Indice

1. I calcolatori

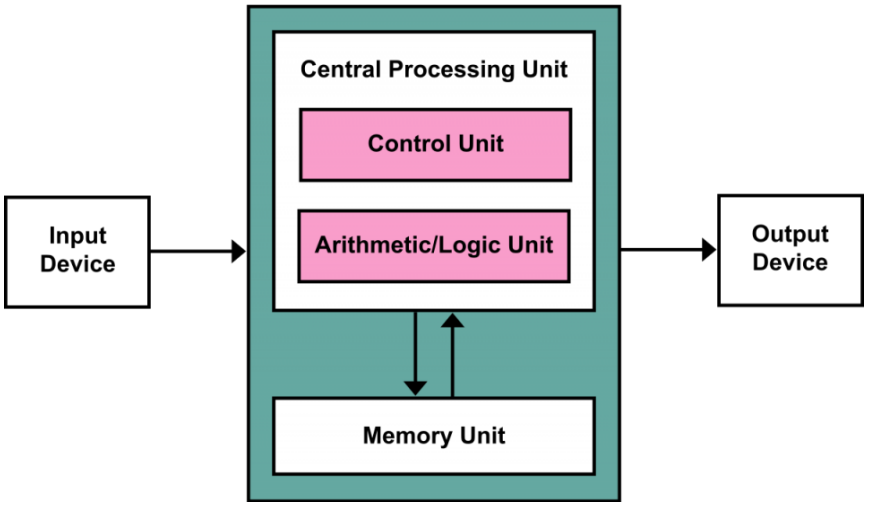
Un calcolatore elettronico è un dispositivo capace di elaborare dati. Questi dispositivi sono fondamentali in informatica, in quanto essa è la scienza della rappresentazione e dell'elaborazione delle informazioni.

I calcolatori, nella fattispecie, sono usati per risolvere problemi a partire da dei dati in ingresso (Input), poi restituendo il risultato (Output). Il processo risolutivo di un problema è il "programma". Perciò, i calcolatori devono essere capaci di comunicare con l'esterno, di eseguire sequenze di operazioni basilari e di memorizzare dati e programmi.

Ogni calcolatore è costituito da hardware e software.

L'architettura di Von Neumann - Hardware

Si tratta di una schematizzazione dei componenti e del funzionamento di un elaboratore elettronico.



La CPU (Central Processing Unit) opera sull'unità di memoria, in cui sono localizzati dati e programmi, e attua effettivamente ogni operazione degli ultimi. L'input e l'output sono gestiti da apposite periferiche, e il tutto è unito da un collegamento: il BUS (oggi suddiviso in più connessioni).

L'unità di memoria comprende la RAM (Random Access Memory) e la ROM (Read Only Memory).

La prima è volatile (è eliminata allo spegnimento della macchina) e permette sia lettura che scrittura ed è quella usata dalla CPU per compiere tutte le operazioni. La seconda è persistente e (di solito) permette soltanto la lettura; contiene istruzioni e dati fondamentali per il calcolatore, come quelle dell'avvio del sistema (BOOT).

A livello logico, la memoria è una serie di celle dalla capienza fissa, identificate da un numero, detto indirizzo. L'unità di misura delle informazioni è il bit: una singola cifra binaria. 8 bit formano un byte, del quale poi esistono relativi multipli, nei cui si cambia suffisso ogni 2^{10} elementi, e non ogni 10^3.

Alla memoria centrale, oggi, sono spesso aggiunti dispositivi di memorizzazione, che permettono di salvare dati attraverso varie modalità. La maggior parte di essi permette l'accesso diretto (si possono scegliere i dati da vedere immediatamente) e sia operazioni di scrittura che lettura. Questi dispositivi differiscono in base al tempo di accesso ai dati, alla velocità del loro trasferimento, alla tecnologia utilizzata e al prezzo.

La CPU esegue operazioni secondo il ritmo scandito da un "metronomo", il clock. I comandi eseguiti sono messi in atto dalla ALU (Arithmetic-Logic Unit), e sono di varia natura:

- Confronto tra valori;
- Operazioni aritmetiche tra valori;
- Modifica del valore del PC;
- Spostamento di dati.

Assieme alla ALU, si ha la CU (Control Unit), che ha vari componenti. L'IR (Instruction Register) contiene l'istruzione in esecuzione, mentre il PC (Program Counter) contiene l'indirizzo della prossima istruzione da eseguire. Le istruzioni dell'IR sono decodificate dall'IDC (Instruction Decoder Circuit) per far sì che l'ALU le "capisca". Queste entità sono responsabili del ciclo Fetch, Decode, Execute (Recupera, Decodifica, Esegui).

Le istruzioni eseguibili dall'ALU dipendono dal suo IS (Instruction Set). Esistono i CISC (Complete IS Computing), con istruzioni avanzate, e i RISC (Reduced ISC), con meno istruzioni, ma costi minori e velocità maggiore rispetto ai CISC.

Oltre che agli elementi sopracitati, la CPU comprende piccole unità di memoria, i registri. Tra di essi figurano registri dedicati a risultati intermedi dei programmi (AX, BX, …) e il MAR (Memory Address Register) e il MDR (M Data R), atti a contenere dati e indirizzi sia in entrata che in uscita.

Software - Macchina astratta

Sono tutti i programmi eseguibili su un computer. Essi possono essere Applicazioni, che svolgono determinate funzioni, o software di sistema (o sistemi operativi, SO).

Gli ultimi sono in costante esecuzione dall'avvio allo spegnimento della macchina (e dunque devono essere molto efficienti) e svolgono mansioni importanti:

- Interfacciare hardware e diversi software mediante collegamenti, periferiche e driver;
- Gestire i profili utente (e relativi privilegi e GUI) e le periferiche I/O;
- Gestire il file system (struttura organizzativa dei file) e il suo uso da parte di applicazioni;
- Gestire la memoria e l'esecuzione dei processi.

Un processo è un programma caricato in memoria di lavoro, dove ha accesso a tutte le risorse necessarie per la sua esecuzione. I diritti allo spazio in memoria e alle altre risorse sono conferiti e revocati dal SO.

Quando si utilizza un computer, si interagisce con una macchina astratta generata dal SO o da un applicativo. Si tratta di un modello teorico dell'hardware e del software del dispositivo effettivo e delle sue funzionalità, utilizzato come interfaccia per interagire con il vero elaboratore.

2. Algoritmi e programmi

Un algoritmo è una sequenza finita e non ambigua di semplici passi ordinati che porta alla soluzione di una classe di problemi in un tempo finito. In sostanza, sono le istruzioni di un metodo risolutivo.

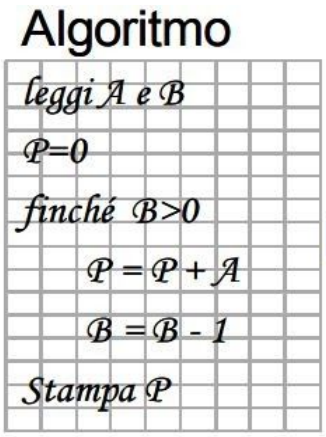
Dunque, l'algoritmo è:

- Eseguibile, ogni passo può essere eseguito in un tempo finito;
- Non-ambiguo, ogni passo può essere interpretato in un solo modo;
- Finito, i passi non devono essere infiniti (evitare LOOP).

Per mettere in atto un algoritmo, lo si deve affidare ad un esecutore capace di interpretare ed attuare.

Molti problemi, infine, hanno diversi algoritmi risolutivi; in questo caso si parla di algoritmi equivalenti. Per sceglierne uno, spesso se ne valuta l'efficienza, ossia la brevità della sua attuazione.

Es. di algoritmo:



Linguaggi di programmazione

Volendo far eseguire un algoritmo ad un calcolatore, è necessario scrivere un programma. Si tratta della trasposizione testuale di un algoritmo in un linguaggio di programmazione.

Un linguaggio di programmazione è una notazione definita formalmente in modo approfondito, usata per impartire comandi a calcolatori. Esso ha regole di sintassi (scrittura dei comandi) e di semantica (attribuzione di significato ai comandi). La prima è definita attraverso diagrammi o notazioni, mentre la seconda si può esprimere tramite azioni, logica o funzioni matematiche.

Questo è un esempio di sintassi nel linguaggio sintattico EBNF (Extended Backus-Naur Form). ::= significa "si definisce come", le <...> indicano un elemento definito, le {...} un elemento presente da 0 a ∞ volte e | significa "oppure".

```
<numero-naturale> ::= 0 | <cifra-non-nulla>{<cifra>}
<cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<cifra> ::= 0 | <cifra-non-nulla>
```

La CPU “capisce” solamente il linguaggio macchina, formato esclusivamente da stringhe di cifre binarie, ovviamente assai scomodo da utilizzare. È per questo che esistono i linguaggi di programmazione; si distinguono in base alla loro vicinanza al linguaggio macchina.

I linguaggi a basso livello (es. Assembly) sono vicini al linguaggio macchina. Hanno comandi basilari e specifici, e una sintassi complessa, talvolta legata all’hardware della macchina che li esegue. I linguaggi ad alto livello (es. C, Java) sono distanti dal linguaggio macchina. Sono indipendenti dall’hardware e hanno istruzioni complesse, con una sintassi vicina ai termini e alla logica umana.

Nell’ambito dei linguaggi ad alto livello, inoltre, si distinguono gli “stili” di risoluzione dei problemi, detti Paradigmi. I principali sono:

- Imperativo (es. C), vicino alla logica della CPU, prevede solo comandi diretti;
- Ad oggetti (es. Java), prevede la creazione di entità software contenenti dati e comandi;
- Funzionale, si basa sulle funzioni;
- Logico, utilizzato nelle IA.

Creazione ed esecuzione di programmi

Un qualsiasi programma è scritto mediante un editor di testo e salvato in ciò che prende il nome di “file sorgente”. Per eseguire il programma è necessario tradurlo in linguaggio macchina. Questa operazione può essere compiuta da un compilatore o da un interprete.

Un interprete traduce il file sorgente in linguaggio macchina, mandando in esecuzione ogni comando non appena lo traduce. In caso di cicli o iterazioni, l’interprete potrebbe trovarsi a tradurre la stessa istruzione più volte. Inoltre, interpretando un programma, diventa necessario avere l’interprete per eseguirlo. Per queste ragioni, usare un interprete può risultare scomodo e non ottimale.

Un compilatore, invece, traduce il file sorgente e pone il risultato finale in un “file compilato” (o “file oggetto”). Le istruzioni sono compilate una sola volta e l’esecuzione è lanciata dal file oggetto, già tradotto. Sebbene la compilazione possa risultare più lunga dell’interpretazione, essa non è inefficiente e permette di lanciare il programma mediante il file oggetto anche se non si ha il compilatore su un dispositivo.

Per programmi molto grandi, diventa scomodo creare un unico file oggetto. Il programma, dunque, è diviso in parti (dette “moduli”), compilate separatamente. I file oggetto di ogni modulo sono uniti da software detto Linker, che, inoltre, può aggiungere ad essi librerie (risorse software, solitamente codice) che supportano il funzionamento dell’applicazione.

Il risultato finale è un’applicazione che può essere eseguita direttamente dal SO.

3. Linguaggio C

Concepito nel ‘72, è un linguaggio di programmazione ad alto livello concepito per essere portabile, efficiente e sintetico, per via della sua intesa applicazione come linguaggio di sistema (si può usare per creare applicativi e SO).

Le sue istruzioni sono strutturate a blocchi (con un solo punto d’entrata e un solo punto d’uscita) e sequenziali (cioè eseguite una per volta). Esso ha un paradigma imperativo e supporta l’assegnamento distruttivo (definendo una seconda volta il valore di un elemento, si può sovrascrivere il valore precedente).

Le basi - “Hello World”

Un qualsiasi programma in C ha la seguente struttura:

```
<programma> ::=
{unità-di-traduzione}
<main>
{unità-di-traduzione}
<main> ::= main() <blocco>
<blocco> ::= {[dichiarazioni-e-definizioni][sequenza-istruzioni]}
```

In qualsiasi punto si possono mettere dei commenti. Si tratta di testo ignorato dal compilatore, solitamente usato dai programmatori per specificare aspetti vari del codice.

```
<commento> ::= /*<frase>*/ | // <frase> \n
<frase> ::= {<carattere>}
```

Il commento scritto con /*...*/ finisce quando si scrive */, a prescindere dalla riga in cui ci si trova. Quello scritto con // vale solo per la riga corrente; \n equivale a premere “Invio”.

Il comando `printf(“Testo”);` visualizza il testo scritto tra virgolette. Il ; è mandatorio in main() poiché indica la fine di un’istruzione e l’inizio di un’altra. In “” si può andare a capo con \n e creare un distacco con \t (equivale a premere “Tab”).

Poichè questo comando non è un’istruzione predefinita di C, ma fa bensì parte della libreria stdio (Standard Input/Output), bisogna importare l’ultima all’inizio del programma. Per fare ciò si usa il comando `#include<libreria.h>`, posto prima di main().

Il codice che stampa “Hello World” è il seguente:

```
#include<stdio.h>
main(){
      //Stampa di “Hello World”
      /* Commento
      su più
      righe*/
      printf(“Hello World”);
```

```
}
```

Notare che printf è indentato rispetto al resto dei comandi. Questa è una convenzione per rendere più leggibile il codice: si incolonnano le istruzioni sullo stesso livello logico.

4. Variabili

Sono astrazioni di celle di memoria utilizzate nella programmazione ad alto livello. Esse permettono di salvare determinati tipi di dati e di dare ad essi dei nomi significativi (che quindi aiutano a leggere il programma), al posto che riferirsi alle celle con gli indirizzi di memoria.

Sintassi

```
<definizione-variabile> ::= <tipo> <identificatore>;
<assegnazione-variabile> ::= <identificatore> <espressione>;
<identificatore> ::= <lettera> {<lettera> | <cifra>}
<lettera> ::= “A” | … | “Z” | “a” | … | “z”
<cifra> ::= “0” | “1” | … | “9”
```

Il tipo di una variabile indica il dominio dei dati che una variabile può contenere, assieme alle operazioni che si possono svolgere su di essa. Inoltre, in base al tipo di una variabile il calcolatore determina la codifica utilizzata per trasporre il suo valore in linguaggio macchina e la decodifica utilizzata per mostrare il la variabile in output.

In ogni linguaggio di programmazione, è bene evitare nomi che contengono caratteri speciali e/o iniziano con numeri, poichè, sebbene sintatticamente corretti, sono di difficile lettura da parte di programmatori e utenti. Il nome di una variabile, inoltre, non può coincidere con quello di un comando. Nel caso di C, infine, i nomi di variabili e comandi sono Case Sensitive: ciò significa che il linguaggio vede “a” ed “A” come lettere diverse.

Esempio di definizione:

```
int a; //int indica valori interi (numeri appartenenti a Z)
```

Per definizione si intende la creazione di una variabile. L’assegnazione, invece, è il “riempimento” di una variabile preesistente con un valore (o espressione, calcolata dal programma) in base al suo tipo. In C, l’assegnamento è distruttivo.

Esempio di assegnazione:

```
int a=10;
```

Per snellire il programma, definizione e assegnazione possono essere fatte nello stesso momento. Questo è possibile anche con più variabili dello stesso tipo.

```
int a=10, b=5, c=1;
```

Left Value e Right Value

Quando legge la definizione e l’assegnazione di una variabile, il compilatore sceglie una cella di memoria, crea corrispondenza tra il suo id numerico e il suo identificatore nel programma, e poi la “riempie” con i dati rispettivi, appositamente codificati. L’identificatore, nel programma, tuttavia, in una sola espressione, si può riferire sia al valore della cella che alla cella stessa.

```
int a=10;
a=a+10;
```

L' "a" a sinistra dell'= indica la cella di memoria assegnata alla variabile a. Questo è il Left Value della variabile. L' "a" a destra dell'=, nell'espressione, invece, indica il valore salvato in a. Questo è il Right Value della variabile.

Il compilatore vede l'istruzione sovrastante come: "Salva nella cella di memoria della variabile a (L. V.) un valore pari al suo contenuto attuale (R. V), incrementato di 10".

Quando ad una variabile è assegnato il risultato di un'operazione a due termini in cui compare la variabile stessa, si possono utilizzare gli operatori di assegnamento compatti. Essi portano vantaggi soltanto nella scrittura e nella lettura del codice, il quale risulta più breve.

```
int a=10, b=5;
a+=4; //equivale ad a=a+4, ossia 14
b*=3; //equivale a b=b*5, ossia 15
```

Per l'incremento o il decremento di 1 di una variabile, inoltre, esistono operatori ancora più compatti.

```
int a=10, b=5;
a++; //equivale ad a=a+1 o a+=1, cioè 11
b--; //equivale a b=b-1 o b-=1, cioè 4
```

Stampare e prendere in input variabili

Per stampare le variabili con printf si utilizzano apposite particelle all'interno del messaggio stampato (detto "stringa formato"). Terminato il messaggio, separate tra di loro e da esso da una virgola, si indicano ordinatamente le variabili da sostituire alle particelle.

```
int a=10;
printf("La variabile a contiene il valore: %d",&a);
/*%d indica valori interi in base decimale*/
```

Ogni particella comprende "%", ed esse variano in base al tipo di dato che si deve dare in input.

Le stesse particelle sono usate analogamente nel comando scanf, utilizzato per identificare gli input da tastiera. Esso si può usare per "riempire" qualsiasi variabile indicata, a patto che essa sia già definita. Non può essere utilizzato per inizializzare una variabile ed assegnare essa un valore in input nella stessa riga.

```
int a;
scanf("%d",a);
```

5. La programmazione strutturata

L'elemento chiave di ogni programma sono le istruzioni: azioni che modificano lo stato del programma e/o del sistema. Le strutture (o istruzioni) di controllo permettono di organizzare in istruzioni complesse più istruzioni semplici per ottenere un determinato risultato.

```
<istruzione> ::= <istruzione-semplice> | <istruzione-controllo>
<istruzione-semplice> ::= <espressione>
```

Le istruzioni di controllo hanno visto applicazione informatica nel '69, dopo il loro concepimento nel teorema Bohm-Jacopini del '66. L'ultimo dice che ogni programma è esemplificabile con controlli composti, condizionali o iterativi.

Nacque così la programmazione strutturata, composta soltanto da istruzioni di controllo. Le ultime hanno il vantaggio di avere un solo punto d'ingresso e un solo punto d'uscita, rendendo i programmi facili da seguire e modificare.

Controllo condizionale - Selezione

Si tratta di un "bivio" nell'esecuzione di un programma. In ingresso, viene verificata la veridicità di un'espressione, e, in base ad essa, sono effettuate varie operazioni (a volte se l'espressione è falsa non viene fatto niente, altre volte si hanno istruzioni diverse). Al termine delle operazioni specificate dalla selezione, in uscita, l'esecuzione del programma ritorna "comune".

```
<selezione> ::= <scelta> | <scelta-multipla>
<scelta> ::= if(<condizione>) <istruzione-V> [else <istruzione-F>]
<condizione> ::= <espressione>
```

In C, una condizione falsa genera un valore pari a 0, mentre una vera genera 1; qualsiasi valore ≠ 0 è interpretato come risultato di una condizione vera. Nell'if, dunque, le condizioni (specificate con operatori relazionali come = e <) sono trasformate in 1 o 0 e il programma procede di conseguenza.

È un bene indentare rispetto al resto del programma le istruzioni che si trovano dentro un if.

```
if(a%2==0)
      printf("La variabile è pari");
else
      printf("La variabile è dispari");
```

È possibile inserire un if all'interno di un altro, potenzialmente fino all'infinito.

```
if(...)
      if(...)
            istruzione;
      else
            istruzione;
```

```
else
      printf(...);
```

Si possono negare o combinare tra di loro le istruzioni mediante gli operatori logici. Essi, in sostanza, si comportano come gli operatori matematici della logica booleana, usando 1 e 0 al posto di V e F.

```
if((a<10) && (a>0))
      printf("La variabile è compresa tra 0 e 10.");
```

Scrivere controlli unici senza operatori logici e senza suddivisione su più if spesso porta ad errori semantici. Il compilatore non dà errori, ma l'output non è quello desiderato.

```
if(0<a<10)
/*Il compilatore svolge prima 0<a, generando un valore 0 o 1, il quale poi viene controllato per determinare se è <10. Dato che 0 e 1 sono <10, qualsiasi valore di a farà sì che la condizione sia vera.*/
```

Per velocizzare l'esecuzione dei programmi, infine, C usa il "Corto circuito". Se nel controllare la prima istruzione di un AND logico vede che essa è falsa, il compilatore passa considera subito l'esito dell'AND come falso. Viceversa, se in un OR il compilatore verifica che la prima condizione è vera, esso considera il controllo OR vero senza verificare la seconda condizione.

Controllo composto - Blocco o sequenza

In cicli e iterazioni è previsto l'inserimento di una singola istruzione, svolta in base alla veridicità della condizione. È possibile "barare", però, inserendo un blocco: una sequenza di istruzioni che il programma vede come una sola.

```
<blocco> ::= {[<dichiarazioni-definizioni][<istruzioni>}}
```

È importante notare che le variabili hanno un campo d'azione all'interno del quale esistono e possono essere utilizzate. Il campo coincide col blocco d'istruzioni all'interno del quale le variabili sono definite, assieme a tutti i blocchi da esso contenuti.

```
int a; //esiste nel programma principale e nell'if
if(...) {
      int b; //esiste solo nell'if
      ...
}
```

Controllo iterativo - Ciclo o iterazione

Sono costrutti che, in seguito al controllo della veridicità di un'espressione, se essa risulta vera, eseguono un'istruzione (o un blocco) e ricontrollano la sua veridicità (come se si stesse entrando nel ciclo per la prima volta). Se essa risulta ancora vera, l'istruzione è eseguita di nuovo. Si prosegue così finchè l'istruzione non risulta falsa.

In genere, una variabile coinvolta nel controllo deve essere soggetta a variazioni all'interno delle istruzioni del blocco. Altrimenti, una volta entrato all'interno del ciclo, il programma rischia di non raggiungere mai che la condizione sia falsa, creando così un loop infinito.

È opportuno notare che, se si definisce un blocco, si esce e si rientra in esso più volte, e che ad ogni uscita le variabili in esso contenute sono eliminate.

Ci sono vari tipi di iterazione:

`<int> ::= <while> | <do-while> | <for>`

`<while> ::= while(<condizione>) <istruzione>`

Il while si usa quando non si conosce (o non è ricavabile) il numero esatto di cicli necessari al programma per raggiungere l'esito desiderato. Esso ha un controllo "in testa", svolto prima dell'inizio delle operazioni del ciclo. In alcuni casi, la condizione potrebbe essere sin dall'inizio falsa, e, perciò, il programma non svolgerebbe nemmeno una volta il ciclo while, scavalcandolo.

`<do-while> ::= do <istruzione> while(<condizione>)`

Questa è una variante del while con controllo "in coda", cioè fatto dopo la prima esecuzione delle istruzioni. Questo comporta che, a prescindere dalla veridicità della condizione, le istruzioni contenute nel ciclo verranno eseguite almeno una volta.

`<for> ::= for(<inizializzazione>;<condizione>;<modifica-contatore>)`

Il for si utilizza quando è conosciuto o ricavabile il numero di volte che il ciclo sarà eseguito. Esso, nella sua dichiarazione, comprende l'inizializzazione e l'assegnazione di un valore al contatore del ciclo, la dichiarazione della condizione di permanenza, e la modifica del contatore. L'inizializzazione è svolta soltanto una volta, durante la prima entrata nel for, mentre la verifica della condizione e la modifica del contatore sono effettuate ad ogni iterazione, rispettivamente, all'inizio e alla fine.

6. Tipi di dati

Il tipo di un dato, indicato nella dichiarazione di una variabile, indica l'insieme di valori che essa può contenere. Inoltre, diversi tipi di dati prevedono diverse codifiche per il loro salvataggio in memoria, e diverse operazioni e controlli che possono essere svolte su di essi.

Ogni tipo di dato, effettivamente, è salvato in memoria come un numero binario di n cifre; n dipende anch'esso dal tipo di dato. La variabile di quel tipo può dunque contenere 2^n dati diversi, dallo 0 a 2^n-1. In ogni linguaggio, però, si ha il problema dell'Overflow, che consiste nell'"uscire" dall'intervallo dei possibili valori (incrementando il massimo o decrementando il minimo).

Alcuni linguaggi danno errore quando si verifica un Overflow, ma questo comporta controlli svolti su ogni operazione di assegnamento. C, per essere più veloce, evita questi controlli e "lascia fare". L'Overflow, dunque, non provoca errori in C, ma modifica il valore della variabile non correttamente:

```
int a=0-1; //a assume il valore massimo contenibile da int
int b=(0-1)+1; //b assume il minimo valore contenibile da int (massimo valore + 1)
```

Numeri naturali e numeri interi

I numeri naturali sono salvati con i loro corrispondenti in decimale in memoria. In C si possono usare tipi di variabili dalla diversa capienza per contenere numeri naturali:

- unsigned char, 8 bit;
- unsigned short, 16 bit;
- unsigned int, 16 o 32 bit (dipende dal compilatore e dal SO);
- unsigned long, 32 bit (raramente 64);
- unsigned long long, 64 bit.

Esistono varie dimensioni per accomodare diverse esigenze. Il vantaggio di avere diverse grandezze è che si può scegliere il tipo di variabile adatto alle proprie esigenze, evitando di occupare più memoria del dovuto con un tipo troppo capiente.

I numeri interi sono rappresentati analogamente, ma le cifre assumono diverso significato. I valori da 0 a 2^{n-1} sono interpretati come numeri positivi, mentre quelli da 0 a 2^{n-1} sono interpretati come numeri negativi; entrambi i gruppi sono in ordine crescente. Così facendo, calando oltre il valore nullo, si arriva all'inizio dei valori negativi, mentre aumentando oltre il valore massimo si raggiunge l'inizio dei valori positivi. Questa codifica trae vantaggio dall'Overflow.

Essa prevede, tuttavia, un Overflow effettivo nel mezzo della variabile, dove il massimo e il minimo valore interpretabili sono adiacenti. Per questo, superare il massimo valore porta all'inizio dei valori più bassi, mentre calare oltre il valore minimo porta alla fine dei valori più alti.

Esistono tipi di variabili analoghi a quelli dei numeri naturali per salvare i numeri interi:

- char, 8 bit (2^7 positivi e 2^7 negativi -1);
- short, 16 bit (2^{15} positivi e 2^{15} negativi -1);
- int, 16 o 32 bit (2^{15} positivi e 2^{15} negativi o 2^{31} positivi e 2^{31} negativi -1);
- long, 32 bit (raramente 64) (2^{31} positivi e 2^{31} negativi -1);
- long long, 64 bit (2^{63} positivi e 2^{63} negativi -1).

Numeri decimali

I numeri decimali sono salvati come potenze in notazione scientifica di 2. In particolare, effettivamente, in memoria sono salvati la mantissa e l'esponente della notazione, in quanto è sempre noto che la base è 2. Questo schema è noto come rappresentazione a virgola mobile.

Nella dichiarazione della variabile, si possono aggiungere più o meno cifre alla mantissa. Una mantissa con poche cifre non è molto precisa, ma permette alla variabile di assumere una gamma di valori più ampia. Viceversa, una mantissa estesa offre grande precisione, ma non una gamma altrettanto ampia di valori raggiungibili.

Ci sono vari tipi di variabili per la virgola mobile:

- float, 32 bit, particella in stringhe formato "%f";
- double, 64 bit, particella in stringhe formato "%lf";
- long double, 80 bit, particella in stringhe formato come double.

Scrivendo in una stringa formato di output qualcosa come "%.4f", si stamperà la variabile con 4 cifre dopo la virgola.

Caratteri e simboli

Sono rappresentati col dato char, analogo a quello dei numeri. Questo si ha perchè in C i simboli e i caratteri sono rappresentati con valori numerici, fatti corrispondere alla tabella ASCII (American Standard Code for Information Interchange), dove gli ultimi sono indici di un determinato simbolo.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

char può contenere un solo carattere alla volta, quindi non stringhe (sequenze di caratteri). Inoltre, come si vede nella tabella soprastante, alcuni input sono classificati come caratteri; tra di essi rientrano lo spazio, \t (tab) e \n (invio).

Infine, le variabili char supportano gli stessi operatori delle variabili dedicate a valori numerici, e nelle operazioni I/O possono essere indicate con due particelle. “%d” interpreta il loro contenuto come un numero intero, mentre “%c” interpreta i contenuti come la corrispondente lettera della tabella ASCII. A prescindere dalla particella usata, il numero è sempre effettivamente salvato in memoria come un numero binario.

```
char a, b;
scanf(“%c”,&a); //a ha valore pari all’indice ASCII del car. inserito
//i caratteri vanno scritti tra apici, es ‘A’
scanf(“%d”, &b); //b assume il valore inserito
```

7. Ambiguità in operazioni e operatori

In C si hanno alcune nozioni riguardanti la natura e l'interpretazione di operazioni e operatori che è meglio sapere per programmare in modo efficace ed efficiente.

Operazioni eterogenee

Il calcolo di espressioni e l'assegnamento di valori a variabili è possibile soltanto se tutti i dati coinvolti sono dello stesso tipo. In tal caso, si parla di operazioni omogenee. Qualora, però, si tentasse di fare un calcolo tra valori di diverso tipo, oppure di assegnare a una variabile un valore non compatibile col tipo dell'ultima, si ha un'operazione eterogenea.

Le operazioni eterogenee possono essere tra dati compatibili o incompatibili. Gli ultimi, spesso valori numerici, sebbene di tipo diverso, sono convertibili nello stesso tipo.

La conversione è messa in atto automaticamente da C in operazioni eterogenee, avvisando l'utente (operazione detta Coercion). Perché l'avviso? Perché le conversioni possono portare perdite di dati: esse cambiano il modo in cui il dato fisico è visualizzato e in memoria, ma non adattano l'ultimo al nuovo formato. Per capire come queste conversioni alterano, talvolta, i valori, basta pensare alle differenze in codifica e lettura di numeri naturali, interi e razionali.

In espressioni eterogenee, si ha la "Promozione" dei dati in formati compatibili. Quando due dati hanno tipo diverso, quello con tipo meno capiente viene interpretato come se fosse dello stesso tipo dell'altro operando.

Nella pratica, quindi, l'operando di tipo più capiente è sommato a una variabile temporanea dello stesso tipo, contenente lo stesso dato fisico in memoria dell'altro operando.

La sequenza delle promozioni è:

char → short → int → long → float → double → long double

Negli assegnamenti, invece, a prescindere del tipo dell'espressione, essa è adattata a quello della variabile. Questo comporta una conversione non solo in "espansione", ma anche in "contrazione". Contrarre un dato di un tipo all'interno di una variabile di tipo più piccolo comporta che vengano salvati in essa soltanto i bit meno significativi del dato originale.

Anche l'utente può effettuare operazioni di assegnamento, tramite le cosiddette istruzioni di casting. Esse si comportano come operatori unari, e, in quanto inserite dall'utente, le conversioni da esse provocate non sono segnalate dal compilatore.

La sintassi per gli operatori di casting è:

```
<casting> ::= (<tipo>) <espressione>;
```

Priorità e associatività degli operatori

```
<espressione> ::= (<variabile> | <costante> | <espressione> <operatore>
```

```
<espressione> | <operatore-unario> <espressione>;
```

Nonostante questa definizione, talvolta si incontrano espressioni comprendenti più operatori, che, per definizione, sono espressioni di espressioni. Tali espressioni "complesse" sono fonte di ambiguità: come le interpreta C?

In questi casi, C esegue le singole espressioni secondo una gerarchia di priorità:

- 1, parentesi (viene eseguito prima ciò che è in () interne a un'espressione);
- 2, operatori unari (es. casting, ++, !);
- 3, moltiplicazione e divisione (*, /, %);
- 4, addizione e sottrazione (+, -);
- 6, controlli di disuguaglianza (es. <, <=);
- 7, controlli di uguaglianza (==, !=);
- 11, and (&&);
- 12, or (||);
- 14, assegnamenti (es. +=, =, /=).

In caso di più operatori consecutivi aventi priorità uguale, si fa riferimento all'associatività degli ultimi, che può essere destra o sinistra. Gli operatori con associatività sinistra hanno le loro espressioni eseguite da quella più a destra a quella più a sinistra, mentre per gli operatori con associatività destra (molto pochi) è il contrario.

Somma e sottrazione sono esempi di operatori con associatività sinistra.

```
a = x + y - z;
//equivale a
a = (x + y) - z;
```

Un esempio, invece, di operatore con associatività a destra, è quello dell'assegnamento. Notare che l'assegnamento è visto come una vera e propria operazione con un risultato. L'ultimo, in ogni caso, è il valore assegnato alla variabile, quindi l'espressione che si trova a destra dell'uguale.

Sapendo ciò, si possono scrivere istruzioni come:

```
a = x = y = z = 3;
//tutte le variabili coinvolte hanno valore 3
```

8. Funzioni

Una funzione è un tipo di sottoprogramma consistente in un operatore non primitivo. Essa è come un operatore in quanto riceve dati in input (detti parametri) ed esegue elaborazioni su di essi, restituendo un valore finale. Ogni funzione è identificata da un nome.

In C, le funzioni sono molto diffuse, ma, in altri linguaggi, si hanno anche le procedure, che, invece, costituiscono istruzioni non primitive.

Spesso, determinate funzioni richiedono l'importazione di librerie ("pacchetti" predefiniti di funzioni) per il linguaggio. Importata la giusta libreria, si può invocare una funzione nel codice con la sintassi:

```
<invocazione> ::= <nome-funzione> (<parametri-attuali>);
<parametri-attuali> ::= [<espressione>]{,<espressione>};
```

L'invocazione può essere in qualsiasi punto, anche dentro espressioni o controlli condizionali.

Creare una funzione

L'utente può definire funzioni proprie all'interno dei programmi. La definizione della funzione va posta prima del main, e indica i parametri in input, le operazioni svolte (un blocco, detto corpo), e il valore in output (di "ritorno"). La sintassi è la seguente:

```
<definizione-funzione> ::=
<tipo-return> <nome-funzione> (<parametri-formali>)
{ <blocco>; return <valore>; }
<parametri-formali> ::= void | {<definizione-variabile>}
<valore> ::= <costante> | {<variabile>}
```

Scrivendo void al posto dei parametri si indica che la funzione non ne necessita alcuno in entrata.

Return è l'istruzione che restituisce un valore elaborato all'interno della funzione. Il tipo del valore di ritorno deve essere quello di `<tipo-return>`. Return mette fine all'esecuzione del corpo della funzione, a prescindere dalla presenza di istruzioni successive.

L'invocazione di una funzione personalizzata è uguale a quella di una funzione di libreria.

I parametri sono detti formali in sede di definizione in quanto sono variabili "segnaposto" per i valori effettivi che saranno utilizzati nelle invocazioni nel main; in tal sede, i parametri si dicono attuali. Se i parametri attuali non sono nello stesso tipo, ordine e quantità dei parametri formali si avranno errori di compilazione e/o semantica.

Nella stesura del corpo della funzione si possono utilizzare i parametri formali, altre variabili definite all'interno del blocco e costanti. Questi dati esistono (e sono quindi utilizzabili) solo all'interno del blocco.

Scrivendo void al posto dei parametri si indica che la funzione non ne necessita alcuno in entrata.

Le funzioni possono contenere anche istruzioni di input e di output.

Vantaggi delle funzioni e interfaccia

Le funzioni permettono di scrivere una volta sola un blocco di codice ricorrente. Questo ha vantaggi in ambiti di efficienza, memoria, chiarezza e scrittura del codice.

Esse, inoltre, possono essere condivise tra utenti e utilizzate in programmi diversi senza alcuna necessità di modifica. Le variabili al loro interno, infatti, hanno uno scope limitato alla funzione stessa, quindi non necessitano di essere rinominate e possono essere omonime con altre variabili definite all'interno di altre funzioni o nel main.

Le funzioni, infine, sono isolate, poiché dal main si possono solo dare dati in input alla funzione e operare sul suo valore di ritorno, ma non accedere o modificare variabili e istruzioni interne alla funzione stessa.

In una funzione si hanno variabili formali e parametri formali. Le prime sono definite normalmente, mentre i secondi sono dati direttamente in input alla funzione quando essa è richiamata.

Modificare i parametri formali è l'unico caso in cui un'alterazione alla struttura di una funzione necessita modifiche del codice in cui la si richiama. Questo si ha perchè, così facendo, si modifica, l'interfaccia, ossia come la combinazione del nome della funzione e dei relativi parametri richiesti (in sostanza, ciò che è riportato nell'invocazione).

Modello runtime

Questo modello si riferisce all'esecuzione effettiva dei programmi in C.

La prima cosa ad essere eseguita è il main. Quando è incontrata una funzione al suo interno, vengono messi in atto questi passi:

1) Sono creati degli spazi in memoria riservati ai parametri della funzione;
2) Sono allocati negli spazi rispettivi i valori relativi ai parametri;
3) Sono eseguite tutte le istruzioni presenti nella funzione;
4) È restituito il risultato della funzione;
5) Viene liberata la memoria riservata a parametri e variabili locali della funzione.

Anche il main stesso segue un processo di esecuzione simile.

In questa sequenza, si definisce Lifetime il lasso di tempo per il quale una variabile esiste.

Inoltre, per assicurarsi un'esecuzione corretta, si crea un record di attivazione della funzione, contenente informazioni utili. Tra di esse spiccano:

- I parametri ricevuti;
- Le variabili locali create;
- Il Return Address (RA), l'indirizzo del punto del programma all'interno del quale si è incontrata la funzione; serve per riprendere l'esecuzione del programma proprio da dove la si è interrotta;

- Il Dynamic Link (DL), un collegamento temporaneo tra il main e il file contenente la funzione che permette al primo di mettere in azione la seconda.

Quando sono in esecuzione funzioni contenenti altre funzioni, si crea una catena di link dinamici: il DL che lega il main ad una funzione è seguito da un ulteriore DL tra l'ultima e una funzione in essa richiamata. Questi link sono gestiti con una politica LIFO (Last In, First Out), ossia gli ultimi DL ad essere creati sono i primi ad essere eliminati e viceversa.

È importante specificare che il valore di ritorno di una funzione può essere restituito da un registro della CPU o da uno spazio apposito creato nel record di attivazione, allocato nella memoria RAM. La prima opzione è preferibile, ma non sempre attuabile per via della limitata capacità dei registri della CPU.

Procedure

Le procedure sono tipi di sottoprogrammi che, a differenza delle funzioni, non generano nuovi operatori (es. pow), ma, bensì, nuovi comandi (es. printf). C è basato sulle operazioni aritmetiche, quindi le funzioni prevalgono sulle procedure, che, nella fattispecie, sono visti come una funzione particolare.

Le procedure, infatti, in C, sono semplicemente funzioni che non hanno un valore di ritorno (anche se possono avere parametri in input). La loro definizione è analoga a quella delle comuni funzioni, ma il loro "tipo" è "void". Poiché non hanno valore di ritorno, esse vanno richiamate di per sé, e non all'interno di espressioni aritmetiche.

```
<definizione-procedura> ::=
void <nome-procedura> (<parametri-formali>)
{ <blocco>; }
```

Ricorsività

Una funzione può invocare sè stessa al suo interno. Essa si dice "ricorsiva", e risulta utile per risolvere problemi complessi ma ripetitivi. A livello pratico, ogni ricorsione successiva della funzione corrisponde ad un nuovo record di attivazione.

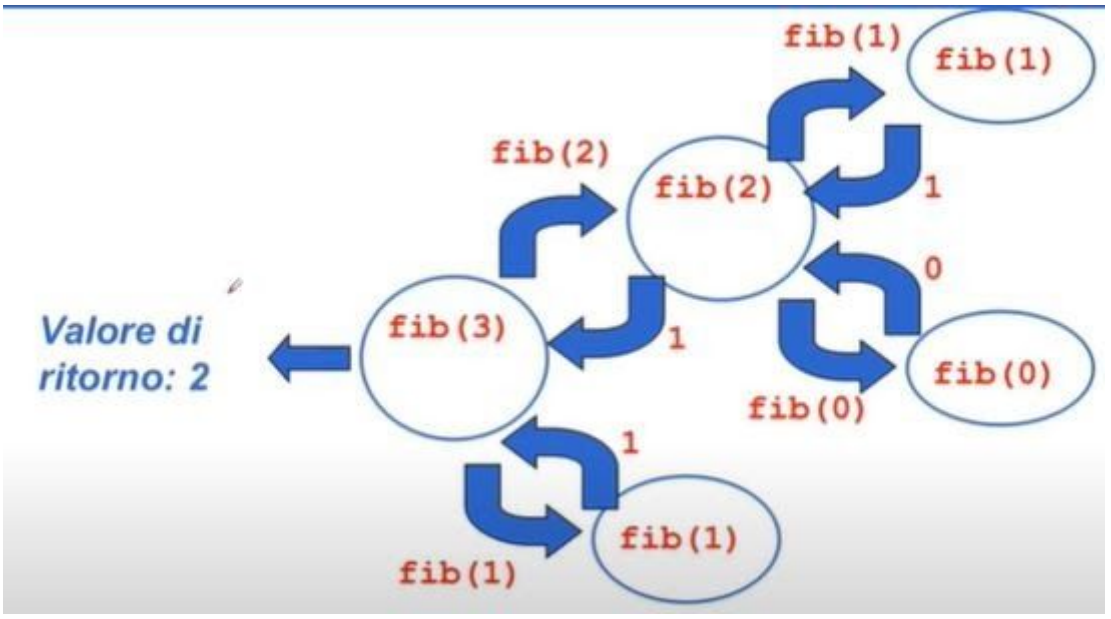
Ogni funzione ricorsiva deve prevedere un caso base, per il quale non è necessario richiamare sé stessa per dare un risultato. Per far ciò ogni funzione ricorsiva prevede controlli atti a verificare se ci si trova nel caso base.

Esempio di funzione ricorsiva per calcolare N!:

```
int fact (int n) {
      if (n == 0) {
      return 1;
      } else {
      return n * fact(n - 1);
}
```

Si parla di ricorsività non lineare quando si ha a che fare con funzioni che richiamano sè stesse più volte al proprio interno. Dunque, prima di compiere un'iterazione di una funzione, anziché compiere una sola sottoiterazione, è necessario compierne molteplici; ogni sottoiterazione comporta molteplici ulteriori sottoiterazioni, e, al termine, si ha una specie di struttura ad albero.

Di seguito, la schematizzazione di una funzione iterativa per calcolare il numero in posizione n della sequenza di Fibonacci, dato dalla somma dei numeri in posizione n-2 e n-1 della sequenza. Le posizioni 0 e 1 contengono il valore 1.



9. Array e stringhe

I dati possono essere di tipi scalari o strutturati. L'array è un dato strutturato, che rappresenta una collezione finita di n variabili dello stesso tipo, identificate da un indice univoco, con valori andanti da 0 a n-1.

`<costruzione-array> ::= <nome-array> [<numero-elementi>];`

È possibile riferirsi alle singole variabili contenute in un array nel seguente modo

`<nome-array> [<indice>] ...`

Le variabili facenti parte di un array sono trattate come comuni variabili dello stesso tipo.

Per inizializzare un array si usano le parentesi graffe

`<nome-array> [<indice>] = { <variabile>, <variabile>, … }`

Preprocessore

Per rendere più dinamici e facilmente modificabili programmi contenenti array, si possono definire le dimensioni degli ultimi utilizzando il preprocessore. L'ultimo opera sul codice prima del compilatore, una volta avviata la compilazione, e collega ad esso librerie ed effettua (se indicate) sostituzioni nel testo.

Nella fattispecie, si possono indicare (prima di qualsiasi istruzione) sostituzioni che generano costanti: si indica al preprocessore di sostituire ogni istanza di un simbolo specifico nel programma con un determinato valore. Questa è solo una delle applicazioni della sostituzione tramite preprocessore; la sintassi è la seguente:

`#define <testo1> <testo2>`

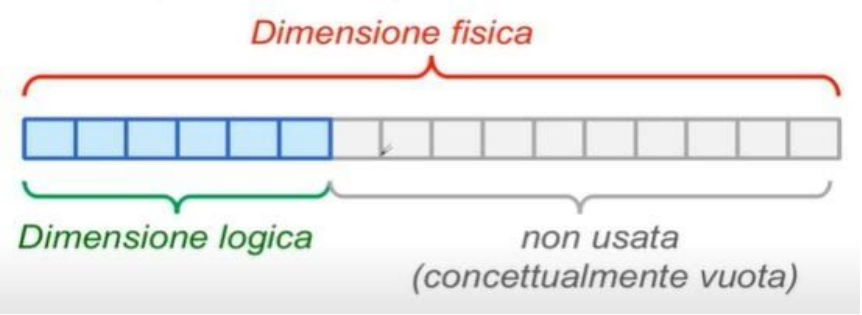
L'istruzione sostituisce ogni istanza di <testo1> con <testo2>.

Notare che non deve essere messo un ; al termine del comando, poiché esso non è un comando di C. I comandi per il preprocessore sono identificati dal # al loro inizio.

Dimensione fisica e dimensione logica

Quando si parla di array, si distinguono:

- Dimensione fisica, lo spazio in memoria dedicato l'array nella sua interità (spazio singola variabile * numero di variabili nell'array);
- DImensione logica, la porzione della dimensione fisica dell'array effettivamente utilizzato, ossia contenente dati significativi (è sempre <= dimensione fisica).



In programmi in cui si sospetta che la dimensione logica non combaci con la dimensione fisica, è opportuno salvare in una variabile quanti (o quali) sono gli effettivi “slot” utilizzati. Questo evita errori di sintassi o semantica dovuti allo svolgimento di operazioni su “slot” vuoti (o contenenti valori casuali), così come migliora l’efficienza dei programmi evitando operazioni inutili.

Stringhe

Una stringa è un array di caratteri. In C, le stringhe possono essere salvati come array di variabili char, e il linguaggio offre funzionalità apposite a supporto di tale metodo.

Ogni stringa ha la sua parte significativa (corrispondente alla dimensione logica) seguita da un char di valore ‘\0’ (indice ASCII 0), interpretato da C come fine della stringa. Se ci sono altri char dopo \0, definiti o nulli che siano, essi saranno ignorati.

Per prendere in input o dare in output le stringhe nella loro interità si usa la particella %s; con l’ultima, non è necessario mettere le [ ].

Per l’input, C piazza automaticamente lo \0 quando trova un input a cui non corrisponde un carattere (es. spazio o invio), e, inoltre, non è necessario piazzare & prima del nome della variabile per indicare che ci si riferisce ad essa.

Per le stringhe si ha la libreria string.h, in cui sono definite le seguenti funzioni:

- strcpy(s1,s2); sovrascrive la prima parte di s1 con s2, seguito da \0.
- strcat(s1,s2); mette s2 al posto di \0 in s1.
- strcmp(s1,s2); restituisce 0 se s1=s2, n>0 se s1 viene dopo s2 in ordine alfabetico, e, viceversa, n<0.
- strlen(s1); restituisce la lunghezza di s1, senza contare \0.

10. Strutture e typedef

Sono un tipo di dato progettato per contenere una quantità finita di “campi”. I campi sono vere e proprie variabili, e, all’interno di una singola struttura, si possono avere campi di tipo diverso. La sintassi per creare una struttura è la seguente

```
struct [<etichetta>] {
     {<definizione-di-variabile>}
} <nome-struttura>;
```

Scrivendo più di un <nome-struttura> si creano più variabili struct contenenti gli stessi campi.

Per indicare un singolo campo, la sintassi è la sottostante. Nella pratica, un campo si comporta come una qualsiasi variabile dello stesso tipo.

```
<nome-struttura>.<nome-campo> ....
```

Etichette, inizializzazione e funzioni

Creare un’etichetta è opzionale. Se fatto, permette di definire ulteriori strutture aventi gli stessi campi senza dover elencare gli ultimi, ma semplicemente richiamando l’etichetta. Sotto si ha un esempio:

```
struct <etichetta> <nome-struttura>;
```

Questo torna utile quando si definiscono variabili in punti intermedi del programma oppure quando si vuole definire una variabile struttura che ha come campo un’altra variabile struttura.

Per inizializzare velocemente una struttura utilizzando un’etichetta, si può usare la notazione con le { }. In questo modo, però, bisogna fare attenzione: i valori devono essere inseriti nello stesso ordine dei campi.

```
struct <etichetta> <nome-struttura> = { {<valore-campo>} };
```

Le strutture, a differenza di array e stringhe, possono essere date come parametri formali e parametri attuali senza alcun accorgimento particolare. Questo è perché array e stringhe sono “casi particolari” di tipi di dati preesistenti, mentre le strutture sono un tipo di dato a sé stante.

Typedef

```
typedef <tipo-esistente> <nuovo-tipo>;
```

La sintassi soprastante permette di creare nuovi tipi di dati all’interno di un programma, a partire da un tipo di dato già esistente. Questo permette di dare nuovi nomi ad altri tipi di dati. Nel caso delle strutture, si può definire una determinata struttura come un tipo di dato a sé stante, mentre con gli array ciò permette di dare un nome agli array di una determinata durata.

Per definire una o più variabili dello stesso tipo si potrà fare come di seguito:

```
<nuovo-tipo> <nome-variabile>;
```

È importante notare che <nuovo-tipo> non è una variabile, anche se definito tramite un comando dall'utente. Fare operazioni su di esso che non siano un assegnamento di variabili ad esso (es. operazioni aritmetiche) porterà dunque ad errori di compilazione.

Dati enumerativi

```
typedef <tipo-esistente> <nuovo-tipo>;
```

Oltre che a definire tipi di dati nuovi che consistono in strutture specifiche, typedef permette di creare i dati enumerativi.

Un dato enumerativo accetta solo valori appartenenti ad un certo pool, specificato dall'utente. Questi valori, indipendentemente da cosa rappresentino, sono interpretati da C come costanti numeriche.

Di seguito si ha la sintassi generica e un esempio di definizione, assieme a una rappresentazione di come sono effettivamente interpretati i dati:

```
typedef enum { a1, a2, … , an } <nome-dato>;
typedef enum { lun, mar, mer, gio, ven, sab, dom } Giorni;
// effettivamente { 0, 1, 2, 3, … }
```

L'utente può anche definire un valore di riferimento in base al quale assegnare le costanti rappresentative ai dati.

```
typedef enum { lun=1, mar, mer, gio, ven, sab, dom } Giorni;
// i valori partiranno da 1, quindi si avrà { 1, 2, … }
```

Più estensivamente, l'utente può anche specificare gli effettivi valori delle costanti corrispondenti ad ogni singolo possibile valore enumerativo.

Detto ciò, due cose sono da evitare in ogni caso per evitare ambiguità:

- Assegnare deliberatamente a più possibili valori la stessa costante rappresentativa in un tipo di dato enumerativo;
- Assegnare possibili valori identici nello stesso tipo di dato enumerativo o in tipi diversi (il programma non saprebbe come interpretare un valore enumerativo, ossia non saprebbe quale costante assegnare ad esso).

Attraverso i tipi di dati enumerativi si possono definire i dati di tipo booleano, non presenti in C:

```
typedef enum { false, true } boolean;
// per esempio, C interpreterà dunque false < true come vero.
```

Union

```
typedef <tipo-esistente> <nuovo-tipo>;
```

Una union è un tipo di dato contenente più campi simile ad una struct, che, però, mantiene il valore di uno solo di essi alla volta. Ad una variabile union è dedicato spazio in memoria

pari alla dimensione del campo più grande. Le union sono utilizzate per immagazzinare in una sola variabile dati di tipi diversi in momenti diversi del programma, risparmiando così memoria.

Se si cerca di compiere operazioni su un campo non inizializzato di una union si otterranno risultati senza senso. Il compilatore infatti interpreterà l'area di memoria "condivisa" come se si trattasse di una variabile del tipo di quella richiamata, che è diverso però da quello della variabile "valida".

C non mette a disposizione strumenti per tenere traccia del valore attualmente contenuto in una struct. Per fare ciò, dunque, è opportuno creare una variabile apposita.

La sintassi per creare una union è simile a quella per creare una struct:

```
union [<etichetta>] {
      {<definizione-di-variabile>}
} <nome-struttura>;
```

11. Puntatori

Un puntatore è un tipo di variabile che contiene un indirizzo in memoria. Esso può essere utilizzato per riferirsi alle variabili mediante il loro indirizzo in memoria. La sua sintassi è:

```
<tipo> * <nome-puntatore>;
```

Il tipo del puntatore indica al compilatore quanti byte considerare a partire dall'indirizzo contenuto da esso, che è sempre il primo di quelli allocati ad una singola variabile. Per indicare l'indirizzo di una variabile si usa l'operatore &.

Si mette * prima del nome dei puntatori:

```
int x;
int * p;
*px = &x; //p contiene l'indirizzo della variabile intera x
```

Passaggio per copia e passaggio per riferimento

In C, dando una variabile come parametro attuale in una funzione, l'ultima è passata per copia. Il suo valore è copiato nel parametro formale della funzione. L'ultimo è coinvolto nelle operazioni e il parametro attuale non è toccato.

Questo ha vari vantaggi. Il principale consiste nel non aver modifiche "occulte" di una variabile in una funzione in cui essa è un parametro. Contemporaneamente, così si hanno delle limitazioni. Modificare una variabile esistente al di fuori di una funzione all'interno dell'ultima potrebbe essere comodo, ad esempio, per avere più valori di ritorno (le strutture sono un'opzione, ma sono scomode e inefficienti).

Il passaggio per riferimento è l'opzione alternativa, non supportata in C. Effettua le operazioni definite in una funzione direttamente sui parametri attuali e non su variabili "usa e getta" contenenti i loro valori. Vantaggi e svantaggi di questo sistema sono opposti a quelli del passaggio per copia.

In C, si può "derivare" il passaggio per riferimento fornendo a funzioni e procedure gli indirizzi in memoria delle variabili da modificare, sulle quali il programma agirà mediante puntatori. Continuando l'esempio di prima:

```
int funzione(int x) {…} //copia usa il valore di x, che è invariata
int funzione(int *px) {…} //agisce su x mediante l'indirizzo in px
```

Volendo, si può inserire const davanti a un parametro per indicare che è impossibile modificarlo. Per esempio:

```
int funzione(const int x) {…}
//modificare x genera un vero e proprio errore nella compilazione
```

Array come parametri

In C, non ci si può riferire ad array e stringhe nella loro interezza. Il nome di un array, infatti, non indica l'array intero, ma bensì l'indirizzo del suo primo elemento. Gli elementi

seguenti sono recuperati dal compilatore perché esso ricava quante celle dovrà saltare per raggiungerli considerando quanti elementi intercorrono tra essi e il primo e quanti byte sono occupati da ognuno.

[] è dunque un operatore analogo a * dei puntatori, soltanto che si riferisce sempre agli array e opera in funzione di un indice inserito tra le parentesi.

Per dare un array come parametro di una funzione si deve creare un parametro formale specificando il tipo e il fatto che si tratta di un array. Nell'invocazione della funzione è sufficiente inserire il nome dell'array; la dimensione è poi ricavata dal compilatore poichè esso conosce quella del parametro attuale.

Gli array, dunque, possono essere passati soltanto per riferimento alle funzioni.

```
void procedura(int a[]) {…}
main() {
      …
      int v[5];
      …
      procedura(int a[]);
      …
}
```

Poiché il nome di un array è un puntatore, nei parametri formali delle funzioni essi si possono richiamare anche tramita la notazione con *, tipica dei puntatori. Essi, però, non possono essere dichiarati in quella maniera, poichè * non dà indicazione alcuna sul numero di campi, che, per default, sarà 1.

Array multidimensionali

```
<tipo> <identificatore>[dim1][dim2]…[dimn]
```

In tal modo si può definire un array comprendente dim1 array di dim2 elementi, ognuno dei quali è un array di dim3 elementi, ecc… I singoli elementi finali sono identificati da una combinazione degli indici di ogni array.

Gli array con due dimensioni sono comunemente detti matrici.

```
int m[3][2];
```

Gli array multidimensionali sono memorizzati secondo l'ordine degli indici, a partire dall'ultimo. Valgono ancora i concetti di dimensione logica e dimensione fisica.

Per esempio (vedi matrice sopra):

Rappresentazione concettuale:

m[0][0]	m[0][1]
m[1][0]	m[1][1]

m[2][0]	m[2][1]

Memorizzazione effettiva:

m[0][0]	m[0][1]	m[1][0]	m[1][1]	m[2][0]	m[2][1]

Segue che l'indirizzo di un preciso elemento [i1][i2]...[in] di un array è dato dalla formula:

```
i1*dim1*…*dimn + i2*dim2*…*dimn + … + in-1*dimn + in
```

Notare che l'unico dato non necessario è la prima dimensione dell'array. Questo è da tenere a mente quando si danno array multidimensionali come parametri formali di funzioni: solo la prima dimensione può essere omessa.

Come i comuni array, anche gli array multidimensionali possono essere inizializzati rapidamente:

```
int m[3][2] = {
{ 1, 2, 3 }
{ 4, 5, 6 }
};
```

12. File

Un file è un'astrazione di informazioni salvate sulla memoria di massa del calcolatore. I file hanno un "formato", ossia un modo in cui le informazioni al loro interno sono organizzate e interpretate. Spesso i file sono composti da una serie di "record" (blocchi di dati) uniformi.

La gestione dei file è affidata al SO, responsabile anche della loro organizzazione in memoria. Ogni file fa parte del "file system", una grande struttura ad albero dove ogni file è identificato da un nome assoluto (combinazione univoca del suo "percorso", cioè la "sequenza" di file che lo contengono) e da un nome relativo (rilevante per la cartella in cui esso si trova).

I file e il linguaggio C

C permette l'interazione coi file. Il tipo di dato FILE, contenuto in <stdio.h>, è un apposito puntatore. Il SO crea corrispondenza tra questi puntatori e i file specificati nella loro creazione; ciò si dice "flusso di comunicazione", e mediante esso si può operare direttamente sui file. La corrispondenza è "aperta" e "chiusa" nel codice mediante apposite istruzioni.

I flussi possono essere binari o testuali. I primi "riportano" i file in linguaggio macchina, mentre i secondi li "traducono" in righe di testo separate da '\n'. Il tipo di flusso influenza la gestione dei dati nel programma. Qualsiasi file può essere aperto in formato binario, ma non è così per il formato testuale.

Un esempio di flussi testuali sono i tre file generati per gestire input e output nell'esecuzione di un programma:

- stdin, input testuale (astrazione della tastiera);
- stdout, output a video (astrazione del monitor) riservato al programma;
- stderr, output a video riservato ai messaggi di errore.

Generando un flusso si deve anche specificare le azioni che esso permetterà di compiere all'interno del programma. È possibile aprire flussi di lettura, di scrittura e/o di modifica.

In ogni caso, l'accesso ai file è "sequenziale", in quanto C li legge come se si trattasse di un array composti dai "blocchi" del file. I flussi, però, permettono solo di passare da un blocco al seguente. L'accesso sequenziale è immaginato come una "testina" che legge un nastro magnetico.

Operazioni basilari

```
FILE *fopen (nome_file , modalità);
```

Apre un flusso di comunicazione. Le modalità di apertura sono:

- r, lettura testuale del file a partire dall'inizio;
- w, sovrascrittura testuale del file a partire dall'inizio;
- a, scrittura testuale alla fine del file;
- b, si può aggiungere a r, w, a per leggere un flusso binario;

- +, si può aggiungere alle altre particelle per permettere sia lettura che scrittura.

Il puntatore vale 0 (NULL) se non è stata effettuata l'apertura. Se ∄ file, con r si avrà un errore, mentre con w e a esso verrà creato e gli sarà dato il nome specificato.

```
int fclose(FILE *fp);
```

Chiude un flusso di comunicazione. Restituisce 0 se l'operazione va a buon fine.

È importante chiudere i file poiché, per fini di efficienza, le operazioni su di essi non sono effettuate istantaneamente, ma in gruppi detti "buffer". L'attuazione dei buffer si verifica in determinati casi, tra i cui la chiusura del flusso. Non chiudere un flusso potrebbe dunque comportare la mancata esecuzione di operazioni sul file.

```
int feof(FILE *fp);
```

Restituisce 0 se non è stata raggiunta la fine del file (ossia l'ultimo blocco) con la precedente operazione di lettura o scrittura.

Lettura e scrittura

```
int fprintf(FILE *fp, stringa_formato);
int fscanf(FILE *fp, stringa_formato);
```

Queste istruzioni sono usate per collegamenti testuali. Rispettivamente, scrivono e leggono nel file del puntatore *fp quanto indicato nella stringa formato.

```
int fwrite(addr, int dim, int n, FILE *fp);
int fread(addr, int dim, int n, FILE *fp);
```

Queste istruzioni scrivono e leggono byte in collegamenti binari. Le operazioni sono svolte su n elementi, ossia gruppi di dim byte; il totale di byte sul quale si agisce è n*dim.

La prima scrive gli elementi a partire dall'indirizzo addr (l'elemento dal quale leggere), mentre la seconda li legge a partire da addr (l'elemento nel quale scrivere). Il valore di ritorno di entrambe è il numero di blocchi su quali si è agito.

```
sizeof(tipo_dato)
```

L'istruzione sopra è utile nell'uso di fwrite e fread. Dato un tipo di dato, restituisce il numero di byte necessari per memorizzare una variabile di quel tipo.

Azioni su "testina" e buffer

```
int fseek(FILE *fp, long offset, int origin);
```

Sposta di offset byte la testina a partire da origin. Ha valore di ritorno 0 se lo spostamento è andato a buon fine.

Se offset > 0, lo spostamento è in "avanti", altrimenti è in "indietro". Origin può avere valore 0 (const. SEEK_SET), 1 (const. SEEK_CUR) o 2 (const. SEEK_END), rispettivamente indicanti l'inizio del file, la posizione attuale della testina e la fine del file.

```
void rewind(FILE *fp);
```

Sposta la testina all'inizio del file.

```
long ftell(FILE *fp);
```

Restituisce il numero del byte sul quale è attualmente posizionata la testina.

```
void fflush(FILE *fp);
```

Forza lo svuotamento dei buffer del file. Ciò è vero anche per rewind e fseek.

Per questa ragione, quando si apre un file in lettura e scrittura con lo stesso collegamento, operazioni di diversa natura vanno separate da una delle tre istruzioni di sopra. In un buffer solo non possono comparire sia istruzioni di scrittura che di lettura.

Le operazioni che agiscono sulla testina sono più efficaci sui collegamenti binari. Nei file testuali, infatti, non sempre si ha corrispondenza diretta tra byte e caratteri per varie ragioni, dunque è difficile agire "oggettivamente".

Azioni sugli errori

```
int ferror(FILE *fp);
```

Restituisce 0 se la precedente operazione effettuata sul file è andata a buon fine.

```
void clearerr(FILE *fp);
```

Elimina tutti gli errori riscontrati nelle operazioni svolte sul file.

Extra - Ordinamento e ricerca

Algoritmi di ordinamento di array

Definita la seguente funzione, gli algoritmi successivi ordinano array numerici.

```
void scambio(int *x, int *y) {
    int z= *x;
    *x = *y;
    *y = z;
} //Scambio di due variabili
```

SELECTION SORT, scorre l'array dal primo elemento e determina il valore minore al suo interno, scambiando poi il valore di esso con quello del primo elemento.

Ad ogni ciclo, dunque, è garantito che l'elemento minore dei rimanenti sia stato piazzato all'inizio dell'array. Il procedimento è poi ripetuto a partire dal secondo elemento, poi dal terzo, e via dicendo.

Codice:

```
void SelectionSort(int a[], int n) {
    int i, j, indice;
    for (i = 0; i < n-1; i++)
    {
        indice = i;
       //il primo elemento considerato è preso come ipotetico minimo
        for (j = i+1; j < n; j++)
          if (a[j] < a[min_idx])
            //se si trova un elemento minore, si il suo indice
            indice = j;
        //scambiati il primo e il minimo elemento
        scambio(&a[indice], &a[i]);
    }
}
```

BUBBLE SORT, considera gli elementi a coppie e, se quello con indice minore è più grande di quello con indice maggiore, li scambia. In questo modo, ad ogni ciclo, l'elemento più grande dei non ordinati è sicuramente posizionato in fondo. Il procedimento è ripetuto più volte, escludendo dai controlli ogni volta un elemento terminale.

Codice:

```
void BubbleSort(int a[], int n) { //n dim logica di arr[]
    int i, j;
    for (i = 0; i < n-1; i++) {
```

```
            //gli ultimi i elementi sono già in ordine
            for (j = 0; j < n-i-1; j++)  {
                  if (a[j] > a[j+1]) {
                        scambio(&a[j], &a[j+1]);
                  }
            }
      }
}
```

Algoritmi di ricerca negli array

Tramite un semplice algoritmo è possibile cercare un elemento x all'interno di un array.

Tuttavia, questa ricerca risulta inefficiente: è possibile trovare lo stesso elemento in un numero molto inferiore di passi.

```
int Search(int a[], int n, int x) { //a ha dim. logica n
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] == x) {
            return i;
        }
    }
    return -1; //x non è presente in a
}
```

La ricerca binaria è più efficiente. Considera un indice massimo e un indice minimo, guardando se l'elemento all'indice medio tra i due è maggiore, minore o uguale all'elemento da cercare. Se esso è maggiore di quello considerato, si ripete il procedimento con gli elementi di indice inferiore al medio, che saranno minori di esso, dato che l'array è ordinato. Se esso è minore dell'elemento da trovare, invece, si considera la seconda metà dell'array.

Il cambio della parte considerata dell'array è effettuato modificando indice massimo o indice minimo. La condizione di uscita, se l'elemento non è trovato, è che l'indice massimo diventi minore dell'indice minimo.

```
int BinarySearch(int a[], int primo, int ultimo, int x) {
    int medio = (primo + ultimo) / 2;
    if (ultimo >= primo) {
        if (a[mid] == x) { //x è l'elemento medio?
            return mid;
        }
        //se x < elemento medio, si considera la prima metà di a
        if (arr[mid] > x) {
```

```
            return BinarySearch(a, primo, medio - 1, x);
        }
        //altrimenti, x > elemento medio, si considera la seconda metà
        return BinarySearch(a, medio + 1, ultimo, x);
    }
    return -1; //x non è presente in a (primo > ultimo)
}
```

Extra - Tail, main() e progetti multifile

Ottimizzazione tail nelle funzioni ricorsive

Quando in un branch di una funzione ricorsiva non si fa altro che richiamare una successiva iterazione e restituirne l'output, il compilatore fa un'ottimizzazione tail. Esso elimina il record di attivazione della precedente e lo sostituisce con quello dell'iterazione successiva. Dato che nel primo non sono effettuate altre operazioni, che il valore ritornato provenga da esso o dal seguente non fa differenza.

Nelle ricorsioni normali, il risultato è calcolato "a ritroso", dall'attivazione più recente a quella più datata. Operando con logica contraria si sfrutta l'ottimizzazione tail, calcolando i risultati di iterazione in iterazione e poi restituendoli dall'ultimo record. Le funzioni ricorsive che sfruttano l'ottimizzazione tail sono riscrivibili come cicli while o for.

La funzione main()

main() è la funzione eseguita per far girare qualsiasi programma. Il suo valore di ritorno è int e comunica al SO se ci sono stati errori nell'esecuzione del codice.

Essa può avere due parametri, che entrano in gioco quando si manda in esecuzione il codice dalla linea di comando. Essi sono convenzionalmente chiamati:

```
int main(int argc, char *argv[]) { … }
```

argc è il numero di argomenti forniti nell'avvio dell'esecuzione da linea di comando, aumentato. argv è l'array di argc+1 stringhe rappresentante tutti gli argomenti forniti.

In ogni caso, argv[argc] è sempre NULL, mentre argv[0] è il percorso relativo o assoluto del programma, utilizzato per segnalare al SO di eseguirlo

Utilizzare questi parametri permette, ad esempio, di ricevere input direttamente dalla linea di comando.

Progetti multifile

In C è possibile creare un singolo eseguibile a partire da più file sorgente. Per fare ciò, deve essere creato un file progetto (solitamente gestito dall'IDE) che fornisce istruzioni al linker su come creare l'eseguibile finale. La compilazione di ogni file sorgente avviene autonomamente, ma poi sono uniti dal linker in un eseguibile solo.

Questa possibilità torna utile in molti ambiti, tra i quali spicca l'importazione di funzioni da altri file. È possibile compilare un file di sole funzioni, ma per usarle in un altro file in cui esse non sono state definite, occorre dichiararle. La dichiarazione di una funzione è:

```
<tipo-return> <nome-funzione> (<parametri-formali>);
```

Questo palesa al compilatore l'interfaccia della funzione. Mediante essa, il compilatore può cercare negli altri file la definizione della funzione e quindi eseguirla.

Una funzione può essere dichiarata più volte, ma deve essere definita una volta sola.

Il sistema standard usato per dichiarare molteplici funzioni è quello degli header. Un header è un file con estensione .h contenente le dichiarazioni delle funzioni definite in un file .c; i due file sono omonimi per convenzione. Usando #include si può “copiare” il contenuto dell’header in un file in modo efficiente, e quindi dichiarare rapidamente molte funzioni.

```
#include <libreria.h> //per le librerie ufficiali
#include “mialibreria.h” //per le librerie non ufficiali, indicate tramite il loro percorso
```

Appunti pratici - Visual Studio Community

IDE: software contenente tutti gli strumenti necessari per la realizzazione di programmi.

I suoi componenti principali sono:

- Editor di testo, per scrivere le istruzioni del file sorgente;
- Compilatore, per trasformare il file sorgente in un file oggetto;
- Debugger, per individuare e segnalare errori sintattici o semantici.

CREARE UN PROGETTO

File → Nuovo → Progetto

Seleziona "Progetto vuoto", poi nome e posizione nel file system. Il progetto risultante conterrà varie cartelle, dedicate ad elementi diversi. La cartella "File di origine" è dedicata al codice sorgente.

CREARE UN FILE .c

File di origine → Aggiungi → Nuovo elemento → File di C++

Cambia il nome da .cpp a .c per scrivere in C.

TRASFORMARE UN PROGETTO IN ESEGUIBILE

Compilazione → Compila

Crea il file oggetto usando il compilatore.

Compilazione → Compila -nomefile-

Crea l'eseguibile tramite il linker.

Compilazione → Compila soluzione

Fa entrambe le cose insieme. Mostra gli errori del compilatore e del linker senza distinzione.

LANCIARE UN PROGRAMMA

Debug → Lancia senza debugger

FILE SYSTEM

Al termine delle operazioni sopracitate, la propria soluzione comprende:

- Cartella del debugger, con file usati dal debugger e l'eseguibile;
- File Visual Studio, contenente parametri relativi al VSC;
- Cartella -nomeprogetto-, contenente il file sorgente.

In un progetto, in quanto il linker unisce i file come se fossero uno solo, l'istruzione main() si può trovare in un singolo file.

DEBUGGER

Software comunemente presente negli IDE che aiuta il programmatore a trovare e correggere errori. Gli ultimi possono riguardare:

- Sintassi, comandi scritti in modo errato, non compilati dal compilatore;
- Semantica, comandi compilati ed eseguiti correttamente, ma che non fanno ciò che dovrebbero per via di sviste.

I primi sono spesso segnalati dagli IDE nella scrittura del codice e dai compilatori nella compilazione nell'ultimo. Sono perciò facili da individuare e da correggere.

I secondi, invece, non sono segnalati e sono, dunque, frequentemente più difficili da localizzare e rettificare.

UTILIZZARE IL DEBUGGER

Per usare il debugger, il file sorgente deve già essere compilato e pronto all'esecuzione.

Cliccando a sinistra di una qualsiasi riga, si imposta in essa un Breakpoint. Consiste in un punto del programma in cui il debugger mette "in pausa" l'esecuzione dell'ultimo, prima dell'attuazione del comando della riga stessa. Col programma fermo, si possono visualizzare i valori di ogni variabile menzionata nel comando di Breakpoint.

Debug → Avvia Debug

Lancia il programma con il debugger.

Barra del debugger → Continua

Da un Breakpoint prosegue ininterrottamente l'esecuzione del programma fino alla fine o alla prossima "pausa".

Barra del debugger → Esegui istruzione (Step over)

Da un Breakpoint esegue l'istruzione attuale e si ferma alla riga seguente, come se si trattasse di un'altra "pausa". Proseguendo con questo comando si può scorrere l'intero codice riga per riga.

Barra del debugger → Esegui istruzione (Step into)

Quando si ha una funzione in un'istruzione e si è fermi su di essa, si può "entrare" nella funzione e scorrere le sue istruzioni come se fossero parte del main.

Barra del debugger → Arresta debug

Ferma le operazioni di debugging e l'esecuzione del codice. Si deve sempre usare questo comando per terminare le sessioni di debugging.