

III Fondamenti di programmazione

III.1 I linguaggi di programmazione

Per introdurre il discorso sui *linguaggi di programmazione* in ambito informatico, occorre prima richiamare alcuni concetti fondamentali dell'informatica quali quelli di *algoritmo*, *programma*, *sintassi* e *linguaggio formale*.

Un **algoritmo** è una sequenza ordinata di passi semplici che hanno lo scopo di portare a termine un compito più complesso (una ricetta da cucina, ad esempio, può essere considerata come un algoritmo che partendo da un insieme di singoli alimenti di base ed eseguendo una sequenza di passi, produce come risultato un piatto composto). In un modo più formale, possiamo quindi definire l'algoritmo come una *sequenza ordinata e finita di istruzioni che, dato uno od una serie di elementi in input, produce uno od una serie di risultati in output*.

La sequenza delle operazioni deve essere chiara, mai ambigua, deve avere un ordine ben preciso, e deve giungere a termine per ogni input.

Un esempio di algoritmo è il seguente:

1. controlla se la porta è aperta
2. nel caso che la porta sia aperta salta il passo seguente
3. apri la porta
 1. protendi il braccio
 2. afferra la maniglia
 3. ruota la mano di 30 gradi in direzione antioraria
 4. applica una pressione alla maniglia diretta di fronte a te
 5. ...
4. avanza di un metro

Una breve analisi dell'esempio sopra, porta a delineare alcune caratteristiche essenziali di un algoritmo:

- non ambiguo: le istruzioni devono essere univocamente interpretabili;
- eseguibile: ogni istruzione deve terminare in tempo finito.

Inoltre, in informatica, si richiede generalmente che un algoritmo sia finito, ovvero termini per ogni insieme di dati di ingresso.

In matematica, logica, informatica e linguistica, per **linguaggio formale** si intende un insieme di stringhe di lunghezza finita costruite sopra un alfabeto finito, cioè sopra un insieme finito di oggetti tendenzialmente semplici che vengono chiamati caratteri, simboli o lettere.

Si può parlare di linguaggio formale anche in senso non strettamente tecnico in vari contesti (scientifici, legali, amministrativi, di progetto, di normativa, ...), per intendere un linguaggio che si serve di termini e di espressioni scelte con cura maggiore di quella impiegata nel parlare comune, al fine di eliminare o ridurre ogni ambiguità di interpretazione. Un criterio per distinguere un linguaggio formale consiste nel valutare se si possa sottoporlo ed elaborazioni automatiche.

La sintassi di un linguaggio formale rappresenta la *struttura* di un linguaggio. Per esempio, Se consideriamo per un momento il linguaggio naturale (italiano) con cui comunichiamo con

i nostri simili, due frasi come “la mela mangia il bambino” e “il bambino mangia la mela” obbediscono entrambe alla sintassi dell’italiano, in quanto sono costruite secondo lo schema <soggetto> <verbo> <complemento-oggetto> a partire da parole costruite correttamente. Inoltre la sintassi determina le regole di costruzione delle *stringhe* di simboli ammesse dal linguaggio (dove per stringa si intende un’unione di simboli), senza riguardo al loro significato.

Da un punto di vista *puramente sintattico*, un **programma** è una stringa o frase appartenente al linguaggio di programmazione.

Nell’informatica, dunque un **linguaggio di programmazione** è un linguaggio formale dotato di una sintassi ben definita che viene utilizzato per scrivere programmi che realizzano algoritmi. Di conseguenza è possibile affermare che un **programma** è l’implementazione di un algoritmo in un linguaggio di programmazione comprensibile all’agente che deve eseguirlo, ovvero una sequenza di istruzioni del linguaggio.

I linguaggi di programmazione sono nati per facilitare la programmazione dei calcolatori rendendo possibile descrivere gli algoritmi e le strutture dei dati in una forma più vicina a quella del linguaggio umano scritto.

Tutti i linguaggi di programmazione esistenti possiedono (almeno) questi due concetti chiave:

Variabile: un dato o un insieme di dati, noti o ignoti, già memorizzati o da memorizzare; ad una variabile corrisponde sempre, da qualche parte, un certo numero (fisso o variabile) di locazioni di memoria che vengono *allocate*, cioè riservate, per contenere i dati stessi. Molti linguaggi inoltre attribuiscono alle variabili un **tipo**, con differenti proprietà (stringhe di testo, numeri, liste, ecc.).

Istruzione: un comando, una azione concreta, oppure una regola descrittiva: anche il concetto di istruzione è molto variabile fra classi di linguaggi differenti. A prescindere dal particolare linguaggio però, ogni volta che un’istruzione viene eseguita, lo stato interno del calcolatore (che sia lo stato reale della macchina oppure un ambiente virtuale, teorico, creato dal linguaggio) cambia.

Alcuni concetti sono poi presenti nella gran parte dei linguaggi:

Espressione: una combinazione di variabili e **costanti**, unite da **operatori**; le espressioni sono state introdotte inizialmente per rappresentare le espressioni matematiche, ma in seguito la loro funzionalità si è estesa. Una espressione viene **valutata** per produrre un valore, e la sua valutazione può produrre “effetti collaterali” sul sistema e/o sugli oggetti che vi partecipano.

Strutture di controllo, che permettono di governare il flusso dell’esecuzione del programma, alterandolo in base al risultato di una espressione (che può ridursi al contenuto di una variabile, o essere anche molto complessa). Nei linguaggi imperativi, le strutture di controllo assumono la forma di *istruzioni di flusso*.

Funzione: un frammento di codice che può essere richiamato da qualsiasi altro punto del programma.

Strutture dati, meccanismi che permettono di organizzare e gestire dati complessi.

Nel corso dell’evoluzione dell’Informatica sono stati elaborati moltissimi *linguaggi di programmazione*. I primi erano assai semplici, sia per numero di istruzioni che per complessità di elaborazione delle stesse. Erano chiamati **linguaggi macchina** poiché nascevano collegati ad un certo tipo di calcolatore e contenevano messaggi operativi (istruzioni) tipici di quella macchina. In un secondo tempo vennero introdotti dei linguaggi chiamati **ad alto livello**, i quali oltre a tendere

ad avvicinarsi al linguaggio umano, avevano la caratteristica di essere svincolati dalla macchina che li utilizzava, nel senso che calcolatori di tipo completamente diverso potevano, sotto certe condizioni, capire il linguaggio ed eseguire le istruzioni con esso scritte.

È possibile quindi suddividere i linguaggi di programmazione in livelli, a seconda di quanto essi sono vicini al linguaggio macchina.

Basso livello:

1. **Linguaggio macchina** Le operazioni disponibili sono quelle direttamente fornite dall'*hardware*; ogni operazione è codificata da una sequenza di bit; ogni dato è indicato dall'indirizzo binario della parola di memoria in cui è memorizzato. Ogni indirizzo è espresso in modo assoluto (rispetto a tutta la memoria disponibile). Un programma è una sequenza di bit, che viene direttamente interpretata dall'*hardware*.

2. **Linguaggio assembler** (o assemblativo) Le operazioni sono quelle direttamente fornite dall'*hardware*, ma sono indicate da nomi convenzionali (mnemonici); i dati sono indicati da nomi, che corrispondono a indirizzi. Gli indirizzi sono espressi in modo relativo rispetto all'inizio del programma, permettendo così più semplici modifiche. Il programma, per essere eseguito, viene tradotto in linguaggio macchina da un programma traduttore detto **assemblatore**.

Linguaggi procedurali Le operazioni disponibili sono ampie e non legate a quelle fornite dall'*hardware*. I dati sono indicati in modo totalmente indipendente dalla loro memorizzazione. Si tratta di linguaggi progettati affinché la scrittura dei programmi sia semplice, elegante e, dunque, di facile comprensione e verifica. Per essere eseguito, un programma deve essere tradotto in linguaggio macchina da un programma traduttore detto **compilatore**, oppure deve essere interpretato (cioè eseguito da un altro programma, l'interprete).



Linguaggi procedurali di rilievo sono (o sono stati) FORTRAN, COBOL, BASIC,

Pascal, C. Tra i linguaggi procedurali alcuni sono detti orientati agli oggetti; tra questi ricordiamo C++ e Java.

In generale programmare in un dato linguaggio di programmazione significa essenzialmente scrivere uno o più semplici file di testo ASCII, chiamato **codice sorgente**. I font, i colori e in generale l'aspetto grafico sono irrilevanti ai fini della programmazione in sé: per questo i

programmatori non usano programmi di videoscrittura ma degli *editor* di testo che invece offrono funzioni avanzate di trattamento testi (espressioni regolari, sostituzioni condizionali e ricerche su file multipli, possibilità di richiamare strumenti esterni ecc). Se un dato editor è in grado di lavorare a stretto contatto con gli altri strumenti di lavoro (compilatore, linker, interprete... : *vedi più avanti*) allora più che di semplice editor si parla di *ambiente di sviluppo integrato*.

Il codice sorgente, contenente le istruzioni da eseguire e (spesso) alcuni dati noti e costanti viene compilato, cioè tradotto in istruzioni di linguaggio macchina da un programma **compilatore**: il risultato è un file binario eseguibile che non ha bisogno di altri programmi per andare in esecuzione, ed è anche molto più veloce di un programma interpretato. La compilazione è la norma per tutti i linguaggi di programmazione di uso generale.

La *compilazione* è il processo per cui il programma, scritto in un linguaggio di programmazione ad alto livello, viene tradotto in un **codice eseguibile** per mezzo di un altro programma detto appunto **compilatore**.

La compilazione offre numerosi vantaggi, primo fra tutti il fatto di ottenere eseguibili velocissimi adattando vari parametri di questa fase all'hardware a disposizione; ma ha lo svantaggio principale nel fatto che bisogna ricompilare ogni volta che si cambia sistema operativo e hardware su cui ci si basa. Risulta evidente che occorre fare n ricompilazioni per ogni modifica se si vuole rendere disponibile il programma su più piattaforme.

Se il programma, come spesso accade, usa delle **librerie** (insieme di funzioni di uso comune, predisposte per essere collegate ad un programma), o è composto da più parti di software (moduli software), questi devono essere collegati tra loro. Lo strumento che effettua questa operazione è detto appunto **linker**, e si occupa principalmente di risolvere le interconnessioni tra i diversi moduli.

Esistono principalmente due tipi differenti di collegamento: **dinamico** e **statico**.

collegamento statico

Tutti i moduli del programma e le librerie utilizzate vengono incluse nel codice eseguibile, che risulta grande, ma contiene tutto quanto necessario per la sua esecuzione. Se si rende necessaria una modifica ad una delle librerie, per correggere un errore o un problema di sicurezza, tutti i programmi che le usano con collegamento statico devono essere ricollegati con le nuove versioni delle librerie.

collegamento dinamico

Le librerie utilizzate sono caricate dal sistema operativo quando necessario (*linking dinamico*; le librerie esterne sono dette DLL, *Dinamyc Linking Libraries*). L'eseguibile risultante è più compatto, ma dipende dalla presenza delle librerie utilizzate nel sistema operativo per poter essere eseguito.

In questo modo, le librerie possono essere aggiornate una sola volta a livello di sistema operativo, senza necessità di ricollegare i programmi. Diventa anche possibile usare diverse versioni della stessa libreria, o usare librerie personalizzate con caratteristiche specifiche per il particolare host.

Nella realizzazione di un progetto software complesso, può succedere che alcune parti del programma vengano realizzate come librerie, per comodità di manutenzione o per poterle usare in diversi programmi che fanno parte dello stesso progetto.

La complicazione aggiunta è che quando si installa un programma con collegamento dinamico è necessario verificare la presenza delle librerie che utilizza, ed eventualmente installare anche

queste. I sistemi di package management, che si occupano di installare i programmi su un sistema operativo, di solito tengono traccia automaticamente di queste dipendenze.

In genere si preferisce il collegamento dinamico, in modo da creare programmi piccoli, assumendo che le DLL necessarie siano già presenti nel sistema, o talvolta distribuendole insieme al programma.

III.2 I diagrammi di flusso

Un computer non è altro che un elaboratore in grado di eseguire un determinato set di istruzioni. Per poter svolgere qualsiasi compito il calcolatore ha bisogno di qualcosa che gli dica passo passo cosa deve fare, o meglio, quale delle istruzioni che conosce deve eseguire in un determinato momento.

Un software non è altro che un compito da svolgere scritto utilizzando il set di istruzioni primitive che il calcolatore è in grado di eseguire. Quindi una volta dato un problema è necessario “tradurlo” in termini di istruzioni per poterlo fare risolvere da un elaboratore. Descrivere la soluzione di un problema o lo svolgimento di un compito sotto forma di passi discreti, eseguibili e non ambigui significa costruire un algoritmo.

Come già affermato in precedenza, le proprietà fondamentali di un algoritmo sono: la *non-ambiguità* (il procedimento deve essere interpretabile in modo univoco da chi lo deve eseguire), *eseguibilità* (ogni istruzione dell'algoritmo deve poter essere eseguita senza ambiguità da parte di un esecutore reale o ideale), *finitezza* (sia il numero di istruzioni che compongono l'algoritmo, che il tempo di esecuzione devono essere finiti).

Ogni *calcolatore* funziona secondo un **algoritmo di base** secondo della forma:

1. finché non trovi un'istruzione di **alt** fai:
2. preleva un'istruzione dalla memoria
3. decodifica l'istruzione
4. esegui l'istruzione

Questo algoritmo è intrinseco nella struttura della CPU e fa in modo che una volta inserito un programma nella memoria, questo possa essere letto istruzione per istruzione ed eseguito. Tale caratteristica sta alla base della programmabilità degli operatori e permette la totale generalità degli elaboratori.

Al fine di rappresentare gli algoritmi in modo comprensibile per il programmatore, vengono principalmente utilizzate due tecniche:

1. rappresentazione mediante **pseudolinguaggio o pseudocodice**
2. rappresentazione mediante **diagramma di flusso (flowchart)**

Più precisamente:

Pseudolinguaggio: linguaggio informale (ma non ambiguo) per la descrizione delle istruzioni.

Ogni riga rappresenta un' istruzione. Se non diversamente specificato, le righe si leggono dall' alto verso il basso

Diagramma di flusso: rappresentazione grafica in cui ogni istruzione è descritta all' interno di un riquadro e l'ordine di esecuzione delle istruzioni è indicato da frecce di flusso tra i riquadri.

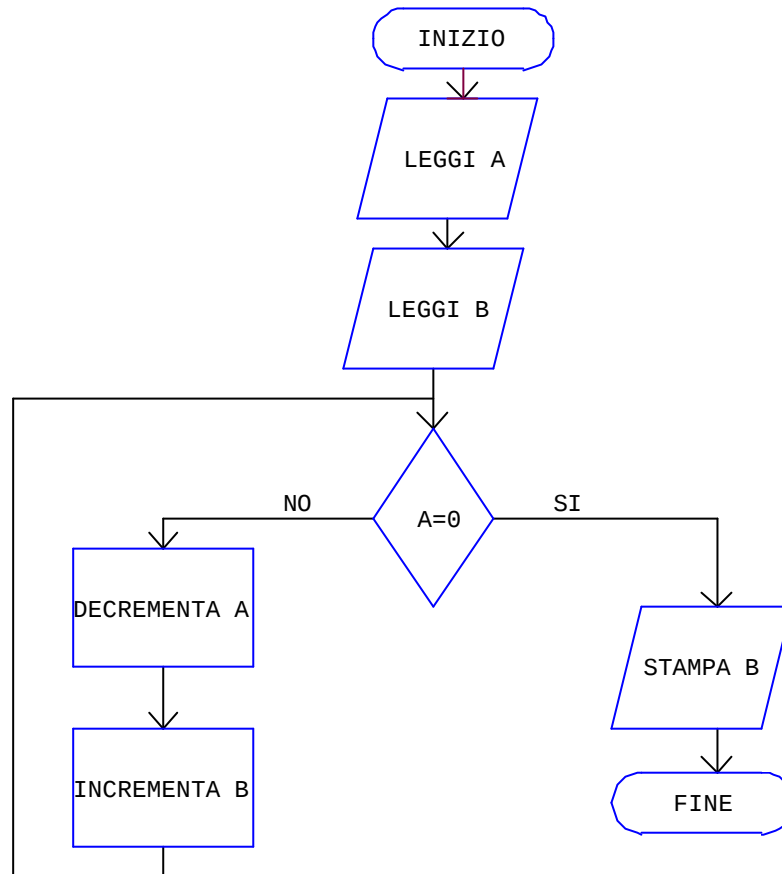
Come esempio si consideri l'algoritmo che esegua la somma di due numeri naturali utilizzando solo incrementi e decrementi.

Nel caso del linguaggio informale (pseudolinguaggio) si ha:

1. Leggi A
2. Leggi B
3. Se A=0 vai all' istruzione 7

4. Decrementa A
5. Incrementa B
6. Vai all'istruzione 3
7. Stampa B

Nel caso del diagramma di flusso relativo allo stesso algoritmo, si ha:



Definizione di uno pseudolinguaggio. Uno *pseudolinguaggio* si basa sul concetto di azione “primitiva”. L’ utilizzo di azioni primitive per descrivere un procedimento fa sì che questo possa essere compreso senza alcuna ambiguità. In altre parole, nel caso in cui si descriva una azione in modo superficiale o si utilizzino vocaboli dal significato non univoco una frase potrebbe essere interpretata in diversi modi. Si prenda, per

esempio, la frase “andare a lezione può essere irritante”. Un lettore che non conosca gli argomenti della lezione, il docente o il luogo in cui la lezione si svolge potrebbe interpretare la frase come: “La lezione causa irritazione” oppure “il tragitto per arrivare dove si svolge la lezione è irritante” oppure in entrambi i modi.

Per evitare un’ interpretazione ambigua, quindi, viene definito un set di attività primitive, note agli interlocutori(uomo – macchina) e un insieme di regole per utilizzare le primitive. Tutto questo, set di primitive più insieme di regole, costituisce un normale **linguaggio di programmazione**. Ogni pseudolinguaggio deve necessariamente definire un insieme minimo di elementi, quali:

1. Istruzioni di inizio e di fine **START** e **END**

Sono i punti d’ ingresso e di uscita del flusso di esecuzione

2. Istruzioni di assegnamento **nome ? espressione**

L' esecuzione di un' istruzione di assegnamento avviene in due fasi: la valutazione dell' espressione (utilizzando i valori correnti delle eventuali variabili che in essa compaiono) e l' aggiornamento del valore della variabile a sinistra del segno ?

3. Scelta tra due possibili attività **IF(condizione) THEN (attività 1) ELSE (attività 2)**

Biforcazione del flusso in due percorsi alternativi (mutuamente esclusivi) la cui esecuzione è condizionata al verificarsi di una condizione. L' esecuzione di un' istruzione condizionale comporta innanzitutto la valutazione della condizione, e quindi l' esecuzione delle istruzioni che compongono uno dei due cammini.

4. Strutture iterative **WHILE(condizione) DO(azione) END WHILE**

Esecuzione ripetuta di una o più istruzioni. La ripetizione dipende dall' esito di un controllo. È fondamentale che l' esito del controllo dipenda da variabili il cui valore può essere modificato dall' esecuzione delle istruzioni del ciclo. In caso contrario il ciclo verrebbe eseguito o mai (risultando inutile) o all' infinito (violando la definizione di algoritmo). Le variabili da cui dipende il controllo sono dette variabili di controllo del ciclo. Ogni ciclo è composto da 4 elementi: l' inizializzazione delle variabili di controllo, il controllo della condizione da cui dipende l' iterazione, il corpo del ciclo (sequenza delle istruzioni da eseguire ciclicamente) e l' aggiornamento delle variabili di controllo.

5. Istruzione di salto **GOTO(riga destinazione)**

L' istruzione di salto sposta il flusso di esecuzione in un particolare punto dell' algoritmo. La riga di destinazione rappresenta, appunto, il punto in cui l' esecuzione riprenderà dopo il goto.

6. Procedure **PROCEDURE** NomeProcedura (

Istruzione 1

Istruzione 2

Istruzione 3

)

Unità di programma utilizzabili attraverso invocazione da un qualsiasi punto del codice. Tutti i linguaggi di programmazione utilizzano questa strategia per rendere più leggibile il codice. Sinonimi di procedura sono: funzione, metodo, subroutine, sottoprogramma ecc. Esempio di utilizzo:

```
procedure SalutaTutti (
```

```
  contatore ? 24
```

```
  WHILE(contatore > 0) DO (
```

```
    Stampa(" Saluti ")
```

```
    contatore ? contatore - 1
```

```
  )END WHILE
```

```
)
```

1. START

2. Stampa("Ora invoco la procedura")
3. SalutaTutti
4. END

Nel caso dei diagrammi flowchart, il flusso inizia e termina in blocchi fittizi (detti inizio e fine) di forma generalmente arrotondata.

I riquadri del diagramma di flusso possono avere forme diverse (convenzionali) per rappresentare graficamente il tipo di istruzione che contengono:

- rombi con una freccia entrante e due uscenti per indicare diramazioni di flusso condizionate;
- rettangoli per indicare qualsiasi istruzione che non sia una condizione o un'operazione di I/O (lettura/scrittura);
- ellissi per indicare inizio e fine;
- parallelogrammi per indicare le operazioni di input/output

Esecuzione di un algoritmo. Un algoritmo può essere visto come un risolutore di problemi che acquisisce dati d'ingresso e produce risultati.

Generalmente, uno stesso algoritmo può essere applicato più volte per risolvere lo stesso tipo di problema con dati d'ingresso diversi, ovvero istanze diverse dello stesso problema. Chiamiamo esecuzione di un algoritmo la sua applicazione a determinati dati d'ingresso. Un algoritmo è detto lineare se l'esecuzione segue un flusso lineare dalla prima all'ultima istruzione, è detto non lineare altrimenti. (Es: L' algoritmo per la somma di due numeri naturali dell' esempio visto in precedenza è non lineare). Chiamiamo **passo d' esecuzione** l'esecuzione di un' istruzione. Il numero di passi d' esecuzione è generalmente diverso dal numero di istruzioni utilizzate per descrivere l' algoritmo e dipende dai dati d' ingresso.

Le grandezze sulle quali un algoritmo va ad agire sono:

- **costante:** quantità nota a priori il cui valore non dipende dai dati d'ingresso e non cambia durante l'esecuzione dell'algoritmo (Es. lo 0 dell' istruzione 4 dell' algoritmo per la somma di due numeri naturali descritto precedentemente).
- **variabile:** nome simbolico cui è assegnato un valore che può dipendere dai dati d'ingresso e cambiare durante l'esecuzione dell'algoritmo (Es: A e B nell' dell' algoritmo per la somma di due numeri naturali descritto precedentemente).
- **espressione:** sequenza di variabili, costanti o espressioni combinate tra loro mediante operatori (Es: $a+b*4$). Nella valutazione di un' espressione si sostituisce ad ogni variabile il suo valore corrente e si seguono le operazioni secondo regole di precedenza prestabilite (o indicate da parentesi).

III.3 Dal primo linguaggio di programmazione al C++

Il primo linguaggio di programmazione della storia è a rigor di termini il Plankalkül di Konrad Zuse, sviluppato da lui nella svizzera neutrale durante la II guerra mondiale e pubblicato nel 1946; ma non venne mai realmente usato per programmare.

La programmazione dei primi elaboratori veniva fatta invece in **Shortcode**, da cui poi si è evoluto l'assembly o assembler, che costituisce una rappresentazione simbolica del **linguaggio macchina**. (Il *linguaggio macchina* è il linguaggio di programmazione che viene compreso dai componenti hardware del computer).

La sola forma di controllo di flusso è l'istruzione di salto condizionato, che porta a scrivere

programmi molto difficili da seguire logicamente per via dei continui salti da un punto all'altro del codice.

La maggior parte dei linguaggi di programmazione successivi cercarono di astrarsi da tale livello basilare, dando la possibilità di rappresentare strutture dati e strutture di controllo più generali e più vicine alla maniera (umana) di rappresentare i termini dei problemi per i quali ci si prefigge di scrivere programmi. Tra i primi linguaggi ad alto livello a raggiungere una certa popolarità ci fu il *Fortran*, creato nel 1957 da John Backus, da cui derivò successivamente il **BASIC** (1964): oltre al salto condizionato, reso con l'istruzione IF, questa nuova generazione di linguaggi introduce nuove strutture di controllo di flusso come i cicli WHILE e FOR e le istruzioni CASE e SWITCH: in questo modo diminuisce molto il ricorso alle istruzioni di salto (GOTO e IF), cosa che rende il codice più chiaro ed elegante, e quindi più facilmente manutenibile.

Dopo la comparsa del Fortran nacquero una serie di altri linguaggi di programmazione storici, che implementarono una serie di idee e paradigmi innovativi: i più importanti sono l'*ALGOL* (1960) e il *Lisp* (1959). Tutti i linguaggi di programmazione oggi esistenti possono essere considerati discendenti da uno o più di questi primi linguaggi, di cui mutuano molti concetti di base; l'ultimo grande progenitore dei linguaggi moderni fu il Simula (1967), che introdusse per primo il concetto (allora appena abbozzato) di *oggetto* software.

Nel 1970 Niklaus Wirth pubblica il **Pascal**, il primo linguaggio strutturato, a scopo didattico; nel 1972 dal BCPL nascono prima il **B** (rapidamente dimenticato) e poi il **C**, che invece fu fin dall'inizio un grande successo. Nello stesso anno compare anche il Prolog, finora unico esempio di linguaggio dichiarativo, che pur non essendo utilizzato per il normale lavoro di programmazione a causa della sua inefficienza rappresenta una possibilità teorica estremamente affascinante.

Con i primi mini e microcomputer e le ricerche a Palo Alto, nel 1983 vede la luce Smalltalk, il primo linguaggio realmente e completamente ad oggetti, che si ispira al Simula e al Lisp: anche lui poco o nulla utilizzato in pratica, ha però una influenza enorme sulla teoria dei linguaggi di programmazione. Suo derivato diretto è l'Eiffel (1986), ma influenza profondamente anche il C++ (che esce nello stesso anno dell'Eiffel) e successivamente **Java**, classe 1995.

Il C++ è un *linguaggio di programmazione orientato agli oggetti* sviluppato da Bjarne Stroustrup ai Bell Labs negli anni '80 e rilasciato nel 1986. È basato sul linguaggio C. Mentre la maggior parte del codice sorgente C è anche valido per il C++, il C non è un sottoinsieme del C++ nel senso più stretto della parola. Il C++ fu standardizzato nel 1998 (ISO/IEC 14882-1998 "Information Technology - Programming Languages - C++").

In aggiunta al supporto alla programmazione orientata agli oggetti, il C++ è distinto dal C per il suo supporto alla programmazione generica e alla metaprogrammazione attraverso l'uso dei templates.

Il C++ è molto diffuso e apprezzato, ma raramente è usato al massimo delle sue potenzialità: la semantica del C++ è molto ricca di dettagli e sfumature che condizionano il comportamento del codice, e che molto spesso i compilatori implementano in maniera scorretta o incompleta: molte delle caratteristiche dello standard ISO del linguaggio non sono ancora implementate nei compilatori attuali, anche se la situazione sta migliorando lentamente. La grande ricchezza semantica del C++, insieme alle librerie che lo accompagnano, lo rende un linguaggio estremamente espressivo e potente, ma che richiede molto tempo per venire appreso e padroneggiato completamente. Inoltre a causa della variabilità del comportamento dei compilatori nel maneggiare le funzioni più avanzate del linguaggio, i programmi C++ che scelgono di farne uso

si rivolgono ad un'architettura (processore, sistema operativo e compilatore) particolare sacrificando la portabilità su altre piattaforme.

Il C++ ha una libreria standard, in modo simile al C. Oltre a questa, particolare importanza nel C++ ricopre la STL, **Standard Template Library**. Si tratta di una libreria che definisce una serie di template generici per strutture dati comuni, come vettori, code, array associativi, e così via. La programmazione ne risulta molto semplificata, al prezzo di un gran lavoro del compilatore per interpretare i complessi template.

Il nome del linguaggio, scritto "C++" e pronunciato "si plas plas", rappresenta la sua evoluzione dal C. Fu suggerito da Rick Mascitti nella metà del 1983, quando il linguaggio veniva usato per la prima volta al di fuori dei centri di ricerca. All'inizio ci si riferiva al linguaggio semplicemente come "C con classi", direttamente sviluppato con CFront.

Il nome è un gioco di parole con un costrutto del C (dove il doppio segno più è un operatore che ha l'effetto di incrementare il valore di una variabile) insieme con la comune convenzione di aggiungere un segno più per indicare una versione potenziata.

In definitiva, il C++ è un linguaggio di programmazione "all purpose", ovvero utilizzabile per la realizzazione di qualsiasi tipo di applicazione, da quelle real time a quelle che operano **su basi di dati**, da applicazioni per utenti finali a sistemi operativi. Ciò tuttavia non implica che qualsiasi applicazione debba essere realizzata in C++, esistono moltissimi linguaggi di programmazione alcuni dei quali altamente specializzati per compiti precisi e che quindi possono essere in molti casi una scelta più conveniente.

Negli ultimi anni il C++ ha ottenuto un notevole successo per diversi motivi:

- Conserva una compatibilità quasi assoluta con il suo più diretto antenato, il C, da cui eredita la sintassi e la semantica per tutti i costrutti comuni, oltre alla notevole flessibilità e potenza;
- Permette di realizzare qualsiasi cosa fattibile in C senza alcun overhead addizionale;
- Estende le caratteristiche del C, rimediando almeno in parte alle carenze del suo predecessore. In particolare l'introduzione di meccanismi quali i *Template* e le *Classi* rende il C++ un linguaggio multiparadigma (con particolare predilezione per il paradigma ad oggetti e la programmazione generica);
- Possibilità di portare facilmente le applicazioni verso altri sistemi;

Comunque il C++ presenta anche degli aspetti negativi (come ogni linguaggio), in parte ereditate dal C:

- La potenza e la flessibilità tipiche del C e del C++ non sono gratuite. Se da una parte è vero che è possibile ottenere applicazioni mediamente assai più efficienti rispetto ad altri linguaggi, e anche vero che tutto questo è ottenuto lasciando in mano al programmatore molti dettagli e compiti che negli altri linguaggi sono svolti dal compilatore; è quindi necessario un maggiore lavoro in fase di progettazione e una maggiore attenzione ai particolari in fase di realizzazione, pena una valanga di errori spesso subdoli e difficili da individuare che possono far levitare drasticamente i costi di produzione;

- Il compilatore e il linker del C++ soffrono di problemi relativi all'ottimizzazione del codice dovuti alla falsa assunzione che programmi C e C++ abbiano comportamenti simili a run time: il compilatore nella stragrande maggioranza dei casi si limita ad eseguire le ottimizzazioni tradizionali, sostanzialmente valide in linguaggi come il C, ma spesso inadatte a linguaggi pesantemente basati sulla programmazione ad oggetti; il linker poi da parte sua non è cambiato molto e non esegue alcune ottimizzazioni che non sono fattibili a compile-time.
La sempre maggiore diffusione del C++ sta comunque cambiando questa situazione ed è prevedibile nei prossimi anni una sostanziale evoluzione di compilatori e linker, grazie anche alla recente adozione di uno standard.