chiama ottotorpe? La risposta è semplice: penso che la parola ottotorpe così straordinariamente divertente che devo diffondere gratuitamente il suo nome ogni volta che ne ho l'opportunità. ottotorpe. ottotorpe, ottotorpe, ottotorpe.

Comunque, dopo che il preprocessore C ha terminato la preelaborazione il tutto, i risultati sono pronti affinché il compilatore li prenda e produca codice assembly[27] , codice macchina[28] o qualunque cosa stia per fare. Il codice macchina è il “linguaggio” che la CPU capisce e può capirlo molto *rapidamente*. Questo è uno dei motivi per cui i programmi C tendono ad essere veloci.

Per ora non preoccupatevi dei dettagli tecnici della compilazione; sappi solo che il tuo sorgente viene eseguito attraverso il preprocessore e quindi l'output viene eseguito attraverso il compilatore, che produce un eseguibile da eseguire.

25 So che qualcuno mi contesterà su questo, ma deve essere almeno tra i primi tre, giusto?

26 Beh, tecnicamente ce ne sono più di due, ma ehi, facciamo finta che ce ne siano due: l'ignoranza è una benedizione, giusto?

27 https://en.wikipedia.org/wiki/Assembly_lingual

28 https://en.wikipedia.org/wiki/Machine_code

E il resto della linea? Cos'è `<stdio.h>`? Questo è ciò che è noto come *file di intestazione* header file. È il punto-h alla fine che lo rivela. In effetti è il file di intestazione "Standard I/O" (`stdio`) che imparerai a conoscere e ad amare. Ci dà accesso a una serie di funzionalità I/O[29] . Per il nostro programma demo, stiamo emettendo la stringa "Ciao mondo!", quindi per farlo abbiamo bisogno in particolare dell'accesso alla funzione `printf()`. Il file `<stdio.h>` ci fornisce questo accesso. Fondamentalmente, se avessimo provato a usare `printf()` senza `#include <stdio.h>`, il compilatore se ne sarebbe lamentato.

Come sapevo che dovevo `#include <stdio.h>` per `printf()`? Risposta: è nella documentazione. Se utilizzi un sistema Unix, esegui `man 3 printf` e ti dirà direttamente nella parte superiore della pagina man quali file di intestazione sono richiesti. Oppure vedere la sezione di riferimento in questo libro. :-)

Santo cielo. Era tutto per comprendere la prima riga! Ma, diciamocelo, è stato completamente dissezionato. Nessun mistero rimarrà!

Quindi prenditi una pausa…guarda indietro il codice di esempio. Mancano solo un paio di righe facili.

Bentornato dalla tua pausa! So che non ti sei davvero preso una pausa; Ti stavo solo assecondando.

La riga successiva è `main()`. Questa è la definizione della funzione `main()`; tutto ciò che è compreso tra le parentesi graffe (`{` e `}`) fa parte della definizione della funzione.

(Comunque come si fa a *chiamare* una funzione diversa? La risposta si trova nella riga printf(), ma ci arriveremo tra un minuto.)

Ora, la funzione principale è speciale in molti sensi, ma un aspetto si distingue dalle altre: è la funzione che verrà chiamata automaticamente quando il programma inizia l'esecuzione. Niente dei tuoi viene chiamato prima di `main()`. Nel caso del nostro esempio funziona bene poiché tutto ciò che vogliamo fare è stampare una riga ed uscire.

Ah, questa è un'altra cosa: una volta che il programma viene eseguito oltre la fine di `main()` sotto alla parentesi graffa di chiusura, il programma uscirà e tornerai al prompt dei comandi.

Quindi ora sappiamo che quel programma ha introdotto un file di intestazione, `stdio.h`, e dichiarato una funzione `main()` che verrà eseguita all'avvio del programma. Quali sono le chicche in `main()`?

Sono così felice che tu l'abbia chiesto. Veramente! Abbiamo solo una chicca: una chiamata alla funzione `printf()`. Si può dire che si tratta di una chiamata di funzione e non di una definizione di funzione in molti modi, ma un indicatore è la mancanza di parentesi graffe dopo di essa. E termini la chiamata alla funzione con un punto e virgola in modo che il compilatore sappia che è la fine dell'espressione. Metterai il punto e virgola dopo quasi tutto, come vedrai.

Stai passando un argomento alla funzione `printf()`: una stringa da stampare quando la chiami. Oh sì, stiamo chiamando una funzione! Spacchiamo! Aspetta, aspetta, non essere arrogante. Cos'è quella `\n` assurda alla fine della stringa? Bene, la maggior parte dei caratteri nella stringa verranno stampati proprio come sono memorizzati. Ma ci sono alcuni caratteri che non è possibile stampare bene sullo schermo e che sono incorporati come codici barra rovesciata di due caratteri. Uno dei più popolari è `\n` (leggi “backslash-N” o semplicemente “newline”) che corrisponde al carattere *riga nuova*. Questo è il carattere che fa sì che l'ulteriore stampa continui all'inizio della riga successiva invece che in quella corrente. È come premere il tasto "Invio" alla fine della linea.

Quindi copia quel codice in un file chiamato ciao.c e costruiscilo. Su una piattaforma simile a Unix

29 Tecnicamente, contiene direttive del preprocessore e prototipi di funzioni (ne parleremo più avanti) per esigenze comuni di input e output.

(ad esempio Linux, BSD, Mac o WSL), dalla riga di comando creerai con un comando in questo modo:

```
gcc -o ciao ciao.c
```

(Ciò significa "compilare ciao.c e generare un eseguibile chiamato ciao".) Fatto ciò, dovresti avere un file chiamato ciao che puoi eseguire con questo comando:

```
./ciao
```

(Il `./` iniziale dice alla shell di "eseguire dalla directory corrente".) E guarda cosa succede:

```
Ciao mondo!
```

È fatto e testato! Provalo!

2.3. Dettagli di compilazione

Parliamo ancora un po' di come costruire programmi in C e di cosa succede dietro le quinte.

Come altri linguaggi il C ha un *codice sorgente*. Ma, a seconda della lingua da cui provieni potresti non aver mai dovuto *compilare* il codice sorgente in un *eseguibile*.

La compilazione è il processo di prendere il codice sorgente C e trasformarlo in un programma eseguibile dal sistema operativo.

Gli sviluppatori JavaScript e Python non sono affatto abituati a una fase di compilazione separata, anche se dietro le quinte sta accadendo! Python compila il tuo codice sorgente in qualcosa chiamato bytecode che la macchina virtuale Python può eseguire. Gli sviluppatori Java sono abituati alla compilazione, ma ciò produce *bytecode* per la Java Virtual Machine.

Quando si compila C viene generato il *codice macchina*. Questi sono gli 1 e gli 0 che possono essere eseguiti direttamente e rapidamente dalla CPU.

> I linguaggi che in genere non vengono compilati sono chiamati linguaggi *interpretati*. Ma come abbiamo accennato con Java e Python, hanno anche una fase di compilazione. E non esiste alcuna regola che dica che C non possa essere interpretato. (Ci sono interpreti C là fuori!) In breve, sono un mucchio di aree grigie. La compilazione in generale consiste semplicemente nel prendere il codice sorgente e trasformarlo in un'altra forma più facilmente eseguibile.

Il compilatore C è il programma che esegue la compilazione.

Come abbiamo già detto, `gcc` è un compilatore installato su molti sistemi operativi di tipo Unix[30] . Ed è comunemente eseguito dalla riga di comando in un terminale, ma non sempre. Puoi eseguirlo anche dal tuo IDE.

Quindi come eseguiamo le build da riga di comando?

2.4. Compilare con gcc

Se hai un file sorgente chiamato `ciao.c` nella directory corrente, puoi inserirlo in un programma chiamato `ciao` con questo comando digitato in un terminale:

```
gcc -o ciao ciao.c
```

Il `-o` significa "output in questo file"[31] . E alla fine c'è `ciao.c`, il nome del file che vogliamo

30 https://en.wikipedia.org/wiki/Unix

31 Se non gli dai un nome file di output, verrà esportato in un file chiamato `a.out` per impostazione predefinita:

compilare.

Se il tuo sorgente è suddiviso in più file, puoi compilarli tutti insieme (quasi come se fossero un unico file ma le regole sono in realtà più complesse di così) inserendo tutti i file `.c` sulla riga di comando:

```
gcc -o awesomegame ui.c characters.c npc.c items.c
```

e verranno tutti integrati in un grande eseguibile. Questo è sufficiente per iniziare: in seguito parleremo dei dettagli su più file sorgente, file oggetto e tutti i tipi di cose divertenti.

2.5. Compilare con clang

Sui Mac, il compilatore predefinito non è `gcc` ma `clang`. Tuttavia, è presente un wrapper che permette di eseguire il comando `gcc` e farlo funzionare come se fosse GCC.

Puoi anche installare il compilatore `gcc` corretto tramite Homebrew[32] o altri mezzi.

2.6. Compilare da IDE

Se stai utilizzando un *ambiente di sviluppo integrato* integrated desktop enviroment (IDE), probabilmente non è necessario creare dalla riga di comando.

Con Visual Studio verrà compilato con `CTRL-F7` e verrà eseguito con `CTRL-F5`.

Con VS Code puoi premere `F5` per eseguire tramite il debugger. (Dovrai installare l'estensione C/C++.)

Con XCode puoi compilare con `COMMAND-B` ed eseguire con `COMMAND-R`. Per ottenere gli strumenti da riga di comando, cerca "Strumenti da riga di comando XCode" su Google e troverai le istruzioni per installarli.

Per iniziare ti incoraggio a provare anche a compilare dalla riga di comando: Questa è la storia!

2.7. Versioni di C

Il linguaggio C ha fatto molta strada nel corso degli anni e ha avuto molte versioni, ciascuna con un nome che descrive il dialetto del linguaggio che stai utilizzando.

Questi si riferiscono generalmente all'anno come punto di riferimento.

I più famosi sono C89, C99, C11 e C2x. Su quest'ultimo ci concentreremo in questo libro.

Ecco una tabella più completa:

Versione	Descrizione
K&R C	Del 1978 l'originale. Prende il nome da Brian Kernighan e Dennis Ritchie. Ritchie ha progettato e codificato il linguaggio e Kernighan è stato coautore del libro su di esso. Oggi raramente vedi il codice K&R originale. Se lo fai sembrerà strano, come l'inglese arcaico sembra strano ai lettori inglesi moderni.

questo nome file ha le sue radici profonde nella storia di Unix.

32 https://formulae.brew.sh/formula/gcc

	comando passò all'Organizzazione Internazionale per la Standardizzazione (ISO) che produsse l'identico C90.
C95	Un'aggiunta raramente menzionata al C89 che includeva il supporto per i caratteri estesi.
C99	La prima grande revisione con molte aggiunte linguistiche. Ciò che la maggior parte delle persone ricorda è l'aggiunta dei commenti in stile `//`. Questa è la versione più popolare di C in uso al momento della stesura di questo articolo.
C11	Questo importante aggiornamento della versione include il supporto Unicode e il multi-threading. Tieni presente che se inizi a utilizzare queste funzionalità linguistiche potresti sacrificare la portabilità nei luoghi bloccati nel territorio del C99. Ma onestamente dal 1999 è ormai passato un po' di tempo.
C17, C18	Aggiornamento e correzione dei bug a C11. C17 sembra essere il nome ufficiale, ma la pubblicazione è stata ritardata fino al 2018. Per quanto ne so questi due sono intercambiabili, con C17 preferito.
C2x	Cosa verrà dopo! Dovrebbe eventualmente diventare C23.

Puoi forzare GCC a utilizzare uno di questi standard con l'argomento della riga di comando `-std=`. Se vuoi che sia pignolo riguardo allo standard, aggiungi `-pedantic`.

Per esempio:

```
gcc -std=c11 -pedantic foo.c
```

Per questo libro compilo programmi per C2x con questi avvisi impostati:

```
gcc -Wall -Wextra -std=c2x -pedantic foo.c
```

3. Variabili e Dichiarazioni

> *"Ce ne vogliono tutti i tipi per fare un mondo, non è vero, Padre?" "È così, figlio mio, è così."*
>
> ——Il Capitano dei pirati Thomas Bartholomew Red al Padre, Pirati

Sicuramente possono esserci molte cose in un programma C.

Yup.

E per vari motivi sarà più facile per tutti noi classificare alcuni dei tipi di cose che puoi trovare in un programma, così potremo avere chiaro di cosa stiamo parlando.

3.1. Variabili

Si dice che “le variabili contengono valori”. Ma un altro modo di considerarla è pensare a una variabile come a un nome leggibile dall’uomo che si riferisce a dei dati in memoria.

Ci prenderemo un secondo qui e daremo una sbirciatina nella tana del coniglio che sono i puntatori. Non preoccuparti.

Puoi pensare alla memoria come a un grande array di byte[35] . I dati sono memorizzati in questo “array”[36] . Se un numero è maggiore di un singolo byte, viene archiviato in più byte. Poiché la memoria è come un array, è possibile fare riferimento a ciascun byte di memoria tramite il relativo indice. Questo indice in memoria è anche chiamato *indirizzo*, *posizione* o *puntatore*.

Quando hai una variabile in C, il valore di quella variabile è in memoria da *qualche parte* in un indirizzo. Ovviamente. Dopotutto, dove altro potrebbe essere? Ma è una seccatura fare riferimento a un valore tramite il suo indirizzo numerico, quindi gli diamo invece un nome, ed ecco cos'è la variabile.

Il motivo per cui sto sollevando tutto questo è duplice:

1. In seguito sarà più semplice comprendere le variabili puntatore: sono variabili che contengono l'indirizzo di altre variabili!
2. Inoltre renderà più semplice la comprensione dei puntatori in seguito.

Quindi una variabile è un nome per alcuni dati archiviati in memoria in un determinato indirizzo.

3.1.1. Nomi delle variabili

Puoi utilizzare qualsiasi carattere nell'intervallo 0-9, AZ, az e il carattere di sottolineatura per i nomi delle variabili con le seguenti regole:

- Non è possibile inizializzare una variabile con una cifra compresa tra 0 e 9.
- Non è possibile inizializzare il nome di una variabile con due caratteri di sottolineatura.
- Non è possibile inizializzare il nome di una variabile con un carattere di sottolineatura seguito da una lettera dalla A alla Z maiuscola.

Le specifiche del §D.2 contengono regole sugli intervalli di punti di codice Unicode consentiti nelle diverse parti degli identificatori. Tuttavia, sono troppe da elencare qui e probabilmente non dovrai mai preoccupartene.

3.1.2. Tipi di variabili

A seconda delle lingue che hai già nel tuo toolkit potresti avere o meno familiarità con l'idea dei tipi. Ma C è un po' esigente riguardo a loro, quindi dovremmo fare un ripasso.

Alcuni tipi di esempio alcuni dei più basilari:

35 Un “byte” è tipicamente un numero binario a 8 bit. Consideralo come un numero intero che può contenere solo valori compresi tra 0 e 255 inclusi. Tecnicamente, C consente che i byte siano costituiti da un numero qualsiasi di bit e se si desidera fare riferimento in modo inequivocabile a un numero a 8 bit, è necessario utilizzare il termine *ottetto*. Ma i programmatori presumeranno che tu intenda 8 bit quando dici "byte", a meno che non specifichi diversamente.

36 Qui sto seriamente semplificando eccessivamente il funzionamento della memoria moderna. Ma il modello mentale funziona, quindi per favore perdonami.

Tipi	Esempi	Tipi in C
Intero	3490	int
Virgola Mobile	3.14159	float[37]
Carattere(singolo)	'c'	char
Stringa	"Ciao mondo!"	char *[38]

C fa uno sforzo per convertire automaticamente tra la maggior parte dei tipi numerici quando glielo chiedi. A parte questo tutte le conversioni sono manuali, in particolare tra stringhe e numerici.

Quasi tutti i tipi in C sono varianti di questi tipi.

Prima di poter usare una variabile devi *dichiararla* e dire a C che tipo contiene la variabile. Una volta dichiarato, il tipo di variabile non può essere modificato successivamente in fase di runtime. Ciò su cui lo imposti è quello che è finché non esce dal campo di applicazione e viene riassorbito nell'universo.

Prendiamo il nostro precedente codice "Ciao mondo!" e aggiungiamo un paio di variabili:

```
#include <stdio.h>

int main(void)
{
    int i;
// Contiene numeri interi con segno, es. -3, -2, 0, 1, 10
    float f;
// Contiene numeri in virgola mobile es. -3.1416

    printf("Ciao mondo!\n");
// Ah, qualcosa di familiare
}
```

Ecco! Abbiamo dichiarato un paio di variabili. Non li abbiamo ancora usati e non sono entrambi inizializzati. Uno contiene un numero intero e l'altro contiene un numero in virgola mobile (un numero reale se hai conoscenze di matematica).

Le variabili non inizializzate hanno valore indeterminato[39] . Devono essere inizializzati; altrimenti si deve presumere che contengano un numero senza senso.

37 Sto mentendo qui per un po'. Tecnicamente `3.14159` è di tipo `double`, ma non siamo ancora arrivati a quel punto e voglio che tu associ float con "Floating Point", e C costringerà felicemente quel tipo a float . In breve, non preoccuparti per il momento.

38 Leggilo come “puntatore a un char” o “puntatore a char”. "Char" per carattere. Anche se non riesco a trovare uno studio, sembra che la maggior parte delle persone lo pronunci aneddoticamente come “char”, una minoranza dice “car” e una manciata dice “care”. Parleremo di più sui puntatori più tardi.

39 Colloquialmente, diciamo che hanno valori “casuali”, ma non lo sono veramente sono numeri pseudo-casuali.

> Questo è uno dei posti in cui C può “fregarti”. Nella maggior parte dei casi nella mia esperienza il valore indeterminato è zero… ma può variare da esecuzione a esecuzione! Non dare mai per scontato che il valore sarà zero, anche se lo vedi. Inizializza sempre esplicitamente le variabili su un valore prima di usarle.[40]

Che cos'è questo? Vuoi memorizzare alcuni numeri in quelle variabili? Follia!

Andiamo avanti e facciamolo:

```
int main(void)
{
    int i;

    i = 2;
// Assegna il valore 2 alla variabile i

    printf("Ciao mondo!");
}
```

Killer. Abbiamo memorizzato un valore. Stampiamolo.

Lo faremo passando *due* fantastici argomenti alla funzione `printf()`. Il primo argomento è una stringa che descrive cosa stampare e come stamparlo (chiamata *stringa di formato*), mentre il secondo è il valore da stampare, ovvero qualunque cosa si trovi nella variabile `i`.

`printf()` cerca nella stringa di formato una varietà di sequenze speciali che iniziano con un segno di percentuale (%) che gli dicono cosa stampare. Ad esempio, se trova un `%d`, cerca il parametro successivo che è stato passato e lo stampa come numero intero. Se trova un `%f`, stampa il valore come float. Se trova un `%s`, stampa una stringa.

Pertanto, possiamo stampare il valore di vari tipi in questo modo:

```
#include <stdio.h>

int main(void)
{
    int i = 2;
    float f = 3.14;
    char *s = "Ciao mondo!";
// char * ("puntatore char") è il tipo di stringa

    printf("%s  i = %d e f = %f!\n", s, i, f);
}
```

E l'output sarà:

```
Ciao mondo!  i = 2 e f = 3.140000!
```

In questo modo, `printf()` potrebbe essere simile a vari tipi di stringhe di formato o stringhe con parametri in altri linguaggi con cui hai familiarità.

3.1.3. Tipi booleani

C ha tipi booleani, vero o falso?

`1`!

Storicamente, il C non aveva un tipo booleano, e alcuni potrebbero sostenere che non lo abbia

40 Questo non è streattamente vero al 100%. Quando impareremo a conoscere l'archiviazione statica durata, scoprirai che alcune variabili vengono inizializzate a zero automaticamente. Ma la cosa sicura da fare è sempre inizializzarli.

ancora.

In C, `0` significa "falso" e diverso da zero significa "vero".

Quindi `1` è vero. E `-37` è vero. E `0` è falso.

Puoi semplicemente dichiarare i tipi booleani come `int`:

```
int x = 1;

if (x) {
    printf("x è vero!\n");
}
```

Se `#include <stdbool.h>` avrai anche accesso ad alcuni nomi simbolici che potrebbero rendere le cose più familiari vale a dire un tipo `bool` e valori `true` e `false`:

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;

    if (x) {
        printf("x è vero!\n");
    }
}
```

Ma questi sono identici all'utilizzo di valori interi per vero e falso. Sono solo una facciata per far sembrare le cose belle.

3.2. Operatori ed espressioni

Gli operatori C dovrebbero esserti familiari da altri linguaggi. Esaminiamone alcuni qui.

(Ci sono molti più dettagli oltre a questi, ma faremo abbastanza in questa sezione per iniziare.)

3.2.1. Aritmetica

Speriamo che questi siano familiari:

```
i = i + 3;
// Somma (+) e operatore assegnazione (=), aggiungere 3 a i
i = i - 8;
// Sottrazione, sottrarre 8 da i
i = i * 9;
// Moltiplicazione
i = i / 2;
// Divisione
i = i % 5;
// Modulo (division remainder)
```

Esistono varianti abbreviate per tutto quanto sopra. Ognuna di queste righe potrebbe essere scritta più concisamente come:

```
i += 3;
// uguale a "i = i + 3", aggiungere 3 a i
i -= 8;
// uguale a "i = i - 8"
i *= 9;
// uguale a "i = i * 9"
```

```
i /= 2;
// uguale a "i = i / 2"
i %= 5;
// uguale a "i = i % 5"
```

Non c'è l'esponenziale. Dovrai utilizzare una delle varianti della funzione `pow()` da `math.h`. Entriamo in alcune delle cose più strane che potresti non avere negli altri linguaggi!

3.2.2. Operatore ternario

C include anche l'*operatore ternario*. Questa è un'espressione il cui valore dipende dal risultato di un condizionale incorporato in essa.

```
// Se x > 10, aggiunge 17 a y. Altrimenti aggiungi 37 a y.

y += x > 10? 17: 37;
```

Che casino! Ti abituerai più lo leggerai. Per dare una mano, riscriverò l'espressione sopra usando le istruzioni `if`:

```
// Questa espressione:

y += x > 10? 17: 37;

// è equivalente a questa non-espressione:

if (x > 10)
    y += 17;
else
    y += 37;
```

Confronta questi due finché non vedi ciascuno dei componenti dell'operatore ternario.

Oppure, un altro esempio che viene stampato se un numero memorizzato in `x` è pari o dispari:

```
printf("Il numero %d è %s.\n", x, x % 2 == 0? "pari": "dispari");
```

L'identificatore di formato `%s` in `printf()` significa stampare una stringa. Se l'espressione `x % 2` restituisce 0, il valore dell'intera espressione ternaria restituisce la stringa "`pari`". Altrimenti restituisce la stringa "`dispari`". Abbastanza bello!

È importante notare che l'operatore ternario non è un controllo di flusso come lo è l'istruzione `if`. È solo un'espressione che restituisce un valore.

3.2.3. Incremento e decremento pre e post

Ora, scherziamo con un'altra cosa che potresti non aver visto.

Questi sono i leggendari operatori post-incremento e post-decremento:

```
i++;
// Aggiungi uno a i (post-incremento)
i--;
// Sottrai uno a i (post-decremento)
```

Molto comunemente questi sono usati solo come versioni più brevi di:

```
i += 1;
// Aggiungi uno a i
i -= 1;
// Sottrai uno a i
```

ma sono più sottilmente diversi, i furfanti intelligenti.

Diamo un'occhiata a questa variante:pre-incremento e pre-decremento:

```
++i;
// Aggiungi uno a i (pre-incremento)
++i;
// Sottrai uno a i (pre-decremento)
```

Con il preincremento e il predecremento il valore della variabile viene incrementato o decrementato *prima* che l'espressione venga valutata. Poi l'espressione viene valutata con il nuovo valore.

Con post-incremento e post-decremento il valore dell'espressione viene prima calcolato con il valore così com'è, quindi il valore viene incrementato o decrementato *dopo* che è stato determinato il valore dell'espressione.

Puoi effettivamente incorporarli in espressioni come questa:

```
i = 10;
j = 5 + i++;
// Calcola 5 + i, _poi_ incrementa i

printf("%d, %d\n", i, j);
// Stampa 11, 15
```

Confrontiamolo con l'operatore di pre-incremento:

```
i = 10;
j = 5 + ++i;
// Incrementa i, _poi_ calcola 5 + i

printf("%d, %d\n", i, j);
// Stampa 11, 16
```

Questa tecnica viene utilizzata frequentemente con l'accesso e la manipolazione di array e puntatori. Ti dà un modo per utilizzare il valore in una variabile e anche aumentare o diminuire quel valore prima o dopo che venga utilizzato.

Ma di gran lunga il posto più comune in cui vedrai questo è in un ciclo `for`:

```
for (i = 0; i < 10; i++)
    printf("i è %d\n", i);
```

Ma ne parleremo più avanti.

3.2.4. L'operatore virgola

Questo è un modo non comunemente utilizzato per separare le espressioni che verranno eseguite da sinistra a destra:

```
x = 10, y = 20;
// Assegna prima 10 a x, poi 20 a y
```

Sembra un po' sciocco dato che potresti semplicemente sostituire la virgola con un punto e virgola giusto?

```
x = 10; y = 20;
// Assegna prima 10 a x, poi 20 a y
```

Ma questo è un po' diverso. Quest'ultima è composta da due espressioni separate, mentre la prima è un'unica espressione!

```
x = (1, 2, 3);

printf("x is %d\n", x);
// Stampa 3, perché 3 è più a destra nell'elenco delle virgole
```

Ma anche questo è piuttosto artificioso. Un posto in cui viene comunemente utilizzato l'operatore virgola è nei cicli `for` per eseguire più operazioni in ciascuna sezione dell'istruzione:

```
for (i = 0, j = 10; i < 100; i++, j++)
    printf("%d, %d\n", i, j);
```

Lo riguarderemo più tardi.

3.2.5. Operatori condizionali

Per i valori booleani abbiamo una serie di operatori standard:

```
a == b;
// Vero se a è equivalente a b
a != b;
// Vero se a non è equivalente a b
a < b;
// Vero se a è minore di b
a > b;
// Vero se a è maggiore di b
a <= b;
// Vero se a è minore o uguale a b
a >= b;
// Vero se a è maggiore o uguale a b
```

Non confondere l'assegnazione `=` con il confronto `==` ! Usa due uguali per confrontare, uno per assegnare.

Possiamo usare le espressioni di confronto con le istruzioni `if`:

```
if (a <= 10)
    printf("Successo!\n");
```

3.2.6. Operatori booleani

Possiamo concatenare o alterare le espressioni condizionali con gli operatori booleani per *and*, *or*, e *not*.

Operatore	significato booleano
&&	and
$\vert \vert$	or
!	not

Un esempio di booleano “and”:

```
// Fa qualcosa se x è minore di 10 ey maggiore di 20:

if (x < 10 && y > 20)
```

```
  printf("Fai qualcosa!\n");
```

Un esempio di booleano “not”:

```
if (!(x < 12))
    printf("x non è inferiore a 12\n");
```

! ha una precedenza maggiore rispetto agli altri operatori booleani, quindi in questo caso dobbiamo usare le parentesi.

Naturalmente è lo stesso di:

```
if (x >= 12)
    printf("x non è inferiore a 12\n");
```

ma mi serviva l'esempio!

3.2.7. L’operatore `sizeof`

Questo operatore indica la dimensione (in byte) che una particolare variabile o tipo di dati utilizza in memoria.

Più in particolare ti dice la dimensione (in byte) che il *tipo di una particolare espressione* (che potrebbe essere solo una singola variabile) utilizza in memoria.

Questo può essere diverso su sistemi diversi ad eccezione di `char` e le sue varianti (che sono sempre 1 byte).

E questo potrebbe non sembrare molto utile ora, ma ne faremo riferimento qua, quindi vale la pena trattarlo.

Dato che questo calcola il numero di byte necessari per immagazzinare un tipo, potresti pensare che restituirebbe un `int`. Oppure… poiché la dimensione non può essere negativa forse un `unsigned`?

Ma scopriamo che C ha un tipo speciale per rappresentare il valore ritornato da sizeof. È `size_t`, pronunciato “size tee”[41] . Sappiamo solo che è un tipo intero senza segno che contiene la dimensione in byte di qualsiasi cosa restituisce `sizeof`.

`size_t` si presenta in molti posti diversi dove il conteggio delle cose vengono passati o restituiti. Consideratelo come un valore che rappresenta il conteggio.

Puoi prendere il `sizeof` di una variabile o di un'espressione:

```
int a = 999;

// %zu è l'identificatore di formato per il tipo size_t

printf("%zu\n", sizeof a);
// Stampa 4 sul mio sistema
printf("%zu\n", sizeof(2 + 7));
// Stampa 4 sul mio sistema
printf("%zu\n", sizeof 3.14);
// Stampa 8 sul mio sistema

// Se è necessario stampare valori size_t negativi utilizzare %zd
```

Ricorda: è la dimensione in byte del *tipo* dell'espressione, non la dimensione dell'espressione stessa. Ecco perché la dimensione di `2+7` è uguale alla dimensione di `a`: sono entrambi di tipo `int`. Rivisiteremo questo numero 4 nel prossimo blocco di codice…

41 _t è l'abbreviazione di `type`.

…Dove vedremo che si può prendere il `sizeof` e di un tipo (nota che le parentesi sono obbligatorie attorno al nome di un tipo a differenza di un'espressione):

```
printf("%zu\n", sizeof(int));
// Stampa 4 sul mio sistema
printf("%zu\n", sizeof(char));
// Stampa 1 su tutti i sistemi
```

È importante notare che `sizeof` è un'operazione in *fase di compilazione*[42] . Il risultato dell'espressione viene determinato interamente in fase di compilazione, non in fase di esecuzione.

Ne faremo uso in seguito.

3.3. Controllo del flusso

I booleani vanno tutti bene ma ovviamente non arriviamo da nessuna parte se non riusciamo a controllare il flusso del programma. Diamo un'occhiata a una serie di costrutti: `if`, `for`, `while` e `do-while`.

Innanzitutto, una nota generale previsionale sulle dichiarazioni e sui blocchi di dichiarazioni fornite dal tuo amichevole sviluppatore C:

Dopo un'istruzione `if` o `while`,puoi mettere una singola istruzione da eseguire o un blocco di istruzioni da eseguire tutte in sequenza.

Cominciamo con una sola dichiarazione:

```
if (x == 10) printf("x is 10\n");
```

Talvolta viene anche scritto su una riga separata.(Gli spazi bianchi sono in gran parte irrilevanti in C: a differenza di Python.)

```
if (x == 10)
    printf("x è 10\n");
```

Ma cosa succede se vuoi che succedano più cose a causa del condizionale? È possibile utilizzare le parentesi graffe per contrassegnare un *blocco* o un'*istruzione composta*.

```
  if (x == 10)
      printf("x è 10\n");
      printf("E anche questo accade quando x è 10\n");
}
```

È uno stile molto comune usare *sempre* le parentesi graffe anche se non è necessario:

```
if (x == 10) {
    printf("x è 10\n");
}
```

Alcuni sviluppatori ritengono che il codice sia più facile da leggere ed eviti errori come questo in cui le cose sembrano visivamente essere nel blocco `if`, ma in realtà non lo sono.

```
// ESEMPIO DI ERRORE PESSIMO

if (x == 10)
    printf("Ciò accade se x è 10\n");
    printf("Questo accade SEMPRE\n");
// Sorpresa!! Incondizionato!
```

`while` e `for` e gli altri loop funzionano allo stesso modo degli esempi precedenti. Se vuoi fare più

42 Tranne che con array di lunghezza variabile, ma questa è una storia per un'altra volta.

cose in un ciclo o dopo un `if`, avvolgile tra parentesi graffe.

In altre parole l'`if` eseguirà l'unica cosa dopo l'`if`. E quell'unica cosa può essere una singola affermazione o un blocco di affermazioni.

3.3.1. L'istruzione `if-else`

Abbiamo già utilizzato `if` per più esempi poiché è probabile che tu l'abbia già visto in un altro linguaggio, eccone un altro:

```
int i = 10;

if (i > 10) {
    printf("Sì, i è maggiore di 10.\n");
    printf("E questo verrà stampato anche se i è maggiore di 10.\n");
}

if (i <= 10) printf("i è inferiore o uguale a 10.\n");
```

Nel codice di esempio il messaggio verrà stampato se `i` è maggiore di 10, altrimenti l'esecuzione continua alla riga successiva. Da notare le parentesi graffe dopo l'istruzione `if`; se la condizione è vera verrà eseguita la prima istruzione o espressione subito dopo l'`if`, oppure verrà eseguita l'altro blocco di codice racchiusa tra le parentesi graffe dopo l'`if`. Questo tipo di comportamento del *blocco di codice* è comune a tutte le istruzioni.

Naturalmente poiché il C è divertente in questo modo, puoi anche fare qualcosa se la condizione è falsa con una clausola `else` sul tuo `if`:

```
int i = 99;

if (i == 10)
    printf("i è 10!\n");
else {
    printf("i decisamente non è 10.\n");
    printf("Il che mi irrita un po', francamente.\n");
}
```

E puoi anche metterli in cascata per testare una varietà di condizioni, come questa:

```
int i = 99;

if (i == 10)
    printf("i è 10!\n");

else if (i == 20)
    printf("i è 20!\n");

else if (i == 99) {
    printf("i è 99! Il mio preferito\n");
    printf("Non riesco a dirti quanto sono felice.\n");
    printf("Davvero.\n");
}

else
    printf("i è un numero pazzesco di cui non ho mai sentito parlare.\n");
```

Tuttavia se persegui questa strada dai un'occhiata all'istruzione `switch` per una soluzione potenzialmente migliore.Il problema è che `switch` funziona solo con confronti di uguaglianza con numeri costanti. La cascata `if-else` di cui sopra potrebbe controllare la disuguaglianza, gli

intervalli, le variabili o qualsiasi altra cosa che puoi creare in un'espressione condizionale.

3.3.2. L'istruzione while

`while` è il costrutto loop ordinario. Fai una cosa mentre l'espressione di una condizione è vera.

Facciamone uno!

```
// Stampa il seguente output:
//
//   i è ora 0!
//   i è ora 1!
//   [ più o meno lo stesso tra 2 e 7 ]
//   i è ora 8!
//   i è ora 9!

int i = 0;

while (i < 10) {
    printf("i è ora %d!\n", i);
    i++;
}

printf("Tutto fatto!\n");
```

Qui otteniamo un ciclo di base. C ha anche un ciclo for che sarebbe stato più pulito per quell'esempio.

Un uso non raro di while è per i cicli infiniti in cui si ripetono finchè è vero:

```
while (1) {
    printf("1 è sempre vero, quindi questo si ripete per sempre.\n");
}
```

3.3.3. L'istruzione `do-while`

Quindi ora che abbiamo sotto controllo l'istruzione `while`, diamo un'occhiata al suo cugino strettamente correlato, `do-while`.

Sono fondamentalmente gli stessi, tranne se la condizione del ciclo è falsa al primo passaggio, `do-while` verrà eseguito una volta, ma `while` non verrà eseguito affatto. In altre parole, il test per vedere se eseguire o meno il blocco avviene alla *fine* del blocco con `do-while`. Succede all'*inizio* del blocco con `while`.

Vediamo con l'esempio:

```
// Utilizzando un'istruzione while:

i = 10;

// questo non viene eseguito perché i non è inferiore a 10:
while(i < 10) {
    printf("mentre: Io è %d\n", i);
    i++;
}

// Utilizzando un'istruzione do- while:

i = 10;
```

```
// questo viene eseguito una volta, perché la condizione del ciclo non viene
controllata fino a
// dopo l'esecuzione del corpo del ciclo:

do {
    printf("do-while: i è %d\n", i);
    i++;
} while (i < 10);

printf("Tutto fatto!\n");
```

Si noti che in entrambi i casi la condizione del ciclo è immediatamente falsa. Quindi se in while, il ciclo fallisce e il blocco di codice successivo non viene mai eseguito. Con il `do-while`, invece, la condizione viene verificata *dopo* l'esecuzione del blocco di codice, quindi viene sempre eseguito almeno una volta. In questo caso, stampa il messaggio, incrementa `i`, poi fallisce la condizione e continua fino alla schermata "Tutto fatto!".

La morale della storia è questa: se vuoi che il ciclo venga eseguito almeno una volta indipendentemente dalle condizioni del ciclo usa `do-while`.

Tutti questi esempi avrebbero potuto essere fatti meglio con un ciclo `for`. Facciamo qualcosa di meno deterministico: si ripete finché non esce un certo numero casuale!

```
#include <stdio.h>
// Per printf
#include <stdlib.h>
// Per rand

int main(void)
{
    int r;

    do {
        r = rand() % 100;
// Ottieni un numero casuale compreso tra 0 e 99
        printf("%d\n", r);
    } while (r != 37);
// Ripeti finché non esce il 37
}
```

Nota a margine: l'hai eseguito più di una volta? Se lo hai fatto, hai notato che la stessa sequenza di numeri è apparsa di nuovo. E di nuovo. E di nuovo? Questo perché rand() è un generatore di numeri pseudocasuali che deve essere *seminato* con un numero diverso per generare una sequenza diversa. Cerca la funzione `srand()`[43] per maggiori dettagli.

3.3.4. L'istruzione `for`

Benvenuti in uno dei cicli più famosi al mondo! Il ciclo `for`!

Questo è un ottimo loop se conosci in anticipo il numero di volte che desideri eseguire il loop.

Potresti fare la stessa cosa usando solo un ciclo `while`, ma il ciclo `for` può aiutare a mantenere il codice più pulito.

Ecco due pezzi di codice equivalenti: nota come il ciclo `for` lo rappresenta in forma più compatta:

```
// Stampa i numeri compresi tra 0 e 9 con loro compresi...

// Utilizzando un'istruzione while:
```

43 https://beej.us/guide/bgclr/html/split/stdlib.html#man-srand

```
i = 0;
while (i < 10) {
    printf("i è %d\n", i);
    i++;
}

// Do the exact same thing with a for-loop:

for (i = 0; i < 10; i++) {
    printf("i è %d\n", i);
}
```

Esatto gente: fanno esattamente la stessa cosa. Ma puoi vedere come l'istruzione `for` sia un po' più compatta e gradevole alla vista.(Gli utenti JavaScript apprezzeranno appieno C a questo punto.)

È diviso in tre parti, separate da punto e virgola. La prima è l'inizializzazione, la seconda è la condizione del ciclo e la terza è ciò che dovrebbe accadere alla fine del blocco se la condizione del ciclo è vera. Tutte e tre queste parti sono opzionali.

```
for (inizializzare cose; ciclo if se true; do fare questo ogni ciclo)
```

Tieni presente che il ciclo non verrà eseguito nemmeno una volta se la condizione del ciclo inizia come falsa.

> **Fatto divertente sul ciclo `for`!** Puoi usare l'operatore virgola per fare più cose in ciascuna clausola del ciclo `for`!
>
> `for (i = 0, j = 999; i < 10; i++, j--) {    printf("%d, %d\n", i, j); }`

Un `for` vuoto funzionerà per sempre:

```
for(;;) {
// "per sempre"
    printf("Lo stamperò ancora e ancora e ancora\n" );
    printf("per tutta l'eternità fino alla morte termica dell'universo.\n");

    printf("O finché non premi CTRL-C.\n");
}
```

3.3.5. L’istruzione switch

A seconda delle linguaggio da cui provieni potresti avere o meno familiarità con `switch` oppure la versione di C potrebbe anche essere più restrittiva di quella a cui sei abituato. Questa è un'istruzione che ti consente di intraprendere una varietà di azioni a seconda del valore di un'espressione intera.

Fondamentalmente valuta un'espressione in un valore intero, salta al `case` che corrisponde a quel valore. L'esecuzione riprende da quel punto. Se viene incontrata un'istruzione `break`, l'esecuzione salta fuori dallo `switch`.

Ecco un esempio in cui, per un dato numero di capre, stampiamo un'idea di quante capre siano.

```
#include <stdio.h>

int main(void)
{
    int capre_contatore = 2;
```

```
    switch (capre_contatore) {
        case 0:
            printf("Non hai capre.\n");
            break;

        case 1:
            printf("Hai una capra sola.\n");
            break;

        case 2:
            printf("Hai un paio di capre.\n");
            break;

        default:
            printf("Hai una vera e propria pletora di capre!\n");
            break;
    }
}
```

In questo esempio, lo `switch` passerà al `case` 2 e verrà eseguito da lì. Quando (if) incontra un `break`, salta fuori dal `switch`. Inoltre, potresti vedere l'etichetta `default` lì in basso. Questo è ciò che accade quando nessun caso corrisponde.

Ogni `case` incluso quello `default` è facoltativo. E possono verificarsi in qualsiasi ordine ma è tipico che il valore `default` se presente, sia elencato per ultimo.

Quindi il tutto si comporta come una cascata `if-else`:

```
if (capre_contatore == 0)
    printf("Non hai capre.\n");
else if (capre_contatore == 1)
    printf("Hai una capra sola.\n");
else if (capre_contatore == 2)
    printf("Hai un paio di capre.\n");
else
    printf("Hai una vera e propria pletora di capre!\n");
```

Con alcune differenze fondamentali:

- `switch` è spesso più veloce nel passare al codice corretto (sebbene le specifiche non forniscano tale garanzia).
- `if-else` può usare i condizionali relazionali come < e >=, i float e altri tipi, mentre `switch` no.

C'è un'altra cosa interessante riguardo allo switch che a volte vedi che è piuttosto interessante: Si dissolve.

Ricorda come `break` ci fa saltare fuori dallo switch?

Bene, cosa succede se *non* usiamo `break`?

Si scopre che continuiamo ad andare al `case` successivo! Dimostrazione!

```
switch (x) {
    case 1:
        printf("1\n");
        // Si Schianta!
    case 2:
        printf("2\n");
        break;
```

```
    case 3:
        printf("3\n");
        break;
}
```

Se `x == 1` questo `switch` prima colpirà il `case` 1 e stamperà 1, ma poi proseguirà con la riga di codice successiva… che stamperà 2!

E poi finalmente usiamo `break`, quindi saltiamo fuori dallo `switch`.

se `x == 2`, allora soddisfiamo il `case` 2, stampiamo 2 e interrompiamo (`break`) normalmente.

Non avere una `break` si chiama *dissolversi*.

Suggerimento: inserisci *SEMPRE* un commento nel codice in cui intendi fallire come ho fatto sopra. Eviterà che altri programmatori si chiedano se intendevi farlo.

Infatti questo è uno dei modi comuni in cui introdurre bug nei programmi C: dimenticare di mettere un `break` nel `case`. Devi farlo se non vuoi finire nel caso successivo[44] . In precedenza ho detto che lo `switch` funziona con i tipi interi: Fallo in questo modo. Non utilizzare tipi a virgola mobile o stringa. Una scappatoia qui è che puoi usare i tipi di carattere perché questi sono segretamente numeri interi. Quindi questo è perfettamente accettabile:

```
char c = 'b';

switch (c) {
    case 'a':
        printf("È 'a'!\n");
        break;

    case 'b':
        printf("È 'b'!\n");
        break;

    case 'c':
        printf("È 'c'!\n");
        break;
}
```

Infine, puoi utilizzare le `enum` in `switch` poiché sono anche tipi interi. Ma ne parleremo più approfonditamente nel capitolo sull'`enum`.

4. Funzioni

> ***“Signore, non in un ambiente come questo. Ecco perché sono stato programmato anche per oltre trenta funzioni secondarie che—”***
>
> —C3PO, prima di essere bruscamente interrotto, riportando un numero ormai insignificante di funzioni aggiuntive, script di Star Wars

Proprio come gli altri linguaggi a cui sei abituato, C ha il concetto di *funzioni*.

Le funzioni possono accettare una varietà di *argomenti* e restituire un valore. Una cosa importante, però: gli argomenti e i tipi di valore restituito sono predichiarati, perché è così che piace al C!

Diamo un'occhiata a una funzione. Questa è una funzione che accetta un `int` come argomento e

44 Questo è stato considerato un pericolo tale che i progettisti del linguaggio di programmazione Go La lingua modificata non è quella predefinita; devi usare esplicitamente Go's dichiarazione fallthrough se vuoi cadere nel caso successivo.

restituisce un `int`.

```
#include <stdio.h>

int plus_one(int n)
// La "definizione"
{
    return n + 1;
}
```

L'`int` prima di `plus_one` indica il tipo restituito.

L'`int n` indica che questa funzione accetta un argomento `int`, memorizzato nel parametro `n`. Un parametro è un tipo speciale di variabile locale in cui vengono copiati gli argomenti.

Spiegherò il punto in cui gli argomenti vengono copiati nei parametri, qui. Molte cose in C sono più facili da capire se sai che il parametro è una copia dell'argomento, non l'argomento stesso. Ne parleremo più avanti tra un minuto.

Continuando il programma fino a `main()` possiamo vedere la chiamata alla funzione, dove assegniamo il valore restituito alla variabile locale `j`:

```
int main(void)
{
    int i = 10, j;

    j = plus_one(i);
// La "chiamata"

    printf("i + 1 è %d\n", j);
}
```

Prima che me ne dimentichi, nota che ho definito la funzione prima di usarla. Se non l'avessi fatto,il compilatore non lo saprebbe quando compila `main()` e avrebbe dato un errore di chiamata di funzione sconosciuta. Esiste un modo più corretto per eseguire il codice precedente con prototipi di funzioni, ma ne parleremo più avanti. Nota anche che `main()` è una funzione!

Restituisce un `int`.

Ma cos'è questa cosa del `void`? Questa è una parola chiave utilizzata per indicare che la funzione non accetta argomenti.

Puoi anche restituire `void` per indicare che non restituisci un valore:

```
#include <stdio.h>

// Questa funzione non accetta argomenti e non restituisce alcun valore:

void ciao(void)
{
    printf("Ciao mondo!\n");
}

int main(void)
{
    ciao();
// Stampa "Ciao mondo!"
}
```

4.1. Passaggio per Valore

Ho menzionato prima che quando passi un argomento a una funzione, una copia di quell'argomento viene creata e memorizzata nel parametro corrispondente.

Se l'argomento è una variabile, una copia del valore di quella variabile viene creata e memorizzata nel parametro.

Più in generale, viene valutata l'intera espressione dell'argomento e ne viene determinato il valore. Tale valore viene copiato nel parametro.

In ogni caso, il valore nel parametro è una cosa propria. È indipendente da qualunque valore o variabile tu abbia usato come argomenti quando hai effettuato la chiamata alla funzione. Quindi diamo un'occhiata a un esempio qui. Studialo e vedi se riesci a determinare l'output prima di eseguirlo:

```
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main(void)
{
    int i = 10;

    increment(i);

    printf("i == %d\n", i);
// Cosa stampa?
}
```

A prima vista sembra che `i` sia 10 e lo passiamo alla funzione `increment()`. Lì il valore viene incrementato, quindi quando lo stampiamo deve essere 11, giusto?

> “Abituati alla delusione.”
>
> —Dread Pirate Roberts, La principessa sposa

Ma non è `11`: stampa `10`! Come?

Riguarda il fatto che le espressioni che passi alle funzioni vengono copiate sui parametri corrispondenti. Il parametro è una copia, non l'originale.

Quindi sono `10` fuori in `main()`. E lo passiamo a `increment()`. Il parametro corrispondente è chiamato a in quella funzione.

E la copia avviene, come per assegnazione. Impropiamente, `a = i`. Quindi `a` quel punto a è `10`. E in `main()`, anche `i` è `10`.

Quindi incrementiamo `a` fino a `11`. Ma non modificheremo affatto `i`! E quindi rimane `10`.

Finalmente la funzione è completa. Tutte le sue variabili locali vengono scartate (ciao, `a`!) e torniamo a `main()`, dove `i` è ancora 10.

E lo stampiamo, ottenendo `10`, e abbiamo finito.

Questo è il motivo per cui nell'esempio precedente con la funzione `plus_one()` abbiamo

restituito il valore modificato localmente in modo da poterlo rivedere in `main()`.

Sembra un po' restrittivo, eh? Come se potessi recuperare solo un dato da una funzione, è quello che stai pensando. Esiste tuttavia un altro modo per recuperare i dati; Quelli in C lo chiamano passaggio per riferimento e questa è una storia che racconteremo un'altra volta.

Ma nessun nome fantasioso ti distrarrà dal fatto che TUTTO ciò che passi `a` una funzione SENZA ECCEZIONI viene copiato nel suo parametro corrispondente e la funzione opera su quella copia locale, NON IMPORTA COSA. Ricordatelo, anche quando parliamo del cosiddetto passaggio per riferimento.

4.2. Prototipi di funzioni

Quindi se ricordi dell'era glaciale, cioè qualche sezione fa, ho detto che dovevi definire la funzione prima di usarla, altrimenti il compilatore non lo avrebbe saputo in anticipo e avrebbe emesso un errore.

Questo non è del tutto vero. Puoi notificare in anticipo al compilatore che utilizzerai una funzione di un certo tipo che ha un determinato elenco di parametri. In questo modo la funzione può essere definita ovunque (anche in un file diverso), purché il *prototipo della funzione* sia stato dichiarato prima di chiamare quella funzione.

Fortunatamente il prototipo della funzione è davvero abbastanza semplice. È semplicemente una copia della prima riga della definizione della funzione con un punto e virgola aggiunto alla fine per sicurezza. Ad esempio questo codice chiama una funzione che viene definita successivamente perché prima è stato dichiarato un prototipo:

```
#include <stdio.h>

int foo(void);
// Questo è il prototipo!

int main(void)
{
    int i;

    // Possiamo chiamare foo() qui prima della sua definizione perché il
    // prototipo è già stato dichiarato, sopra!

    i = foo();

    printf("%d\n", i);
// 3490
}

int foo(void)
// Questa è la definizione, proprio come il prototipo!
{
    return 3490;
}
```

Se non dichiari la tua funzione prima di usarla (con un prototipo o con la sua definizione) stai eseguendo qualcosa chiamato *dichiarazione implicita*. Ciò era consentito nel primo standard C (C89), e quello standard prevede delle regole al riguardo ma oggi non è più consentito. E non esiste alcun motivo legittimo per fare affidamento su di esso nel nuovo codice.

Potresti notare qualcosa nel codice di esempio che abbiamo utilizzato… Cioè, abbiamo utilizzato la buona vecchia funzione `printf()` senza definirla o dichiarare un prototipo! Come possiamo farla

franca con questa illegalità? Non lo sappiamo in realtà. C'è un prototipo; è in quel file di intestazione `stdio.h` che abbiamo incluso con `#include`, ricordi? Quindi siamo ancora in regola agente!

4.3. Elenchi di parametri vuoti

Potresti vederli di tanto in tanto nel codice precedente, ma non dovresti mai codificarne uno nel nuovo codice. Utilizza sempre `void` per indicare che una funzione non accetta parametri. Non c'è mai[45] un motivo per saltare questo nel codice moderno.

Se sei bravo a ricordarti solo di inserire `void` per gli elenchi di parametri `void` nelle funzioni e nei prototipi, puoi saltare il resto di questa sezione.

Ci sono due contesti per questo:

- Omissione di tutti i parametri in cui è definita la funzione
- Omissione di tutti i parametri in un prototipo

Diamo prima un'occhiata a una potenziale definizione di funzione:

```
void foo()  // Dovrebbe davvero esserci un `void` lì dentro
{
    printf("Ciao mondo!\n");
}
```

Anche se le specifiche specificano che il comportamento in questo caso è *come se* avessi indicato `void` (C11 §6.7.6.3¶14), il tipo `void` è lì per un motivo. Usalo.

Ma nel caso di un prototipo di funzione c'è una differenza significativa tra l'utilizzo di `void` e non:

```
void foo();
void foo(void);
// Non sono gli stessi!
```

Lasciare `void` fuori dal prototipo dice al compilatore che non ci sono informazioni aggiuntive sui parametri della funzione. Disattiva effettivamente tutto quel controllo del tipo.

Con un prototipo usa **definitivamente** `void` quando hai un elenco di parametri vuoto.

5. Puntatori: rannicchiarsi per la paura!

> *"Come si arriva alla Carnegie Hall?"*
>
> *"Pratica!"*
>
> —Scherzo del XX secolo di origine sconosciuta

I puntatori sono una delle cose più temute nel linguaggio C. In effetti, sono l'unica cosa che rende questo linguaggio impegnativo. Ma perché?

Perché in tutta onestà, possono causare scosse elettriche attraverso la tastiera e saldare fisicamente le *tue* braccia in modo permanente, maledicendoti a una vita alla tastiera in questo linguaggio degli anni '70!

Veramente? Beh, non proprio. Sto solo cercando di prepararti al successo.

A seconda del linguaggio da cui provieni, potresti già comprendere il concetto di *referenza*, dove

45 Non dire mai "mai".

una variabile si riferisce a un oggetto di qualche tipo.

È più o meno la stessa cosa, tranne che dobbiamo essere più espliciti con C riguardo a quando parliamo del riferimento o della cosa a cui si riferisce.

5.1. Memoria e variabili

La memoria del computer contiene dati di tutti i tipi, giusto? Conterrà `float`, `int` o qualunque cosa tu abbia. Per rendere la memoria facile da gestire, ogni byte di memoria è identificato da un numero intero. Questi numeri interi aumentano in sequenza man mano che si avanza nella memoria [47] . Puoi pensarlo come un mucchio di caselle numerate, dove ciascuna casella contiene un byte[48] di dati.O come un grande array in cui ogni elemento contiene un byte, se provieni da un linguaggio con array. Il numero che rappresenta ciascuna casella è chiamato il suo indirizzo.

Ora, non tutti i tipi di dati utilizzano solo un byte. Ad esempio, un `int` è spesso di quattro byte, così come un `float`,ma in realtà dipende dal sistema. È possibile utilizzare l'operatore `sizeof` per determinare quanti byte di memoria utilizza un determinato tipo.

```
// %zu è lo specificatore di formato per il tipo size_t

printf("un int utilizza %zu byte di memoria\n", sizeof(int));

// A me stampa "4", ma può variare a seconda del sistema.
```

> **Curiosità sulla memoria:** quando si dispone di un tipo di dati (come il tipico `int`) che utilizza più di un byte di memoria, i byte che compongono i dati sono sempre adiacenti l'uno all'altro in memoria. A volte sono nell'ordine previsto, a volte no[49] . Sebbene C non garantisca alcun ordine di memoria particolare (dipende dalla piattaforma), è comunque generalmente possibile scrivere codice in un modo indipendente dalla piattaforma in cui non è necessario nemmeno considerare questi fastidiosi ordinamenti dei byte.

Ad ogni modo, se riusciamo ad andare avanti e ottenerlo, rullo di tamburi e un po' di musica inquietante per la definizione di un puntatore, *un puntatore è una variabile che contiene un indirizzo*. Immagina la colonna sonora classica di 2001: Odissea nello spazio a questo punto. Ba bum ba bum ba bum BAAAAH!

Ok, forse un po' esagerato qui, vero? Non c'è molto mistero sui puntatori. Sono l'indirizzo dei dati. Proprio come una variabile `int` può contenere il valore `12`, una variabile puntatore può contenere l'indirizzo dei dati.

Ciò significa che tutte queste cose significano la stessa cosa, cioè un numero che rappresenta un punto in memoria:

- Indice in memoria (se stai pensando alla memoria come a un grande array)
- Indirizzo
- Posizione

Li userò in modo intercambiabile. E sì, ho semplicemente inserito la posizione perché non si hanno

46 Tipicamente. Sono sicuro che ci siano delle eccezioni là fuori nei corridoi bui della storia informatica.

47 Tipicamente. Sono sicuro che ci siano delle eccezioni là fuori nei corridoi bui della storia informatica.

48 Un byte è un numero composto da non più di 8 cifre binarie, o bit in breve. Ciò significa che in cifre decimali, proprio come usava la nonna, può contenere un numero senza segno compreso tra 0 e 255 inclusi.

49 L'ordine in cui arrivano i byte viene definito endianness del numero. I soliti sospetti sono big-endian (con il byte più significativo per primo) e little-endian (con il byte più significativo per ultimo) o, raramente ora, mixed-endian (con i byte più significativi altrove).

mai abbastanza parole che significano la stessa cosa.

E una variabile puntatore contiene quel numero di indirizzo. Proprio come una variabile `float` potrebbe contenere `3.14159`.

Immagina di avere un mucchio di foglietti Post-it® tutti numerati in sequenza con il loro indirizzo. (Il primo è all'indice numerato `0`, il successivo all'indice `1` e così via.)

Oltre al numero che rappresenta la loro posizione, su ciascuno puoi anche scrivere un altro numero a tua scelta. Potrebbe essere il numero di cani che hai. O il numero di lune attorno a Marte…

…Oppure *potrebbe essere l'indice di un altro Post-it!*

Se hai scritto il numero di cani che hai, è solo una variabile normale. Ma se lì dentro hai scritto l'indice di un altro Post-it, *quello è un puntatore*. Indica l'altra nota!

Un'altra analogia potrebbe essere con gli indirizzi delle case. Puoi avere una casa con determinate qualità, cortile, tetto in metallo, solare, ecc. Oppure potresti avere l'indirizzo di quella casa. L'indirizzo non è lo stesso della casa stessa. Una è una casa in piena regola e l'altra è solo poche righe di testo. Ma l'indirizzo della casa è un *puntatore* a quella casa. Non è la casa in sé, ma ti dice dove trovarla.

E possiamo fare la stessa cosa nel computer con i dati. Puoi avere una variabile di dati che contenga un certo valore. E quel valore è in memoria a qualche indirizzo. E potresti avere una *variabile puntatore* diversa che contenga l'indirizzo di quella variabile di dati.

Non è la variabile dati in sé, ma, come nel caso dell'indirizzo di casa, ci dice dove trovarla.

Quando lo abbiamo, diciamo che abbiamo un "puntatore a" quei dati. E possiamo seguire il puntatore per accedere ai dati stessi.

(Sebbene non sembri ancora particolarmente utile, tutto questo diventa indispensabile se usato con chiamate di funzione. Abbi pazienza finché non arriviamo a quel punto.)

Quindi, se abbiamo un `int`, diciamo, e vogliamo un puntatore ad esso, quello che vogliamo è un modo per ottenere l'indirizzo di quell'`int`, giusto? Dopotutto, il puntatore contiene solo *l'indirizzo* dei dati. Quale operatore pensi che utilizzeremmo per trovare *l'indirizzo* dell'`int`?

Bene, con una sorpresa scioccante che deve essere una sorta di shock per te, gentile lettore, usiamo l'indirizzo dell'operatore (che sembra essere una e commerciale: "&") per trovare l'indirizzo dei dati. E commerciale.

Quindi, per un rapido esempio, introdurremo un nuovo *identificatore di formato* per `printf()` in modo da poter stampare un puntatore. Sai già come %d stampa un intero decimale, vero? Bene, %p stampa un puntatore. Ora, questo puntatore sembrerà un numero spazzatura (e potrebbe essere stampato in formato esadecimale[50] anziché decimale), ma è semplicemente l'indice nella memoria in cui sono archiviati i dati. (O l'indice in memoria in cui è archiviato il primo byte di dati, se i dati sono multibyte.) Praticamente in tutte le circostanze, inclusa questa, il valore effettivo del numero stampato non è importante per te e lo mostro qui solo per la dimostrazione dell'indirizzo dell'operatore.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("Il valore di i è %d\n", i);
```

50 Cioè, base 16 con cifre 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E e F.

```
    printf("E il suo indirizzo è %p\n", (void *)&i);

    // %p si aspetta che l'argomento sia un puntatore a void
    // quindi lo castiamo per rendere felice il compilatore.
}
```

Sul mio computer, viene stampato:

```
Il valore di i è 10
E il suo indirizzo è 0x7ffddf7072a4
```

Se sei curioso, quel numero esadecimale è 140.727.326.896.068 in decimale (base 10, proprio come usava la nonna). Questo è l'indice in memoria in cui sono archiviati i dati della variabile `i`. E' l'indirizzo di `i`. È la posizione di `i`. E' un puntatore a `i`.

> **Aspetta: hai 140 terabyte di RAM?** SÌ! Non è vero? Ma mi dilungo; ovviamente no (anno 2024). I computer moderni utilizzano una tecnologia miracolosa chiamata memoria virtuale[51] che fa credere ai processi di avere tutto lo spazio di memoria del computer per sé, indipendentemente dalla quantità di RAM fisica che ne supporta il backup. Quindi, anche se l'indirizzo era un numero enorme è stato mappato su un indirizzo di memoria fisica inferiore dal sistema di memoria virtuale della mia CPU. In particolare questo computer ha 16 GB di RAM (di nuovo, intorno al 2024, ma utilizzo Linux, quindi è abbastanza). Terabyte di RAM? Sono un insegnante, non un bazillionario delle dot-com. Niente di tutto questo è qualcosa di cui nessuno di noi deve preoccuparsi a parte il fatto che non sono straordinariamente ricco.

È un puntatore perché ti consente di sapere dove `i` si trova in memoria. Come l'indirizzo di casa scritto su un pezzo di carta indica dove si trova una determinata casa, questo numero ci indica dove nella memoria possiamo trovare il valore di `i`. Indica `i`.

Ancora una volta, in genere non ci interessa quale sia il numero esatto dell'indirizzo. Ci interessa solo che sia un puntatore a `i`.

5.2. Tipi di puntatore

Quindi… va tutto bene. Ora puoi prendere con successo l'indirizzo di una variabile e stamparlo sullo schermo. C'è qualcosina per il vecchio curriculum, vero? È qui che mi prendi per la collottola e mi chiedi educatamente a cosa cappero servono i puntratori.

Ottima domanda e ci arriveremo subito dopo questi messaggi dal nostro sponsor.

> SERVIZI DI PULIZIA DI UNITÀ ALLOGGIATIVE ROBOTIZZATE ACME. LA TUA FATTORIA SARÀ DRAMMATICAMENTE MIGLIORATA O SARAI ELIMINATO. IL MESSAGGIO FINISCE.

Bentornati ad un'altra puntata della Guida di Beej. L'ultima volta che ci siamo incontrati stavamo parlando di come utilizzare i puntatori. Bene, quello che faremo è memorizzare un puntatore in una variabile in modo da poterlo utilizzare in seguito. Puoi identificare il tipo di puntatore perché è presente un asterisco (`*`) prima del nome della variabile e dopo il suo tipo:

```
int main(void)
{
    int i;
// il tipo di i è "int"
    int *p;
```

51 https://en.wikipedia.org/wiki/Virtual_memory

```
// il tipo di p è "puntatore a un int", o "puntatore-int"
}
```

Ehi, quindi qui abbiamo una variabile che è un tipo puntatore e può puntare ad altri `int`. Cioè, può contenere l'indirizzo di altri `int`. Sappiamo che punta a `int`, poiché è di tipo `int*` (leggi “int-pointer”).

Quando esegui un'assegnazione in una variabile puntatore, il tipo del lato destro dell'assegnazione deve essere dello stesso tipo della variabile puntatore. Fortunatamente per noi, quando prendi l'indirizzo di una variabile, il tipo risultante è un puntatore a quel tipo di variabile, quindi assegnazioni come le seguenti sono perfette:

```
int i;
int *p;
// p è un puntatore, ma non è inizializzato e punta a spazzatura

p = &i;
// p viene assegnato all'indirizzo di i--p ora "punta a" i
```

A sinistra dell'assegnazione abbiamo una variabile di tipo puntatore-a `int (int*)` e sul lato destro abbiamo un'espressione di tipo puntatore-a `int` poiché i è un `int` (perché indirizzo di `int` ti dà un puntatore a `int`). L'indirizzo di una cosa può essere memorizzato in un puntatore a quella cosa.

Chiaro? So che non ha ancora molto senso dato che non hai visto un utilizzo effettivo della variabile puntatore, ma stiamo facendo piccoli passi qui in modo che nessuno si perda. Quindi ora presentiamo l'operatore anti-indirizzo di. È un po' come sarebbe `address-of` (indirizzo-di) in Mondo Bizzarro.

5.3. Dereferenziazione

Si può pensare che una variabile puntatore si *riferisca* a un'altra variabile puntando ad essa. È raro sentire qualcuno in terra C parlare di "riferimento" o "riferimenti", ma ne parlo solo per dare più senso al nome di questo operatore.

Quando hai un puntatore a una variabile (più o meno "un riferimento a una variabile"), puoi utilizzare la variabile originale attraverso il puntatore *dereferenziando* il puntatore. (Puoi pensare a questo come a “depuntare” il puntatore, ma nessuno dice mai “depuntatore”.)

Tornando alla nostra analogia, è vagamente come guardare l'indirizzo di una casa e poi andare a quella casa.

Ora, cosa intendo con "accedi alla variabile originale"? Bene, se hai una variabile chiamata `i`, e hai un puntatore a `i` chiamato `p`, puoi usare il puntatore dereferenziato `p` *esattamente come se fosse la variabile originale `i`!*

Hai quasi abbastanza conoscenze per gestire un esempio. L'ultima curiosità che devi sapere è in realtà questa: cos'è l'operatore di dereferenziazione?In realtà è chiamato *operatore indiretto*, perché accedi ai valori indirettamente tramite il puntatore. Ed è l'asterisco, ancora: `*`. Ora, non confonderlo con l'asterisco che hai usato nella dichiarazione del puntatore, in precedenza. Sono lo stesso personaggio, ma hanno significati diversi in contesti diversi[52] .

Ecco un esempio in piena regola:

```
#include <stdio.h>
```

52 Non è tutto! È utilizzato nei ***commenti***, nelle moltiplicazioni e nelle funzioni prototipi con array di lunghezza variabile! È tutto uguale * , tranne il contesto gli dà un significato diverso.

```
int main(void)
{
    int i;
    int *p;  // questo NON è una deferenziazione--questo è un tipo "int*"

    p = &i;  // p ora punta a i, p contiene l'indirizzo di i

    i = 10;  // i è ora 10
    *p = 20; // la cosa a cui punta p (nominalmente i!) è ora 20!!

    printf("i è %d\n", i);   // stampa "20"
    printf("i è %d\n", *p);  // "20"! dereferenzia-p è lo stesso di i!
}
```

Ricorda che `p` contiene l'indirizzo di `i`, come puoi vedere dove abbiamo fatto l'assegnazione a `p` alla riga 8. Ciò che fa l'operatore indiretto è dire al computer *di utilizzare l'oggetto a cui punta il puntatore* invece di utilizzare il puntatore stesso. In questo modo, abbiamo trasformato `*p` in una sorta di alias per `i`.

Ottimo, ma *perché*? Perché fare tutto questo?

5.4. Passaggio di puntatori come argomenti

In questo momento stai pensando di avere moltissima conoscenza sui puntatori, ma assolutamente zero applicazioni, giusto? Voglio dire, a che serve `*p` se invece potessi semplicemente dire `i`??

Bene, amico mio, il vero potere dei puntatori entra in gioco quando inizi a passarli alle funzioni. Perché questo è un grosso problema? Potresti ricordare da prima che potresti passare tutti i tipi di argomenti alle funzioni e sarebbero debitamente copiati in parametri, e quindi potresti manipolare copie locali di quelle variabili dall'interno della funzione, e quindi potresti restituire un singolo valore.

Cosa succederebbe se volessi riportare più di un singolo dato dalla funzione? Voglio dire, puoi restituire solo una cosa, giusto? E se rispondessi a quella domanda con un'altra domanda? …Ehm, due domande?

Cosa succede quando passi un puntatore come argomento a una funzione? Una copia del puntatore viene inserita nel parametro corrispondente? *Puoi scommetterci che è così.* Ricordi come prima continuavo a divagare su come *OGNI SINGOLO ARGOMENTO* viene copiato nei parametri e la funzione utilizza una copia dell'argomento? Ebbene, lo stesso vale qui. La funzione otterrà una copia del puntatore.

Ma, e questa è la parte intelligente: avremo impostato in anticipo il puntatore per puntare a una variabile… e poi la funzione potrà dereferenziare la sua copia del puntatore per tornare alla variabile originale! La funzione non può vedere la variabile stessa, ma può certamente dereferenziare un puntatore a quella variabile!

Ciò è analogo a scrivere l'indirizzo di casa su un pezzo di carta e poi copiarlo su un altro pezzo di carta. Ora hai due indicazioni per quella casa ed entrambe sono ugualmente efficaci nel portarti alla casa stessa.

Nel caso di una chiamata di funzione. una delle copie è memorizzata in una variabile puntatore nell'ambito chiamante e l'altra è memorizzata in una variabile puntatore che è il parametro della funzione.

Esempio! Rivisitiamo la nostra vecchia funzione `increment()`, ma questa volta facciamo in modo che incrementi effettivamente il valore nel chiamante.

```c
#include <stdio.h>

void increment(int *p)
// notare che accetta un puntatore a un int
{
    *p = *p + 1;
// aggiungi uno alla cosa a cui punta p
}

int main(void)
{
    int i = 10;
    int *j = &i;  // nota l'indirizzo-of; lo trasforma in un puntatore a i

    printf("i is %d\n", i);
// stampa "10"
    printf("i is also %d\n", *j);
// stampa "10"

    increment(j);
// j è un int*--a i

    printf("i is %d\n", i);
// stampa "11"!
}
```

OK! Ci sono un paio di cose da vedere qui… non ultimo il fatto che la funzione `increment()` accetta un `int*` come argomento. Gli passiamo un `int*` nella chiamata cambiando la variabile `int i` in un `int*` utilizzando l'operatore address-of. (Ricorda, un puntatore contiene un indirizzo, quindi creiamo puntatori a variabili facendoli passare attraverso l'operatore indirizzo di.)

La funzione `increment()` ottiene una copia del puntatore. Entrambi: Il puntatore originale `j` (in `main()`) e la copia di quel puntatore `p` (il parametro in `increment()`) punta allo stesso indirizzo, ovvero quello che contiene il valore `i`. (Ancora una volta, per analogia, come due pezzi di carta con scritto sopra lo stesso indirizzo di casa.) Anche il dereferenziamento ti consentirà di modificare la variabile originale `i`! La funzione può modificare una variabile in un altro ambito! Forza!

L'esempio precedente è spesso scritto in modo più conciso nella chiamata semplicemente utilizzando l'indirizzo di destra nell'elenco degli argomenti:

```c
printf("i is %d\n", i);
// stampa "10"
increment(&i);
printf("i is %d\n", i);
// stampa "11"!
```

Come regola generale, se vuoi che la funzione modifichi l'oggetto che stai passando in modo da vedere il risultato, dovrai passare un puntatore a quell'oggetto.

5.5. Il puntatore `NULL`

Qualsiasi variabile puntatore di qualsiasi tipo può essere impostata su un valore speciale chiamato `NULL`. Ciò indica che questo puntatore non punta a nulla.

```c
int *p;

p = NULL;
```

Poiché non punta a un valore, dereferenziarlo è un comportamento indefinito e probabilmente provocherà un arresto anomalo:

```
int *p = NULL;

*p = 12;
// INCIDENTE O QUALCOSA DI PROBABILMENTE BRUTTO. MEGLIO EVITARE.
```

Nonostante sia stato definito l'errore da un miliardo di dollari dal suo creatore[53] , il puntatore `NULL` è un buon valore sentinella[54] e un indicatore generale che un puntatore non è stato ancora inizializzato.

(Ovviamente, come altre variabili, il puntatore punta a spazzatura a meno che non lo si assegni esplicitamente per puntare a un indirizzo o `NULL`.)

5.6. Una nota sulla dichiarazione dei puntatori

La sintassi per dichiarare un puntatore può diventare un po' strana. Diamo un'occhiata a questo esempio:

```
int a;
int b;
```

Possiamo condensarlo in un'unica riga, giusto?

```
int a, b;  // Stessa cosa
```

Quindi `a` e `b` sono entrambi `int`. Nessun problema. Ma che dire di questo?

```
int a;
int *p;
```

Possiamo metterlo in una riga? Noi possiamo. Ma dove va a finire `*`?

La regola è che `*` va davanti a qualsiasi variabile che sia un tipo puntatore. Così è. `*` non fa parte dell'`int` in questo esempio. fa parte della variabile `p`.

Tenendo presente questo, possiamo scrivere questo:

```
int a, *p;
// Stessa cosa
```

È importante notare che la riga seguente non dichiara due puntatori:

```
int *p, q;
// p è un puntatore a un int; q è solo un int.
```

Ciò può risultare particolarmente insidioso se il programmatore scrive la seguente riga di codice (valida) che è funzionalmente identica a quella sopra.

```
int* p, q;
// p è un puntatore a un int; q è solo un int.
```

Quindi dai un'occhiata a questo e determina quali variabili sono puntatori e quali no:

```
int *a, b, c, *d, e, *f, g, h, *i;
```

Lascerò la risposta in una nota[55] .

53 https://en.wikipedia.org/wiki/Null_pointer#History
54 https://en.wikipedia.org/wiki/Sentinel_value
55 Le variabili di tipo puntatore sono a , d , f , e i , perché sono quelle con * davanti.

5.7. `sizeof` e puntatori

Solo un po' di sintassi qui che potrebbe creare confusione e potresti vedere di tanto in tanto.

Ricordiamo che `sizeof` opera sul *tipo* dell'espressione.

```
int *p;

// Stampa la dimensione di un 'int'
printf("%zu\n", sizeof(int));

// p è di tipo 'int *', quindi stampa la dimensione di un 'int*'
printf("%zu\n", sizeof p);

// *p è di tipo 'int', quindi stampa la dimensione di un 'int'
printf("%zu\n", sizeof *p);
```

Potresti vedere del codice in giro con quell'ultima `sizeof` lì dentro. Ricorda solo che `sizeof` riguarda il tipo di espressione, non le variabili nell'espressione stessa.

6. Arrays

> *"Gli indici degli array dovrebbero iniziare da 0 o da 1? Il mio compromesso 0,5 è stato respinto a mio avviso senza la dovuta considerazione."*
>
> —Stan Kelly-Bootle, informatico

Fortunatamente C ha array. Voglio dire, so che è considerato un linguaggio di basso livello[56] ma almeno ha il concetto di array integrato. E poiché moltissimi linguaggi si sono ispirati alla sintassi del C probabilmente hai già familiarità con l'uso di `[` e `]` per dichiarare e usare gli array.

Ma C ha *chiaramente* solo array! Come scopriremo più avanti gli array sono solo zucchero sintattico in C: in realtà sono tutti puntatori e cose del genere. *Fuori di testa!* Ma per ora usiamoli solo come array. *Uff*.

6.1. Esempio semplice

Facciamo solo un esempio:

```
#include <stdio.h>

int main(void)
{
    int i;
    float f[4];
// Dichiara un array di 4 float

    f[0] = 3.14159;
// L'indicizzazione inizia da 0 ovviamente.
    f[1] = 1.41421;
    f[2] = 1.61803;
    f[3] = 2.71828;

    // Stampiamoli tutti:

    for (i = 0; i < 4; i++) {
```

56 In questi giorni, comunque.

```
        printf("%f\n", f[i]);
    }
}
```

Quando dichiari un array devi dargli una dimensione. E la dimensione deve essere fissa[57] .

Nell'esempio sopra abbiamo creato un array di 4 `float`. Il valore tra parentesi quadre nella dichiarazione ce lo fa sapere.

Successivamente nelle righe successive accediamo ai valori dell'array impostandoli o ottenendoli sempre tramite parentesi quadre.

Spero che questo ti sembri familiare dalle lingue che già conosci!

6.2. Ottenere la lunghezza di un array

Non puoi… cioè. C non registra queste informazioni[58] . Devi gestirlo separatamente in un'altra variabile.

Quando dico "non si può" in realtà intendo che ci sono alcune circostanze in cui *puoi*. Esiste un trucco per ottenere il numero di elementi in un array nell'ambito in cui viene dichiarato un array. Ma in generale questo non funzionerà come vorresti se passi l'array a una funzione[59] .

Diamo un'occhiata a questo trucco. L'idea di base è prendere il `sizeof` dell'array e poi dividerla per la dimensione di ciascun elemento per ottenere la lunghezza. Ad esempio se un `int` è di 4 byte e l'array è lungo 32 byte deve esserci spazio per `$\frac{32}{4}$` o `int` qua.

```
int x[12];  // 12 ints

printf("%zu\n", sizeof x);
// 48 byte totali
printf("%zu\n", sizeof(int));
// 4 bytes per int

printf("%zu\n", sizeof x / sizeof(int));
// 48/4 = 12 ints!
```

Se si tratta di un array di `char` allora `sizeof` dell'array è il numero di elementi poiché `sizeof(char)` è definito come 1. Per qualsiasi altra cosa devi dividere per la dimensione di ciascun elemento.

Ma questo trucco funziona solo nell'ambito in cui è stato definito l'array. Se passi l'array a una funzione non funziona. Anche se lo rendi "grande" nell'ambito della funzione:

```
void foo(int x[12])
{
    printf("%zu\n", sizeof x);
// 8?! Cosa è successo a 48?
    printf("%zu\n", sizeof(int));
// 4 bytes per int

    printf("%zu\n", sizeof x / sizeof(int));
// 8/4 = 2 ints?? SBAGLIATO.
}
```

57 Ancora una volta, non proprio, ma gli array a lunghezza variabile, di cui non sono un vero fan, sono storia di altri tempi.

58 Poiché gli array sono solo puntatori al primo elemento dell'array sotto il cofano, non ci sono informazioni aggiuntive che registrano la durata.

59 Perché quando passi un array a una funzione, in realtà stai semplicemente passando a puntatore al primo elemento dell'array, non all'intero array.

Questo perché quando si “passano” gli array alle funzioni si passa solo un puntatore al primo elemento e questo è ciò che misura `sizeof`. Maggiori informazioni su questo nella sezione Passaggio di array monodimensionali alle funzioni di seguito.

Un'altra cosa che puoi fare con `sizeof` e arrays è ottenere la dimensione di un array di un numero fisso di elementi senza dichiarare l'array. È come ottenere la dimensione di un `int` con `sizeof(int)`.

Ad esempio per vedere quanti byte sarebbero necessari per un array di 48 `double` puoi fare questo:

```
sizeof(double [48]);
```

6.3. Inizializzatori di array

È possibile inizializzare un array con costanti in anticipo:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};  // Inizializza con questi valori

    for (i = 0; i < 5; i++) {
        printf("%d\n", a[i]);
    }
}
```

Non dovresti mai avere più elementi nel tuo inizializzatore di quanti ce ne sia spazio nell'array, altrimenti il compilatore diventerà irritabile:

```
foo.c: In function ‘main’:
foo.c:6:39: warning: excess elements in array initializer
    6 |     int a[5] = {22, 37, 3490, 18, 95, 999};
      |                                       ^~~
foo.c:6:39: note: (near initialization for ‘a’)
```

Ma (fatto divertente!) puoi avere *meno* elementi nel tuo inizializzatore di quanti ce ne sia spazio nell'array. Gli elementi rimanenti nell'array verranno automaticamente inizializzati con zero. Questo è vero in generale per tutti i tipi di inizializzatori di array: se hai un inizializzatore tutto ciò che non è impostato esplicitamente su un valore verrà impostato su zero.

```
int a[5] = {22, 37, 3490};

// equivale a:

int a[5] = {22, 37, 3490, 0, 0};
```

È una comune vederlo in un inizializzatore quando vuoi impostare un intero array su zero:

```
int a[100] = {0};
```

Il che significa: "Rendi zero il primo elemento quindi azzera automaticamente anche il resto".

È inoltre possibile impostare elementi specifici dell'array nell'inizializzatore specificando un indice per il valore! Quando lo fai C continuerà felicemente a inizializzare i valori successivi per te finché l'inizializzatore non si esaurisce, riempiendo tutto il resto con `0`.

Per farlo inserisci l'indice tra parentesi quadre con un `=` dopo, quindi imposta il valore.

Ecco un esempio in cui costruiamo un array:

```
int a[10] = {0, 11, 22, [5]=55, 66, 77};
```

Poiché abbiamo elencato l'indice 5 come inizio per 55 i dati risultanti nell'array sono:

```
0 11 22 0 0 55 66 77 0 0
```

Puoi inserire anche semplici espressioni costanti.

```
#define COUNT 5

int a[COUNT] = {[COUNT-3]=3, 2, 1};
```

che ci dà:

```
0 0 3 2 1
```

Infine puoi anche fare in modo che C calcoli la dimensione dell'array dall'inizializzatore semplicemente lasciando disattivata la dimensione:

```
int a[3] = {22, 37, 3490};

// equivale a:

int a[] = {22, 37, 3490};  // Lascia fuori la dimensione!
```

6.4. Fuori dai limiti!

C non ti impedisce di accedere agli array fuori dai limiti. Potrebbe anche non avvisarti.

Rubiamo l'esempio sopra e continuiamo a stampare la fine dell'array. Ha solo 5 elementi ma proviamo a stamparne 10 e vediamo cosa succede:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};

    for (i = 0; i < 10; i++) {  // CATTIVE NOTIZIE: stampa troppi elementi!
        printf("%d\n", a[i]);
    }
}
```

Eseguendolo sul mio computer viene stampato:

```
22
37
3490
18
95
32765
1847052032
1780534144
-56487472
21890
```

Accidenti! Che cos'è? Bene, risulta che la stampa della fine di un array si traduce in quello che gli sviluppatori C chiamano *comportamento indefinito*. Parleremo più approfonditamente di questa bestia dopo, ma per ora significa: "Hai fatto qualcosa di brutto e potrebbe succedere di tutto durante l'esecuzione del programma".

E con qualsiasi cosa intendo tipicamente cose come trovare zeri, trovare numeri spazzatura o bloccarsi. Ma in realtà le specifiche C dicono che in queste circostanze il compilatore può emettere codice che fa *qualsiasi cosa*[60] .

Versione breve: non fare nulla che causi un comportamento indefinito. Mai[61] .

6.5. Array multidimensionali

Puoi aggiungere tutte le dimensioni che desideri ai tuoi array.

```
int a[10];
int b[2][7];
int c[4][5][6];
```

Questi sono archiviati in memoria sulla riga in ordine crescente[62] . Ciò significa che con un array 2D, il primo indice elencato indica la riga e il secondo la colonna.

Puoi anche utilizzare gli inizializzatori su array multidimensionali nidificandoli:

```
#include <stdio.h>

int main(void)
{
    int row, col;

    int a[2][5] = {      // Inizializza un array 2D
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9}
    };

    for (row = 0; row < 2; row++) {
        for (col = 0; col < 5; col++) {
            printf("(%d,%d) = %d\n", row, col, a[row][col]);
        }
    }
}
```

Per l'output di:

```
(0,0) = 0
(0,1) = 1
(0,2) = 2
(0,3) = 3
(0,4) = 4
(1,0) = 5
(1,1) = 6
(1,2) = 7
(1,3) = 8
(1,4) = 9
```

E puoi inizializzare con indici espliciti:

```
// Crea una matrice identità 3x3
```

60 Ai vecchi tempi di MS-DOS, prima che la protezione della memoria esistesse, scrivevo del codice C particolarmente abusivo che coinvolgeva deliberatamente tutti tipi di comportamento indefiniti. Ma sapevo cosa stavo facendo, e le cose stavano funzionando piuttosto bene. Finché non ho fatto un passo falso che mi ha causato il blocco e, al riavvio ho visto che ha distrutto tutte le mie impostazioni del BIOS. È stato divertente. (Un grido a @man per quei momenti divertenti.)

61 Ci sono molte cose che causano un comportamento indefinito, non solo fuori luogo limita gli accessi all'array. Questo è ciò che rende il linguaggio C così entusiasmante.

62 https://en.wikipedia.org/wiki/Row-_and_column-major_order

```
int a[3][3] = {[0][0]=1, [1][1]=1, [2][2]=1};
```

che costruisce un array 2D come questo:

```
1 0 0
0 1 0
0 0 1
```

6.6. Array e puntatori

[Casualmente] Quindi… forse avrei potuto mezionare lassù che gli array erano puntatori in fondo? Dovremmo fare un tuffo superficiale in questo momento in modo che le cose non siano completamente confuse. Più avanti vedremo qual è la reale relazione tra array e puntatori, ma per ora voglio solo esaminare il passaggio di array alle funzioni.

6.6.1. Ottenere un puntatore a un array

Voglio svelarti un segreto. In generale quando un programmatore C parla di un puntatore a un array parla di un puntatore al *primo elemento* dell'array[63] .

Quindi prendiamo un puntatore al primo elemento di un array.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];  // p punta all'array
                // Beh, al primo elemento, in realtà

    printf("%d\n", *p);  // Stampa "11"
}
```

Questo è così comune da fare in C che il linguaggio ci consente una scorciatoia:

```
p = &a[0];
// p punta all'array

// equivale a:

p = a;      // p punta all'array, ma molto più carino da vedere!
```

Fare riferimento al nome dell'array isolatamente equivale a ottenere un puntatore al primo elemento dell'array! Lo useremo ampiamente nei prossimi esempi.

Ma aspetta un secondo: `p` non è un `int*`? E `*p` ci dà `11`, uguale a `a[0]`? Siìì. Stai iniziando a intravedere la relazione tra array e puntatori in C.

6.6.2. Passaggio di array unidimensionali alle funzioni

Facciamo un esempio con un array monodimensionale. Scriverò un paio di funzioni a cui possiamo passare l'array che fanno cose diverse.

Preparati per alcune funzioni strabilianti!

63 Questo è tecnicamente errato, in quanto puntatore a un array e puntatore al primo gli elementi di un array hanno tipi diversi. Ma potremo bruciare quel ponte quando ci arriviamo.

```c
#include <stdio.h>

// Passando come puntatore al primo elemento
void times2(int *a, int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 2);
}

// Stessa cosa, ma usando la notazione di array
void times3(int a[], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 3);
}

// Stessa cosa, ma usando la notazione di array con size
void times4(int a[5], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 4);
}

int main(void)
{
    int x[5] = {11, 22, 33, 44, 55};

    times2(x, 5);
    times3(x, 5);
    times4(x, 5);
}
```

Tutti questi metodi per elencare l'array come parametro nella funzione sono identici.

```c
void times2(int *a, int len)
void times3(int a[], int len)
void times4(int a[5], int len)
```

Utilizzato dagli utenti abituali del C, il primo è di gran lunga il più comune.

E infatti in quest'ultima situazione al compilatore non interessa nemmeno quale numero passi (a parte il fatto che deve essere maggiore di zero[64]). Non impone assolutamente nulla.

Ora che l'ho detto, la dimensione dell'array nella dichiarazione della funzione in realtà è importante quando si passano array multidimensionali in funzioni, ma torniamo a questo.

6.6.3. Modifica degli array nelle funzioni

Abbiamo detto che gli array sono solo puntatori sotto mentite spoglie. Ciò significa che se passi un array a una funzione probabilmente stai passando un puntatore al primo elemento dell'array.

Ma se la funzione ha un puntatore ai dati è in grado di manipolarli! Pertanto le modifiche apportate da una funzione a un array saranno visibili nel chiamante.

64 C11 §6.7.6.2¶1 richiede che sia maggiore di zero. Ma potresti vedere il codice fuori lì con array dichiarati di lunghezza zero alla fine di struct s e GCC is particolarmente indulgente a riguardo a meno che non si compili con -pedantic . Questo zero- length array era un meccanismo hacker per creare strutture di lunghezza variabile. Sfortunatamente, accedere a un array di questo tipo è un comportamento tecnicamente indefinito anche se praticamente funzionava ovunque. C99 ha codificato una cosa ben definita sostituzione per esso chiamato membri dell'array flessibile, di cui parleremo più tardi.

Ecco un esempio in cui passiamo un puntatore a un array a una funzione la funzione manipola i valori in quell'array e tali modifiche sono visibili nel chiamante.

```c
#include <stdio.h>

void double_array(int *a, int len)
{
    // Moltiplica ogni elemento per 2
    //
    // Questo raddoppia i valori in x in main() poiché x ed entrambi puntano
    // allo stesso array in memoria!

    for (int i = 0; i < len; i++)
        a[i] *= 2;
}

int main(void)
{
    int x[5] = {1, 2, 3, 4, 5};

    double_array(x, 5);

    for (int i = 0; i < 5; i++)
        printf("%d\n", x[i]);  // 2, 4, 6, 8, 10!
}
```

Anche se abbiamo passato l'array come parametro a che è di tipo `int*`, guarda come accediamo utilizzando la notazione di array con `a[i]`! Coooosa? Ciò è totalmente consentito.

Più avanti quando parleremo dell'equivalenza tra array e puntatori vedremo come questo abbia molto più senso. Per ora è sufficiente sapere che le funzioni possono apportare modifiche agli array visibili nel chiamante.

6.6.4. Passaggio di array multidimensionali alle funzioni

La storia cambia leggermente quando parliamo di array multidimensionali. C ha bisogno di conoscere tutte le dimensioni (tranne la prima) quindi ha informazioni sufficienti per sapere dove cercare in memoria per trovare un valore.

Ecco un esempio in cui siamo espliciti con tutte le dimensioni:

```c
#include <stdio.h>

void print_2D_array(int a[2][3])
{
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int x[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
```

```
    print_2D_array(x);
}
```

Ma in questo caso questi due[65] sono equivalenti:

```
void print_2D_array(int a[2][3])
void print_2D_array(int a[][3])
```

Il compilatore in realtà necessita solo della seconda dimensione in modo da poter capire quanto saltare in memoria per ogni incremento della prima dimensione. In generale è necessario conoscere tutte le dimensioni tranne la prima.

Inoltre ricorda che il compilatore esegue un controllo minimo dei limiti in fase di compilazione (se sei fortunato) e C non esegue alcun controllo dei limiti in fase di esecuzione. Niente cinture di sicurezza! Non andare in crash accedendo a elementi dell'array fuori dai limiti!

7. stringhe

Finalmente! Stringhe! Cosa potrebbe essere più semplice?

Beh, a quanto pare le stringhe non sono effettivamente stringhe in C. Esatto! Sono indicatori! Certo che lo sono!

Proprio come gli array, le stringhe in C *esistono a malapena*.

Ma diamo un'occhiata: non è poi un grosso problema.

7.1. Stringhe letterali

Prima di iniziare, parliamo delle stringhe letterali in C. Si tratta di sequenze di caratteri racchiuse tra *virgolette* doppie (`"`). (Le virgolette singole racchiudono caratteri e sono un animale completamente diverso.) Esempi:

```
"Ciao mondo!\n"
"Questa è una prova."
"Quando le è stato chiesto se questa stringa contenesse virgolette, ha
risposto: \"Sì.\""
```

Il primo ha un ritorno a capo alla fine: una cosa abbastanza comune da vedere.

L'ultimo ha delle virgolette incorporate al suo interno ma vedi che ognuna è preceduta da (diciamo "sequenza di uscita") un backslash (`\`) indica che a questo punto nella stringa appartiene una virgoletta letterale. Questo è il modo in cui il compilatore C può distinguere tra la stampa di una virgoletta doppia e la virgoletta doppia alla fine della stringa.

7.2. Variabili stringa

Ora che sappiamo come rendere letterale una stringa assegniamola a una variabile in modo da poterci fare qualcosa.

```
char *s = "Ciao mondo!";
```

Dai un'occhiata a quel tipo: puntatore a un `char`. La variabile stringa `s` è in realtà un puntatore al primo carattere di quella stringa, vale a dire `H`.

E possiamo stamparlo con l'identificatore di formato `%s` (per "string"):

```
char *s = "Ciao mondo!";
```

65 Anche questo è equivalente: `void print_2D_array(int (*a)[3])`, ma questo è più di quello in cui voglio addentrarmi adesso.

```
printf("%s\n", s);
// "Ciao mondo!"
```

7.3. Variabili stringa come array

Un'altra opzione è questa quasi equivalente all'utilizzo di `char*` sopra:

```
char s[14] = "Ciao mondo!";

// oppure, se fossimo adeguatamente pigri e avessimo il compilatore
// che calcola la lunghezza per noi:

char s[] = "Ciao mondo!";
```

Ciò significa che puoi utilizzare la notazione di array per accedere ai caratteri in una stringa. Facciamo esattamente questo per stampare tutti i caratteri di una stringa sulla stessa riga:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Ciao mondo!";

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);
}
```

Tieni presente che stiamo utilizzando l'identificatore di formato `%c` per stampare un singolo carattere.

Inoltre controlla questo. Il programma funzionerà comunque correttamente se modifichiamo la definizione di `s` in modo che sia di tipo `char*`:

```
#include <stdio.h>

int main(void)
{
    char *s = "Ciao mondo!";
// char* qui

    for (int i = 0; i < 13; i++)
        printf("%c\n", s[i]);
// Ma usi ancora gli array qui ...?
}
```

E possiamo ancora usare la notazione di array per portare a termine il lavoro quando lo stampiamo! Ciò è sorprendente ma lo è solo perché non abbiamo ancora parlato dell'equivalenza array/puntatore. Ma questo è ancora un altro indizio del fatto che array e puntatori sono in fondo la stessa cosa.

7.4. Inizializzatori di stringa

Abbiamo già visto alcuni esempi con l'inizializzazione di variabili stringa con valori stringa letterali:

```
char *s = "Ciao mondo!";
char t[] = "Ciao mondo!";
```

Ma questi due sono sottilmente diversi.

Questo è un puntatore a una stringa letterale (ovvero un puntatore al primo carattere in una stringa):

```
char *s = "Ciao mondo!";
```

Se provi a modificare quella stringa con questo:

```
char *s = "Ciao mondo!";

s[0] = 'z';
// CATTIVE NOTIZIE: ho provato a mutare una stringa letterale!
```

Il comportamento non è definito. Probabilmente a seconda del sistema si verificherà un arresto anomalo.

Ma dichiararlo come array è diverso. Questa è una *copia* mutabile della stringa che possiamo modificare a piacimento:

```
char t[] = "Ciao mondo!";
// t è una copia dell'array della stringa
t[0] = 'z';
//  Nessun problema

printf("%s\n", t);
// "ziao mondo!"
```

Quindi ricorda: se hai un puntatore a una stringa letterale non provare a cambiarlo! E se usi una stringa tra virgolette doppie per inizializzare un array in realtà non è una stringa letterale.

7.5. Ottenere la lunghezza della stringa

Non puoi dal momento che C non lo tiene traccia per te. E quando dico “non puoi” in realtà intendo “può”[66] . C'è una funzione in `<string.h>` chiamata `strlen()` che può essere utilizzata per calcolare la lunghezza di qualsiasi stringa in bytes[67] .

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s = "Ciao mondo!";

    printf("La stringa è lunga %zu byte.\n", strlen(s));
}
```

La funzione `strlen()` restituisce il tipo `size_t` che è un tipo intero quindi puoi usarlo per i calcoli con numeri interi. Stampiamo `size_t` con `%zu`.

Il programma sopra stampa:

```
La stringa è lunga 13 byte.
```

Grande! Quindi *è* possibile ottenere la lunghezza della stringa!

Ma… se C non tiene traccia della lunghezza della stringa da nessuna parte come fa a sapere quanto è lunga la stringa?

66 Però è vero che il C non tiene traccia della lunghezza delle stringhe.

67 Se utilizzi il set di caratteri di base o un set di caratteri a 8 bit, sei abituato un carattere è un byte. Questo non è vero per tutte le codifiche dei caratteri, però.

7.6. Terminazione della stringa

Il C gestisce le stringhe in modo leggermente diverso rispetto a molti linguaggi di programmazione e in effetti in modo diverso rispetto a quasi tutti i linguaggi di programmazione moderni.

Quando crei un nuovo linguaggio hai fondamentalmente due opzioni per archiviare una stringa in memoria:

1. Memorizza i byte della stringa insieme a un numero che indica la lunghezza della stringa.
2. Memorizza i byte della stringa e contrassegna la fine della stringa con un byte speciale chiamato *terminatore*.

Se desideri stringhe più lunghe di 255 caratteri l'opzione 1 richiede almeno due byte per memorizzare la lunghezza. Mentre l'opzione 2 richiede solo un byte per terminare la stringa. Quindi un po' di risparmio lì.

Naturalmente al giorno d'oggi sembra ridicolo preoccuparsi di salvare un byte (o 3: molti linguaggi ti permetteranno felicemente di avere stringhe lunghe 4 gigabyte). Ma in passato era un buon affare.

Quindi C ha adottato l'approccio n. 2. In C, una “stringa” è definita da due caratteristiche fondamentali:

- Puntatore al primo carattere nella stringa.
- Un byte con valore zero (o `NUL` carattere[68]) da qualche parte nella memoria dopo il puntatore che indica la fine della stringa.

Un carattere `NUL` può essere scritto nel codice C come `\0`, anche se non è necessario farlo spesso.

Quando includi una stringa tra virgolette doppie nel tuo codice il carattere `NUL` viene automaticamente implicitamente incluso.

```
char *s = "Ciao!";  // In realtà "Ciao!\0" dietro le quinte
```

Quindi con questo in mente scriviamo la nostra funzione `strlen()` che conta i caratteri in una stringa finché non trova un `NUL`.

La procedura consiste nel cercare nella stringa un singolo carattere `NUL` contando man mano che si procede[69] :

```
int my_strlen(char *s)
{
    int count = 0;

   // Virgolette singole per il carattere singolo
   while (s[count] != '\0')
        count++;

    return count;
}
```

E questo è fondamentalmente il modo in cui il `strlen()` integrato porta a termine il lavoro.

7.7. Copia di una stringa

Non è possibile copiare una stringa tramite l'operatore di assegnazione `(=)`. Tutto ciò che fa è

68 Questo è diverso dal puntatore NULL e lo abbrevierò NUL quando parlerò sul carattere rispetto a NULL per il puntatore.

69 Più tardi impareremo un modo più accurato per farlo con l'aritmetica dei puntatori.

creare una copia del puntatore al primo carattere… così ti ritroverai con due puntatori alla stessa stringa:

```c
#include <stdio.h>

int main(void)
{
    char s[] = "Ciao mondo!";
    char *t;

    // Questo crea una copia del puntatore, non una copia della stringa!
    t = s;

    // modifichiamo t
    t[0] = 'z';

    // Ma la stampa mostra la modifica!
    // Perché t e s puntano alla stessa stringa!

    printf("%s\n", s);
// "ziao mondo!"
}
```

Se vuoi fare una copia di una stringa devi copiarla un byte alla volta ma questo è reso più semplice con la funzione `strcpy()` [70] .

Prima di copiare la stringa assicurati di avere spazio in cui copiarla ovvero l'array di destinazione che conterrà i caratteri deve essere lungo almeno quanto la stringa che stai copiando.

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[] = "Ciao mondo!";
    char t[100];
// Ogni carattere è un byte, quindi c'è molto spazio

    // Questo crea una copia della stringa!
    strcpy(t, s);

    // Modifichiamo t
    t[0] = 'z';

    // E s rimane inalterato perché è una stringa diversa
    printf("%s\n", s);  // "ciao mondo!"

    // Ma è stato cambiato
    printf("%s\n", t);  // "ziao mondo!"
}
```

Nota con `strcpy()` il puntatore di destinazione è il primo argomento e il puntatore di origine è il secondo. Un modo che utilizzo per ricordarlo è l'ordine che avresti messo `t` e `s` se una assegnazione `=` funzionasse per le stringhe con l'origine a destra e la destinazione a sinistra.

70 Esiste una funzione più sicura chiamata strncpy() che probabilmente dovresti usare invece, ma di questo ci arriveremo più tardi.

8. Strutture

In C abbiamo qualcosa chiamato `struct` che è un tipo definibile dall'utente che contiene più dati potenzialmente di tipi diversi.

È un modo conveniente per raggruppare più variabili in una sola. Questo può essere utile per passare variabili alle funzioni (quindi devi passarne solo una invece di molte) e utile per organizzare i dati e rendere il codice più leggibile.

Se provieni da un'altro linguaggio potresti avere familiarità con l'idea di *classi* e *oggetti*. Questi non esistono in C nativamente[71] . Puoi pensare a una `struct` come a una classe con solo membri dati e nessun metodo.

8.1. Dichiarazione di una struttura

Puoi dichiarare una `struct` nel tuo codice in questo modo:

```
struct macchina {
    char *nome;
    float prezzo;
    int velocità;
};
```

Questo viene spesso fatto nell'ambito globale al di fuori di qualsiasi funzione in modo che la `struct` sia disponibile a livello globale.

Se lo fai stai creando un nuovo *tipo*. Il nome completo del tipo è `struct macchina`. (Non solo `macchina`: non funzionerà.)

Non esistono ancora variabili di quel tipo ma possiamo dichiararne alcune:

```
struct macchina saturn;
// Variabile "saturn" di tipo "struct macchina"
```

E ora abbiamo una variabile non inizializzata `saturn`[72] di tipo `struct macchina`.

Dovremmo inizializzarlo! Ma come impostiamo i valori di quei singoli campi?

Come in molti altri linguaggi che lo hanno rubato al C utilizzeremo l'operatore punto `(.)` per accedere ai singoli campi.

```
saturn.nome = "Saturn SL/2";
saturn.prezzo = 15999.99;
saturn.velocità = 175;

printf("Nome:           %s\n", saturn.nome);
printf("Prezzo (USD):    %f\n", saturn.prezzo);
printf("Velocità massima (km): %d\n", saturn.velocità);
```

Lì nelle prime righe impostiamo i valori nella `struct macchina` e poi nella parte successiva stampiamo quei valori.

8.2. Inizializzatori di struttura

L'esempio nella sezione precedente era un po' complicato. Deve esserci un modo migliore per

71 Sebbene in C i singoli elementi in memoria come int s siano indicati come “oggetti” non sono oggetti nel senso della programmazione orientata agli oggetti.

72 Fino ad allora la Saturn era una marca popolare di auto economiche negli Stati Uniti è stato chiuso dal crollo del 2008 purtroppo per noi fan.

inizializzare quella variabile `struct`!

Puoi farlo con un inizializzatore inserendo i valori per i campi *nell'ordine in cui appaiono nella* `struct` quando definisci la variabile. (Questo non funzionerà dopo che la variabile è stata definita: deve avvenire nella definizione).

```
struct macchina {
    char *nome;
    float prezzo;
    int velocità;
};

// Ora con un inizializzatore!
// Stesso ordine dei campi come nella dichiarazione della struttura:
struct macchina saturn = {"Saturn SL/2", 16000.99, 175};

printf("Nome:      %s\n", saturn.nome);
printf("Prezzo:     %f\n", saturn.prezzo);
printf("Velocità massima: %d km\n", saturn.velocità);
```

Il fatto che i campi nell'inizializzatore debbano essere nello stesso ordine è un po' strano. Se qualcuno cambiasse l'ordine in `struct macchina` potrebbe rompere tutto l'altro codice!

Possiamo essere più specifici con i nostri inizializzatori:

```
struct macchina saturn = {.velocità=175, .nome="Saturn SL/2"};
```

Ora è indipendente dall'ordine nella dichiarazione `struct`. Che è sicuramente un codice più sicuro.

Similmente agli inizializzatori di array qualsiasi campo mancante nel design viene inizializzato a zero (in questo caso sarebbe `.prezzo` che ho omesso).

8.3. Passaggio di strutture alle funzioni

Puoi fare un paio di cose per passare una `struct` a una funzione.

1. Passa la `struct`.
2. Passa un puntatore alla `struct`.

Ricorda che quando passi qualcosa a una funzione viene creata una *copia* di quella cosa affinché la funzione possa operare sia che si tratti di una copia di un puntatore di un `int` di una struttura o di qualsiasi altra cosa.

Ci sono fondamentalmente due motivi per cui vorresti passare un puntatore alla `struct`:

1. È necessaria la funzione per poter apportare modifiche alla `struct` che è stata passata e visualizzare tali modifiche nel chiamante.
2. La `struct` è piuttosto grande ed è più costoso copiarla nello stack piuttosto che copiare semplicemente un puntatore[73] .

Per questi due motivi è molto più comune passare un puntatore a una `struct` a una funzione sebbene non sia affatto illegale passare la `struct` stessa.

Proviamo a passare un puntatore creando una funzione che ti permetta di impostare il campo `.prezzo` della `struct macchina`:

```
#include <stdio.h>
```

73 Un puntatore è probabilmente lungo 8 byte su un sistema a 64 bit.

```
struct macchina {
    char *nome;
    float prezzo;
    int velocità;
};

int main(void)
{
    struct macchina saturn = {.velocità=175, .nome="Saturn SL/2"};

    // Passa un puntatore a questa struct macchina,
    // insieme a un nuovo prezzo più realistico:
    imposta_prezzo(&saturn, 799.99);

    printf("Prezzo: %f\n", saturn.prezzo);
}
```

Dovresti essere in grado di trovare la dichiarazione della funzione per `imposta_prezzo()` semplicemente osservando i tipi di argomenti che abbiamo lì.

`saturn` è una `struct macchina` quindi `&saturn` deve essere l'indirizzo della `struct macchina` ovvero un puntatore a una `struct macchina` vale a dire una `struct macchina*`.

E `799,99` è un `float`.

Quindi la dichiarazione della funzione deve assomigliare a questa:

```
void imposta_prezzo(struct macchina *c, float nuovo_prezzo)
```

Dobbiamo solo scrivere il corpo. Un tentativo potrebbe essere:

```
void imposta_prezzo(struct macchina *c, float nuovo_prezzo) {
    c.price = nuovo_prezzo;  // ERRORE!!
}
```

Non funzionerà perché l'operatore punto funziona solo su `struct`... non funziona su *puntatori* a `struct`.

Ok quindi possiamo dereferenziare la variabile `c` per de-puntarla per arrivare alla `struct` stessa. Dereferenziare una `struct macchina*` dà come risultato la `struct macchina` a cui punta il puntatore su cui dovremmo essere in grado di utilizzare l'operatore punto:

```
void imposta_prezzo(struct macchina *c, float nuovo_prezzo) {
    (*c).prezzo = nuovo_prezzo;  // Funziona ma è brutto e non idiomatico :(
}
```

E funziona! Ma è un po' complicato digitare tutte quelle parentesi e l'asterisco. C ha uno zucchero sintattico chiamato *operatore freccia* che aiuta in questo.

8.4. L'operatore della freccia

L'operatore freccia aiuta a fare riferimento ai campi nei puntatori a `struct`.

```
void imposta_prezzo(struct macchina *c, float nuovo_prezzo) {
    // (*c).prezzo = nuovo_prezzo;  // Funziona ma è brutto e non idiomatico :(
    //
    // La riga sopra è equivalente al 100% a quella sottostante:

    c->prezzo = nuovo_prezzo;  // È lui!
```

```
}
```

Quindi quando accediamo ai campi? Quando usiamo il punto? E quando usiamo la freccia?

- Se hai una `struct`, usa il punto `(.)`.
- Se hai un puntatore a una `struct`, usa la freccia `(->)`.

8.5. Copia e restituzione di `struct`

Eccone uno facile per te!

Basta assegnare dall'uno all'altro!

```
struct macchina a, b;

b = a;  // copia la struct
```

E anche la restituzione di una `struct` (in contrapposizione a un puntatore a uno) da una funzione crea una copia simile nella variabile ricevente.

Questa non è una "copia profonda"[74] . Tutti i campi vengono copiati così come sono, inclusi i puntatori alle cose.

8.6. Comparare le `struct`

C'è solo un modo sicuro per farlo: confrontare ogni campo uno alla volta.

Potresti pensare di poter usare `memcmp()` [75] ma potrebbero esserci dei byte di riempimento e in quel caso non viene gestito.

Se prima azzeri la `struct` con `memset()` [76] allora potrebbe funzionare anche se potrebbero esserci elementi strani che potrebbero non essere paragonabili a quelli previsti[77] .

9. File Input/Output

Abbiamo già visto alcuni esempi di I/O con `printf()` per eseguire I/O dalla console.

Ma in questo capitolo approfondiremo ulteriormente questi concetti.

9.1. Il tipo di dati `FILE*`

Quando eseguiamo qualsiasi tipo di I/O in C lo facciamo attraverso un pezzo di dati che ottieni sotto forma di un tipo `FILE*`. Questo `FILE*` contiene tutte le informazioni necessarie per comunicare con il sottosistema I/O su quale file hai aperto, dove ti trovi nel file e così via.

Le specifiche si riferiscono a questi come *flussi* ovvero un flusso di dati da un file o da qualsiasi fonte. Utilizzerò "file" e "stream" in modo intercambiabile ma in realtà dovresti pensare a un "file" come a un caso speciale di "stream". Esistono altri modi per trasmettere i dati in un programma oltre alla semplice lettura da un file.

Vedremo tra poco come passare dall'avere un nome file all'ottenere un `FILE*` aperto ma prima voglio menzionare tre flussi che sono già aperti per te e pronti per l'uso.

74 Una copia profonda segue il puntatore nella struttura e copia i dati a cui puntano, come BENE. Una copia superficiale copia semplicemente i puntatori ma non le cose a cui puntano. C non viene fornito con alcuna funzionalità di copia approfondita incorporata.
75 https://beej.us/guide/bgclr/html/split/stringref.html#man-strcmp
76 https://beej.us/guide/bgclr/html/split/stringref.html#man-memset
77 https://stackoverflow.com/questions/141720/how-do-you-compare-structs-for-equality-in-c

Nome `FILE*`	Descrizione
`stdin`	Standard Input, generalmente la tastiera per impostazione predefinita
`stdout`	Standard Output, generalmente lo schermo per impostazione predefinita
`stderr`	Standard Error, generalmente anche lo schermo per impostazione predefinita

In realtà li abbiamo già usati implicitamente a quanto pare. Ad esempio queste due chiamate sono uguali:

```
printf("Ciao mondo!\n");
fprintf(stdout, "Ciao mondo!\n");
// printf in un file
```

Ma ne parleremo più avanti.

Inoltre noterai che sia `stdout` che `stderr` appaiono sullo schermo. Anche se a prima vista sembra una svista o una ridondanza in realtà non lo è. I sistemi operativi tipici consentono di *reindirizzare* l'output di uno di questi in file diversi e può essere conveniente poter separare i messaggi di errore dal normale output non di errore. Ad esempio in una shell POSIX (come sh, ksh, bash, zsh, ecc.) su un sistema simile a Unix, potremmo eseguire un programma e reindirizzare solo l'output non di errore (`stdout`) a un file, e tutto l'output dell'errore (`stderr`) in un altro file.

```
./foo > output.txt 2> errori.txt   # Questo comando è specifico per Unix
```

Per questo motivo dovresti inviare messaggi di errore gravi a `stderr` invece che a `stdout`.

Daremo maggiori informazioni su come farlo più tardi.

9.2. Lettura di file di testo

I flussi sono in gran parte classificati in due modi diversi: *testo* e *binario*.

Ai flussi di testo è consentito tradurre in modo significativo i dati, in particolare tradurre i fine riga nelle loro diverse rappresentazioni[78] . I file di testo sono logicamente una sequenza di *righe* separate da ritorni a capo. Per essere portabili i dati di input dovrebbero sempre terminare con un ritorno a capo.

Ma la regola generale è che se riesci a modificare il file in un normale editor di testo, si tratta di un file di testo. Altrimenti è binario. Per avere maggiori informazioni sui binario li troverai più in avanti.

Allora mettiamoci al lavoro: come possiamo aprire un file per la lettura ed estrarne i dati?

Creiamo un file chiamato `ciao.txt` che contiene questo:

```
Ciao mondo!
```

78 Avevamo tre diversi ritorni a capo con effetto generale: Carriage Return (CR, utilizzato sui vecchi Mac), Linefeed (LF, utilizzato sui sistemi Unix) e Carriage Return/Linefeed (CRLF, utilizzato sui sistemi Windows). Per fortuna il l'introduzione di OS X, essendo basato su Unix ha ridotto questo numero a due.

E scriviamo un programma per aprire il file leggerne un carattere e poi chiudere il file quando abbiamo finito. Questo è il piano del gioco!

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
// Variabile per rappresentare il file aperto

   fp = fopen("ciao.txt", "r");
// Apri il file per la lettura

   int c = fgetc(fp);
// Legge un singolo carattere
   printf("%c\n", c);
// Stampa il carattere su stdout

   fclose(fp);
// Al termine, chiudere il file
}
```

Guarda come quando abbiamo aperto il file con `fopen()`, ci ha restituito il `FILE*` in modo da poterlo utilizzare in seguito.

(Lo tralascio per brevità ma `fopen()` restituirà `NULL` se qualcosa va storto risulterà come file non trovato quindi dovresti controllare gli errori!)

Nota anche la "`r`" che abbiamo inserito: significa "apri uno stream di testo per la lettura". (Esistono varie stringhe che possiamo passare a `fopen()` con significati aggiuntivi come scrivere, aggiungere e così via.)

Successivamente abbiamo utilizzato la funzione `fgetc()` per ottenere un carattere dallo stream. Forse ti starai chiedendo perché ho creato `c` un `int` invece di un `char`: tienilo a mente!

Infine chiudiamo lo stream quando abbiamo finito. Tutti i flussi vengono chiusi automaticamente all'uscita del programma ma è buona norma e buona pulizia chiudere esplicitamente tutti i file una volta terminati.

Il `FILE*` tiene traccia della nostra posizione nel file. Quindi le chiamate successive a `fgetc()` otterrebbero il carattere successivo nel file, e poi quello successivo, fino alla fine.

Ma sembra una seccatura. Vediamo se riusciamo a renderlo più semplice.

9.3. End of File: EOF

Esiste un carattere speciale definito come macro: `EOF`. Questo è ciò che `fgetc()` ti restituirà quando viene raggiunta la fine del file e hai tentato di leggere un altro carattere.

Che ne dici di condividere questo Fatto Divertente adesso. Risulta che `EOF` è il motivo per cui `fgetc()` e funzioni simili restituiscono un `int` invece di un `char`. `EOF` non è un carattere vero e proprio e il suo valore probabilmente non rientra nell'intervallo di `char`. Poiché `fgetc()` deve essere in grado di restituire qualsiasi byte **e** `EOF`, deve essere un tipo più ampio che possa contenere più valori. quindi `int` lo è. Ma a meno che non confronti il valore restituito con `EOF` puoi sapere in fondo che è un `char`.

Va bene! Torna alla realtà! Possiamo usarlo per leggere l'intero file in un ciclo.

```
#include <stdio.h>
```

```
int main(void)
{
   FILE *fp;
   int c;

   fp = fopen("ciao.txt", "r");

   while ((c = fgetc(fp)) != EOF)
       printf("%c", c);

   fclose(fp);
}
```

(Se la riga 10 è troppo strana scomponila semplicemente iniziando dalle parentesi annidate più interne. La prima cosa che facciamo è assegnare il risultato di `fgetc()` in `c` e *poi* confrontarlo *con* `EOF`. L'abbiamo appena stipato in un'unica riga. Potrebbe sembrare difficile da leggere, ma studialo: C è idiomatico.) Ed eseguendo questo, vediamo:

```
Ciao mondo!
```

Tuttavia stiamo operando un carattere alla volta e molti file di testo hanno più senso a livello di linea. Passiamo a quello.

9.3.1. Leggere una riga alla volta

Allora come possiamo ottenere un'intera linea in una sola volta? `fgets()` in soccorso! Per gli argomenti è necessario un puntatore a un buffer di caratteri per contenere i byte, un numero massimo di byte da leggere e un `FILE*` da cui leggere. Restituisce `NULL` alla fine del file o all'errore. `fgets()` è anche abbastanza carino da terminare la stringa con NULL una volta finito[79]
.

Facciamo un ciclo simile a quello precedente, tranne che abbiamo un file multilinea e leggiamolo una riga alla volta.

Ecco un file quote.txt:

```
A wise man can learn more from
a foolish question than a fool
can learn from a wise answer.
                --Bruce Lee
```

Ed ecco del codice che legge il file una riga alla volta e stampa un numero di riga prima di ognuno di essi:

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char s[1024];
// Abbastanza grande per qualsiasi riga incontrerà questo programma
   int linecount = 0;

   fp = fopen("quote.txt", "r");

   while (fgets(s, sizeof s, fp) != NULL)
```

79 Se il buffer non è abbastanza grande per leggere un'intera riga, smetterà di leggere mid-line e la successiva chiamata a fgets() continuerà a leggere il resto del file linea.

```
       printf("%d: %s", ++linecount, s);

   fclose(fp);
}
```

Che ci dà l'output:

```
1: A wise man can learn more from
2: a foolish question than a fool
3: can learn from a wise answer.
4:                   --Bruce Lee
```

9.4. Formatted Input

Sai come ottenere un output formattato con `printf()` (e, quindi, `fprintf()` come vedremo di seguito)?

Puoi fare la stessa cosa con `fscanf()`.

> Prima di iniziare tieni presente che l'utilizzo di funzioni in stile `scanf()` può essere pericoloso con input non attendibili. Se non specifichi la larghezza del campo con il tuo `%s`, potresti sovraccaricare il buffer. Peggio ancora, la conversione numerica non valida determina un comportamento indefinito. La cosa sicura da fare con input non attendibili è utilizzare `%s` con una larghezza di campo quindi utilizzare funzioni come `strtol()` o `strtod()` per eseguire le conversioni.

Prendiamo un file con una serie di record di dati al suo interno. In questo caso, balene, con nome, lunghezza in metri e peso in tonnellate. `balene.txt`:

```
blue 29.9 173
right 20.7 135
gray 14.9 41
humpback 16.0 30
```

Sì, potremmo leggerli con `fgets()` e poi analizzare la stringa con `sscanf()` (e in questo è più resistente ai file danneggiati), ma in questo caso usiamo semplicemente `fscanf()` ed estraiamolo direttamente.

La funzione `fscanf()` salta gli spazi bianchi iniziali durante la lettura e restituisce `EOF` alla fine del file o in caso di errore.

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char name[1024];
// Abbastanza grande per qualsiasi riga incontrerà questo programma
   float length;
   int mass;

   fp = fopen("whales.txt", "r");

   while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
       printf("%s whale, %d tonnes, %.1f meters\n", name, mass, length);

   fclose(fp);
}
```

che ci dà questo risultato:

```
blue whale, 173 tonnes, 29.9 meters
right whale, 135 tonnes, 20.7 meters
gray whale, 41 tonnes, 14.9 meters
humpback whale, 30 tonnes, 16.0 meters
```

9.5. Scrittura di file di testo

Allo stesso modo possiamo usare `fgetc()`, `fgets()` e `fscanf()` per leggere flussi di testo, possiamo usare `fputc()`, `fputs()` e `fprintf( )` per scrivere flussi di testo.

Per fare ciò, dobbiamo `fopen()` il file in modalità scrittura passando `"w"` come secondo argomento. L'apertura di un file esistente in modalità `"w"` troncherà immediatamente il file a 0 byte per una sovrascrittura completa.

Metteremo insieme un semplice programma che genera un file `output.txt` utilizzando una varietà di funzioni di output.

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   int x = 32;

   fp = fopen("output.txt", "w");

   fputc('B', fp);
   fputc('\n', fp);   // newline
   fprintf(fp, "x = %d\n", x);
   fputs("Hello, world!\n", fp);

   fclose(fp);
}
```

And this produces a file, output.txt, with these contents:

```
B
x = 32
Hello, world!
```

Curisità: poiché `stdout` è un file, potresti sostituire la riga 8 con:

```
fp = stdout;
```

e il programma verrebbe visualizzato sulla console anziché su un file. Provalo!

9.6. File binario I/O

Finora abbiamo parlato solo di file di testo. Ma c'è quell'altra bestia di cui abbiamo parlato all'inizio chiamata file *binari* o flussi binari.

Funzionano in modo molto simile ai file di testo, tranne che il sottosistema I/O non esegue alcuna traduzione sui dati come potrebbe fare con un file di testo. Con i file binari ottieni un flusso grezzo di byte e questo è tutto.

La grande differenza nell'aprire il file è che devi aggiungere `"b"` alla modalità. Cioè per leggere un file binario aprilo in modalità `"rb"`. Per scrivere un file, aprilo in modalità `"wb"`.

Poiché si tratta di flussi di byte e i flussi di byte possono contenere caratteri NUL e il carattere NUL

è l'indicatore di fine stringa in C è raro che le persone utilizzino le funzioni `fprintf()`-e-amici per operare su binari File.

Invece le funzioni più comuni sono `fread()` e `fwrite()`. Le funzioni leggono e scrivono un numero specificato di byte nel flusso.

Per fare una dimostrazione, scriveremo un paio di programmi. Scriverà una sequenza di valori di byte su tutto il disco in una volta. E il secondo programma leggerà un byte alla volta e li stamperà[80] .

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   unsigned char bytes[6] = {5, 37, 0, 88, 255, 12};

   fp = fopen("output.bin", "wb");  // wb mode per "scrivere binario"!

   // Nella chiamata a fwrite, gli argomenti sono:
   //
   // * Puntatore ai dati da scrivere
   // * Dimensione di ogni "pezzo" di dati
   // * Conteggio di ogni "pezzo" di dati
   // * FILE*

   fwrite(bytes, sizeof(char), 6, fp);

   fclose(fp);
}
```

Quei due argomenti centrali di `fwrite()` sono piuttosto strani. Ma fondamentalmente quello che vogliamo dire alla funzione è: "Abbiamo articoli *così* grandi e vogliamo scriverne *così* tanti". Ciò lo rende conveniente se hai un record di lunghezza fissa e ne hai molti in un array. Puoi semplicemente dirgli la dimensione di un record e quanti scriverne.

Nell'esempio sopra diciamo che ogni record ha la dimensione di un `char` e ne abbiamo 6.

L'esecuzione del programma ci fornisce un file `output.bin` ma aprirlo in un editor di testo non mostra nulla di amichevole! Sono dati binari e non testo. E i dati binari casuali che ho appena inventato!

Se lo eseguo tramite un programma hex dump[81] , possiamo vedere l'output come byte:

```
05 25 00 58 ff 0c
```

E quei valori in esadecimale corrispondono ai valori (in decimale) che abbiamo scritto.

Ma ora proviamo a rileggerli con un programma diverso. Questo aprirà il file per la lettura binaria (modalità `"rb"`) e leggerà i byte uno alla volta in un ciclo.

`fread()` ha la caratteristica interessante di restituire il numero di byte letti o `0` su EOF. Quindi possiamo ripetere il ciclo finché non lo vediamo stampando i numeri mentre procediamo.

```
#include <stdio.h>

int main(void)
{
```

80 Normalmente il secondo programma leggerebbe tutti i byte contemporaneamente e poi stamperebbe li fuori in un ciclo. Sarebbe più efficiente. Ma stiamo andando per la demo valore qui.

81 https://en.wikipedia.org/wiki/Hex_dump

```
   FILE *fp;
   unsigned char c;

   fp = fopen("output.bin", "rb"); // rb per "read binary"!

   while (fread(&c, sizeof(char), 1, fp) > 0)
       printf("%d\n", c);

   fclose(fp);
}
```

E eseguendolo vediamo i nostri numeri originali!

```
5
37
0
88
255
12
```

Woo hoo!

9.6.1. `struct` e avvertenze sui numeri

Come abbiamo visto nella sezione `struct` il compilatore è libero di aggiungere il riempimento a una `struct` come ritiene opportuno. E diversi compilatori potrebbero farlo diversamente. E lo stesso compilatore su architetture diverse potrebbe farlo diversamente. E lo stesso compilatore sulle stesse architetture potrebbe farlo diversamente.

Ciò a cui voglio arrivare è questo: non è portabile semplicemente `fwrite()` un'intera `struct` in un file quando non si sa dove andrà a finire il riempimento.

come lo aggiustiamo? Tieni questo pensiero: esamineremo alcuni modi per farlo dopo aver esaminato un altro problema correlato.

Numeri! Risulta che tutte le architetture non rappresentano i numeri in memoria allo stesso modo.

Diamo un'occhiata a un semplice `fwrite()` di un numero di 2 byte. Lo scriveremo in esadecimale in modo che ogni byte sia chiaro. Il byte più significativo avrà il valore `0x12` e il meno significativo avrà il valore `0x34`.

```
unsigned short v = 0x1234;
// Due byte, 0x12 e 0x34

fwrite(&v, sizeof v, 1, fp);
```

Cosa finisce nel flusso?

Beh, sembra che dovrebbe essere `0x12` seguito da `0x34` giusto?

Ma se lo eseguo sul mio computer e faccio il hex dump del risultato ottengo:

```
34 12
```

Sono invertiti! Cosa dà?

Questo ha qualcosa a che fare con quella che viene chiamata l'endianess[82] dell'architettura. Alcuni scrivono prima i byte più significativi altri quelli meno significativi.

Ciò significa che se si scrive un numero multibyte direttamente dalla memoria non è possibile farlo in modo portabile[83] .

82 https://en.wikipedia.org/wiki/Endianess

Un problema simile esiste con la virgola mobile. La maggior parte dei sistemi utilizza lo stesso formato per i numeri in virgola mobile ma alcuni no. Nessuna garanzia!

Quindi… come possiamo risolvere tutti questi problemi con i numeri e le `struct` per scrivere i nostri dati in modo portabile?

La sintesi è *serializzare* i dati, che è un termine generale che significa prendere tutti i dati e scriverli in un formato che controlli che sia ben noto e programmabile per funzionare allo stesso modo su tutte le piattaforme.

Come puoi immaginare questo è un problema risolto. Esistono numerose librerie di serializzazione di cui puoi trarre vantaggio, come i *buffer di protocollo* di Google[84] , disponibili e pronte all'uso. Si prenderanno cura di tutti i dettagli più importanti per te e consentiranno persino ai dati dei tuoi programmi C di interagire con altri linguaggi che supportano gli stessi metodi di serializzazione. Fate un favore a voi stessi e a tutti!

Serializza i tuoi dati binari quando li scrivi in un flusso! Ciò manterrà le cose belle e portatili, anche se trasferisci file di dati da un'architettura a un'altra.

10. `typedef`: Creare nuovi tipi

Beh, non tanto creare *nuovi* tipi quanto ottenere nuovi nomi per i tipi esistenti. Sembra un po' inutile in superficie, ma possiamo davvero usarlo per rendere il nostro codice più pulito.

10.1. `typedef` in teoria

Fondamentalmente prendi un tipo esistente e ne crei un alias con `typedef`.

Come questo:

```
typedef int antilope;
// Rendi "antilope" un alias per "int"

antilope x = 10;
// Il tipo "antilope" è uguale al tipo "int"
```

Puoi prendere qualsiasi tipo esistente e farlo. Puoi anche creare diversi tipi con un elenco di virgole:

```
typedef int antilope, tarallo, fungo;
// Questi sono tutti "int"
```

È davvero utile vero? Che puoi digitare `fungo` invece di `int`? Devi essere *super entusiasta* di questa funzionalità!

OK, professor Sarcasmo, tra un attimo parleremo di alcune applicazioni più comuni di questo.

10.1.1. Ambito

`typedef` segue le normali regole dell'ambito.

Per questo motivo è abbastanza comune trovare `typedef` nell'ambito del file (“globale”) in modo che tutte le funzioni possano utilizzare i nuovi tipi a piacimento.

10.2. `typedef` in pratica

Quindi rinominare `int` in qualcos'altro non è così entusiasmante. Vediamo dove `typedef` fa comunemente la sua comparsa.

84 https://en.wikipedia.org/wiki/Protocol_buffers

10.2.1. `typedef` e `struct`

A volte una `struct` verrà `typedef` con un nuovo nome in modo da non dover digitare la parola `struct` più e più volte.

```
struct animal {
    char *nome;
    int gambe_contatore, velocità;
};

//  nome originale     nuovo nome
//           |          |
//           v          v
//      |-----------| |----|
typedef struct animal animal;

struct animal y;
// Questo funziona
animal z;
// Funziona anche questo perché "animal" è un alias
```

Personalmente non mi interessa questa pratica. Mi piace la chiarezza che ha il codice quando aggiungi la parola `struct` al tipo; i programmatori sanno cosa stanno ottenendo. Ma è davvero comune, quindi lo includo qui.

Ora voglio eseguire esattamente lo stesso esempio in un modo che potresti vedere comunemente. Inseriremo la `struct animal` nella `typedef`. Puoi schiacciare il tutto in questo modo:

```
//  nome originale
//           |
//           v
//      |-----------|
typedef struct animal {
    char *name;
    int leg_count, speed;
} animal;                          // <-- nuovo nome

struct animal y;
// Questo funziona
animal z;
// Funziona anche questo perché "animal" è un alias
```

È esattamente lo stesso dell'esempio precedente solo più conciso.

Ma non è tutto! Esiste un'altra scorciatoia comune che potresti vedere nel codice utilizzando quelle che vengono chiamate *strutture anonime*[85] . Si scopre che in realtà non è necessario nominare la struttura in posti differenti, e con `typedef` si può fare.

Facciamo lo stesso esempio con una struttura anonima:

```
//  Struttura anonima! Non ha nome!
//         |
//         v
//      |----|
typedef struct {
    char *nome;
    int gambe_contatore, velocità;
} animal;                          // <-- nuovo nome
```

85 Ne parleremo più avanti.

```
//struct animal y;
// ERRORE: questo non funziona più: nessuna struttura del genere!
animal z;
// Questo funziona perché "animal" è un alias
```

Come altro esempio potremmo trovare qualcosa del genere:

```
typedef struct {
    int x, y;
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 40
```

10.2.2. `typedef` e altri tipi

Non è che usare `typedef` con un tipo semplice come `int` sia completamente inutile… ti aiuta ad astrarre i tipi per rendere più semplice modificarli in seguito.

Ad esempio se hai il `float` su tutto il codice in 100 milioni di posti sarà doloroso cambiarli tutti in `double` se scopri di doverlo fare in seguito per qualche motivo.

Ma se ti preparassi un po' con:

```
typedef float app_float;

// e

app_float f1, f2, f3;
```

Quindi se in seguito desideri passare a un altro tipo come `long double` devi solo modificare il `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// e non è necessario modificare questa riga:

app_float f1, f2, f3;  // Ora questi sono tutti long double
```

10.2.3. `typedef` e puntatori

Puoi creare un tipo che sia un puntatore.

```
typedef int *intptr;

int a = 10;
intptr x = &a;  // "intptr" è di tipo "int*"
```

Onestamente non mi piace questa pratica. Nasconde il fatto che `x` è un tipo puntatore perché non vedi `*` nella dichiarazione.

IMHO è meglio mostrare esplicitamente che stai dichiarando un tipo di puntatore in modo che altri sviluppatori possano vederlo chiaramente e non confondere `x` con un tipo non puntatore.

Ma all'ultimo conteggio circa 832.007 persone avevano un'opinione diversa.

10.2.4. `typedef` e Capitalizzazione

Ho visto tutti i tipi di maiuscole su `typedef`.

```
typedef struct {
    int x, y;
} my_point;
// lower snake case

typedef struct {
    int x, y;
} MyPoint;
// CamelCase

typedef struct {
    int x, y;
} Mypoint;
// Leading uppercase

typedef struct {
    int x, y;
} MY_POINT;
// UPPER SNAKE CASE
```

La specifica C11 non impone un modo o l'altro e mostra esempi tutti in maiuscolo e tutti in minuscolo.

K&R2 utilizza prevalentemente lettere maiuscole ma mostra alcuni esempi in maiuscolo e serpente (con `_t`).

Se hai una guida di stile in uso attieniti ad essa. Se non lo fai prendine uno e mantienilo.

10.3. Array e `typedef`

La sintassi è un po' strana e questo si vede raramente nella mia esperienza ma puoi `typedef` un array di un certo numero di elementi.

```
// Rendi type five_ints un array di 5 int
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

Non mi piace perché nasconde la natura di array della variabile ma è possibile farlo.

11. Puntatori II: Aritmetica

È ora di approfondire l'argomento con una serie di nuovi argomenti sui puntamento! Se non sei aggiornato con i puntatori, consulta la prima sezione della guida sull'argomento.

11.1. Aritmetica dei puntatori

Vediamo insieme che puoi fare calcoli sui puntatori in particolare addizioni e sottrazioni.

Ma cosa significa quando lo fai?

In breve se si dispone di un puntatore a un tipo aggiungendone uno al puntatore si passa all'elemento successivo di quel tipo direttamente dopo in memoria.

È **importante** ricordare che quando spostiamo i puntatori e ci muoviamo in diversi luoghi della

memoria, dobbiamo assicurarci di puntare sempre a un luogo valido della memoria prima di dereferenziare. Se ci allontaniamo tra le erbacce e proviamo a vedere cosa c'è, il comportamento è indefinito e un incidente è un risultato comune.

Questo è un po' un po' come l'uovo e la gallina con l'equivalenza array/puntatore, qui sotto, ma ci proveremo comunque.

11.1.1. Somma con i puntatori

Per prima cosa, prendiamo una serie di numeri.

```
int a[5] = {11, 22, 33, 44, 55};
```

Quindi otteniamo un puntatore al primo elemento dell'array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];
// O "int *p = a;" funziona altrettanto bene
```

Quindi stampiamo il valore lì dereferenziando il puntatore:

```
printf("%d\n", *p);
// Stampa 11
```

Ora usiamo l'aritmetica dei puntatori per stampare l'elemento successivo nell'array quello all'indice 1:

```
printf("%d\n", *(p + 1));
// Stampa 22!!
```

Cosa è successo? C sa che `p` è un puntatore a un `int`. Quindi conosce il `sizeof` di un `int`[86] e sa saltare tanti byte per arrivare all' `int` successivo dopo il primo!

In effetti l'esempio precedente potrebbe essere scritto in questi due modi equivalenti:

```
printf("%d\n", *p);
// Stampa 11
printf("%d\n", *(p + 0));
// Stampa 11
```

perché aggiungendo 0 a un puntatore si ottiene lo stesso puntatore.

Pensiamo al risultato qui. Possiamo scorrere gli elementi di un array in questo modo invece di utilizzare un array:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // O "int *p = a;" funziona altrettanto bene

for (int i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));
// Uquivalente a p[i]!
}
```

E funziona come se usassimo la notazione di array! Oooo! Ci avviciniamo a quella storia dell'equivalenza array/puntatore! Avremo maggiori informazioni su questo argomento più avanti in questo capitolo.

Ma cosa sta succedendo realmente qui? Come funziona?

86 Ricorda che l'operatore `sizeof` ti dice la dimensione in byte di un oggetto in memoria.

Ricorda fin dall'inizio che la memoria è come un grande array, in cui un byte è archiviato in ciascun indice dell'array?

E l'indice dell'array in memoria ha alcuni nomi:

- Indice nella memoria
- Posizione
- Indirizzo
- *Puntatore!*

Quindi un punto è un indice nella memoria da qualche parte.

Per un esempio casuale supponiamo che un numero 3490 sia stato memorizzato all'indirizzo ("indice") 23.237.489.202. Se abbiamo un puntatore `int` a quel 3490, il valore di quel puntatore è 23.237.489.202… perché il puntatore è l'indirizzo di memoria. Parole diverse per la stessa cosa.

E ora diciamo di avere un altro numero, 4096, memorizzato subito dopo il 3490 all'indirizzo 23.237.489.210 (8 più alto del 3490 perché ogni `int` in questo esempio è lungo 8 byte).

Se aggiungiamo `1` a quel puntatore in realtà salta avanti `sizeof(int)` byte all'`int` successivo. Sa saltare così lontano perché è un puntatore a `int`. Se fosse un puntatore `float` salterebbe avanti `sizeof(float)` byte per arrivare al `float` successivo!

Puoi vedere nell' `int` successivo aggiungendo `1` al puntatore, quello successivo aggiungendo `2` al puntatore e così via.

11.1.2. Cambiando i Puntatori

Abbiamo visto come aggiungere un numero intero a un puntatore nella sezione precedente. Questa volta *modifichiamo il puntatore stesso*.

Puoi semplicemente aggiungere (o sottrarre) valori interi direttamente a (o da) qualsiasi puntatore!

Facciamo di nuovo l'esempio, con un paio di modifiche. Per prima cosa aggiungerò un `999` alla fine dei nostri numeri che servirà da valore sentinella. Questo ci farà sapere dove si trova la fine dei dati.

```
int a[] = {11, 22, 33, 44, 55, 999};
// Add 999 here as a sentinel

int *p = &a[0];
// p points to the 11
```

E abbiamo anche `p` che punta all'elemento con indice `0` di `a` cioè `11`, proprio come prima.

Ora iniziamo a *incrementare* `p` in modo che punti agli elementi successivi dell'array. Lo faremo finché p non indicherà `999`; cioè lo faremo fino a `*p == 999`:

```
while (*p != 999) {
// Mentre la cosa a cui punta p non è 999
    printf("%d\n", *p);
// Lo stampa
    p++;
// Muovi p per puntare al successivo int!
}
```

Pazzesco vero?

Quando lo proviamo prima `p` punta a `11`. Poi incrementiamo `p`, e punta a `22` e poi di nuovo punta a `33`. E così via, finché non punta a `999` e usciamo.

11.1.3. Sottrazione con i puntatori

Puoi anche sottrarre un valore da un puntatore per ottenere l'indirizzo precedente proprio come li stavamo aggiungendo prima.

Ma possiamo anche sottrarre due puntatori per trovare la differenza tra loro ad esempio possiamo calcolare quanti `int` ci sono tra due `int*`. Il problema è che funziona solo all'interno di un singolo array[87] : se i puntatori puntano a qualcos'altro si ottiene un comportamento indefinito.

Ti ricordi come le stringhe sono `char*` in C? Vediamo se possiamo usarlo per scrivere un'altra variante di `strlen()` per calcolare la lunghezza di una stringa che utilizza la sottrazione del puntatore.

L'idea è che se abbiamo un puntatore all'inizio della stringa possiamo trovare un puntatore alla fine della stringa cercando in anticipo il carattere `NUL`.

E se abbiamo un puntatore all'inizio della stringa e calcoliamo il puntatore alla fine della stringa, possiamo semplicemente sottrarre i due puntatori per ottenere la lunghezza!

```
#include <stdio.h>

int my_strlen(char *s)
{
    // Inizia la scansione dall'inizio della stringa
    char *p = s;

    // Esegue la scansione finché non troviamo il carattere NUL
    while (*p != '\0')
        p++;

    // Restituisce la differenza nei puntatori
    return p - s;
}

int main(void)
{
    printf("%d\n", my_strlen("Ciao mondo!"));
// Prints "11"
}
```

Ricorda che puoi utilizzare la sottrazione di puntatori solo tra due puntatori che puntano allo stesso array!

11.2. Equivalenza array/puntatore

Siamo finalmente pronti a parlarne! Abbiamo visto molti esempi di luoghi in cui abbiamo mescolato la notazione di array ma diamo la formula *fondamentale dell'equivalenza di array/puntatore:*

```
a[b] == *(a + b)
```

Studialo! Quelli sono equivalenti e possono essere usati in modo intercambiabile!

Ho semplificato un po' troppo, perché nel mio esempio sopra `a` e `b` possono essere entrambe espressioni e potremmo aver bisogno di qualche parentesi in più per forzare l'ordine delle operazioni nel caso in cui le espressioni siano complesse.

87 Oppure string che in realtà è un array di `char`. Un po' stranamente puoi anche avere un puntatore che fa riferimento a uno oltre la fine dell'array senza problemi e continuare a fare calcoli su di esso. Non puoi semplicemente dereferenziarlo quando è là fuori.

Le spec sono specifiche, come sempre, dichiarando (in C11 §6.5.2.1¶2):

> `E1[E2]` è identico a `(*((E1)+(E2)))`

ma è un po' più difficile da capire. Assicurati solo di includere le parentesi se le espressioni sono complicate in modo che tutti i tuoi calcoli avvengano nell'ordine giusto.

Ciò significa che possiamo *decidere* se utilizzare la notazione di array o puntatore per qualsiasi array o puntatore (assumendo che punti a un elemento di un array).

Usiamo un array e un puntatore con la notazione sia dell'array che del puntatore:

```
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};

    int *p = a;
// p punta al primo elemento di a, 11

    // Stampa tutti gli elementi dell'array in vari modi:

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);
// Notazione di array con a

    for (int i = 0; i < 5; i++)
        printf("%d\n", p[i]);
// Notazione degli array con p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(a + i));
// Notazione del puntatore con a

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p + i));
// Notazione del puntatore con p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p++));
// Muovere il puntatore p
        //printf("%d\n", *(a++));    // Muovere l'array variabile a--ERROR!
}
```

Quindi puoi vedere che in generale se hai una variabile di array puoi usare il puntatore o la nozione di array per accedere agli elementi. Lo stesso con una variabile puntatore.

L'unica grande differenza è che puoi modificare un puntatore in modo che punti a un indirizzo diverso ma non puoi farlo con una variabile di array.

11.2.1. Equivalenza di array/puntatori nelle chiamate di funzioni

Questo è sicuramente il luogo in cui incontrerai di più questo concetto.

Se hai una funzione che accetta un argomento puntatore ad esempio:

```
int my_strlen(char *s)
```

questo significa che puoi passare un array o un puntatore a questa funzione e farlo funzionare!

```
char s[] = "Antilope";
char *t = "Vombatidi";

printf("%d\n", my_strlen(s));
// Funziona!
printf("%d\n", my_strlen(t));
// Funziona anche lui!
```

Ed è anche il motivo per cui queste due ambiti di funzione sono equivalenti:

```
int my_strlen(char *s)
// Funziona!
int my_strlen(char s[])
// Funziona anche lui!
```

11.3. Puntatori `void`

Hai già visto la parola chiave `void` utilizzata con le funzioni ma questa è tutt'altro, un animale non correlato.

A volte è utile avere un puntatore a qualcosa di *cui non si conosce il tipo*.

Lo so. Abbi pazienza solo un secondo.

Ci sono fondamentalmente due casi d'uso per questo.

1. Una funzione opererà su qualcosa byte per byte. Ad esempio `memcpy()` copia byte di memoria da un puntatore a un altro, ma questi puntatori possono puntare a qualsiasi tipo. `memcpy()` sfrutta il fatto che se si esegue l'iterazione su `char*`, si esegue l'iterazione sui byte di un oggetto indipendentemente dal tipo dell'oggetto. Maggiori informazioni su questo nella sottosezione Valori multibyte.

1. Un'altra funzione sta chiamando una funzione che le hai passato (un callback) e ti sta passando i dati. Conosci il tipo di dati ma la funzione che ti chiama no. Quindi ti passa dei vuoti perché non conosce il tipo e li converti nel tipo che ti serve. Le funzioni integrate `qsort()`[88] e `bsearch()`[89] utilizzano questa tecnica.

Diamo un'occhiata ad un esempio, la funzione integrata `memcpy()`:

```
void *memcpy(void *s1, void *s2, size_t n);
```

Questa funzione copia `n` byte di memoria a partire dall'indirizzo `s2` nella memoria che inizia all'indirizzo `s1`.

Osserva! `s1` e `s2` sono `void*`! Perché? Cosa significa? Facciamo più esempi per vedere.

Ad esempio, potremmo copiare una stringa con `memcpy()` (sebbene `strcpy()` sia più appropriato per le stringhe):

```
#include <stdio.h>
#include <string.h>

int main(void)
{
```

88 https://beej.us/guide/bgclr/html/split/stdlib.html#man-qsort
89 https://beej.us/guide/bgclr/html/split/stdlib.html#man-bsearch

```
    char s[] = "Capre!";
    char t[100];

    memcpy(t, s, 7);
// Copia 7 byte, incluso il terminatore NUL!

    printf("%s\n", t);
// "Capre!"
}
```

Oppure possiamo copiare alcuni `int`:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int a[] = {11, 22, 33};
    int b[3];

    memcpy(b, a, 3 * sizeof(int));
// Copia 3 int di dati

    printf("%d\n", b[1]);  // 22
}
```

Questo è un po' strano: vedi cosa abbiamo fatto con `memcpy()`? Abbiamo copiato i dati da `a` a `b` ma dovevamo specificare quanti *byte* copiare e un `int` è più di un byte.

OK, allora: quanti byte richiede un `int`? Risposta: dipende dal sistema. Ma possiamo dire quanti byte prende ogni tipo con l'operatore `sizeof`.

Quindi c'è la risposta: un `int` richiede `sizeof(int)` byte di memoria da archiviare.

E se ne abbiamo 3 nel nostro array, come abbiamo fatto in quell'esempio, l'intero spazio utilizzato per i 3 `int` deve essere `3 * sizeof(int)`.

(Nell'esempio della stringa precedente, sarebbe stato tecnicamente più accurato copiare `7 * sizeof(char)` byte. Ma i caratteri sono sempre grandi un byte per definizione, quindi si riduce a `7 * 1`.)

Potremmo anche copiare un `float` o una `struct` con `memcpy()`! (Anche se questo è un abuso —dovremmo semplicemente usare `=` per quello):

```
struct antilope mio_antilope;
struct antilope mio_clone_antilope;

// ...

memcpy(&mio_clone_antilope, &mio_antilope, sizeof mio_antilope);
```

Guarda quanto è versatile `memcpy()`! Se hai un puntatore a un'origine e un puntatore a una destinazione e hai il numero di byte che desideri copiare puoi copiare *qualsiasi tipo di dati*.

Immagina se non avessimo il `void*`. Dovremmo scrivere funzioni `memcpy()` specializzate per ogni tipo:

```
memcpy_int(int *a, int *b, int count);
memcpy_float(float *a, float *b, int count);
memcpy_double(double *a, double *b, int count);
memcpy_char(char *a, char *b, int count);
```

```
memcpy_unsigned_char(unsigned char *a, unsigned char *b, int count);

// etc... blech!
```

Molto meglio usare semplicemente `void*` e avere una funzione che possa fare tutto.

Questo è il potere del `void*`. Puoi scrivere funzioni a cui non interessa il tipo ed è comunque in grado di farci cose.

Ma da un grande potere derivano grandi responsabilità. Forse non *eccezionale* in questo caso, ma ci sono alcuni limiti.

1. Non puoi eseguire l'aritmetica dei puntatori su un `void*`. 2. Non è possibile dereferenziare a `void*`. 3. Non è possibile utilizzare l'operatore freccia su a `void*` poiché è anche un dereferenziamento. 4. Non è possibile utilizzare la notazione di array su un `void*`, poiché è anche una dereferenziazione[90] .

E se ci pensi, queste regole hanno un senso. Tutte queste operazioni si basano sulla conoscenza di `sizeof` il tipo di dati puntati e con `void*`, non conosciamo la dimensione dei dati a cui si fa riferimento: potrebbe essere qualsiasi cosa!

Ma aspetta: se non riesci a dereferenziare un `void*`, a cosa potrà mai servirti?

Come con `memcpy()` ti aiuta a scrivere funzioni generiche in grado di gestire più tipi di dati. Ma il segreto è che di nascosto *converti il `void*` in un altro tipo prima di usarlo!*

E la conversione è semplice: puoi semplicemente assegnare una variabile del tipo desiderato[91] .

```
char a = 'X';
// Carattere singolo

void *p = &a;
// p punta alla 'X'
char *q = p;
// q anche questa punta 'X'

printf("%c\n", *p);
// ERRORE--non può dereferenziare void*!
printf("%c\n", *q);
// Stampa "X"
```

Scriviamo il nostro `memcpy()` per provarlo. Possiamo copiare byte (`char`) e conosciamo il numero di byte perché viene passato.

```
void *my_memcpy(void *dest, void *src, int byte_count)
{
    // Converti void*s a char*s
    char *s = src, *d = dest;

    // Ora che abbiamo
    // char*s, possiamo
    // dereferenziarlo
    // e copiarli
    while (byte_count--) {
        *d++ = *s++;
    }

    // La maggior parte
```

90 Perché ricorda che la notazione dell'array è solo una dereferenziazione e alcuni calcoli sui puntatori, e non puoi dereferenziare un `void*`!

91 Puoi anche lanciare il `void*` su un altro tipo, ma non siamo ancora arrivati ai cast.

```
    // di queste
    // funzioni restituisce
    // la destinazione,
    // per ogni evenienza
    // è utile al chiamante.
    return dest;
}
```

Proprio all'inizio, copiamo i `void*` in `char*` in modo da poterli usare come `char*`. E' così semplice.

Poi divertendoci un po' in un ciclo while decrementiamo `byte_count` finché non diventa false (0). Ricordare che con il post-decremento viene calcolato il valore dell'espressione (per il `while` usato) e *quindi* la variabile viene decrementata.

E un po' ci divertiamo nel copiare dove assegniamo `*d = *s` per copiare il byte, ma lo facciamo con post-incremento in modo che sia `d` che `s` si spostino al byte successivo dopo aver effettuato l'assegnazione.

Infine, la maggior parte delle funzioni di memoria e di stringa restituiscono una copia di un puntatore alla stringa di destinazione nel caso in cui il chiamante desideri utilizzarla.

Ora che lo abbiamo fatto, voglio solo sottolineare rapidamente che possiamo usare questa tecnica per scorrere i byte di *qualsiasi* oggetto in C `float`, `struct` o qualsiasi altra cosa!

Facciamo un altro esempio del mondo reale con la routine `qsort()` incorporata che può ordinare qualsiasi cosa grazie al magico `void*`.

(Nell'esempio seguente puoi ignorare la parola `const` di cui non abbiamo ancora parlato.)

```
#include <stdio.h>
#include <stdlib.h>

// Il tipo di struttura
// che ordineremo
struct animale {
    char *nome;
    int gambe_contatore;
};

// Questa è una funzione
// di confronto chiamata
// da qsort() per
// aiutarla a determinare
// in base a cosa ordinare
// esattamente. Lo useremo
// per ordinare un
// array di struct
// animals di leg_count.
int compar(const void *elem1, const void *elem2)
{
    // Sappiamo che stiamo
    // classificando gli
    // struct animali,
    // quindi creiamoli entrambi
    // argomenti puntatori
    // per strutturare animali
    const struct animale *animal1 = elem1;
    const struct animale *animal2 = elem2;
```

```
    // Restituisce <0 =0 o >0
    // a seconda di cosa
    // vogliamo ordinare.

    // Ordiniamo in ordine
    // crescente in base a
    // gambe_contatore,
    // quindi restituiremo
    // la differenza in
    // gambe_contatore
    if (animal1->gambe_contatore > animal2->gambe_contatore)
        return 1;

    if (animal1->gambe_contatore < animal2->gambe_contatore)
        return -1;

    return 0;
}

int main(void)
{
    // Costruiamo una serie
    // di 4 animali con
    // strutture diverse
    // caratteristiche.
    // Questo array è fuori
    // ordine per leg_count,
    // ma lo sistemeremo
    // tra un secondo.
    struct animale a[4] = {
        {.name="Cane", .gambe_contatore=4},
        {.name="Scimmia", .gambe_contatore=2},
        {.name="Antilope", .gambe_contatore=4},
        {.name="Serpente", .gambe_contatore=0}
    };

    // Chiama qsort()
    // per ordinare l'array.
    // A qsort() gli deve
    // essere detto esattamente
    // in base a cosa ordinare
    // questi dati e lo faremo
    // all'interno di funzione
    // compar().
    //
    // Questa chiamata dice:
    // qsort array a, che
    // ha 4 elementi, e
    // ogni elemento
    // è grande sizeof(struct animale) byte,
    // e questo è la
    // funzione che confronterà
    // due elementi qualsiasi.
    qsort(a, 4, sizeof(struct animale), compar);

    // Stampateli tutti
    for (int i = 0; i < 4; i++) {
        printf("%d: %s\n", a[i].gambe_contatore, a[i].nome);
    }
```

```
}
```

Finché dai a `qsort()` una funzione in grado di confrontare due elementi che hai nel tuo array da ordinare può ordinare qualsiasi cosa. E lo fa senza la necessità di avere i tipi degli elementi codificati ovunque. `qsort()` riorganizza semplicemente blocchi di byte in base ai risultati della funzione `compar()` che hai passato.

12. Allocazione Manuale della Memoria

Questa è una delle grandi aree in cui il C probabilmente diverge dai linguaggi che già conosci: *la gestione manuale della memoria*.

Altri linguaggi utilizzano il conteggio dei riferimenti, la garbage collection o altri mezzi per determinare quando allocare nuova memoria per alcuni dati e quando deallocarla quando nessuna variabile vi fa riferimento.

Ed è carino. È bello poter non preoccuparsene, non pensare ai riferimenti di un elemento e avere fiducia che a un certo punto la memoria ad esso associata verrà liberata.

Ma C non è del tutto così.

Naturalmente in C, alcune variabili vengono automaticamente allocate e deallocate quando entrano nell'ambito e lasciano l'ambito (scope). Chiamiamo queste variabili automatiche. Sono le normali variabili "locali" dell'ambito del blocco. Nessun problema.

Ma cosa succede se vuoi che qualcosa persista più a lungo di un particolare blocco? È qui che entra in gioco la gestione manuale della memoria.

Puoi dire a C esplicitamente di allocare per te un certo numero di byte che puoi usare come preferisci. E questi byte rimarranno allocati finché non libererai esplicitamente quella memoria[92] .

È importante liberare la memoria con cui hai finito di lavorare! In caso contrario, avremo un *memory leak* e il processo continuerà a riservare quella memoria fino alla sua chiusura.

Se lo hai assegnato manualmente dovrai liberarlo manualmente quando hai finito.

Allora come lo facciamo? Impareremo un paio di nuove funzioni e utilizzeremo l'operatore `sizeof` per aiutarci a capire quanti byte allocare.

Nel linguaggio C comune gli sviluppatori dicono che le variabili locali automatiche vengono allocate "nello stack" e la memoria allocata manualmente è "nell'heap". Le specifiche non parlano di nessuna di queste cose ma tutti gli sviluppatori C sapranno di cosa stai parlando se ne parli.

Tutte le funzioni che impareremo in questo capitolo possono essere trovate in `<stdlib.h>`.

12.1. Allocazione e deallocazione, `malloc()` e `free()`

La funzione `malloc()` accetta un numero di byte da allocare e restituisce un puntatore vuoto a quel blocco di memoria appena allocata.

Dato che è un `void*` puoi assegnarlo a qualunque tipo di puntatore desideri… normalmente questo corrisponderà in qualche modo al numero di byte che stai allocando.

Quindi… quanti byte dovrei allocare? Possiamo usare `sizeof` per aiutarci. Se vogliamo allocare spazio sufficiente per un singolo `int`, possiamo usare `sizeof(int)` e passarlo a `malloc()`.

92 Oppure finché il programma non esce, nel qual caso tutta la memoria da esso allocata viene liberata. Asterisco: alcuni sistemi ti consentono di allocare memoria che persiste dopo l'uscita di un programma, ma dipende dal sistema, esula dall'ambito di questa guida e sicuramente non lo farai mai per errore.

Dopo aver finito con la memoria allocata possiamo chiamare `free()` per indicare che abbiamo finito con quella memoria e che può essere usata per qualcos'altro. Come argomento passi lo stesso puntatore che hai ottenuto da `malloc()` (o una sua copia). È un comportamento indefinito utilizzare una regione di memoria dopo averla liberata con `free()`.

Proviamo. Allocheremo memoria sufficiente per un `int`, quindi memorizzeremo qualcosa lì e lo stamperemo.

```
// Allocare spazio per un singolo int (sizeof(int) valore in byte):

int *p = malloc(sizeof(int));

*p = 12;
// Conserva qualcosa lì

printf("%d\n", *p);
// Stampalo: 12

free(p);
// Tutto finito con quella memoria

//*p = 3490;
//ERRORE: comportamento indefinito!
//Utilizzato dopo free()!
```

Ora in quell'esempio artificioso non c'è davvero alcun vantaggio. Avremmo potuto semplicemente usare un int automatico e avrebbe funzionato. Ma vedremo come la possibilità di allocare la memoria in questo modo presenta i suoi vantaggi, soprattutto con strutture dati più complesse.

Un'altra cosa che vedrai comunemente sfrutta il fatto che `sizeof` può darti la dimensione del tipo di risultato di qualsiasi espressione costante. Quindi potresti inserire anche un nome di variabile e usarlo. Eccone un esempio proprio come il precedente:

```
int *p = malloc(sizeof *p);
// *p è un int, quindi uguale a sizeof(int)
```

12.2. Controllo degli errori

Tutte le funzioni di allocazione restituiscono un puntatore alla porzione di memoria appena allocata o `NULL` se per qualche motivo la memoria non può essere allocata.

Alcuni S.O. come Linux possono essere configurati in modo tale che `malloc()` non restituisca mai `NULL` anche se hai esaurito la memoria. Ma nonostante ciò dovresti sempre scrivere il codice tenendo presente le misure di sicurezza.

```
int *x;

x = malloc(sizeof(int) * 10);

if (x == NULL) {
    printf("Errore allocati 10 int\n");
    // fai qualcosa per gestirlo
}
```

Ecco uno schema comune che vedrai in cui eseguiamo l'assegnazione e la condizione sulla stessa riga:

```
int *x;
```

```
if ((x = malloc(sizeof(int) * 10)) == NULL)
    printf("Errore allocati 10 int\n");
    // fai qualcosa qui per gestirlo
}
```

12.3. Assegnazione dello spazio per un array

Abbiamo visto come allocare lo spazio per un singolo elemento; ora che ne dici di un gruppo di quelli in un array?

In C un array è un insieme di elementi uguali uno dopo l'altro in un tratto di memoria contiguo.

Possiamo allocare un tratto contiguo di memoria: abbiamo visto come farlo. Se volessimo 3490 byte di memoria potremmo semplicemente chiederlo:

```
char *p = malloc(3490);
// Voila
```

E——infatti! si tratta di un array di 3490 `char` (ovvero una stringa!) poiché ogni `char` è 1 byte. In altre parole `sizeof(char)` è 1.

Nota: non viene eseguita alcuna inizializzazione sulla memoria appena allocata: è piena di spazzatura. Cancellalo con `memset()` se vuoi, o vedi `calloc()` di sotto.

Ma possiamo semplicemente moltiplicare la dimensione dell'oggetto che vogliamo per il numero di elementi che vogliamo e quindi accedervi utilizzando la notazione del puntatore o dell'array. Esempio!

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Assegna lo spazio per 10 int
    int *p = malloc(sizeof(int) * 10);

    // Assegna loro dei valori da 0-45:
    for (int i = 0; i < 10; i++)
        p[i] = i * 5;

    // Stampa tutti i valori 0, 5, 10, 15, ..., 40, 45
    for (int i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    // Libera lo spazio
    free(p);
}
```

La chiave è nella riga `malloc()`. Se sappiamo che ogni `int` richiede `sizeof(int)` byte per contenerlo e sappiamo che ne vogliamo 10, possiamo semplicemente allocare esattamente quel numero di byte con:

```
sizeof(int) * 10
```

E questo trucco funziona per ogni tipo. Basta passarlo a `sizeof` e moltiplicarlo per la dimensione dell'array.

12.4. Un'alternativa: `calloc()`

Questa è un'altra funzione di allocazione che funziona in modo simile a `malloc()` con due

differenze fondamentali:

- Invece di un singolo argomento passi la dimensione di un elemento e il numero di elementi che desideri allocare. È come se fosse fatto per l'allocazione degli array.
- Ripulisce la memoria a zero.

Utilizzi ancora `free()` per deallocare la memoria ottenuta tramite `calloc()`.

Ecco un confronto tra `calloc()` e `malloc()`.

```
// Assegna lo spazio per 10 int con calloc(), inizializziamolo a 0:
int *p = calloc(10, sizeof(int));

// Assegna lo spazio per 10 int con malloc(), inizializzato a 0:
int *q = malloc(10 * sizeof(int));
memset(q, 0, 10 * sizeof(int));   // impostalo a 0
```

Ancora una volta il risultato è lo stesso per entrambi tranne che `malloc()` non azzera la memoria per impostazione predefinita.

12.5. Modifica della dimensione allocata con `realloc()`

Se hai già assegnato 10 `int` ma in seguito decidi che te ne servono 20, cosa puoi fare?

Un'opzione è allocare un po' di nuovo spazio e poi con `memcpy()` occupare la memoria… ma a volte non è necessario spostare nulla. E c'è una funzione abbastanza intelligente da fare la cosa giusta in tutte le circostanze giuste: `realloc()`.

Richiede un puntatore a una memoria precedentemente allocata (da `malloc()` o `calloc()`) e una nuova dimensione per la regione di memoria.

Di conseguenza aumenta o riduce la memoria e restituisce un puntatore ad essa. A volte potrebbe restituire lo stesso puntatore (se i dati non dovevano essere copiati altrove) oppure potrebbe restituirne uno diverso (se i dati dovevano essere copiati).

Assicurati che quando chiami `realloc()` specifichi il numero di *byte* da allocare e non solo il numero di elementi dell'array! Che è:

```
num_floats *= 2;

np = realloc(p, num_floats);
// SBAGLIATO: sono necessari byte,
// non numero di elementi!

np = realloc(p, num_floats * sizeof(float));
// Meglio!
```

Assegniamo un array di 20 `float` e dopo cambiamo idea e rendiamolo un array di 40.

Assegneremo il valore restituito di `realloc()` a un altro puntatore solo per assicurarci che non sia `NULL`. Se non lo è possiamo riassegnarlo al nostro puntatore originale. (Se assegnassimo il valore restituito direttamente al puntatore originale, perderemmo quel puntatore se la funzione restituisse `NULL` e non avremmo modo di recuperarlo.)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    // Assegna spazio per 20 float
```

```
    float *p = malloc(sizeof *p * 20);
    // sizeof *p uguale a sizeof(float)

    // Assegna loro valori frazionari 0.0-1.0:
    for (int i = 0; i < 20; i++)
        p[i] = i / 20.0;

    // Ma aspetta! Realizziamo un array di 40 elementi
    float *new_p = realloc(p, sizeof *p * 40);

    // Controlla se la riassegnazione
    // è avvenuta con successo
    if (new_p == NULL) {
        printf("Errore di reallocazione\n");
        return 1;
    }

    // Se lo facessimo, possiamo semplicemente riassegnare p
    p = new_p;

    // E assegna i valori dei nuovi elementi nell'intervallo 1.0-2.0
    for (int i = 20; i < 40; i++)
        p[i] = 1.0 + (i - 20) / 20.0;

    // Stampa tutti i valori 0.0-2.0 nei 40 elementi:
    for (int i = 0; i < 40; i++)
        printf("%f\n", p[i]);

    // Libera lo spazio
    free(p);
}
```

Nota qui come abbiamo preso il valore restituito da `realloc()` e lo abbiamo riassegnato nella stessa variabile puntatore `p` che abbiamo passato. È abbastanza comune da fare.

Inoltre se la riga 7 sembra strana con quella `sizeof *p` lì dentro, ricorda che `sizeof` funziona sulla dimensione del tipo dell'espressione. E il tipo di `*p` è `float` quindi quella riga è equivalente a `sizeof(float)`.

12.5.1. Lettura in righe di lunghezza arbitraria

Voglio dimostrare due cose con questo esempio in piena regola.

1. Uso di `realloc()` per aumentare un buffer man mano che leggiamo più dati.
2. Utilizzo di `realloc()` per ridurre il buffer alla dimensione perfetta dopo aver completato la lettura.

Quello che vediamo qui è un ciclo che chiama `fgetc()` più e più volte per aggiungerlo a un buffer finché non vediamo che l'ultimo carattere è una nuova riga.

Una volta trovata la nuova riga riduce il buffer alla dimensione giusta e lo restituisce.

```
#include <stdio.h>
#include <stdlib.h>

// Legge una riga di dimensione
// arbitraria da un file
//
// Restituisce un puntatore alla linea.
```

```c
// Restituisce NULL in caso di EOF o errore.
//
// Spetta al chiamante free()
// questo puntatore una volta terminato.
//
// Nota che questo rimuove
// la nuova riga dal risultato.
// Se avete bisogno
// è lì, probabilmente è
// meglio passarlo a
// una attività temporanea.

char *readline(FILE *fp)
{
    int offset = 0;
// Il carattere successivo dell'indice va nel buffer
    int bufsize = 4;
// Preferibilmente potenza di 2 dimensioni iniziali
    char *buf;
// Il buffer
    int c;
// Il personaggio che abbiamo letto dentro

    buf = malloc(bufsize);
// Assegnare il buffer iniziale

// Controllo degli errori
    if (buf == NULL)
        return NULL;

    // Ciclo principale: leggi fino a nuova riga o EOF
    while (c = fgetc(fp), c != '\n' && c != EOF) {

        // Controlla se abbiamo esaurito lo spazio nel buffer contabile
        // per il byte aggiuntivo per il terminatore NUL
        if (offset == bufsize - 1) {  // -1 per il terminatore NUL
            bufsize *= 2;  // 2x the space

            char *new_buf = realloc(buf, bufsize);

            if (new_buf == NULL) {
                free(buf);   // Per errore, free e bail
                return NULL;
            }

            buf = new_buf;  // realloc riuscito
        }

        buf[offset++] = c;  // Aggiungi il byte al buffer
    }

    // Premiamo newline o EOF...

    // Se a EOF non legge byte, libera il buffer e
    // restituisce NULL per indicare che siamo a EOF:
    if (c == EOF && offset == 0) {
        free(buf);
        return NULL;
    }
```

```
    // Restringi la dimensione
    if (offset < bufsize - 1) {
        // Se siamo vicini alla fine
        char *new_buf = realloc(buf, offset + 1);
        // +1 per il terminatore NUL

        // In caso di successo, punta buf su new_buf;
        // altrimenti lo lasceremo dov'è
        if (new_buf != NULL)
            buf = new_buf;
    }

    // Aggiungi il terminatore NUL
    buf[offset] = '\0';

    return buf;
}

int main(void)
{
    FILE *fp = fopen("foo.txt", "r");

    char *line;

    while ((line = readline(fp)) != NULL) {
        printf("%s\n", line);
        free(line);
    }

    fclose(fp);
}
```

Quando si accresce la memoria in questo modo è comune (e a volte una legge) raddoppiare lo spazio necessario per ogni passaggio solo per ridurre al minimo il numero di `realloc()` che si verificano.

Infine potresti notare che `readline()` restituisce un puntatore a un buffer `malloc()~d. In quanto tale spetta al chiamante esplicitare ~free()` la memoria quando avrà finito.

12.5.2. realloc() with NULL

È tempo di curiosità! Queste due righe sono equivalenti:

```
char *p = malloc(3490);
char *p = realloc(NULL, 3490);
```

Ciò potrebbe essere utile se si dispone di una sorta di ciclo di allocazione e non si desidera utilizzare un caso speciale per il primo `malloc()`.

```
int *p = NULL;
int length = 0;

while (!done) {
    // Assegna altri 10 int:
    length += 10;
    p = realloc(p, sizeof *p * length);
```

```
    // Fai cose straordinarie
    // ...
}
```

In questo esempio non avevamo bisogno di una `malloc()` iniziale poiché `p` era inizialmente `NULL`.

12.6. Allocazioni allineate

Probabilmente non avrai bisogno di usarlo.

E non voglio dilungarmi troppo parlandone adesso ma c'è questa cosa chiamata *allineamento della memoria* ha a che fare con l'indirizzo di memoria (valore del puntatore) che è un multiplo di un certo numero.

Ad esempio, un sistema potrebbe richiedere che i valori a 16 bit inizino su indirizzi di memoria multipli di 2. Oppure che i valori a 64 bit inizino su indirizzi di memoria multipli di 2, 4 o 8, ad esempio. Dipende dalla CPU.

Alcuni sistemi richiedono questo tipo di allineamento per un accesso rapido alla memoria altri addirittura per l'accesso totale alla memoria.

Ora se usi `malloc()`, `calloc()` o `realloc()` C ti darà una porzione di memoria ben allineata per qualsiasi valore anche per le `struct`. Funziona in tutti i casi.

Ma potrebbero esserci momenti in cui sai che alcuni dati possono essere allineati a un confine più piccolo o devono essere allineati a uno più grande per qualche motivo. Immagino che questo sia più comune con la programmazione di sistemi embedded.

In questi casi, è possibile specificare un allineamento con `aligned_alloc()`.

L'allineamento è una potenza intera di due maggiore di zero, quindi 2, 4, 8, 16, ecc. e lo passi a `linked_alloc()` prima del numero di byte che ti interessano.

L'altra restrizione è che il numero di byte allocati deve essere un multiplo dell'allineamento. Ma questo potrebbe cambiare. Vedere il rapporto sui difetti C 460[93]

Facciamo un esempio allocando su un range di 64 byte:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Allocare 256 byte allineati
    // su un limite di 64 byte
    char *p = aligned_alloc(64, 256);
    // 256 == 64 * 4

    // Copia una stringa lì dentro e stampala
    strcpy(p, "Hello, world!");
    printf("%s\n", p);

    // Free lo spazio
    free(p);
}
```

Voglio lanciare una nota qui a riguardo `realloc()` e `aligned_alloc()`. `realloc()` non ha

93 http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

alcuna garanzia di allineamento quindi se hai bisogno di ottenere uno spazio riallocato allineato dovrai farlo nel modo più duro con `memcpy()`.

Ecco una funzione non standard `aligned_alloc()` se ne hai bisogno:

```
void *aligned_realloc(void *ptr, size_t old_size, size_t alignment, size_t
size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    size_t copy_size = old_size < size? old_size: size;  // get min

    if (ptr != NULL)
        memcpy(new_ptr, ptr, copy_size);

    free(ptr);

    return new_ptr;
}
```

Tieni presente che copia *sempre* i dati impiegando tempo, mentre il vero `realloc()` lo eviterà se possibile. Quindi non è affatto efficiente. Evita la necessità di riallocare i dati personalizzati.

13. Ambito

L'ambito riguarda quali variabili sono visibili in quali contesti.

13.1. Ambito del blocco

Questo è l'ambito di quasi tutte le variabili definite dagli sviluppatori. Include ciò che altri linguaggi potrebbero chiamare “ambito della funzione”, ovvero variabili dichiarate all'interno delle funzioni.

La regola di base è che se hai dichiarato una variabile in un blocco delimitato da parentesi graffe, l'ambito di quella variabile è quel blocco.

Se c'è un blocco all'interno di un blocco le variabili dichiarate nel blocco *interno* sono locali a quel blocco e non possono essere viste nell'ambito esterno.

Una volta terminato l'ambito di una variabile non è più possibile fare riferimento a quella variabile e si può considerare che il suo valore sia finito nel grande secchio[94] nel cielo.

Un esempio con un ambito nidificato:

```
#include <stdio.h>

int main(void)
{
    int a = 12;
// Locale al blocco esterno, ma visibile nel blocco interno

    if  (a == 12) {
        int b = 99;
// Locale al blocco interno, non visibile nel blocco esterno

        printf("%d %d\n", a, b);
// OK: "12 99"
```

94 https://en.wikipedia.org/wiki/Bit_bucket

```
    }

    printf("%d\n", a);
// OK, siamo ancora nell'ambito di a

    printf("%d\n", b);
// ILLEGALE, fuori dall'ambito di b
}
```

13.1.1. Dove definire le variabili

Un altro fatto divertente è che puoi definire variabili ovunque nel blocco entro limiti ragionevoli: hanno l'ambito di quel blocco ma non possono essere utilizzate prima di essere definite.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    printf("%d\n", i);
// OK: "0"

    //printf("%d\n", j);
// ILLEGALE--non può usare j prima che sia definito

    int j = 5;

    printf("%d %d\n", i, j);
// OK: "0 5"
}
```

Storicamente il C richiedeva che tutte le variabili fossero definite prima di qualsiasi codice nel blocco, ma questo non è più il caso nello standard C99.

13.1.2. Nascondere le variabili

Se hai una variabile con lo stesso nome in un ambito interno di una in un ambito esterno, quella nell'ambito interno ha la precedenza finché sei in esecuzione nell'ambito interno. Cioè *nasconde* quello all'esterno per tutta la durata della sua vita.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    {
        int i = 20;

        printf("%d\n", i);
// Ambito interno i, 20 (i esterno è nascosto)
    }

    printf("%d\n", i);
// Ambito esterno i, 10
}
```

Potresti aver notato in quell'esempio che ho appena inserito un blocco alla riga 7 non tanto

un'istruzione `for` o `if` per avviarlo! Questo è perfettamente legale. A volte uno sviluppatore vorrà raggruppare insieme un gruppo di variabili locali per un calcolo rapido e lo farà, ma è raro vederlo.

13.2. Ambito del file

Se definisci una variabile all'esterno di un blocco tale variabile ha *ambito del file*. È visibile in tutte le funzioni del file successivo e condiviso tra loro. (Un'eccezione è se un blocco definisce una variabile con lo stesso nome nasconderebbe quella nell'ambito del file.)

Questo è il più vicino a quello che considereresti un ambito “globale” in un’altro linguaggio.

Per esempio:

```
#include <stdio.h>

int shared = 10;
// Ambito del file! Visibile all'intero file dopo questo!

void func1(void)
{
    shared += 100;
// Now shared holds 110
}

void func2(void)
{
    printf("%d\n", shared);
// Stampa "110"
}

int main(void)
{
    func1();
    func2();
}
```

Tieni presente che se `shared` fosse dichiarato in fondo al file non verrebbe compilato. Deve essere dichiarato *prima* che qualsiasi funzione lo utilizzi.

Esistono modi per modificare ulteriormente gli elementi nell'ambito del file vale a dire con static ed extern, ma ne parleremo più avanti.

13.3. Ambito del ciclo `for`

Non so davvero come chiamarlo nel C11 §6.8.5.3¶1 non gli dà un nome proprio. Lo abbiamo già fatto alcune volte anche in questa guida. È quando dichiari una variabile all'interno della prima clausola di un ciclo `for`:

```
for (int i = 0; i < 10; i++)
    printf("%d\n", i);

printf("%d\n", i);
// ILLEGALE--i rientra solo nell'ambito del ciclo-for
```

In questo esempio la vita di `i` inizia nel momento in cui viene definita e continua per tutta la durata del ciclo.

Se il corpo del ciclo è racchiuso in un blocco, le variabili definite nel ciclo `for` sono visibili da quell'ambito interno.

A meno che ovviamente quell'ambito interiore non li nasconda. Questo esempio pazzesco stampa 999 cinque volte:

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++) {
        int i = 999;
// Nasconde la i nell'ambito del ciclo for
        printf("%d\n", i);
    }
}
```

13.4. Una nota sull'ambito della funzione

La specifica C fa riferimento *all'ambito della funzione*, ma viene utilizzata esclusivamente con le *etichette*, cosa di cui non abbiamo ancora parlato. Ne parleremo un altro giorno.

14. Tipi II: molti più tipi!

Siamo abituati ai tipi `char`, `int` e `float` ma ora è il momento di portare queste cose al livello successivo e vedere cos'altro abbiamo là fuori nel reparto tipi!

14.1. Signed e Unsigned Integers

Finora abbiamo utilizzato `int` come tipo con *segno* ovvero un valore che può essere negativo o positivo. Ma C ha anche specifici tipi interi *senza segno* che possono contenere solo numeri positivi.

Questi tipi sono preceduti dalla parola chiave `unsigned`.

```c
int a;
// signed
signed int a;
// signed
signed a;
// signed, "abbreviazione" per "int" o "signed int", rara
unsigned int b;
// unsigned
unsigned c;
// unsigned, abbreviazione di "unsigned int"
```

Perché? Perché dovresti decidere di voler mantenere solo numeri positivi?

Risposta: puoi ottenere numeri più grandi in una variabile senza segno rispetto a quelli con segno.

Ma perché?

Si può pensare che gli interi siano rappresentati da un certo numero di *bit*[95] . Sul mio computer, un `int` è rappresentato da 64 bit.

E ogni permutazione di bit 1 o 0 rappresenta un numero. Possiamo decidere come spartire questi numeri.

Con i numeri con segno utilizziamo (approssimativamente) metà delle permutazioni per rappresentare i numeri negativi e l'altra metà per rappresentare i numeri positivi.

Con unsigned utilizziamo *tutte* le permutazioni per rappresentare i numeri positivi.

95 "Bit" è l'abbreviazione di cifra binaria. Il binario è solo un altro modo di rappresentare i numeri. Invece delle cifre 0-9 come siamo abituati ci sono le cifre 0-1

Sul mio computer con `int` a 64 bit che utilizzano il complemento a due[97] per rappresentare numeri senza segno ho i seguenti limiti sull'intervallo di numeri interi:

Tipo	Minimo	Massimo
int	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned int	0	18,446,744,073,709,551,615

Si noti che il più grande `unsigned int` positivo è circa il doppio del più grande `int` positivo. Quindi puoi ottenere una certa flessibilità lì.

14.2. Tipi carattere

Ricordate `char`? Il tipo che possiamo usare per contenere un singolo carattere?

```
char c = 'B';

printf("%c\n", c);  // "B"
```

Ho una cosa scioccante per te: in realtà è un numero intero.

```
char c = 'B';

// Cambiato da %c a %d:
printf("%d\n", c);  // 66 (!!)
```

In fondo `char` è solo un piccolo `int` ovvero un numero intero che occupa solo un singolo byte di spazio, limitando il suo intervallo a…

Qui le specifiche C diventano un po' stravaganti. Le specifiche ci assicurano che un carattere è un singolo byte, es. `sizeof(char) == 1`. Ma poi in C11 §3.6¶3 si esagera nel dire:

> Un byte è composto da una sequenza contigua di bit *il cui numero è definito dall'implementazione*.

Aspetta cosa? Alcuni di voi potrebbero essere abituati all'idea che un byte sia composto da 8 bit giusto? Voglio dire, è proprio quello giusto? E la risposta è: "Quasi certamente".[98] Ma il C è un linguaggio vecchio e le macchine a quei tempi avevano, per così dire un'opinione più *rilassata* su quanti bit ci fossero in un byte. E nel corso degli anni, C ha mantenuto questa flessibilità.

Ma supponendo che i tuoi byte in C siano 8 bit, come lo sono praticamente per tutte le macchine al mondo che vedrai mai, l'intervallo di un `char` è…

—Quindi prima che possa dirtelo risulta che i `char` potrebbero essere signed o unsigned a seconda del compilatore. A meno che tu non lo specifichi esplicitamente.

In molti casi, avere solo `char` va bene perché non ti interessa il segno dei dati. Ma se hai bisogno di caratteri signed o unsigned *devi* essere specifico:

```
char a;
// Potrebbe essere signed o unsigned
signed char b;
// Decisamente signed
```

97 https://en.wikipedia.org/wiki/Two%27s_complement

98 Il termine industriale per una sequenza di 8 bit esatti e indiscutibili è un ottetto.

```
unsigned char c;
// Decisamente unsigned
```

OK, ora finalmente possiamo capire l'intervallo di numeri se assumiamo che un `char` sia di 8 bit e il tuo sistema utilizzi la rappresentazione virtualmente universale del complemento a due per con segno e senza segno[99] .

Quindi, tenendo a mente questi vincoli possiamo finalmente calcolare i nostri intervalli:

tipo `char`	Minimo	Massimo
signed char	-128	127
unsigned char	0	255

E gli intervalli per `char` sono definiti dall'implementazione.

Fatemi capire bene. `char` è in realtà un numero, quindi possiamo fare i conti?

Sì! Ricorda solo di mantenere le cose nel raggio del `char`!

```
#include <stdio.h>

int main(void)
{
    char a = 10, b = 20;

    printf("%d\n", a + b);  // 30!
}
```

Che dire di quei caratteri costanti tra virgolette singole come `'B'`? Come può avere un valore numerico?

Anche qui le specifiche sono instabili poiché C non è progettato per essere eseguito su un singolo tipo di sistema sottostante.

Ma supponiamo per il momento che il tuo set di caratteri sia basato su ASCII[100] almeno per i primi 128 caratteri. In tal caso la costante carattere verrà convertita in un carattere il cui valore è uguale al valore ASCII del carattere.

Era solo un boccone. Facciamo solo un esempio:

```
#include <stdio.h>

int main(void)
{
    char a = 10;
    char b = 'B';  // valore ASCII 66

    printf("%d\n", a + b);  // 76!
}
```

Ciò dipende dall'ambiente di esecuzione e dal set di caratteri utilizzato[101] . Uno dei set di caratteri

99 In generale, se hai un `$n$` in complemento a due del bit l'intervallo con segno è `$-2^{n-1}$` a `$2^{n-1}-1$` . E l'intervallo senza segno è `$0$` a `$2^n-1$`.

100 https://en.wikipedia.org/wiki/ASCII

101 https://en.wikipedia.org/wiki/List_of_information_system_character_sets

più popolari oggi è Unicode[103] (che è un superset di ASCII) quindi per i tuoi 0-9 di base AZ, az e la punteggiatura quasi sicuramente otterrai i valori ASCII da essi.

14.3. Altri tipi interi: `short`, `long`, `long long`

Finora abbiamo utilizzato generalmente solo due tipi di numeri interi:

- char
- int

e recentemente abbiamo appreso delle varianti senza segno dei tipi interi. E abbiamo scoperto che `char` era segretamente un piccolo `int` sotto mentite spoglie. Quindi sappiamo che gli `int` possono avere più dimensioni di bit.

Ma ci sono un altro paio di tipi interi che dovremmo considerare, e il *minimo* minimo e i valori massimi che possono contenere.

Sì ho detto "minimo" due volte. Le specifiche dicono che questi tipi conterranno numeri *almeno* di queste dimensioni quindi la tua implementazione potrebbe essere diversa. Il file di intestazione `<limits.h>` definisce le macro che contengono i valori interi minimo e massimo; fate affidamento su questo per essere sicuri e non *codificare mai o dichiarare questi valori*.

I tipi aggiungiamo sono `short int`, `long int` e `long long int`. Di solito, quando si utilizzano questi tipi gli sviluppatori C omettono `int` (ad esempio `long long`) e il compilatore è assolutamente felice.

```
// Queste due righe sono equivalenti:
long long int x;
long long x;

// E anche questi:
short int x;
short x;
```

Diamo un'occhiata ai tipi e alle dimensioni dei dati interi in ordine crescente raggruppati per segno.

Tipo	Byte minimi	Valore minimo	Massimo Valore
char	1	-127 o 0	127 o 255 [104]
signed char	1	-127	127
short	2	-32767	32767
int	2	-32767	32767
long	4	-2147483647	2147483647
long long	8	-9223372036854775807	9223372036854775807

103https://en.wikipedia.org/wiki/Unicode
104Dipende se un `carattere` viene impostato di default su `signed char` o `unsigned char`

unsigned char	1	0	255
unsigned short	2	0	65535
unsigned int	2	0	65535
unsigned long	4	0	4294967295
unsigned long long	8	0	18446744073709551615

Non c'è un tipo `long long long`. Non puoi continuare ad aggiungere `long` in quel modo. Non essere sciocco.

> I fan del complemento a due potrebbero aver notato qualcosa di divertente in quei numeri. Perché ad esempio il `signed char` si ferma a -127 invece che a -128? Ricorda: questi sono solo i minimi richiesti dalle specifiche. Alcune rappresentazioni numeriche (come il segno e la magnitudine[105]) terminano con ±127.

Eseguiamo la stessa tabella sul mio sistema in complemento a due a 64 bit e vediamo cosa viene fuori:

Tipo	I miei byte	Valore minimo	Valore massimo
char	1	-128	127[106]
signed char	1	-128	127
short	2	-32768	32767
int	4	-2147483648	2147483647
long	8	-9223372036854775808	9223372036854775807
long long	8	-9223372036854775808	9223372036854775807
unsigned char	1	0	255
unsigned short	2	0	65535
unsigned int	4	0	4294967295

105https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation
106Il mio `char` è signed

unsigned long	8	0	18446744073709551615
unsigned long long	8	0	18446744073709551615

Questo è un po' più sensato ma possiamo vedere come il mio sistema abbia limiti più ampi rispetto ai minimi nelle specifiche.

Allora quali sono le macro in `<limits.h>`?

Tipo	`Min Macro`	`Max Macro`
char	`CHAR_MIN`	`CHAR_MAX`
signed char	`SCHAR_MIN`	`SHRT_MAX`
short	`SHRT_MIN`	`SCHAR_MAX`
int	`INT_MIN`	`INT_MAX`
long	`LONG_MIN`	`LONG_MAX`
long long	`LLONG_MIN`	`LLONG_MAX`
unsigned char	0	`UCHAR_MAX`
unsigned short	0	`USHRT_MAX`
unsigned int	0	`UINT_MAX`
unsigned long	0	`ULONG_MAX`
unsigned long long	0	`ULLONG_MAX`

Nota che c'è un modo nascosto lì dentro per determinare se un sistema utilizza caratteri signed o unsigned `char`. Se `CHAR_MAX == UCHAR_MAX`, deve essere unsigned.

Nota inoltre che non esiste una macro minima per le varianti `unsigned`: sono solo 0.

14.4. Altri Float: `double` e `long double`

Vediamo cosa dicono le specifiche sui numeri in virgola mobile nel §5.2.4.2.2¶1-2:

I seguenti parametri vengono utilizzati per definire il modello per ciascun tipo a virgola mobile:

Parametri	Definizione

s	segno(±1)
b	base o radice della rappresentazione dell'esponente (un numero intero > 1)
e	esponente (un numero intero compreso tra un minimo e_{min} e un massimo e_{max})
p	precisione (il numero di cifre base-b nel significando)
f_k	numeri interi non negativi inferiori a b (le cifre significative)

Un numero in virgola mobile (x) è definito dal seguente modello:

$x=sb^e\sum\limits_{k=1}^p f_kb^{-k}$, $e_{min}\le e\le e_{max}$

Spero che questo ti abbia chiarito le cose.

Ok bene. Facciamo un passo indietro e vediamo cosa è pratico.

Nota: in questa sezione facciamo riferimento a una serie di macro. Si trovano nell'intestazione <float.h>.

I numeri in virgola mobile sono codificati in una sequenza specifica di bit (il formato IEEE-754101 è estremamente popolare) in byte.

Immergendoci un po' di più il numero è sostanzialmente rappresentato come il *significativo* (che è la parte numerica: le cifre significative stesse a volte chiamate anche *mantissa*) e *l'esponente*, che è la potenza a cui elevare le cifre. Ricorda che un esponente negativo può rendere un numero più piccolo.

Immagina che stiamo usando 10 come numero da elevare di un esponente. Potremmo rappresentare i seguenti numeri utilizzando un significando di 12345, e esponenti di $-3, -4$ e 0 per codificare i seguenti valori in virgola mobile:

$$12345\times10^{-3}=12.345$$

$$12345\times10^{-3}=12.345$$

$$12345\times10^0=12345$$

Per tutti questi numeri il significativo rimane lo stesso. L'unica differenza è l'esponente.

Sulla tua macchina la base dell'esponente è probabilmente $$2$$, non $$10$$, poiché ai computer piace il formato binario. Puoi verificarlo stampando la macro FLT_RADIX.

Quindi abbiamo un numero rappresentato da un numero di byte codificati in qualche modo. Poiché esiste un numero limitato di modelli di bit è possibile rappresentare un numero limitato di numeri in

virgola mobile.

Ma più in particolare solo un certo numero di cifre decimali significative può essere rappresentato con precisione.

Come puoi ottenere di più? Puoi utilizzare tipi di dati più grandi!

E ne abbiamo un paio. Conosciamo già il `float` ma per maggiore precisione abbiamo `double`. E per una precisione ancora maggiore abbiamo `long double` (non correlato a `long int` tranne che per il nome).

Le specifiche non indicano quanti byte di spazio di archiviazione dovrebbe richiedere ciascun tipo ma sul mio sistema possiamo vedere gli aumenti delle dimensioni relative:

Tipo	`sizeof`
float	4
double	8
long double	16

Quindi ciascuno dei tipi (sul mio sistema) utilizza quei bit aggiuntivi per una maggiore precisione.

Ma *di quanta* precisione parliamo qui? Quanti numeri decimali possono essere rappresentati da questi valori?

Bene, C ci fornisce un sacco di macro in `<float.h>` per aiutarci a capirlo.

Diventa un po' instabile se si utilizza un sistema base-2 (binario) per memorizzare i numeri (che sono praticamente tutti sul pianeta probabilmente incluso te) ma abbi pazienza mentre troviamo una soluzione.

14.4.1. Quante cifre decimali?

La domanda da un milione di dollari è: "Quante cifre decimali significative posso memorizzare in un dato tipo a virgola mobile in modo da ottenere lo stesso numero decimale quando lo stampo?"

Il numero di cifre decimali che puoi memorizzare in un tipo a virgola mobile e ottenere sicuramente lo stesso numero quando lo stampi è dato da queste macro:

Tipo	Cifre decimali che è possibile memorizzare	Minimo
float	`FLT_DIG`	6
double	`DBL_DIG`	10
long double	`LDBL_DIG`	10

Sul mio sistema `FLT_DIG` è 6 quindi posso essere sicuro che se stampo un `float` a 6 cifre otterrò

la stessa cosa. (Potrebbero essere più cifre: alcuni numeri verranno restituiti correttamente con più cifre. Ma 6 tornerà sicuramente.) Ad esempio stampare i `float` seguendo questo schema di cifre crescenti, apparentemente arriviamo a 8 cifre prima che qualcosa vada storto ma dopo torniamo a 7 cifre corrette.

```
0.12345
0.123456
0.1234567
0.12345678
0.123456791  <-- Le cose cominciano ad andare male
0.1234567910
```

Facciamo un'altra demo. In questo codice avremo due `float` che contengono entrambi numeri che hanno `FLT_DIG` cifre decimali significative[107] . Poi li sommiamo insieme per quello che dovrebbero essere 12 cifre decimali significative. Ma questo è più di quanto possiamo memorizzare in un `float` e recuperare correttamente come stringa quindi vediamo che quando lo stampiamo le cose iniziano ad andare storte dopo la settima cifra significativa.

```
#include <stdio.h>
#include <float.h>

int main(void)
{
    // Entrambi questi numeri
    // hanno 6 cifre significative
    // quindi possono essere
    // memorizzati accuratamente
    // in un float:

    float f = 3.14159f;
    float g = 0.00000265358f;

    printf("%.5f\n", f);
    // 3.14159       -- corretto!
    printf("%.11f\n", g);
    // 0.00000265358 -- corretto!

    // Now add them up
    f += g;
    // 3.14159265358 è cosa f _dovrebbe_ essere

    printf("%.11f\n", f);
    // 3.14159274101 -- sbagliato!
}
```

(Il codice precedente ha una `f` dopo le costanti numeriche: ciò indica che la costante è di tipo `float` al contrario del valore predefinito `double`. Ne parleremo più avanti.)

Ricorda che `FLT_DIG` è il numero sicuro di cifre che puoi memorizzare in un `float` e recuperarlo correttamente.

A volte potresti ricavarne uno o due in più. Ma a volte otterrai solo le cifre `FLT_DIG`. La cosa sicura: se memorizzi un numero qualsiasi di cifre fino a `FLT_DIG` incluso in un `float`, sei sicuro di recuperarle correttamente.

Quindi questa è la storia. `FLT_DIG`. Fine.

107Questo programma viene eseguito come indicato nei commenti su un sistema con `FLT_DIG` pari a 6 che utilizza numeri in virgola mobile IEEE-754 base-2. Altrimenti, potresti ottenere un output diverso.

…O è?

14.4.2. Conversione in decimale e viceversa

Ma memorizzare un numero in base 10 in un numero in virgola mobile e recuperarlo è solo metà della storia.

Si scopre che i numeri in virgola mobile possono codificare numeri che richiedono più cifre decimali per essere stampati completamente. È solo che il tuo grande numero decimale potrebbe non corrispondere a uno di quei numeri.

Cioè quando guardi i numeri in virgola mobile da uno a quello successivo c'è un divario. Se provi a codificare un numero decimale in tale intervallo verrà utilizzato il numero in virgola mobile più vicino. Ecco perché puoi codificare solo `FLT_DIG` per un `float`.

Ma che dire di quei numeri in virgola mobile che *non sono* nell'intervallo? Di quanti posti hai bisogno per stamparli accuratamente?

Un altro modo per formulare questa domanda è: per ogni dato numero in virgola mobile, quante cifre decimali devo conservare se voglio riconvertire il numero decimale in un numero in virgola mobile identico? Cioè quante cifre devo stampare in base 10 per recuperare **tutte** le cifre in base 2 nel numero originale?

A volte potrebbero essere solo pochi. Ma per sicurezza ti consigliamo di convertire in decimale con un certo numero sicuro di cifre decimali. Quel numero è codificato nelle seguenti macro:

Macro	Descrizione
`FLT_DECIMAL_DIG`	Numero di cifre decimali codificate in un `float`.
`DBL_DECIMAL_DIG`	Numero di cifre decimali codificate in un `double`.
`LDBL_DECIMAL_DIG`	Numero di cifre decimali codificate in un `long double`.
`DECIMAL_DI`	Uguale alla codifica più ampia `LDBL_DECIMAL_DIG`.

Vediamo un esempio in cui `DBL_DIG` è 15 (quindi è tutto ciò che possiamo avere in una costante) ma `DBL_DECIMAL_DIG` è 17 (quindi dobbiamo convertire in 17 numeri decimali per preservare tutti i bit del `double` originale).

Assegniamo il numero a 15 cifre significative 0,123456789012345 a `x` e assegniamo il numero a 1 cifra significativa 0,0000000000000006 a `y`.

```
x is exact: 0.12345678901234500    Stampato con 17 cifre decimali
y is exact: 0.00000000000000060
```

Ma aggiungiamoli insieme. Questo dovrebbe dare 0.1234567890123456 ma è più di `DBL_DIG`, quindi potrebbero accadere cose strane… guardiamo:

```
x + y non del tutto giusto: 0.12345678901234559    Dovrebbe finire in 4560!
```

Questo è ciò che otteniamo stampando più di `DBL_DIG` giusto? Ma guarda un po'… quel numero sopra è esattamente rappresentabile così com'è!

se assegniamo `0.12345678901234559` (17 cifre) a `z` e lo stampiamo otteniamo:

```
z è esatto: 0.12345678901234559   17 cifre corrette! Più di DBL_DIG!
```

Se avessimo troncato `z` a 15 cifre non sarebbe stato lo stesso numero. Ecco perché per preservare tutti i bit di un `double` abbiamo bisogno di `DBL_DECIMAL_DIG` e non solo del minore `DBL_DIG`.

Detto questo è chiaro che quando si scherza con i numeri decimali in generale non è sicuro stampare più di cifre `FLT_DIG`, `DBL_DIG` o `LDBL_DIG` per essere sensati in relazione ai numeri in base 10 originali e a qualsiasi matematica successiva.

Ma quando si converte da `float` a rappresentazione decimale e di *nuovo* a `float`, utilizzare sicuramente `FLT_DECIMAL_DIG` per fare in modo che tutti i bit vengano conservati esattamente.

14.5. Tipi numerici costanti

Quando scrivi un numero costante come `1234` ha un tipo. Ma che tipo è? Diamo un'occhiata a come C decide di che tipo è la costante e come forzarla a scegliere un tipo specifico.

14.5.1. Esadecimale e ottale

Oltre al buon vecchio decimale come quello che faceva la nonna C supporta anche costanti di basi diverse.

Se inserisci un numero con `0x` viene letto come un numero esadecimale:

```
int a = 0x1A2B;
// Esadecimale
int b = 0x1a2b;
// Le maiuscole e minuscole non hanno importanza per le cifre esadecimali

printf("%x", a);
// Stampa un numero esadecimale, "1a2b"
```

Se si precede un numero con uno 0, viene letto come un numero ottale:

```
int a = 012;

printf("%o\n", a);
// Stampa un numero ottale, "12"
```

Ciò è particolarmente problematico per i programmatori principianti che cercano di riempire i numeri decimali a sinistra con 0 per allineare le cose in modo carino e carino cambiando inavvertitamente la base del numero:

```
int x = 11111;
// Decimale 11111
int y = 00111;
// Decimale 73 (Ottale 111)
int z = 01111;
// Decimale 585 (Ottale 1111)
```

1. Una nota sui binario

Un'estensione non ufficiale[108] in molti compilatori C consente di rappresentare un numero binario

108È davvero sorprendente per me che C non lo abbia ancora nelle specifiche. Nel documento C99 Rational scrivono:

con prefisso `0b`:

```
int x = 0b101010;    // Binario 101010

printf("%d\n", x);   // Stampa 42 decimali
```

Non esiste un identificatore di formato `printf()` per stampare un numero binario. Devi farlo un carattere alla volta con operatori bit a bit.

14.5.2. Costanti intere

È possibile forzare un intero costante ad essere di un certo tipo aggiungendovi un suffisso che indichi il tipo.

Faremo alcuni compiti per la demo ma molto spesso gli sviluppatori tralasciano i suffissi a meno che non sia necessario per essere precisi. Il compilatore è piuttosto bravo a garantire che i tipi siano compatibili.

```
int           x = 1234;
long int      x = 1234L;
long long int x = 1234LL

unsigned int           x = 1234U;
unsigned long int      x = 1234UL;
unsigned long long int x = 1234ULL;
```

Il suffisso può essere maiuscolo o minuscolo. E `U` e `L` o `LL` possono apparire per primi.

Tipo	Suffisso
int	None
long int	L
long long int	LL
unsigned int	U
unsigned long int	UL
unsigned long long int	ULL

Ho menzionato nella tabella che “nessun suffisso” significa `int` … ma in realtà è più complesso di così.

Quindi cosa succede quando hai un numero senza suffisso come:

```
int x = 1234;
```

Di che tipo è?

Ciò che C generalmente farà è scegliere il tipo più piccolo da sopra `int` che può contenere il

> "Una proposta per aggiungere costanti binarie è stata respinta a causa della mancanza di precedenti e di un'utilità insufficiente". Il che sembra un po' sciocco alla luce di alcune delle altre caratteristiche che hanno messo lì dentro! Scommetto che una delle prossime uscite ce l'ha.

valore.

Ma nello specifico ciò dipende anche dalla base del numero (decimale, esadecimale o ottale).

Le specifiche hanno un'ottima tabella che indica quale tipo viene utilizzato per quale valore senza suffisso. In effetti lo copierò all'ingrosso proprio qui.

C11 §6.4.4.1¶5 recita: “Il tipo di una costante intera è il primo della prima lista corrispondente in cui il suo valore può essere rappresentato”.

E poi continua mostrando questa tabella:

Suffisso	Costante decimale	ottale o esadecimale Costante
niente	int	int
	long int	unsigned int
		long int
		unsigned long int
		long long int
		unsigned long long int
`u` o `U`	unsigned int	unsigned int
	unsigned long int	unsigned long int
	unsigned long long int	unsigned long long int
`l` o `L`	long int	long int
	long long int	unsigned long int
		long long int
		unsigned long long int
Entrambi `u` o `U`	unsigned long int	unsigned long int
e `l` o `L`	unsigned long long int	unsigned long long int
`ll` o `LL`	long long int	long long int
		unsigned long long int
Entrambi `u` o `U`	unsigned long long int	unsigned long long int

e `ll` o `LL`

Ciò che significa è che ad esempio se specifichi un numero come `123456789U`, prima C vedrà se può essere `unsigned int`. Se non ci sta, proverà un `unsigned long int`. Poi unsigned `long long int`. Utilizzerà il tipo più piccolo che può contenere il numero.

14.5.3. Costanti in virgola mobile

Penseresti che una costante in virgola mobile come `1.23` avrebbe un tipo `float` predefinito giusto?

Sorpresa! Risulta che i numeri in virgola mobile senza suffisso sono di tipo `double`! Buon compleanno in ritardo!

Puoi forzarlo ad essere di tipo `float` aggiungendo una `f` (o `F`: non fa distinzione tra maiuscole e minuscole). Puoi forzarlo ad essere di tipo `long double` aggiungendo `l` (o `L`).

Tipo	Suffisso
float	F
double	Niente
long double	L

Per esempio:

```
float x       = 3.14f;
double x      = 3.14;
long double x = 3.14L;
```

Per tutto questo tempo però abbiamo fatto solo questo giusto?

```
float x = 3.14;
```

Quello a sinistra non è un `float` e la destra un `double`? Sì! Ma il C è abbastanza buono con le conversioni numeriche automatiche quindi è più comune avere una costante in virgola mobile senza suffisso che senza. Ne parleremo più avanti.

1. Notazione scientifica

Ricordi prima quando abbiamo parlato di come un numero in virgola mobile può essere rappresentato da un significantivo una base e un esponente?

Bene, esiste un modo comune di scrivere un numero del genere mostrato qui seguito dal suo equivalente più riconoscibile che è quello che ottieni quando esegui effettivamente i calcoli:

`$$1.2345\times10^3 = 1234.5$$`

Scrivere numeri nella forma `$s\times b^e$` si chiama notazione scientifica[109] . In C questi sono scritti usando la “notazione E” quindi sono equivalenti:

109https://en.wikipedia.org/wiki/Scientific_notation

Notazione scientifica	Notazione E
`$1.2345\times10^{-3}=0.0012345$`	`1.2345e-3`
`$1.2345\times10^8=123450000$`	`1.2345e+8`

Puoi stampare un numero in questa notazione con `%e`:

```
printf("%e\n", 123456.0);
// Stampa 1.234560e+05
```

Un paio di piccole curiosità sulla notazione scientifica:

1. Non è necessario scriverli con una sola cifra iniziale prima del punto decimale. Qualsiasi numero di numeri può andare davanti.

```
double x = 123.456e+3;
// 123456
```

Tuttavia quando lo stampi l'esponente cambierà quindi c'è solo una cifra davanti al punto decimale.

- Il più può essere lasciato fuori dall'esponente poiché è predefinito, ma questo è raro nella pratica da quello che ho visto.

```
1.2345e10 == 1.2345e+10
```

- È possibile applicare i suffissi `F` o `L` alle costanti della notazione E:

```
1.2345e10F
1.2345e10L
```

2. Costanti esadecimali in virgola mobile

Ma aspetta, c'è ancora molto da fare!

Si scopre che ci sono anche costanti esadecimali in virgola mobile!

Funzionano in modo simile ai numeri in virgola mobile decimali ma iniziano con `0x` proprio come i numeri interi.

Il problema è che *devi* specificare un esponente e questo esponente produce una potenza di 2. Cioè: `$2^x$`.

E poi usi un `p` invece di `e` quando scrivi il numero:

Quindi `0xa.1p3` è `$10.0625\times2^3 == 80.5$`.

Quando si utilizzano costanti esadecimali in virgola mobile possiamo stampare la notazione scientifica esadecimale con `%a`:

```
double x = 0xa.1p3;

printf("%a\n", x);
// 0x1.42p+6
printf("%f\n", x);
// 80.500000
```

15. Tipi III: conversioni

In questo capitolo vogliamo parlare della conversione da un tipo all'altro. C ha una varietà di modi per farlo e alcuni potrebbero essere leggermente diversi da quelli a cui sei abituato in altri linguaggi.

Prima di parlare di come fare le conversioni parliamo di come funzionano quando *si* verificano.

15.1. Conversioni di stringhe

A differenza di molti linguaggi il C non esegue conversioni da stringa a numero (e viceversa) in modo così semplice come avviene con le conversioni numeriche.

Per questi dovremo chiamare funzioni per fare il lavoro sporco.

15.1.1. Valore numerico a stringa

Quando vogliamo convertire un numero in una stringa possiamo usarne uno `sprintf()` (pronunciato *SPRINT-f*) o `snprintf()` (*s-n-print-f*)[110]

Fondamentalmente funzionano come `printf()` tranne per il fatto che vengono invece restituiti in una stringa e puoi stampare quella stringa in seguito o qualsiasi altra cosa.

Ad esempio trasformando parte del valore `$\pi$` in una stringa:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    float f = 3.14159;

    // Converti "f" in stringa, memorizzandola in "s", scrivendo al massimo 10
caratteri
    // compreso il terminatore NUL

    snprintf(s, 10, "%f", f);

    printf("String value: %s\n", s);  // Valore stringa: 3.141590
}
```

Quindi puoi usare %d o %u come sei abituato per i numeri interi.

15.1.2. Da stringa a valore numerico

Ci sono un paio di famiglie di funzioni per farlo in C. Le chiameremo famiglia `atoi` (pronunciato *a-to-i*) e famiglia `strtol` *(stir-to-long)*.

Per una conversione di base da una stringa a un numero prova le funzioni `atoi` da `<stdlib.h>`. Questi hanno pessime caratteristiche di gestione degli errori (incluso un comportamento indefinito se si passa una stringa errata) quindi usali con attenzione.

Funzione	Descrizione
atoi	Stringa a int

110Sono uguali tranne che `snprintf()` ti consente di specificare un numero massimo di byte da inviare in output impedendo il superamento della fine della stringa. Quindi è più sicuro.

atof	Stringa a float
atol	Stringa a long int
atoll	Stringa a long long int

Sebbene le specifiche non lo coprano la a all'inizio della funzione sta per ASCII[111] quindi in realtà `atoi()` è "ASCII-to-integer" ma dirlo oggi è un po' ASCII-centrico.

Ecco un esempio di conversione di una stringa in un `float`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *pi = "3.14159";
    float f;

    f = atof(pi);

    printf("%f\n", f);
}
```

Ma come ho detto otteniamo comportamenti indefiniti da cose strane come questa:

```
int x = atoi("what");
// "Cosa" non è un numero
// che abbia mai sentito nominare
```

(Quando lo eseguo ottengo indietro 0 ma non dovresti farci affidamento. Potresti ottenere qualcosa di completamente diverso.)

Per migliori caratteristiche di gestione degli errori diamo un'occhiata a tutte quelle funzioni `strtol` anche in `<stdlib.h>`. Non solo ma si convertono anche in più tipi e più basi!

Funzione	Descrizione
strtol	Stringa a long int
strtoll	Stringa a long long int
strtoul	Stringa a unsigned long int
strtoull	Stringa a unsigned long long int
strtof	Stringa a float
strtod	Stringa a double
strtold	Stringa a long double

111https://en.wikipedia.org/wiki/ASCII

Queste funzioni seguono tutte uno schema di utilizzo simile e rappresentano la prima esperienza di molte persone con i puntatori a puntatori! Ma non preoccuparti: è più facile di quanto sembri.

Facciamo un esempio in cui convertiamo una stringa in un `unsigned long` scartando le informazioni sull'errore (ad esempio le informazioni sui caratteri errati nella stringa di input):

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "3490";

    // Converte la stringa s,
    // un numero in base 10,
    // in un int lungo senza segno.
    // NULL significa che non ci
    // interessa conoscere
    // alcuna informazione sull'errore.

    unsigned long int x = strtoul(s, NULL, 10);

    printf("%lu\n", x);  // 3490
}
```

Nota un paio di cose lì. Anche se non ci siamo degnati di catturare alcuna informazione sui caratteri di errore nella stringa `strtoul()` non ci darà un comportamento indefinito; restituirà semplicemente 0.

Inoltre abbiamo specificato che si tratta di un numero decimale (base 10).

Questo significa che possiamo convertire numeri di basi diverse? Sicuro! Facciamo il binario!

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "101010";
    // Qual è il significato di questo numero?

    // Converte la stringa s, un numero
    // in base 2, in un int lungo senza segno.

    unsigned long int x = strtoul(s, NULL, 2);

    printf("%lu\n", x);  // 42
}
```

OK questo è tutto divertimento e gioco ma cosa significa quel `NULL` lì dentro? A cosa serve?

Questo ci aiuta a capire se si è verificato un errore nell'elaborazione della stringa. È un puntatore a un puntatore a un `char` il che sembra spaventoso ma non lo è una volta che ci si capisce.

Facciamo un esempio in cui inseriamo un numero deliberatamente errato e vedremo come `strtol()` ci fa sapere dove si trova la prima cifra non valida.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
int main(void)
{
    char *s = "34x90";
    // "x" non è una cifra valida in base 10!
    char *badchar;

    // Converte la stringa s,
    // un numero in base 10,
    // in un unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Cerca di convertire
    // il più possibile,
    // quindi arriva fin qui:

    printf("%lu\n", x);  // 34

    // Ma possiamo vedere
    // il cattivo carattere
    // offensivo a causa di badchar
    // lo punta!

    printf("Invalid character: %c\n", *badchar);  // "x"
}
```

Quindi ecco che abbiamo `strtoul()` che modifica ciò a cui punta `badchar` per mostrarci dove le cose sono andate storte[112] .

Ma cosa succede se nulla va storto? In tal caso `badchar` punterà al terminatore `NUL` alla fine della stringa. Quindi possiamo testarlo:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *s = "3490";
    // "x" non è una cifra valida in base 10!
    char *badchar;

    // Converti la stringa s in
    // un numero base 10 in
    // un unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Controlla se le cose sono andate bene

    if (*badchar == '\0') {
        printf("Successo! %lu\n", x);
    } else  {
        printf("Conversione parziale: %lu\n", x);
        printf("Carattere non valido: %c\n", *badchar);
    }
}
```

112Dobbiamo passare un puntatore a `badchar` a `strtoul()` altrimenti non sarà in grado di modificarlo in nessun modo visibile analogamente al motivo per cui devi passare un puntatore a un int a una funzione se tu voglio che quella funzione sia in grado di cambiare quel valore di quell'int.

Così il gioco è fatto. Le funzioni in stile `atoi()` sono utili in un contesto controllato ma le funzioni in stile `strtol()` offrono un controllo molto maggiore sulla gestione degli errori e sulla base dell'input.

15.2. Conversioni char

Cosa succede se hai un singolo carattere con una cifra al suo interno come `'5'`… È uguale al valore 5?

Proviamolo e vediamo.

```
printf("%d %d\n", 5, '5');
```

Sul mio sistema UTF-8 stampa questo:

```
5 53
```

Quindi… no. E 53? Che cos'è? Questo è il punto di codice UTF-8 (e ASCII) per il simbolo del carattere `'5'`[113]

Allora come convertiamo il carattere `'5'` (che apparentemente ha valore 53) nel valore 5?

Con un trucco intelligente ecco come!

Lo standard C garantisce che questi caratteri avranno punti di codice che sono in sequenza e in questo ordine:

```
0  1  2  3  4  5  6  7  8  9
```

Rifletti per un secondo: come possiamo usarlo? Spoiler avanti…

Diamo un'occhiata ai caratteri e ai relativi punti di codice in UTF-8:

```
0  1  2  3  4  5  6  7  8  9
48 49 50 51 52 53 54 55 56 57
```

Vedi lì che `'5'` è 53, proprio come stavamo ottenendo. E `'0'` è 48.

Quindi possiamo sottrarre `'0'` da qualsiasi carattere cifra per ottenere il suo valore numerico:

```
char c = '6';

int x = c;  // x ha valore 54, il codice punta per '6'

int y = c - '0'; // y ha valore 6 proprio come vogliamo
```

E possiamo convertire anche il contrario semplicemente aggiungendo il valore sopra.

```
int x = 6;

char c = x + '0';  // c ha valore 54

printf("%d\n", c);  // stampa 54
printf("%c\n", c);  // stampa 6 con %c
```

Potresti pensare che questo sia un modo strano per eseguire questa conversione e per gli standard odierni certamente lo è. Ma ai vecchi tempi quando i computer erano fatti letteralmente di legno, questo era il metodo per eseguire questa conversione. E non era rotto quindi C non lo ha mai aggiustato.

113Ad ogni carattere è associato un valore per ogni dato schema di codifica dei caratteri.

15.3. Conversioni numeriche

15.3.1. Boolean

Se converti uno zero in `bool` il risultato è `0`. Altrimenti è `1`.

15.3.2. Conversioni da numero intero a numero intero

Se un tipo intero viene convertito in unsigned e non ci sta il risultato senza segno si incarta in stile contachilometri finché non si adatta a unsigned[114] .

Se un tipo intero viene convertito in un numero con segno e non si adatta, il risultato è definito dall'implementazione! Accadrà qualcosa di documentato ma dovrai cercarlo[115]

15.3.3. Conversioni di numeri interi e in virgola mobile

Se un tipo a virgola mobile viene convertito in un tipo intero la parte frazionaria viene scartata con pregiudizio[116] .

Ma—ed ecco il problema—se il numero è troppo grande per rientrare nell'intero si ottiene un comportamento indefinito. Quindi non farlo.

Passando da un numero intero o una virgola mobile a una virgola mobile C fa del suo meglio per trovare il numero in virgola mobile più vicino possibile all'intero.

Ancora una volta però se il valore originale non può essere rappresentato si tratta di un comportamento indefinito.

15.4. Conversioni implicite

Queste sono le conversioni che il compilatore esegue automaticamente per te quando mescoli e abbini i tipi.

15.4.1. Le promozioni intere

In molti posti se un `int` può essere utilizzato per rappresentare un valore da `char` o `short` (signed o unsigned) quel valore viene *promosso* fino a `int`. Se non rientra in un `int` viene promosso a `unsigned int`.

Ecco come possiamo fare qualcosa del genere:

```
char x = 10, y = 20;
int i = x + y;
```

In tal caso `x` e `y` vengono promossi a `int` da C prima che avvengano i calcoli.

Le promozioni degli interi avvengono durante le solite conversioni aritmetiche con funzioni variadiche[117] , operatori unari `+` e `-` o quando si passano valori a funzioni senza prototipi[118] .

114In pratica ciò che probabilmente accade nella tua implementazione è che i bit di ordine superiore vengono semplicemente eliminati dal risultato, quindi un numero a 16 bit `0x1234` convertito in un numero a 8 bit finisce come `0x0034` o semplicemente `0x34`.

115Ancora una volta in pratica ciò che probabilmente accadrà sul tuo sistema è che lo schema di bit dell'originale verrà troncato e quindi utilizzato solo per rappresentare il numero con segno complemento a due. Ad esempio il mio sistema prende il carattere senza segno 192 e lo converte nel carattere con segno -64. Nel complemento a due, lo schema di bit per entrambi questi numeri è binario `11000000`.

116Non proprio: viene semplicemente scartato regolarmente.

117Funzioni con un numero variabile di argomenti.

118Questo viene fatto raramente perché il compilatore si lamenterà e avere un prototipo è la cosa giusta da fare. Penso

15.4.2. Le solite conversioni aritmetiche

Queste sono conversioni automatiche che C esegue attorno alle operazioni numeriche richieste. (In realtà è così che vengono chiamati tra l'altro da C11 §6.3.1.8.) Notate che per questa sezione stiamo parlando solo di tipi numerici: le stringhe verranno fornite più avanti.

Queste conversioni rispondono a domande su cosa succede quando mescoli i tipi in questo modo:

```
int x = 3 + 1.2;   // Mescolando int e double
                   // 4.2 viene convertito in int
                   // 4 è memorizzato in x

float y = 12 * 2;  // Mescolando float e int
                   // 24 viene convertito in float
                   // 24.0 è memorizzato in y
```

Diventano `int`? Diventano `float`? Come funziona?

Ecco i passaggi parafrasati per un facile consumo.

1. Se un elemento nell'espressione è di tipo mobile converti gli altri elementi in quel tipo mobile.
1. Altrimenti se entrambi i tipi sono interi eseguire le promozioni di interi su ciascuno quindi rendere i tipi di operando grandi quanto è necessario per mantenere il valore più grande comune. A volte ciò comporta la modifica del segno in un segno senza segno.

Se vuoi conoscere i dettagli più concreti, consulta C11 §6.3.1.8. Ma probabilmente no.

In generale ricorda solo che i tipi int diventano tipi float se è presente un tipo in virgola mobile da qualche parte e il compilatore fa uno sforzo per assicurarsi che i tipi interi misti non trabocchino.

Infine se converti da un tipo a virgola mobile a un altro il compilatore proverà a effettuare una conversione esatta. Se non può farà la migliore approssimazione possibile. Se il numero è troppo grande per adattarsi al tipo in cui stai convertendo *boom*: comportamento indefinito!

15.4.3. `void*`

Il tipo `void*` è interessante perché può essere convertito da o in qualsiasi tipo di puntatore.

```
int x = 10;

void *p = &x;  // &x è tipo int*, ma lo memorizziamo in un void*

int *q = p;    // p è void*, ma lo memorizziamo in un file int*
```

15.5. Conversioni esplicite

Queste sono le conversioni da tipo a tipo che devi chiedere; il compilatore non lo farà per te.

Puoi convertire da un tipo a un altro assegnando un tipo a un altro con un `=`.

Puoi anche convertire esplicitamente con un *cast*.

15.5.1. Casting

Puoi modificare esplicitamente il tipo di un'espressione inserendo un nuovo tipo tra parentesi davanti ad essa. Alcuni sviluppatori C disapprovano questa pratica a meno che non sia

che questo funzioni ancora per ragioni storiche prima che i prototipi esistessero.

assolutamente necessaria; ma è probabile che ti imbatterai in del codice C che li contenga.

Facciamo un esempio in cui vogliamo convertire un `int` in un `long` in modo da poterlo archiviare in un `long`.

Nota: questo esempio è artificioso e il cast in questo caso è completamente inutile perché l'espressione `x + 12` verrebbe automaticamente modificata in `long int` per corrispondere al tipo più ampio di `y`.

```
int x = 10;
long int y = (long int)x + 12;
```

In quell'esempio anche se prima `x` era di tipo `int`, l'espressione `(long int)x` ha tipo `long int`. Si dice che “Castiamo `x` a `long int`.”

Più comunemente potresti vedere un cast utilizzato per convertire un `void*` in un tipo di puntatore specifico in modo che possa essere dereferenziato.

Un richiamo dalla funzione incorporata `qsort()` potrebbe mostrare questo comportamento poiché ha ricevuto `void*`:

```
int compar(const void *elem1, const void *elem2)
{
    if (*((const int*)elem2) > *((const int*)elem1)) return 1;
    if (*((const int*)elem2) < *((const int*)elem1)) return -1;
    return 0;
}
```

Ma potresti anche scriverlo chiaramente con un compito:

```
int compar(const void *elem1, const void *elem2)
{
    const int *e1 = elem1;
    const int *e2 = elem2;

    return *e2 - *e1;
}
```

Un posto in cui vedrai i cast più comunemente è quello di evitare un avviso quando si stampano i valori dei puntatori con `%p` usato raramente che diventa schizzinoso con qualsiasi cosa diversa da un `void*`:

```
int x = 3490;
int *p = &x;

printf("%p\n", p);
```

genera questo avviso:

```
warning: format ‘%p’ expects argument of type ‘void *’, but argument
         2 has type ‘int *’
```

Puoi risolverlo con un cast:

```
printf("%p\n", (void *)p);
```

Un altro posto è con modifiche esplicite del puntatore se non vuoi utilizzare un `void*` intermedio ma anche questi sono piuttosto rari:

```
long x = 3490;
long *p = &x;
unsigned char *c = (unsigned char *)p;
```

Un terzo posto in cui è spesso richiesto è con le funzioni di conversione dei caratteri in `<ctype.h>`[119] dove dovresti eseguire il cast di valori con segno discutibile su `unsigned char` per evitare comportamenti indefiniti.

Ancora una volta nella pratica il casting è raramente *necessario*. Se ti ritrovi a castare potrebbe esserci un altro modo per fare la stessa cosa o forse stai castando inutilmente.

O forse è necessario. Personalmente cerco di evitarlo ma non ho paura di usarlo se necessario.

16. Tipi IV: Qualificatori e Specificatori

Ora che abbiamo altri tipi al nostro bagaglio sembra che possiamo dare a questi tipi alcuni attributi aggiuntivi che controllano il loro comportamento. Questi sono i *qualificatori del tipo* e gli *specificatori della classe di archiviazione*.

16.1. Qualificatori di tipo

Questi ti permetteranno di dichiarare valori costanti e anche di fornire suggerimenti di ottimizzazione del compilatore che può utilizzare.

16.1.1. `const`

Questo è il qualificatore di tipo più comune che vedrai. Significa che la variabile è costante e qualsiasi tentativo di modificarla si tradurrà in un compilatore molto arrabbiato.

```
```

Non è possibile modificare un valore `const`.

Spesso vedi `const` negli elenchi di parametri per le funzioni:

```
const int x = 2;

x = 4;
// IL COMPILATORE EMETTE DEI SUONI,
// non è possibile assegnare a una costante
```

1. `const` e puntatori

Questo diventa un po' strano, perché ci sono due usi che hanno due significati quando si tratta di puntatori.

Per prima cosa possiamo fare in modo che tu non possa cambiare l'oggetto a cui punta il puntatore. Puoi farlo inserendo `const` in primo piano con il nome del tipo (prima dell'asterisco) nella dichiarazione del tipo.

```
int x[] = {10, 20};
const int *p = x;

p++;
// Possiamo modificare p, nessun problema

*p = 30;
// Errore di compilazione!
// Non è possibile
// modificare ciò a cui punta
```

In modo un po’ confuso queste due cose sono equivalenti:

119https://beej.us/guide/bgclr/html/split/ctype.html

```
const int *p;
// Non è possibile
// modificare ciò a cui punta p
int const *p;
// Non è possibile modificare
// ciò a cui punta p,
// proprio come la riga precedente
```

Ottimo quindi non possiamo cambiare l'oggetto a cui punta il puntatore, ma possiamo cambiare il puntatore stesso. E se volessimo il contrario? Vogliamo essere in grado di cambiare ciò a cui punta il puntatore ma non il puntatore stesso?

Basta spostare `const` dopo l'asterisco nella dichiarazione:

```
int *const p;
// Non possiamo modificare "p"
// con l'aritmetica dei puntatori

p++;
// Errore di compilazione!
```

Ma possiamo modificare ciò a cui indicano:

```
int x = 10;
int *const p = &x;

*p = 20;
// Imposta "x" su 20,
// nessun problema
```

Puoi anche rendere entrambe le cose `const`:

```
const int *const p;
// Impossibile modificare p o *p!
```

Infine se disponi di più livelli di riferimento indiretto, dovresti `const` i livelli adeguati. Solo perché un puntatore è `const` non significa che anche il puntatore a cui punta debba esserlo. Puoi impostarli esplicitamente come negli esempi seguenti:

```
char **p;
p++;     // OK!
(*p)++;  // OK!

char **const p;
p++;     // Errore!
(*p)++;  // OK!

char *const *p;
p++;     // OK!
(*p)++;  // Errore!

char *const *const p;
p++;     // Errore!
(*p)++;  // Errore!
```

2. `const` Correttezza

Un'altra cosa che devo menzionare è che il compilatore avviserà qualcosa del genere:

```
const int x = 20;
int *p = &x;
```

dire qualcosa del tipo:

```
initialization discards 'const' qualifier from pointer type target
```

Cosa sta succedendo lì?

Bene, dobbiamo esaminare i tipi su entrambi i lati dell'assegnazione:

```
    const int x = 20;
    int *p = &x;
//    ^       ^
//    |       |
//  int*    const int*
```

Il compilatore ci avverte che il valore a destra dell'assegnazione è `const` ma quello a sinistra no. E il compilatore ci sta facendo sapere che sta scartando la "cost" dell'espressione a destra. Cioè possiamo *ancora* provare a fare quanto segue, ma è semplicemente sbagliato.Il compilatore avviserà e il comportamento è indefinito:

```
const int x = 20;
int *p = &x;

*p = 40;
// Comportamento indefinito--forse
// modifica "x", forse no!

printf("%d\n", x);  // 40, se sei fortunato
```

16.1.2. `restrict`

TLDR: non devi mai usarlo e puoi ignorarlo ogni volta che lo vedi. Se lo usi correttamente probabilmente realizzerai un aumento delle prestazioni. Se lo usi in modo errato realizzerai un comportamento indefinito.

`restrict` è un suggerimento al compilatore che a un particolare pezzo di memoria potrà accedere solo un puntatore e mai un altro. (Cioè non ci sarà alcun aliasing dell'oggetto particolare a cui `restrict` puntatore punta.) Se uno sviluppatore dichiara che un puntatore è `restrict` e quindi accede all'oggetto a cui punta in un altro modo (ad esempio tramite un altro puntatore) il comportamento è indefinito. Fondamentalmente stai dicendo a C: "Ehi ti garantisco che questo singolo puntatore è l'unico modo per accedere a questa memoria e se sto mentendo, puoi restituirmi un comportamento indefinito".

E C utilizza tali informazioni per eseguire determinate ottimizzazioni. Ad esempio se stai dereferenziando il puntatore `restrict` ripetutamente in un ciclo C potrebbe decidere di memorizzare nella cache il risultato in un registro e memorizzare il risultato finale solo al termine del ciclo. Se qualsiasi altro puntatore si riferisse alla stessa memoria e vi accedesse nel ciclo i risultati non sarebbero accurati. (Si noti che la `restrict` non ha alcun effetto se l'oggetto puntato non viene mai scritto. Si tratta di ottimizzazioni che circondano le scritture in memoria.)

Scriviamo una funzione per scambiare due variabili e utilizziamo la parola chiave `restrict` per assicurare a C che non passeremo mai puntatori alla stessa cosa. E poi lasciamo perdere e proviamo a passare puntatori alla stessa cosa.

```
void swap(int *restrict a, int *restrict b)
{
    int t;

    t = *a;
```

```
    *a = *b;
    *b = t;
}

int main(void)
{
    int x = 10, y = 20;

    swap(&x, &y);  // OK! "a" e "b", sopra, puntano a cose diverse

    swap(&x, &x);  // Comportamento indefinito! "a" e "b" puntano alla stessa
cosa
}
```

Se dovessimo eliminare le parole chiave `restrict` sopra, ciò consentirebbe a entrambe le chiamate di funzionare in sicurezza. Ma in questo caso il compilatore potrebbe non essere in grado di ottimizzare. `restrict` ha un ambito di blocco, ovvero la restrizione dura solo per l'ambito in cui viene utilizzata. Se è in un elenco di parametri per una funzione è nell'ambito del blocco di quella funzione. Se il puntatore limitato punta a un array si applica solo ai singoli oggetti nell'array. Altri puntatori potrebbero leggere e scrivere dall'array purché non leggano o scrivano nessuno degli stessi elementi di quello limitato. Se è esterno a qualsiasi funzione nell'ambito del file, la restrizione copre l'intero programma.

Probabilmente lo vedrai nelle funzioni di libreria come `printf()`:

```
int printf(const char * restrict format, ...);
```

Ancora una volta si tratta semplicemente di dire al compilatore che all'interno della funzione `printf()` ci sarà solo un puntatore che fa riferimento a qualsiasi parte di quel `format` stringa.

Un'ultima nota: se per qualche motivo stai utilizzando la notazione di array nel parametro della funzione invece della notazione del puntatore puoi utilizzare la `restrict` in questo modo:

```
void foo(int p[restrict])
// Senza dimensione

void foo(int p[restrict 10])
// Oppure con una dimensione
```

Ma la notazione del puntatore sarebbe più comune.

16.1.3. `volatile`

È improbabile che tu lo veda o ne abbia bisogno a meno che tu non abbia a che fare direttamente con l'hardware.

`volatile` dice al compilatore che un valore potrebbe cambiare dietro le quinte e dovrebbe essere cercato ogni volta.

Un esempio potrebbe essere il caso in cui il compilatore cerca in memoria un indirizzo che si aggiorna continuamente dietro le quinte, ad esempio una sorta di timer hardware.

Se il compilatore decide di ottimizzarlo e di memorizzare il valore in un registro per un periodo prolungato, il valore in memoria verrà aggiornato e non si rifletterà nel registro.

Dichiarando qualcosa di `volatile` stai dicendo al compilatore: "Ehi, ciò a cui punta potrebbe cambiare in qualsiasi momento per ragioni esterne a questo al codice di questo programma".

```
volatile int *p;
```

16.1.4. `_Atomic`

Questa è una funzionalità opzionale del C di cui parleremo nel capitolo Atomica.

16.2. Specificatori della classe di archiviazione

Gli specificatori della classe di archiviazione sono simili ai quantificatori del tipo. Forniscono al compilatore maggiori informazioni sul tipo di una variabile.

16.2.1. `auto`

Questa parola chiave non viene quasi mai vista, poiché `auto` è l'impostazione predefinita per le variabili con ambito blocco. È implicito.

Questi sono gli stessi:

```
{
    int a;
// auto è l'impostazione predefinita...
    auto int a;
// Quindi questo è ridondante
}
```

La parola chiave `auto` indica che questo oggetto ha una *durata di archiviazione automatica*. Esiste cioè nell'ambito in cui è definito e viene deallocato automaticamente quando si esce dall'ambito. Un problema sulle variabili automatiche è che il loro valore è indeterminato finché non le inizializzi esplicitamente. Diciamo che sono pieni di dati “casuali” o “spazzatura”, anche se nessuno di questi mi rende davvero felice. In ogni caso non saprai cosa c'è dentro a meno che non lo inizializzi.

Inizializzare sempre tutte le variabili automatiche prima dell'uso!

16.2.2. `static`

Questa parola chiave ha due significati a seconda che la variabile abbia un ambito file o un ambito blocco.

Cominciamo con l'ambito del blocco.

1. `static` nell'ambito del blocco

In questo caso stiamo sostanzialmente dicendo: "Voglio solo che esista una singola istanza di questa variabile condivisa tra le chiamate".

Cioè il suo valore persisterà tra le chiamate.

`static` nell'ambito del blocco con un inizializzatore verrà inizializzato solo una volta all'avvio del programma, non ogni volta che viene chiamata la funzione.

Facciamo un esempio:

```
#include <stdio.h>

void counter(void)
{
    static int count = 1;  // Questo viene inizializzato una volta

    printf("Questo è stato chiamato %d volt(e)\n", count);

    count++;
```

```
}

int main(void)
{
    counter();  // "Questo è stato chiamato 1 volt(e)"
    counter();  // "Questo è stato chiamato 2 volt(e)"
    counter();  // "Questo è stato chiamato 3 volt(e)"
    counter();  // "Questo è stato chiamato 4 volt(e)"
}
```

Vedi come il valore di `count` persiste tra le chiamate?

Una cosa degna di nota è che le variabili dell'ambito del blocco `static` sono inizializzate su 0 per impostazione predefinita.

```
static int foo;      // Il valore iniziale predefinito è `0`...
static int foo = 0;  // Quindi l'assegnazione "0" è ridondante
```

Infine, tieni presente che se stai scrivendo programmi multithread devi essere sicuro di non lasciare che più thread calpestino la stessa variabile.

2. static nell'ambito del file

Quando si entra nell'ambito del file al di fuori di qualsiasi blocco, il significato cambia piuttosto.

Le variabili nell'ambito del file persistono già tra le chiamate di funzione, quindi quel comportamento è già presente.

Invece ciò che statico significa in questo contesto che quella variabile non è visibile al di fuori di questo particolare file sorgente. Come se fosse “globale” ma solo in questo file. Maggiori informazioni nella sezione sulla costruzione con più file sorgente.

16.2.3. extern

Il `extern` identificatore della classe di archiviazione ci fornisce un modo per fare riferimento a oggetti in altri file sorgente.

Supponiamo, ad esempio, che il file `bar.c` contenga nella sua interezza quanto segue:

```
// bar.c

int a = 37;
```

Solo quello. Dichiarare un nuovo `int` a nell'ambito del file.

Ma cosa succederebbe se avessimo un altro file sorgente, `foo.c` e volessimo fare riferimento a `a` che è in `bar.c`?

È facile con la parola chiave `extern`:

```
// foo.c

extern int a;

int main(void)
{
    printf("%d\n", a);  // 37, da bar.c!

    a = 99;

    printf("%d\n", a);  // Stessa "a" di bar.c, ma è adesso 99
}
```

Avremmo potuto anche creare `extern int a` nell'ambito del blocco e si sarebbe comunque riferito a `a` in `bar.c`:

```
// foo.c

int main(void)
{
    extern int a;

    printf("%d\n", a);  // 37, da bar.c!

    a = 99;

    printf("%d\n", a);  // Stessa "a" da bar.c, ma adesso è 99
}
```

Ora se a in `bar.c` fosse stato contrassegnato come `static`. questo non avrebbe funzionato. Le variabili `static` nell'ambito del file non sono visibili all'esterno di quel file.

Un'ultima nota su `extern` sulle funzioni. Per le funzioni `extern` è l'impostazione predefinita, quindi è ridondante. Puoi dichiarare una funzione `static` se vuoi che sia visibile solo in un singolo file sorgente.

16.2.4. `register`

Questa è una parola chiave per suggerire al compilatore che questa variabile viene utilizzata di frequente e che dovrebbe essere resa il più veloce possibile per accedervi. Il compilatore non ha alcun obbligo di accettarlo.

Ora i moderni ottimizzatori del compilatore C sono piuttosto efficaci nel capirlo da soli, quindi è raro vederlo al giorno d'oggi.

Ma se devi:

```
#include <stdio.h>

int main(void)
{
    register int a;
// Rendi "a" il più veloce possibile da usare.

    for (a = 0; a < 10; a++)
        printf("%d\n", a);
}
```

Tuttavia ha un prezzo. Non puoi prendere l'indirizzo di una registro:

```
register int a;
int *p = &a;
// ERRORE DI COMPILAZIONE!
// Impossibile prendere l'indirizzo di un registro
```

Lo stesso vale per qualsiasi parte di un array:

```
register int a[] = {11, 22, 33, 44, 55};
int *p = a;
// ERRORE DI COMPILAZIONE!
// Impossibile prendere l'indirizzo a[0]
```

O dereferenziare parte di un array:

```
register int a[] = {11, 22, 33, 44, 55};

int a = *(a + 2);
// ERRORE DI COMPILAZIONE! Indirizzo di a[0] preso
```

È interessante notare che, per l'equivalente con la notazione di array, gcc avvisa solo:

```
register int a[] = {11, 22, 33, 44, 55};

int a = a[2];  // AVVISO DEL COMPILATORE!
```

con:

```
warning: ISO C forbids subscripting ‘register’ array
```

Dato che non è possibile prendere l'indirizzo di una variabile di registro, libera il compilatore dalla possibilità di apportare ottimizzazioni tenendo conto di questo presupposto se non le ha già individuate. Inoltre l'aggiunta di `register` a una variabile `const` impedisce di passare accidentalmente il suo puntatore a un'altra funzione che ignora volontariamente la sua costanza[120] .

Un po' di retroscena storico qui: nel profondo della CPU ci sono piccole "variabili" dedicate chiamate registri[121] . Sono super veloci da accedere rispetto alla RAM quindi usarli ti dà un aumento di velocità. Ma non sono nella RAM, quindi non hanno un indirizzo di memoria associato(ecco perché non puoi prendere l'indirizzo di o ottenere un puntatore ad essi).

Ma come ho detto, i compilatori moderni sono davvero bravi a produrre codice ottimale utilizzando i registri quando possibile indipendentemente dal fatto che tu abbia specificato o meno la parola chiave `register`. Non solo, ma le specifiche consentono loro di trattarlo semplicemente come se avessi digitato `auto` se lo desiderano. Quindi nessuna garanzia.

16.2.5. `_Thread_local`

Quando utilizzi più thread e hai alcune variabili nell'ambito del blocco globale o `statico` questo è un modo per assicurarti che ogni thread ottenga la propria copia della variabile. Questo ti aiuterà a evitare condizioni di gara e discussioni o che si pestano i piedi a vicenda.

Se sei nell'ambito del blocco devi usarlo insieme a `extern` o `static`.

Inoltre se includi `<threads.h>` puoi usare il più appetibile `thread_local` come alias per il più brutto `_Thread_local`.

17. Progetti multifile

Finora abbiamo esaminato programmi giocattolo che per la maggior parte stanno in un unico file. Ma i programmi C complessi sono costituiti da molti file che vengono tutti compilati e collegati insieme in un unico eseguibile.

In questo capitolo esamineremo alcuni dei modelli e delle pratiche comuni per mettere insieme progetti più ampi.

17.1. Include e prototipi di funzioni

Una situazione molto comune è che alcune delle tue funzioni sono definite in un file e vuoi chiamarle da un altro.

In realtà funziona immediatamente con un avviso… proviamolo prima e poi vediamo il modo giusto

120https://gustedt.wordpress.com/2010/08/17/a-common-misconsception-the-register-keyword/
121https://en.wikipedia.org/wiki/Processor_register

per correggere l'avviso.

Per questi esempi inseriremo il nome del file come primo commento nell'origine.

Per compilarli dovrai specificare tutti i sorgenti sulla riga di comando:

```
# output file   source files
#     v            v
#   |----| |---------|
gcc -o foo foo.c bar.c
```

In questi esempi `foo.c` e `bar.c` vengono integrati nell'eseguibile denominato `foo`.

Diamo quindi un'occhiata al file sorgente `bar.c`:

```
// File bar.c

int add(int x, int y)
{
    return x + y;
}
```

E il file `foo.c` con main al suo interno:

```
// File foo.c

#include <stdio.h>

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Guarda come da `main()` chiamiamo `add()` ma `add()` si trova in un file sorgente completamente diverso! È in `bar.c`, mentre la chiamata è in `foo.c`!

Se lo compiliamo con:

```
gcc -o foo foo.c bar.c
```

otteniamo questo errore:

```
error: implicit declaration of function 'add' is invalid in C99
```

(Oppure potresti ricevere un avviso. Che non dovresti ignorare. Non ignorare mai gli avvisi in C; affrontali tutti.)

Se ricordi la sezione sui prototipi le dichiarazioni implicite sono vietate nel C moderno e non c'è motivo legittimo per introdurle nel nuovo codice. Dovremmo aggiustarlo.

Ciò che la `dichiarazione implicita` significa è che stiamo usando una funzione in questo caso `add()`, senza che C ne sappia nulla in anticipo. C vuole sapere cosa restituisce quali tipi accetta come argomenti e cose del genere.

Abbiamo visto come risolvere il problema in precedenza con un *prototipo di funzione*.Infatti se ne aggiungiamo uno a `foo.c` prima di effettuare la chiamata tutto funziona bene:

```
// File foo.c

#include <stdio.h>

int add(int, int);  // Aggiungi il prototipo
```

```
int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Niente più errori!

Ma è una seccatura: dover digitare il prototipo ogni volta che si desidera utilizzare una funzione. Voglio dire abbiamo usato `printf()` proprio lì e non abbiamo avuto bisogno di digitare un prototipo; cosa dà?

Se ricordi da cosa restituisce con `hello.c` all'inizio del libro *in realtà abbiamo incluso il prototipo per `printf()`!* È nel file `stdio.h`! E lo abbiamo incluso con `#include`!

Possiamo fare lo stesso con la nostra funzione `add()`? Realizzarne un prototipo e inserirlo in un file di intestazione?

Certamente!

I file di intestazione in C hanno un'estensione `.h` per impostazione predefinita. E spesso, ma non sempre, hanno lo stesso nome del file `.c` corrispondente. Quindi creiamo un file `bar.h` per il nostro file `bar.c` e inseriamo al suo interno il prototipo:

```
// File bar.h

int add(int, int);
```

E ora modifichiamo `foo.c` per includere quel file. Supponendo che sia nella stessa cartella lo includiamo tra virgolette doppie (invece che tra parentesi angolari):

```
// File foo.c

#include <stdio.h>

#include "bar.h"
// Includi dalla directory corrente

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Notate come non abbiamo più il prototipo in `foo.c` —lo abbiamo incluso da `bar.h`. Ora *ogni* file che desideri quella funzionalità `add()` può semplicemente `#include "bar.h"` per ottenerlo e non devi preoccuparti di digitare il prototipo della funzione.

Come avrai intuito, `#include` include letteralmente il file con nome *proprio lì* nel tuo codice sorgente proprio come se lo avessi digitato.

E compilandolo e eseguendolo:

```
./foo
5
```

In effetti otteniamo il risultato di `$2+3$`! Sìì!

Ma non aprire ancora la tua bevanda preferita. Ci siamo quasi! C'è solo un altro tassello che dobbiamo aggiungere.

17.2. Gestire le inclusioni ripetute

Non è raro che un file header `#include` altre intestazioni necessarie per le funzionalità dei suoi file C corrispondenti. Voglio dire, perché no?

E potrebbe darsi che tu abbia un'intestazione `#include` più volte da luoghi diversi. Forse non è un problema ma forse potrebbe causare errori del compilatore. E non possiamo controllare quanti posti `#include` c'è!

Ancora peggio potremmo ritrovarci in una situazione assurda in cui l'intestazione `a.h` include l'intestazione `b.h` e `b.h` include `a.h`! È un ciclo infinito `#include`! Provare a compilare una cosa del genere dà un errore:

```
error: #include nested depth 200 exceeds maximum of 200
```

Ciò che dobbiamo fare è fare in modo che se un file viene incluso una volta, i successivi `#include` per quel file vengano ignorati.

Le cose che stiamo per fare sono così comuni che dovresti farlo automaticamente ogni volta che crei un file di intestazione!

E il modo comune per farlo è con una variabile del preprocessore che impostiamo la prima volta che `#include` il file. Quindi per i successivi `#include` controlliamo prima per assicurarci che la variabile non sia definita.

Per dare un nome di variabile è molto comune prendere il nome del file di intestazione come `bar.h`, renderlo maiuscolo e sostituire il punto con un carattere di sottolineatura: `BAR_H`.

Quindi un controllo nella parte superiore del file dove vedi se è già stato incluso e commenta l'intera cosa se lo è.

(Non inserire un carattere di sottolineatura iniziale (perché è riservato un carattere di sottolineatura seguito da una lettera maiuscola) o un doppio carattere di sottolineatura iniziale (perché anche quello è riservato.))

```
#ifndef BAR_H
// Se BAR_H non è definito...
#define BAR_H
// Definiscilo (senza particolare valore)

// File bar.h

int add(int, int);

#endif          // Fine del #ifndef BAR_H
```

Ciò farà sì che il file di intestazione venga incluso solo una volta indipendentemente da quanti posti tentano di `#include` rlo.

17.3. `static` e `extern`

Quando si tratta di progetti multifile puoi assicurarti che le variabili e le funzioni dell'ambito file non siano visibili da altri file sorgente con la parola chiave `static`.

E puoi fare riferimento a oggetti in altri file con `extern`. Per ulteriori informazioni consulta le sezioni del libro sugli specificatori di classe di archiviazione `static` ed `extern`.

17.4. Compilazione con file oggetto

Questo non fa parte delle specifiche ma è comune al 99,999% nel mondo C.

È possibile compilare file C in una rappresentazione intermedia denominata *file oggetto*. Si tratta di codice macchina compilato che non è stato ancora inserito in un eseguibile.

I file oggetto in Windows hanno un'estensione `.OBJ`; in sistemi Unix-like, sono `.o`.

In gcc possiamo crearne alcuni come questo con il flag -c (solo compilazione!):

```
gcc -c foo.c     # produce foo.o
gcc -c bar.c     # produce bar.o
```

E poi possiamo *collegarli* insieme in un unico eseguibile:

```
gcc -o foo foo.o bar.o
```

Voilà abbiamo prodotto un eseguibile `foo` dai due file oggetto.

Ma stai pensando perché preoccuparsi? Non possiamo semplicemente:

```
gcc -o foo foo.c bar.c
```

e uccidere due boids[122] con una fava?

Per i piccoli programmi va bene. Lo faccio sempre.

Ma per i programmi più grandi possiamo trarre vantaggio dal fatto che la compilazione dai file sorgente a quelli oggetto è relativamente lenta e il collegamento di un gruppo di file oggetto è relativamente veloce.

Questo si vede davvero con l'utilità `make` che ricostruisce solo le fonti che sono più recenti dei loro output.

Diciamo che hai mille file C. Potresti compilarli tutti in file oggetto da avviare (lentamente) e quindi combinare tutti quei file oggetto in un eseguibile (velocemente).

Ora supponiamo che tu abbia modificato solo uno di quei file sorgente C— ecco la magia: *Devi solo ricostruire quel file oggetto per quel file sorgente*! E poi ricostruisci l'eseguibile (veloce). Tutti gli altri file C non devono essere toccati.

In altre parole ricostruendo solo i file oggetto di cui abbiamo bisogno riduciamo radicalmente i tempi di compilazione. (A meno che ovviamente, non si stia eseguendo una build "pulita" nel qual caso è necessario creare tutti i file oggetto.)

18. L'ambiente esterno

Quando esegui un programma in realtà dici alla shell: "Ehi, per favore esegui questa cosa". E la shell dice "Certo" e poi dice al sistema operativo "Ehi, potresti creare un nuovo processo ed eseguire questa cosa?" E se tutto va bene il sistema operativo è conforme e il programma viene eseguito.

Ma c'è un intero mondo al di fuori dei programma nella shell con cui è possibile interagire dall'interno di C. Ne vedremo alcuni in questo capitolo.

18.1. Argomenti della riga di comando

Molte utilità della riga di comando accettano *input della riga di comando*. Ad esempio se vogliamo

122https://en.wikipedia.org/wiki/Boids

vedere tutti i file che terminano con `.txt` possiamo scrivere qualcosa di questo tipo su un sistema simile a Unix:

```
ls *.txt
```

(o `dir` invece di `ls` su un sistema Windows).

In questo caso il comando è `ls` ma gli argomenti sono tutti i file che terminano con `.txt`[123] .

Allora come possiamo vedere cosa viene passato al programma dalla riga di comando?

Supponiamo di avere un programma chiamato `add` che somma tutti i numeri passati sulla riga di comando e stampiamo il risultato:

```
./add 10 30 5
45
```

Questo ripagherà sicuramente tutti gli sforzi!

Seriamente! Questo è un ottimo strumento per vedere come ottenere quegli argomenti dalla riga di comando e scomporli.

Per prima cosa vediamo come ottenerli. Per questo avremo bisogno di un nuovo `main()`!

Ecco un programma che stampa tutti gli argomenti della riga di comando. Ad esempio se chiamiamo l'eseguibile `foo` possiamo eseguirlo in questo modo:

```
./foo i like turtles
```

e vedremo questo output:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

È un po' strano perché l'argomento zero è il nome stesso dell'eseguibile. Ma è solo qualcosa a cui abituarsi. Gli argomenti stessi seguono direttamente.

Source:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

Whoa! Cosa sta succedendo nella dichiarazione della funzione `main()`? Cosa sono `argc` e `argv`[125] (pronunciato *arg-cee* e *arg-vee*)? Cominciamo prima con quello facile: `argc`. Questo è il *numero degli argomenti* incluso il nome del programma stesso. Se pensi a tutti gli argomenti come a un array di stringhe che è esattamente quello che sono allora puoi pensare ad `argc` come alla

123 Storicamente i programmi MS-DOS e Windows lo avrebbero fatto in modo diverso rispetto a Unix. In Unix la shell *espanderebbe* il carattere jolly in tutti i file corrispondenti prima che il programma lo vedesse, mentre le varianti Microsoft passerebbero l'espressione del carattere jolly al programma per gestirla. In ogni caso ci sono argomenti che vengono passati al programma.

124 Dato che sono solo nomi di parametri regolari, in realtà non è necessario chiamarli `argc` e `argv`. Ma è davvero idiomatico usare quei nomi, se diventi creativo gli altri programmatori C ti guarderanno con occhio sospettoso anzi!

125 Dato che sono solo nomi di parametri regolari, in realtà non è necessario chiamarli `argc` e `argv`. Ma è davvero idiomatico usare quei nomi, se diventi creativo gli altri programmatori C ti guarderanno con occhio sospettoso anzi!

lunghezza di quell'array che è esattamente quello che è.

E quindi quello che stiamo facendo in quel ciclo è esaminare tutti gli `argv` e stamparli uno alla volta quindi per un dato input:

```
./foo i like turtles
```

otteniamo un output corrispondente:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

Con questo in mente dovremmo essere pronti a portare avanti il nostro programma per le addizioni.

Il nostro piano:

- Guarda tutti gli argomenti della riga di comando (dopo `argv[0]`, il nome del programma)
- Convertili in numeri interi
- Aggiungili al totale parziale
- Stampa il risultato

Andiamo al dunque!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 1; i < argc; i++) {
// Inizia da 1, il primo argomento
        int value = atoi(argv[i]);
// Usa strtol() per una migliore gestione degli errori

        total += value;
    }

    printf("%d\n", total);
}
```

Esempio esecuzione:

```
$ ./add
0
$ ./add 1
1
$ ./add 1 2
3
$ ./add 1 2 3
6
$ ./add 1 2 3 4
10
```

Naturalmente potrebbe vomitare se gli passi un valore non intero ma la difficoltà viene lasciato come esercizio al lettore.

18.1.1. L'ultimo `argv` è `NULL`

Una curiosità divertente su `argv` è che dopo l'ultima stringa c'è un puntatore a `NULL`.

Che è:

```
argv[argc] == NULL
```

è sempre vero!

Questo potrebbe sembrare inutile ma risulta essere utile in un paio di posti; daremo un'occhiata a uno di quelli adesso.

18.1.2. L'alternativa: `char **argv`

Ricorda che quando chiami una funzione C non distingue tra notazione di array e notazione di puntatore nella dichiarazione della funzione.

Cioè sono uguali:

```
void foo(char a[])
void foo(char *a)
```

Ora è stato conveniente pensare a `argv` come a un array di stringhe ovvero un array di `char*` quindi aveva senso:

```
int main(int argc, char *argv[])
```

ma a causa dell'equivalenza potresti anche scrivere:

```
int main(int argc, char **argv)
```

Sì è un puntatore a un puntatore va bene! Se lo rende più semplice consideralo come un puntatore a una stringa. Ma in realtà è un puntatore a un valore che punta a un `char`.

Ricordiamo inoltre che questi sono equivalenti:

```
argv[i]
*(argv + i)
```

il che significa che puoi eseguire l'aritmetica dei puntatori su `argv`.

Quindi un modo alternativo per utilizzare gli argomenti della riga di comando potrebbe essere semplicemente quello di scorrere lungo l'array `argv` spingendo un puntatore finché non raggiungiamo il `NULL` alla fine.

Modifichiamo il nostro addzionatore e facciamo:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    int total = 0;

    // Trucco carino per fare
    // in modo che il compilatore
    // smetta di avvisare dela
    // variabile non utilizzata argc:
    (void)argc;
```

```
    for (char **p = argv + 1; *p != NULL; p++) {
        int value = atoi(*p);  // Usa strtol() per una migliore gestione degli
errori

        total += value;
    }

    printf("%d\n", total);
}
```

Personalmente utilizzo la notazione di array per accedere ad `argv` ma ho visto anche questo stile guardando in giro.

18.1.3. Fatti divertenti

Solo qualche altra cosa a riguardo `argc` e `argv`.

- Alcuni ambienti potrebbero non impostare `argv[0]` sul nome del programma. Se non è disponibile, `argv[0]` diventerà una stringa vuota. Non l'ho mai visto accadere.
- Le specifiche sono in realtà piuttosto liberali riguardo a ciò che un'implementazione può fare con `argv` e da dove provengono questi valori. Ma ogni sistema su cui ho lavorato funziona allo stesso modo come abbiamo discusso in questa sezione.
- Puoi modificare `argc`, `argv` o qualsiasi stringa a cui punta `argv`. (Basta non rendere quelle stringhe più lunghe di quanto lo siano già!)
- Su alcuni sistemi simili a Unix, la modifica della stringa `argv[0]` comporta la modifica dell'output di `ps`[126] .

Normalmente se hai un programma chiamato `foo` che hai eseguito con `./foo` potresti vederlo nell'output di `ps`:

```
4078 tty1     S      0:00 ./foo
```

Ma se modifichi `argv[0]` in questo modo fai attenzione che la nuova stringa `"Hi! "` ha la stessa lunghezza di quello vecchio `"./foo"`:

```
strcpy(argv[0], "Hi!  ");
```

e poi esegui `ps` mentre il programma `./foo` è ancora in esecuzione vedremo invece questo:

```
4079 tty1     S      0:00 Hi!
```

- Questo comportamento non è nelle specifiche ed è fortemente dipendente dal sistema.

18.2. Exit Status

Hai notato che le dichiarazioni delle funzioni per `main()` restituiscono il tipo `int`? Di cosa si tratta? Ha a che fare con una cosa chiamata *exit status*, è un numero intero che può essere restituito al programma che ha lanciato il vostro per fargli sapere come sono andate le cose. Ora ci sono diversi modi in cui un programma può uscire in C incluso `return` da `main()` o chiamando una delle varianti `exit()`.

Tutti questi metodi accettano `int` come argomento.

Nota a margine: hai visto che praticamente in tutti i miei esempi anche se `main()` dovrebbe restituire un `int`, in realtà non `return` nulla? In qualsiasi altra funzione ciò sarebbe illegale ma c'è un caso speciale in C: se l'esecuzione raggiunge la fine di `main()` senza trovare un `return`

126`ps` Process Status è un comando Unix per vedere quali processi sono in esecuzione in questo momento.

esegue automaticamente un `return 0`. Ma cosa significa lo `0`? Quali altri numeri possiamo inserire? E come vengono utilizzati?

Le specifiche sono chiare e vaghe sull'argomento come è comune. Chiaro perché spiega cosa puoi fare ma vago perché non lo limita particolarmente.

Non resta altro che *andare avanti* e capirlo!

Prendiamo Inception[127] (inizio) per un secondo: scopriamo che quando esegui il tuo programma *lo stai eseguendo da un altro programma*. Di solito quest'altro programma è qualche tipo di shell[128] che non fa molto da sola se non avviare altri programmi.

Ma questo è un processo multifase particolarmente visibile nelle shell della riga di comando:

1. La shell avvia il tuo programma
2. La shell in genere va in modalità di sospensione (per le shell della riga di comando)
3. Il tuo programma viene eseguito
4. Il tuo programma termina
5. La shell si sveglia e attende un altro comando

Ora c'è un piccolo pezzo di comunicazione che avviene tra i passaggi 4 e 5: il programma può restituire un *valore di stato* che la shell può interrogare. In genere questo valore viene utilizzato per indicare il successo o il fallimento del programma e in caso di fallimento, il tipo di fallimento.

Questo valore è quello che stiamo `return` da `main()`. Questo è lo status.

Ora le specifiche C consentono due diversi valori di stato che hanno nomi di macro definiti in `<stdlib.h>`:

Stato	Descrizione
`EXIT_SUCCESS` o 0	Programma terminato con successo.
`EXIT_FAILURE`	Programma terminato con un errore.

Scriviamo un breve programma che moltiplichi due numeri dalla riga di comando. Richiederemo di specificare esattamente due valori. In caso contrario stamperemo un messaggio di errore e usciremo con uno stato di errore.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("usage: mult x y\n");
        return EXIT_FAILURE;
// Indica alla shell che non ha funzionato
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));
```

127https://en.wikipedia.org/wiki/Inception
128https://en.wikipedia.org/wiki/Shell_(computing)

```
    return 0;
// uguale a EXIT_SUCCESS, tutto andava bene.
}
```

Ora se proviamo a eseguirlo otteniamo l'effetto previsto finché non specifichiamo esattamente il numero corretto di argomenti della riga di comando:

```
$ ./mult
usage: mult x y

$ ./mult 3 4 5
usage: mult x y

$ ./mult 3 4
12
```

Ma questo non mostra realmente lo stato di uscita che abbiamo restituito vero? Possiamo però far sì che la shell lo stampi. Supponendo che tu stia eseguendo Bash o un'altra shell POSIX, puoi usare `echo $?` per vederlo[129] .

Proviamo:

```
$ ./mult
usage: mult x y
$ echo $?
1

$ ./mult 3 4 5
usage: mult x y
$ echo $?
1

$ ./mult 3 4
12
$ echo $?
0
```

Interessante! Vediamo che sul mio sistema `EXIT_FAILURE` è 1. Le specifiche non lo specificano quindi potrebbe essere qualsiasi numero. Ma provalo; probabilmente è 1 anche sul tuo sistema.

18.2.1. Altri valori Exit Status

Lo stato 0 significa sicuramente successo, ma che dire di tutti gli altri numeri interi anche quelli negativi?

Qui usciremo dalle specifiche C ed entreremo nel territorio Unix. In generale mentre 0 significa successo un numero positivo diverso da zero significa fallimento. Quindi puoi avere solo un tipo di successo e più tipi di fallimento.

In breve, se vuoi indicare diversi stati di uscita degli errori in un ambiente Unix, puoi iniziare con `1` e procedere verso l'alto.

Su Linux, se provi qualsiasi codice al di fuori dell'intervallo 0-255, eseguirà l'AND bit per bit del codice con `0xff`, bloccandolo effettivamente in quell'intervallo.

È possibile eseguire lo script della shell per utilizzare successivamente questi codici di stato per prendere decisioni su cosa fare dopo.

129In Windows da `cmd.exe` digitare `echo %errorlevel%`. In PowerShell digitare `$LastExitCode`

18.3. variabili di ambiente

Prima di entrare in questo argomento devo avvisarti che C non specifica cosa sia una variabile d'ambiente. Quindi descriverò il sistema di variabili d'ambiente che funziona su tutte le principali piattaforme di cui sono a conoscenza.

Fondamentalmente l'ambiente è il programma che eseguirà il tuo programma ad esempio la shell bash. E potrebbe avere alcune variabili bash definite. Nel caso non lo sapessi la shell può creare le proprie variabili. Ogni shell è diversa, ma in bash puoi semplicemente digitare `set` e te li mostrerà tutti.

Ecco un estratto dalle 61 variabili definite nella mia shell bash:

```
HISTFILE=/home/beej/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/beej
HOSTNAME=FBILAPTOP
HOSTTYPE=x86_64
IFS=$' \t\n'
```

Nota che sono sotto forma di coppie chiave/valore. Ad esempio una chiave è `HOSTTYPE` e il suo valore è `x86_64`. Da una prospettiva C tutti i valori sono stringhe anche se sono numeri[130] . Quindi *comunque*! Per farla breve è possibile ottenere questi valori dall'interno del tuo programma C.

Scriviamo un programma che utilizza la funzione standard `getenv()` per cercare un valore impostato nella shell.

`getenv()` restituirà un puntatore alla stringa del valore, altrimenti `NULL` se la variabile d'ambiente non esiste.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *val = getenv("FROTZ");  // Prova a ottenere il valore

    // Controlla per assicurarti che esista
    if (val == NULL) {
        printf("Cannot find the FROTZ environment variable\n");
        return EXIT_FAILURE;
    }

    printf("Value: %s\n", val);
}
```

Se lo eseguo direttamente ottengo questo:

```
$ ./foo
Cannot find the FROTZ environment variable
```

il che ha senso dato che non l'ho ancora impostato.

In bash posso impostarlo su qualcosa con[131] :

```
$ export FROTZ="C is awesome!"
```

130Se hai bisogno di un valore numerico, converti la stringa con qualcosa come `atoi()` o `strtol()`.
131In Windows CMD.EXE, utilizzare `set FROTZ=value`. In PowerShell utilizzare `$Env:FROTZ=value`.

Quindi se lo eseguo ottengo:

```
$ ./foo
Value: C is awesome!
```

In questo modo puoi impostare i dati nelle variabili di ambiente e puoi inserirli nel tuo codice C e modificare il tuo comportamento di conseguenza.

18.3.1. Impostazione delle variabili d'ambiente

Questo non è standard, ma molti sistemi forniscono modi per impostare le variabili di ambiente. Se utilizzi un sistema Unix, cerca la documentazione per `putenv()`, `setenv()` e `unsetenv()`. Su Windows, vedere `_putenv()`.

18.3.2. Unix-like Variabili d'ambiente alternative

Se utilizzi un sistema simile a Unix è probabile che tu abbia un altro paio di modi per ottenere l'accesso alle variabili di ambiente. Si noti che sebbene le specifiche lo indichino come un'estensione comune non è veramente parte dello standard C. Fa tuttavia parte dello standard POSIX. Una di queste è una variabile chiamata `environ` che va dichiarata così:

```
extern char **environ;
```

È un array di stringhe terminato con un puntatore `NULL`.

Dovresti dichiararlo tu stesso prima di usarlo altrimenti potresti trovarlo nel file di intestazione `<unistd.h>` non standard. Ogni stringa è nella forma `"key=value"` quindi dovrai dividerlo e analizzarlo tu stesso se vuoi ottenere chiavi e valori. Ecco un esempio di looping e stampa delle variabili di ambiente in un paio di modi diversi:

```
#include <stdio.h>

extern char **environ;
// DEVE essere extern E chiamato "environ"

int main(void)
{
    for (char **p = environ; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

Per un gruppo di output simile a questo:

```
SHELL=/bin/bash
COLORTERM=truecolor
TERM_PROGRAM_VERSION=1.53.2
LOGNAME=beej
HOME=/home/beej
... etc ...
```

Usa `getenv()` se possibile perché è più portabile. Ma se devi iterare sulle variabili di ambiente usare `environ` potrebbe essere la strada da percorrere. Un altro modo non standard per ottenere le variabili d'ambiente è di usarlo come parametro di `main()`. Funziona più o meno allo stesso modo

ma eviti di dover aggiungere la variabile `extern environ`. Per quanto ne so nemmeno le specifiche POSIX lo supportano[132] ma è comune nel territorio Unix.

```
#include <stdio.h>

int main(int argc, char **argv, char **env)  // <-- env!
{
    (void)argc; (void)argv;  // Elimina gli avvisi inutilizzati

    for (char **p = env; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; env[i] != NULL; i++) {
        printf("%s\n", env[i]);
    }
}
```

Proprio come usare `environ` ma ancora *meno portabile*. È bello avere degli obiettivi.

19. Il preprocessore C

Prima che il tuo programma venga compilato attraversa una fase chiamata *preelaborazione*. È quasi come se ci fosse un linguaggio *sopra* il linguaggio C che viene eseguito per primo.

Che restituisce il codice C e poi viene compilato. Lo abbiamo già visto in una certa misura con `#include`! Questo è il preprocessore C! Vedendo quella direttiva, include il file nominato proprio lì proprio come se lo avessi digitato lì. E *poi* il compilatore costruisce il tutto.

Ma si scopre che è molto più potente della semplice capacità di includere cose. Puoi definire *macro* che vengono sostituite… e anche macro che accettano argomenti!

19.1. `#include`

Cominciamo con quello che abbiamo già visto molte volte. Questo è ovviamente un modo per includere altre fonti nella tua fonte. Molto comunemente usato con i file header. Sebbene le specifiche consentano tutti i tipi di comportamento con `#include` adotteremo un approccio più pragmatico e parleremo del modo in cui funziona su ogni sistema che abbia mai visto. Possiamo dividere i file header in due categorie: sistema e locale. Cose che sono integrate come `stdio.h`, `stdlib.h`, `math.h`, e così via è possibile includerli con parentesi angolari:

```
#include <stdio.h>
#include <stdlib.h>
```

Le parentesi angolari indicano C: “Ehi, non cercare questo file di intestazione nella directory corrente: cerca invece nella directory di inclusione a livello di sistema.” Il che ovviamente implica che debba esserci un modo per includere i file locali dalla directory corrente. E c'è: con virgolette doppie:

```
#include "mioheader.h"
```

Oppure molto probabilmente puoi cercare nelle directory relative usando barre e punti, come questo:

```
#include "miadir/mioheader.h"
#include "../qualcheheader.py"
```

132https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

Non utilizzare una barra rovesciata (`\`) per i separatori di percorso nel tuo `#include`! È un comportamento indefinito! Utilizzare solo la barra (`/`) anche su Windows. In sintesi utilizza le parentesi angolari (< e >) per le inclusioni di sistema e le virgolette doppie (") per le inclusioni personali.

19.2. Macro semplici

Una *macro* è un identificatore che viene *espanso* in un altro pezzo di codice prima ancora che il compilatore lo veda. Consideralo come un segnaposto: quando il preprocessore vede uno di questi identificatori lo sostituisce con un altro valore che hai definito.

Lo facciamo con `#define` (spesso leggiamo "pound define"). Ecco un esempio:

```
#include <stdio.h>

#define CIAO "Ciao mondo!"
#define PI 3.14159

int main(void)
{
    printf("%s, %f\n", CIAO, PI);
}
```

Nelle righe 3 e 4 abbiamo definito un paio di macro. Ovunque questi appaiano altrove nel codice (riga 8), verranno sostituiti con i valori definiti.

Dal punto di vista del compilatore C, è esattamente come se avessimo scritto questo, invece:

```
#include <stdio.h>

int main(void)
{
    printf("%s, %f\n", "Ciao mondo!", 3.14159);
}
```

Guarda come `CIAO` è stato sostituito con `"Ciao mondo!"` e `PI` è stato sostituito con `3.14159`? Dal punto di vista del compilatore è proprio come se quei valori fossero apparsi proprio lì nel codice.

Tieni presente che le macro non hanno un tipo specifico di *per sé*. In realtà tutto ciò che accade è che vengono sostituiti all'inizio con qualunque cosa siano `#define`. Se il codice C risultante non è valido il compilatore vomiterà.

È inoltre possibile definire una macro senza valore:

```
#define EXTRA_FELICE
```

in tal caso, la macro esiste ed è definita ma è definita come nulla. Quindi ovunque si trovi nel testo verrà semplicemente sostituito con nulla. Vedremo l'utilizzo di questo in seguito.

È convenzionale scrivere i nomi delle macro in `TUTTO_MAIUSCOLO` anche se tecnicamente non è richiesto.

Nel complesso questo ti dà un modo per definire valori costanti che sono effettivamente globali e possono essere utilizzati *ovunque*. Anche in quei luoghi in cui una variabile `const` non funziona ad esempio nei casi di `switch` e con lunghezze fisse di array.

Detto questo online infuria il dibattito se una variabile `const` digitata sia migliore della macro `#define` nel caso generale.

Può anche essere usato per sostituire o modificare parole chiave un concetto completamente estraneo a `const` anche se questa pratica dovrebbe essere usata con parsimonia.

19.3. Compilazione condizionale

È possibile fare in modo che il preprocessore decida se presentare o meno determinati blocchi di codice al compilatore o semplicemente rimuoverli completamente prima della compilazione.

Lo facciamo fondamentalmente racchiudendo il codice in blocchi condizionali simili alle istruzioni `if-else`.

19.3.1. If definito, `#ifdef` e `#endif`

Prima di tutto proviamo a compilare un codice specifico a seconda che sia definita o meno una macro.

```
#include <stdio.h>

#define EXTRA_FELICE

int main(void)
{

#ifdef EXTRA_FELICE
    printf("Sono extra felice!\n");
#endif

    printf("OK!\n");
}
```

In questo esempio definiamo `EXTRA_FELICE` (ad essere nulla ma *è* definito), poi alla riga 8 controlliamo se è definito con una direttiva `#ifdef`. Se viene definito, il codice successivo verrà incluso fino al `#endif`.

Quindi poiché è definito il codice verrà incluso per la compilazione e l'output sarà:

```
Sono extra felice!
OK!
```

Se dovessimo commentare `#define` in questo modo:

```
//#define EXTRA_FELICE
```

quindi non verrebbe definito e il codice non verrebbe incluso nella compilazione. E l'output sarebbe semplicemente:

```
OK!
```

È importante ricordare che queste decisioni avvengono in fase di compilazione! Il codice viene effettivamente compilato o rimosso a seconda della condizione. Ciò è in contrasto con un'istruzione `if` standard che viene valutata mentre il programma è in esecuzione.

19.3.2. If Non definito, `#ifndef`

C'è anche il senso negativo di “if definito”: “if non definito” o `#ifndef`. Potremmo modificare l'esempio precedente per produrre cose diverse a seconda che qualcosa sia stato definito o meno:

```
#ifdef EXTRA_FELICE
    printf("Sono extra felice!\n");
#endif
```

```
#ifndef EXTRA_FELICE
    printf("Sono normale\n");
#endif
```

Vedremo un modo più pulito per farlo nella prossima sezione.

Ricollegando il tutto ai file header abbiamo visto come possiamo far sì che i file header vengano inclusi solo una volta racchiudendoli in direttive del preprocessore come questa:

```
#ifndef MIOHEADER_H  // Prima linea di mioheader.h
#define MIOHEADER_H

int x = 12;

#endif  // Ultima riga di mioheader.h
```

Ciò dimostra come una macro persista tra file e più `#include`. Se non è ancora definito definiamolo e compiliamo l'intero file header.

Ma la prossima volta che viene incluso vediamo che `MYHEADER_H` *è* definito quindi non inviamo il file di intestazione al compilatore: viene effettivamente rimosso.

19.3.3. `#else`

Ma non è tutto ciò che possiamo fare! C'è anche un `#else` che possiamo aggiungere al mix.

Modifichiamo l'esempio precedente:

```
#ifdef EXTRA_FELICE
    printf("Sono extra felice!\n");
#else
    printf("Sono normale\n");
#endif
```

Ora se `EXTRA_FELICE` non è definito colpirà la clausola `#else` e verrà stampato:

```
Sono normale
```

19.3.4. Else-If: `#elifdef`, `#elifndef`

Questa funzionalità è nuova in C23!

E se volessi qualcosa di più complesso però? Forse hai bisogno di una struttura a cascata if-else per creare correttamente il tuo codice?

Fortunatamente abbiamo queste direttive a nostra disposizione. Possiamo usare `#elifdef` per "else if definito":

```
#ifdef MODE_1
    printf("Questo è mode 1\n");
#elifdef MODE_2
    printf("Questo è mode 2\n");
#elifdef MODE_3
    printf("Questo è mode 3\n");
#else
    printf("Ci sono altre mode\n");
#endif
```

D'altra parte puoi usare `#elifndef` per "else if non definito".

19.3.5. Condizionale generale: `#if`, `#elif`

Funziona in modo molto simile alle direttive `#ifdef` e `#ifndef` in quanto puoi anche avere un `#else` e il tutto si conclude con `#endif`.

L'unica differenza è che l'espressione costante dopo `#if` deve essere valutata come vera (diversa da zero) affinché il codice in `#if` venga compilato. Quindi invece di stabilire se qualcosa sia definito o meno, vogliamo un'espressione che valga come vero.

```
#include <stdio.h>

#define FATTORE_FELICITA 1

int main(void)
{

#if FATTORE_FELICITA == 0
    printf("Non sono felice!\n");
#elif FATTORE_FELICITA == 1
    printf("Sono solo normale\n");
#else
    printf("Sono extra felice!\n");
#endif

    printf("OK!\n");
}
```

Ancora una volta per le clausole `#if` senza corrispondenza il compilatore non vedrà nemmeno quelle righe. Per il codice precedente una volta terminato il preprocessore tutto ciò che vede il compilatore è:

```
#include <stdio.h>

int main(void)
{

    printf("Sono solo normale\n");

    printf("OK!\n");
}
```

Una cosa da hacker per cui viene utilizzato è commentare rapidamente un gran numero di righe[133] .

Se metti un `#if 0` ("if false") all'inizio del blocco da commentare e un `#endif` alla fine puoi ottenere questo effetto:

```
#if 0
    printf("Tutto questo è codice"); /* è effettivo */
    printf("commento fuori"); // a causa di #if 0
#endif
```

Cosa succede se utilizzi un compilatore precedente a C23 e non disponi del supporto per la direttiva `#elifdef` o `#elifndef`? Come possiamo ottenere lo stesso effetto con `#if`? Cioè, e se lo volessi?:

```
#ifdef FOO
    x = 2;
#elifdef BAR
```

133Non puoi sempre racchiudere il codice tra i commenti `/* */` perché questi non si nidificano.

```
// POTENZIALE ERRORE: Non supportato prima di C23
    x = 3;
#endif
```

Come potrei farlo?

Si scopre che esiste un operatore del preprocessore chiamato `defined` che possiamo usare con un'istruzione `#if`.

Questi sono equivalenti:

```
#ifdef FOO
#if defined FOO
#if defined(FOO)
// Parentesi facoltative
```

Come anche questi:

```
#ifndef FOO
#if !defined FOO
#if !defined(FOO)
// Parentesi facoltative
```

Nota come possiamo usare lo standard NOT operatore (!) per “not defined”.

Quindi ora siamo di nuovo nella terra `#if` e possiamo usare `#elif` impunemente!

Questo codice rotto:

```
#ifdef FOO
    x = 2;
#elifdef BAR
// ERRORE POTENZIALE: non supportato in precedenza a C23
    x = 3;
#endif
```

può essere sostituito con:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

19.3.6. Perdere una macro: `#undef`

Se hai definito qualcosa ma non ti serve più puoi annullarne la definizione con `#undef`.

```
#include <stdio.h>

int main(void)
{
#define CAPRE

#ifdef CAPRE
    printf("Capre rilevate!\n");  // stampa
#endif

#undef CAPRE  // Rendi CAPRE non più definito

#ifdef CAPRE
    printf("Capre rilevate, di nuovo!\n"); // non stampa
```

```
#endif
}
```

19.4. Macro integrate

Lo standard definisce molte macro integrate che puoi testare e utilizzare per la compilazione condizionale. Diamo un'occhiata a quelli qui.

19.4.1. Macro obbligatorie

Questi sono tutti definiti:

Macro	Descrizione
`__DATE__`	La data di compilazione, ad esempio quando stai compilando questo file, in formato `Mmm dd yyyy`
`__TIME__`	L'ora della compilazione in formato `hh:mm:ss`
`__FILE__`	Una stringa contenente il nome di questo file
`__LINE__`	Il numero di riga del file su cui appare questa macro
`__func__`	Il nome della funzione in cui appare, come stringa[134]
`__STDC__`	Definito con 1 se si tratta di un compilatore C standard
`__STDC_HOSTED__`	Sarà 1 se il compilatore è un'implementazione ospitata[135] , altrimenti 0
`__STDC_VERSION__`	Questa versione di C, un long int e costante nella forma yyyymmL, e.g. 201710L

Mettiamoli insieme.

```
#include <stdio.h>
```

134Questa non è realmente una macro: tecnicamente è un identificatore. Ma è l'unico identificatore predefinito e sembra molto simile a una macro, quindi lo includo qui. Come un ribelle.

135Un'implementazione ospitata significa fondamentalmente che stai utilizzando lo standard C completo, probabilmente su un sistema operativo di qualche tipo. E probabilmente lo sei. Se stai utilizzando il bare metal in una sorta di sistema incorporato, probabilmente stai utilizzando un'implementazione autonoma.

```
int main(void)
{
    printf("Questa funzione: %s\n", __func__);
    printf("Questo file: %s\n", __FILE__);
    printf("Questa linea: %d\n", __LINE__);
    printf("Compilato il: %s %s\n", __DATE__, __TIME__);
    printf("Versione C: %ld\n", __STDC_VERSION__);
}
```

L'output sul mio sistema è:

```
Questa funzione: main
Questo file: foo.c
Questa line: 7
Compilato il on: Nov 23 2020 17:16:27
Versione C: 201710
```

`__FILE__`, `__func__` e `__LINE__` sono particolarmente utili per segnalare condizioni di errore nei messaggi agli sviluppatori. La macro `assert()` in `<assert.h>` li utilizza per evidenziare il punto del codice in cui l'asserzione ha avuto esito negativo.

1. `__STDC_VERSION__`

Nel caso te lo stia chiedendo, ecco i numeri di versione per le diverse versioni principali della specifica del linguaggio C:

Release	ISO/IEC version	`__STDC_VERSION__`
C89	ISO/IEC 9899:1990	undefined
C89	ISO/IEC 9899:1990/Amd. 1:1995	199409L
C99	ISO/IEC 9899:1999	199901L
C11	ISO/IEC 9899:2011/Amd. 1:2012	201112L

Nota che la macro non esisteva originariamente in C89.

Tieni inoltre presente che il piano prevede che i numeri di versione aumentino notevolmente, quindi puoi sempre verificare ad esempio "almeno C99" con:

```
#if __STDC_VERSION__ >= 1999901L
```

19.4.2. Macro facoltative

La tua implementazione potrebbe definire anche questi. Oppure potrebbe non esserlo.

Macro	Descrizione
`__STDC_ISO_10646__`	Se definito, `wchar_t` contiene valori

	Unicode, altrimenti qualcos'altro
`__STDC_MB_MIGHT_NEQ_WC__`	Un 1 indica che i valori in caratteri multibyte potrebbero non corrispondere allo stesso modo ai valori in caratteri estesi
`__STDC_UTF_16__`	Un 1 indica che il sistema utilizza la codifica UTF-16 nel tipo $char16_t$
`__STDC_UTF_32__`	Un 1 indica che il sistema utilizza la codifica UTF-32 nel tipo $char32_t$
`__STDC_ANALYZABLE__`	Un 1 indica che il codice è analizzabile[136]
`__STDC_IEC_559__`	1 se è supportata il floating point IEEE-754 (aka IEC 60559).
`__STDC_IEC_559_COMPLEX__`	1 se è supportata la complex floating point IEC 60559
`__STDC_LIB_EXT1__`	1 se questa implementazione supporta una varietà di funzioni di libreria standard alternative “sicure” (hanno i suffissi _s nel nome)
`__STDC_NO_ATOMICS__`	1 se questa implementazione non supporta _Atomic o <stdatomic.h>
`__STDC_NO_COMPLEX__`	1 se questa implementazione non supporta tipi complessi o <complex.h>
`__STDC_NO_THREADS__`	1 se questa implementazione non supporta <threads.h>
`__STDC_NO_VLA__`	1 se questa implementazione non supporta array di lunghezza variabiles

19.5. Macro con argomenti

Tuttavia le macro sono più potenti della semplice sostituzione. Puoi impostarli per accettare anche argomenti sostituiti. Spesso sorge la domanda su quando utilizzare le macro con parametri rispetto alle funzioni. Risposta breve: utilizza le funzioni. Ma vedrai molte macro in natura e nella libreria

136OK lo so, era una risposta di fuga. Fondamentalmente esiste un'estensione opzionale che i compilatori possono implementare in cui accettano di limitare determinati tipi di comportamenti indefiniti in modo che il codice C sia più suscettibile all'analisi statica del codice. È improbabile che tu abbia bisogno di usarlo.

standard. Le persone tendono a usarli per cose brevi e matematiche e anche per funzionalità che potrebbero cambiare da piattaforma a piattaforma. Puoi definire parole chiave diverse per una piattaforma o per un'altra.

19.5.1. Macro con un argomento

Cominciamo con uno semplice che eleva al quadrato un numero:

```
#include <stdio.h>

#define SQR(x) x * x
// Non del tutto giusto, ma abbi pazienza

int main(void)
{
    printf("%d\n", SQR(12));  // 144
}
```

Ciò che significa è "ovunque vedi `SQR` con un certo valore sostituiscilo con quel valore moltiplicato per se stesso".

Quindi la riga 7 verrà modificata in:

```
printf("%d\n", 12 * 12);  // 144
```

che C converte comodamente in 144.

Ma abbiamo commesso un errore elementare in quella macro, un errore che dobbiamo evitare.

Controlliamolo. Cosa succederebbe se volessimo calcolare `SQR(3 + 4)`? Bene, `$3+4=7$`, quindi dobbiamo voler calcolare `$7^2=49$`. Questo è tutto; 49: risposta finale.

Inseriamolo nel nostro codice e vediamo che otteniamo… 19?

```
printf("%d\n", SQR(3 + 4));  // 19!!??
```

Cosa è successo?

Se seguiamo la macroespansione otteniamo

```
printf("%d\n", 3 + 4 * 3 + 4);  // 19!
```

Ops! Poiché la moltiplicazione ha la precedenza eseguiamo `$4\times3=12$` prima e otteniamo `$3+12+4=19$`. Non quello che cercavamo.

Quindi dobbiamo sistemare questo problema per farlo bene.

Questo è così comune che dovresti farlo automaticamente ogni volta che crei una macro matematica con parametri!

La soluzione è semplice: basta aggiungere alcune parentesi!

```
#define SQR(x) (x) * (x)
// Meglio... ma ancora non abbastanza buono!
```

E ora la nostra macro si espande a:

```
printf("%d\n", (3 + 4) * (3 + 4));  // 49! Woo hoo!
```

Ma in realtà abbiamo ancora lo stesso problema che potrebbe manifestarsi se abbiamo un operatore con precedenza maggiore rispetto a quello di moltiplicazione `(*)` nelle vicinanze.

Quindi il modo sicuro e corretto per mettere insieme la macro è racchiudere il tutto tra parentesi

aggiuntive in questo modo:

```
#define SQR(x) ((x) * (x))   // Bene!
```

Prendi l'abitudine di farlo quando crei una macro matematica e non puoi sbagliare.

19.5.2. Macro con più di un argomento

Puoi impilare queste cose quanto vuoi:

```
#define TRIANGLE_AREA(w, h) (0.5 * (w) * (h))
```

Facciamo alcune macro che risolvono utilizzando la formula quadratica. Nel caso non lo avessi in mente dice per le equazioni della forma: puoi risolvere `$ax^2+bx+c=0$` con la formula quadratica: `$x=\displaystyle\frac{-b\pm\sqrt{b^2-4ac}}{2a}$` Il che è pazzesco. Notare anche il più o il meno (`$\pm$`) lì, indicando che in realtà ci sono due soluzioni.

Quindi creiamo macro per entrambi:

```
#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
```

Quindi questo ci fa fare qualche calcolo. Ma definiamone un altro che possiamo usare come argomento per `printf()` per stampare entrambe le risposte.

```
//          macro              replacement
//      |------------| |-----------------------------|
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

Sono solo un paio di valori separati da una virgola e possiamo usarli come una sorta di argomento "combinato" per `printf()` in questo modo:

```
printf("x = %f or x = %f\n", QUAD(2, 10, 5));
```

Mettiamolo insieme in un codice:

```
#include <stdio.h>
#include <math.h>  // For sqrt()

#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)

int main(void)
{
    printf("2*x^2 + 10*x + 5 = 0\n");
    printf("x = %f or x = %f\n", QUAD(2, 10, 5));
}
```

E questo ci dà l'output:

```
2*x^2 + 10*x + 5 = 0
x = -0.563508 or x = -4.436492
```

Inserendo uno di questi valori otteniamo approssimativamente zero (un po' fuori perché i numeri non sono esatti):

`$2\times-0.563508^2+10\times-0.563508+5\approx0.000003$`

19.5.3. Macro con argomenti variabili

C'è anche un modo per far passare un numero variabile di argomenti a una macro, utilizzando i

puntini di sospensione (…) dopo gli argomenti noti e denominati. Quando la macro viene espansa tutti gli argomenti aggiuntivi saranno in un elenco separato da virgole nella macro __VA_ARGS__ e potranno essere sostituiti da lì:

```
#include <stdio.h>

// Combine the first two arguments to a single number,
// then have a commalist of the rest of them:

#define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__

int main(void)
{
    printf("%d %f %s %d\n", X(5, 4, 3.14, "Hi!", 12));
}
```

La sostituzione che avviene sulla riga 10 sarebbe:

```
printf("%d %f %s %d\n", (10*(5) + 20*(4)), 3.14, "Hi!", 12);
```

per output:

```
130 3.140000 Hi! 12
```

Puoi anche “stringificare” __VA_ARGS__ mettendo un # davanti ad esso:

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Stampa "1, 2, 3"
```

19.5.4. Stringification

Come abbiamo già menzionato sopra puoi trasformare qualsiasi argomento in una stringa facendolo precedere da un # nel testo sostitutivo.

Ad esempio potremmo stampare qualsiasi cosa come stringa con questa macro e `printf()`:

```
#define STR(x) #x

printf("%s\n", STR(3.14159));
```

In tal caso la sostituzione comporta:

```
printf("%s\n", "3.14159");
```

Vediamo se possiamo usarlo con maggiore efficacia in modo da poter passare qualsiasi nome di variabile `int` in una macro e farne stampare il nome e il valore.

```
#include <stdio.h>

#define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)

int main(void)
{
    int a = 5;

    PRINT_INT_VAL(a);  // prints "a = 5"
}
```

Alla riga 9 otteniamo la seguente sostituzione di macro:

```
printf("%s = %d\n", "a", 5);
```

19.5.5. Concatenazione

Possiamo anche concatenare due argomenti insieme con ##. Momenti divertenti!

```
#define CAT(a, b) a ## b

printf("%f\n", CAT(3.14, 1592));   // 3.141592
```

19.6. Macro multilinea

È possibile continuare una macro su più righe se si evita il ritorno a capo con una barra rovesciata (`\`).

Scriviamo una macro multilinea che stampi i numeri da 0 al prodotto dei due argomenti passati.

```
#include <stdio.h>

#define PRINT_NUMS_TO_PRODUCT(a, b) do { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
} while(0)

int main(void)
{
    PRINT_NUMS_TO_PRODUCT(2, 4);  // Emette numeri da 0 a 7
}
```

Un paio di cose da notare:

- Esce alla fine di ogni riga tranne l'ultima per indicare che la macro continua.
- Il tutto è racchiuso in un ciclo `do- while(0)` con parentesi graffe.

Quest'ultimo punto potrebbe essere un po' strano ma si tratta solo di assorbire il finale `;` il programmatore lascia dopo la macro.

All'inizio pensavo che sarebbe bastato usare le parentesi graffe, ma c'è un caso in cui fallisce se il programmatore inserisce un punto e virgola dopo la macro. Ecco il caso:

```
#include <stdio.h>

#define FOO(x) { (x)++; }

int main(void)
{
    int i = 0;

    if (i == 0)
        FOO(i);
    else
        printf(":-(\n");

    printf("%d\n", i);
}
```

Sembra abbastanza semplice, ma non verrà creato senza un errore di sintassi:

```
foo.c:11:5: error: ‘else’ without a previous ‘if’
```

Lo vedi?

Diamo un'occhiata all'espansione:

```
if (i == 0) {
    (i)++;
};              // <-- Problemi con la maiuscola-T!

else
    printf(":-(\n");
```

Il `;` mette fine all'istruzione `if` quindi l'altro `else` è semplicemente fluttuante là fuori illegalmente [138] . Quindi metti in mezzo quella macro multilinea con un `do-while(0)`.

19.7. Esempio: una macro di asserzione

Aggiungere asserzioni al codice è un buon modo per individuare condizioni che ritieni non dovrebbero verificarsi. C fornisce la funzionalità `assert()`. Controlla una condizione e se è falsa il programma si blocca dicendoti il file e il numero di riga su cui l'asserzione non è riuscita. Ma questo manca.

- Prima di tutto non puoi specificare un messaggio aggiuntivo con l'asserzione.
- In secondo luogo non esiste un semplice interruttore di accensione e spegnimento per tutte le affermazioni.

Possiamo affrontare il primo con le macro.

Fondamentalmente quando ho questo codice:

```
ASSERT(x < 20, "x must be under 20");
```

Voglio che accada qualcosa del genere (supponendo che `ASSERT()` sia sulla riga 220 di `foo.c`):

```
if (!(x < 20)) {
    fprintf(stderr, "foo.c:220: assertion x < 20 failed: ");
    fprintf(stderr, "x must be under 20\n");
    exit(1);
}
```

Possiamo ottenere il nome del file dalla macro `__FILE__`, e il numero di riga da `__LINE__`. Il messaggio è già una stringa ma `x < 20` non lo è quindi dovremo stringarlo con #. Possiamo creare una macro su più righe utilizzando la barra rovesciata alla fine della riga.

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

(Sembra un po' strano con `__FILE__` in primo piano in questo modo ma ricorda che è una stringa letterale e le stringhe letterali una accanto all'altra vengono concatenate automaticamente. `__LINE__` d'altra parte è solo un `int`.)

137Infrangere la legge… infrangere la legge…
138Infrangere la legge… infrangere la legge…

E funziona! Se eseguo questo:

```
int x = 30;

ASSERT(x < 20, "x must be under 20");
```

ottengo questo output:

`foo.c:23: assertion x < 20 failed: x must be under 20` Molto bello!

L'unica cosa rimasta è un modo per accenderlo e spegnerlo, e potremmo farlo con la compilazione condizionale.

Ecco l'esempio completo:

```
#include <stdio.h>
#include <stdlib.h>

#define ASSERT_ENABLED 1

#if ASSERT_ENABLED
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
#else
#define ASSERT(c, m)  // Macro vuota se non abilitata
#endif

int main(void)
{
    int x = 30;

    ASSERT(x < 20, "x must be under 20");
}
```

Questo ha l'output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

19.8. La direttiva `#error`

Questa direttiva causa un errore nel compilatore non appena la vede. Comunemente viene utilizzato all'interno di un condizionale per impedire la compilazione a meno che non siano soddisfatti alcuni prerequisiti:

```
#ifndef __STDC_IEC_559__
    #error I really need IEEE-754 floating point to compile. Sorry!
#endif
```

Alcuni compilatori hanno una direttiva `#warning` complementare non standard che genererà un avviso ma non interromperà la compilazione ma questo non è nelle specifiche C11.

19.9. La Direttiva #embed

Novità in C23!

Al momento non funziona con nessuno dei miei compilatori quindi prendi questa sezione con le pinze!

Il succo di ciò è che puoi includere byte di un file come costanti intere come se li avessi digitati.

Ad esempio se hai un file binario denominato `foo.bin` che contiene quattro byte con valori decimali 11, 22, 33 e 44 e fa questo:

```
int a[] = {
#embed "foo.bin"
};
```

Sarà proprio come se avessi digitato questo:

```
int a[] = {11,22,33,44};
```

Questo è un modo davvero potente per inizializzare un array con dati binari senza dover prima convertirli tutti in codice: il preprocessore lo fa per te!

Un caso d'uso più tipico potrebbe essere un file contenente una piccola immagine da visualizzare che non si desidera caricare in fase di runtime.

Ecco un altro esempio:

```
int a[] = {
#embed <foo.bin>
};
```

Se si utilizzano parentesi angolari il preprocessore cerca in una serie di posizioni definite dall'implementazione per individuare il file proprio come farebbe `#include`. Se usi le virgolette doppie e la risorsa non viene trovata il compilatore lo proverà come se avessi usato le parentesi angolari in un ultimo disperato tentativo di trovare il file. `#embed` funziona come `#include` in quanto incolla effettivamente i valori prima che il compilatore li veda. Ciò significa che puoi usarlo in tutti i tipi di luogo:

```
return
#embed "somevalue.dat"
;
or
int x =
#embed "xvalue.dat"
;
```

Ora questi sono sempre byte? Significa che avranno valori da 0 a 255 inclusi? La risposta è sicuramente “sì” per impostazione predefinita tranne quando è “no”.

Tecnicamente gli elementi saranno larghi `CHAR_BIT` bit. E questo è molto probabilmente 8 sul tuo sistema quindi otterresti quell'intervallo 0-255 nei tuoi valori. (Saranno sempre non negativi.)

Inoltre è possibile che un'implementazione consenta di sovrascriverlo in qualche modo ad esempio sulla riga di comando o con parametri.

La dimensione del file in bit deve essere un multiplo della dimensione dell'elemento. Cioè se ogni elemento è di 8 bit la dimensione del file (in bit) deve essere un multiplo di 8. Nell'uso quotidiano questo è un modo confuso per dire che ogni file deve essere un numero intero di byte… e ovviamente è così. Onestamente non sono nemmeno sicuro del motivo per cui mi sono preso la

briga di leggere questo paragrafo. Leggi le specifiche se sei davvero così curioso.

19.9.1. Parametri #embed

Esistono tutti i tipi di parametri che puoi specificare nella direttiva `#embed`. Ecco un esempio con il parametro `limit()` non ancora introdotto:

```
int a[] = {
#embed "/dev/random" limit(5)
};
```

Ma cosa succede se hai già definito un `limit` da qualche altra parte?? Fortunatamente puoi inserire __ attorno alla parola chiave e funzionerà allo stesso modo:

```
int a[] = {
#embed "/dev/random" __limit__(5)
};
```

Ora… cos'è questa cosa del `limit`?

19.9.2. Il parametro `limit()`

È possibile specificare un limite al numero di elementi da incorporare con questo parametro.

Questo è un valore massimo non un valore assoluto. Se il file incorporato è più corto del limite specificato verrà importato solo quel numero di byte.

L'esempio `/dev/random` sopra è un esempio della motivazione di ciò—in Unix questo è un *file di dispositivo a caratteri* che restituirà un flusso infinito di numeri piuttosto casuali.

Incorporare un numero infinito di byte è impegnativo per la RAM quindi il parametro `limit` ti dà un modo per fermarti dopo un certo numero.

Infine puoi utilizzare le macro `#define` nel tuo `limit` nel caso fossi curioso.

19.9.3. Il parametro `if_empty`

Questo parametro definisce quale dovrebbe essere il risultato dell'incorporamento se il file esiste ma non contiene dati. Diciamo che il file `foo.dat` contiene un singolo byte con il valore 123. Se facciamo questo:

```
int x =
#embed "foo.dat" if_empty(999)
;
```

otterremo:

```
int x = 123;   // Quando foo.dat contiene 123 byte
```

Ma cosa succede se il file `foo.dat` è lungo zero byte (cioè non contiene dati ed è vuoto)? Se così fosse si espanderebbe a:

```
int x = 999;   // Quando foo.dat è vuoto
```

In particolare se il `limit` è impostato su 0 allora `if_empty` verrà sempre sostituito. Cioè un limite pari a zero significa effettivamente che il file è vuoto.

Questo emetterà sempre `x = 999`, indipendentemente da cosa ci sia in `foo.dat`:

```
int x =
#embed "foo.dat" limit(0) if_empty(999)
```

```
;
```

19.9.4. I parametri `prefix()` e `suffix()`

Questo è un modo per anteporre alcuni dati all'incorporamento.

Tieni presente che questi riguardano solo i dati non vuoti! Se il file è vuoto né il `prefix` né il `suffix` hanno alcun effetto.

Ecco un esempio in cui incorporiamo tre numeri casuali ma il prefisso ai numeri 11 e il suffisso con 99:

```
int x[] = {
#embed "/dev/urandom" limit(3) prefix(11,) suffix(,99)
};
```

Risultato di esempio:

```
int x[] = {11,135,116,220,99};
```

Non è necessario utilizzarli entrambi `prefix` e `suffix`. Puoi usarli entrambi uno e l'altro o nessuno dei due.

Possiamo sfruttare in modo efficace la caratteristica che questi vengono applicati solo a file non vuoti come mostrato nell'esempio seguente spudoratamente rubato alle specifiche.

Diciamo di avere un file `foo.dat` che contiene alcuni dati. E vogliamo usarlo per inizializzare un array e poi vogliamo un suffisso sull'array che sia un elemento zero.

Nessun problema, giusto?

```
int x[] = {
#embed "foo.dat" suffix(,0)
};
```

Se `foo.dat` contenesse 11, 22 e 33, otterremmo:

```
int x[] = {11,22,33,0};
```

Ma aspetta! Cosa succede se `foo.dat` è vuoto? Allora otteniamo:

```
int x[] = {};
```

e questo non va bene.

Ma possiamo risolvere il problema in questo modo:

```
int x[] = {
#embed "foo.dat" suffix(,)
    0
};
```

Poiché il parametro `suffix` viene omesso se il file è vuoto questo si trasformerebbe semplicemente in:

```
int x[] = {0};
```

il che va bene.

19.9.5. L'identificatore `__has_embed()`

Questo è un ottimo modo per verificare se un particolare file è disponibile per essere incorporato e

anche se è vuoto o meno.

Lo usi con la direttiva `#if`.

Ecco un pezzo di codice che otterrà 5 numeri casuali dal dispositivo a caratteri del generatore di numeri casuali. Se questo non esiste tenta di ottenerli da un file `myrandoms.dat`. Se ciò non esiste utilizza alcuni valori codificati:

```
    int random_nums[] = {
#if __has_embed("/dev/urandom")
    #embed "/dev/urandom" limit(5)
#elif __has_embed("myrandoms.dat")
    #embed "myrandoms.dat" limit(5)
#else
    140,178,92,167,120
#endif
    };
```

Tecnicamente l'identificatore `__has_embed()` si risolve in uno di tre valori:

`__has_embed()` Result	Descrizione
`__STDC_EMBED_NOT_FOUND__`	Se il file non viene trovato.
`__STDC_EMBED_FOUND__`	Se il file viene trovato e non è vuoto.
`__STDC_EMBED_EMPTY`	Se il file viene trovato ed è vuoto.

Ho buone ragioni per credere che `__STDC_EMBED_NOT_FOUND__` sia `0` e gli altri non siano zero (perché è implicito nella proposta e ha senso logico) ma ho difficoltà a trovarlo in questa versione della bozza delle specifiche. TODO

19.9.6. Altri parametri

Un'implementazione del compilatore può definire altri parametri di incorporamento quanto desidera: cerca questi parametri non standard nella documentazione del tuo compilatore.

Ad esempio:

```
#embed "foo.bin" limit(12) frotz(lamp)
```

Questi potrebbero comunemente avere un prefisso per aiutare con lo spazio dei nomi:

```
#embed "foo.bin" limit(12) fmc::frotz(lamp)
```

Potrebbe essere sensato provare a rilevare se questi sono disponibili prima di usarli e fortunatamente possiamo usare `__has_embed` per aiutarci qui. Normalmente `__has_embed()` ci dirà semplicemente se il file è presente o meno. Ma… ed ecco la parte divertente—restituirà anche false se anche eventuali parametri aggiuntivi non sono supportati!

Quindi se gli forniamo un file di cui *sappiamo* che esiste insieme a un parametro di cui vogliamo testare l'esistenza ci dirà effettivamente se quel parametro è supportato.

Quale file *esiste* sempre però? Risulta che possiamo usare la macro `__FILE__` che si espande nel nome del file sorgente che fa riferimento ad esso! Quel file *deve* esistere altrimenti qualcosa non va nel reparto uova e gallina.

Testiamo il parametro `frotz` per vedere se possiamo usarlo:

```
#if __has_embed(__FILE__ fmc::frotz(lamp))
    puts("fmc::frotz(lamp) is supported!");
#else
    puts("fmc::frotz(lamp) is NOT supported!");
#endif
```

19.9.7. Incorporamento di valori multibyte

Che ne dici di inserire degli `int` invece dei singoli byte? Che dire dei valori multibyte nel file incorporato?

Questo non è supportato dallo standard C23 ma in futuro potrebbero essere definite delle estensioni di implementazione.

19.10. La Direttiva `#pragma`

Questa è una direttiva originale abbreviazione di “pragmatico”. Puoi usarlo per fare… beh qualsiasi cosa il tuo compilatore ti supporti a fare con esso.

Fondamentalmente l'unica volta che lo aggiungerai al tuo codice è se qualche documentazione ti dice di farlo.

19.10.1. Pragma non standard

Ecco un esempio non standard dell'utilizzo di `#pragma` per far sì che il compilatore esegua un ciclo `for` in parallelo con più thread (se il compilatore supporta l'estensione OpenMP[139]):

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) { ... }
```

Esistono tutti i tipi di direttive `#pragma` documentate in tutti e quattro gli angoli del globo. Tutti i `#pragma` non riconosciuti vengono ignorati dal compilatore.

19.10.2. Pragma standard

Ce ne sono anche alcuni standard che iniziano con `STDC` e seguono la stessa forma:

```
#pragma STDC pragma_name on-off
```

La parte `on-off` può essere `ON`, `OFF` o `DEFAULT`.

E `pragma_name` può essere uno di questi:

Pragma Name	Descrizione
`FP_CONTRACT`	Consenti espressioni in floating point di essere contratti in un'unica operazione per evitare errori di arrotondamento che potrebbero verificarsi da più operazioni.
`FENV_ACCESS`	Impostare su ON se si prevede di accedere ai flag di stato in floating point. Se OFF, il compilatore potrebbe eseguire ottimizzazioni

139https://www.openmp.org/

	che rendono i valori nei flag incoerenti o non validi.
`CX_LIMITED_RANGE`	Imposta su ON per consentire al compilatore di ignorare i controlli di overflow durante l'esecuzione di operazioni aritmetiche complesse. Il valore predefinito è OFF.

Per esempio:

```
#pragma STDC FP_CONTRACT OFF
#pragma STDC CX_LIMITED_RANGE ON
```

Quanto a `CX_LIMITED_RANGE` sottolinea la specifica:

> Lo scopo del pragma è consentire all'implementazione di utilizzare le formule:
>
> `$(x+iy)\times(u+iv) = (xu-yv)+i(yu+xv)$`
>
> `$(x+iy)/(u+iv) = [(xu+yv)+i(yu-xv)]/(u^2+v^2)$`
>
> `|$x+iy$|$=\sqrt{x^2+y^2}$`
>
> dove il programmatore può determinare che sono sicuri.

19.10.3. Operatore `_Pragma`

Questo è un altro modo per dichiarare un pragma che potresti utilizzare in una macro.

Questi sono equivalenti:

```
#pragma "Unnecessary" quotes
_Pragma("\"Unnecessary\" quotes")
```

Questo può essere utilizzato in una macro se necessario:

```
#define PRAGMA(x) _Pragma(#x)
```

19.11. The `#line` Directive

Ciò consente di sovrascrivere i valori di `__LINE__` e `__FILE__`. Se vuoit.

Non ho mai voluto farlo ma in K&R2 scrivono:

> A vantaggio di altri preprocessori che generano programmi C […]

Quindi forse è proprio così.

Per sovrascrivere il numero di riga in, ad esempio 300:

```
#line 300
```

e `__LINE__` continuerà a contare da lì.

Per sovrascrivere il numero di riga e il nome del file:

```
#line 300 "newfilename"
```

19.12. La direttiva Null

Un # su una riga da solo viene ignorato dal preprocessore. Ora ad essere del tutto onesto non so quale sia il caso d'uso per questo.

Ho visto esempi come questo:

```
  #ifdef FOO
    #
#else
    printf("Something");
#endif
```

che è solo cosmetico; la riga con il # solitario può essere eliminata senza effetti negativi.

O forse per consistenza cosmetica come questa:

```
#
#ifdef FOO
    x = 2;
#endif
#
#if BAR == 17
    x = 12;
#endif
#
```

Ma per quanto riguarda l'aspetto cosmetico è semplicemente brutto.

Un altro post menziona l'eliminazione dei commenti: in GCC, un commento dopo a un # non verrà visto dal compilatore. Di cui non dubito ma le specifiche non sembrano dire che questo è un comportamento standard.

Le mie ricerche delle motivazioni non stanno dando molti frutti. Quindi dirò semplicemente che questo è un buon vecchio C esoterico.

20. `struct` II: Più divertimento con `struct`

Adesso scopriamo che puoi fare molto di più con le `struct` di quanto abbiamo detto, sono solo tante cose aggregate. E dunque li inseriremo in questo capitolo.

Se sei bravo con le nozioni di base sulle `struct` puoi completare le tue conoscenze qui.

20.1. Inizializzatori di `structs` nidificati e array

Ricordi come potresti inizializzare i membri della struttura in questo modo?

```
struct foo x = {.a=12, .b=3.14};
```

Sembra che in questi inizializzatori c'è più potenza di quella che avevamo discusso in precedenza. Emozionante!

Per prima cosa se hai una sottostruttura nidificata come la seguente puoi inizializzare i membri di quella sottostruttura seguendo i nomi delle variabili lungo la riga:

```
struct foo x = {.a.b.c=12};
```

Diamo un'occhiata ad un esempio:

```
#include <stdio.h>

struct cabin_information {
    int window_count;
    int o2level;
};

struct spaceship {
    char *manufacturer;
    struct cabin_information ci;
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .ci.window_count = 8,   // <-- INIZIALIZZATORE ANNIDATO!
        .ci.o2level = 21
    };

    printf("%s: %d seats, %d%% oxygen\n",
        s.manufacturer, s.ci.window_count, s.ci.o2level);
}
```

Controlla le righe 16-17! È qui che inizializziamo i membri della `struct cabin_information` nella definizione di `s` nella nostra `struct spaceship`.

Ed ecco un'altra opzione per lo stesso inizializzatore—questa volta faremo qualcosa di più standard ma entrambi gli approcci funzionano:

```
struct spaceship s = {
    .manufacturer="General Products",
    .ci={
        .window_count = 8,
        .o2level = 21
    }
};
```

Ora come se le informazioni di cui sopra non fossero già abbastanza spettacolari possiamo anche inserire lì dentro anche gli inizializzatori di array.

Modifichiamolo per ottenere una serie di informazioni sui passeggeri e possiamo anche verificare come funzionano gli inizializzatori.

```
#include <stdio.h>

struct passenger {
    char *name;
    int covid_vaccinated; // Booleano
};

#define MAX_PASSENGERS 8

struct spaceship {
    char *manufacturer;
    struct passenger passenger[MAX_PASSENGERS];
};
```

```
int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .passenger = {
            // Initialize a field at a time
            [0].name = "Gridley, Lewis",
            [0].covid_vaccinated = 0,

            // Or all at once
            [7] = {.name="Brown, Teela", .covid_vaccinated=1},
        }
    };

    printf("Passengers for %s ship:\n", s.manufacturer);

    for (int i = 0; i < MAX_PASSENGERS; i++)
        if (s.passenger[i].name != NULL)
            printf("    %s (%svaccinated)\n",
                s.passenger[i].name,
                s.passenger[i].covid_vaccinated? "": "not ");
}
```

20.2. `struct` anonimo

Queste sono “le `struct` senza nome”. Li menzioniamo anche nella sezione `typedef` ma aggiorneremo qui.

Ecco un normale `struct`:

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

Ed ecco l'equivalente anonimo:

```
struct {              // <-- Senza nome!
    char *name;
    int leg_count, speed;
};
```

Okaaaaay. Quindi abbiamo una `struct` ma non ha nome quindi non abbiamo modo di usarla in seguito? Sembra piuttosto inutile.

Certo in quell'esempio lo è. Ma possiamo ancora usarlo in un paio di modi.

Uno di questi è raro ma poiché la `struct` anonima rappresenta un tipo possiamo semplicemente inserire alcuni nomi di variabili dopo e usarli.

```
struct {                    // <-- No name!
    char *name;
    int leg_count, speed;
} a, b, c;                  // 3 variables of this struct type

a.name = "antelope";
c.leg_count = 4;            // Per esempio
```

Ma non è ancora così utile.

Molto più comune è l'uso di `struct` anonime con `typedef` in modo da poterle utilizzare in

seguito (ad esempio per passare variabili alle funzioni).

```
typedef struct {                    // <-- Senza nome!
    char *name;
    int leg_count, speed;
} animal;                           // Nuovo tipo: animale

animal a, b, c;

a.name = "antelope";
c.leg_count = 4;            // Per esempio
```

Personalmente non uso molti `struct` anonimi. Penso che sia più piacevole vedere l'intera `struct animal` prima del nome della variabile in una dichiarazione.

Ma questa è solo la mia opinione amico.

20.3. Autoreferenziale `struct`

Per qualsiasi struttura dati simile a un grafico è utile poter avere puntatori ai nodi/vertici connessi. Ma questo significa che nella definizione di nodo è necessario avere un puntatore a un nodo. È pollo e uova!

Ma a quanto pare puoi farlo in C senza alcun problema.

Ad esempio ecco un nodo di elenco collegato:

```
struct node {
    int data;
    struct node *next;
};
```

È importante notare che il `next` è un puntatore. Questo è ciò che permette al tutto di essere costruito. Anche se il compilatore non sa ancora cosa possa essere l'intero `struct node` tutti i puntatori hanno la stessa dimensione.

Ecco un gustoso programma di elenchi collegati per testarlo:

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};

int main(void)
{
    struct node *head;

    // Hackishly set up a linked list (11)->(22)->(33)
    head = malloc(sizeof(struct node));
    head->data = 11;
    head->next = malloc(sizeof(struct node));
    head->next->data = 22;
    head->next->next = malloc(sizeof(struct node));
    head->next->next->data = 33;
    head->next->next->next = NULL;

    // Traverse it
    for (struct node *cur = head; cur != NULL; cur = cur->next) {
```

```
        printf("%d\n", cur->data);
    }
}
```

Eseguamo la stampa:

```
11
22
33
```

20.4. Membri dell'array flessibile

Ai bei vecchi tempi quando le persone scolpivano il codice C nel legno, alcune persone pensavano che sarebbe stato carino se avessero potuto allocare `struct` che avessero array di lunghezza variabile alla fine.

Voglio che sia chiaro che la prima parte della sezione è il vecchio modo di fare le cose, dopodiché faremo le cose nel modo nuovo.

Ad esempio potresti definire una `struct` per contenere le stringhe e la lunghezza di quella stringa. Avrebbe una lunghezza e un array per contenere i dati. Forse qualcosa del genere:

```
struct len_string {
    int length;
    char data[8];
};
```

Ma ha `8` codificato come lunghezza massima di una stringa e non è molto. E se facessimo qualcosa di *intelligente* e semplicemente `malloc()` aggiungessimo un po' di spazio extra alla fine dopo la struttura e poi lasciassimo che i dati uscissero in quello spazio?

Facciamolo e poi assegniamoci altri 40 byte:

```
struct len_string *s = malloc(sizeof *s + 40);
```

Poiché i `data` sono l'ultimo campo della `struct` se superiamo il campo si esaurirà nello spazio che abbiamo già allocato! Per questo motivo questo trucco funziona solo se l'array breve è *l'ultimo* campo nella `struct`.

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");
// Non crascerà. Probabilmente.
```

In effetti esisteva una soluzione alternativa comune al compilatore per eseguire questa operazione in cui alla fine si allocava un array di lunghezza zero:

```
struct len_string {
    int length;
    char data[0];
};
```

E poi ogni byte extra allocato era pronto per l'uso in quella stringa.

Poiché i `data` sono l'ultimo campo della `struct` se superiamo il campo si esaurirà nello spazio che abbiamo già allocato!

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");
// Non crascerà. Probabilmente.
```

Ma ovviamente l'accesso effettivo ai dati oltre la fine di tale array è un comportamento indefinito! In questi tempi moderni non ci degniamo più di essere a tanto selvaggi.

Fortunatamente per noi possiamo ancora ottenere lo stesso effetto con C99 e versioni successive ma ora è legale.

Cambiamo semplicemente la nostra definizione precedente per non avere alcuna dimensione per l'array[140] :

```
struct len_string {
    int length;
    char data[];
};
```

Ancora una volta funziona solo se il membro dell'array flessibile è /l'ultimo /campo nella `struct`.

E poi possiamo allocare tutto lo spazio che vogliamo per quelle stringhe `malloc()` eseguendo un metodo più grande della `struct len_string` come facciamo in questo esempio che crea una nuova `struct len_string` da una stringa C:

```
struct len_string *len_string_from_c_string(char *s)
{
    int len = strlen(s);

    // Assegna a "len" più byte di quelli di cui avremmo normalmente bisogno
    struct len_string *ls = malloc(sizeof *ls + len);

    ls->length = len;

    // Copia la stringa in quei byte extra
    memcpy(ls->data, s, len);

    return ls;
}
```

20.5. Byte di riempimento

Fai attenzione che C può aggiungere byte di riempimento all'interno o dopo una `struct` come ritiene opportuno. Non puoi fidarti che saranno direttamente adiacenti nella memoria[141] .

Diamo un'occhiata a questo programma. Diamo in output due numeri. Uno è la somma di `sizeof` i singoli tipi di campo. L'altro è la `sizeof` dell'intera `struct`.

Ci si aspetterebbe che fossero uguali. La dimensione del totale è la dimensione della somma delle sue parti giusto?

```
#include <stdio.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
```

140Tecnicamente diciamo che ha un tipo incompleto.

141Sebbene alcuni compilatori abbiano opzioni per forzare questo `occur–search for __attribute__((packed))` per vedere come farlo con GCC.

```
    printf("%zu\n", sizeof(int) + sizeof(char) + sizeof(int) + sizeof(char));
    printf("%zu\n", sizeof(struct foo));
}
```

Ma sul mio sistema questo viene visualizzato:

```
10
16
```

Non sono la stessa cosa! Il compilatore ha aggiunto 6 byte di riempimento per renderlo più performante. Forse hai ottenuto un output diverso con il tuo compilatore ma a meno che non lo stai forzando non puoi essere sicuro che non ci sia riempimento.

20.6. `offsetof`

Nella sezione precedente abbiamo visto che il compilatore può inserire byte di riempimento a piacimento all'interno di una struttura.

E se avessimo bisogno di sapere dove si trovano? Possiamo misurarlo con `offsetof`, definito in `<stddef.h>`.

Modifichiamo il codice di sopra per stampare gli offset dei singoli campi nel file `struct`:

```
#include <stdio.h>
#include <stddef.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", offsetof(struct foo, a));
    printf("%zu\n", offsetof(struct foo, b));
    printf("%zu\n", offsetof(struct foo, c));
    printf("%zu\n", offsetof(struct foo, d));
}
```

A me, rende questo outputs:

```
0
4
8
12
```

indicando che stiamo utilizzando 4 byte per ciascuno dei campi. È un po' strano perché `char` è solo 1 byte giusto? Il compilatore inserisce 3 byte di riempimento dopo ogni `char` in modo che tutti i campi siano lunghi 4 byte. Presumibilmente questo funzionerà più velocemente sulla mia CPU.

20.7. OOP falso

C'è una cosa leggermente abusiva che è una specie di OOP che puoi fare con le `struct`.

Dal momento che il puntatore a `struct` equivale a un puntatore al primo elemento di `struct` puoi liberamente lanciare un puntatore alla `struct` a un puntatore al primo elemento.

Ciò significa che possiamo creare una situazione come questa:

```
struct parent {
    int a, b;
};

struct child {
    struct parent super;  // DEVE essere il primo
    int c, d;
};
```

Quindi siamo in grado di passare un puntatore a una `struct child` a una funzione che si aspetta questo o un puntatore a una `struct parent`!

Perché `struct parent super` è il primo elemento in `struct child` un puntatore a qualsiasi `struct child` è lo stesso di un puntatore a quello `super` campo[142] .

Facciamo un esempio qui. Creeremo `struct` come sopra ma poi passeremo un puntatore ad a `struct child` a una funzione che necessita di un puntatore ad a ~struct parent~… e funzionerà ancora.

```
#include <stdio.h>

struct parent {
    int a, b;
};

struct child {
    struct parent super;  // DEVE essere il primo
    int c, d;
};

// Rendendo l'argomento `void*` in modo da potervi passare qualsiasi tipo
// (vale a dire un struct parent o struct child)
void print_parent(void *p)
{
    // Si apetta una struct parent, ma funzionerà anche una struct child
    // perché il puntatore punta alla struttura genitore nel primo
    // campo:
    struct parent *self = p;

    printf("Parent: %d, %d\n", self->a, self->b);
}

void print_child(struct child *self)
{
    printf("Child: %d, %d\n", self->c, self->d);
}

int main(void)
{
    struct child c = {.super.a=1, .super.b=2, .c=3, .d=4};

    print_child(&c);
    print_parent(&c);  // Also works even though it's a struct child!
}
```

Hai visto cosa abbiamo fatto nell'ultima riga di `main()`? Abbiamo chiamato `print_parent()` ma passato una `struct child*` come argomento! Anche se `print_parent()` necessita che l'argomento punti a una `struct parent` *ce la stiamo cavando* perché il primo campo nella

142super non è una parola chiave, per inciso. Sto solo rubando un po' di terminologia OOP.

`struct child` è una `struct parent`.

Ancora una volta funziona perché un puntatore a una `struct` ha lo stesso valore di un puntatore al primo campo in quella `struct`.

Tutto dipende da questa parte delle specifiche:

> §6.7.2.1¶15 […] Un puntatore ad un oggetto struttura opportunamente convertito punta al suo membro iniziale […] e viceversa.

e

> §6.5¶7 Un oggetto può avere accesso al suo valore memorizzato solo tramite un'espressione lvalue che ha uno dei seguenti tipi:

- un tipo compatibile con il tipo effettivo dell'oggetto
- […]

e la mia ipotesi che “opportunamente convertito” significhi “fuso nel tipo effettivo del membro iniziale”.

20.8. Campi di bit

Nella mia esperienza questi sono usati raramente ma potresti vederli di tanto in tanto specialmente nelle applicazioni di livello inferiore che raggruppano i pezzi in spazi più ampi.

Diamo un'occhiata ad alcuni codici per dimostrare un caso d'uso:

```
#include <stdio.h>

struct foo {
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};

int main(void)
{
    printf("%zu\n", sizeof(struct foo));
}
```

Per me questo stampa `16`. Il che ha senso poiché i `unsigned` sono 4 byte sul mio sistema.

Ma cosa succederebbe se sapessimo che tutti i valori che verranno archiviati in `a` e `b` potrebbero essere archiviati in 5 bit e che i valori in `c` e `d` potrebbero essere archiviati in 3 bit? Sono solo 16 bit in totale. Perché riservare loro 128 bit se ne useremo solo 16?

Bene possiamo dire a C di provare a inserire questi valori. Possiamo specificare il numero massimo di bit che i valori possono assumere (da 1 in su la dimensione del tipo contenitore).

Lo facciamo inserendo i due punti dopo il nome del campo seguito dalla larghezza del campo in bit.

```
struct foo {
    unsigned int a:5;
    unsigned int b:5;
    unsigned int c:3;
    unsigned int d:3;
};
```

Ora quando chiedo a C quanto è grande la mia `struct foo` mi dice 4! Erano 16 byte ma ora sono solo 4. Ha "compresso" quei 4 valori in 4 byte il che significa un risparmio di memoria quadruplicato.

Il compromesso è ovviamente che i campi a 5 bit possono contenere solo valori da 0 a 31 e i campi a 3 bit possono contenere solo valori da 0 a 7. Ma la vita è tutta una questione di compromessi dopo tutto.

20.8.1. Campi bit non adiacenti

Un problema: C combinerà solo campi di bit **adiacenti**. Se vengono interrotti da campi non bit, non ottieni alcun risparmio:

```
struct foo {
// sizeof(struct foo) == 16 (per me)
    unsigned int a:1;
// poiché a non è adiacente a c.
    unsigned int b;
    unsigned int c:1;
    unsigned int d;
};
```

In questo esempio poiché `a` non è adiacente a `c` sono entrambi “impacchettati” nei propri numeri `int`.

Quindi abbiamo un `int` ciascuno per `a`, `b`, `c` e `d`. Dato che i miei numeri `int` sono 4 byte, il totale è di 16 byte.

Una rapida riorganizzazione consente di risparmiare spazio da 16 byte a 12 byte (sul mio sistema):

```
struct foo {
// sizeof(struct foo) == 12 (per me)
    unsigned int a:1;
    unsigned int c:1;
    unsigned int b;
    unsigned int d;
};
```

E ora poiché `a` è accanto a `c`, il compilatore li mette insieme in un unico `int`.

Quindi abbiamo un `int` per una combinazione `a` e `c` e un `int` ciascuno per `b` e `d`. Per un totale complessivo di 3 `int` o 12 byte.

Metti insieme tutti i tuoi bitfield per fare in modo che il compilatore li combini.

20.8.2. Signed o Unsigned `int`

Se dichiari semplicemente che un campo bit è `int`, i diversi compilatori lo tratteranno come `signed` o `unsigned`. Proprio come la situazione con `char`.

Sii specifico riguardo alla firma quando usi i campi bit.

20.8.3. Campi bit senza nome

In alcune circostanze specifiche potrebbe essere necessario riservare alcuni bit per motivi hardware, ma non è necessario utilizzarli nel codice.

Ad esempio supponiamo che tu abbia un byte in cui i primi 2 bit hanno un significato l'ultimo bit ha un significato, ma i 5 bit centrali non vengono utilizzati da te[143] .

143Supponendo caratteri a 8 bit, ovvero `CHAR_BIT == 8`.

Potremmo fare qualcosa del genere:

```
struct foo {
    unsigned char a:2;
    unsigned char dummy:5;
    unsigned char b:1;
};
```

E funziona: nel nostro codice usiamo `a` e `b` ma mai `dummy`. È lì solo per mangiare 5 bit per assicurarsi che `a` e `b` siano nella casella "richiesta" (da questo esempio artificioso) posizioni all'interno del byte.

C ci consente un modo per ripulirlo: *campi di bit senza nome*. Puoi semplicemente lasciare il nome (`dummy`) in questo caso e C è perfettamente soddisfatto dello stesso effetto:

```
struct foo {
    unsigned char a:2;
    unsigned char :5;   // <-- campo di bit senza nome!
    unsigned char b:1;
};
```

20.8.4. Campi bit senza nome di larghezza zero

Un po' più di esoterismo qui fuori… Diciamo che stavi inserendo dei pezzi in un `unsigned int` e avevi bisogno di alcuni campi bit adiacenti da inserire nel *successivo* `unsigned int`.

Cioè, se lo fai:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:3;
    unsigned int d:4;
};
```

il compilatore li racchiude tutti in un solo `unsigned int`. Ma cosa succederebbe se avessi bisogno di `a` e `b` in un `int`, e `c` e `d` in uno diverso?

C'è una soluzione per questo: inserisci un campo bit senza nome di larghezza `0` dove vuoi che il compilatore ricominci con la pacchettizzazione dei bit in un `int` diverso:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int :0;   // <--Campo di bit senza nome di larghezza zero
    unsigned int c:3;
    unsigned int d:4;
};
```

È analogo a un'interruzione di pagina esplicita in un elaboratore di testi. Stai dicendo al compilatore: "Smetti di comprimere i bit in questo `unsigned` e inizia a comprimerli in quello successivo".

Aggiungendo il campo di bit senza nome di larghezza zero in quel punto il compilatore inserisce `a` e `b` in un `unsigned int` `c` e `d` in un altro `unsigned int`. Due in totale per una dimensione di 8 byte sul mio sistema (`unsigned int`s sono 4 byte ciascuno).

20.9. Unions

Fondamentalmente sono proprio come le `struct` che per il fatto che i campi si sovrappongono in

memoria. L'`union` sarà abbastanza grande solo per il campo più grande e potrai utilizzare solo un campo alla volta.

È un modo per riutilizzare lo stesso spazio di memoria per diversi tipi di dati.

Li dichiari proprio come le `struct` tranne che è `union`. Guarda questo:

```
union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};
```

Ora ci sono molti campi. Se questa fosse una `struct` il mio sistema mi direbbe che ci sono voluti 36 byte per contenerla tutta.

Ma è un'`union`, quindi tutti questi campi si sovrappongono nello stesso tratto di memoria. Il più grande è `int` (o `float`), occupando 4 byte sul mio sistema. E infatti se chiedo il `sizeof` di `union foo` mi dice 4!

Il compromesso è che puoi utilizzare portabilmente solo uno di questi campi alla volta. Tuttavia…

20.9.1. Unions and Type Punning

È possibile scrivere in modo non portabile su un campo di `union` e leggerlo da un altro!

Questa operazione si chiama type punning[144] e lo dovresti usare se sai davvero cosa stai facendo in genere con la programmazione a basso livello.

Poiché i membri di un union condividono la stessa memoria scrivere a un membro influisce necessariamente sugli altri. E se leggi da uno quello che è stato scritto da un altro ottieni degli effetti strani.

```
#include <stdio.h>

union foo {
    float b;
    short a;
};

int main(void)
{
    union foo x;

    x.b = 3.14159;

    printf("%f\n", x.b);  // 3.14159, abbastanza giusto

    printf("%d\n", x.a);  // Ma che dire di questo??
}
```

Sul mio sistema, stampa:

```
3.141590
```

```
4048
```

perché sotto il cofano la rappresentazione dell'oggetto per il float 3.14159 era la stessa della rappresentazione dell'oggetto per lo short 4048. Sul mio sistema. I risultati potrebbero variare.

144https://en.wikipedia.org/wiki/Type_punning

20.9.2. Puntatori a union

Se hai un puntatore a `union` puoi castare quel puntatore a qualsiasi tipo di campo in quell'`union` e ottenere i valori in questo modo.

In questo esempio vediamo che l'`union` contiene `int` e `float`. E otteniamo puntatori all'`union` ma li castiamo sui tipi `int*` e `float*` (il cast mette a tacere gli avvisi del compilatore). E poi se li dereferenziamo, vediamo che hanno i valori che abbiamo archiviato direttamente nell'`union`.

```
#include <stdio.h>

union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};

int main(void)
{
    union foo x;

    int *foo_int_p = (int *)&x;
    float *foo_float_p = (float *)&x;

    x.a = 12;
    printf("%d\n", x.a);           // 12
    printf("%d\n", *foo_int_p);    // 12, Ancora

    x.g = 3.141592;
    printf("%f\n", x.g);           // 3.141592
    printf("%f\n", *foo_float_p);  // 3.141592, Ancora
}
```

È vero anche il contrario. Se abbiamo un puntatore a un tipo all'interno dell'`union` possiamo castarlo a un puntatore all'`union` e accedere ai suoi membri.

```
union foo x;
int *foo_int_p = (int *)&x;             // Puntatore a campo int
union foo *p = (union foo *)foo_int_p;  // Torna al puntatore a union

p->a = 12;  // This line the same as...
x.a = 12;   // this one.
```

Tutto ciò ti fa solo capire è che dietro le quinte tutti questi valori in un'`union` iniziano nello stesso punto della memoria, ed è lo stesso in cui si trova l'intera `union`.

20.9.3. Sequenze iniziali comuni nelle unioni

Se hai un'`union` di `struct` e tutte quelle `struct` iniziano con una *sequenza iniziale comune* è valido per accedere ai membri di quella sequenza da qualsiasi membro dell'`union`.

Cosa?

Di seguito sono riportate due `struct` con una sequenza iniziale comune:

```
struct a {
    int x;     //
    float y;   // Sequenza iniziale comune
```

```
    char *p;
};

struct b {
    int x;     //
    float y;   // Sequenza iniziale comune

    double *p;
    short z;
};
```

Lo vedi? Il fatto è che iniziano con `int` seguito da `float`: questa è la sequenza iniziale comune. I membri nella sequenza delle `struct` devono essere di tipi compatibili. E lo vediamo con `x` e `y` che sono rispettivamente `int` e `float`.

Ora costruiamo un'unione di questi:

```
union foo {
    struct a sa;
    struct b sb;
};
```

Ciò che questa regola ci dice è che abbiamo la garanzia che i membri delle sequenze iniziali comuni siano intercambiabili nel codice. Che è:

- f.sa.x is the same as f.sb.x.

e

- f.sa.y is the same as f.sb.y.

Perché i campi `x` e `y` sono entrambi nella sequenza iniziale comune.

Inoltre i nomi dei membri nella sequenza iniziale comune non hanno importanza—tutto ciò che conta è che i tipi siano gli stessi.

Nel complesso questo ci consente di aggiungere in modo sicuro alcune informazioni condivise tra le `struct` nell'`union`. Il miglior esempio di ciò è probabilmente l'utilizzo di un campo per determinare il tipo di `struct` tra tutte le `struct` nell'`union` che è attualmente "in uso".

Cioè se questo non ci fosse permesso e passassimo l'~union~ a qualche funzione, come farebbe quella funzione a sapere quale membro dell' `union` dovrebbe guardare?

Dai un'occhiata a queste `struct`. Notare la sequenza iniziale comune:

```
#include <stdio.h>

struct common {
    int type;   // sequenza iniziale comune
};

struct antelope {
    int type;   // sequenza iniziale comune

    int loudness;
};

struct octopus {
    int type;   // sequenza iniziale comune

    int sea_creature;
    float intelligence;
```

```
};
```

Ora buttiamoli in un `union`:

```
union animal {
    struct common common;
    struct antelope antelope;
    struct octopus octopus;
};
```

Inoltre per favore, concedetemi queste due `#define` per la demo:

```
#define ANTELOPE 1
#define OCTOPUS  2
```

Finora qui non è successo nulla di speciale. Sembra che il campo `type` sia completamente inutile.

Ma ora creiamo una funzione generica che stampi un `union animal`. Deve in qualche modo essere in grado di dire se sta guardando una `struct antelope` o una `struct octopus`.

A causa della magia delle sequenze iniziali comuni può cercare il tipo di animale in uno qualsiasi di questi posti per un particolare `union animal x`:

```
int type = x.common.type;    \\ o...
int type = x.antelope.type;  \\ o...
int type = x.octopus.type;
```

Tutti questi si riferiscono allo stesso valore in memoria.

E come avrai intuito la `struct common` è lì quindi il codice può guardare in modo agnostico il tipo senza menzionare un particolare animale.

Diamo un'occhiata al codice per stampare un `union animal`:

```
void print_animal(union animal *x)
{
    switch (x->common.type) {
        case ANTELOPE:
            printf("Antelope: loudness=%d\n", x->antelope.loudness);
            break;

        case OCTOPUS:
            printf("Octopus : sea_creature=%d\n", x->octopus.sea_creature);
            printf("          intelligence=%f\n", x->octopus.intelligence);
            break;

        default:
            printf("Unknown animal type\n");
    }

}

int main(void)
{
    union animal a = {.antelope.type=ANTELOPE, .antelope.loudness=12};
    union animal b = {.octopus.type=OCTOPUS, .octopus.sea_creature=1,
                                        .octopus.intelligence=12.8};

    print_animal(&a);
    print_animal(&b);
}
```

Nota come sulla linea 29 stiamo solo passando nell'~union~—non abbiamo idea di quale tipo di `struct` animale sia in uso al suo interno.

Ma va bene! Perché alla riga 31 controlliamo la tipologia per vedere se è un'antilope o un polipo. E poi possiamo esaminare la `struct` corretta per ottenere i membri.

È sicuramente possibile ottenere lo stesso effetto usando solo `struct` ma puoi farlo in questo modo se vuoi gli effetti salva-memoria di un'~union~.

20.10. Union e Struct Senza nome

Sai come puoi avere una `struct` senza nome come questa:

```
struct {
    int x, y;
} s;
```

Ciò definisce una variabile `s` di tipo `struct` anonima (perché la `struct` non ha un tag name) con i membri `x` e `y`.

Quindi cose del genere sono valide:

```
s.x = 34;
s.y = 90;

printf("%d %d\n", s.x, s.y);
```

Scopriamo che puoi eliminare quelle `strutct` senza nome nelle `union` proprio come potresti aspettarti:

```
union foo {
    struct {        // senza nome!
        int x, y;
    } a;

    struct {        // senza nome!
        int z, w;
    } b;
};
```

E poi accedervi normalmente:

```
union foo f;

f.a.x = 1;
f.a.y = 2;
f.b.z = 3;
f.b.w = 4;
```

Nessun problema!

20.11. Passare e ritornare `struct` e `union`

Puoi passare una `struct` o un'`union` a una funzione in base al valore (al contrario di un puntatore ad esso)—una copia di quell'oggetto nel parametro verrà fatta per assegnazione come al solito.

Puoi anche restituire una `struct` o un'`union` da una funzione e che viene anche restituita per valore.

```
#include <stdio.h>
```

```
struct foo {
    int x, y;
};

struct foo f(void)
{
    return (struct foo){.x=34, .y=90};
}

int main(void)
{
    struct foo a = f();  // Viene eseguita la copia

    printf("%d %d\n", a.x, a.y);
}
```

Fatto divertente: se lo fai puoi usare il `.` operatore subito dopo la chiamata di funzione:

```
printf("%d %d\n", f().x, f().y);
```

(Ovviamente quell'esempio chiama la funzione due volte, in modo inefficiente.)

E lo stesso vale per la restituzione dei puntatori a `struct` e `union` —assicurati solo di utilizzare l'operatore freccia `->` in questo caso.

21. Caratteri e stringhe II

Abbiamo parlato di come i tipi di `char` siano in realtà solo piccoli tipi interi… ma è lo stesso per un carattere tra virgolette singole.

Ma una stringa tra virgolette doppie è di tipo `const char *`.

Risulta che ci sono pochi altri tipi di stringhe e caratteri e questo porta a uno delle tane del coniglio più infami del languaggio: l'intero mondo multibyte/wide/Unicode/localization.

Sbirceremo nella tana del coniglio ma non ci entreremo. …Ancora!

21.1. Sequenze di uscita

Siamo abituati a stringhe e caratteri con lettere, punteggiatura e numeri regolari:

```
char *s = "Hello!";
char t = 'c';
```

E se volessimo inserire alcuni caratteri speciali che non possiamo digitare sulla tastiera perché non esistono? (es. “€”), o anche se vogliamo un carattere che sia una virgoletta singola? Chiaramente non possiamo farlo:

```
char t = ''';
```

Per fare queste cose, usiamo qualcosa chiamato *sequenze di uscita*. Sono i caratteri backslash (`\`) seguito da un altro carattere. I due (o più) caratteri insieme hanno un significato speciale.

Per il nostro esempio di virgoletta singola possiamo inserire un escape (ovvero `\`) davanti alla virgoletta singola centrale per risolverlo:

```
char t = '\'';
```

Ora C sa che `\'` significa semplicemente una citazione regolare che vogliamo stampare non la fine della sequenza di caratteri.

In questo contesto puoi dire "backslash" o "uscita". (“uscire dalla citazione”) e gli sviluppatori C sapranno di cosa stai parlando. Inoltre “uscita” in questo contesto è diverso dal tasto `Esc` o dal codice ASCII `ESC`.

21.1.1. Fughe utilizzati di frequente

A mio modesto parere, questi sequenze di uscita costituiscono il 99,2%[145] di tutte le uscite.

Code	Description
`\n`	Carattere di nuova riga: durante la stampa continua l'output successivo sulla riga successiva
`\'`	Virgoletta singola—utilizzato per una costante di carattere apice singolo
`\"`	Doppia virgoletta—utilizzato per una doppia virgoletta in una stringa letterale
`\\`	Backslash—utilizzato per una lettera \ in una stringa o in un carattere

Ecco alcuni esempi di uscite e di cosa restituiscono una volta stampati.

```
printf("Use \\n for newline\n");  // Utilizzare \n per riga nuova
printf("Say \"hello\"!\n");       // Dì "hello"!
printf("%c\n", '\'');             // '
```

21.1.2. Uscite usate raramente

Ma ci sono altre uscite! Semplicemente non li vedi così spesso.

Code	Descrizione
`\a`	Allarme. Ciò fa sì che il terminale emetta un suono o lampeggi, o entrambi!
`\b`	Backspace. Sposta il cursore indietro di un carattere. Non elimina il carattere.
`\f`	Formfeed. Questo passa alla “pagina” successiva, ma non è molto moderno. Sul mio sistema, questo si comporta come `\v`.
`\r`	Return. Si muove all'inizio della stessa riga.

145Ho appena inventato quel numero, ma probabilmente non è lontano

`\t`	Horizontal tab. Passa alla tabulazione orizzontale successiva. Sulla mia macchina, questo si allinea su colonne che sono multipli di 8, ma YMMV.
`\v`	Vertical tab. Passa alla tabulazione verticale successiva. Sulla mia macchina, questo si sposta nella stessa colonna nella riga successiva.
`\?`	Literal question mark. A volte ne hai bisogno per evitare trigrafi, come mostrato di seguito.

---

1. Single Line Status Updates

Un caso d'uso per `\b` o `\r` è mostrare gli aggiornamenti di stato che appaiono sulla stessa riga dello schermo e non provocano lo scorrimento del display. Ecco un esempio che esegue un conto alla rovescia da 10. (Nota che questo fa uso della funzione POSIX non standard `sleep()` da `<unistd.h>` —se non usi Unix cerca la tua piattaforma e usa `sleep` per l'equivalente.)

```
#include <stdio.h>
#include <threads.h>

int main(void)
{
    for (int i = 10; i >= 0; i--) {
        printf("\rT minus %d second%s... \b", i, i != 1? "s": "");

        fflush(stdout);  // Force output to update

        // Sleep for 1 second
        thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    }

    printf("\rLiftoff!              \n");
}
```

Stanno accadendo parecchie cose sulla linea 7. Prima di tutto iniziamo con `\r` per portarci all'inizio della riga corrente poi sovrascriviamo tutto ciò che c'è con il conto alla rovescia attuale. (C'è un operatore ternario là fuori per assicurarci di stampare 1 `second` o invece di 1 `second` i.)

Inoltre c'è uno spazio dopo il … È così che sovrascriviamo correttamente l'ultimo `.` quando `i` scende da 10 a 9 e otteniamo una colonna più stretta. Provalo senza lo spazio e vedi cosa intendo.

E lo avvolgiamo con un `\b` per tornare indietro su quello spazio in modo che il cursore si trovi all'estremità esatta della riga in un modo esteticamente gradevole.

Nota che anche la riga 14 ha molti spazi alla fine per sovrascrivere i caratteri già presenti dal conto alla rovescia.

Infine abbiamo uno strano `fflush(stdout)` lì dentro Qualsiasi cosa significhi. La risposta breve è che la maggior parte dei terminali dispone di *buffer di linea* per impostazione predefinita il che significa che in realtà non visualizzano nulla finché non viene incontrato un carattere di nuova riga. Dal momento che non abbiamo una nuova riga (abbiamo solo `\r`) senza questa riga il programma rimarrebbe lì fino a quando `Liftoff!` e poi stampare tutto in un istante. `fflush()` sovrascrive

questo comportamento e impone che l'output venga eseguito *immediatamente*.

2. La fuga dal punto interrogativo

Perché preoccuparsi di questo? Dopotutto, funziona perfettamente:

```
printf("Doesn't it?\n");
```

E funziona bene anche con l'uscita:

```
printf("Doesn't it\?\n");   // Nota \?
```

E quindi qual è il punto??!

Diventiamo più enfatici con un altro punto interrogativo e un punto esclamativo:

```
printf("Doesn't it??!\n");
```

Quando lo compilo ricevo questo avviso:

```
foo.c: In function ‘main’:
foo.c:5:23: warning: trigraph ??! converted to | [-Wtrigraphs]
    5 |     printf("Doesn't it??!\n");
      |
```

E eseguirlo dà questo risultato improbabile:

```
Doesn't it|
```

Quindi *trigrafi*? Che diavolo è questo???!

Sono sicuro che rivisiteremo questo angolo polveroso del linguaggio più tardi ma in breve il compilatore cerca certe triplette di caratteri che iniziano con `??` e sostituisce altri caratteri al loro posto. Quindi se ti trovi su un terminale antico senza il simbolo pipe `(|)` sulla tastiera è possibile digitare Invece `??!`.

Puoi risolvere questo problema evitando il secondo punto interrogativo in questo modo:

```
printf("Doesn't it?\?!\n");
```

E poi si compila e funziona come previsto.

Al giorno d'oggi ovviamente nessuno usa mai i trigrafi. Ma questo `??!` a volte appare se lo usiamo in una stringa per enfatizzare.

21.1.3. Uscite numeriche

Inoltre esistono modi per specificare costanti numeriche o altri valori di carattere all'interno di stringhe o costanti di carattere.

Se conosci una rappresentazione ottale o esadecimale di un byte puoi includerla in una stringa o in una costante di carattere.

La tabella seguente contiene numeri di esempio, ma è possibile utilizzare qualsiasi numero esadecimale o ottale. Imbottirlo con zeri iniziali se necessario per leggere il conteggio corretto delle cifre.

Code	Descrizione
`\123`	Incorpora il byte con valore ottale `123`, 3

	cifre esatte.
`\x4D`	Incorpora il byte con valore esadecimale `4D`, 2 cifre.
`\u2620`	Incorpora il carattere Unicode nel punto di codice con valore esadecimale `2620`, 4 cifre.
`\U0001243F`	Incorpora il carattere Unicode nel punto di codice con valore esadecimale `1243F`, 8 cifre.

Ecco un esempio della notazione ottale meno comunemente usata per rappresentare la lettera `B` tra `A` e `C`. Normalmente questo verrebbe usato per qualche tipo di carattere speciale non stampabile, ma abbiamo altri modi per farlo, di seguito, e questa è solo una dimostrazione ottale:

```
printf("A\102C\n");  // 102 is `B` in ASCII/UTF-8
```

Nota che non c'è lo zero iniziale sul numero ottale quando lo includi in questo modo. Ma deve contenere tre caratteri quindi aggiungi zeri iniziali se necessario.

Ma oggigiorno è molto più comune utilizzare costanti esadecimali. Ecco una demo che non dovresti usare, ma dimostra l'incorporamento dei byte UTF-8 `0xE2`, `0x80` e `0xA2` in una stringa, che corrisponde all'Unicode carattere “punto elenco” (•).

```
printf("\xE2\x80\xA2 Bullet 1\n");
printf("\xE2\x80\xA2 Bullet 2\n");
printf("\xE2\x80\xA2 Bullet 3\n");
```

Produce il seguente output se utilizzi una console UTF-8 (o probabilmente spazzatura se non lo usi):

```
• Bullet 1
• Bullet 2
• Bullet 3
```

Bma è un pessimo modo di fare Unicode. Puoi usare la sequenza di uscita `\u` (16-bit) o `\U` (32-bit) fare riferimento a Unicode semplicemente tramite il numero del punto di codice. Il punto elenco è 2022 (hex) in Unicode puoi farlo così e ottenere risultati più portabili:

```
printf("\u2022 Bullet 1\n");
printf("\u2022 Bullet 2\n");
printf("\u2022 Bullet 3\n");
```

Assicurati di riempire `\u` con un numero sufficiente di zeri iniziali per arrivare a quattro caratteri e `\U` con abbastanza zeri per arrivare a otto.

Ad esempio il punto elenco potrebbe essere composto da `\U` e quattro zeri iniziali:

```
printf("\U00002022 Bullet 1\n");
```

Ma chi ha tempo per essere così prolisso?

22. Tipi enumerati: `enum`

C ci offre un altro modo per avere valori interi costanti per nome: `enum`.

Per esempio:

```
enum {
  ONE=1,
  TWO=2
};

printf("%d %d", ONE, TWO);  // 1 2
```

enum { ONE=1, TWO=2};printf("%d %d", ONE, TWO); // 1 2

In un certo senso, può essere migliore—o diverso—piuttosto che usare un `#define`. Differenze chiave:

- `enum` possono essere solo tipi interi.
- `#define` può definire qualsiasi cosa.
- `enum` vengono spesso visualizzati tramite il nome dell'identificatore simbolico in un debugger.
- `#defined` i numeri vengono visualizzati solo come numeri grezzi di cui è più difficile conoscere il significato durante il debug.

Poiché sono tipi interi possono essere utilizzati ovunque sia possibile utilizzare gli interi incluse le dimensioni dell'array e le istruzioni `case`.

Approfondiamo ulteriormente questo aspetto.

22.1. Comportamento di `enum`

22.1.1. Numerazione

`enum` vengono numerati automaticamente a meno che non li sovrascrivi.

Iniziano da `0` e aumentano automaticamente da lì per impostazione predefinita:

```
enum {
    SHEEP,  // Il valore è 0
    WHEAT,  // Il valore è 1
    WOOD,   // Il valore è 2
    BRICK,  // Il valore è 3
    ORE     // Il valore è 4
};

printf("%d %d\n", SHEEP, BRICK);  // 0 3
```

Puoi forzare particolari valori interi come abbiamo visto in precedenza:

```
enum {
  X=2,
  Y=18,
  Z=-2
};
```

I duplicati non sono un problema:

```
enum {
  X=2,
  Y=2,
  Z=2
};
```

se i valori vengono omessi la numerazione continua a contare i positivi a partire dall'ultimo valore

specificato. Per esempio:

```
enum {
  A,     // 0, valore iniziale predefinito
  B,     // 1
  C=4,   // 4, impostato manualmente
  D,     // 5
  E,     // 6
  F=3    // 3, impostato manualmente
  G,     // 4
  H      // 5
}
```

22.1.2. Virgole finali

Questo va benissimo se questo è il tuo stile:

```
enum {
  X=2,
  Y=18,
  Z=-2,   // <-- Virgola finale
};
```

È diventato più popolare nelle linguaggi degli ultimi decenni quindi potresti essere felice di vederlo.

22.1.3. Ambito

`enum` ha ambito come ti aspetteresti. Se nell'ambito del file l'intero file può vederlo. Se ha ambito in un blocco è locale a quel blocco.

È molto comune che `enum` sia definito nei file di intestazione in modo che possano essere `#include` nell'ambito del file.

22.1.4. Stile

Come hai notato è comune dichiarare i simboli `enum` in maiuscolo (con gli underscore).

Questo non è un requisito ma è una sintassi molto molto comune.

22.2. Il tuo `enum` è un tipo

Questa è una cosa importante da sapere su `enum`: sono un tipo analogamente a come una `struct` è un tipo.

Puoi assegnare loro un nome di tag in modo da poter fare riferimento al tipo in seguito e dichiarare variabili di quel tipo.

Ora poiché le `enum` sono tipi interi perché non usare semplicemente `int`?

In C la ragione migliore per ciò è la chiarezza del codice–è un modo carino e scrivibile per descrivere il tuo pensiero in codice. C (a differenza del C++) in realtà non impone alcun valore compreso nell'intervallo per una particolare `enum`.

Facciamo un esempio in cui dichiariamo una variabile `r` di tipo `enum resource` che possa contenere quei valori:

```
// Di nome enum, il tipo è "enum resource"

enum resource {
    SHEEP,
```

```
    WHEAT,
    WOOD,
    BRICK,
    ORE
};

// Dichiarare una variabile "r" di tipo "enum resource"

enum resource r = BRICK;

if (r == BRICK) {
    printf("I'll trade you a brick for two sheep.\n");
}
```

Puoi anche `typedef` questi ovviamente, anche se personalmente non mi piacciono.

```
typedef enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} RESOURCE;

RESOURCE r = BRICK;
```

Un'altra scorciatoia legale ma rara è dichiarare le variabili quando dichiari l' `enum`:

```
// Dichiara un enum e alcune variabili inizializzate di quel tipo:

enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

Puoi anche dare un nome all' `enum` in modo da poterlo utilizzare in seguito che è probabilmente ciò che vorrai fare nella maggior parte dei casi:

```
// Declare an enum and some initialized variables of that type:

enum resource {   // <-- il tipo è adesso "enum resource"
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

In breve, le `enum` sono un ottimo modo per scrivere codice pulito con ambito, tipizzato e pulito.

23. Puntatori III: puntatori a puntatori e altro

Da qui in poi verranno trattati alcuni utilizzi intermedi e avanzati del puntatore. Se non conosci bene i puntatori riguarda i capitoli precedenti sui puntatori e sull'aritmetica dei puntatori prima di iniziare con questo argomento.

23.1. Puntatori a puntatori

Se puoi avere un puntatore a una variabile e una variabile può essere un puntatore, puoi avere un puntatore a una variabile che sia essa stessa un puntatore?

SÌ! Questo è un puntatore a un puntatore ed è contenuto in una variabile di tipo puntatore-puntatore.

Prima di addentrarci voglio *avere un'idea* di come funzionano i puntatori ai puntatori.

Ricorda che un puntatore è solo un numero. È un numero che rappresenta un indice nella memoria del computer, in genere uno che contiene un valore a cui siamo interessati per qualche motivo.

Quel puntatore che è un numero deve essere memorizzato da qualche parte. E quel luogo è memoria come ogni altra cosa[146] .

Ma poiché è archiviato in memoria deve avere un indice in cui è archiviato giusto? Il puntatore deve avere un indice in memoria in cui è archiviato. E quell'indice è un numero. È l'indirizzo del puntatore. È un puntatore al puntatore.

Iniziamo con un puntatore regolare a un `int` ripreso dai capitoli precedenti:

```
#include <stdio.h>

int main(void)
{
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int

    printf("%d\n", *p);  // 3490
}
```

Abbastanza semplice vero? Abbiamo due tipi rappresentati: `int` e `int*`, e impostiamo `p` in modo che punti a `x`. Quindi possiamo dereferenziare `p` sulla riga 8 e stampare il valore 3490.

Ma come abbiamo detto possiamo avere un puntatore a qualsiasi variabile… quindi significa che possiamo avere un puntatore a `p`?

In altre parole di che tipo è questa espressione?

```
int x = 3490;  // Type: int
int *p = &x;   // Type: pointer to an int

&p  // <-- What type is the address of p? AKA a pointer to p?
```

Se `x` è un `int` quindi `&x` è un puntatore a un `int` che abbiamo memorizzato in `p` che è di tipo `int*`. E dopo? (Rileggi questo paragrafo finché lo chiedi!)

E quindi `&p` è un puntatore ad un `int*` AKA un “puntatore a un puntatore a un `int`~”. AKA “~`int`-puntatore-puntatore”.

Capito? (Rileggi questo paragrafo finché lo chiedi!)

Scriviamo questo tipo con due asterischi: `int **`. Vediamolo in azione.

```
#include <stdio.h>

int main(void)
{
```

146C'è del diavolo nei dettagli con i valori memorizzati solo nei registri, ma possiamo tranquillamente ignorarlo per i nostri scopi qui. Inoltre le specifiche C non prendono posizione su questi elementi di "registro" al di là della parola chiave registro, la cui descrizione non menziona i registri.

```
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int
    int **q = &p;  // Type: pointer to pointer to int

    printf("%d %d\n", *p, **q);  // 3490 3490
}
```

Creiamo alcuni indirizzi fittizi per i valori di cui sopra come esempi e vediamo come potrebbero apparire queste tre variabili in memoria. I valori degli indirizzi riportati di seguito sono stati inventati da me a scopo di esempio:

Variable	Stored at Address	Value Stored There
x	28350	3490—the value from the code
p	29122	28350—the address of x!
q	30840	29122—the address of p!

In effetti proviamolo sul serio sul mio computer[147] e stampiamo i valori dei puntatori con `%p` e farò di nuovo la stessa tabella con i riferimenti reali (stampato in esadecimale).

Variable	Stored at Address	Value Stored There
x	0x7ffd96a07b94	3490—the value from the code
p	0x7ffd96a07b98	0x7ffd96a07b94—the address of x!
q	0x7ffd96a07ba0	0x7ffd96a07b98—the address of p!

Puoi vedere che gli indirizzi sono gli stessi tranne l'ultimo byte quindi concentrati solo su quelli.

Sul mio sistema `int` sono 4 byte motivo per cui vediamo l'indirizzo salire di 4 da `x` a `p`[148] e poi sale di 8 da `p` a `q`. Sul mio sistema tutti i puntatori sono 8 byte.

Importa se è un `int*` o un `int**`? È un byte in più byte? No! Ricorda che tutti i puntatori sono indirizzi, cioè indici in memoria. E sulla mia macchina puoi rappresentare un indice con 8 byte… non importa cosa è memorizzato in quell'indice.

Ora controlla cosa abbiamo fatto alla riga 9 dell'esempio precedente: abbiamo *dereferenziato due* volte `q` per tornare al nostro `3490`.

Questo è l'aspetto importante dei puntatori e dei puntatori ai puntatori:

147È molto probabile che otterrai numeri diversi sul tuo.

148Non c'è assolutamente nulla nelle specifiche che dica che funzionerà sempre in questo modo, ma sembra che funzioni in questo modo sul mio sistema.

- Puoi ottenere un puntatore a qualsiasi cosa con & (incluso un puntatore!)
- Puoi ottenere l'oggetto a cui punta un puntatore * (incluso un puntatore!)

Quindi puoi pensare & per creare puntatori e * per l'inverso—va nella direzione opposta a ~&~—per arrivare alla cosa puntata.

In termini di tipo ogni volta che tu usi &, questo aggiunge un altro livello del puntatore al tipo.

If you have	Then you run	The result type is
int x	&x	int *
int *x	&x	int **
int **x	&x	int ***
int ***x	&x	int ****

E ogni volta che usi `dereference (*)` fa il contrario:

If you have	Then you run	The result type is
int ****x	*x	int ***
int ***x	*x	int **
int **x	*x	int *
int *x	*x	int

Tieni presente che puoi utilizzarne più di uno `*s` di fila per dereferenziare rapidamente, proprio come abbiamo visto nel codice di esempio con `**q` sopra. Ognuno elimina un livello di indiretto.

If you have	Then you run	The result type is
int ****x	***x	int *
int ***x	**x	int *
int **x	**x	int

Generalmente `&*E == E`[149] . Il dereferenziamento “annulla” l'indirizzo di.

Ma & non funziona allo stesso modo—puoi farlo solo uno alla volta e devi memorizzare il risultato in una variabile intermedia:

149Anche se `E` è `NULL`, risulta strano.

```
int x = 3490;     // Type: int
int *p = &x;      // Type: int *, pointer to an int
int **q = &p;     // Type: int **, pointer to pointer to int
int ***r = &q;    // Type: int ***, pointer to pointer to pointer to int
int ****s = &r;   // Type: int ****, you get the idea
int *****t = &s;  // Type: int *****
```

23.1.1. Puntatore a Puntatori e `const`

Se ricordi dichiarando un puntatore come questo:

```
int *const p;
```

significa che non puoi modificare `p`. Provare `p++` ti darebbe un errore in fase di compilazione.

Ma come funziona con `int **` o `int ***`? Dove va a finire la `const` e cosa significa?

Cominciamo con la parte semplice. Il `const` proprio accanto al nome della variabile si riferisce a quella variabile. Quindi se vuoi un'`int***` che non puoi cambiare puoi fare così:

```
int ***const p;

p++;  // Not allowed
```

Ma è qui che le cose diventano un po' strane.

E se avessimo questa situazione:

```
int main(void)
{
    int x = 3490;
    int *const p = &x;
    int **q = &p;
}
```

Quando lo compilo ricevo un avviso:

```
warning: initialization discards ‘const’ qualifier from pointer target type
    7 |     int **q = &p;
      |               ^
```

Cosa sta succedendo? Il compilatore ci sta dicendo qui che avevamo una variabile che era `const` e stiamo assegnando il suo valore a un'altra variabile che non è `const` nello stesso modo. La `“const”` viene scartata, che probabilmente non è quello che volevamo fare.

Il tipo di `p` è `int *const p` e quindi `&p` è il tipo `int *const *`. E proviamo ad assegnarlo a `q`.

Ma `q` è `int **`! Un tipo con diverso `const` anza sul primo `*`! Quindi riceviamo un avviso che il `const` in `p` di `int *const *` viene ignorato e buttato via.

Possiamo risolvere il problema assicurandoci che il tipo `q` è almeno `const` di `p`.

```
int x = 3490;
int *const p = &x;
int *const *q = &p;
```

E ora siamo felici.

Potremmo rendere `q` ancora più `const`. Così com'è sopra diciamo “~q~ non è esso stesso `const` ma la cosa a cui punta è `const`.” Ma potremmo renderli entrambi `const`:

```
int x = 3490;
int *const p = &x;
int *const *const q = &p;  // More const!
```

E funziona anche questo. Ora non possiamo modificare `q` o il puntatore `q` punta.

23.2. Valori multibyte

Abbiamo accennato di questo argomento precedentemente in vari punti, ma chiaramente non tutti i valori possono essere archiviati in un singolo byte di memoria. Molte cose occupano più byte di memoria (ammesso che non siano `char`), e posso dirti quanti byte usando `sizeof`. E puoi dire quale indirizzo è in memoria il *primo* byte dell'oggetto utilizzando l'operatore standard `&` dato che restituisce sempre l'indirizzo del primo byte.

Ed ecco un altro fatto divertente! Se esegui un'iterazione sui byte di qualsiasi oggetto ottieni la *sua rappresentazione dell'oggetto*. Due cose con la stessa rappresentazione dell'oggetto in memoria sono uguali.

Se vuoi scorrere la rappresentazione dell'oggetto dovresti farlo con i puntatori a `unsigned char`.

Creiamo la nostra versione di `memcpy()`[150] fa esattamente questo:

```
void *my_memcpy(void *dest, const void *src, size_t n)
{
    // Make local variables for src and dest, but of type unsigned char

    const unsigned char *s = src;
    unsigned char *d = dest;

    while (n-- > 0)   // For the given number of bytes
        *d++ = *s++;  // Copy source byte to dest byte

    // Most copy functions return a pointer to the dest as a convenience
    // to the caller

    return dest;
}
```

(Ci sono anche alcuni buoni esempi di post-incremento e post-decremento da studiare.)

È importante notare che la versione sopra riportata è probabilmente meno efficiente di quella fornita con il tuo sistema.

Ma puoi passare puntatori a qualsiasi cosa al suo interno e copierà quegli oggetti. Potrebbe essere `int*`, `struct animal*` o qualsiasi altra cosa.

Facciamo un altro esempio che stampa i byte di rappresentazione dell'oggetto di una `struct` in modo da poter vedere se c'è del riempimento e quali valori ha[151] .

```
#include <stdio.h>

struct foo {
    char a;
    int b;
};

int main(void)
```

150 https://beej.us/guide/bgclr/html/split/stringref.html#man-memcpy

151 Non è necessario che il compilatore C aggiunga byte di riempimento e i valori di eventuali byte di riempimento aggiunti sono indeterminati.

```
{
    struct foo x = {0x12, 0x12345678};
    unsigned char *p = (unsigned char *)&x;

    for (size_t i = 0; i < sizeof x; i++) {
        printf("%02X\n", p[i]);
    }
}
```

Ciò che abbiamo è una `struct foo` costruita in modo tale da incoraggiare un compilatore a iniettare byte di riempimento (anche se non è necessario). E poi otteniamo un `unsigned char *` al primo byte di `struct foo` variabile `x`.

Da lì tutto ciò che dobbiamo sapere è il `sizeof x` e possiamo scorrere quel numero di byte stampando i valori (in esadecimale per comodità).

L'esecuzione di questo fornisce un gruppo di numeri come output. L'ho annotato di seguito per identificare dove sono stati archiviati i valori:

```
12  | x.a == 0x12

AB  |
BF  | padding bytes with "random" value
26  |

78  |
56  | x.b == 0x12345678
34  |
12  |
```

Su tutti i sistemi `sizeof(char)` è 1, osserviamo che il primo byte nella parte superiore dell'output contiene il valore `0x12` che abbiamo memorizzato lì.

Poi abbiamo alcuni byte di riempimento—per me questi variavano da esecuzione a esecuzione.

Infine sul mio sistema `sizeof(int)` è 4, possiamo vedere quei 4 byte alla fine. Nota come sono gli stessi byte del valore esadecimale `0x12345678` ma stranamente in ordine inverso[152] .

Tutto ciò è una piccola sbirciatina sotto il cofano dei byte di un'entità più complessa in memoria.

23.3. Il puntatore `NULL` e lo zero

Queste cose possono essere usate in modo intercambiabile:

- `NULL`
- `0`
- `'\0'`
- `(void *)0`

Personalmente utilizzo sempre `NULL` quando intendo `NULL` ma di tanto in tanto potresti vedere altre varianti. Anche se `'\0'` (un byte con tutti i bit impostati a zero) confronterà anche uguale, è *strano* confrontarlo con un puntatore; dovresti confrontare `NULL` con il puntatore. (Naturalmente molte volte nell'elaborazione delle stringhe stai confrontando *l'oggetto a cui punta il puntatore* a '\0', ed è giusto.)

152Questo varierà a seconda dell'architettura ma il mio sistema è little endian, il che significa che il byte meno significativo del numero viene memorizzato per primo. I sistemi big endian avranno i primi 12 e gli ultimi 78. Ma le specifiche non dettano nulla su questa rappresentazione.

0 è chiamata *costante del puntatore nullo*, e se confrontato o assegnato a un altro puntatore viene convertito in un puntatore nullo dello stesso tipo.

23.4. Puntatori come numeri interi

Puoi castare puntatori a numeri interi e viceversa (poiché un puntatore è solo un indice in memoria), ma probabilmente avrai bisogno di farlo solo se stai facendo cose hardware di basso livello. I risultati di tali macchinazioni sono definiti dall'implementazione quindi non sono portabili. E potrebbero accadere *cose strane*.

Tuttavia C fornisce una garanzia: puoi convertire un puntatore in un tipo `uintptr_t` e sarai in grado di riconvertirlo in un puntatore senza perdere alcun dato.

`uintptr_t` è definito in `<stdint.h>`[153] .

Inoltre se ti piace dichiararlo puoi utilizzare `intptr_t` per lo stesso effetto.

23.5. Castare puntatori ad altri puntatori

Esiste solo una conversione sicura del puntatore:

1. Conversione in `intptr_t` o `uintptr_t`.
2. Conversione da e verso `void*`.

DUE! Due conversioni di puntatori sicure.

1. Conversione da e verso `char*` (o `signed char*/unsigned char*`).

TRE! Tre conversioni sicure!

1. Conversione da e verso un puntatore a una `struct` e un puntatore al suo primo membro e viceversa.

QUATTRO! Quattro conversioni sicure!

Se esegui il cast su un puntatore di un altro tipo e poi accedi all'oggetto a cui punta, il comportamento non è definito a causa di qualcosa chiamato *alias stretto*.

Il semplice vecchio *alias* si riferisce alla capacità di avere più di un modo per accedere allo stesso oggetto. I punti di accesso sono alias l'uno dell'altro.

Alias stretto dice che puoi accedere a un oggetto solo tramite puntatori ai *tipi compatibili* con quell'oggetto.

Ad esempio questo è assolutamente consentito:

```
int a = 1;
int *p = &a;
```

`p` è un puntatore a un `int` e punta a un tipo compatibile—vale a dire ~int~—quindi è oro.

Ma quanto segue non va bene perché `int` e `float` non sono tipi compatibili:

```
int a = 1;
float *p = (float *)&a;
```

Ecco un programma demo che esegue alcuni alias. Prende una variabile `v` di tipo `int32_t` e lo trasforma in un puntatore ad `struct words`. Quella `struct` contiene due `int16_t`. Questi tipi sono incompatibili e dunque violiamo le rigide regole di alias. Il compilatore presuppone che questi due puntatori non puntino mai allo stesso oggetto… ma stiamo facendo in modo che lo

153È una funzionalità opzionale quindi potrebbe non essere presente, ma probabilmente lo è.

facciano. Il che è sbagliato da parte nostra.

Vediamo se riusciamo a rompere qualcosa.

```
#include <stdio.h>
#include <stdint.h>

struct words {
    int16_t v[2];
};

void fun(int32_t *pv, struct words *pw)
{
    for (int i = 0; i < 5; i++) {
        (*pv)++;

        // Print the 32-bit value and the 16-bit values:

        printf("%x, %x-%x\n", *pv, pw->v[1], pw->v[0]);
    }
}

int main(void)
{
    int32_t v = 0x12345678;

    struct words *pw = (struct words *)&v;  // Violates strict aliasing

    fun(&v, pw);
}
```

Vedi come passo i due puntatori incompatibili a `fun()`? Uno dei tipi è `int32_t*` e l'altro lo è struct words*.

Ma entrambi puntano allo stesso oggetto: il valore a 32 bit inizializzato su `0x12345678`.

Quindi se guardiamo i campi in `struct words` dovremmo vedere le due metà a 16 bit di quel numero. Giusto?

E nel ciclo `fun()` incrementiamo il puntatore a `int32_t`. Questo è tutto. Ma poiché la `struct` punta alla stessa memoria anch'essa dovrebbe essere aggiornata allo stesso valore.

Quindi eseguiamolo e otteniamo questo, con il valore a 32 bit a sinistra e le due porzioni a 16 bit a destra. Dovrebbe corrispondere[154] :

```
12345679, 1234-5679
1234567a, 1234-567a
1234567b, 1234-567b
1234567c, 1234-567c
1234567d, 1234-567d
```

e lo fa… *FINO A DOMANI!*

Proviamolo compilando GCC con `-O3` e `-fstrict-aliasing`:

```
12345679, 1234-5678
1234567a, 1234-5679
1234567b, 1234-567a
1234567c, 1234-567b
1234567d, 1234-567c
```

154Sto stampando i valori a 16 bit invertiti poiché sono su una macchina little-endian e questo rende più facile la lettura qui.

Erano fuori di uno! Ma rimandano allo stessa memoria! Come può essere? Risposta: è un comportamento indefinito creare alias di memoria in questo modo. *Tutto è possibile*, ma non siamo sulla buona strada.

Se il tuo codice viola rigide regole di aliasing che funziona o meno dipende da come qualcuno decide di compilarlo. E questo è un peccato perché è fuori dal tuo controllo. A meno che tu non sia una specie di divinità onnipotente.

Improbabile, mi dispiace.

GCC può essere costretto a non utilizzare le rigide regole di aliasing con `-fno-strict-aliasing`. Compilando il programma demo sopra con `-O3` e questo flag fa sì che l'output sia quello previsto.

Infine *il gioco di parole* utilizza puntatori di tipo diverso per esaminare gli stessi dati. Prima dell'aliasing stretto questo genere di cose era abbastanza comune:

```
int a = 0x12345678;
short b = *((short *)&a);   // Violates strict aliasing
```

Se vuoi fare giochi di parole (relativamente) sicuri dai un'occhiata la sezione sulle unioni e sui tipi di gioco.

23.6. Differenze di puntatore

Come sai dalla sezione sull'aritmetica dei puntatori puoi sottrarre un puntatore da un altro[155] per ottenere la differenza tra loro nel conteggio degli elementi dell'array.

Ora il *tipo di differenza* dipende dall'implementazione, quindi potrebbe variare da sistema a sistema.

Per essere più portabile puoi memorizzare il risultato in una variabile di tipo `ptrdiff_t` definita in `<stddef.h>`.

```
int cats[100];

int *f = cats + 20;
int *g = cats + 60;

ptrdiff_t d = g - f;  // difference is 40
```

E puoi stamparlo anteponendo l'identificatore del formato intero con `t`:

```
printf("%td\n", d);  // Print decimal: 40
printf("%tX\n", d);  // Print hex:     28
```

23.7. Puntatori a funzioni

Le funzioni sono solo raccolte di istruzioni macchina in memoria, quindi non c'è motivo per cui non possiamo ottenere un puntatore alla prima istruzione della funzione.

E poi chiamarlo.

Ciò può essere utile per passare un puntatore a una funzione in un'altra funzione come argomento. Quindi il secondo poteva chiamare qualunque cosa fosse passata.

La parte difficile di questi però, è che C ha bisogno di sapere il tipo della variabile che è il puntatore alla funzione.

E vorrebbe davvero conoscere tutti i dettagli.

155Supponendo che puntino allo stesso oggetto array.

Come "questo è un puntatore a una funzione che accetta due argomenti `int` e restituisce `void`".

Come scrivi tutto questo in modo da poter dichiarare una variabile?

Bene, sembra che assomigli molto ad un prototipo di funzione tranne che con alcune parentesi extra:

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int, int);
```

Si noti inoltre che non è necessario fornire nomi ai parametri. Ma puoi se vuoi; vengono semplicemente ignorati.

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int a, int b);
```

Quindi ora che sappiamo come dichiarare una variabile, come facciamo a sapere cosa assegnarle? Come otteniamo l'indirizzo di una funzione?

Si scopre che esiste una scorciatoia proprio come quando si ottiene un puntatore a un array: puoi semplicemente fare riferimento al nome della cruda funzione senza parentesi. (Se preferisci puoi mettere un & davanti a questo, ma non è necessario e non è idiomatico.)

Una volta che hai un puntatore a una funzione puoi chiamarla semplicemente aggiungendo parentesi e un elenco di argomenti.

Facciamo un semplice esempio in cui creo effettivamente un alias per una funzione impostando un puntatore ad essa. Poi lo chiameremo.

Questo codice viene stampa 3490:

```
#include <stdio.h>

void print_int(int n)
{
    printf("%d\n", n);
}

int main(void)
{
    // Assign p to point to print_int:

    void (*p)(int) = print_int;

    p(3490);          // Call print_int via the pointer
}
```

Notare come il tipo di `p` rappresenta il valore restituito e i tipi di parametro di $\text{print}_{\text{int}}$. Deve farlo altrimenti C si lamenterà dei tipi di puntatori incompatibili.

Un altro esempio qui mostra come potremmo passare un puntatore a una funzione come argomento a un'altra funzione.

Scriveremo una funzione che accetta una coppia di argomenti interi più un puntatore a una funzione che opera su questi due argomenti. Poi stampa il risultato.

```
#include <stdio.h>
```

```
int add(int a, int b)
{
    return a + b;
}

int mult(int a, int b)
{
    return a * b;
}

void print_math(int (*op)(int, int), int x, int y)
{
    int result = op(x, y);

    printf("%d\n", result);
}

int main(void)
{
    print_math(add, 5, 7);   // 12
    print_math(mult, 5, 7);  // 35
}
```

Prenditi un momento per digerirlo. L'idea qui è che passeremo un puntatore a una funzione a `print_math()` e chiamerà quella funzione per fare un po' di matematica.

In questo modo possiamo cambiare il comportamento di `print_math()` passandogli un'altra funzione. Puoi vedere che lo facciamo alle righe 22-23 quando passiamo i puntatori alle funzioni `add` e `mult` rispettivamente.

Ora alla riga 13, penso che possiamo essere tutti d'accordo sulla dichiarazione della funzione di `print_math()` è uno spettacolo da vedere. E se puoi crederci, questo è in realtà piuttosto semplice rispetto ad alcune cose che puoi costruire[156] .

Ma per ora digeriamolo. Risulta che ci sono solo tre parametri ma sono un po' difficili da vedere:

```
//                    op             x      y
//              |------------------|  |---|  |---|
void print_math(int (*op)(int, int), int x, int y)
```

Il primo `op` è un puntatore a una funzione che accetta due `int` come argomenti e restituisce un `int`. Questo corrisponde alle dichiarazioni sia per `add()` che per `mult()`.

Il secondo e il terzo `x` e `y`, sono solo parametri `int` standard.

Lentamente e deliberatamente lascia che i tuoi occhi dichiarazione sulla firma mentre identifichi le parti funzionanti.Una cosa che mi colpisce sempre è la sequenza `(*op)(`, le parentesi e l'asterisco. Questo è in regalo: è un puntatore a una funzione.

Infine torna al capitolo Puntatori II per un esempio di puntatore a funzione utilizzando il metodo `qsort()` integrato.

24. Operazioni bit a bit

Queste operazioni numeriche consentono effettivamente di manipolare singoli bit nelle variabili, in

156Il linguaggio di programmazione Go ha tratto ispirazione per la sintassi della dichiarazione del tipo dall'opposto di ciò che fa C.

modo appropriato poiché il C è un linguaggio di basso livello[158] .

Se non hai familiarità con le operazioni bit a bit, Wikipedia ha un buon articolo su i bit a bit[159] .

24.1. Bit a bit AND, OR, XOR e NOT

Per ognuno di questi avvengono le consuete conversioni aritmetiche sugli operandi (che in questo caso deve essere di tipo intero) e quindi viene eseguita l'operazione bit a bit appropriata.

Operazione	Operatore	Esempio
AND	&	a = b & c
OR	|	a = b | c
XOR	^	a = b ^ c
NOT	~	a = ~c

Nota come sono simili agli operatori booleani && e ||.

Questi hanno varianti abbreviate di assegnazione simili a += e -=:

Operatore	Esempio	Equivalente a esteso
&=	a &= c	a = a & c
|=	a |= c	a = a | c
^=	a ^= c	a = a ^ c

24.2. Spostamento bit a bit

Per questi, le promozioni di numeri interi vengono eseguite su ciascun operando (che deve essere un tipo intero) e quindi viene eseguito uno spostamento bit a bit. Il tipo del risultato è il tipo dell'operando sinistro promosso.

I nuovi bit vengono riempiti con zeri, con una possibile eccezione annotata nel comportamento definito dall'implementazione di seguito.

Operazione	Operatore	Esempio
Shift left	<<	a = b << c

158 Non che gli altri linguaggi non lo facciano: lo fanno. È interessante notare come molti linguaggi moderni utilizzino gli stessi operatori per bit a bit del C.

159 https://en.wikipedia.org/wiki/Bitwise_operation

Shift right	>>	a = b >> c

C'è anche la stessa abbreviazione simile per lo spostamento:

Operatore	Esempio	Equivalente a esteso
>>=	a >>= c	a = a >> c
<<=	a <<= c	a = a << c

Osserva il comportamento indefinito: nessuno spostamento negativo e nessuno spostamento maggiore della dimensione dell'operando sinistro promosso.

Presta attenzione anche al comportamento definito dall'implementazione: se sposti a destra un numero negativo i risultati sono definiti dall'implementazione. (Va benissimo spostare a destra un signed `Int`, assicurati solo che sia positivo.)

25. Funzioni variadiche

Variadico è una parola elegante per funzioni che accettano un numero arbitrario di argomenti.

Una funzione regolare ad esempio, accetta un numero specifico di argomenti:

```
int add(int x, int y)
{
    return x + y;
}
```

Puoi chiamarlo solo con esattamente due argomenti che corrispondono ai parametri `x` e `y`.

```
add(2, 3);
add(5, 12);
```

Ma se lo provi con altro, il compilatore non te lo permetterà:

```
add(2, 3, 4);  // ERRORE
add(5);        // ERRORE
```

Le funzioni variadiche aggirano in una certa misura questa limitazione.

Ne abbiamo già visto un famoso esempio in `printf()`! Puoi passargli ogni genere di cose.

```
printf("Hello, world!\n");
printf("The number is %d\n", 2);
printf("The number is %d and pi is %f\n", 2, 3.14159);
```

Sembra che non gli importi quanti argomenti gli fornisci!

Beh, non è del tutto vero. Zero argomenti ti daranno un errore:

```
printf();  // ERRORE
```

Questo ci porta a uno dei limiti delle funzioni variadiche in C: devono avere almeno un argomento.

Ma a parte questo sono piuttosto flessibili e consentono persino agli argomenti di avere tipi diversi proprio come fa `printf()`.

Vediamo come funzionano!

25.1. Ellissi nell'ambito delle funzioni

Allora come funziona sintatticamente?

Quello che fai è mettere prima tutti gli argomenti che *devono* essere passati (e ricorda che ce ne deve essere almeno uno) e dopo, metti …. Come questo:

```
void func(int a, ...)   // Literally 3 dots here
```

Ecco del codice per dimostrarlo:

```
#include <stdio.h>

void func(int a, ...)
{
    printf("a is %d\n", a);
// Stampa "a is 2"
}

int main(void)
{
    func(2, 3, 4, 5, 6);
}
```

Quindi fantastico, possiamo ottenere il primo argomento che è nella variabile `a`, ma per quanto riguarda il resto degli argomenti? Come puoi arrivarci??

Ecco dove inizia il divertimento!

25.2. Ottenere gli argomenti aggiuntivi

Avrai bisogno di includere `<stdarg.h>` per far funzionare tutto questo.

Per prima cosa, utilizzeremo una variabile speciale di tipo `va_list` (elenco di argomenti variabili) per tenere traccia di quale variabile stiamo accedendo ogni volta.

L'idea è che iniziamo prima a elaborare gli argomenti con una chiamata a `va_start()`, elabora ogni argomento a turno con `va_arg()` e poi, una volta finito concludilo con `va_end()`.

Quando chiami `va_start()` è necessario passare *l'ultimo parametro denominato* (quello appena prima del …) quindi sa da dove iniziare a cercare gli argomenti aggiuntivi.

E quando chiami `va_arg()` per ottenere l'argomento successivo, devi dirgli il tipo di argomento da ottenere dopo.

Ecco una demo che somma un numero arbitrario di numeri interi. Il primo argomento è il numero di numeri interi da sommare. Ne approfitteremo per calcolare quante volte dobbiamo chiamare `va_arg()`.

```
#include <stdio.h>
#include <stdarg.h>

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);
// Inizia con gli argomenti dopo "count"
```

```
    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);
// Prendi il prossimo int

        total += n;
    }

    va_end(va);  // Tutto fatto

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));
// 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));
// 22 + 44 = 66
}
```

(Tieni presente che quando viene chiamato `printf()` utilizza il numero di `%d` (o qualunque cosa) nella stringa di formato per sapere quanti altri argomenti ci sono!)

Se la sintassi di `va_arg()` ti sembra strana (a causa di quel nome vago che fluttua lì dentro) non sei solo. Questi sono implementati con macro del preprocessore per ottenere tutta la magia giusta.

25.3. Funzionalità `va_list`

Qual è la variabile `va_list` che stiamo usando lassù? È una variabile opaca[160] che contiene informazioni su quale argomento affronteremo successivamente `va_arg()`. Vedi come chiamiamo `va_arg()` ripetutamente? La variabile `va_list` è un segnaposto che tiene traccia dei progressi fatti finora.

Ma dobbiamo inizializzare quella variabile su un valore ragionevole. È qui che entra in gioco `va_start()`.

Quando abbiamo chiamato `va_start(va, count)` sopra, dicevamo “Inizializza la variabile `va` in modo che punti all'argomento della variabile *immediatamente successivo* `count`.”

Ed è *per questo* che dobbiamo avere almeno una variabile con nome nella nostra lista di argomenti[162].

Una volta che hai il puntatore al parametro iniziale, puoi facilmente ottenere i valori degli argomenti successivi chiamando `va_arg()` ripetutamente. Quando lo fai devi passare la variabile `va_list` (così può continuare a tenere traccia di dove ti trovi), così come il tipo di argomento che stai per copiare.

160 Cioè noi umili sviluppatori non dovremmo sapere cosa c'è dentro o cosa significa. Le specifiche non determinano di cosa si tratta in dettaglio.

161 Onestamente sarebbe possibile rimuovere questa limitazione dal linguaggio, ma l'idea è che le macro `va_start(), va_arg() e va_end()` dovrebbero poter essere scritte in C. E per far sì che ciò accada, dobbiamo trovare un modo per inizializzare un puntatore alla posizione del primo parametro. E per fare ciò abbiamo bisogno del nome del primo parametro. Sarebbe necessaria un’estensione linguistica per renderlo possibile, e finora il comitato non ha trovato una motivazione per farlo.

162 Onestamente sarebbe possibile rimuovere questa limitazione dal linguaggio, ma l'idea è che le macro `va_start(), va_arg() e va_end()` dovrebbero poter essere scritte in C. E per far sì che ciò accada, dobbiamo trovare un modo per inizializzare un puntatore alla posizione del primo parametro. E per fare ciò abbiamo bisogno del nome del primo parametro. Sarebbe necessaria un’estensione linguistica per renderlo possibile, e finora il comitato non ha trovato una motivazione per farlo.

Sta a te come programmatore capire a quale tipo passerai `va_arg()`. Nell'esempio sopra abbiamo appena fatto `int`. Ma nel caso di `printf()` utilizza l'identificatore di formato per determinare quale tipo eseguire successivamente.

E quando hai finito chiama `va_end()` per concludere. **Devi** (dice la specifica) chiamalo su una particolare variabile `va_list` prima di decidere chiamare anche `va_start()` o `va_copy()` di nuovo su di esso. So che non abbiamo parlato ancora `va_copy()`.

Quindi l'avanzamento standard è:

- `va_start()` per inizializzare la tua variabile `va_list`
- Ripetutere `va_arg()` per ottenere i valori
- `va_end()` per deinizializzare la tua variabile `va_list`

Ho anche menzionato `va_copy()` lassù; crea una copia della variabile `va_list` nello stesso identico stato. Cioè se non avviato con `va_arg()` con la variabile source, anche quella nuova non verrà avviata. Se finora hai utilizzato 5 variabili con `va_arg()` anche la copia lo rifletterà.

`va_copy()` può essere utile se hai bisogno di scorrere gli argomenti in anticipo ma devi anche ricordare la tua posizione attuale.

25.4. Funzioni di libreria che utilizzano `va_list`

Uno degli altri usi per questi è molto figo: scrivendo la tua variante `printf()` personalizzata. Sarebbe una seccatura dover gestire tutti quegli specificatori di formato giusto? Tutti milioni di quelli?

Fortunatamente ci sono varianti `printf()` che accettano una `va_list` funzionante come argomento. Puoi usarli per concludere e creare il tuo personale `printf()`!

Queste funzioni iniziano con la lettera `v` come anche `vprintf()`, `vfprintf()`, `vsprintf()` e `vsnprintf()`. Praticamente tutte i vecchi gloriosi `printf()` tranne che con una `v` davanti.

Creiamo una funzione `my_printf()` che funziona come `printf()` tranne per il fatto che ci vuole un argomento in più prima.

```
#include <stdio.h>
#include <stdarg.h>

int my_printf(int serial, const char *format, ...)
{
    va_list va;

    // Do my custom work
    printf("The serial number is: %d\n", serial);

    // Then pass the rest off to vprintf()
    va_start(va, format);
    int rv = vprintf(format, va);
    va_end(va);

    return rv;
}

int main(void)
{
    int x = 10;
    float y = 3.2;
```

```
    my_printf(3490, "x is %d, y is %f\n", x, y);
}
```

Vedi cosa abbiamo fatto lì? Sulle righe 12-14 abbiamo iniziato una nuova variabile `va_list`, e poi l'ho semplicemente passato dentro `vprintf()`. E sa di volerlo fare, perché ha tutte le funzionalità di `printf()` integrate.

Dobbiamo comunque chiamare `va_end()` quando abbiamo finito quindi non dimenticarlo!

26. Localizzazione e internazionalizzazione

La *localizzazione* è il processo che rende la tua app pronta per funzionare bene in diverse località (o paesi).

Come forse saprai non tutti usano lo stesso carattere per i separatori decimali o per i separatori delle migliaia… o per valuta.

Queste impostazioni locali hanno nomi ed è possibile selezionarne una da utilizzare. Ad esempio una locale statunitense potrebbe scrivere un numero come:

100,000.00

Mentre in Brasile lo stesso potrebbe essere scritto con virgole e punti decimali invertiti:

100.000,00

Semplifica la scrittura del codice in modo che possa essere facilmente trasferito ad altre nazionalità!

Beh, più o meno. Risulta che C ha solo una locale integrata ed è limitata. Le specifiche lasciano davvero molta ambiguità qui; è difficile essere completamente portatili.

Ma faremo del nostro meglio!

26.1. Impostazione della localizzazione, rapida e sporca

Per queste chiamate includi `<locale.h>`.

C'è fondamentalmente una cosa che puoi fare in modo portabile qui in termini di dichiarazione di una locale specifica. Questo è probabilmente ciò che vuoi fare se hai intenzione di fare qualcosa a livello locale:

```
setlocale(LC_ALL, "");
// Utilizza le impostazioni
// locali di questo ambiente per tutto
```

Ti consigliamo di chiamarlo in modo che il programma venga inizializzato con la tua variabile locale.

Entrando più nel dettaglio, c'è un'altra cosa che puoi fare e rimanere portatile:

```
setlocale(LC_ALL, "C");
// Utilizzare la locale C predefinita
```

ma viene chiamato per impostazione predefinita ogni volta che si avvia il programma, quindi non c'è molto bisogno di farlo da soli.

In quella seconda stringa puoi specificare qualsiasi supportato locale dal tuo sistema. Questo dipende completamente dal sistema quindi varierà. Sul mio sistema posso specificarlo:

```
setlocale(LC_ALL, "en_US.UTF-8");
```

```
// Non portabile!
```

E funzionerà. Ma è portabile solo su sistemi che hanno lo stesso identico nome per la stessa identica localizzazione e non puoi garantirlo.

Passando una stringa vuota ("") per il secondo argomento stai dicendo a C: "Ehi, scopri qual è la locale corrente su questo sistema, così non devo dirtelo".”

26.2. Ottenere le impostazioni locali monetarie

Poiché spostare pezzi di carta verdi promette di essere la chiave della felicità[163] parliamo di contesto monetario. Quando scrivi codice portatile devi sapere cosa digitare per i contanti giusto? Che si tratti di "$", "€", "¥" o "£".

Come puoi scrivere quel codice senza impazzire? Fortunatamente una volta che chiami setlocale `(LC_ALL, "")` puoi semplicemente cercarli chiamando a `localeconv()`:

```
struct lconv *x = localeconv();
```

Questa funzione restituisce un puntatore a una `struct lconv` allocata staticamente che contiene tutte le informazioni interessanti che stai cercando.

Ecco i campi della `struct lconv` e il loro significato.

Innanzitutto alcune convenzioni. Un `_p_` significa "positivo" e `_n_` significa "negativo" e `int_` significa "internazionale". Sebbene molti di questi siano di tipo `char` o `char*` la maggior parte (o le stringhe a cui puntano) sono effettivamente trattati come numeri interi[164] .

Prima di andare oltre sappi che `CHAR_MAX` (da `<limits.h>`) è il valore massimo che può essere contenuto in un `char`. E che molti dei seguenti valori `char` lo utilizzano per indicare che il valore non è disponibile nella locale specificata.

Campo	Descrizione
`char *mon_decimal_point`	Carattere puntatore decimale per denaro, ad es `"."`.
`char *mon_thousands_sep`	Carattere separatore delle migliaia per denaro, ad es `","`.
`char *mon_grouping`	Descrizione del raggruppamento per denaro (vedi sotto).
`char *positive_sign`	Segno positivo per il denaro, ad es `"+"` o `""`.
`char *negative_sign`	Segno negativo per il denaro, ad es `"-"`.
`char *currency_symbol`	Simbolo di valuta, ad es `"$"`.

163“Questo pianeta ha, o meglio aveva, un problema: la maggior parte delle persone che vivevano su di esso erano infelici per gran parte del tempo. Sono state suggerite molte soluzioni per questo problema, ma la maggior parte di queste riguardava in gran parte il movimento di piccoli pezzi di carta verde, il che era strano perché nel complesso non erano i piccoli pezzi di carta verdi ad essere infelici. —Guida galattica per autostoppisti, Douglas Adams

164Ricorda che `char` è solo un numero intero di dimensioni byte

`char frac_digits`	Quando si stampano importi monetari, quante cifre stampare dopo il punto decimale, ad es `2`.
`char p_cs_precedes`	`1` se la `currency_symbol` viene prima del valore per un importo monetario non negativo, 0 se successivo.
`char n_cs_precedes`	`1` se il `currency_symbol` viene prima del valore per un importo monetario negativo, `0` se successivo.
`char p_sep_by_space`	Determina la separazione dei `currency symbol` dal valore per importi non negativi (vedi sotto).
`char n_sep_by_space`	Determina la separazione dei `currency symbol` dal valore per gli importi negativi (vedi sotto).
`char p_sign_posn`	Determina il `positive_sign` posizione per valori non negativi.
`char p_sign_posn`	Determina il `positive_sign` posizione per valori negativi.
`char *int_curr_symbol`	Simbolo di valuta internazionale, ad es `"USD "`.
`char int_frac_digits`	Valore internazionale per `frac_digits`.
`char int_p_cs_precedes`	Valore internazionale per `p_cs_precedes`.
`char int_n_cs_precedes`	Valore internazionale per `n_cs_precedes`.
`char int_p_sep_by_space`	Valore internazionale per `p_sep_by_space`.
`char int_n_sep_by_space`	Valore internazionale per `n_sep_by_space`.
`char int_p_sign_posn`	Valore internazionale per `p_sign_posn`.
`char int_n_sign_posn`	Valore internazionale per `n_sign_posn`.

26.2.1. Raggruppamento di cifre monetarie

OK, questo è stravagante. `mon_grouping` è un `char*` quindi potresti pensare che sia una stringa. Ma in questo caso no è davvero una serie di `char`. Dovrebbe sempre terminare con uno `0` o `CHAR_MAX`.

Questi valori descrivono come raggruppare insiemi di numeri in valuta a *sinistra* del decimale (l'intera parte numerica).

Ad esempio potremmo avere:

```
  2   1   0
 --- --- ---
$100,000,000.00
```

Questi sono gruppi di tre. Il gruppo 0 (appena a sinistra del decimale) ha 3 cifre. Il gruppo 1 (gruppo successivo a sinistra) ha 3 cifre e anche l'ultimo ne ha 3.

Quindi potremmo descrivere questi gruppi da destra (il decimale) a sinistra con una serie di valori interi che rappresentano le dimensioni del gruppo:

```
3 3 3
```

E questo funzionerebbe per valori fino a $100,000,000.

E se ne avessimo di più? Potremmo continuare ad aggiungere 3…

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

ma è pazzesco. Fortunatamente possiamo specificare 0 per indicare che la dimensione del gruppo precedente si ripete:

```
3 0
```

Il che significa ripetere ogni 3. Sarebbe bastato $100, $1,000, $10,000, $10,000,000, $100,000,000,000, e così via.

Puoi legittimamente impazzire con questi per indicare alcuni strani raggruppamenti.

Per esempio:

```
4 3 2 1 0
```

indicherebbe:

```
$1,0,0,0,0,00,000,0000.00
```

Un altro valore che può verificarsi è `CHAR_MAX`. Ciò indica che non dovrebbe verificarsi alcun ulteriore raggruppamento e può apparire ovunque nell'array incluso il primo valore.

```
3 2 CHAR_MAX
```

indicherebbe:

```
100000000,00,000.00
```

Per esempio.

E il semplice fatto di avere `CHAR_MAX` nella prima posizione dell'array ti direbbe che non ci sarebbe stato alcun raggruppamento.

26.2.2. Separatori e posizione dei segni

Tutte le varianti `sep_by_space` riguardano la spaziatura attorno al segno della valuta. I valori validi sono:

Valore	Descrizione
`0`	Nessuno spazio tra il simbolo della valuta e il valore.
`1`	Separa il simbolo della valuta (e il segno, se presente) dal valore con uno spazio.
`2`	Separare il simbolo del segno dal simbolo della valuta (se adiacente) con uno spazio, altrimenti separare il simbolo del segno dal valore con uno spazio.

Le varianti `sign_posn` sono determinate dai seguenti valori:

Value	Description
`0`	Metti tra parentesi il valore e il simbolo della valuta.
`1`	Metti la stringa del segno davanti al simbolo e al valore della valuta.
`2`	Inserisci la stringa del segno dopo il simbolo e il valore della valuta.
`3`	Metti la stringa del segno direttamente davanti al simbolo della valuta.
`4`	Metti la stringa del segno direttamente dietro il simbolo della valuta.

26.2.3. Valori di esempio

Quando ottengo i valori sul mio sistema questo è ciò che vedo (stringa di raggruppamento visualizzata come valori di byte singoli):

```
mon_decimal_point  = "."
```

```
mon_thousands_sep  = ","
mon_grouping       = 3 3 0
positive_sign      = ""
negative_sign      = "-"
currency_symbol    = "$"
frac_digits        = 2
p_cs_precedes      = 1
n_cs_precedes      = 1
p_sep_by_space     = 0
n_sep_by_space     = 0
p_sign_posn        = 1
n_sign_posn        = 1
int_curr_symbol    = "USD "
int_frac_digits    = 2
int_p_cs_precedes  = 1
int_n_cs_precedes  = 1
int_p_sep_by_space = 1
int_n_sep_by_space = 1
int_p_sign_posn    = 1
int_n_sign_posn    = 1
```

26.3. Specifiche di localizzazione

Notate come in precedenza abbiamo passato la macro `LC_ALL` a `setlocale()`… questo suggerisce che potrebbe esserci qualche variante che ti permette di essere più preciso su quali *parti* locali stai impostando.

Diamo un'occhiata ai valori che puoi vedere per questi:

Macro	**Descrizione**
`LC_ALL`	Imposta tutto quanto segue sulla locale specificata.
`LC_COLLATE`	Controlla il comportamento delle funzioni `strcoll()` e `strxfrm()`.
`LC_CTYPE`	Controlla il comportamento delle funzioni di gestione dei caratteri[165] .
`LC_MONETARY`	Controlla i valori restituiti da `localeconv()`.
`LC_NUMERIC`	Controlla il punto decimale per famiglia di funzioni `printf()`.
`LC_TIME`	Controlla la formattazione dell'ora di `strftime()` e `wcsftime()` funzioni di stampa di data e ora.

165Ad eccezione di `isdigit()` e `isxdigit()`.

È abbastanza comune da vedere `LC_ALL` venire impostati ma ehi, almeno hai delle opzioni.

Inoltre dovrei sottolineare che `LC_CTYPE` è uno dei pezzi grossi perchè si lega a caratteri ampi, un significant cesto di vermi di cui parleremo più avanti.

27. Unicode, caratteri estesi e tutto il resto

Prima di iniziare tieni presente che questa è un'area attiva dello sviluppo del linguaggio in C per funzionare deve superare alcuni, ehm, *dolori della crescita*. Quando uscirà C2x ci saranno probabili aggiornamenti qui.

La maggior parte delle persone è fondamentalmente interessata alla domanda apparentemente semplice, come posso usare questo e quel set di caratteri in C? Ci arriveremo. Ma come vedremo potrebbe già funzionare sul tuo sistema. Oppure potresti dover puntare su una libreria di terze parti.

Parleremo di molte cose in questo capitolo–alcuni sono indipendenti dalla piattaforma e altri sono specifici del C.

Innanzitutto diamo uno schema di ciò che vedremo:

- Il contesto di Unicode
- Situazione della codifica dei caratteri
- Set di caratteri di origine ed esecuzione
- Utilizzare Unicode e UTF-8
- Utilizzando altri tipi di caratteri come `wchar_t`, `char16_t`, e `char32_t`

Immergiamoci!

27.1. Cos'è Unicode?

In passato negli Stati Uniti e in gran parte del mondo era popolare utilizzare una codifica a 7 o 8 bit per i caratteri in memoria. Ciò significava che avremmo potuto avere 128 o 256 caratteri totali(compresi i caratteri non stampabili). Andava bene per un mondo incentrato sugli Stati Uniti, ma si scopriamo che in realtà ci sono altri alfabeti là fuori–chi lo sapeva? Il cinese ha più di 50.000 caratteri e non rientrano in un byte.

Quindi le persone hanno escogitato tutti i tipi di modi alternativi per rappresentare i propri set di caratteri personalizzati. E ciò andava bene ma si è trasformato in un incubo di compatibilità.

Per sfuggirgli fu inventato Unicode. Un set di caratteri per dominarli tutti. Si estende all'infinito (effettivamente) quindi non rimarremo mai a corto di spazio per nuovi caratteri. Contiene simboli cinesi, latini, greci, cuneiformi, degli scacchi, emoji… praticamente di tutto, davvero! E ne vengono aggiunti altri continuamente!

27.2. Punti codice

Voglio parlare di due concetti qui. Crea confusione perchè sono entrambi numeri… numeri diversi per la stessa cosa. Ma abbi pazienza.

Definiamo in modo approssimativo *punto di codice* per indicare un valore numerico che rappresenta un carattere. (I punti codice possono anche rappresentare caratteri di controllo non stampabili ma supponiamo che intenda qualcosa come la lettera "B" o il carattere "`$\pi$`".)

Ogni punto di codice rappresenta un carattere univoco. E a ogni carattere è associato un punto di codice numerico univoco.

Ad esempio in Unicode il valore numerico 66 rappresenta “B” e 960 rappresenta “`$\pi$`”. Altre mappature di caratteri che non sono Unicode utilizzano valori potenzialmente diversi, ma dimentichiamoli e concentriamoci su Unicode, il futuro!

Un dato di fatto: c'è un numero che rappresenta ogni carattere. In Unicode, questi numeri vanno da 0 a oltre 1 milione.

Chiaro?

Perchè stiamo per ribaltare un po' la situazione.

27.3. Encoding

Se ricordi un byte da 8 bit può contenere valori da 0 a 255 inclusi. È fantastico per "B" che è 66–che sta in un byte. Ma “`$\pi$`” è 960 e non sta in un byte! Abbiamo bisogno di un altro byte. Come immagazziniamo tutto ciò in memoria?? O che dire dei numeri più grandi ad esempio 195,024? Avrà bisogno di un certo numero di byte da conservare.

La grande domanda: come sono rappresentati questi numeri nella memoria? Questo è ciò che chiamiamo *codifica* dei caratteri.

Quindi abbiamo due cose: una è il punto di codice che ci dice effettivamente il numero seriale di un particolare carattere. E abbiamo la codifica che ci dice come rappresenteremo quel numero in memoria.

Ci sono molte codifiche. Puoi inventarne uno tuo adesso se vuoi[166] .Ma esamineremo alcune codifiche davvero comuni utilizzate con Unicode.

Encoding	Description
UTF-8	Una codifica orientata ai byte che utilizza un numero variabile di byte per carattere. Questo è uno da usare.
UTF-16	Una codifica a 16 bit per carattere[167] .
UTF-32	Una codifica a 32 bit per carattere.

Con UTF-16 e UTF-32 l'ordine dei byte è importante, quindi potresti vedere UTF-16BE per big-endian e UTF-16LE per little-endian. Lo stesso per UTF-32. Tecnicamente se non specificato dovresti assumere big-endian. Ma poichè Windows utilizza ampiamente UTF-16 ed è little-endian a volte si presume questo[168] .

Diamo un'occhiata ad alcuni esempi. Scriverò i valori in esadecimale perchè sono esattamente due cifre per byte da 8 bit e rende più facile vedere come sono organizzate le cose in memoria.

166Ad esempio potremmo memorizzare il punto di codice in un intero big-endian a 32 bit. Semplice! Abbiamo appena inventato una codifica! In realtà no; ecco cos'è la codifica `UTF-32BE`. Oh bene torniamo alla routine!

167Cioè. Tecnicamente ha una larghezza variabile: esiste un modo per rappresentare punti di codice più alti di `$2^{16}$` mettendo insieme due caratteri UTF-16.

168C'è un carattere speciale chiamato Byte Order Mark (BOM), punto di codice `0xFEFF` che può facoltativamente precedere il flusso di dati e indicare l'endianess. Tuttavia non è obbligatorio.

Character	Code Point	UTF-16BE	UTF-32BE	UTF-16LE	UTF-32LE	UTF-8
A	41	0041	00000041	4100	41000000	41
B	42	0042	00000042	4200	42000000	42
~	7E	007E	0000007E	7E00	7E000000	7E
π	3C0	03C0	000003C0	C003	C0030000	CF80
€	20AC	20AC	000020AC	AC20	AC200000	E282AC

Cerca lì gli schemi. Tieni presente che UTF-16BE e UTF-32BE sono semplicemente i punti di codice rappresentato direttamente come valori a 16 e 32 bit[169] .

Little-endian è lo stesso tranne che i byte sono in ordine little-endian.

Quindi alla fine abbiamo UTF-8. Innanzitutto potresti notare che i punti di codice a byte singolo sono rappresentati come un singolo byte. Bello. Potresti anche notare che diversi punti di codice richiedono un numero diverso di byte. Questa è una codifica a larghezza variabile.

Pertanto non appena superiamo un determinato valore, UTF-8 inizia a utilizzare byte aggiuntivi per memorizzare i valori. E non sembrano nemmeno essere correlati al valore del punto di codice.

I dettagli della codifica UTF-8[170] vanno oltre lo scopo di questa guida, ma è sufficiente sapere che ha un numero variabile di byte per punto di codice e che i valori di byte non corrispondono ai punti di codice *ad eccezione dei primi 128 punti di codice*. Se vuoi davvero saperne di più Computerphile ha un fantastico video UTF-8 con Tom Scott[171] .

Quest'ultimo aspetto è interessante su Unicode e UTF-8 dal punto di vista nordamericano: è retrocompatibile con la codifica ASCII a 7 bit! Quindi, se sei abituato ad ASCII UTF-8 è lo stesso! Ogni documento con codifica ASCII è anche codificato UTF-8! (Ma non il contrario, ovviamente.)

Probabilmente è proprio quest'ultimo punto, più di ogni altro a spingere UTF-8 a conquistare il mondo.

27.4. Set di caratteri di origine ed esecuzione

Quando si programma in C ci sono (almeno) tre set di caratteri in gioco:

- Quello con cui il tuo codice esiste sul disco.
- Quello in cui il compilatore lo traduce proprio all'inizio della compilazione (il *set di caratteri di origine*). Potrebbe essere uguale a quello sul disco oppure no.
- Quello in cui il compilatore traduce il set di caratteri di origine per l'esecuzione (il set di *caratteri di esecuzione*). Potrebbe essere lo stesso set di caratteri di origine oppure no.

Il tuo compilatore probabilmente ha opzioni per selezionare questi set di caratteri in fase di compilazione.

169Ancora una volta questo è vero solo in UTF-16 per i caratteri che rientrano in due byte.
170https://en.wikipedia.org/wiki/UTF-8
171https://www.youtube.com/watch?v=MijmeoH9LT4

Il set di caratteri di base sia per l'origine che per l'esecuzione conterrà i seguenti caratteri:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
space tab vertical-tab
form-feed end-of-line
```

Questi sono i caratteri che puoi utilizzare nella tua fonte e rimanere portatili al 100%..

Il set di caratteri di esecuzione avrà inoltre caratteri per avviso (campanello/flash), backspace, ritorno a capo e nuova riga.

Ma la maggior parte delle persone non arriva a questo estremo e usa liberamente i propri set di caratteri estesi nel sorgente e nell'eseguibile, soprattutto ora che Unicode e UTF-8 stanno diventando più comuni. Voglio dire, il set di caratteri di base non consente nemmeno @, $ o `!

In particolare è una seccatura (anche se possibile con le sequenze di escape) inserire caratteri Unicode utilizzando solo il set di caratteri di base.

27.5. Unicode in C

Prima di addentrarmi nella codifica in C, parliamo di Unicode dal punto di vista dei punti di codice. C'è un modo in C per specificare i caratteri Unicode e questi verranno tradotti dal compilatore nel set di caratteri di esecuzione[172] .

Quindi come lo facciamo?

Che ne dici del simbolo dell'euro punto di codice `0x20AC`. (L'ho scritto in esadecimale perchè entrambi i modi di rappresentarlo in C richiedono esadecimale.) Come possiamo inserirlo nel nostro codice C?

Usa `\u` uscita per inserirlo in una stringa ad es `"\u20AC"` (il caso dell'esadecimale non ha importanza). È necessario inserire esattamente **quattro cifre** esadecimali dopo `\u` aggiungendo zeri iniziali se necessario.

Ecco un esempio:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Quindi `\u` funziona con punti di codice Unicode a 16 bit, ma che dire di quelli più grandi di 16 bit? Per questo abbiamo bisogno della maiuscola: `\U`.

Per esempio:

```
char *s = "\U0001D4D1";

printf("%s\n", s);
// Stampa una lettera matematica "B"
```

È uguale a `\u` solo con 32 bit invece di 16. Sono equivalenti:

```
\u03C0
```

172Presumibilmente il compilatore fa del suo meglio per tradurre il punto di codice in qualunque sia la codifica dell'output ma non riesco a trovare alcuna garanzia nelle specifiche.

```
\U000003C0
```

Anche in questo caso questi vengono tradotti nel set di caratteri di esecuzione durante la compilazione. Rappresentano punti di codice Unicode non alcuna codifica specifica. Inoltre se un punto di codice Unicode non è rappresentabile nel set di caratteri di esecuzione, il compilatore può farne quello che vuole.

Ora potresti chiederti perchè non puoi semplicemente farlo:

```
char *s = "€1.23";

printf("%s\n", s);  // €1.23
```

E probabilmente puoi dato che hai un compilatore moderno. Il set di caratteri di origine verrà tradotto per te nel set di caratteri di esecuzione dal compilatore. Ma i compilatori sono liberi di vomitare se trovano caratteri che non sono inclusi nel loro set di caratteri esteso, e il simbolo € certamente non è nel set di caratteri di base.

Avvertenza dalle specifiche: non è possibile utilizzare `\u` o `\U` per codificare punti di codice inferiori a 0xA0 ad eccezione di 0x24 ($), 0x40 (@) e 0x60 (`)–sì, questi sono proprio i tre segni di punteggiatura comuni che mancano nel set di caratteri di base. Apparentemente questa restrizione sarà allentata nella prossima versione delle specifiche.

Infine puoi anche usarli negli identificatori nel tuo codice con alcune restrizioni. Ma non voglio entrare in questo argomento qui. In questo capitolo parleremo esclusivamente della gestione delle stringhe.

E questo è tutto per Unicode in C (eccetto la codifica).

27.6. Una breve nota su UTF-8 Prima di sterzare tra le erbacce

È possibile che i sorgenti sul disco, i caratteri sorgenti estesi e i caratteri di esecuzione estesi siano tutti in formato UTF-8. E le librerie si aspettano che usi UTF-8. Questo è il glorioso futuro di UTF-8 ovunque.

Se è così e non ti dispiace non essere portabile su sistemi differenti allora eseguilo e basta. Appiccica i caratteri Unicode nella tua fonte e nei dati a tuo piacimento. Usa le normali stringhe di C e sii felice.

Molte cose funzioneranno (anche se non in modo portabile) dato che le stringhe UTF-8 possono essere tranquillamente terminate con NUL proprio come qualsiasi altra stringa in C. Forse perdere la portabilità in cambio di una gestione più semplice dei caratteri è un compromesso che per te vale la pena.

Ci sono tuttavia alcuni avvertimenti:

- Cose come `strlen()` riportano il numero di byte in una stringabnon per forza il numero di caratteri. (Il `mbstowcs()` restituisce il numero di caratteri in una stringa quando la converti in caratteri estesi. POSIX lo estende in modo da poter passare `NULL` per il primo argomento se vuoi solo il conteggio dei caratteri.)
- Quanto segue non funzionerà correttamente con caratteri con più di un byte: `strtok()`, `strchr()` (usa invece `strstr()`), funzioni di tipo-`strspn()`, `toupper()`, `tolower()`, funzioni di tipo-`isalpha()` e probabilmente altro. Fai attenzione a tutto ciò che opera sui byte.
- `printf()` le sue varianti consentono di stampare solo un determinato numero di byte di una stringa[1/3] . Vuoi assicurarti di stampare il numero corretto di byte per terminare su un

limite di carattere.

- Se vuoi `malloc()` spazio per una stringa o dichiarare un array di `char` per uno, tieni presente che la dimensione massima potrebbe essere superiore a quanto ti aspettavi. Ogni carattere potrebbe richiedere fino a `MB_LEN_MAX` byte (da `<limits.h>`)–ad eccezione dei caratteri del set di caratteri di base che sono garantiti come un byte.

E probabilmente altri che non ho scoperto. Fammi sapere quali insidie ci sono là fuori…

27.7. Diversi tipi di caratteri

Voglio introdurre più tipi di caratteri. Siamo abituati a usare `char` giusto?

Ma è troppo facile. Rendiamo le cose molto più difficili! Sìì!

27.7.1. Caratteri multibyte

Prima di tutto voglio potenzialmente cambiare la tua idea su cosa una stringa (array di `char`) è. Si tratta di *stringhe multibyte* composte da *caratteri multibyte*.

Giusto–la tua stringa di caratteri ordinaria è multibyte. Quando qualcuno dice stringa C "intende una stringa multibyte C".

Anche se un particolare carattere nella stringa è costituito da un solo byte o se una stringa è composta da soli caratteri singoli, è nota come stringa multibyte.

Per esempio:

```
char c[128] = "Hello, world!";  // Stringa multibyte
```

Quello che stiamo dicendo qui è che un carattere particolare che non è nel set di caratteri di base potrebbe essere composto da più byte. Fino a `MB_LEN_MAX` (da `<limits.h>`). Certo, sembra solo un carattere sullo schermo ma potrebbero essere più byte.

Puoi anche inserire valori Unicode lì come abbiamo visto in precedenza:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Ma qui stiamo entrando in qualche stranezza perchè guarda questo:

```
char *s = "\u20AC1.23";  // €1.23

printf("%zu\n", strlen(s));  // 7!
```

La lunghezza della stringa "€1,23" è 7?! Sì! Bene sul mio sistema sì! Ricorda che `strlen()` restituisce il numero di byte nella stringa non il numero di caratteri. (Quando arriveremo ai caratteri larghi" vedremo un modo per ottenere il numero di caratteri nella stringa.)

Si noti che mentre C consente costanti di `char` multibyte individuali (al contrario di `char*`) il comportamento di questi varia in base all'implementazione e il compilatore potrebbe avvisarlo.

GCC, ad esempio avverte della presenza di costanti di caratteri multicarattere per le due righe seguenti (e sul mio sistema, stampa la codifica UTF-8):

```
printf("%x\n", '€');
printf("%x\n", '\u20ac');
```

27.7.2. Caratteri estesi

Se non sei un carattere multibyte allora sei un *carattere esteso*.

Un carattere esteso è un valore singolo che può rappresentare in modo univoco qualsiasi carattere nelle impostazioni internazionali correnti. È analogo ai punti di codice Unicode. Ma potrebbe non essere così. O potrebbe esserlo.

Fondamentalmente dove le stringhe di caratteri multibyte sono array di bytevle stringhe di caratteri estesi sono array di *caratteri*. Quindi puoi iniziare a pensare carattere per carattere anzichè byte per byte (l'ultimo dei quali diventa tutto confuso quando i caratteri iniziano a occupare un numero variabile di byte).

I caratteri estesi possono essere rappresentati da diversi tipi, ma quello più importante è `wchar_t`. È quello principale. È come `char`, a parte per il fatto che è .

Forse ti starai chiedendo se non riesci a capire se è Unicode o novcome ti consente tanta flessibilità in termini di scrittura del codice? `wchar_t` apre alcune di queste porte poichè esiste un ricco set di funzioni che puoi utilizzare per gestire le stringhe con `wchar_t` (come ottenere la lunghezza ecc.) senza preoccuparsi della codifica.

27.8. Utilizzo di caratteri estesi e wchar$_t$

È tempo di un nuovo tipo: `wchar_t`. Questo è il tipo di carattere ampio principale. Ricorda che un `char` è solo un byte? E un byte non è sufficiente per rappresentare potenzialmente tutti i caratteri? Beh, questo è abbastanza.

Per usare `wchar_t` includi `<wchar.h>`.

Quanti byte è grande? Beh non è del tutto chiaro. Potrebbero essere 16 bit. Potrebbe essere 32 bit.

Ma aspetta stai dicendo–se è solo 16 bit non è abbastanza grande da contenere tutti i punti di codice Unicode vero? Hai ragione–non lo è. Le specifiche non lo richiedono. Deve solo essere in grado di rappresentare tutti i caratteri nella posizione locale.

Ciò può causare problemi con Unicode su piattaforme a 16 bit `wchar_ts` (ahem–Windows). Ma questo non rientra nell'ambito di questa guida.

Puoi dichiarare una stringa o un carattere di questo tipo con il prefisso `L` e puoi stamparli con l'identificatore di formato `%ls` ("ell ess"). Oppure stamprlo individualmente un `wchar_t` con ~%lc.

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';

printf("%ls %lc\n", s, c);
```

Ora–quei caratteri sono memorizzati come punti di codice Unicode o no? Dipende dall'implementazione. Ma puoi verificare se lo sono con la macro `__STDC_ISO_10646__`. Se è definito la risposta è: "È Unicode!"

Più in dettaglio il valore in quella macro è un numero intero nella forma `yyyymm` che ti consente di sapere su quale standard Unicode puoi fare affidamento–qualunque cosa fosse in vigore in quella data.

Ma come usarli?

27.8.1. Conversioni Multibyte a `wchar_t`

Allora come possiamo passare dalle stringhe standard di byte alle stringhe larghe di caratteri e viceversa?

Possiamo usare un paio di funzioni di conversione delle stringhe per far sì che ciò accada.

Innanzitutto alcune convenzioni di denominazione che vedrai in queste funzioni:

- mb: multibyte
- wc: wide character
- mbs: multibyte string
- wcs: wide character string

Quindi se vogliamo convertire una stringa multibyte in una stringa di caratteri ampia possiamo chiamare il `mbstowcs()`. E viceversa: `wcstombs()`.

Funzione di conversione	Descrizione
`mbtowc()`	Converti un multibyte character ad un wide character.
`wctomb()`	Converti un wide character ad un multibyte character.
`mbstowcs()`	Converti un multibyte string ad un wide string.
`wcstombs()`	Converti un wide string ad un multibyte string.

Facciamo una breve demo in cui convertiamo una stringa multibyte in una stringa di caratteri ampia e confrontiamo le lunghezze delle due stringhe utilizzando le rispettive funzioni.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Esci dalla locale C
    // verso a una che
    // probabilmente ha
    // il simbolo dell'euro
    setlocale(LC_ALL, "");

    // Stringa multibyte originale
    // con il simbolo dell'euro
    // (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
```

```
    size_t mb_len = strlen(mb_string);

    // Array di caratteri estesi
    // che conterrà
    // la stringa convertita
    wchar_t wc_string[128];
    // Può contenere fino a
    // 128 caratteri larghi

    // Converti la stringa MB in WC;
    // questo restituisce il numero di wide chars
    size_t wc_len = mbstowcs(wc_string, mb_string, 128);

    // Print result--note the %ls for wide char strings
    printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
    printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
}
```

Sul mio sistema visualizzo questo:

```
multibyte: "The cost is €1.23" (19 bytes)
wide char: "The cost is €1.23" (17 characters)
```

(Il numero di byte del sistema potrebbe variare a seconda delle impostazioni locali.)

Una cosa interessante da notare è questa `mbstowcs()` oltre a convertire la stringa multibyte in wide restituisce la lunghezza (in caratteri) della stringa di caratteri estesi. Sui sistemi conformi a POSIX puoi sfruttare una modalità speciale in cui restituisce solo la lunghezza in caratteri di una determinata stringa multibyte: devi semplicemente passare `NULL` alla destinazione e `0` al numero massimo di caratteri da convertire (questo valore viene ignorato).

(Nel codice seguente sto utilizzando il mio set di caratteri di origine esteso: potresti doverli sostituire con `\u` escape.)

```
setlocale(LC_ALL, "");

// La seguente stringa ha 7 caratteri
size_t len_in_chars = mbstowcs(NULL, "§¶°±$\pi$€•", 0);

printf("%zu", len_in_chars);  // 7
```

Ancora una volta si tratta di un'estensione POSIX non portatile.

E naturalmente se vuoi convertire il contrario è `wcstombs()`.

27.9. Funzionalità dei caratteri estesi

Una volta che siamo nel territorio dei caratteri più ampi, abbiamo a nostra disposizione tutti i tipi di funzionalità. Qui riassumerò solo un sacco di funzioni ma fondamentalmente quello che abbiamo qui sono le versioni a caratteri estesi delle funzioni di stringa multibyte a cui siamo abituati. (Ad esempio conosciamo `strlen()` per stringhe multibyte; c'è un `wcslen()` per stringhe di caratteri estesi.)

27.9.1. `wint_t`

Molte di queste funzioni utilizzano `wint_t` per contenere singoli caratteri sia che vengano passati o restituiti.

È connesso a `wchar_t`. Un `wint_t` è un numero intero che può rappresentare tutti i valori nel set

di caratteri esteso e anche un carattere speciale di fine file `WEOF`.

Viene utilizzato da numerose funzioni di caratteri estesi orientate a un singolo carattere.

27.9.2. I/O Stream Orientation

Il tl;dr qui è di non mescolare e abbinare funzioni orientate ai byte (come `fprintf()`) con funzioni ad ampio raggio (come `fwprintf()`). Decidi se un flusso sarà orientato ai byte o all'ampiezza e manterrà questi tipi di funzioni di I/O.

Più in dettaglio: i stream possono essere orientati ai byte o per gli estesi. Quando uno stream viene creato per la prima volta non ha orientamento, ma la prima lettura o scrittura imposterà l'orientamento.

Se utilizzi per la prima volta un'operazione ampia (come `fwprintf()`) orienterà il flusso in modo ampio.

Se utilizzi per la prima volta un'operazione byte (come `fprintf()`) orienterà il flusso in byte.

Puoi impostare manualmente un flusso non orientato in un modo o nell'altro con una chiamata a `fwide()`. Puoi utilizzare la stessa funzione per ottenere l'orientamento di un flusso.

Se hai bisogno di cambiare l'orientamento a metà strada puoi farlo con `freopen()`.

27.9.3. I/O Functions

In genere includono `<stdio.h>` e `<wchar.h>` per questi.

I/O Funzione	Descrizione
`wprintf()`	Output della console formattato.
`wscanf()`	Input della console formattato.
`getwchar()`	Input della console basato sui caratteri.
`putwchar()`	Output della console basato sui caratteri.
`fwprintf()`	Output di file formattato.
`fwscanf()`	Ingresso file formattato.
`fgetwc()`	Input di file basato su caratteri.
`fputwc()`	Output di file basato su caratteri.
`fgetws()`	Input di file basato su stringhe.
`fputws()`	Output di file basato su stringhe.
`swprintf()`	Output di stringa formattata.

`swscanf()`	Input di stringa formattata.
`vfwprintf()`	Output di file con formattazione variadica.
`vfwscanf()`	Input di file con formattazione variadica.
`vswprintf()`	Output di stringa in formato variadico.
`vswscanf()`	Input di stringa in formato variadico.
`vwprintf()`	Output della console con formattazione variadica.
`vwscanf()`	Ingresso console con formattazione variadica.
`ungetwc()`	Reimposta un carattere ampio su un flusso di output.
`fwide()`	Ottieni o imposta l'orientamento multibyte/ ampio del flusso.

27.9.4. Funzioni di conversione del tipo

In genere includono `<wchar.h>` per questi.

Funzione di conversione	Descrizione
`wcstod()`	Converti stringa a `double`.
`wcstof()`	Converti stringa a `float`.
`wcstold()`	Converti stringa a `long double`.
`wcstol()`	Converti stringa a `long`.
`wcstoll()`	Converti stringa a `long long`.
`wcstoul()`	Converti stringa a `unsigned long`.
`wcstoull()`	Converti stringa a `unsigned long long`.

27.9.5. Funzioni di copia di stringhe e memoria

In genere includono `<wchar.h>` per questi.

Funzione di copia	Descrizione
`wcscpy()`	Copia stringa.
wcsncpy()~	Copia stringa, con lunghezza limitata.
wmemcpy()~	Copiare la memoria.
wmemmove()~	Copia la memoria potenzialmente sovrapposta.
wcscat()~	Concatenare stringhe.
wcsncat()~	Concatena stringhe, con lunghezza limitata.

27.9.6. Funzioni di confronto di stringhe e memoria

In genere includere `<wchar.h>` per questi.

Funzione di confronto	Descrizione
`wcscmp()`	Confronta le stringhe lessicograficamente.
`wcsncmp()`	Confronta le stringhe lessicograficamente, con lunghezza limitata.
`wcscoll()`	Confronta le stringhe nell'ordine del dizionario in base alla lingua.
`wmemcmp()`	Confrontare la memoria lessicograficamente.
`wcsxfrm()`	Trasforma le stringhe in versioni tali che `wcscmp()` si comporti come `wcscoll()` [175].

27.9.7. String Searching Functions

In genere includere `<wchar.h>` per questi.

Funzione di ricerca	Descrizione

174`wcscoll()` è uguale a `wcsxfrm()` seguito da `wcscmp()`.
175`wcscoll()` è uguale a `wcsxfrm()` seguito da `wcscmp()`.

`wcschr()`	Trova un carattere in una stringa.
`wcsrchr()`	Trova un carattere precedente in una stringa.
`wmemchr()`	Trova un carattere in memoria.
`wcsstr()`	Trova una sottostringa in una stringa.
`wcspbrk()`	Trova uno qualsiasi dei set di caratteri in una stringa.
`wcsspn()`	Trova la lunghezza della sottostringa incluso un qualsiasi set di caratteri.
`wcscspn()`	Trova la lunghezza della sottostringa prima di qualsiasi set di caratteri.
`wcstok()`	Trova token in una stringa.

27.9.8. Length/Miscellaneous Functions

In genere includi `<wchar.h>` per questi.

Length/Misc Function	Descrizione
`wcslen()`	Restituisce la lunghezza della stringa.
`wmemset()`	Imposta i caratteri in memoria.
`wcsftime()`	Output formattato di data e ora.

27.9.9. Funzioni di classificazione dei caratteri

Includere `<wctype.h>` per questi.

Length/Misc Function	Descrizione
`iswalnum()`	Vero se il carattere è alfanumerico.
`iswalpha()`	Vero se il carattere è alfabetico.
`iswblank()`	Vero se il carattere è vuoto (spazio, ma non una nuova riga).

`iswcntrl()`	Vero se il carattere è un carattere di controllo.
`iswdigit()`	Vero se il carattere è una cifra.
`iswgraph()`	Vero se il carattere è stampabile (eccetto space).
`iswlower()`	Vero se il carattere è minuscolo.
`iswprint()`	Vero se il carattere è stampabile (compreso lo spazio).
`iswpunct()`	Vero se il carattere è di punteggiatura.
`iswspace()`	Vero se il carattere è uno spazio bianco.
`iswupper()`	Vero se il carattere è maiuscolo.
`iswxdigit()`	Vero se il carattere è una cifra esadecimale.
`towlower()`	Converti il carattere in minuscolo.
`towupper()`	Converti il carattere in maiuscolo.

27.10. Parse State, Restartable Functions

Entreremo un po' nel vivo della conversione multibyte, ma è una buona cosa da capire concettualmente.

Immagina come il tuo programma prende una sequenza di caratteri multibyte e li trasforma in caratteri estesi o viceversa. Potrebbe ad un certo punto essere in fase di analisi di un carattere o potrebbe dover attendere più byte prima di determinare il valore finale.

Questo stato di analisi viene archiviato in una variabile opaca di tipo `mbstate_t` e viene utilizzato ogni volta che viene eseguita la conversione. È così che le funzioni di conversione tengono traccia di dove si trovano a metà lavoro.

E se passi a una sequenza di caratteri diversa nel mezzo del flusso o provi a cercare un punto diverso nella sequenza di input potrebbe confondersi.

Ora potresti chiamarmi per questo: abbiamo appena fatto alcune conversioni, sopra, e non ho mai menzionato alcun `mbstate_t` da nessuna parte.

Questo perchè le funzioni di conversione come `mbstowcs()`, `wctomb()` ecc. ognuno ha la propria variabile `mbstate_t` usata. Ce n'è solo uno per funzione, quindi se stai scrivendo codice multithread non sono sicuri da usare.

Fortunatamente C definisce versioni *riavviabili* di queste funzioni in cui puoi passare il tuo `mbstate_t` in base al thread, se necessario. Se stai facendo cose multithread usa questi!

Nota rapida sull'inizializzazione di una variabile `mbstate_t`: usa `memset()` a zero. Non esiste

una funzione integrata per forzarne l'inizializzazione.

```
mbstate_t mbs;

// Set the state to the initial state
memset(&mbs, 0, sizeof mbs);
```

Di seguito è riportato un elenco delle funzioni di conversione riavviabili–notare la convenzione di denominazione di inserire una "`r`" dopo il tipo "da".:

- `mbrtowc()` –da caratteri multibyte a estesi
- `wcrtomb()` –carattere estesi a multibyte
- `mbsrtowcs()` –da una stringa multibyte a una stringa di caratteri estesi
- `wcsrtombs()` –stringa di caratteri estesi in stringa multibyte

Sono molto simili alle loro controparti non riavviabili tranne che richiedono il passaggio di un puntatore alla propria variabile `mbstate_t`. Inoltre modificano il puntatore della stringa sorgente (per aiutarti se vengono trovati byte non validi), quindi potrebbe essere utile salvare una copia dell'originale.

Ecco l'esempio di prima nel capitolo rielaborato per passare al nostro `mbstate_t`.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Esci dalla locale C verso una che probabilmente ha il simbolo dell'euro
    setlocale(LC_ALL, "");

    // Stringa multibyte originale con il simbolo dell'euro (Unicode point
20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Array di caratteri estesi che conterrà la stringa convertita
    wchar_t wc_string[128];  // Può contenere fino a 128 caratteri larghi

    // Set up the conversion state
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);  // Stato iniziale

    // mbsrtowcs() modifica il puntatore di input in modo che punti al primo
    // carattere non valido o NULL in caso di esito positivo. Facciamo una
copia del
    // puntatore per mbsrtowcs() per fare confusione, quindi il nostro
originale
    // è invariato.
    //
    // Questo esempio probabilmente avrà successo, ma controlliamo più a fondo
    // giù per vedere.
    const char *invalid = mb_string;

    // Converti la stringa MB in WC; questo restituisce il numero di wide chars
```

```
    size_t wc_len = mbsrtowcs(wc_string, &invalid, 128, &mbs);

    if (invalid == NULL) {
        printf("No invalid characters found\n");

        // Print result--note the %ls for wide char strings
        printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
        printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
    } else {
        ptrdiff_t offset = invalid - mb_string;
        printf("Invalid character at offset %td\n", offset);
    }
}
```

Per le funzioni di conversione che gestiscono il proprio stato è possibile reimpostare il proprio stato interno su quello iniziale passando `NULL` per gli argomenti `char*` ad esempio:

```
mbstowcs(NULL, NULL, 0);
// Reimposta lo stato di analisi per mbstowcs()
mbstowcs(dest, src, 100);
// Analizza alcune cose
```

Per l'I/O ciascun flusso ampio gestisce il proprio `mbstate_t` e lo utilizza per le conversioni di input e output man mano che procede.

E alcune delle funzioni I/O orientate ai byte come `printf()` e `scanf()` mantengono il proprio stato interno mentre svolgono il proprio lavoro.

Infine queste funzioni di conversione riavviabili hanno effettivamente il proprio stato interno se si passa `NULL` per il parametro `mbstate_t`. Ciò li fa comportare in modo più simile alle loro controparti non riavviabili.

27.11. Codifiche Unicode e C

In questa sezione vedremo cosa può (e cosa non può) fare il C quando si tratta di tre codifiche Unicode specifiche: UTF-8, UTF-16 e UTF-32.

27.11.1. UTF-8

Un rinfresco prima di questa sezione leggere la nota rapida UTF-8 sopra.

A parte questo quali sono le funzionalità UTF-8 di C?

Beh, non molto purtroppo.

Puoi dire a C che desideri specificamente che una stringa letterale sia codificata UTF-8 e lo farà per te. È possibile prefissare una stringa con `u8`:

```
char *s = u8"Hello, world!";

printf("%s\n", s);
// Hello, world!--se puoi produrre UTF-8
```

Ora puoi inserire i caratteri Unicode lì dentro?

```
char *s = u8"€123";
```

Sicuro! Se il set di caratteri di origine esteso lo supporta. (gcc lo fa.)

E se così non fosse? Puoi specificare un punto di codice Unicode con il tuo amichevole vicino `\u` e `\U` come notato sopra.

Ma questo è tutto. Non esiste un modo portatile nella libreria standard per prendere input arbitrari e trasformarlo in UTF-8 a meno che la tua locale non sia UTF-8. Oppure per analizzare UTF-8 a meno che la tua locale non sia UTF-8.

Quindi se vuoi farlo deve essere in una locale UTF-8 e:

```
setlocale(LC_ALL, "");
```

oppure scopri un nome locale UTF-8 sul tuo computer locale e impostalo esplicitamente in questo modo:

```
setlocale(LC_ALL, "en_US.UTF-8");
// Nome non portabile
```

Oppure utilizza una libreria di terze parti.

27.11.2. UTF-16, UTF-32, `char16_t` e `char32_t`

`char16_t` e `char32_t` sono un paio di altri tipi di caratteri potenzialmente ampi con dimensioni rispettivamente di 16 bit e 32 bit. Non necessariamente ampio perchè se non possono rappresentare tutti i caratteri nella localizzazione attuale, perdono la loro natura di caratteri estesi. Ma le specifiche li definiscono ovunque come tipi a "carattere esteso" quindi eccoci qui.

Questi sono qui per rendere le cose un po' più compatibili con Unicode potenzialmente.

Usare, per includere `<uchar.h>`. (Quella è "u", non "w".)

Questo file di intestazione non esiste su OS X–peccato. Se vuoi solo i tipi puoi:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

Ma se vuoi anche le funzioni dipende tutto da te.

Supponendo che tu sia ancora a posto puoi dichiarare una stringa o un carattere di questi tipi con i prefissi `u` e `U`:

```
char16_t *s = u"Hello, world!";
char16_t c = u'B';

char32_t *t = U"Hello, world!";
char32_t d = U'B';
```

Ora–sono valori memorizzati in UTF-16 o UTF-32? Dipende dall'implementazione.

Ma puoi fare un test per vedere se lo sono. Se le macro `__STDC_UTF_16__` o `__STDC_UTF_32__` sono definiti (a `1`), significa che i tipi contengono rispettivamente UTF-16 o UTF-32.

Se sei curioso e so che lo sei, i valori se UTF-16 o UTF-32 sono memorizzati nell'endianess nativo. Cioè dovresti essere in grado di confrontarli direttamente con i valori dei punti di codice Unicode:

```
char16_t pi = u"\u03C0";
// simbolo pi greco

#if __STDC_UTF_16__
pi == 0x3C0;  // Sempre vero
#else
pi == 0x3C0;  // Probabilmente non è vero
```

```
#endif
```

27.11.3. Conversioni multibyte

Puoi convertire dalla tua codifica multibyte a `char16_t` o `char32_t` con una serie di funzioni di supporto.

(Come ho detto però, il risultato potrebbe non essere UTF-16 o UTF-32 a meno che la macro corrispondente non sia impostata su `1`.)

Tutte queste funzioni sono riavviabili (ovvero si passa al proprio `mbstate_t`) e tutti operano carattere per carattere[176] .

Funzione di conversione	Descrizione
`mbrtoc16()`	Converte un carattere multibyte in un carattere `char16_t`.
`mbrtoc32()`	Converte un carattere multibyte in un carattere `char32_t`.
`c16rtomb()`	Converte un carattere `char16_t` in un carattere multibyte.
`c32rtomb()`	Converte un carattere `char32_t` in un carattere multibyte.

27.11.4. Librerie di terze parti

Per la conversione intensiva tra diverse codifiche specifiche, ci sono un paio di librerie mature che vale la pena controllare. Tieni presente che non ho usato nessuno dei due.

- iconv[177] –Internationalization Conversion, un'API comune standard POSIX disponibile sulle principali piattaforme.
- ICU[178] –International Components for Unicode. Almeno un blogger lo ha trovato facile da usare.

Se avete librerie più degne di nota fatemelo sapere.

28. Uscita da un programma

Risulta che ci sono molti modi per farlo e anche modi per impostare dei "ganci" in modo che una funzione venga eseguita quando un programma esce.

In questo capitolo approfondiremo e li esamineremo.

Abbiamo già trattato il significato del codice dello stato di uscita nella Exit Status sezione, quindi torna indietro e rivedi se necessario.

176Già, le cose si fanno bizzarre con le codifiche `multi-char16_t UTF-16`.
177https://en.wikipedia.org/wiki/Iconv
178http://site.icu-project.org/

Sono presenti tutte le funzioni di questa sezione in `<stdlib.h>`.

28.1. Uscite normali

Inizieremo con i modi normali per uscire da un programma per poi passare ad alcuni dei modi più rari ed esoterici.

Quando si esce normalmente da un programma tutti i flussi I/O aperti vengono cancellati e i file temporanei vengono rimossi. Fondamentalmente è una bella uscita in cui tutto viene ripulito e gestito. È quello che vuoi fare quasi sempre a meno che tu non abbia motivi per fare diversamente.

28.1.1. Di ritorno da `main()`

Se hai notato `main()` ha restituito un tipo `int`~… `eppure l'ho fatto raramente se mai ho ~return` qualcosa da `main()`.

Questo perché solo per `main()` (e non posso sottolineare abbastanza che questo caso speciale si applica *solo* a `main()` e nessun'altra funzione da nessuna parte) ha un *implicito* `return 0` se esci dalla fine.

Puoi esplicitamente `return` da `main()` ogni volta che vuoi e alcuni programmatori ritengono che sia più *giusto* per avere sempre un `return` alla fine `main()`. Ma se lo lasci disattivato C ne inserirà uno per te.

Quindi… ecco le regole di `return` per `main()`:

- È possibile restituire uno stato di uscita da `main()` con una dichiararzione `return`. `main()` è l'unica funzione con questo comportamento speciale. L'uso di `return` in qualsiasi altra funzione restituisce semplicemente il risultato da quella funzione al chiamante.
- Se non `return` esplicitamente e semplicemente esce fuori dal `main()` è come se avessi restituito `0` o `EXIT_SUCCESS`.

28.1.2. `exit()`

Anche questo è apparso alcune volte. Se chiami `exit()` da qualsiasi punto del tuo programma uscirà in quel punto.

L'argomento che passi a `exit()` è lo stato di uscita.

28.1.3. Impostazione dei gestori di uscita con `atexit()`

È possibile registrare le funzioni da chiamare all'uscita da un programma qualunque cosa sia ritornato da `main()` o chiamando la funzione `exit()`.

Una chiamata a `atexit()` con il nome della funzione del gestore lo farà. È possibile registrare più gestori di uscita e verranno chiamati nell'ordine inverso rispetto alla registrazione.

Ecco un esempio:

```
#include <stdio.h>
#include <stdlib.h>

void on_exit_1(void)
{
    printf("Exit handler 1 called!\n");
}
```

```
void on_exit_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(on_exit_1);
    atexit(on_exit_2);

    printf("About to exit...\n");
}
```

E l'output è:

```
About to exit...
Exit handler 2 called!
Exit handler 1 called!
```

28.2. Esce più velocemente con `quick_exit()`

Questo è simile ad un'uscita normale tranne:

- I file aperti potrebbero non essere cancellati.
- I file temporanei potrebbero non essere rimossi.
- `atexit()` i referenti non verranno chiamati.

Ma esiste un modo per registrare i gestori di uscita: chiama `at_quick_exit()` analogamente a come chiameresti `atexit()`.

```
#include <stdio.h>
#include <stdlib.h>

void on_quick_exit_1(void)
{
    printf("Quick exit handler 1 called!\n");
}

void on_quick_exit_2(void)
{
    printf("Quick exit handler 2 called!\n");
}

void on_exit(void)
{
    printf("Normal exit--I won't be called!\n");
}

int main(void)
{
    at_quick_exit(on_quick_exit_1);
    at_quick_exit(on_quick_exit_2);

    atexit(on_exit);  // Questo non verrà chiamato

    printf("About to quick exit...\n");

    quick_exit(0);
}
```

Il che dà questo output:

```
About to quick exit...
Quick exit handler 2 called!
Quick exit handler 1 called!
```

Funziona proprio come `exit()/atexit()` tranne per il fatto che lo svuotamento e la pulizia dei file potrebbero non essere eseguiti.

28.3. Nuke it from Orbit: `_Exit()`

La chiamata a `_Exit()` esce immediatamente punto. Non viene eseguita alcuna funzione di callback in uscita. I file non verranno cancellati. I file temporanei non verranno rimossi.

Usalo se devi uscire *proprio adesso*.

28.4. Uscire a volte: assert()

L'istruzione `assert()` viene utilizzata per insistere sul fatto che qualcosa sia vero altrimenti il programma uscirà.

Gli sviluppatori utilizzano spesso un'asserzione per rilevare errori di tipo "Non dovrebbe mai accadere".

```
#define PI 3.14159

assert(PI > 3);
// Sicuramente lo è, quindi vai avanti
```

contro:

```
goats -= 100;

assert(goats >= 0);
// Non si possono avere goats negative
```

In tal caso se provo a eseguirlo e `goats` scende sotto `0` succede questo:

```
goat_counter: goat_counter.c:8: main: Assertion `goats >= 0' failed.
Aborted
```

e sono tornato alla riga di comando.

Questo non è molto facile da usare quindi viene utilizzato solo per cose che l'utente non vedrà mai. E spesso le persone scrivono le proprie macro di asserzione che possono essere disattivate più facilmente.

28.5. Uscita anomala: `abort()`

Puoi usarlo se qualcosa è andato terribilmente storto e vuoi indicarlo all'ambiente esterno. Inoltre questo non pulirà necessariamente tutti i file aperti ecc.

Raramente l'ho visto usato.

Alcune anticipazioni sui *segnali*: questo in realtà funziona generando un `SIGABRT` che terminerà il processo.

Ciò che accade dopo dipende dal sistema, ma su sistemi Unix era comune eseguire il dump di core [180] quando il programma terminava.

179https://en.wikipedia.org/wiki/Core_dump
180https://en.wikipedia.org/wiki/Core_dump

29. Gestione del segnale

Prima di iniziare ti consiglierò semplicemente di ignorare in generale l'intero capitolo e di utilizzare le funzioni di gestione del segnale superiori (molto probabilmente) del tuo sistema operativo. I sistemi Unix hanno la famiglia di funzioni `sigaction()` e Windows ha… qualunque cosa faccia [182].

Detto questo cosa sono i segnali?

29.1. Cosa sono i segnali?

Un *segnale* viene *generato* su una varietà di eventi esterni. Il programma può essere configurato per essere interrotto per *gestire* il segnale ed eventualmente continuare da dove era stato interrotto una volta che il segnale è stato gestito.

Pensatela come una funzione che viene chiamata automaticamente quando si verifica uno di questi eventi esterni.

Quali sono questi eventi? Sul tuo sistema probabilmente ce ne sono molti ma nelle specifiche C ce ne sono solo alcuni:

Segnale	Descrizione
`SIGABRT`	Terminazione anomala—cosa succede quando viene chiamato `abort()`.
`SIGFPE`	Eccezione in virgola mobile.
~SIGILL~Istruzione illegale.	
`SIGINT`	Interrompere—di solito il risultato della pressione di `CTRL-C`.
`SIGSEGV`	“Violazione del segmento”: accesso alla memoria non valido.
`SIGTERM`	Richiesta di terminazione.

Puoi impostare il tuo programma in modo che ignori, gestisca o consenta l'azione predefinita per ciascuno di questi utilizzando la funzione `signal()`.

29.2. Gestire i segnali con `signal()`

La chiamata `signal()` accetta due parametri: il segnale in questione e un'azione da intraprendere quando tale segnale viene generato.

L'azione può essere una delle tre cose:

- Puntatore a una funzione del gestore.

181Apparentemente non esegue affatto segnali in stile Unix e sono simulati per le app della console.
182Apparentemente non esegue affatto segnali in stile Unix e sono simulati per le app della console.

- `SIG_IGN` per ignorare il segnale.
- `SIG_DFL` per ripristinare il gestore predefinito per il segnale.

Scriviamo un programma dal quale non è possibile uscire premendo `CTRL-C`. (Non preoccuparti—nel seguente programma puoi anche premere `return` e il programma uscirà.)

```
#include <stdio.h>
#include <signal.h>

int main(void)
{
    char s[1024];

    signal(SIGINT, SIG_IGN);
// Ignora SIGINT, causato da ^C

    printf("Try hitting ^C... (hit RETURN to exit)\n");

    // Attendi una riga di input
    // in modo che il programma
    // non esca semplicemente
    fgets(s, sizeof s, stdin);
}
```

Controlla la riga 8—diciamo al programma di ignorare `SIGINT` il segnale di interruzione che viene generato quando viene premuto `CTRL-C`. Non importa quanto lo premi il segnale rimane ignorato. Se commenti la riga 8 vedrai che puoi `CTRL-C` impunemente e uscire immediatamente dal programma.

29.3. Scrittura di gestori di segnali

Ho detto che potresti anche scrivere una funzione di gestione che venga chiamata quando il segnale viene generato.

Questi sono piuttosto semplici ma hanno anche capacità molto limitate per quanto riguarda le specifiche.

Prima di iniziare diamo un'occhiata al prototipo della funzione per la chiamata `signal()`:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Abbastanza facile da leggere vero?

SBAGLIATO! :)

Prendiamoci un momento per smontarlo e fare pratica.

`signal()` accetta due argomenti: un integer `sig` rche rappresenta il segnale e un puntatore `func` al gestore(il gestore restituisce `void` e accetta un `int` come argomento) evidenziato di seguito:

```
               sig          func
             |-----|  |---------------|
void (*signal(int sig, void (*func)(int)))(int);
```

Fondamentalmente passeremo il numero del segnale che ci interessa catturare e passeremo un puntatore a una funzione del modulo:

```
void f(int x);
```

questo farà la cattura.

Ora—che dire sul resto di quel prototipoe? Fondamentalmente è tutto il tipo ritornato. Vedi `signal()` restituirà tutto ciò che hai passato come `func` in caso di successo… quindi ciò significa che sta restituendo un puntatore a una funzione che restituisce `void` e accetta un `int` come argomento.

```
returned
function    indicates we're             and
returns     returning a                 that function
void        pointer to function         takes an int
|--|        |                                 |---|
void       (*signal(int sig, void (*func)(int)))(int);
```

Inoltre può restituire `SIG_ERR` in caso di errore.

Facciamo un esempio in cui lo facciamo in modo che tu debba premere `CTRL-C` due volte per uscire.

Voglio essere chiaro che questo programma si impegna in un comportamento indefinito in un paio di modi. Ma probabilmente funzionerà per te ed è difficile trovare demo portatili e non banali.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int count = 0;

void sigint_handler(int signum)
{
    // Il compilatore può eseguire:
    //
    //   signal(signum, SIG_DFL)
    //
    // quando viene chiamato il handler. Quindi reimpostiamo il handler qui:
    signal(SIGINT, sigint_handler);

    (void)signum;   // Elimina gli avvisi sulle variabili inutilizzate

    count++;                        // Comportamento indefinito
    printf("Count: %d\n", count);   // Comportamento indefinito

    if (count == 2) {
        printf("Exiting!\n");       // Comportamento indefinito
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Try hitting ^C...\n");

    for(;;);  // Wait here forever
}
```

Una delle cose che noterai è che alla riga 14 resettiamo il gestore del segnale. Questo perché C ha la possibilità di reimpostare il gestore del segnale sul suo comportamento `SIG_DFL` prima di eseguire il gestore personalizzato. In altre parole potrebbe trattarsi di un caso isolato. Quindi lo resettiamo per prima cosa in modo da poterlo gestire di nuovo per quello successivo.

In questo caso stiamo ignorando il valore restituito da `signal()`. Se lo impostassimo prima su un gestore diverso restituirebbe un puntatore a quel gestore che potremmo ottenere in questo modo:

```
// old_handler è di tipo "puntatore a una funzione che accetta un singolo
// int parametro e restituisparametro e restituiscece void":

void (*old_handler)(int);

old_handler = signal(SIGINT, sigint_handler);
```

Detto questo non sono sicuro di un caso d'uso comune per questo. Ma se per qualche motivo hai bisogno del vecchio gestore puoi ottenerlo in questo modo.

Breve nota alla riga 16—questo serve solo per dire al compilatore di non avvisare che non stiamo utilizzando questa variabile. È come dire: “So che non lo sto usando; non devi avvisarmi.”

E infine vedrai che ho contrassegnato un comportamento indefinito in un paio di punti. Ne parleremo più avanti nella prossima sezione.

29.4. Cosa possiamo realmente fare?

Risulta che siamo piuttosto limitati in ciò che possiamo e non possiamo fare nei nostri gestori di segnale. Questo è uno dei motivi per cui dico che non dovresti nemmeno preoccuparti di questo e utilizzare invece la gestione del segnale del sistema operativo (ad esempio `sigaction()` per sistemi a Unix-like).

Wikipedia arriva al punto di dire che l'unica cosa veramente portatile che puoi fare è chiamare `signal()` con `SIG_IGN` o `SIG_DFL` e il gioco è fatto.

Ecco cosa **non possiamo fare** in modo portabile:

- Chiama qualsiasi funzione della libreria standard.
- Come `printf()`, per esempio.
- Penso che probabilmente sia sicuro chiamare funzioni riavviabili/rientranti ma le specifiche non consentono questa libertà.
- Ottieni o imposta valori da una variabile `static` locale nell'ambito del file o da una variabile locale del thread.
- A meno che non sia un oggetto atomico sblocato o…
- Stai assegnando una variabile di tipo `volatile sig_atomic_t`.

L'ultima parte–~sig$_{\text{atomict}}$~–è il tuo biglietto per ottenere dati da un gestore di segnale. (A meno che non si desideri utilizzare oggetti atomici senza blocco che esulano dall'ambito di questa sezione[183] .) È un tipo intero che potrebbe o meno essere firmato. Ed è limitato da ciò che puoi inserire lì dentro.

Puoi controllare i valori minimi e massimi consentiti nelle macro `SIG_ATOMIC_MIN` e `SIG_ATOMIC_MAX`[184] .

In modo confuso le specifiche dicono anche che non è possibile fare riferimento "a qualsiasi oggetto con durata di archiviazione statica o thread che non sia un oggetto atomico privo di blocchi se non assegnando un valore a un oggetto dichiarato come `volatile sig_atomic_t` […]”

La mia interpretazione al riguardo è che non puoi leggere o scrivere nulla che non sia un oggetto

183In modo confuso `sig_atomic_t` è antecedente agli atomi senza blocchi e non è la stessa cosa.

184Se `sig_action_t` è signed l'intervallo sarà compreso almeno tra -127 e 127. Se fosse unsigned sarebbe compreso tra 0 e 255.

atomico privo di blocchi. Inoltre puoi assegnare a un oggetto che è `volatile sig_atomic_t`.

Ma puoi leggerlo da lì? Onestamente non vedo perché no se non per il fatto che le specifiche sono molto precise nel menzionare l'assegnazione interna. Ma se devi leggerlo e prendere qualsiasi tipo di decisione basata su di esso potresti aprire una stanza per qualche tipo di condizioni di corsa.

Con questo in mente possiamo riscrivere il nostro "premi `CTRL-C` due volte per uscire" in modo che il codice sia un po' più portabile anche se meno dettagliato nell'output.

Modifichiamo il nostro gestore `SIGINT` in modo che non faccia altro che incrementare un valore di tipo `volatile sig_atomic_t`. Quindi conterà il numero di `CTRL-C` che sono stati premuti.

Quindi nel nostro ciclo principale controlleremo se il contatore è superiore a `2` ed eseguiremo il salvataggio se lo è.

```
#include <stdio.h>
#include <signal.h>

volatile sig_atomic_t count = 0;

void sigint_handler(int signum)
{
    (void)signum;
// Avviso per variabile non utilizzata

    signal(SIGINT, sigint_handler);
// Ripristina il gestore del segnale

    count++;
// Comportamento indefinito
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(count < 2);
}
```

Di nuovo un comportamento indefinito? La mia lettura è questa perché dobbiamo leggere il valore per incrementarlo e memorizzarlo.

Se vogliamo posticipare l'uscita solo premendo `CTRL-C` possiamo farlo senza troppi problemi. Ma qualsiasi ulteriore rinvio richiederebbe un ridicolo concatenamento di funzioni.

Ciò che faremo è gestirlo una volta e il gestore ripristinerà il segnale sul suo comportamento predefinito (cioè uscire):

```
#include <stdio.h>
#include <signal.h>

void sigint_handler(int signum)
{
    (void)signum;
// Avviso per variabile non utilizzata
    signal(SIGINT, SIG_DFL);
// Ripristina il gestore del segnale
}
```

```
int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

Più avanti quando esamineremo le variabili atomiche prive di blocco vedremo un modo per correggere la versione di conteggio (presupponendo che le variabili atomiche prive di blocco siano disponibili sul tuo particolare sistema).

Questo è il motivo per cui all'inizio suggerivo di verificare il sistema di segnali integrato nel tuo sistema operativo come alternativa probabilmente migliore.

29.5. Gli amici non lasciano che gli amici `signal()`

Ancora una volta utilizza la gestione del segnale integrata nel tuo sistema operativo o un equivalente. Non è nelle specifiche non è così portatile ma probabilmente ha più funzionalità. Inoltre il tuo sistema operativo probabilmente ha un numero di segnali definiti che non sono nelle specifiche C. Ed è comunque difficile scrivere codice portabile utilizzando `signal()`.

30. Array a lunghezza variabile (VLA)

C fornisce un modo per dichiarare un array la cui dimensione è determinata in fase di esecuzione. Questo ti offre i vantaggi del dimensionamento dinamico del runtime come quello che ottieni con `malloc()`, ma senza doversi preoccupare di usare `free()` sulla memoria in seguito.

Ora a molte persone non piacciono i VLA. Ad esempio sono stati banditi dal kernel Linux. Approfondiremo meglio questa logica più avanti.

Questa è una caratteristica opzionale della lingua. La macro `__STDC_NO_VLA__` è impostata su `1` se i VLA *non* sono presenti. (Erano obbligatori nel C99 poi divennero facoltativi nel C11.)

```
#if __STDC_NO_VLA__ == 1
   #error Sorry, need VLAs for this program!
#endif
```

Ma poiché né GCC né Clang si preoccupano di definire questa macro, potresti ottenere un conteggio limitato da questo.

Immergiamoci prima con un esempio e poi cercheremo il diavolo nei dettagli.

30.1. Le basi

Un array normale viene dichiarato con una dimensione costante come questo:

```
int v[10];
```

Ma con i VLA, possiamo utilizzare una dimensione determinata in fase di esecuzione per impostare l'array in questo modo:

```
int n = 10;
int v[n];
```

Ora sembra la stessa cosa e per molti versi lo è ma questo ti dà la flessibilità di calcolare la dimensione che ti serve e quindi ottenere un array di esattamente quella dimensione.

Chiediamo all'utente di inserire la dimensione dell'array e quindi di memorizzare l'indice-10-volte in ciascuno di questi elementi dell'array:

```
#include <stdio.h>

int main(void)
{
    int n;
    char buf[32];

    printf("Enter a number: "); fflush(stdout);
    fgets(buf, sizeof buf, stdin);
    n = strtoul(buf, NULL, 10);

    int v[n];

    for (int i = 0; i < n; i++)
        v[i] = i * 10;

    for (int i = 0; i < n; i++)
        printf("v[%d] = %d\n", i, v[i]);
}
```

(Alla riga 7 ho un `fflush()` che dovrebbe forzare l'output della riga anche se non ho un ritorno a capo alla fine.)

La riga 10 è dove dichiariamo il VLA—una volta che l'esecuzione supera quella riga la dimensione dell'array viene impostata su qualunque fosse `n` in quel momento. La lunghezza dell'array non può essere modificata in seguito.

Puoi anche inserire un'espressione tra parentesi:

```
int v[x * 100];
```

Alcune limitazioni:

- Non è possibile dichiarare un VLA nell'ambito del file e non è possibile crearne uno `static` nell'ambito del blocco[185] .
- Non è possibile utilizzare un elenco di inizializzatori per inizializzare l'array.

Inoltre l'immissione di un valore negativo per la dimensione dell'array richiama un comportamento indefinito—in questo universo comunque.

30.2. `sizeof` e VLA

Siamo abituati a `sizeof` che ci fornisce la dimensione in byte di qualsiasi oggetto particolare inclusi gli array. E i VLA non fanno eccezione.

La differenza principale è che `sizeof` su un VLA viene eseguito in fase di *runtime*, mentre su una variabile di dimensioni non variabili viene calcolata in *fase di compilazione*.

Ma l'utilizzo è lo stesso.

Puoi anche calcolare il numero di elementi in un VLA con il solito trucco degli array:

```
size_t num_elems = sizeof v / sizeof v[0];
```

185Ciò è dovuto al modo in cui i VLA vengono generalmente allocati nello stack mentre le variabili statiche si trovano nell'heap. E l'idea generale dei VLA è che verranno allocati automaticamente quando lo stack frame viene visualizzato alla fine della funzione.

C'è un'implicazione sottile e corretta dalla riga sopra: l'aritmetica dei puntatori funziona proprio come ci si aspetterebbe da un array normale. Quindi vai avanti e usalo a tuo piacimento:

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    int v[n];

    int *p = v;

    *(p+2) = 12;
    printf("%d\n", v[2]);  // 12

    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

Come con gli array regolari, puoi usare le parentesi con `sizeof()` per ottenere la dimensione di un potenziale VLA senza dichiararne effettivamente uno:

```
int x = 12;

printf("%zu\n", sizeof(int [x]));
// Stampa 48 sul mio sistema
```

30.3. VLA multidimensionali

Puoi andare avanti e creare tutti i tipi di VLA con una o più dimensioni da impostate su una variabile

```
int w = 10;
int h = 20;

int x[h][w];
int y[5][w];
int z[10][w][20];
```

Ancora una volta puoi spostarti tra questi come faresti con un normale array.

30.4. Passaggio di VLA unidimensionali alle funzioni

Il passaggio di VLA unidimensionali in una funzione non può essere diverso dal passaggio di un array regolare. Provaci e basta.

```
#include <stdio.h>

int sum(int count, int *v)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += v[i];

    return total;
}

int main(void)
{
```

```
    int x[5];   // Standard array

    int a = 5;
    int y[a];   // VLA

    for (int i = 0; i < a; i++)
        x[i] = y[i] = i + 1;

    printf("%d\n", sum(5, x));
    printf("%d\n", sum(a, y));
}
```

Ma c'è qualcosa di più oltre a questo. Puoi anche far sapere a C che l'array ha una dimensione VLA specifica passandolo prima e quindi fornendo quella dimensione nell'elenco dei parametri:

```
int sum(int count, int v[count])
{
    // ...
}
```

Per inciso ci sono un paio di modi per elencare un prototipo per la funzione di cui sopra; uno di questi implica un `*` se non vuoi nominare specificamente il valore nel VLA. Indica semplicemente che il tipo è un VLA anziché un puntatore normale.

Prototipi VLA:

```
void do_something(int count, int v[count]);
// Con nomi
void do_something(int, int v[*]);
// Senza nomi
```

Ancora una volta quella `*` funziona solo con il prototipo—nella funzione stessa dovrai inserire la dimensione esplicita.

Ora—/diventiamo multidimensionali/! È qui che inizia il divertimento.

30.5. Passaggio di VLA multidimensionali alle funzioni

Stessa cosa che abbiamo fatto con la seconda forma di VLA unidimensionali sopra, ma questa volta stiamo passando in due dimensioni e utilizzando quelle.

Nell'esempio seguente costruiamo una matrice di tabella di moltiplicazione di larghezza e altezza variabili quindi la passiamo a una funzione per stamparla.

```
#include <stdio.h>

void print_matrix(int h, int w, int m[h][w])
{
    for (int row = 0; row < h; row++) {
        for (int col = 0; col < w; col++)
            printf("%2d ", m[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int rows = 4;
    int cols = 7;
```

```
    int matrix[rows][cols];

    for (int row = 0; row < rows; row++)
        for (int col = 0; col < cols; col++)
            matrix[row][col] = row * col;

    print_matrix(rows, cols, matrix);
}
```

30.5.1. VLA multidimensionali parziali

È possibile avere alcune dimensioni fisse e altre variabili.Diciamo che abbiamo una lunghezza record fissata a 5 elementi ma non sappiamo quanti record ci siano.

```
#include <stdio.h>

void print_records(int count, int record[count][5])
{
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < 5; j++)
            printf("%2d ", record[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int rec_count = 3;
    int records[rec_count][5];

    // Fill with some dummy data
    for (int i = 0; i < rec_count; i++)
        for (int j = 0; j < 5; j++)
            records[i][j] = (i+1)*(j+2);

    print_records(rec_count, records);
}
```

30.6. Compatibilità con array regolari

Poiché i VLA sono proprio come i normali array in memoria è perfettamente consentito passarli in modo intercambiabile… finché le dimensioni corrispondano.

Ad esempio se abbiamo una funzione che richiede specificamente un array 3×5 possiamo ancora passarvi un VLA.

```
int foo(int m[5][3]) {...}

\\ ...

int w = 3, h = 5;
int matrix[h][w];

foo(matrix);   // OK!
```

Allo stesso modo se hai una funzione VLA puoi passarle un array regolare:

```
int foo(int h, int w, int m[h][w]) {...}

\\ ...
```

```
int matrix[3][5];

foo(3, 5, matrix);   // OK!
```

Attenzione però: se le tue dimensioni non corrispondono probabilmente avrai un comportamento indefinito.

30.7. `typedef` e VLA

Puoi `typedef` un VLA ma il comportamento potrebbe non essere quello previsto.

Fondamentalmente `typedef` crea un nuovo tipo con i valori esistenti nel momento in cui è stata eseguita il `typedef`.

Quindi non è un `typedef` di un VLA tanto quanto un nuovo tipo di array di dimensioni fisse in quel momento.

```
#include <stdio.h>

int main(void)
{
    int w = 10;

    typedef int goat[w];

    // goat is an array of 10 ints
    goat x;

    // Init with squares of numbers
    for (int i = 0; i < w; i++)
        x[i] = i*i;

    // Print them
    for (int i = 0; i < w; i++)
        printf("%d\n", x[i]);

    // Now let's change w...

    w = 20;

    // Ma goat è ANCORA un array di 10 int, perché quello era il
    // valore di w quando viene eseguito il typedef.
}
```

Quindi si comporta come un array di dimensione fissa.

Ma non puoi ancora utilizzare un elenco di inizializzatori su di esso.

30.8. Saltare le trappole

Devi fare attenzione quando usi `goto` vicino ai VLA perché molte cose non sono permesse.

E quando usi `longjmp()` c'è un caso in cui potresti perdere memoria con VLA.

Ma di entrambe queste cose tratteremo nei rispettivi capitoli.

30.9. General Issues

I VLA sono stati banditi dal kernel Linux per alcuni motivi:

- Molti dei posti in cui venivano utilizzati avrebbero dovuto essere di dimensioni fisse.
- Il codice dietro i VLA è più lento (a un livello che la maggior parte delle persone non noterebbe ma che fa la differenza in un sistema operativo).
- I VLA non sono supportati allo stesso modo da tutti i compilatori C.
- La dimensione dello stack è limitata e i VLA vanno nello stack. Se qualche codice passa accidentalmente (o in modo dannoso) un valore elevato in una funzione del kernel che alloca un VLA potrebbero succedere *cose brutte*™.

Altre persone online sottolineano che non c'è modo di rilevare la mancata allocazione di un VLA e che i programmi che hanno subito tali problemi probabilmente si bloccherebbero semplicemente. Sebbene anche gli array a dimensione fissa abbiano lo stesso problema è molto più probabile che qualcuno crei accidentalmente un *VLA di dimensioni insolite* piuttosto che dichiarare in qualche modo accidentalmente un array a dimensione fissa ad esempio da 30 megabyte.

31. `goto`

L'istruzione `goto` è universalmente rispettata e può essere qui presentato senza contestazioni.

Stavo solo scherzando! Nel corso degli anni ci sono stati molti avanti e indietro sulla questione se o meno (spesso no) `goto` è considerato dannoso[186] .

Secondo l'opinione di questo programmatore dovresti utilizzare qualunque costrutto porti al *miglior* codice, tenendo conto della manutenibilità e della velocità. E a volte questo potrebbe essere `goto`!

In questo capitolo vedremo come funziona `goto` in C, quindi esamineremo alcuni dei casi più comuni in cui viene utilizzato[187] .

31.1. Un semplice esempio

In questo esempio utilizzeremo `goto` per saltare una riga di codice e passare a un'*etichetta*. L'etichetta è l'identificatore che può essere un bersaglio di ~goto~—termina con i due punti (:).

```
#include <stdio.h>

int main(void)
{
    printf("One\n");
    printf("Two\n");

    goto skip_3;

    printf("Three\n");

skip_3:

    printf("Five!\n");
}
```

L’output è:

```
One
Two
Five!
```

186 https://en.wikipedia.org/wiki/Goto#Criticism↩

187 Vorrei sottolineare che l'uso di `goto` in tutti questi casi è evitabile. Puoi invece utilizzare variabili e loop. È solo che alcune persone pensano che goto produca il codice migliore in quelle circostanze.

`goto` manda l'esecuzione saltando all'etichetta specificata saltando tutto il resto.

Puoi saltare avanti o indietro con `goto`.

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

Le etichette vengono saltate durante l'esecuzione. Quanto segue stamperà tutti e tre i numeri in ordine proprio come se le etichette non fossero presenti:

```
    printf("Zero\n");
label_1:
label_2:
    printf("One\n");
label_3:
    printf("Two\n");
label_4:
    printf("Three\n");
```

Come hai notato è una convenzione comune giustificare le etichette tutte a sinistra. Ciò aumenta la leggibilità perché un lettore può rapidamente trovare la destinazione.

Le etichette hanno un *ambito di funzione*. Cioè non importa quanti livelli di blocchi profondi appaiano puoi comunque accedere a `goto` da qualsiasi punto della funzione.

Significa anche che puoi solo andare con `goto` alle etichette che hanno la stessa funzione del `goto` stesso. Le etichette in altre funzioni non rientrano nell'ambito del punto di vista di `goto`. E significa che puoi usare lo stesso nome di etichetta in due funzioni—semplicemente non lo stesso nome di etichetta nella stessa funzione.

31.2. `continue` Etichettato

In alcuni linguaggi è possibile effettivamente specificare un'etichetta per un'istruzione `continue`. C non lo permette ma puoi facilmente usare invece `goto`.

Per mostrare il problema controlla `continue` in questo ciclo nidificato:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        continue;
    // Va sempre al successivo j
    }
}
```

Come vediamo che questo `continue` come tutti i `continue` va alla successiva iterazione del ciclo di inclusione più vicino. E se volessimo `continue` con il ciclo successivo, il ciclo con `i`?

Bene, possiamo `break` per tornare al ciclo esterno giusto?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;
    // Gets us to the next iteration of i
    }
}
```

Questo ci dà due livelli di ciclo annidato. Ma se annidiamo un altro ciclo non abbiamo più opzioni. Che dire di questo in cui non abbiamo alcuna istruzione che ci porti alla successiva iterazione di `i`?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
            printf("%d, %d, %d\n", i, j, k);

            continue;  // Ci porta alla successiva iterazione di k
            break;     // Ci porta alla successiva iterazione di j
            ????;      // Ci porta alla successiva iterazione di i???

        }
    }
}
```

L'istruzione `goto` ci offre un modo!

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                printf("%d, %d, %d\n", i, j, k);

                goto continue_i;   // Ora continuando il ciclo i!!
            }
        }
continue_i: ;
    }
```

Noi abbiamo un `;` alla fine lì—questo perché non è possibile avere un'etichetta che punti alla fine di un'istruzione composta (o prima di una dichiarazione di variabile).

31.3. Bailing Out

Quando sei super annidato nel mezzo di un codice puoi usare `goto` per uscirne in un modo che spesso è più pulito rispetto all'annidamento di più `if` e all'utilizzo di variabili flag.

```
    // Pseudocode

    for(...) {
        for (...) {
            while (...) {
                do {
                    if (some_error_condition)
                        goto bail;

                } while(...);
            }
        }
    }

bail:
    // Cleanup here
```

Senza `goto` dovresti controllare un flag di condizione di errore in tutti i loop per uscire completamente.

31.4. `break` Etichettato

Questa è una situazione molto simile a come `continue` continua solo il ciclo più interno. `break` inoltre esce solo dal ciclo più interno.

```
for (int i = 0; i < 3; i++) {
```

```
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;   // Esce solo dal j loop
    }
}

printf("Done!\n");
```

Ma possiamo usare `goto` per andare più lontano:

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            goto break_i;   // Ora esco dal loop i!
        }
    }

break_i:

    printf("Done!\n");
```

31.5. Pulizia multilivello

Se stai chiamando più funzioni per inizializzare più sistemi e uno di essi fallisce, dovresti deinizializzare solo quelli a cui sei arrivato finora.

Facciamo un esempio falso in cui iniziamo a inizializzare i sistemi e a controllare se qualcuno restituisce un errore (useremo `-1` per indicare un errore). Se uno di loro lo fa dobbiamo spegnere solo i sistemi che abbiamo inizializzato finora.

```
    if (init_system_1() == -1)
        goto shutdown;

    if (init_system_2() == -1)
        goto shutdown_1;

    if (init_system_3() == -1)
        goto shutdown_2;

    if (init_system_4() == -1)
        goto shutdown_3;

    do_main_thing();   // Esegui il nostro programma

    shutdown_system4();

shutdown_3:
    shutdown_system3();

shutdown_2:
    shutdown_system2();

shutdown_1:
    shutdown_system1();

shutdown:
    print("All subsystems shut down.\n");
```

Tieni presente che stiamo spegnendo nell'ordine inverso rispetto a quando abbiamo inizializzato i sottosistemi. Pertanto se il sottosistema 4 non riesce ad avviarsi verranno arrestati 3, 2, e poi 1 in

quest'ordine.

31.6. Tail Call Optimization

Tipo. Solo per funzioni ricorsive.

Se non hai familiarità Tail Call Optimization (TCO) [188] è un modo per non sprecare spazio nello stack quando chiami altre funzioni in circostanze molto specifiche. Sfortunatamente i dettagli vanno oltre lo scopo di questa guida.

Ma se disponi di una funzione ricorsiva che sai può essere ottimizzata in questo modo, puoi utilizzare questa tecnica. (Tieni presente che non puoi chiamare in coda altre funzioni a causa dell'ambito della funzione delle etichette.)

Facciamo un esempio semplice: un fattoriale.

Ecco una versione ricorsiva che non è TCO ma può esserlo!

```
#include <stdio.h>
#include <complex.h>

int factorial(int n, int a)
{
    if (n == 0)
        return a;

    return factorial(n - 1, a * n);
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %ld\n", i, factorial(i, 1));
}
```

Per realizzarlo puoi sostituire la chiamata con due passaggi:

1. Imposta i valori dei parametri su quelli che sarebbero nella chiamata successiva.
2. `goto` a un'etichetta sulla prima riga della funzione.

Proviamolo:

```
#include <stdio.h>

int factorial(int n, int a)
{
tco:  // Aggiungi questo

    if (n == 0)
        return a;

    // sostituire return impostando nuovi valori dei parametri e
    // goto-ing l'inizio della funzione

    //return factorial(n - 1, a * n);

    int next_n = n - 1;  // Vedi come si abbinano
    int next_a = a * n;  // gli argomenti ricorsivi sopra?
```

188https://en.wikipedia.org/wiki/Tail_call

```
    n = next_n;   // Set the parameters to the new values
    a = next_a;

    goto tco;   // And repeat!
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %d\n", i, factorial(i, 1));
}
```

Ho usato variabili temporanee lassù per impostare i valori successivi dei parametri prima di saltare all'inizio della funzione. Guarda come corrispondono agli argomenti ricorsivi presenti nella chiamata ricorsiva?

Ora perché usare le variabili temporanee? Invece avrei potuto fare :

```
a *= n;
n -= 1;

goto tco;
```

e in realtà funziona perfettamente. Ma se invertisco con noncuranza quelle due righe di codice:

```
n -= 1;  // CATTIVE NOTIZIE
a *= n;
```

—ora siamo nei guai. Abbiamo modificato `n` prima di usarlo per modificare `a`. Questo è un male perché non è così che funziona quando chiami in modo ricorsivo. L'uso delle variabili temporanee evita questo problema anche se non te ne preoccupi. E il compilatore probabilmente li ottimizza comunque.

31.7. Riavvio delle chiamate di sistema interrotte

Questo è fuori dalle specifiche ma comunemente visto nei sistemi Unix-like.

Alcune chiamate di sistema di lunga durata potrebbero restituire un errore se vengono interrotte da un segnale e `errno` sarà impostato su `EINTR` per indicare che la chiamata di sistema in esecuzione; è stato semplicemente interrotto.

In questi casi è molto comune che il programmatore voglia riavviare la chiamata e riprovare.

```
retry:
    byte_count = read(0, buf, sizeof(buf) - 1);  // Unix read() syscall

    if (byte_count == -1) {            // Si è verificato un errore...
        if (errno == EINTR) {          // Ma è stato semplicemente interrotto
            printf("Restarting...\n");
            goto retry;
        }
```

Molti Unix-likes hanno una flag `SA_RESTART` a cui puoi passare `sigaction()` per richiedere al sistema operativo di riavviare automaticamente eventuali chiamate di sistema lente invece di fallire con `EINTR`.

Ancora una volta, questo è specifico di Unix ed è al di fuori dello standard C.

Detto questo è possibile utilizzare una tecnica simile ogni volta che si desidera riavviare una funzione.

31.8. `goto` and Thread Preemption

Questo esempio è stato estratto direttamente da *Operating Systems: Three Easy Pieces* un altro libro eccellente di autori che la pensano allo stesso modo e che ritengono che i libri di qualità dovrebbero essere scaricabili gratuitamente. Non che io sia supponente o altro.

```
retry:

    pthread_mutex_lock(L1);

    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto retry;
    }

    save_the_day();

    pthread_mutex_unlock(L2);
    pthread_mutex_unlock(L1);
```

Lì il thread acquisisce felicemente il mutex (mutua escusione `L1` ma poi potenzialmente non riesce a ottenere la seconda risorsa protetta da mutex `L2` (se qualche altro thread non cooperativo lo tiene, diciamo). Se il nostro thread non riesce a ottenere il blocco `L2` sblocca `L1` e quindi utilizza `goto` per riprovare in modo pulito.

Ci auguriamo che il nostro thread eroico alla fine riesca ad acquisire entrambi i mutex e a salvare la situazione il tutto evitando il malvagio stallo.

31.9. `goto` e ambito variabile

Abbiamo già visto che le etichette hanno un ambito di funzione ma possono accadere cose strane se saltiamo oltre l'inizializzazione di alcune variabili.

Guarda questo esempio in cui saltiamo da un punto in cui la variabile `x` è fuori ambito al centro del suo ambito (nel blocco).

```
    goto label;

    {
        int x = 12345;

label:
        printf("%d\n", x);
    }
```

Questo verrà compilato ed eseguito ma mi dà un avviso:

```
warning: ‘x’ is used uninitialized in this function
```

E poi stampa `0` quando lo eseguo (il tuo conteggio può variare).

Fondamentalmente quello che è successo è che siamo entrati nell'ambito di `x` (quindi era giusto farne riferimento nel `printf()`) ma abbiamo saltato la riga che effettivamente lo inizializzava a `12345`. Quindi il valore era indeterminato.

La soluzione è, ovviamente, ottenere l'inizializzazione *dopo* l'etichetta in un modo o nell'altro.

```
    goto label;
```

```
    {
        int x;

label:
        x = 12345;
        printf("%d\n", x);
    }
```

Diamo un'occhiata a un altro esempio.

```
    {
        int x = 10;

label:

        printf("%d\n", x);
    }

    goto label;
```

Che succede qui?

La prima volta che attraversiamo il blocco, siamo a posto. `x` è `10` e questo è ciò che viene stampato.

Ma dopo il `goto`, entriamo nell'ambito di x, ma dopo la sua inizializzazione. Ciò significa che possiamo ancora stamparlo, ma il valore è indeterminato (poiché non è stato reinizializzato).

Sulla mia macchina stampa di nuovo `10` (all'infinito) ma è solo fortuna. Potrebbe stampare qualsiasi valore dopo il `goto` poiché `x` non è inizializzato.

31.10. `goto` e array a lunghezza variabile

Quando si tratta di VLA e `goto` c'è una regola: non è possibile passare dall'esterno dell'ambito di un VLA all'ambito di tale VLA. Se provo a farlo:

```
    int x = 10;

    goto label;

    {
        int v[x];

label:

        printf("Hi!\n");
    }
```

Ottengo un errore:

```
error: jump into scope of identifier with variably modified type
```

Puoi anticipare la dichiarazione VLA in questo modo:

```
    int x = 10;

    goto label;

    {
label:  ;
        int v[x];
```

```
        printf("Hi!\n");
    }
```

Perché in questo modo il VLA viene allocato correttamente prima della sua inevitabile deallocazione una volta uscito dall'ambito di applicazione.

32. Tipi Parte V: Letterali composti e selezioni generiche

Questo è il capitolo finale per i tipi! Parleremo di due cose:

- Come avere oggetti "anonimi" senza nome e come è utile.
- Come generare codice dipendente dal tipo.

Non sono particolarmente correlati, ma in realtà non sono abbastanza per i propri capitoli. Quindi li ho stipati qui come un ribelle!

32.1. Compound Literals

Questa è una caratteristica interessante del linguaggio che ti consente di creare al volo un oggetto di qualche tipo senza mai assegnarlo a una variabile. Puoi creare tipi semplici, array e `struct` dandogli un nome

Uno degli usi principali è passare argomenti complessi alle funzioni quando non vuoi creare una variabile temporanea per contenere il valore.

Il modo per creare un valore letterale composto è mettere il nome del tipo tra parentesi e poi inserire un elenco di inizializzatori. Ad esempio un array di numeri `int` senza nome potrebbe assomigliare a questo:

```
(int []){1,2,3,4}
```

Ora quella riga di codice non fa nulla da sola. Crea un array senza nome di 4 `int` e poi li butta via senza usarli. Potremmo usare un puntatore per memorizzare un riferimento all'array…

```
int *p = (int []){1 ,2 ,3 ,4};

printf("%d\n", p[1]);  // 2
```

Ma sembra un po' un modo prolisso per avere un array. Voglio dire, avremmo potuto farlo e basta [190] :

```
int p[] = {1, 2, 3, 4};

printf("%d\n", p[1]);  // 2
```

Quindi diamo un'occhiata a un esempio più utile.

32.1.1. Passaggio di oggetti senza nome alle funzioni

Diciamo che abbiamo una funzione per sommare un array di numeri `int`:

```
int sum(int p[], int count)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += p[i];
```

189Il che non è esattamente la stessa cosa, dato che è un array non un puntatore a un `int`.
190Il che non è esattamente la stessa cosa, dato che è un array non un puntatore a un `int`.

```
    return total;
}
```

Se volessimo chiamarlo normalmente dovremmo fare qualcosa del genere dichiarando un array e memorizzandovi valori da passare alla funzione:

```
int a[] = {1, 2, 3, 4};

int s = sum(a, 4);
```

Ma gli oggetti senza nome ci danno un modo per saltare la variabile passandola direttamente (nomi dei parametri elencati sopra). Controlla—adesso sostituiremo la variabile a con un array senza nome che passeremo come primo argomento:

```
//                  p[]         count
//           |-----------------|  |
int s = sum((int []){1, 2, 3, 4}, 4);
```

Abbastanza lucido!

32.1.2. `struct` Senza nome

Possiamo fare qualcosa di simile con `struct`.

Per prima cosa facciamo le cose escudendo oggetti senza nome.

Definiremo una `struct` per contenere alcune coordinate `x/y`. E perciò ne definiremo uno passando i valori al suo inizializzatore. Infine lo passeremo a una funzione per stampare i valori:

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord c)
{
    printf("%d, %d\n", c.x, c.y);
}

int main(void)
{
    struct coord t = {.x=10, .y=20};

    print_coord(t);   // stampa "10, 20"
}
```

Abbastanza semplice?

Modifichiamolo per utilizzare un oggetto senza nome invece della variabile `t` a cui stiamo passando `print_coord()`.

Toglieremo semplicemente `t` da lì e lo sostituiremo con un `struct` senza nome:

```
//struct coord t = {.x=10, .y=20};

print_coord((struct coord){.x=10, .y=20});   // stampa "10, 20"
```

Funziona ancora!

32.1.3. Puntatori a oggetti senza nome

Potresti aver notato nell'ultimo esempio che anche in questo caso stavamo usando una `struct`, stavamo passando una copia della `struct` a `print_coord()` invece di passare un puntatore a `struct`.

Nota che possiamo semplicemente prendere l'indirizzo di un oggetto senza nome con & come sempre.

Questo perché in generale, se un operatore avesse lavorato su una variabile di quel tipo puoi usare quell'operatore su un oggetto senza nome di quel tipo.

Modifichiamo il codice precedente in modo da passare un puntatore a un oggetto senza nome

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord *c)
{
    printf("%d, %d\n", c->x, c->y);
}

int main(void)
{
    //     Note the &
    //           |
    print_coord(&(struct coord){.x=10, .y=20});   // stampa "10, 20"
}
```

Inoltre questo può essere un buon modo per passare anche i puntatori a oggetti semplici:

```
// Pass a pointer to an int with value 3490
foo(&(int){3490});
```

Facile così.

1. Oggetti senza nome e ambito

La durata di un oggetto senza nome termina alla fine del suo ambito. Il più grande modo in cui potrebbe farti male è nel caso crei un nuovo oggetto senza nome, ottenere un puntatore ad esso e quindi lasciare l'ambito dell'oggetto. In tal caso, il puntatore si riferirà a un oggetto morto.

Quindi questo è un comportamento indefinito:

```
int *p;

{
    p = &(int){10};
}

printf("%d\n", *p);  // NON VALIDO: Il (int){10} è fuori portata
```

Allo stesso modo non è possibile restituire un puntatore a un oggetto senza nome da una funzione. L'oggetto viene deallocato quando esce dall'ambito:

```
#include <stdio.h>
```

```
int *get3490(void)
{
    // Don't do this
    return &(int){3490};
}

int main(void)
{
    printf("%d\n", *get3490());  // NON VALIDO: (int){3490} è fuori portata
}
```

Basti pensare al loro ambito come a quello di una normale variabile locale. Non è nemmeno possibile restituire un puntatore a una variabile locale.

32.1.4. Stupido esempio di oggetto senza nome

Puoi inserire qualsiasi tipo e creare un oggetto senza nome.

Ad esempio questi sono effettivamente equivalenti:

```
int x = 3490;

printf("%d\n", x);
// 3490 (variabile)
printf("%d\n", 3490);
// 3490 (costante)
printf("%d\n", (int){3490});
// 3490 (oggetto senza nome)
```

Quest'ultimo non ha nome, ma è stupido. Tanto vale fare quello semplice nella linea precedente.

Ma si spera che questo fornisca un po’ più di chiarezza sulla sintassi.

32.2. Generic Selections

Questa è un'espressione che ti consente di selezionare diverse parti di codice a seconda del *tipo* del primo argomento dell'espressione.

Vedremo un esempio tra un secondo ma è importante sapere che questo viene elaborato in fase di compilazione, *non in fase di runtime*. Non è in corso alcuna analisi di runtime qui.

L'espressione inizia con `_Generic`, funziona un po' come un `switch` e richiede almeno due argomenti.

Il primo argomento è un'espressione (o variabile[191]) che ha un *tipo*. Tutte le espressioni hanno un tipo. I rimanenti argomenti di `_Generic` rappresentano i casi in cui sostituire il risultato dell'espressione se il primo argomento è di quel tipo.

Che cosa?

Proviamolo e vediamo.

```
#include <stdio.h>

int main(void)
{
    int i;
    float f;
    char c;
```

191Una variabile utilizzata qui è un'espressione.

```
    char *s = _Generic(i,
                    int: "that variable is an int",
                    float: "that variable is a float",
                    default: "that variable is some type"
                );

    printf("%s\n", s);
}
```

Controlla l'espressione `_Generic` che inizia alla riga 9.

Quando il compilatore lo vede, esamina il tipo del primo argomento. (In questo esempio il tipo della variabile `i`.) Quindi esamina i casi per qualcosa di quel tipo. Quindi sostituisce l'argomento al posto dell'intera espressione `_Generic`.

In questo caso `i` è un `int`, quindi corrisponde a quel caso. Quindi la stringa viene sostituita nell'espressione. E perciò la riga diventa questa quando il compilatore la vede:

```
char *s = "that variable is an int";
```

Se il compilatore non riesce a trovare una corrispondenza di tipo in `_Generic`, cerca il caso `default` opzionale e lo utilizza.

Se non riesce a trovare una corrispondenza di tipo e non esiste un valore `default` riceverai un errore di compilazione. La prima espressione **deve** corrispondere a uno dei tipi o al valore `default`.

Poiché è scomodo scrivere `_Generic` più e più volte, viene spesso utilizzato per creare il corpo di una macro che può essere facilmente riutilizzato ripetutamente.

Facciamo una macro `TYPESTR(x)` che accetta un argomento e restituisce una stringa con il tipo dell'argomento.

Quindi `TYPESTR(1)` restituirà ad esempio la stringa "`int`".

Eccoci qui:

```
#include <stdio.h>

#define TYPESTR(x) _Generic((x), \
                        int: "int", \
                        long: "long", \
                        float: "float", \
                        double: "double", \
                        default: "something else")

int main(void)
{
    int i;
    long l;
    float f;
    double d;
    char c;

    printf("i is type %s\n", TYPESTR(i));
    printf("l is type %s\n", TYPESTR(l));
    printf("f is type %s\n", TYPESTR(f));
    printf("d is type %s\n", TYPESTR(d));
    printf("c is type %s\n", TYPESTR(c));
}
```

Questo output:

```
i is type int
l is type long
f is type float
d is type double
c is type something else
```

Il che non dovrebbe sorprendere, perché come abbiamo detto il codice in `main()` viene sostituito con il seguente quando viene compilato:

```
printf("i is type %s\n", "int");
printf("l is type %s\n", "long");
printf("f is type %s\n", "float");
printf("d is type %s\n", "double");
printf("c is type %s\n", "something else");
```

E questo è esattamente l'output che vediamo.

Facciamone un altro. Ho incluso alcune macro qui in modo che quando esegui:

```
int i = 10;
char *s = "Foo!";

PRINT_VAL(i);
PRINT_VAL(s);
```

ottieni l'output:

```
i = 10
s = Foo!
```

Per farlo dovremo usare qualche macro magiaca.

```
#include <stdio.h>
#include <string.h>

// Macro che restituisce un identificatore di formato per un tipo
#define FMTSPEC(x) _Generic((x), \
                        int: "%d", \
                        long: "%ld", \
                        float: "%f", \
                        double: "%f", \
                        char *: "%s")
                        // TODO: add more types

// Macro che stampa una variabile nel modulo "name = value"
#define PRINT_VAL(x) do { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
} while(0)

int main(void)
{
    int i = 10;
    float f = 3.14159;
    char *s = "Hello, world!";

    PRINT_VAL(i);
    PRINT_VAL(f);
    PRINT_VAL(s);
```

```
}
```

per l'output:

```
i = 10
f = 3.141590
s = Hello, world!
```

Avremmo potuto racchiudere tutto in un unico grande macro, ma l'ho diviso in due per evitare sanguinamenti agli occhi.

33. Array Parte II

In questo capitolo esamineremo alcune cose extra riguardanti gli array.

- Qualificatori di tipo con parametri di array
- La parola chiave `static` con parametri di array
- Inizializzatori di array multidimensionali parziali

Non si vedono molto comunemente ma gli daremo un'occhiata dato che fanno parte delle specifiche più recenti.

33.1. Qualificatori di tipo per array negli elenchi di parametri

Se ricordi in precedenza queste due cose sono equivalenti negli elenchi di parametri di funzione:

```
int func(int *p) {...}
int func(int p[]) {...}
```

E potresti anche ricordare che puoi aggiungere qualificatori di tipo a una variabile puntatore in questo modo:

```
int *const p;
int *volatile p;
int *const volatile p;
// etc.
```

Ma come possiamo farlo quando utilizziamo la notazione di array nell'elenco dei parametri?

Scopriamo ora che va tra parentesi. E puoi inserire il conteggio facoltativo dopo. Le due righe seguenti sono equivalenti:

```
int func(int *const volatile p) {...}
int func(int p[const volatile]) {...}
int func(int p[const volatile 10]) {...}
```

Se hai un array multidimensionale devi inserire i qualificatori del tipo nella prima serie di parentesi.

33.2. `static` per gli array negli elenchi di parametri

Allo stesso modo è possibile utilizzare la parola chiave static nell'array in un elenco di parametri. Questo è qualcosa che non ho mai visto in natura. È **sempre** seguito da una dimensione:

```
int func(int p[static 4]) {...}
```

Cosa significa? Nell'esempio sopra è il compilatore che presuppone qualsiasi array passato alla funzione sarà composto da *almeno* 4 elementi. Tutto il resto è un comportamento indefinito.

```
int func(int p[static 4]) {...}
```

```
int main(void)
{
    int a[] = {11, 22, 33, 44};
    int b[] = {11, 22, 33, 44, 55};
    int c[] = {11, 22};

    func(a);  // OK! a è di 4 elementi, il minimo
    func(b);  // OK! b è di almeno 4 elementi
    func(c);  // Comportamento indefinito! c è sotto i 4 elementi!
}
```

Questo imposta fondamentalmente la dimensione minima dell'array che puoi avere.

Nota importante: non c'è nulla nel compilatore che vieti di passare in un array più piccolo. Il compilatore probabilmente non ti avviserà e non lo rileverà in fase di esecuzione.

Inserendo `static` lì dentro stai dicendo: “Faccio la doppia PROMESSA segreta che non passerò mai in un array più piccolo di questo.” E il compilatore dice: "Sì, va bene" e confida che tu non lo faccia.

E poi il compilatore può apportare determinate ottimizzazioni al codice con la certezza che tu, il programmatore farai sempre la cosa giusta.

33.3. Inizializzatori equivalenti

C è un po' diciamo… *flessibile* quando si tratta di inizializzatori di array.

Ne abbiamo già visto alcuni in cui tutti i valori mancanti vengono sostituiti con zero.

Ad esempio possiamo inizializzare un array di 5 elementi su `1,2,0,0,0` con questo:

```
int a[5] = {1, 2};
```

Oppure imposta un array interamente di zero con:

```
int a[5] = {0};
```

Ma le cose si fanno interessanti quando si inizializzano array multidimensionali. Creiamo un array di 3 righe e 2 colonne:

```
int a[3][2];
```

Scriviamo del codice per inizializzarlo e stampare il risultato:

```
#include <stdio.h>

int main(void)
{
    int a[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 2; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}
```

E quando lo eseguiamo otteniamo ciò che ci aspettiamo:

```
1 2
3 4
5 6
```

Tralasciamo alcuni degli elementi inizializzatori e vediamo che vengono impostati su zero:

```
int a[3][2] = {
    {1, 2},
    {3},    // Lascia fuori il 4!
    {5, 6}
};
```

che produce:

```
1 2
3 0
5 6
```

Ora tralasciamo l'intero ultimo elemento centrale:

```
    int a[3][2] = {
        {1, 2},
        // {3, 4},
// Basta, taglia via tutta questa cosa
        {5, 6}
    };
```

E ora capiamo questo che potrebbe non essere quello che ti aspetti:

```
1 2
5 6
0 0
```

Ma se ti fermi a pensarci abbiamo fornito inizializzatori sufficienti solo per due righe quindi sono stati utilizzati per le prime due righe. E gli elementi rimanenti sono stati inizializzati a zero. Fin qui tutto bene. Generalmente se tralasciamo parti dell'inizializzatore il compilatore imposta gli elementi corrispondenti a `0`. Ma diventiamo *pazzi*.

```
int a[3][2] = { 1, 2, 3, 4, 5, 6 };
```

Che cosa—? Questo è un array 2 ma ha solo un inizializzatore 1D! Risulta che è legale (anche se GCC lo avviserà con gli avvisi appropriati attivati). Fondamentalmente ciò che fa è iniziare a riempire gli elementi nella riga 0, quindi nella riga 1, quindi nella riga 2 da sinistra a destra. Perciò quando stampiamo viene stampato in ordine:

```
1 2
3 4
5 6
```

Se ne lasciamo alcuni fuori:

```
int a[3][2] = { 1, 2, 3 };
```

si riempiono con `0`:

```
1 2
3 0
0 0
```

Quindi se vuoi riempire l'intero array con `0` vai avanti e:

```
int a[3][2] = {0};
```

Ma il mio consiglio è se hai un array 2D di utilizzare un inizializzatore 2D. Rende semplicemente il codice più leggibile. (Tranne che per inizializzare l'intero array con 0 nel qual caso è idiomatico utilizzare `{0}` indipendentemente dalla dimensione dell'array.)

34. Salti in lungo con `setjmp, longjmp`

Abbiamo già visto `goto`, che salta nell'ambito della funzione. Ma `longjmp()` ti consente di tornare a un punto precedente dell'esecuzione a una funzione che ha chiamato questo.

Ci sono molte limitazioni e avvertenze ma questa può essere una funzione utile per uscire dal profondo dello stack di chiamate fino a uno stato precedente.

Nella mia esperienza questa è una funzionalità utilizzata molto raramente.

34.1. Utilizzando `setjmp` e `longjmp`

La danza che faremo qui consiste sostanzialmente nel mettere in esecuzione un segnalibro con `setjmp()`. Successivamente chiameremo `longjmp()` e torneremo al punto precedente dell'esecuzione in cui abbiamo impostato il segnalibro con `setjmp()`.

E può farlo anche se hai chiamato sottofunzioni.

Ecco una breve demo in cui chiamiamo funzioni a un paio di livelli di profondità e poi ne usciamo.

Utilizzeremo una variabile di ambito file `env` per mantenere lo *stato* delle cose quando chiamiamo `setjmp()` in modo da poterli ripristinare quando chiamiamo `longjmp()` in seguito. Questa è la variabile in cui ricordiamo il nostro "posto".

La variabile `env` è di tipo `jmp_buf`, un tipo opaco dichiarato in `<setjmp.h>`.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);
// Bail out
    printf("Leaving depth 2\n");
// Questo non succedrà
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // Questo non succedrà
}

int main(void)
{
    switch (setjmp(env)) {
      case 0:
          printf("Calling into functions, setjmp() returned 0\n");
          depth1();
          printf("Returned from functions\n");  // Questo non succedrà
          break;
```

```
        case 3490:
            printf("Bailed back to main, setjmp() returned 3490\n");
            break;
    }
}
```

Quando viene eseguito viene visualizzato questo output:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Se provi a prendere quell'output e ad abbinarlo al codice è chiaro che stanno succedendo alcune cose davvero *strane*.

Una delle cose più notevoli è che `setjmp()` restituisce *due volte*. Qual è quello sincero? Cos'è questa stregoneria?!

Quindi ecco l'accordo: se `setjmp()` restituisce `0`, significa che a quel punto hai impostato correttamente il “segnalibro”.

Se restituisce un valore diverso da zero, significa che sei appena tornato al "segnalibro" impostato in precedenza. (E il valore restituito è quello che passi a `longjmp()`.)

In questo modo puoi distinguere tra l'impostazione del segnalibro e il ritorno ad esso in un secondo momento.

Pertanto quando il codice sopra chiama `setjmp()` per la prima volta, `setjmp()` *memorizza* lo stato nella variabile `env` e restituisce `0`. Successivamente quando chiamiamo `longjmp()` con lo stesso `env` ripristina lo stato e `setjmp()` restituisce il valore che `longjmp()` è stato passato.

34.2. Insidie

Sotto il cofano questo è piuttosto semplice. In genere il *puntatore dello stack* tiene traccia delle posizioni in memoria in cui sono archiviate le variabili locali e il *contatore del programma* tiene traccia dell'indirizzo dell'istruzione attualmente in esecuzione[192] .

Quindi se vogliamo tornare a una funzione precedente è fondamentalmente solo questione di ripristinare il puntatore dello stack e il contatore del programma sui valori mantenuti nella variabile `jmp_buf` e assicurarci che il valore restituito sia impostato correttamente. E poi l'esecuzione riprenderà lì.

Ma una serie di fattori confondono questo creando un numero significativo di trappole comportamentali indefinite.

34.2.1. I valori delle variabili locali

Se vuoi i valori di automatic (non-`static` e non-`extern`) variabili locali per persistere nella funzione che ha chiamato `setjmp()` dopo che si è verificato un `longjmp()` è necessario dichiarare tali variabili come `volatile`.

Tecnicamente devono essere `volatile` solo se cambiano tra il momento in cui viene chiamato `setjmp()` e il momento in cui viene chiamato `longjmp()`[193] .

192Sia il "puntatore dello stack" che il "contatore del programma" sono correlati all'architettura sottostante e all'implementazione C e non fanno parte delle specifiche.

193La logica qui è che il programma potrebbe memorizzare temporaneamente un valore in un registro della CPU mentre sta lavorando su di esso. In tale intervallo di tempo il registro mantiene il valore corretto e il valore sullo

Ad esempio se eseguiamo questo codice:

```
int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

e poi `longjmp()` dietro il valore di `x` sarà indeterminato. Se vogliamo risolvere questo problema, `x` deve essere `volatile`:

```
volatile int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

Ora il valore sarà il `30` corretto dopo che `longjmp()` ci riporta a questo punto.

34.2.2. How Much State is Saved?

Quando si esegue `longjmp()`, l'esecuzione riprende dal punto corrispondente a `setjmp()`. E questo è tutto.

Le specifiche sottolineano che è come se a quel punto fossi tornato alla funzione con le variabili locali impostate su qualunque valore avessero quando è stata effettuata la chiamata `longjmp()`.

Le cose che non vengono ripristinate includono parafrasando le specifiche:

- Flag di stato in virgola mobile
- File aperti
- Qualsiasi altro componente della macchina astratta

34.2.3. Non puoi nominare nulla `setjmp`

Non puoi avere identificatori `extern` con il nome `setjmp`. Oppure se `setjmp` è una macro non puoi annullarne la definizione.

Entrambi sono comportamenti indefiniti.

34.2.4. Non puoi usare `setjmp()` in un'espressione più grande

Cioè non puoi fare qualcosa del genere:

```
if (x == 12 && setjmp(env) == 0) { ... }
```

È troppo complesso per essere consentito dalle specifiche a causa delle macchinazioni che devono verificarsi quando si srotola la pila e tutto il resto. Non possiamo riportare `longjmp()` in un'espressione complessa che è stata eseguita solo parzialmente.

Quindi ci sono dei limiti alla complessità di tale espressione.

- Può essere l'intera espressione di controllo del condizionale.
- `if (setjmp(env)) {...}`
- `switch (setjmp(env)) {...}`

stack potrebbe non essere aggiornato. Successivamente i valori del registro verrebbero sovrascritti e le modifiche alla variabile andrebbero perse.

- Può far parte di un'espressione relazionale o di uguaglianza purché l'altro operando sia una costante intera. E il tutto è l'espressione di controllo del condizionale.
- `if (setjmp(env) == 0) {...}`

L'operando di un'operazione logica NOT (`!`) essendo l'intera espressione di controllo.

- ~if (!setjmp(env)) {…}
- Un'espressione autonoma possibilmente castata a `void`.
- `setjmp(env);`
- `(void)setjmp(env);`

34.2.5. Quando non puoi usare `longjmp()`?

È un comportamento indefinito se:

- Non hai chiamato `setjmp()` prima
- Hai chiamato `setjmp()` da un altro thread
- Hai chiamato `setjmp()` nell'ambito di un array a lunghezza variabile (VLA), e l'esecuzione ha lasciato l'ambito di quel VLA prima che venisse chiamato `longjmp()`.
- La funzione contenente `setjmp()` è terminata prima che fosse chiamato `longjmp()`.

Su quest'ultimo "uscito" include i normali ritorni dalla funzione, così come il caso in cui un altro `longjmp()` tornasse a "precedente" nello stack di chiamate rispetto alla funzione in questione.

34.2.6. Non puoi passare `0` a `longjmp()`

Se provi a passare il valore `0` a `longjmp()` cambierà silenziosamente quel valore in `1`.

Poiché `setjmp()` alla fine restituisce questo valore e il fatto che `setjmp()` restituisca `0` ha un significato speciale è vietato restituire `0`.

34.2.7. `longjmp()` e array a lunghezza variabile

Se sei nell'ambito di un VLA e `longjmp()` disponibile la memoria allocata al VLA potrebbe avere una perdita[194] .

La stessa cosa accade se si esegue `longjmp()` su qualsiasi funzione precedente che aveva ancora VLA nell'ambito.

Questa è una cosa che mi ha davvero infastidito la capacità di VLA—che potresti scrivere codice C perfettamente legittimo che spreca memoria. Ma hey—Non sono responsabile delle specifiche.

35. Tipi incompleti

Potrebbe sorprenderti apprendere che questo si compila senza errori:

```
extern int a[];

int main(void)
{
    struct foo *x;
    union bar *y;
    enum baz *z;
```

194Cioè rimani allocato finché il programma non termina senza alcuna possibilità di liberarlo.

```
}
```

Non abbiamo mai indicato una taglia per `a`. E abbiamo puntatori alle `strutct`, `foo`, `bar` e `baz` che non sembrano mai essere dichiarati da nessuna parte.

E gli unici avvisi che ricevo sono che `x, y` e `z` non sono utilizzati.

Questi sono esempi di *tipi incompleti*.

Un tipo incompleto è un tipo della dimensione (cioè la dimensione che otterresti da `sizeof`) per il quale non è noto. Un altro modo di pensarlo è un tipo che non hai finito di dichiarare.

Puoi avere un puntatore a un tipo incompleto ma non puoi dereferenziarlo o utilizzare l'aritmetica del puntatore su di esso. E non puoi usare `sizeof` su di esso.

Quindi cosa puoi farci?

35.1. Caso d'uso: strutture autoreferenziali

Conosco solo un caso d'uso reale: inoltrare riferimenti a `struct` o `union` con strutture autoreferenziali o co-dipendenti. (Utilizzerò `struct` per il resto di questi esempi ma si applicano tutti allo stesso modo anche alle `union`.)

Facciamo prima il classico esempio.

Ma prima di farlo sappi questo! Quando dichiari una `struct`, la `struct` è incompleta finché non viene raggiunta la parentesi graffa di chiusura!

```
struct antelope {               // struct antelope è incompleto qui
    int leg_count;              // Ancora incompleto
    float stomach_fullness;     // Ancora incompleto
    float top_speed;            // Ancora incompleto
    char *nickname;             // Ancora incompleto
};                              // ORA è completo.
```

E allora? Sembra abbastanza sano.

Ma cosa succede se stiamo creando un elenco collegato? Ogni nodo dell'elenco collegato deve avere un riferimento a un altro nodo. Ma come possiamo creare un riferimento a un altro nodo se non abbiamo ancora finito nemmeno di dichiarare il nodo?

L'indennità di C per i tipi incompleti lo rende possibile. Non possiamo dichiarare un nodo ma *possiamo* dichiarare un puntatore a uno anche se è incompleto!

```
struct node {
    int val;
    struct node *next;
// struct node è incompleto, ma va bene!
};
```

Anche se `struct node` è incompleto alla riga 3 possiamo comunque dichiarare un puntatore a uno[195] .

Possiamo fare la stessa cosa se abbiamo due `struct` diverse che si riferiscono l'una all'altra:

```
struct a {
    struct b *x;
// Si riferisce ad un `struct b`
};
```

195https://en.wikipedia.org/wiki/Complex_number

```
struct b {
    struct a *x;
// Si riferisce ad un `struct a`
};
```

Non saremmo mai in grado di realizzare quella coppia di strutture senza le regole rilassate per i tipi incompleti.

35.2. Messaggi di errore di tipo incompleto

Ricevi errori come questi?

```
invalid application of ‘sizeof’ to incomplete typeinvalid use of undefined
typedereferencing pointer to incomplete type
```

Molto probabilmente il problema è che probabilmente hai dimenticato di usare `#include` il file di intestazione che dichiara il tipo.

35.3. Altri tipi incompleti

Dichiarare una `struct` o `union` senza corpo crea un tipo incompleto, ad es. `struct foo;`.

`enums` sono incompleti fino alla parentesi graffa di chiusura.

`void` è un tipo incompleto.

Gli array dichiarati `extern` senza dimensione sono incompleti, ad es:

```
extern int a[];
```

Se si tratta di un array non `extern` senza dimensione seguito da un inizializzatore è incompleto fino alla chiusura della parentesi graffa dell'inizializzatore.

35.4. Caso d'uso: array nei file di intestazione

Può essere utile dichiarare tipi di array incompleti nei file header. In questi casi l'archiviazione effettiva (dove viene dichiarato l'array completo) dovrebbe essere in un singolo file `.c`. Se lo inserisci nel file `.h`, verrà duplicato ogni volta che viene incluso il file di intestazione.

Quindi quello che puoi fare è creare un file di intestazione con un tipo incompleto che si riferisca all'array in questo modo:

```
// File: bar.h

#ifndef BAR_H
#define BAR_H

extern int my_array[];
// Tipo incompleto

#endif
```

E nel file `.c`, definisci effettivamente l'array:

```
// File: bar.c

int my_array[1024];
// Tipo completo!
```

Quindi puoi includere l'intestazione da tutti i posti che desideri e ognuno di questi posti farà riferimento allo stesso sottostante `my_array`.

```c
// File: foo.c

#include <stdio.h>
#include "bar.h"
// includes the incomplete
// type for my_array

int main(void)
{
    my_array[0] = 10;

    printf("%d\n", my_array[0]);
}
```

Quando compili più file ricordati di specificare tutti i file `.c` nel compilatore ma non i file `.h`, ad es.:

```
gcc -o foo foo.c bar.c
```

35.5. Completamento di tipi incompleti

Se hai un tipo incompleto puoi completarlo definendo la `struct`, l'`union`, l'`enum` o l'array completi nello stesso ambito.

```c
struct foo;
// tipo incompleto

struct foo *p;
// puntatore, nessun problema

// struct foo f;
// Errore: tipo incompleto!

struct foo {
    int x, y, z;
};
// Ora la struct foo è completa!

struct foo f;
// Successo!
```

Tieni presente che sebbene `void` sia un tipo incompleto non c'è modo di completarlo. Non che qualcuno pensi mai di fare quella cosa strana. Ma spiega perché puoi farlo:

```c
void *p;
// OK: puntatore al tipo incompleto
```

e nessuno dei due:

```c
void v;
// Errore: dichiarare
// variabile di tipo incompleto

printf("%d\n", *p);
// Errore: dereferenziazione
// tipo incompleto
```

Più si conosce…

36. Numeri complessi

Un piccolo manuale sui numeri complessi[197] rubato direttamente da Wikipedia:

> Un **numero complesso** è un numero che può essere espresso nella forma

Ma questo è tutto ciò che intendo fare. Daremo per scontato che se stai leggendo questo capitolo sai cos'è un numero complesso e cosa vuoi farci.

E tutto ciò di cui abbiamo bisogno sono le facoltà di C per farlo.

Risulta tuttavia, che il supporto dei numeri complessi in un compilatore è una funzionalità *opzionale*. Non tutti i compilatori conformi possono farlo. E quelli che lo fanno potrebbero farlo a vari gradi di completezza.

Puoi verificare se il tuo sistema supporta i numeri complessi con:

```
#ifdef __STDC_NO_COMPLEX__
#error Complex numbers not supported!
#endif
```

Inoltre è presente una macro che indica l'aderenza allo standard ISO 60559 (IEEE 754) per la matematica in virgola mobile con numeri complessi, nonché la presenza del di tipo simbolo `_Imaginary`.

```
#if __STDC_IEC_559_COMPLEX__ != 1
#error Need IEC 60559 complex support!
#endif
```

Maggiori dettagli al riguardo sono descritti nell'Allegato G nelle specifiche C11.

36.1. Tipi complessi

Per usare numeri complessi `#include <complex.h>`.

Con questo ottieni almeno due tipi:

```
_Complex
complex
```

Entrambi significano la stessa cosa quindi potresti anche usare il più carino `complex`.

Ottieni anche alcuni tipi di numeri immaginari se l'implementazione è conforme a IEC 60559:

```
_Imaginary
imaginary
```

Anche questi significano la stessa cosa quindi potresti anche usare il più carino `imaginary`.

Ottieni anche valori per il numero immaginario `$i$` stesso:

```
I
_Complex_I
_Imaginary_I
```

La macro `I` è impostata a `_Imaginary_I` (se disponibile) `or _Complex_I`. Quindi usa semplicemente `I` per i numero immaginario.

Uno a parte: Ho detto che se un compilatore ha `__STDC_IEC_559_COMPLEX__` impostato su 1 deve supportare i tipi `_Imaginary` per essere conforme. Questa è la mia lettura delle specifiche.

197https://en.wikipedia.org/wiki/Complex_number

Tuttavia non conosco un singolo compilatore che supporti effettivamente `_Imaginary` anche se hanno impostato `__STDC_IEC_559_COMPLEX__`. Quindi scriverò del codice con quel tipo qui che non ho modo di testare. Scusa!

OK quindi ora sappiamo che esiste un tipo `complex`, come possiamo usarlo?

36.2. Assegnazione di numeri complessi

Poiché il numero complesso ha una parte reale e una immaginaria ma entrambi si basano su numeri in virgola mobile per memorizzare valori dobbiamo anche dire a C quale precisione usare per quelle parti del numero complesso.

Lo facciamo semplicemente aggiungendo un `float`, `double` o `long double` al ~complex prima o dopo di esso.

Definiamo un numero complesso che utilizza `float` per i suoi componenti:

```
float complex c;
// Le specifiche preferiscono così
complex float c;
// Stessa cosa: l'ordine non ha importanza
```

Quindi è ottimo per le dichiarazioni ma come dobbiamo inizializzarle o assegnarle?

Risulta che possiamo usare una notazione piuttosto naturale. Esempio!

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

Per `$5+2i$` e `$10+3i$` rispettivamente.

36.3. Costruire, decostruire e stampare

Ci stiamo arrivando…

Abbiamo già visto un modo per scrivere un numero complesso:

```
double complex x = 5 + 2*I;
```

Inoltre non ci sono problemi nell'usare altri numeri in virgola mobile per costruirlo:

```
double a = 5;
double b = 2;
double complex x = a + b*I;
```

C'è anche una serie di macro per aiutare a costruirli. Il codice precedente potrebbe essere scritto utilizzando la macro `CMPLX()` in questo modo:

```
double complex x = CMPLX(5, 2);
```

Per quello posso dire nella mia ricerca questi sono *quasi* equivalenti:

```
double complex x = 5 + 2*I;
double complex x = CMPLX(5, 2);
```

Ma la macro `CMPLX()` gestirà ogni volta correttamente gli zeri negativi nella parte immaginaria mentre nell'altro modo potrebbe convertirli in zeri positivi. Io *penso*[198] che ciò sembra implicare

198Questo è stato più difficile da ricercare e prenderò tutte le ulteriori informazioni che chiunque possa darmi. Potrei essere definito come `_Complex_I` o `_Imaginary_I` se quest'ultimo esiste. `_Imaginary_I` gestirà gli zeri con segno, ma `_Complex_I` potrebbe non farlo. Ciò ha implicazioni con i tagli dei rami e altre cose matematiche con numeri complessi. Forse vuoi dire che sto davvero uscendo dal mio campo? In ogni caso le macro `CMPLX()` si comportano come se fosse definita `_Imaginary_I` con zeri con segno anche se `_Imaginary_I` non esiste sul

che se c'è una possibilità che la parte immaginaria sia zero, dovresti usare la macro… ma qualcuno dovrebbe correggermi se sbaglio!

La macro `CMPLX()` funziona sui tipi `double`. Esistono altre due macro per `float` e `long double`: `CMPLXF()` e `CMPLXL()`. (Questi suffissi "f" e "l" compaiono praticamente in tutte le funzioni relative ai numeri complessi.)

Ora proviamo il contrario: se abbiamo un numero complesso come lo scomponiamo nelle sue parti reali e immaginarie?

Qui abbiamo un paio di funzioni che estrarranno le parti reali e immaginarie dal numero: `creal()` e `cimag()`:

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;

printf("x = %f + %fi\n", creal(x), cimag(x));
printf("y = %f + %fi\n", creal(y), cimag(y));
```

per l'output:

```
x = 5.000000 + 2.000000i
y = 10.000000 + 3.000000i
```

Nota la `i` che ho nella stringa di formato `printf()` è una `i` letterale che viene stampata—non fa parte dell'identificatore di formato. Entrambi restituiscono valori da `creal()` e `cimag()` sono `double`.

E come al solito ci sono varianti `float` e `long double` di queste funzioni: `crealf()`, `cimagf()`, `creall()` e `cimagl()`.

36.4. Aritmetica complessa e confronti

L'aritmetica può essere eseguita su numeri complessi anche se il modo in cui funziona matematicamente va oltre lo scopo della guida.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;
    double complex z;

    z = x + y;
    printf("x + y = %f + %fi\n", creal(z), cimag(z));

    z = x - y;
    printf("x - y = %f + %fi\n", creal(z), cimag(z));

    z = x * y;
    printf("x * y = %f + %fi\n", creal(z), cimag(z));

    z = x / y;
    printf("x / y = %f + %fi\n", creal(z), cimag(z));
}
```

per un risultato di:

sistema.

```
x + y = 4.000000 + 6.000000i
x - y = -2.000000 + -2.000000i
x * y = -5.000000 + 10.000000i
x / y = 0.440000 + 0.080000i
```

Puoi anche confrontare due numeri complessi per verificarne l'uguaglianza (o la disuguaglianza):

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;

    printf("x == y = %d\n", x == y);  // 0
    printf("x != y = %d\n", x != y);  // 1
}
```

per l'output:

```
x == y = 0
x != y = 1
```

Sono uguali se entrambi i componenti al test risultano uguali. Tieni presente che come con tutti i valori in virgola mobile potrebbero essere uguali se sono sufficientemente vicini a causa di un errore di arrotondamento[199] .

36.5. Matematica complessa

Ma aspetta! C'è molto più che una semplice aritmetica complessa!

Ecco una tabella riepilogativa di tutte le funzioni matematiche a tua disposizione con i numeri complessi.

Elencherò solo la versione `double` di ciascuna funzione ma per tutti esiste una versione `float` che puoi ottenere aggiungendo `f` al nome della funzione e una versione `long double` che puoi ottenere aggiungendo `l`.

Ad esempio la funzione `cabs()` per calcolare il valore assoluto di un numero complesso ha anche le varianti `cabsf()` e `cabsl()`. Li ometto per brevità.

36.5.1. Funzioni trigonometriche

Funzione	Descrizione
`ccos()`	Coseno
`csin()`	Seno
`ctan()`	Tangente
`cacos()`	Arc coseno

199La semplicità di questa affermazione non rende giustizia all'incredibile quantità di lavoro necessaria per comprendere semplicemente come funziona effettivamente la virgola mobile. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

`casin()`	Arc seno
`catan()`	Play Settlers of Catan
`ccosh()`	Coseno iperbolico
`csinh()`	Seno iperbolico
`ctanh()`	Tangente iperbolica
`cacosh()`	Arco coseno iperbolico
`casinh()`	Arcoseno iperbolico
`catanh()`	Arcotangente iperbolica

36.5.2. Funzioni esponenziali e logaritmiche

Funzione	Descrizione
cexp()	Base-esponenziale
clog()	Logaritmo (base-)naturale.

36.5.3. Power and Absolute Value Functions

Funzione	Descrizione
`cabs()`	Valore assoluto
`cpow()`	Potenza
`csqrt()`	Radice quadrata

36.5.4. Funzioni di manipolazione

Funzione	Descrizione
~creal()	Restituisci la parte reale

`cimag()`	Restituisci la parte immaginaria
`CMPLX()`	Costruisci un numero complesso
`carg()`	Argomento/angolo di fase
`conj()`	Coniugare[200]
`cproj()`	Proiezione sulla sfera di Riemann

37. Tipi interi a larghezza fissa

C ha tutti quei tipi interi piccoli, grandi e più grandi come `int` e `long` e tutto il resto. E puoi guardare nella sezione sui limiti per vedere qual è l'int più grande con `INT_MAX` e così via.

Quanto sono grandi questi tipi? Cioè quanti byte occupano? Potremmo usare `sizeof` per ottenere quella risposta.

E se volessi andare dall'altra parte? Cosa succederebbe se avessi bisogno di un tipo che fosse esattamente 32 bit (4 byte) o almeno 16 bit o qualcosa del genere?

Come possiamo dichiarare un tipo che abbia una certa dimensione?

L'intestazione `<stdint.h>` ci dà un modo.

37.1. I tipi a dimensione di bit

Sia per gli interi con segno che per quelli senza segno possiamo specificare un tipo che corrisponde a un certo numero di bit ovviamente con alcune avvertenze.

E ci sono tre classi principali di questi tipi (in questi esempi la `N` verrebbe sostituita da un certo numero di bit):

- Interi di esattamente una certa dimensione (`intN_t`)
- Interi che hanno almeno una certa dimensione (`int_leastN_t`)
- Numeri interi che abbiano almeno una certa dimensione e siano il più veloci possibile (`int_fastN_t`)[201]

Quanto più veloce è `fast`? Sicuramente forse un po' più velocemente. Probabilmente. Le specifiche non dicono quanto saranno più veloci solo che saranno i più veloci su questa architettura. La maggior parte dei compilatori C sono piuttosto buoni quindi probabilmente lo vedrai usato solo in luoghi in cui deve essere garantita la massima velocità possibile (piuttosto che sperare semplicemente che il compilatore stia producendo un codice piuttosto veloce e lo è).

Infine questi tipi di numeri senza segno hanno una `u` iniziale per differenziarli.

Ad esempio questi tipi hanno il significato elencato corrispondente:

200 Questo è l'unico che stranamente non inizia con una `c` extra iniziale.

201 Alcune architetture hanno dati di dimensioni diverse con cui CPU e RAM possono operare a una velocità maggiore rispetto ad altre. In questi casi, se hai bisogno del numero a 8 bit più veloce, potrebbe invece darti un tipo a 16 o 32 bit perché è semplicemente più veloce. Quindi con questo non saprai quanto è grande il carattere ma sarà almeno grande quanto dici.

```
int32_t w;
// x è esattamente 32 bits, signed
uint16_t x;
// y è esattamente 16 bits, unsigned

int_least8_t y;
// y è di almeno 8 bit, signed

uint_fast64_t z;
// z la rappresentazione più
// veloce è almeno 64 bits, unsigned
```

È garantita la definizione dei seguenti tipi:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

Potrebbero essercene anche altri di diverse larghezze ma questi sono facoltativi.

EHI! Dove sono i tipi fissi come `int16_t`?

Si scopre che sono del tutto facoltativi…a meno che non siano soddisfatte determinate condizioni [203] . E se disponi di un moderno sistema informatico medio tali condizioni probabilmente sono soddisfatte. E se lo sono avrai questi tipi:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Potrebbero essere definite altre varianti con larghezze diverse ma sono facoltative.

37.2. Tipo di dimensione intera massima

Esiste un tipo che puoi utilizzare che contiene i più grandi numeri interi rappresentabili disponibili sul sistema sia con segno che senza segno:

```
intmax_t
uintmax_t
```

Usa questi tipi quando vuoi avere il più grande possibile.

Ovviamente i valori di qualsiasi altro tipo intero con lo stesso segno si adatteranno necessariamente a questo tipo.

37.3. Utilizzo di costanti di dimensione fissa

Se hai una costante che vuoi inserire in un certo numero di bit, puoi usare queste macro per aggiungere automaticamente il suffisso corretto al numero (es. 22L o 3490ULL).

202 Vale a dire il sistema ha interi a 8, 16, 32 o 64 bit senza riempimento che utilizzano la rappresentazione in complemento a due nel qual caso deve essere definita la variante `intN_t` per quel particolare numero di bit.

203 Vale a dire il sistema ha interi a 8, 16, 32 o 64 bit senza riempimento che utilizzano la rappresentazione in complemento a due nel qual caso deve essere definita la variante `intN_t` per quel particolare numero di bit.

```
INT8_C(x)     UINT8_C(x)
INT16_C(x)    UINT16_C(x)
INT32_C(x)    UINT32_C(x)
INT64_C(x)    UINT64_C(x)
INTMAX_C(x)   UINTMAX_C(x)
```

Ancora una volta funzionano solo con valori interi costanti.

Ad esempio possiamo usare uno di questi per assegnare valori costanti in questo modo:

```
uint16_t x = UINT16_C(12);
intmax_t y = INTMAX_C(3490);
```

37.4. Limits of Fixed Size Integers

Abbiamo anche definito alcuni limiti in modo da poter ottenere i valori massimi e minimi per questi tipi:

```
INT8_MAX           INT8_MIN           UINT8_MAX
INT16_MAX          INT16_MIN          UINT16_MAX
INT32_MAX          INT32_MIN          UINT32_MAX
INT64_MAX          INT64_MIN          UINT64_MAX

INT_LEAST8_MAX     INT_LEAST8_MIN     UINT_LEAST8_MAX
INT_LEAST16_MAX    INT_LEAST16_MIN    UINT_LEAST16_MAX
INT_LEAST32_MAX    INT_LEAST32_MIN    UINT_LEAST32_MAX
INT_LEAST64_MAX    INT_LEAST64_MIN    UINT_LEAST64_MAX

INT_FAST8_MAX      INT_FAST8_MIN      UINT_FAST8_MAX
INT_FAST16_MAX     INT_FAST16_MIN     UINT_FAST16_MAX
INT_FAST32_MAX     INT_FAST32_MIN     UINT_FAST32_MAX
INT_FAST64_MAX     INT_FAST64_MIN     UINT_FAST64_MAX

INTMAX_MAX         INTMAX_MIN         UINTMAX_MAX
```

Nota che il `MIN` per tutti i tipi senza segno è `0`, quindi in quanto tale non esiste una macro per questo.

37.5. Specificatori di formato

Per stampare questi tipi è necessario inviare l'identificatore di formato corretto a `printf()`. (E lo stesso problema per ottenere input con `scanf()`.)

Ma come fai a sapere quali sono le dimensioni dei tipi sotto il cofano? Fortunatamente ancora una volta C fornisce alcune macro per aiutare in questo.

Tutto questo può essere trovato in `<inttypes.h>`.

Ora abbiamo un sacco di macro. Come un'esplosione di complessità di macro. Quindi smetterò di elencarli tutti e inserirò semplicemente la lettera minuscola `n` nel punto in cui dovresti inserire `8, 16, 32` o `64` a seconda delle tue esigenze.

Diamo un'occhiata alle macro per stampare interi con segno:

```
PRIdn    PRIdLEASTn    PRIdFASTn    PRIdMAX
PRIin    PRIiLEASTn    PRIiFASTn    PRIiMAX
```

Cerca gli schemi lì. Puoi vedere che esistono varianti per i tipi fisso, minimo, veloce e massimo.

E hai anche una `d` minuscola e una `i` minuscola. Questi corrispondono agli identificatori di formato `printf()` `%d` e `%i`.

Quindi, se ho qualcosa del genere:

```
int_least16_t x = 3490;
```

Posso stamparlo con l'identificatore di formato equivalente per `%d` utilizzando `PRIdLEAST16`.

Ma come? Come usiamo quella macro?

Prima di tutto quella macro specifica una stringa contenente la lettera o le lettere che `printf()` deve utilizzare per stampare quel tipo. Ad esempio potrebbe essere `"d"` o `"ld"`.

Quindi tutto ciò che dobbiamo fare è incorporarlo nella nostra stringa di formato nella chiamata `printf()`.

Per fare ciò possiamo trarre vantaggio da un fatto relativo a C che potresti aver dimenticato: i valori letterali di stringa adiacenti vengono automaticamente concatenati in una singola stringa. Per esempio:

```
printf("Hello, " "world!\n");
// Stampa "Hello, world!"
```

E poiché queste macro sono stringhe letterali possiamo usarle in questo modo:

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main(void)
{
    int_least16_t x = 3490;

    printf("The value is %" PRIdLEAST16 "!\n", x);
}
```

Abbiamo anche una serie di macro per stampare tipi senza segno:

```
PRIon    PRIoLEASTn    PRIoFASTn    PRIoMAX
PRIun    PRIuLEASTn    PRIuFASTn    PRIuMAX
PRIxn    PRIxLEASTn    PRIxFASTn    PRIxMAX
PRIXn    PRIXLEASTn    PRIXFASTn    PRIXMAX
```

In questo caso `o`, `u`, `x` e `X` corrispondono agli identificatori di formato documentati in `printf()`.

E come prima la `n` minuscola dovrebbe essere sostituita con `8`, `16`, `32` o `64`.

Ma proprio quando pensi di averne abbastanza delle macro scopriamo che ne abbiamo un set complementare completo per `scanf()`!

```
SCNdn    SCNdLEASTn    SCNdFASTn    SCNdMAX
SCNin    SCNiLEASTn    SCNiFASTn    SCNiMAX
SCNon    SCNoLEASTn    SCNoFASTn    SCNoMAX
SCNun    SCNuLEASTn    SCNuFASTn    SCNuMAX
SCNxn    SCNxLEASTn    SCNxFASTn    SCNxMAX
```

Ricorda: quando vuoi stampare un tipo intero a dimensione fissa con `printf()` o `scanf()`, prendi la specifica di formato corrispondente corretta da `<inttypes.h>`.

38. Funzionalità di data e ora

> “Il tempo è un'illusione. L'ora di pranzo doppiamente.”

—Ford Prefect, Guida galattica per gli autostoppisti.

Non è troppo complesso ma all'inizio può intimidire, sia con i diversi tipi disponibili sia con il modo in cui possiamo convertirli tra loro.

Combina GMT (UTC) e ora locale e avrai tutto il /solito divertimento/™ con orari e date.

E ovviamente non dimenticare mai la regola d'oro delle date e degli orari: *Non tentare mai di scrivere la propria funzionalità di data e ora. Usa solo quello che ti dà la libreria.*

Il tempo è troppo complesso perché i semplici programmatori mortali possano gestirlo correttamente. Sul serio siamo debitori a tutti coloro che hanno lavorato su qualsiasi libreria di data e ora, quindi inseriscilo nel tuo budget.

38.1. Terminologia e informazioni rapide

Solo un paio di termini veloci nel caso non li avessi già scritti.

- UTC: Coordinated Universal Time è un tempo assoluto, universalmente concordato[204] . Tutti sul pianeta pensano che in questo momento sia la stessa ora UTC… anche se hanno orari locali diversi.
- GMT: Greenwich Mean Time, di fatti lo stesso dell'UTC[205] . Probabilmente vuoi dire UTC o “universal time”. Se stai parlando specificamente del fuso orario GMT, dì GMT. In modo confuso molte delle funzioni UTC di C sono antecedenti all'UTC e si riferiscono ancora al Greenwich Mean Time. Quando lo vedi sappi che C intrende UTC.
- Local time: che ora è il luogo in cui si trova il computer che esegue il programma. Questo è descritto come un offset dall'UTC. Sebbene esistano molti fusi orari nel mondo, la maggior parte dei computer funziona nell'ora locale o nell'ora UTC.

Come regola generale se stai descrivendo un evento che accade una volta, come una voce di registro, o il lancio di un razzo, o quando i puntatori alla fine hanno fatto clic per te, usa UTC.

D'altra parte se si tratta di qualcosa che accade alla stessa ora in ogni fuso orario, come Capodanno o l'ora di cena, utilizza l'ora locale.

Poiché molte linguaggi sono efficaci solo nella conversione tra l'ora UTC e l'ora locale puoi causare molti problemi scegliendo di memorizzare le date nella forma sbagliata. (Chiedimi come lo so.)

38.2. Tipi di data

Ci sono due tipi principali in C quando si tratta di date: `time_t` e `struct tm`.

Le specifiche in realtà non dicono molto su di loro:

- `time_t`: un vero tipo capace di contenere un tempo. Quindi secondo le specifiche potrebbe essere un tipo mobile o un tipo intero. In POSIX (Unix-likes) è un numero intero. Questo contiene *l'ora del calendario*. Che puoi pensare come ora UTC.
- `struct tm`: contiene i componenti di un tempo di calendario. Questo è un /tempo scomposto/vcioè i componenti del tempo come ora, minuto, secondo, giorno, mese, anno, ecc.

Su molti sistemi `time_t` rappresenta il numero di secondi trascorsi da Epoch[206] . Epoch è in un

204Sulla Terra comunque. Chissà quali sistemi pazzeschi usano là fuori…

205OK non uccidermi! GMT è tecnicamente un fuso orario mentre UTC è un sistema orario globale. Inoltre alcuni paesi potrebbero modificare il GMT per l'ora legale, mentre l'UTC non viene mai modificato per l'ora legale.

206https://en.wikipedia.org/wiki/Unix_time

certo senso l'inizio del tempo dal punto di vista del computer che comunemente è il 1 gennaio 1970 UTC. `time_t` può diventare negativo per rappresentare tempi prima di Epoch. Windows si comporta allo stesso modo di Unix da quello che posso dire.

E cosa c'è in una `struct tm`? I seguenti campi:

```
struct tm {
    int tm_sec;    // secondi dopo il minuto -- [0, 60]
    int tm_min;    // minuti dopo l'ora -- [0, 59]
    int tm_hour;   // ore dalla mezzanotte -- [0, 23]
    int tm_mday;   // giorno del mese -- [1, 31]
    int tm_mon;    // mesi da gennaio -- [0, 11]
    int tm_year;   // anni da 1900
    int tm_wday;   // giorni a partire da domenica -- [0, 6]
    int tm_yday;   // giorni da gennaio 1 -- [0, 365]
    int tm_isdst;  // Ora legale flag
};
```

Tieni presente che tutto è a base zero tranne il giorno del mese.

È importante sapere che puoi inserire qualsiasi valore in questi tipi che desideri. Esistono funzioni che aiutano a ottenere l'ora *adesso* ma i tipi contengono un'ora non *l'ora*.

Quindi la domanda diventa: “Come inizializzi i dati di questi tipi e come li converti tra loro?”

38.3. Inizializzazione e conversione tra tipi

Innanzitutto puoi ottenere l'ora corrente e memorizzarla in un `time_t` con la funzione `time()`.

```
time_t now;  // Variabile per tenere il tempo adesso

now = time(NULL);  // Puoi ottenerlo in questo modo...

time(&now);        // ...o questo. Uguale alla riga precedente.
```

Grande! Hai una variabile che ti dà il tempo adesso.

In modo divertente c'è solo un modo portatile per stampare cosa c'è in un `time_t` e questa è la funzione `ctime()` usata raramente che stampa il valore nell'ora locale:

```
now = time(NULL);
printf("%s", ctime(&now));
```

Ciò restituisce una stringa con una forma molto specifica che include una nuova riga alla fine:

```
Sun Feb 28 18:47:25 2021
```

Quindi è un po' inflessibile. Se vuoi più controllo dovresti convertire `time_t` in un `struct tm`.

38.3.1. Conversione di `time_t` in `struct tm`

Esistono due modi sorprendenti per eseguire questa conversione:

- `localtime()`: questa funzione converte `time_t` in una `struct tm` nell'ora locale.
- `gmtime()`: questa funzione converte `time_t` in una `struct tm` in UTC. (Vedi il vecchio GMT che si insinua nel nome di quella funzione?)

Vediamo che ore sono adesso stampando una `struct tm` con la funzione `asctime()`:

```
printf("Local: %s", asctime(localtime(&now)));
printf("  UTC: %s", asctime(gmtime(&now)));
```

Output (Mi trovo nel fuso orario standard del Pacifico):

```
Local: Sun Feb 28 20:15:27 2021
  UTC: Mon Mar  1 04:15:27 2021
```

Una volta che hai il tuo `time_t` in una `struct tm` apre tutti i tipi di porte. Puoi stampare l'ora in vari modi capire quale giorno della settimana è una data e così via. Oppure riconvertirlo in una `time_t`.

Ne parleremo presto!

38.3.2. Convertire `struct tm` in `time_t`

Se vuoi andare dall'altra parte puoi usare `mktime()` per ottenere queste informazioni.

`mktime()` imposta i valori di `tm_wday` e `tm_yday` per te quindi non preoccuparti di compilarli perché verranno semplicemente sovrascritti.

Inoltre puoi impostare `tm_isdst` su `-1` affinché sia lui a determinarlo per te. Oppure puoi impostarlo manualmente su vero o falso.

```
// Don't be tempted to put leading zeros on these numbers (unless you
// mean for them to be in octal)!

struct tm some_time = {
    .tm_year=82,   // anni dal 1900
    .tm_mon=3,     // mesi da gennaio -- [0, 11]
    .tm_mday=12,   // giorno del mese -- [1, 31]
    .tm_hour=12,   // ore dalla mezzanotte -- [0, 23]
    .tm_min=0,     // minuti dopo l'ora -- [0, 59]
    .tm_sec=4,     // secondi dopo il minuto -- [0, 60]
    .tm_isdst=-1,  // Ora legale flag
};

time_t some_time_epoch;

some_time_epoch = mktime(&some_time);

printf("%s", ctime(&some_time_epoch));
printf("Is DST: %d\n", some_time.tm_isdst);
```

Output:

```
Mon Apr 12 12:00:04 1982
Is DST: 0
```

Quando carichi manualmente una `struct tm` del genere dovrebbe essere nell'ora locale. `mktime()` convertirà l'ora locale in un'ora del calendario `time_t`.

Stranamente tuttavia lo standard non ci fornisce un modo per caricare una `struct tm` con un'ora UTC e convertirla in un `time_t`. Se vuoi farlo in Unix-likes, provare il non standard `timegm()`. Su Windows, `_mkgmtime()`.

38.4. Output della data formattata

Abbiamo già visto un paio di modi per stampare sullo schermo l'output della data formattata. Con `time_t` possiamo usare `ctime()` e con `struct tm` possiamo usare `asctime()`.

```
time_t now = time(NULL);
struct tm *local = localtime(&now);
```

```
struct tm *utc = gmtime(&now);

printf("Local time: %s", ctime(&now));     // Ora locale con time_t
printf("Local time: %s", asctime(local));  // Ora locale con struct tm
printf("UTC       : %s", asctime(utc));    // UTC con un struct tm
```

Ma cosa succederebbe se ti dicessi caro lettore che esiste un modo per avere molto più controllo su come è stata stampata la data?

Certo potremmo estrarre singoli campi dalla `struct tm` ma c'è un'ottima funzione chiamata `strftime()` che farà gran parte del duro lavoro per te. È come `printf()` tranne che per le date!

Vediamo alcuni esempi. In ognuno di questi passiamo un buffer di destinazione, un numero massimo di caratteri da scrivere e quindi una stringa di formato (nello stile di—ma non lo stesso—~printf()~) che dice a `strftime()` quali componenti di una `struct tm` stampare e come.

Puoi anche aggiungere altri caratteri costanti da includere nell'output nella stringa di formato proprio come con `printf()`.

In questo caso otteniamo una `struct tm` da `localtime()` ma qualsiasi sorgente funziona bene.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
    puts(s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

Ci sono un *sacco* di specificatori del formato di stampa della data per `strftime()` quindi assicurati di controllarli in strftime() reference page[207] .

207https://beej.us/guide/bgclr/html/split/time.html#man-strftime

38.5. Più risoluzione con `timespec_get()`

Puoi ottenere il numero di secondi e nanosecondi trascorsi da Epoch con `timespec_get()`.

Forse.

Le implementazioni potrebbero non avere una risoluzione di nanosecondi (ovvero un miliardesimo di secondo) quindi chissà quanti posti significativi otterrai ma provaci e vedrai.

`timespec_get()` accetta due argomenti. Uno è un puntatore a una `struct timespec` per contenere le informazioni sull'ora. E l'altra è la `base` che le specifiche ti consentono di impostare su `TIME_UTC` indicando che ti interessano i secondi a partire da Epoch. (Altre implementazioni potrebbero darti più opzioni per la `base`.)

E la struttura stessa ha due campi:

```
struct timespec {
    time_t tv_sec;   // Secondi
    long   tv_nsec;  // Nanosecondi (miliardesimi di secondo)
};
```

Ecco un esempio in cui otteniamo l'ora e la stampiamo sia come valori interi che come valore mobile:

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Esempio di output:

```
1614581530 s, 806325800 ns
1614581530.806326 seconds since epoch
```

`struct timespec` fa anche la sua comparsa in una serie di funzioni di threading che devono essere in grado di specificare l'ora con quella risoluzione.

38.6. Differenze tra i tempi

Una breve nota su come ottenere la differenza tra due `time_t`s: poiché le specifiche non determinano come quel tipo rappresenta un tempo potresti non essere in grado di sottrarre semplicemente due `time_t` e ottenere qualcosa di sensato[208] .

Fortunatamente puoi usare `difftime()` per calcolare la differenza in secondi tra due date.

Nell'esempio seguente abbiamo due eventi che si verificano a distanza di tempo e utilizziamo `difftime()` per calcolare la differenza.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct tm time_a = {
```

208Lo farai su POSIX dove `time_t` è sicuramente un numero intero. Sfortunatamente il mondo intero non è POSIX quindi eccoci qui.

```
        .tm_year=82,    // anni dal 1900
        .tm_mon=3,      // mesi da gennaio -- [0, 11]
        .tm_mday=12,    // giorno del mese -- [1, 31]
        .tm_hour=4,     // ore dalla mezzanotte -- [0, 23]
        .tm_min=00,     // minuti dopo l'ora -- [0, 59]
        .tm_sec=04,     // secondi dopo il minuto -- [0, 60]
        .tm_isdst=-1,   // Ora legale flag
    };

    struct tm time_b = {
        .tm_year=120,   // anni dal 1900
        .tm_mon=10,     // mesi da gennaio -- [0, 11]
        .tm_mday=15,    // giorno del mese -- [1, 31]
        .tm_hour=16,    // ore dalla mezzanotte -- [0, 23]
        .tm_min=27,     // minuti dopo l'ora -- [0, 59]
        .tm_sec=00,     // secondi dopo il minuto -- [0, 60]
        .tm_isdst=-1,   // Ora legale flag
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // abbastanza vicino

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

E il gioco è fatto! Ricordati di usare `difftime()` per prendere la differenza di tempo. Anche se puoi semplicemente sottrarre su un sistema POSIX magari rimane anche portatile.

39. Multithreading

C11 ha introdotto formalmente il multithreading nel linguaggio C che è molto stranamente simile ai thread POSIX[209] se li hai mai usati.

E se non lo hai usati non preoccuparti. Ne parleremo.

Tieni presente tuttavia non voglio che questo sia un classico tutorial sul multithreading in piena regola[210] ; dovrai consultare molto spesso un altro libro per quel argomento in particolare. Mi spiace!

Il threading è una funzionalità opzionale. Se un compilatore C11+ definisce `__STDC_NO_THREADS__` i thread **non** saranno presenti nella libreria. Il motivo per cui abbiano deciso di procedere con un senso negativo in quella macro è oltre la mia comprensione ma eccoci qui.

Puoi testarlo in questo modo:

```
#ifdef __STDC_NO_THREADS__
#error I need threads to build this program!
#endif
```

209https://en.wikipedia.org/wiki/POSIX_Threads

210Personalmente sono più un fan del nulla condiviso e le mie abilità con i classici costrutti multithreading sono arrugginite, per usare un eufemismo.

Inoltre potrebbe essere necessario specificare alcune opzioni del linker durante la compilazione. Nel caso di Unix-like prova ad aggiungere un `-lpthreads` alla fine della riga di comando per collegare la libreria `pthreads`[211] :

```
gcc -std=c11 -o foo foo.c -lpthreads
```

Se ricevi errori del linker sul tuo sistema potrebbe essere perché la libreria appropriata non è stata inclusa.

39.1. Background

I thread sono un modo per far sì che tutti quei brillanti core della CPU per cui hai pagato funzionino per te nello stesso programma.

Normalmente un programma C viene eseguito su un singolo core della CPU. Ma se sai come suddividere il lavoro puoi assegnarne alcune parti a un numero di thread e fare in modo che svolgano il lavoro simultaneamente.

Sebbene le specifiche non lo dicano sul tuo sistema è molto probabile che C (o il sistema operativo secondo i suoi ordini) tenterà di bilanciare i thread su tutti i core della CPU.

E se hai più thread che core, e questo va bene. Semplicemente non avrai tutti questi vantaggi se tutti provano a competere per il tempo della CPU.

39.2. Cose che puoi fare

Puoi creare un thread. Inizierà a eseguire la funzione specificata. Anche il thread principale che lo ha generato continuerà a essere eseguito.

E puoi aspettare che il thread venga completato. Questo è chiamato *joining*.

Oppure se non ti interessa quando il thread viene completato e non vuoi aspettare puoi *staccarlo*.

Un thread può *uscire* esplicitamente o può chiudersi implicitamente ritornando dalla sua funzione principale.

Un thread può anche *dormire* per un periodo di tempo, senza fare nulla mentre gli altri thread sono in esecuzione.

Anche il programma `main()` è un thread.

Inoltre abbiamo archiviazione locale del thread, mutex e variabili condizionali. Ma ne parleremo più avanti. Diamo solo un'occhiata alle basi per ora.

39.3. Gare di dati e libreria standard

Alcune delle funzioni nella libreria standard (per esempio `asctime()` e `strtok()`) restituisce o utilizza `static` elementi di dati che non sono thread-safe. Ma in generale a meno che non venga detto diversamente la libreria standard fa uno sforzo per esserlo[212] .

Ma tieni gli occhi aperti. Se una funzione della libreria standard mantiene lo stato tra le chiamate in una variabile che non possiedi o se una funzione restituisce un puntatore a qualcosa che non hai passato non è thread-safe.

211Sì thread con una "p". È l'abbreviazione di POSIX threads, una libreria da cui C11 ha preso liberamente in prestito per l'implementazione dei thread.

212Secondo §7.1.4¶5.

39.4. Creazione e attesa di thread

Hackeriamo qualcosa!

Faremo alcuni thread (create) e attendiamo che vengano completati (join).

Prima però dobbiamo capire delle cose.

Ogni singolo thread è identificato da una variabile opaca di tipo `thrd_t`. È un identificatore univoco per thread nel tuo programma. Quando crei un thread gli viene assegnato un nuovo ID.

Inoltre quando crei il thread devi dargli un puntatore a una funzione da eseguire e un puntatore a un argomento da passargli (o `NULL` se non hai nulla da passare).

Il thread inizierà l'esecuzione sulla funzione specificata.

Quando vuoi attendere il completamento di un thread, devi specificare il suo ID thread in modo che C sappia quale attendere.

Quindi l'idea di base è:

1. Scrivi una funzione che agisca come quella del thread "~main~". Non è propiamente `main()` ma è analogo ad esso. Il thread inizierà a funzionare da lì.
2. Dal thread principale avvia un nuovo thread con `thrd_create()` e passa un puntatore alla funzione da eseguire.
3. In quella funzione chiedi al thread di fare tutto ciò che deve fare.
4. Nel frattempo il thread principale può continuare a fare qualunque cosa *lui* debba fare.
5. Quando il thread principale decide di farlo può attendere il completamento del thread figlio chiamando `thrd_join()`. Generalmente **devi** `thrd_join()` il thread per ripulirlo altrimenti perderai memoria[213]

`thrd_create()` accetta un puntatore alla funzione da eseguire ed è di tipo `thrd_start_t` ovvero `int (*)(void *)`. Questo è greco "un puntatore a una funzione che accetta `void*` come argomento e restituisce un `int`." Facciamo un thread! Lo avvieremo dal thread principale con `thrd_create()` per eseguire una funzione fare qualcos'altro cose e poi attendere il completamento con `thrd_join()`. Ho chiamato la funzione principale `run()` del thread ma puoi nominarlo come preferisci purché i tipi corrispondano a `thrd_start_t`.

```
#include <stdio.h>
#include <threads.h>

// Questa è la funzione che verrà
// eseguita dal thread. Può
// essere chiamato qualsiasi cosa.
//
// arg è il puntatore
// dell'argomento
// passato `thrd_create()`.
//
// Il thread genitore
// otterrà il valore
// restituito da `thrd_join()`'
// dopo.

int run(void *arg)
```

213A meno che tu non `thrd_detach()`. Ne parleremo più avanti.

```
{
    int *a = arg;
// Passeremo in un int* da thrd_create()

    printf("THREAD: Running thread with arg %d\n", *a);

    return 12;
// Valore da ritirare con thrd_join() (ne ho scelti 12 a caso)
}

int main(void)
{
    thrd_t t;
// t will hold the thread ID
    int arg = 3490;

    printf("Launching a thread\n");

    // Avvia un thread
    // sulla funzione run()
    // passando un puntatore a 3490
    // come argomento.
    // Memorizzato anche l'ID del thread in t:

    thrd_create(&t, run, &arg);

    printf("Doing other things while the thread runs\n");

    printf("Waiting for thread to complete...\n");

    int res;  // Holds return value from the thread exit

    // Attendi qui il completamento
    // del thread; memorizzare
    // il valore restituito
    // in res:

    thrd_join(t, &res);

    printf("Thread exited with return value %d\n", res);
}
```

Vedi come abbiamo fatto il `thrd_create()` per chiamare la funzione `run()`? Quindi abbiamo fatto altre cose in `main()` e poi ci siamo fermati e abbiamo aspettato il completamento del thread con `thrd_join()`. Stesso output (il tuo potrebbe variare):

```
Launching a thread
Doing other things while the thread runs
Waiting for thread to complete...
THREAD: Running thread with arg 3490
Thread exited with return value 12
```

Il `arg` che passi alla funzione deve avere una durata sufficientemente lunga in modo che il thread possa rilevarlo prima che scompaia. Inoltre non è necessario che venga sovrascritto dal thread principale prima che il nuovo thread possa utilizzarlo. Diamo un'occhiata a un esempio che avvia 5 thread. Una cosa da notare qui è il modo in cui utilizziamo un array di `thrd_ts` per tenere traccia di tutti gli ID dei thread.

```
#include <stdio.h>
```

```c
#include <threads.h>

int run(void *arg)
{
    int i = *(int*)arg;

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++)

        // NOTA! Nella riga successiva passiamo un puntatore a i,
        // ma ogni thread vede lo stesso puntatore. Quindi
        // stampa cose strane mentre cambio valore qui dentro
        // il principale thread! (Maggiori informazioni nel testodi sotto.)

        thrd_create(t + i, run, &i);

    printf("Doing other things while the thread runs...\n");
    printf("Waiting for thread to complete...\n");

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d complete!\n", res);
    }

    printf("All threads complete!\n");
}
```

Quando eseguo i thread conto `i` da 0 a 4. E passa un puntatore a `thrd_create()`. Questo puntatore finisce nella routine `run()` dove ne creiamo una copia.

Abbastanza semplice? Ecco il risultato:

```
Launching threads...
THREAD 2: running!
THREAD 3: running!
THREAD 4: running!
THREAD 2: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 2 complete!
Thread 2 complete!
THREAD 5: running!
Thread 3 complete!
Thread 4 complete!
Thread 5 complete!
```

```
All threads complete!
```

Cheeee—? Dov'è `THREAD 0`? E perché abbiamo un `THREAD 5` quando chiaramente `i` non è mai più di `4` quando chiamiamo `thrd_create()`? E due `THREAD 2`? Follia!

Questo è entrare nel divertente mondo delle *condizioni di gara*. Il thread principale sta modificando `i` prima che il thread abbia la possibilità di copiarlo. In effetti `i` arriva fino a `5` e termina il ciclo prima che l'ultimo thread abbia la possibilità di copiarlo.

Dobbiamo avere una variabile per thread a cui possiamo fare riferimento in modo da poterla passare come `arg`.

Potremmo avere un grande array di loro. Oppure potremmo `malloc()` spazi (e liberarlo da qualche parte—magari nel thread stesso.)

Proviamoci:

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

int run(void *arg)
{
    int i = *(int*)arg;  // Copia il arg

    free(arg);  // Fatto con questo

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++) {

        // Prendi un po' di spazio per un argomento per thread:

        int *arg = malloc(sizeof *arg);
        *arg = i;

        thrd_create(t + i, run, arg);
    }

    // ...
```

Nota sulle righe 27-30 noi `malloc()` spazi per un `int` e copiamo il valore di `i` dentro. Ogni nuovo thread ottiene il suo nuovo-`malloc()` variabile e passiamo un puntatore a quello alla funzione `run()`.

Una volta `run()` crea la propria copia dell'`arg` alla riga 7, lo `free()` il `malloc()` `int`. E ora che ha la sua copia può farne ciò che vuole.

E una esecuzione mostra il risultato:

```
Launching threads...
THREAD 0: running!
THREAD 1: running!
THREAD 2: running!
THREAD 3: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 0 complete!
Thread 1 complete!
Thread 2 complete!
Thread 3 complete!
THREAD 4: running!
Thread 4 complete!
All threads complete!
```

Eccoci qua! Threads 0-4 tutti attivi!

L'esecuzione potrebbe variare—il modo in cui i thread vengono pianificati per l'esecuzione va oltre le specifiche C. Vediamo nell'esempio sopra che il thread 4 non è nemmeno iniziato finché i thread 0-1 non sono stati completati. In effetti se lo eseguo di nuovo probabilmente otterrò un output diverso. Non possiamo garantire un ordine di esecuzione del thread.

39.5. Distacco dei Threads

Se vuoi sparare e dimenticare un thread (es. così non devi `thrd_join()` più tardi) puoi farlo con `thrd_detach()`.

Ciò rimuove la capacità del thread principale di ottenere il valore restituito dal thread figlio ma se non ti interessa e vuoi solo che i thread si ripuliscano bene da soli questa è la strada da percorrere.

Fondamentalmente lo faremo:

```
thrd_create(&t, run, NULL);
thrd_detach(t);
```

dove la chiamata `thrd_detach()` è il thread genitore che dice "Ehi, non aspetterò che questo thread figlio venga completato con `thrd_join()`. Quindi vai avanti e puliscilo da solo una volta completato."

```
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    (void)arg;

    //printf("Thread running! %lu\n", thrd_current()); // non-portable!
    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t;
```

```
    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);               // <-- DETACH!
    }

    // Sleep for a second to let all the threads finish
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
}
```

Tieni presente che in questo codice mettiamo in pausa il thread principale per 1 secondo con ~thrd$_{sleep}$()~—ne parleremo più avanti.

Anche nella funzione `run()` ho una riga commentata che stampa l'ID del thread come `unsigned long`. Questo non è portatile perché le specifiche non dicono di che tipo è un `thrd_t` sotto il cofano—potrebbe essere una `struct` per quanto ne sappiamo. Ma quella linea funziona sul mio sistema.

Qualcosa di interessante che ho visto quando ho eseguito il codice sopra e stampato gli ID dei thread è stato che alcuni thread avevano ID duplicati! Sembra che dovrebbe essere impossibile ma a C è consentito *riutilizzare* gli ID dei thread dopo che il thread corrispondente è terminato. Quindi quello che stavo vedendo era che alcuni thread completavano la loro esecuzione prima che altri thread venissero avviati.

39.6. Thread Dati locali

I thread sono interessanti perché non hanno una propria memoria oltre alle variabili locali. Se desideri una variabile `static` o una variabile con ambito file tutti i thread vedranno la stessa variabile.

Ciò può portare a condizioni di gara, dove succedono *cose strane*™.

Dai un'occhiata a questo esempio. Abbiamo una variabile `static foo` nell'ambito del blocco in `run()`. Questa variabile sarà visibile a tutti i thread che passano attraverso la funzione `run()`. E i vari thread possono effettivamente pestarsi i piedi a vicenda.

Ogni thread copia `foo` in una variabile locale `x` (che non è condiviso tra i thread—tutti i thread hanno i propri stack di chiamate). Quindi *dovrebbero* essere uguali giusto?

E la prima volta che li stampiamo lo sono[214] . Ma subito dopo controlliamo per assicurarci che siano sempre gli stessi.

E di *solito* lo sono. Ma non sempre!

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

int run(void *arg)
{
    int n = *(int*)arg;  // Thread numero che gli esseri umani possono
differenziare

    free(arg);

    static int foo = 10;  // Valore statico condiviso tra thread
```

214Anche se non penso che debbano esserlo. È solo che i thread non sembrano essere riprogrammati finché non avviene qualche chiamata di sistema come con un `printf()`~… `ed è per questo che ho il ~printf()` lì dentro.

```
    int x = foo;  // Variabile locale automatica: ogni thread ha la propria

    // Abbiamo appena assegnato x da foo, quindi è meglio che siano uguali qui.
    // (In tutti i miei test lo erano, ma anche questo non è garantito!)

    printf("Thread %d: x = %d, foo = %d\n", n, x, foo);

    // E dovrebbero essere uguali qui, ma non lo sono sempre!
    // (A volte lo sono a volte no!)

    // Ciò che accade è che un altro thread entra e aumenta foo
    // radesso, ma la x di questo thread rimane quella di prima!

    if (x != foo) {
        printf("Thread %d: Craziness! x != foo! %d != %d\n", n, x, foo);
    }

    foo++;  // Incrementa il valore condiviso

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Contiene un numero di serie del thread
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Ecco un esempio di output (anche se questo varia da esecuzione a esecuzione):

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 1: Craziness! x != foo! 10 != 11
Thread 2: x = 12, foo = 12
Thread 4: x = 13, foo = 13
Thread 3: x = 14, foo = 14
```

Nel thread 1 tra le due `printf()` il valore di foo è cambiato in qualche modo da `10` a `11`, anche se chiaramente non c'è alcun incremento tra `printf()`!

Era un altro thread che è arrivato lì (probabilmente il thread 0, a quanto pare) e ha incrementato il valore di `foo` dietro il thread 1!

Risolviamo questo problema in due modi diversi. (Se vuoi che tutti i thread condividano la variabile *e* non si pestino i piedi a vicenda dovrai leggere la sezione mutex.)

39.6.1. Classe di archiviazione `_Thread_local`

Per prima cosa diamo un'occhiata al modo più semplice per aggirare questo problema: la classe di

archiviazione `_Thread_local`.

Fondamentalmente lo inseriremo semplicemente nella parte anteriore della nostra variabile `static` nell'ambito del blocco e le cose funzioneranno! Dice a C che ogni thread dovrebbe avere la propria versione di questa variabile quindi nessuno di loro si pesta i piedi a vicenda.

Il header `<threads.h>` definisce `thread_local` come aliass `to _Thread_local` quindi il tuo codice non deve sembrare così brutto.

Prendiamo l'esempio precedente e trasformiamo `foo` in una variabile `thread_local` in modo da non condividere tali dati.

```
int run(void *arg)
{
    int n = *(int*)arg;  // Numero di thread per consentire agli esseri umani
di differenziarsi

    free(arg);

    thread_local static int foo = 10;  // <-- Non più condiviso!!
```

E eseguendolo otteniamo:

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 2: x = 10, foo = 10
Thread 4: x = 10, foo = 10
Thread 3: x = 10, foo = 10
```

Niente più problemi strani!

Una cosa: se una variabile `thread_local` è l'ambito del blocco deve `essere static`. Queste sono le regole. (Ma questo va bene perché le variabili non `static` sono già per thread poiché ogni thread ha le proprie variabili non `static`.)

Una piccola una bugia lì: l'ambito del blocco delle variabili `thread_local` può anche essere `extern`.

39.6.2. Un'altra opzione: Archiviazione specifica del thread

Thread-specific storage (TSS) è un altro modo per ottenere dati per thread.

Una caratteristica aggiuntiva è che queste funzioni consentono di specificare un distruttore che verrà chiamato sui dati quando la variabile TSS viene eliminata. Comunemente questo distruttore è `free()` per pulire automaticamente `malloc()` dati per thread. O `NULL` se non hai bisogno di distruggere nulla.

Il distruttore è tipo `tss_dtor_t` che è un puntatore a una funzione che restituisce `void` e accetta un `void*` come argomento (Il `void*` punta ai dati memorizzati nella variabile). In altre parole è un `void (*)(void*)` se questo chiarisce la situazione. Ammetto che probabilmente non è così. Guarda l'esempio qui sotto.

Generalmente `thread_local` è probabilmente la tua scelta ma se ti piace l'idea del distruttore puoi farne uso.

L'utilizzo è un po' strano in quanto abbiamo bisogno che una variabile di tipo `tss_t` sia attiva per rappresentare il valore in base al thread. Quindi lo inizializziamo con `tss_create()`. E alla fine ce ne liberiamo con `tss_delete()`. Tieni presente che la chiamata `tss_delete()` non esegue tutti i distruttori—è `thrd_exit()` (o di ritorno dalla funzione di esecuzione) che fa quello.

`tss_delete()` rilascia semplicemente tutta la memoria allocata da `tss_create()`.

Nel mezzo i thread possono chiamare `tss_set()` e `tss_get()` per impostare e ottenere il valore.

Nel codice seguente impostiamo la variabile TSS prima di creare i thread quindi puliamo dopo i thread.

Nella funzione `run()` i thread `malloc()` lasciano spazio per una stringa e memorizzano quel puntatore nella variabile TSS.

Quando il thread esce, la funzione distruttore (`free()` in questo caso) viene chiamato per *tutti* i thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <threads.h>

tss_t str;

void some_function(void)
{
    // Recupera il valore per thread di questa stringa
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Ottieni il numero di serie di questo thread
    free(arg);

    // malloc() spazi per contenere i dati per questo thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Felice piccol stringa

    // Imposta questa variabile TSS in modo che punti alla stringa
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0;   // Equivalente a thrd_exit(0)
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
```

```
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Tutti i thread sono terminati, quindi con questo abbiamo finito
    tss_delete(str);
}
```

Ancora una volta questo è un modo piuttosto doloroso di fare le cose rispetto a `thread_local`, quindi a meno che tu non abbia davvero bisogno della funzionalità del distruttore lo userai alternativamente.

39.7. Mutexes

Se vuoi consentire l'accesso a un solo thread alla volta in una sezione critica del codice puoi proteggere quella sezione con un mutex[215] .

Ad esempio se avessimo una variabile `static` e volessimo poterla ottenere e impostarlo in due operazioni senza che un altro thread salti nel mezzo e lo corrompa potremmo usare un mutex per quello.

Puoi acquisire un mutex o rilasciarlo. Se tenti di acquisire il mutex e ci riesci puoi continuare l'esecuzione. Se provi e fallisci (perché qualcun altro lo tiene) lo *bloccherai*[216] finché il mutex non verrà rilasciato.

Se più thread vengono bloccati in attesa del rilascio di un mutex uno di essi verrà scelto per l'esecuzione (a caso dal nostro punto di vista) e gli altri continueranno a dormire.

La strategia prevede che prima inizializziamo una variabile mutex per renderla pronta all'uso con `mtx_init()`.

Quindi i thread successivi possono chiamare `mtx_lock()` e `mtx_unlock()` per ottenere e rilasciare il mutex.

Quando abbiamo completamente finito con il mutex possiamo distruggerlo con `mtx_destroy()` il logico opposto di `mtx_init()`.

Innanzitutto diamo un'occhiata al codice che *non* utilizza un mutex e tentiamo di stampare un numero di serie (`static`) e poi incrementarlo. Perché non stiamo utilizzando un mutex per ottenere il valore (per stamparlo) e l'ambientazione (per incrementarlo) threads potrebbero intralciarsi a vicenda in quella sezione critica.

```
#include <stdio.h>
#include <threads.h>

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Variabile statica condivisa!

    printf("Thread running! %d\n", serial);

    serial++;
```

215Abbreviazione di "mutua esclusione" ovvero un "blocco" su una sezione di codice che solo un thread può eseguire.
216Cioè il tuo processo andrà a dormire.

```
    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Quando lo eseguo ottengo qualcosa che assomiglia a questo:

```
Thread running! 0
Thread running! 0
Thread running! 0
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Chiaramente più thread stanno entrando ed eseguendo il `printf()` prima che qualcuno ottenga una modifica per aggiornare la variabile `serial`.

Ciò che vogliamo fare è racchiudere il recupero della variabile e la sua impostazione in un unico tratto di codice protetto da mutex.

Aggiungeremo una nuova variabile per rappresentare il mutex di tipo `mtx_t` nell'ambito del file inizializzarla e quindi i thread potranno bloccarla e sbloccarla nella funzione `run()`.

```
#include <stdio.h>
#include <threads.h>

mtx_t serial_mtx;     // <-- VARIABILE MUTEX

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Shared static variable!

    // Acquire the mutex--all threads will block on this call until
    // they get the lock:

    mtx_lock(&serial_mtx);           // <-- ACQUISISCE MUTEX

    printf("Thread running! %d\n", serial);

    serial++;
```

```
    // Done getting and setting the data, so free the lock. This will
    // unblock threads on the mtx_lock() call:

    mtx_unlock(&serial_mtx);         // <-- RILASCIO MUTEX

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Initialize the mutex variable, indicating this is a normal
    // no-frills, mutex:

    mtx_init(&serial_mtx, mtx_plain);        // <-- CREA MUTEX

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Done with the mutex, destroy it:

    mtx_destroy(&serial_mtx);                // <-- DISTRUGGE MUTEX
}
```

Scopri come alle righe 38 e 50 del `main()` inizializziamo e distruggiamo il mutex.

Ma ogni singolo thread acquisisce il mutex alla riga 15 e lo rilascia alla riga 24.

In mezzo a `mtx_lock()` e `mtx_unlock()` è la *sezione critica* l'area del codice in cui non vogliamo che più thread si muovano contemporaneamente.

E ora otteniamo l'output corretto!

```
Thread running! 0
Thread running! 1
Thread running! 2
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Se hai bisogno di più mutex nessun problema: otteniamo più variabili mutex.

E ricorda sempre la regola numero uno dei mutex multipli: *Sblocca i mutex nell'ordine opposto a quello in cui li blocchi!*

39.7.1. Diversi tipi di mutex

Come accennato in precedenza abbiamo alcuni tipi di mutex che puoi creare con `mtx_init()`.

(Alcuni di questi tipi sono il risultato di un'operazione OR bit per bit come indicato nella tabella.)

Tipo	Descrizione
`mtx_plain`	Il vecchio mutex normale
`mtx_timed`	Mutex che supporta i timeout
`mtx_plain\vertmtx_recursive`	Mutex ricorsivo
`mtx_timed\vertmtx_recursive`	Mutex ricorsivo che supporta i timeout

“Ricursivo” significa che il titolare di un blocco può chiamare `mtx_lock()` più volte sullo stesso blocco. (Devono sbloccarlo un numero uguale di volte prima che qualcun altro possa prendere il mutex.) Ciò potrebbe facilitare la codifica di tanto in tanto specialmente se chiami una funzione che deve bloccare il mutex quando hai già il mutex.

E il timeout dà a un thread la possibilità di provare a *ottenere* il blocco per un po' ma poi salvarsi se non riesce a ottenerlo in quel lasso di tempo.

Per un mutex timeout assicurati di crearlo con `mtx_timed`:

```
mtx_init(&serial_mtx, mtx_timed);
```

E poi quando lo aspetti devi specificare un'ora in UTC in cui si sbloccherà[217] .

La funzione `timespec_get()` di `<time.h>` può essere di aiuto in questo caso. Ti darà l'ora corrente in UTC in una `struct timespec` che è proprio ciò di cui abbiamo bisogno. In effetti sembra esistere solo per questo scopo.

Ha due campi: `tv_sec` ha l'ora corrente in secondi dall'epoca e `tv_nsec` ha i nanosecondi (miliardesimi di secondo) come parte “frazionaria”.

Quindi puoi caricarlo con l'ora corrente e quindi aggiungerlo per ottenere un timeout specifico.

Poi chiama `mtx_timedlock()` invece di `mtx_lock()`. Se restituisce il valore `thrd_timedout` è scaduto.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Ottieni l'ora corrente
timeout.tv_sec += 1;               // Timeout 1 secondo dopo adesso

int result = mtx_timedlock(&serial_mtx, &timeout));

if (result == thrd_timedout) {
    printf("Mutex lock timed out!\n");
}
```

A parte questo le serrature temporizzate sono identiche alle serrature normali.

39.8. Variabili di condizione

Le variabili di condizione sono l'ultimo pezzo del puzzle di cui abbiamo bisogno per realizzare

217Potresti aspettarti che fosse "tempo da adesso", ma ti piacerebbe solo pensarlo non è vero?

applicazioni multithread performanti e per comporre strutture multithread più complesse.

Una variabile di condizione fornisce un modo per i thread di andare in modalità di sospensione finché non si verifica un evento su un altro thread.

In altre parole potremmo avere un certo numero di thread in fase di elaborazione ma devono attendere fino a quando si verifica un evento prima di continuare. Fondamentalmente gli viene detto "aspetta!" finché non vengono avvisati.

E questo funziona di pari passo con i mutex poiché ciò che aspetteremo generalmente dipende dal valore di alcuni dati e tali dati generalmente devono essere protetti da un mutex.

È importante notare che dal nostro punto di vista la variabile di condizione in sé non è proprietaria di alcun dato particolare. È semplicemente la variabile con cui C tiene traccia dello stato di attesa/non attesa di un particolare thread o gruppo di thread.

Scriviamo un programma artificioso che legga in gruppi di 5 numeri dal thread principale uno alla volta. Quindi, quando sono stati inseriti 5 numeri il thread figlio si sveglia somma quei 5 numeri e stampa il risultato.

I numeri verranno memorizzati in un array globale e condiviso così come l'indice nell'array del numero che sta per essere inserito.

Poiché questi sono valori condivisi dobbiamo almeno nasconderli dietro un mutex sia per il thread principale che per quello figlio. (Il principale scriverà loro i dati e il figlio leggerà i dati da loro.)

Ma non basta. Il thread figlio deve bloccarsi (“dorme”) finché non vengono letti 5 numeri nell'array. E quindi il thread genitore deve riattivare il thread figlio in modo che possa svolgere il suo lavoro.

E quando si sveglia deve trattenere quel mutex. E lo farà! Quando un thread attende una variabile di condizione acquisisce anche un mutex quando si riattiva.

Tutto ciò avviene attorno ad una variabile aggiuntiva di tipo `cnd_t` che è la *variabile di condizione*. Creiamo questa variabile con la funzione `cnd_init()` e la distruggiamo quando abbiamo finito con la funzione `cnd_destroy()`.

Ma come funziona tutto questo? Diamo un'occhiata allo schema di ciò che farà il thread figlio:

1. Blocca il mutex con `mtx_lock()`
2. Se non abbiamo inserito tutti i numeri, attendiamo la variabile di condizione con `cnd_wait()`
3. Fai il lavoro che deve essere fatto
4. Sblocca il mutex con `mtx_unlock()`

Nel frattempo il thread principale si occuperà di questo:

1. Blocca il mutex con `mtx_lock()`
2. Memorizza il numero letto di recente nell'array
3. Se l'array è pieno segnala al figlio di svegliarsi `cnd_signal()`
4. Sblocca il mutex con `mtx_unlock()`

Se non l'hai sfogliato troppo attentamente (È OK—Non sono offeso) potresti notare qualcosa di strano: come può il thread principale mantenere il blocco mutex e segnalare al figlio se il figlio deve tenere il blocco mutex per attendere il segnale? Non possono tenere entrambi la serratura!

E infatti non lo fanno! C'è della magia dietro le quinte con le variabili di condizione: quando usi

`cnd_wait()` rilascia il mutex specificato e il thread va in stop. E quando qualcuno segnala a quel thread di attivarsi riacquista il blocco come se nulla fosse successo.

È un po' diverso per quanto riguarda `cnd_signal()`. Questo non fa nulla con il mutex. Il thread di segnalazione deve comunque rilasciare manualmente il mutex prima che i thread in attesa possano attivarsi.

Ancora una cosa su `cnd_wait()`. Probabilmente chiamerai `cnd_wait()` se qualche condizione [219] non è ancora soddisfatta (es. in questo caso ad esempio se non sono stati ancora inseriti tutti i numeri). Ecco l'accordo: questa condizione dovrebbe trovarsi in un ciclo `while` non in un'istruzione `if`. Perché?

È a causa di un fenomeno misterioso chiamato risveglio spurio. A volte in alcune implementazioni, un thread può essere svegliato da una modalità di sospensione `cnd_wait()` apparentemente senza motivo. [X-Files music][220] . Dobbiamo quindi verificare che la condizione di cui abbiamo bisogno sia effettivamente ancora soddisfatta al risveglio. E se così non fosse torna a dormire con noi!

Quindi facciamo questa cosa! A partire dal main thread:

- Il thread principale imposterà il mutex e la variabile di condizione e avvierà il thread figlio.
- Quindi in un ciclo infinito otterrà i numeri come input dalla console.
- Acquisirà inoltre il mutex per memorizzare il numero immesso in un array globale.
- Quando l'array contiene 5 numeri il thread principale segnalerà al thread figlio che è ora di svegliarsi e fare il suo lavoro.
- Quindi il thread principale sbloccherà il mutex e tornerà a leggere il numero successivo dalla console.

Nel frattempo il thread figlio ha fatto le sue losche attività:

- Il thread figlio cattura il mutex
- Mentre la condizione non è soddisfatta (es. mentre l'array condiviso non contiene ancora 5 numeri) il thread figlio dorme aspettando la variabile di condizione. Quando attende sblocca implicitamente il mutex.
- Una volta che il thread principale segnala al thread figlio di attivarsi si attiva per eseguire il lavoro e ripristina il blocco mutex.
- Il thread figlio somma i numeri e reimposta la variabile che è l'indice nell'array.
- Quindi rilascia il mutex e viene eseguito nuovamente in un ciclo infinito.

Ed ecco il codice! Studialo un po' in modo da poter vedere dove vengono gestiti tutti i pezzi di sopra:

```
#include <stdio.h>
#include <threads.h>

#define VALUE_COUNT_MAX 5

int value[VALUE_COUNT_MAX];  // Globale condiviso
int value_count = 0;   // Anche questa globale condiviso
```

218Ed è per questo che si chiamano variabili di condizione!
219Ed è per questo che si chiamano variabili di condizione!
220Non sto dicendo che siano alieni… ma sono alieni. OK molto più probabilmente un altro thread potrebbe essere stato attivato e messo al lavoro per primo.

```c
mtx_t value_mtx;   // Mutex attorno al valore
cnd_t value_cnd;   // Variabile condizionale sul valore

int run(void *arg)
{
    (void)arg;

    for (;;) {
        mtx_lock(&value_mtx);      // <-- PRENDI IL MUTEX

        while (value_count < VALUE_COUNT_MAX) {
            printf("Thread: is waiting\n");
            cnd_wait(&value_cnd, &value_mtx);  // <-- CONDIZIONE ATTESA
        }

        printf("Thread: is awake!\n");

        int t = 0;

        // Add everything up
        for (int i = 0; i < VALUE_COUNT_MAX; i++)
            t += value[i];

        printf("Thread: total is %d\n", t);

        // Reimposta l'indice di input per il thread principale
        value_count = 0;

        mtx_unlock(&value_mtx);   // <-- SBLOCCA MUTEX
    }

    return 0;
}

int main(void)
{
    thrd_t t;

    // Apri un nuovo thread

    thrd_create(&t, run, NULL);
    thrd_detach(t);

    // Imposta il mutex e la variabile di condizione

    mtx_init(&value_mtx, mtx_plain);
    cnd_init(&value_cnd);

    for (;;) {
        int n;

        scanf("%d", &n);

        mtx_lock(&value_mtx);    // <-- BLOCCA MUTEX

        value[value_count++] = n;

        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
```

```
            cnd_signal(&value_cnd);  // <-- CONDIZIONE DEL SEGNALE
        }

        mtx_unlock(&value_mtx);  // <-- SBLOCCA MUTEX
    }

    // Ripulire (So che qui sopra c'è un ciclo infinito, ma
    // voglio almeno fingere di essere corretto):

    mtx_destroy(&value_mtx);
    cnd_destroy(&value_cnd);
}
```

Ed ecco alcuni esempi di output (i singoli numeri sulle righe sono il mio input):

```
Thread: is waiting
1
1
1
1
1
Main: signaling thread
Thread: is awake!
Thread: total is 5
Thread: is waiting
2
8
5
9
0
Main: signaling thread
Thread: is awake!
Thread: total is 24
Thread: is waiting
```

È un uso comune delle variabili di condizione in situazioni produttore-consumatore come questa. Se non avessimo un modo per mettere in stop il thread figlio mentre attende che vengano soddisfatte alcune condizioni sarebbe forzato chiedere le statistiche il che è un grande spreco di CPU.

39.8.1. Timed Condition Wait

Esiste una variante di `cnd_wait()` che ti consente di specificare un timeout in modo da poter interrompere l'attesa. Poiché il thread figlio deve ribloccare il mutex ciò non significa necessariamente che tornerai in vita nell'istante in cui si verifica il timeout; devi ancora attendere che altri thread rilascino il mutex. Ma significa che non aspetterai finché non si verifica `cnd_signal()`. Per farlo funzionare chiama `cnd_timedwait()` invece di `cnd_wait()`. Se restituisce il valore `thrd_timedout` è scaduto. Il timestamp è un tempo assoluto in UTC non un tempo da adesso. Per fortuna la funzione `timespec_get()` in `<time.h>` sembra fatta su misura esattamente per questo caso.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Ottieni l'ora corrente
timeout.tv_sec += 1;               // Timeout 1 secondo dopo adesso

int result = cnd_timedwait(&condition, &mutex, &timeout));

if (result == thrd_timedout) {
```

```
    printf("Condition variable timed out!\n");
}
```

39.8.2. Broadcast: Riattiva tutti i thread in attesa

`cnd_signal()` funzione]] `cnd_signal()` riattiva solo un thread per continuare a lavorare. A seconda di come hai fatto la logica potrebbe avere senso riattivare più di un thread per continuare una volta soddisfatta la condizione.

Ovviamente solo uno di loro può afferrare il mutex ma se hai una situazione in cui:

- Il thread appena risvegliato è responsabile del risveglio di quello successivo e—
- Allora c'è la possibilità che la condizione del ciclo di risveglio spurio gli impedisca di farlo —

ti consigliamo di trasmettere la sveglia in modo da essere sicuro di far uscire almeno uno dei thread da quel ciclo per avviare quello successivo.

Come? Ti chiedi?

Usa semplicemente `cnd_broadcast()` invece di `cnd_signal()`. Stesso identico utilizzo, tranne `cnd_broadcast()` riattiva **tutti** i thread dormienti che erano in attesa su quella variabile di condizione.

39.9. Esecuzione di una funzione una volta

Diciamo che hai una funzione che *potrebbe* essere eseguita da molti thread, ma non sai quando e non funziona prova a scrivere tutta quella logica.

C'è un modo per aggirare il problema: usa `call_once()`. Tonnellate di thread potrebbero provare a eseguire la funzione ma conta solo il primo[221]

Per lavorare con questo hai bisogno di una variabile flag speciale che dichiari per tenere traccia se l'operazione è stata eseguita o meno. E hai bisogno di una funzione da eseguire che non accetta parametri e non restituisce alcun valore.

```
once_flag of = ONCE_FLAG_INIT;  // Inizializzalo in questo modo

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    call_once(&of, run_once_function);

    // ...
```

In questo esempio non importa quanti thread arrivano alla funzione `run()`, `run_once_function()` verrà chiamata una sola volta.

40. Atomici

> “Ci hanno provato e hanno fallito, tutti?” "Oh no." Scosse la testa. "Ci hanno provato e

221La sopravvivenza del più forte! Giusto? Ammetto che in realtà non è niente del genere.

sono morti."

—Paul Atreides and The Reverend Mother Gaius Helen Mohiam, Dune

Questo è uno degli aspetti più impegnativi del multithreading con C. Ma proveremo a prendercela comoda.

Fondamentalmente parlerò degli usi più diretti delle variabili atomiche, cosa sono e come funzionano, ecc. E menzionerò alcuni dei percorsi più follemente complessi a tua disposizione.

Ma non seguirò quelle strade. Non solo sono a malapena qualificato anche solo per scriverne ma immagino che se sai di averne bisogno ne sai già più di me.

Ma ci sono alcune cose strane qui fuori anche dalle basi. Quindi allacciate tutti le cinture di sicurezza, perché il Kansas sta per dire addio.

40.1. Test per il supporto atomico

Gli atomi sono una funzionalità opzionale. C'è una macro `__STDC_NO_ATOMICS__` che è `1` se *non* hai gli atomi.

Quella macro potrebbe non esistere prima di C11, quindi dovremmo testare la versione del linguaggio con `__STDC_VERSION__`[222] .

```
#if __STDC_VERSION__ < 201112L || __STDC_NO_ATOMICS__ == 1
#define HAS_ATOMICS 0
#else
#define HAS_ATOMICS 1
#endif
```

Se questi test vengono superati, puoi tranquillamente includerli `<stdatomic.h>`, l'intestazione su cui si basa il resto di questo capitolo. Ma se non esiste un supporto atomico, quell'intestazione potrebbe anche non esistere.

Su alcuni sistemi potrebbe essere necessario aggiungere `-latomic` alla fine della riga di comando di compilazione per utilizzare qualsiasi funzione nel file di intestazione.

40.2. Variabili atomiche

Ecco *parte* di come funzionano le variabili atomiche:

Se hai una variabile atomica condivisa e la scrivi da un thread quella scrittura sarà *tutto o niente* in un thread diverso.

Cioè l'altro thread vedrà l'intera scrittura, ad esempio di un valore a 32 bit. Nemmeno la metà. Non è possibile che un thread ne interrompa un altro che si trova nel *mezzo* di una scrittura atomica multibyte.

È quasi come se ci fosse un piccolo lucchetto attorno all'acquisizione e all'impostazione di quella variabile. (E *potrebbe* esserci! Vedere Variabili atomiche senza blocchi di seguito.)

E in quella nota puoi farla franca senza mai usare gli atomi se usi i mutex per bloccare le sezioni critiche. È solo che esiste una classe di *strutture dati prive di blocco* che consentono sempre ad altri thread di fare progressi invece di essere bloccati da un mutex… ma questi sono difficili da creare correttamente da zero e purtroppo sono una delle cose che vanno oltre lo scopo della guida.

222La macro `__STDC_VERSION__` non esisteva all'inizio del C89, quindi se sei preoccupato per questo, controllalo con `#ifdef`.

Questa è solo una parte della storia. Ma è la parte da cui inizieremo.

Prima di andare oltre, come si dichiara una variabile per essere atomica?

Innanzitutto includi `<stdatomic.h>`.

Questo ci dà tipi come `atomic_int`.

E poi possiamo semplicemente dichiarare che le variabili sono di quel tipo.

Ma facciamo una demo in cui abbiamo due thread. Il primo viene eseguito per un po' quindi imposta una variabile su un valore specifico poi esce. L'altro viene eseguito finché non vede il valore impostato e poi esce.

```
#include <stdio.h>
#include <threads.h>
#include <stdatomic.h>

atomic_int x;
// IL POTERE DELL'ATOMICA! BWHAHAHA!

int thread1(void *arg)
{
    (void)arg;

    printf("Thread 1: Sleeping for 1.5 seconds\n");
    thrd_sleep(&(struct timespec){.tv_sec=1, .tv_nsec=500000000}, NULL);

    printf("Thread 1: Setting x to 3490\n");
    x = 3490;

    printf("Thread 1: Exiting\n");
    return 0;
}

int thread2(void *arg)
{
    (void)arg;

    printf("Thread 2: Waiting for 3490\n");
    while (x != 3490) {}  // spin here

    printf("Thread 2: Got 3490--exiting!\n");
    return 0;
}

int main(void)
{
    x = 0;

    thrd_t t1, t2;

    thrd_create(&t1, thread1, NULL);
    thrd_create(&t2, thread2, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Main    : Threads are done, so x better be 3490\n");
    printf("Main    : And indeed, x == %d\n", x);
}
```

Il secondo thread gira sul posto guardando il flag e aspettando che venga impostato sul valore `3490`. E il primo fa così.

E ottengo questo risultato:

```
Thread 1: Sleeping for 1.5 seconds
Thread 2: Waiting for 3490
Thread 1: Setting x to 3490
Thread 1: Exiting
Thread 2: Got 3490--exiting!
Main    : Threads are done, so x better be 3490
Main    : And indeed, x == 3490
```

Guarda mamma! Stiamo accedendo a una variabile da thread diversi e non utilizziamo un mutex! E funzionerà sempre grazie alla natura atomica delle variabili atomiche.

Forse ti starai chiedendo cosa succede se si tratta di un regolare non atomico `int` invece. Bene, sul mio sistema funziona ancora… a meno che non esegua una build ottimizzata nel qual caso si blocca sul thread 2 in attesa di vedere il 3490 per essere impostato[223] .

Ma questo è solo l'inizio della storia. La parte successiva richiederà più potenza cerebrale e avrà a che fare con qualcosa chiamato *sincronizzazione*.

40.3. Sincronizzazione

La parte successiva della nostra storia riguarda il momento in cui determinati porzioni di memoria scritti in un thread diventano visibili a quelli in un altro thread.

Potresti pensare che sia subito vero? Ma non lo è. Molte cose possono andare storte. Stranamente male.

Il compilatore potrebbe aver riorganizzato gli accessi alla memoria in modo che quando pensi di impostare un valore relativo a un altro potrebbe non essere vero. E anche se il compilatore non lo avesse fatto la tua CPU avrebbe potuto farlo al volo. O forse c'è qualcos'altro in questa architettura che fa sì che le scritture su una CPU vengano ritardate prima che siano visibili su un'altra.

La buona notizia è che possiamo condensare tutti questi potenziali problemi in uno solo: gli accessi alla memoria non sincronizzati possono apparire fuori ordine a seconda del thread che sta osservando, come se le righe di codice stesse fossero state riorganizzate.

Ad esempio cosa avviene prima nel codice seguente scrivere su `x` o scrivere su `y`?

```
int x, y;  // global

// ...

x = 2;
y = 3;

printf("%d %d\n", x, y);
```

Risposta: non lo sappiamo. Il compilatore o la CPU potrebbero invertire silenziosamente le righe 5 e 6 e non ne sapremmo nulla. Il codice verrebbe eseguito a thread singolo *come se* fosse eseguito nell'ordine del codice.

In uno scenario multithread potremmo avere qualcosa di simile a questo pseudocodice:

223Il motivo è che una volta ottimizzato, il mio compilatore ha inserito il valore di x in un registro per rendere veloce il ciclo while. Ma il registro non ha modo di sapere che la variabile è stata aggiornata in un altro thread, quindi non vede mai il 3490. Questo non è realmente correlato alla parte tutto o niente dell'atomicità ma è più correlato agli aspetti di sincronizzazione in la sezione successiva.

```
int x = 0, y = 0;

thread1() {
    x = 2;
    y = 3;
}

thread2() {
    while (y != 3) {}  // spin
    printf("x is now %d\n", x);  // 2? ...or 0?
}
```

Qual è l'output del thread 2?

Bene, se a `x` viene assegnato `2` *prima* che a `y` venga assegnato `3` allora mi aspetto che l'output sia molto sensato:

```
x is now 2
```

Ma qualcosa di subdolo potrebbe riorganizzare le righe 4 e 5 facendoci vedere il valore `0` per `x` quando lo stampiamo.

In altre parole tutte le scommesse sono annullate a meno che non possiamo dirlo in qualche modo "A questo punto, mi aspetto che tutte le scritture precedenti in un altro thread siano visibili in questo thread."

Due thread si *sincronizzano* quando concordano sullo stato della memoria condivisa. Come abbiamo visto non sempre sono d'accordo con il codice. Allora come fanno ad essere d'accordo?

L'uso di variabili atomiche può forzare l'accordo[224] . Se un thread scrive su una variabile atomica dice "chiunque leggerà questa variabile atomica in futuro vedrà anche tutte le modifiche che ho apportato alla memoria (atomica o meno) fino alla variabile atomica inclusa".

Oppure in termini più umani, sediamoci attorno al tavolo delle conferenze e assicuriamoci di essere sulla stessa lunghezza d'onda riguardo a quali sono i pezzi di memoria condivisa e quali valori contengono. Accetti che le modifiche alla memoria apportate fino al deposito atomico saranno visibili a me dopo aver caricato la stessa variabile atomica.

Quindi possiamo facilmente correggere il nostro esempio:

```
int x = 0;
atomic int y = 0;  // Rendi y atomico

thread1() {
    x = 2;
    y = 3;             // Sincronizza in scrittura
}

thread2() {
    while (y != 3) {}  // Sincronizza in lettura
    printf("x is now %d\n", x);  // 2, period.
}
```

Perché i thread si sincronizzano attraverso `y`, tutte le scritture nel thread 1 avvenute *prima* della scrittura su `y` sono visibili nel thread 2 *dopo* la lettura da `y` (nel ciclo `while`).

È importante notare un paio di cose qui:

1. Niente dorme. La sincronizzazione non è un'operazione bloccante. Entrambi i thread

224Finché non dico diversamente sto parlando in generale di operazioni sequenzialmente coerenti. Maggiori informazioni su cosa significhi presto.

funzionano a pieno ritmo finché non escono. Anche quello bloccato nello spin loop non impedisce a nessun altro di scappare.

2. La sincronizzazione avviene quando un thread legge una variabile atomica scritta da un altro thread. Pertanto quando il thread 2 legge `y` tutta la memoria precedente scrive nel thread 1 (ovvero impostando `x`) che sarà visibile nel thread 2.
3. Nota che `x` non è atomico. Va bene perché non stiamo sincronizzando su `x` e la sincronizzazione su `y` quando lo scriviamo nel thread 1 significa che tutte le scritture precedenti—compreso `x~—nel thread 1 diventerà visibile agli altri thread… se gli altri thread leggono ~y` per la sincronizzazione.

Forzare questa sincronizzazione è inefficiente e può essere molto più lento rispetto usare semplicemente una variabile normale. Questo è il motivo per cui non usiamo l'atomica a meno che non sia necessario per un'applicazione particolare.

Quindi queste sono le basi. Diamo un'occhiata più in profondità.

40.4. Acquisisci e rilascia

Più terminologia! Ti ripagherà impararlo adesso.

Quando un thread legge una variabile atomica si dice che sia un'operazione di *acquisizione*.

Quando un thread scrive una variabile atomica si dice che sia un'operazione di *rilascio*.

Cosa sono questi? Allineiamoli con i termini che già conosci quando si tratta di variabili atomiche:

Read = Load = Acquire. Come quando confronti una variabile atomica o la leggi per copiarla su un altro valore.

Write = Store = Release. Come quando assegni un valore a una variabile atomica.

Quando si utilizzano variabili atomiche con queste semantiche di acquisizione/rilascio C specifica cosa può accadere e quando.

Acquisire/rilasciare costituisce la base per la sincronizzazione di cui abbiamo appena parlato.

Quando un thread acquisisce una variabile atomica può vedere i valori impostati in un altro thread che ha rilasciato la stessa variabile.

In altre parole:

Quando un thread legge una variabile atomica può vedere i valori impostati in un altro thread che ha scritto su quella stessa variabile.

La sincronizzazione avviene attraverso la coppia acquisizione/rilascio.

Più dettagli:

Con read/load/acquiredi una particolare variabile atomica:

- Tutte le scritture (atomico o non atomico) in un altro thread accaduto prima dell'altro thread wrote/stored/released questa variabile atomica è ora visibile in questo thread.
- Il nuovo valore della variabile atomica impostata dall'altro thread è visibile anche in questo thread.
- Nessuna lettura o scrittura di variabili/memoria nel thread corrente può essere riordinata in modo che avvenga prima di questa acquisizione.
- L'acquisizione funge da barriera unidirezionale quando si tratta di riordinare il codice; le

letture e le scritture nel thread corrente possono essere spostate da *prima* dell'acquisizione a *dopo*. Ma cosa ancora più importante per la sincronizzazione nulla può spostarsi da *dopo* l'acquisizione a *prima* di essa.

Con write/store/release di una particolare variabile atomica:

- Tutte le scritture (atomico o non atomico) nel thread corrente che si è verificato prima di questa versione diventa visibile ad altri thread che hanno read/loaded/acquired la stessa variabile atomica.
- Il valore scritto su questa variabile atomica da questo thread è visibile anche ad altri thread.
- Nessuna lettura o scrittura di alcuna variables/memory nel thread corrente può essere riordinato dopo questo rilascio.
- Il rilascio funge da barriera unidirezionale quando si tratta di riordinare il codice: le letture e le scritture nel thread corrente possono essere spostate da *dopo* il rilascio a *prima* di esso. Ma cosa ancora più importante per la sincronizzazione, nulla può spostarsi da *prima* del rilascio a *dopo*.

Ancora una volta il risultato è la sincronizzazione della memoria da un thread all'altro. Il secondo thread può essere sicuro che le variabili e la memoria siano scritte nell'ordine previsto dal programmatore.

```
int x, y, z = 0;
atomic_int a = 0;

thread1() {
    x = 10;
    y = 20;
    a = 999;  // Release
    z = 30;
}

thread2()
{
    while (a != 999) { } // Acquisire

    assert(x == 10);
// non afferma mai, x è sempre 10
    assert(y == 20);
// non afferma mai, y è sempre 20

    assert(z == 0);
// potrebbe affermare!!
}
```

Nell'esempio sopra `thread2` può essere sicuro dei valori in `x` e `y` dopo aver acquisito a perché sono stati impostati prima che `thread1` rilasciasse l'atomica `a`.

Ma `thread2` non può essere sicuro del valore di `z` perché è avvenuto dopo il rilascio. Forse l'assegnazione a `z` è stata spostata prima dell'assegnazione ad `a`.

Una nota importante: il rilascio di una variabile atomica non ha alcun effetto sull'acquisizione di diverse variabili atomiche. Ogni variabile è isolata dalle altre.

40.5. Coerenza sequenziale

Li lasci lì? Abbiamo superato il nocciolo dell'uso più semplice dell'atomica. E poiché qui non

parleremo nemmeno degli usi più complessi puoi rilassarti un po'.

La *coerenza sequenziale* è ciò che viene chiamato *ordinamento della memoria*. Esistono molti ordinamenti di memoria, ma la coerenza sequenziale è la cosa più sana che il [225] C ha da offrire. È anche l'impostazione predefinita. Devi fare di tutto per utilizzare altri ordinamenti di memoria.

Tutto ciò di cui abbiamo parlato finora è avvenuto nell'ambito della coerenza sequenziale.

Abbiamo parlato di come il compilatore o la CPU possono riorganizzare le letture e le scritture della memoria in un singolo thread purché segua la regola *"come se"*.

E abbiamo visto come possiamo frenare questo comportamento sincronizzandoci sulle variabili atomiche.

Formalizziamo ancora un po'.

Se le operazioni sono *sequenzialmente coerenti* significa che alla fine della giornata quando tutto è stato detto e fatto, tutti i thread si possono alzare in piedi, aprire la bevanda preferita e tutti concordare sull'ordine in cui si sono verificati i cambiamenti di memoria durante la sessione di esecuzione. E quell'ordine è quello specificato dal codice.

Uno non si dirà, “Ma *B* non è successo prima di *A*?” se lo dicono gli altri “ *A* è sicuramente successo prima di *B* ”. Sono tutti amici qua.

In particolare all'interno di un thread nessuno degli acquisiti e dei rilasci può essere riordinato l'uno rispetto all'altro. Ciò è in aggiunta alle regole su quali altri accessi alla memoria possono essere riordinati attorno ad essi.

Questa regola conferisce un ulteriore livello di sanità mentale alla progressione atomica loads/acquires e stores/releases.

Ogni altro ordine di memoria in C comporta un allentamento delle regole di riordino sia per acquisizioni/rilasci che per altri accessi alla memoria atomici o meno. Lo faresti se sapessi *davvero* cosa stai facendo e avessi bisogno di un aumento di velocità. *Ecco eserciti di draghi…*

Ne parleremo più avanti ma per ora atteniamoci alla sicurezza e alla praticità.

40.6. Assegnazioni e operatori atomici

Alcuni operatori sulle variabili atomiche sono atomici. E altri no.

Cominciamo con un controesempio:

```
atomic_int x = 0;

thread1() {
    x = x + 3;  // NON atomica!
}
```

Poiché c'è una lettura di `x` sul lato destro dell'assegnazione e una scrittura effettiva sulla sinistra queste sono due operazioni. Un altro thread potrebbe insinuarsi nel mezzo e renderti infelice.

Ma *puoi* usare la scorciatoia `+=` per ottenere un'operazione atomica:

```
atomic_int x = 0;

thread1() {
    x += 3;   // ATOMICO!
}
```

225Il più sano dal punto di vista del programmatore.

In tal caso, `x` verrà incrementato atomicamente di ~3~—nessun altro thread può saltare nel mezzo.

In particolare i seguenti operatori sono atomici read-modify-write operations con coerenza sequenziale, quindi usali con gioioso abbandono. (Nell'esempio, `a` è atomico.)

```
a++       a--       --a       ++a
a += b    a -= b    a *= b    a /= b    a %= b
a &= b    a |= b    a ^= b    a >>= b   a <<= b
```

40.7. Funzioni di libreria che si sincronizzano automaticamente

Finora abbiamo parlato di come sincronizzarsi con le variabili atomiche ma risulta che ci sono alcune funzioni di libreria che eseguono da sole una sincronizzazione limitata dietro le quinte.

```
call_once()       thrd_create()       thrd_join()
mtx_lock()        mtx_timedlock()     mtx_trylock()
malloc()          calloc()            realloc()
aligned_alloc()
```

`call_once()` —Si sincronizza con tutte le chiamate successive a `call_once()` per una bandiera particolare. In questo modo le chiamate successive possono essere certi che se un altro thread imposta il flag, lo vedranno.

`thrd_create()` —Si sincronizza con l'inizio del nuovo thread. Il nuovo thread può essere sicuro che vedrà tutte le scritture di memoria condivisa dal thread principale prima della chiamata `thrd_create()`.

`thrd_join()` —Quando un thread muore si sincronizza con questa funzione. Il thread che ha chiamato `thrd_join()` puoi essere certo che può vedere tutte le scritture condivise dell'ultimo thread.

`mtx_lock()` —Chiamate precedenti a `mtx_unlock()` sullo stesso mutex per sincronizzarsi su questa chiamata. Questo è il caso che rispecchia maggiormente il acquire/release processo di cui abbiamo già parlato. `mtx_unlock()` esegue un rilascio sulla variabile mutex assicurando qualsiasi thread successivo che effettui un'acquisizione con `mtx_lock()` può vedere tutte le modifiche alla memoria condivisa nella sezione critica.

`mtx_timedlock()` e `mtx_trylock()` —Simile alla situazione con `mtx_lock()` se questa chiamata ha esito positivo le chiamate precedenti a `mtx_unlock()` si sincronizzano con questo.

Dynamic Memory Functions: se allochi memoria si sincronizza con la precedente deallocazione di quella stessa memoria. Inoltre le allocazioni e deallocazioni di quella particolare regione di memoria avvengono in un unico ordine totale su cui tutti i thread possono concordare. *Penso* che l'idea qui sia che la deallocazione possa cancellare la regione se lo desidera e vogliamo essere sicuri che un'allocazione successiva non veda i dati non cancellati. Qualcuno mi faccia sapere se c'è altro.

40.8. Identificatore di tipo atomico, qualificatore

Abbassiamo il livello e vediamo quali tipi abbiamo a disposizione e come possiamo creare nuovi tipi atomici.

Per prima cosa diamo un'occhiata ai tipi atomici incorporati e a cosa sono `typedef`. (Spoiler: `_Atomic` è un qualificatore di tipo!)

Atomic type	Longhand equivalent

atomic_bool	_Atomic _Bool
atomic_char	_Atomic char
atomic_schar	_Atomic signed char
atomic_uchar	_Atomic unsigned char
atomic_short	_Atomic short
atomic_ushort	_Atomic unsigned short
atomic_int	_Atomic int
atomic_uint	_Atomic unsigned int
atomic_long	_Atomic long
atomic_ulong	_Atomic unsigned long
atomic_llong	_Atomic long long
atomic_ullong	_Atomic unsigned long long
atomic_char16_t	_Atomic char16_t
atomic_char32_t	_Atomic char32_t
atomic_wchar_t	_Atomic wchar_t
atomic_int_least8_t	_Atomic int_least8_t
atomic_uint_least8_t	_Atomic uint_least8_t
atomic_int_least16_t	_Atomic int_least16_t
atomic_uint_least16_t	_Atomic uint_least16_t
atomic_int_least32_t	_Atomic int_least32_t
atomic_uint_least32_t	_Atomic uint_least32_t
atomic_int_least64_t	_Atomic int_least64_t
atomic_uint_least64_t	_Atomic uint_least64_t
atomic_int_fast8_t	_Atomic int_fast8_t

`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

Usali a piacimento! Sono coerenti con gli alias atomici trovati in C++ se questo aiuta.

E se volessi di più?

Puoi farlo con un qualificatore di tipo o uno specificatore di tipo.

Innanzitutto lo specificatore! È la parola chiave `_Atomic` con un tipo tra parentesi dopo[226] — adatto per l'uso con `typedef`:

```
typedef _Atomic(double) atomic_double;

atomic_double f;
```

Restrizioni sullo specificatore: il tipo che stai rendendo atomico non può essere di tipo array o funzione, né può essere atomico o altrimenti qualificato.

Prossimo qualificatore! È la parola chiave `_Atomic` *senza* tipo tra parentesi.

Quindi questi fanno cose simili[227] :

```
_Atomic(int) i;   // identificatore di tipo
_Atomic int  j;   // qualificatore di tipo
```

Il fatto è che puoi includere altri qualificatori di tipo con quest'ultimo:

```
_Atomic volatile int k;   // qualified atomic variable
```

226Apparentemente C++23 lo sta aggiungendo come macro.
227Le specifiche rilevano che potrebbero differire per dimensioni, rappresentazione e allineamento.

Restrizioni sulla qualificazione: il tipo che stai rendendo atomico non può essere di tipo array o funzione.

40.9. Variabili atomiche prive di lock

Le architetture hardware sono limitate nella quantità di dati che possono leggere e scrivere atomicamente. Dipende da come è collegato insieme. E varia.

Se utilizzi un tipo atomico puoi essere certo che gli accessi a quel tipo saranno atomici… ma c'è un problema: se l'hardware non può farlo viene invece eseguito con un blocco.

Quindi l'accesso atomico diventa lock-access-unlock che è molto più lento e ha alcune implicazioni per i gestori di segnale.

I flag atomici di seguito sono l'unico tipo atomico che è garantito essere privo di blocchi in tutte le implementazioni conformi. Nel tipico mondo dei computer desktop/laptop altri tipi più grandi sono probabilmente privi di blocchi.

Fortunatamente abbiamo un paio di modi per determinare se un particolare tipo è un atomico senza lock o meno.

Prima di tutto alcune macro—puoi usarli in fase di compilazione con `#if`. Si applicano sia ai tipi signed che a quelli unsigned.

Atomic Type	Lock Free Macro
`atomic_bool`	`ATOMIC_BOOL_LOCK_FREE`
`atomic_char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_char16_t`	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	`ATOMIC_LLONG_LOCK_FREE`
`atomic_intptr_t`	`ATOMIC_POINTER_LOCK_FREE`

È interessante notare che queste macro possono avere *tre* valori diversi:

Valore	Significato

`0`	Never lock-free.
`1`	Sometimes lock-free.
`2`	Always lock-free.

Aspetta—come può qualcosa essere *a volte* senza serratura? Ciò significa semplicemente che la risposta non è nota in fase di compilazione ma potrebbe essere nota in seguito in fase di esecuzione. Forse la risposta varia a seconda che tu stia eseguendo o meno questo codice su Intel o AMD originali o qualcosa del genere[228] .

Ma puoi sempre testare in fase di esecuzione con la funzione `atomic_is_lock_free()`. Questa funzione restituisce vero o falso se il tipo particolare è atomico in questo momento.

Allora perché ci preoccupiamo?

Senza blocco è più veloce quindi forse c'è un problema di velocità che potresti codificare in un altro modo. O forse devi utilizzare una variabile atomica in un gestore di segnale.

40.9.1. Gestori di segnale e atomi senza lock

Se leggi o scrivi una variabile condivisa (durata della memorizzazione statica o `_Thread_Local`) in un gestore di segnali, è un comportamento indefinito [gasp!]… A meno che non si effettui una delle seguenti operazioni:

1. Scrivere in una variabile di tipo `volatile sig_atomic_t`.
2. Leggere o scrivere una variabile atomica senza lock.

Per quanto ne so le variabili atomiche prive di lock sono uno dei pochi modi per ottenere informazioni in modo portabile da un gestore di segnali.

Le specifiche sono un po' vaghe a quanto ho letto sull'ordine della memoria quando si tratta di acquisire o rilasciare variabili atomiche nel gestore del segnale. Il C++ dice, ed è logico, che tali accessi non sono sequenziati rispetto al resto del programma[229] . Dopotutto il segnale può essere avviato in qualsiasi momento. Quindi presumo che il comportamento di C sia simile.

40.10. Bandiere atomiche

C'è solo un tipo che lo standard garantisce sarà un atomico senza blocco: `atomic_flag`. Questo è un tipo opaco per le operazioni test-and-set[230] .

Può essere emtrami *impostato* o *cancellato*. Puoi inizializzarlo per cancellarlo:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

Puoi impostare la bandiera in modo atomico con `atomic_flag_test_and_set()` che imposterà il flag e restituirà il suo stato precedente come un `_Bool` (vero per l'insieme).

Puoi eliminare atomicamente la flag con atomic$_{\text{flagclear}}$().

Ecco un esempio in cui inizializziamo il flag per cancellarlo lo impostiamo due volte, quindi lo cancelliamo di nuovo.

228Ho appena tirato fuori quell'esempio dal nulla. Forse non ha importanza su Intel/AMD ma potrebbe avere importanza da qualche parte dannazione!

229Il C++ spiega che se il segnale è il risultato di una chiamata a `raise()` viene sequenziato dopo `raise()`.

230https://en.wikipedia.org/wiki/Test-and-set

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdatomic.h>

atomic_flag f = ATOMIC_FLAG_INIT;

int main(void)
{
    bool r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0

    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 1

    atomic_flag_clear(&f);
    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0
}
```

40.11. `struct` e `union` Atomiche

Utilizzando il qualificatore o specificatore `_Atomic` puoi renderlo atomico `struct` o `union`! Abbastanza sorprendente.

Se non ci sono molti dati lì dentro (es. una manciata di byte) il tipo atomico risultante potrebbe essere privo di blocchi. Provalo con `atomic_is_lock_free()`.

```c
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;

    printf("Is lock free: %d\n", atomic_is_lock_free(&p));
}
```

Ecco il problema: non è possibile accedere ai campi di un atomico `struct` o `union`~… `quindi qual è il punto? Bene, puoi /copiare/ atomicamente l'intero ~struct` in una variabile non atomica e poi usarla. Puoi copiare atomicamente anche nell'altro modo.

```c
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;
    struct point t;

    p = (struct point){1, 2};  // Copia atomica
```

```
    //printf("%f\n", p.x);  // Errore

    t = p;   // Atomic copy

    printf("%f\n", t.x);  // OK!
}
```

Puoi anche dichiarare una `struct` in cui i singoli campi sono atomici. L'implementazione è definita se i tipi atomici sono consentiti sui bitfield.

40.12. Puntatori atomici

Solo una nota qui sul posizionamento di `_Atomic` quando si tratta di puntatori.

Innanzitutto i puntatori agli atomi (es. il valore del puntatore non è atomico ma la cosa a cui punta lo è):

#+BEGIN$_{SRC}$ C _Atomic int x; _Atomic int *p; // p è un puntatore a un atomic int

p = &x; // OK! #+END$_{SRC}$

In secondo luogo puntatori atomici a valori non atomici (es. il valore del puntatore stesso è atomico ma l'oggetto a cui punta non lo è):

```
int x;
int * _Atomic p;  // p è un puntatore atomico a un int

p = &x;  // OK!
```

Infine puntatori atomici a valori atomici (es. il puntatore e l'oggetto a cui punta sono entrambi atomici):

```
_Atomic int x;
_Atomic int * _Atomic p;  // p è un puntatore atomico a an atomic int

p = &x;  // OK!
```

40.13. Ordine della memoria

Abbiamo già parlato della coerenza sequenziale che è quella sensata del gruppo. Ma ce ne sono molti altri:

`memory_order`	Description
`memory_order_seq_cst`	Sequential Consistency
`memory_order_acq_rel`	Acquire/Release
`memory_order_release`	Release
`memory_order_acquire`	Acquire
`memory_order_consume`	Consume
`memory_order_relaxed`	Relaxed

È possibile specificarne altri con determinate funzioni di libreria. Ad esempio puoi aggiungere un valore a una variabile atomica come questa:

```
atomic_int x = 0;

x += 5;  // Coerenza sequenziale l'impostazione predefinita
```

Oppure puoi fare lo stesso con questa funzione di libreria:

```
atomic_int x = 0;

atomic_fetch_add(&x, 5);  // Coerenza sequenziale l'impostazione predefinita
```

Oppure puoi fare la stessa cosa con un ordinamento esplicito della memoria:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_seq_cst);
```

Ma cosa succederebbe se non volessimo la coerenza sequenziale? E tu volevi acquire/release invece per qualche motivo? Basta nominarlo:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_acq_rel);
```

Faremo una ripartizione dei diversi ordini di memoria di seguito. Non scherzare con nient'altro che la coerenza sequenziale a meno che tu non sappia cosa stai facendo. È davvero facile commettere errori che causeranno guasti rari e difficili da riprodurre.

40.13.1. Coerenza sequenziale

- Acquisizione delle operazioni di caricamento (vedi sotto).
- Rilascio delle operazioni di immagazzinamento (vedi sotto).
- Read-modify-write le operazioni acquisiscono e poi rilasciano.

Inoltre al fine di mantenere l'ordine totale di acquisizioni e rilasci nessuna acquisizione o rilascio verrà riordinato l'uno rispetto all'altro. (Il acquire/release le regole non vietano di riordinare un rilascio seguito da un'acquisizione. Ma le regole sequenzialmente coerenti lo fanno.)

40.13.2. Acquisire

Questo è ciò che accade su una operazione load/read su una variabile atomica.

- Se un altro thread ha rilasciato questa variabile atomica tutte le scritture effettuate da quel thread sono ora visibili in questo thread.
- Gli accessi alla memoria in questo thread che si verificano dopo questo caricamento non possono essere riordinati prima.

40.13.3. Rilascio

Questo è ciò che accade su un store/write di una variabile atomica.

- Se un altro thread successivamente acquisisce questa variabile atomica, tutta la memoria scrive in questo thread prima che la sua scrittura atomica diventi visibile a quell'altro thread.
- Gli accessi alla memoria in questo thread che si verificano prima del rilascio non possono

essere riordinati dopo di esso.

40.13.4. Consuma

Questo è strano simile a una versione meno rigorosa dell'acquisizione. Ha effetto sugli accessi alla memoria che *dipendono dai dati* dalla variabile atomica.

Essere “dipendente dai dati” significa vagamente che la variabile atomica viene utilizzata in un calcolo.

Cioè se un thread consuma una variabile atomica, tutte le operazioni in quel thread che continuano a utilizzare quella variabile atomica saranno in grado di vedere le scritture in memoria nel thread di rilascio.

Confronta per acquisire dove la memoria scrive nel thread di rilascio sarà visibile a *tutte* le operazioni nel thread corrente, non solo a quelle dipendenti dai dati.

Inoltre come l'acquisizione esiste una restrizione su quali operazioni possono essere riordinate *prima* del consumo. Con l'acquisizione non è possibile riordinare nulla prima. Con consuma non puoi riordinare nulla che dipenda dal valore atomico caricato prima di esso.

40.13.5. Acquire/Release

Questo vale solo per operazioni read-modify-write. È un'acquisizione e un rilascio raggruppati in uno solo.

- Un'acquisizione avviene per la lettura.
- Per la scrittura avviene un rilascio.

40.13.6. Relaxed

Niente regole; è anarchia! Tutti possono riordinare tutto ovunque! Cani e gatti che vivono insieme —isteria di massa!

In realtà esiste una regola. Le letture e le scritture atomiche sono ancora tutto o niente. Ma le operazioni possono essere riordinate in modo stravagante e non c'è sincronizzazione tra i thread.

Esistono alcuni casi d'uso per questo ordine di memoria che puoi trovare con una piccola ricerca, es. contatori semplici.

E puoi usare una recinzione per forzare la sincronizzazione dopo una serie di scritture rilassate.

40.14. Recinzioni

Sai come avvengono i rilasci e le acquisizioni delle variabili atomiche mentre le leggi e le scrivi?

Bene, è possibile effettuare un rilascio o un'acquisizione anche *senza* una variabile atomica.

Questo si chiama *recinto*. Quindi se vuoi che tutte le scritture in un thread siano visibili altrove puoi creare una recinzione di rilascio in un thread e una recinzione di acquisizione in un altro proprio come funzionano le variabili atomiche.

Dal momento che un'operazione di consumo non ha davvero senso in un recinto[231] , `memory_order_consume` viene trattato come un'acquisizione.

Puoi montare una recinzione con qualsiasi ordine specificato:

231Perché il consumo riguarda le operazioni che dipendono dal valore della variabile atomica acquisita e non esiste una variabile atomica con una recinzione.

```
atomic_thread_fence(memory_order_release);
```

Esiste anche una versione leggera di una recinzione da utilizzare con i gestori di segnali chiamata `atomic_signal_fence()`.

Funziona proprio allo stesso modo di `atomic_thread_fence()` tranne che:

- Si occupa solo della visibilità dei valori all'interno dello stesso thread; non c'è sincronizzazione con altri thread.
- Non viene emessa alcuna istruzione di recinzione hardware.

Se vuoi essere sicuro degli effetti collaterali delle operazioni non atomiche (e operazioni atomiche rilassate) sono visibili nel gestore del segnale, puoi utilizzare questa recinzione.

L'idea è che il gestore del segnale sia in esecuzione in *questo* thread, non un altro, quindi questo è un modo più leggero per assicurarsi che le modifiche all'esterno del gestore del segnale siano visibili al suo interno (es. non sono stati riordinati).

40.15. Referenze

Se vuoi saperne di più su queste cose, ecco alcune delle cose che mi hanno aiutato ad affrontarle: Herb Sutter's atomic<> Weapons talk: Part 1[232] part 2[233] Jeff Preshing's materials[234] , in particolare: An Introduction to Lock-Free Programming[235] Acquire and Release Semantics[236] The Happens-Before Relation[237] The Synchronizes-With Relation[238] The Purpose of $memory_{orderconsume}$ in C++11[239] You Can Do Any Kind of Atomic Read-Modify-Write Operation [241] CPPReference: Memory Order[242] Atomic Types[243] Bruce Dawson's Lockless Programming Considerations[244] Le persone disponibili e competenti su $r/C_{Programming}$[245]

41. Specificatori di funzioni, Specificatori/Operatori di allineamento

Nella mia esperienza personale non li ho trovati utili ma li tratteremo qui per completezza.

41.1. Specificatori di funzioni

Quando dichiari una funzione puoi dare al compilatore un paio di suggerimenti su come le funzioni potrebbero o saranno utilizzate. Ciò consente o incoraggia il compilatore a apportare determinate ottimizzazioni.

41.1.1. `inline` per la velocità:Forse

Puoi dichiarare una funzione inline in questo modo:

232https://www.youtube.com/watch?v=A8eCGOqgvH4
233https://www.youtube.com/watch?v=KeLBd2EJLOU
234https://preshing.com/archives/
235https://preshing.com/20120612/an-introduction-to-lock-free-programming/
236https://preshing.com/20120913/acquisire-and-release-semantics/
237https://preshing.com/20130702/the-happens-before-relation/
238https://preshing.com/20130823/the-synchronizes-with-relation/
239https://preshing.com/20140709/the-scopo-of-memory_order_consume-in-cpp11/
240https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operazione/
241https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operazione/
242https://en.cppreference.com/w/c/atomic/memory_order
243https://en.cppreference.com/w/c/lingual/atomic
244https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming
245https://www.reddit.com/r/C_Programming/↩

```
static inline int add(int x, int y) {
    return x + y;
}
```

Questo ha lo scopo di incoraggiare il compilatore a effettuare questa chiamata di funzione il più velocemente possibile. E storicamente un modo per farlo era l' `inlining`, significa che il corpo della funzione sarebbe stato incorporato nella sua interezza nel punto in cui veniva effettuata la chiamata. Ciò eviterebbe il carico derivante dall'impostazione della chiamata di funzione e dalla sua eliminazione al posto di avere una maggiore quantità di codice poiché la funzione veniva copiata ovunque invece di essere riutilizzata.

Le cose veloci e sporche da ricordare sono:

1. Probabilmente non è necessario utilizzare `inline` per la velocità. I compilatori moderni sanno cosa è meglio.
2. Se lo usi per la velocità usalo con l'ambito file cioè `static inline`. Ciò evita le regole disordinate del collegamento esterno e delle funzioni in linea.

Smetti di leggere questa sezione adesso.

Voglioso di essere punito eh?

Proviamo a lasciare disattivata `static`.

```
#include <stdio.h>

inline int add(int x, int y)
{
    return x + y;
}

int main(void)
{
    printf("%d\n", add(1, 2));
}
```

`gcc` ci dà un errore del linker su `add()`[246] a meno che non si compili con le ottimizzazioni attivate (probabilmente)!

Vedi, un compilatore può scegliere di incorporarlo o meno ma se sceglie di non farlo rimarrai senza alcuna funzione. `gcc` non è incorpora a meno che tu non stia eseguendo una build ottimizzata.

Un modo per aggirare questo problema è definire altrove una versione di collegamento esterno non `inline` della funzione e quella verrà utilizzata quando quella `inline` non lo è. Ma tu come programmatore non puoi determinare quale non è portabile. Se sono disponibili entrambi non è specificato quale sceglierà il compilatore. Con `gcc` verrà utilizzata la funzione inline se stai compilando con ottimizzazioni mentre quella non inline verrà utilizzata altrimenti. Anche se i corpi di queste funzioni sono completamente diversi. Pazzesco!

Un altro modo è dichiarare la funzione come `extern inline`. Questo tenterà di incorporarsi in questo file ma creerà anche una versione con collegamento esterno. E quindi gcc utilizzerà l'uno o l'altro a seconda delle ottimizzazioni ma almeno hanno la stessa funzione.

41.1.2. `noreturn` e `_Noreturn`

Ciò indica al compilatore che una particolare funzione non tornerà mai al suo chiamante, cioè il

246A meno che tu non compili con le ottimizzazioni attive (probabilmente)! Ma penso che quando lo fa, non si comporta secondo le specifiche.

programma uscirà con qualche meccanismo prima che la funzione ritorni.

Consente al compilatore di eseguire eventualmente alcune ottimizzazioni attorno alla chiamata di funzione.

Consente inoltre di indicare ad altri sviluppatori che parte della logica del programma dipende dalla mancata restituzione di una funzione.

Probabilmente non avrai mai bisogno di usarlo ma lo vedrai in alcune chiamate di libreria come `exit()` [247] e `abort()` [248] .

La parola chiave incorporata è `_Noreturn` ma se non interrompe il codice esistente tutti consiglierebbero di includere `<stdnoreturn.h>` e di utilizzare invece il `noreturn` più facile da leggere.

È un comportamento indefinito se una funzione specificata come `noreturn` restituisce effettivamente un risultato. È computazionalmente disonesto, vedi.

Ecco un esempio di utilizzo corretto di `noreturn`:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void foo(void)
// Questa funzione non dovrebbe mai ritornare!
{
    printf("Happy days\n");

    exit(1);
// E non ritorna: esce da qui!
}

int main(void)
{
    foo();
}
```

Se il compilatore rileva che una funzione potrebbe restituire `noreturn` ti avviserà.

Sostituendo la funzione `foo()` con questa:

```
noreturn void foo(void)
{
    printf("Breakin' the law\n");
}
```

mi dà un avviso:

```
foo.c:7:1: warning: function declared 'noreturn' should not return
```

41.2. Specificatori e operatori di allineamento

L'*allineamento* [249] riguarda i multipli di indirizzi su cui gli oggetti possono memorizzare. Puoi immagazzinarli a qualsiasi indirizzo? Oppure deve essere un indirizzo di partenza divisibile per 2? O 8? O 16?

Se stai codificando a basso livello come un allocatore di memoria che si interfaccia con il tuo

247https://beej.us/guide/bgclr/html/split/stdlib.html#man-exit
248https://beej.us/guide/bgclr/html/split/stdlib.html#man-abort
249https://en.wikipedia.org/wiki/Data_structure_alignment

sistema operativo potresti doverlo tenere presente. La maggior parte degli sviluppatori prosegue la propria carriera senza utilizzare questa funzionalità in C.

41.2.1. `alignas` e `_Alignas`

Questa non è una funzione. Piuttosto è uno *specificatore di allineamento* che puoi utilizzare con una dichiarazione di variabile.

L'identificatore integrato è `_Alignas` ma l'intestazione `<stdalign.h>` lo definisce come `alignas` per visualizzarlo in un aspetto migliore.

Se hai bisogno che il tuo `char` sia allineato come un `int` puoi forzarlo in questo modo quando lo dichiari:

```
char alignas(int) c;
```

Puoi anche passare un valore costante o un'espressione per l'allineamento. I valori che scegli devono essere supportati dal sistema ma le specifiche non stabiliscono quali valori puoi inserire. Le piccole potenze di 2 (1, 2, 4, 8 e 16) sono generalmente valori su cui puoi fare affidamento.

```
char alignas(8) c;
// allineare su limiti di 8 byte
```

Se vuoi allinearti al massimo che puoi nel tuo sistema includi `<stddef.h>` e usa il tipo `max_align_t` in questo modo:

```
char alignas(max_align_t) c;
```

Potresti potenzialmente *sovraallineare* specificando un allineamento maggiore di quello di `max_align_t` ma il fatto che tali cose siano consentite o meno dipende dal sistema.

41.2.2. `alignof` e `_Alignof`

Questo operatore restituirà l'indirizzo multiplo utilizzato da un particolare tipo per l'allineamento su questo sistema. Ad esempio forse i `char` sono allineati ogni +1 indirizzo e gli `int` sono allineati ogni +4 indirizzi.

L'operatore integrato è `_Alignof`, ma l'intestazione `<stdalign.h>` lo definisce come `alignof` se vuoi sembrare più figo.

Ecco un programma che stamperà gli allineamenti di una varietà di tipi diversi. Ancora una volta, questi varieranno da sistema a sistema. Tieni presente che il tipo `max_align_t` ti fornirà l'allineamento massimo utilizzato dal sistema.

```
#include <stdalign.h>
#include <stdio.h>     // per printf()
#include <stddef.h>    // per max_align_t

struct t {
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char       : %zu\n", alignof(char));
    printf("short      : %zu\n", alignof(short));
    printf("int        : %zu\n", alignof(int));
```

```
    printf("long       : %zu\n", alignof(long));
    printf("long long  : %zu\n", alignof(long long));
    printf("double     : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
    printf("struct t   : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Output sul mio sistema:

```
char       : 1
short      : 2
int        : 4
long       : 8
long long  : 8
double     : 8
long double: 16
struct t   : 16
max_align_t: 16
```

41.3. Funzione `memalignment()`

Novità in C23!

(Avvertenza: nessuno dei miei compilatori supporta ancora questa funzione, quindi il codice è in gran parte non testato.)

`alignof` è ottimo se conosci il tipo dei tuoi dati. Ma cosa succede se sei del *tutto ignorante* del tipo e hai solo un puntatore ai dati?

Come è potuto succedere?

Beh, con il nostro buon amico il `void*`, ovviamente. Non possiamo passarlo a `alignof` ma cosa succede se abbiamo bisogno di conoscere l'allineamento dell'oggetto a cui punta?

Potremmo volerlo sapere se stiamo per utilizzare la memoria per qualcosa che ha esigenze di allineamento significative. Ad esempio i tipi atomici e fluttuanti spesso si comportano male se disallineati.

Quindi con questa funzione possiamo verificare l'allineamento di alcuni dati purché abbiamo un puntatore a quei dati anche se è un `void*`.

Facciamo un rapido test per vedere se un puntatore void è ben allineato per l'uso come tipo atomico e in tal caso otteniamo una variabile per usarlo come quel tipo:

```
void foo(void *p)
{
    if (memalignment(p) >= alignof(atomic int)) {
        atomic int *i = p;
        do_things(i);
    } else
        puts("This pointer is no good as an atomic int\n");

...
```

Sospetto che raramente (probabilmente mai) avrai bisogno di utilizzare questa funzione a meno che tu non stia facendo cose di basso livello.

E il gioco è fatto. Allineamento!