

# Programmazione Strutturata in C++

**Alessandro Armando, Enrico Giunchiglia &  
Roberto Sebastiani**

D.I.S.T. - Università degli Studi di Genova

D.I.S.I. - Università degli Studi di Trento

# Indice

1. Modello di un calcolatore

2. Sviluppo di un programma

3. Sintassi del C++

4. Variabili, Costanti e Tipi

5. Puntatori e riferimenti

6. Istruzioni semplici

7. Ingresso e uscita dei dati

8. Istruzioni strutturate

9. Funzioni

10. Array

11. Strutture e unioni

12. Gestione dinamica della memoria

13. Strutture Dati e Algoritmi

14. Visibilità e moduli

15. Astrazione dei dati

# Gerarchia di astrazioni

- Un computer, con i programmi ad esso associati, presenta una gerarchia di ***livelli di astrazione***, chiamate ***macchine virtuali***
- Ogni livello, ad eccezione del più basso, è realizzato **traducendo o interpretando** le sue istruzioni mediante le ***primitive*** fornite dai livelli inferiori

## LIVELLO   ASTRAZIONE

- 6   Programma applicativo
- 5   Linguaggio di programmazione
- 4   Linguaggio assemblativo
- 3   Nucleo del sistema operativo
- 2   Linguaggio macchina
- 1   Microprogramma
- 0   Logica digitale

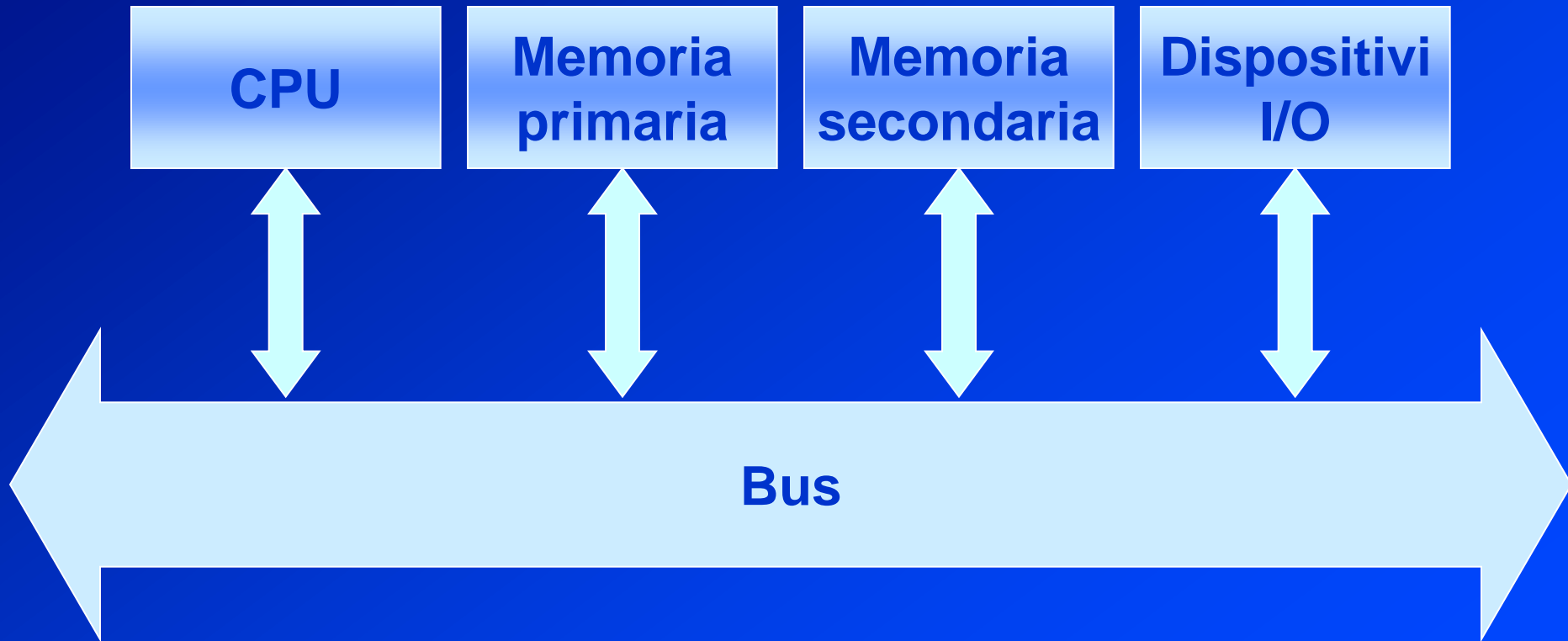
# Livelli di astrazione (1)

- **Logica digitale:** è un'astrazione della circuiteria elettronica, realizzata mediante circuiti detti *porte*, con uno o più *ingressi digitali* (rappresentati da 0 e 1) e che producono in uscita una *funzione logica* dei propri dati in ingresso (es. AND o OR)
- **Microprogramma:** è il primo livello costituito in modo prevalente da *linguaggio*, non è posseduto da tutti i computer, si tratta di un *insieme di passi* usati per realizzare le istruzioni del linguaggio macchina
- **Linguaggio macchina:** è costituito da istruzioni primitive sufficienti a eseguire qualsiasi programma o applicazione dei livelli più alti

## Livelli di astrazione (2)

- **Nucleo del sistema operativo:** ha le primitive per *ordinare* e *allocare* le *risorse* del calcolatore per i diversi programmi che vengono eseguiti
- **Linguaggio assembler:** è una *rappresentazione simbolica* delle istruzioni dei livelli inferiori. Un programma in linguaggio assembler viene tradotto in istruzioni di livello inferiore mediante un traduttore, detto *assemblatore*
- **Linguaggio di programmazione:** consente una più rapida risoluzione di problemi di quanto consenta il linguaggio assembler. Es: C, C++, JAVA,... La traduzione viene effettuata dal *compilatore*

# Struttura del computer



## Il Bus

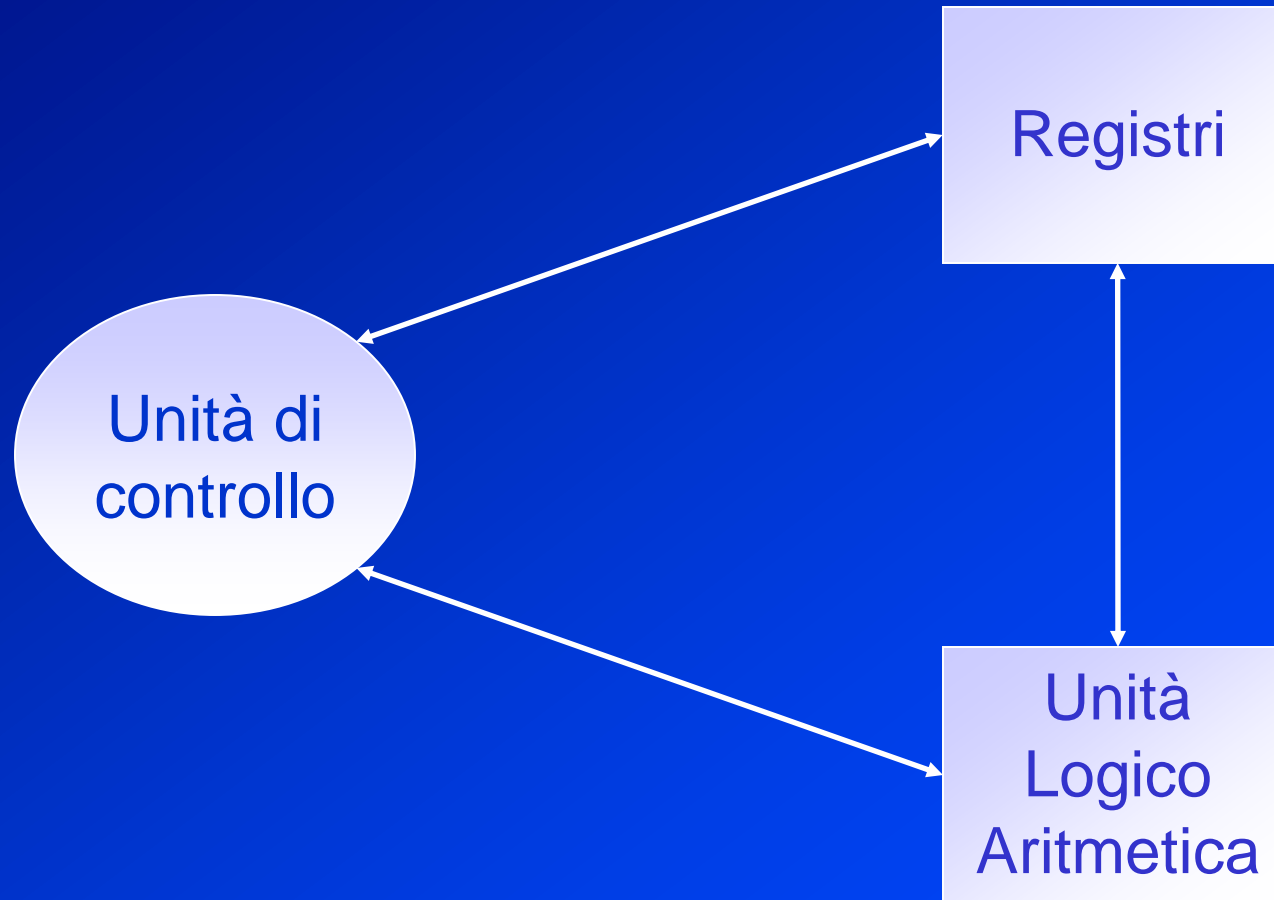
- Il **bus** è lo strumento centrale di **comunicazione** del computer
- I dispositivi che vi sono collegati scambiano istruzioni **attraverso** il **bus**
- A differenza di collegamenti diretti, detti “punto-punto”, consente una maggiore **configurabilità** e **affidabilità**
- Il clock di un bus esprime la velocità con cui vengono scambiati i dati (1-3GHz)

# Unità centrale di elaborazione (CPU)

- L'unità centrale di elaborazione (Central Processing Unit) è un insieme di circuiti, detto microprocessore, che controlla l'attività centrale del computer
- Il clock di una CPU, misurato in GHz (ad esempio 3 GHz), è la frequenza con cui vengono eseguite le istruzioni elementari
- Una delle sue funzioni principali è quella di accedere alle istruzioni della memoria principale, decodificarle ed eseguirle
- Esempi di CPU: Intel Pentium o XEON, ...



# Schema della CPU



# Elementi della CPU

L'unità centrale di elaborazione è costituita da:

- un insieme di ***registri*** che sono degli spazi di memorizzazione accessibili ad una velocità superiore di quella della memoria principale
- una ***unità logico aritmetica*** (ALU) che esegue operazioni aritmetiche, logiche e confronti sui dati della memoria principale o dei registri
- una ***unità di controllo*** che esegue le istruzioni secondo quello che viene detto **ciclo accesso-decodifica-esecuzione**

# Ciclo accesso-decodifica-esecuzione

Si tratta di una sequenza di passi nella quale:

1. si accede all'istruzione successiva portandola dalla memoria nell'apposito registro istruzione
2. si decodifica il tipo dell'istruzione
3. si individuano i dati usati dall'istruzione, che vengono portati negli opportuni registri
4. si esegue l'istruzione

# Memoria principale (1)

- La memoria principale è detta RAM (Random Access Memory) ed è una memoria volatile che consente un rapido accesso ai suoi dati
- Allo spegnimento del computer le informazioni contenuti in tale memoria vanno perse
- Contiene il codice del programma in esecuzione e i valori dei dati che questo usa
- Dimensioni tipiche delle memorie RAM di un Personal Computer sono, ad esempio, 2-8 GByte

## Memoria principale (2)

- la memoria può essere vista come un insieme di **celle** contenente dati e aventi un univoco **indirizzo** che ne consente l'individuazione
- in ogni cella è contenuta una sequenza di byte, ad esempio quattro, detta **parola**
- dato un indirizzo di memoria, le operazioni effettuabili sono:
  - **lettura** della corrispondente parola per il trasferimento nel Bus
  - **scrittura** della parola proveniente dal Bus nella corrispondente cella di memoria

# Memoria secondaria

- La memoria secondaria è un dispositivo, o un insieme di dispositivi, capace di contenere molte più informazioni della memoria principale, a discapito della velocità
- Le informazioni contenute nella memoria secondaria sono conservate in maniera permanente anche quando il computer viene spento
- Esempi di dispositivi di memoria secondaria sono: dischi rigidi (hard disk), CD-Rom, DVD, memory sticks,...

# Indice

1. Modello di un calcolatore

2. Sviluppo di un programma

3. Sintassi del C++

4. Variabili, Costanti e Tipi

5. Puntatori e riferimenti

6. Istruzioni semplici

7. Ingresso e uscita dei dati

8. Istruzioni strutturate

9. Funzioni

10. Array

11. Strutture e unioni

12. Gestione dinamica della memoria

13. Strutture Dati e Algoritmi

14. Visibilità e moduli

15. Astrazione dei dati

# Scrittura di un programma

- scrittura in un file chiamato *sorgente*
- creazione e modifica tramite opportuno strumento chiamato *editor* (ad esempio **emacs** di linux)
- è consigliabile dare al nome del file sorgente l'estensione **cc** (**nomefile.cc**), oppure **cpp** (**nomefile.cpp**)



# Esempio

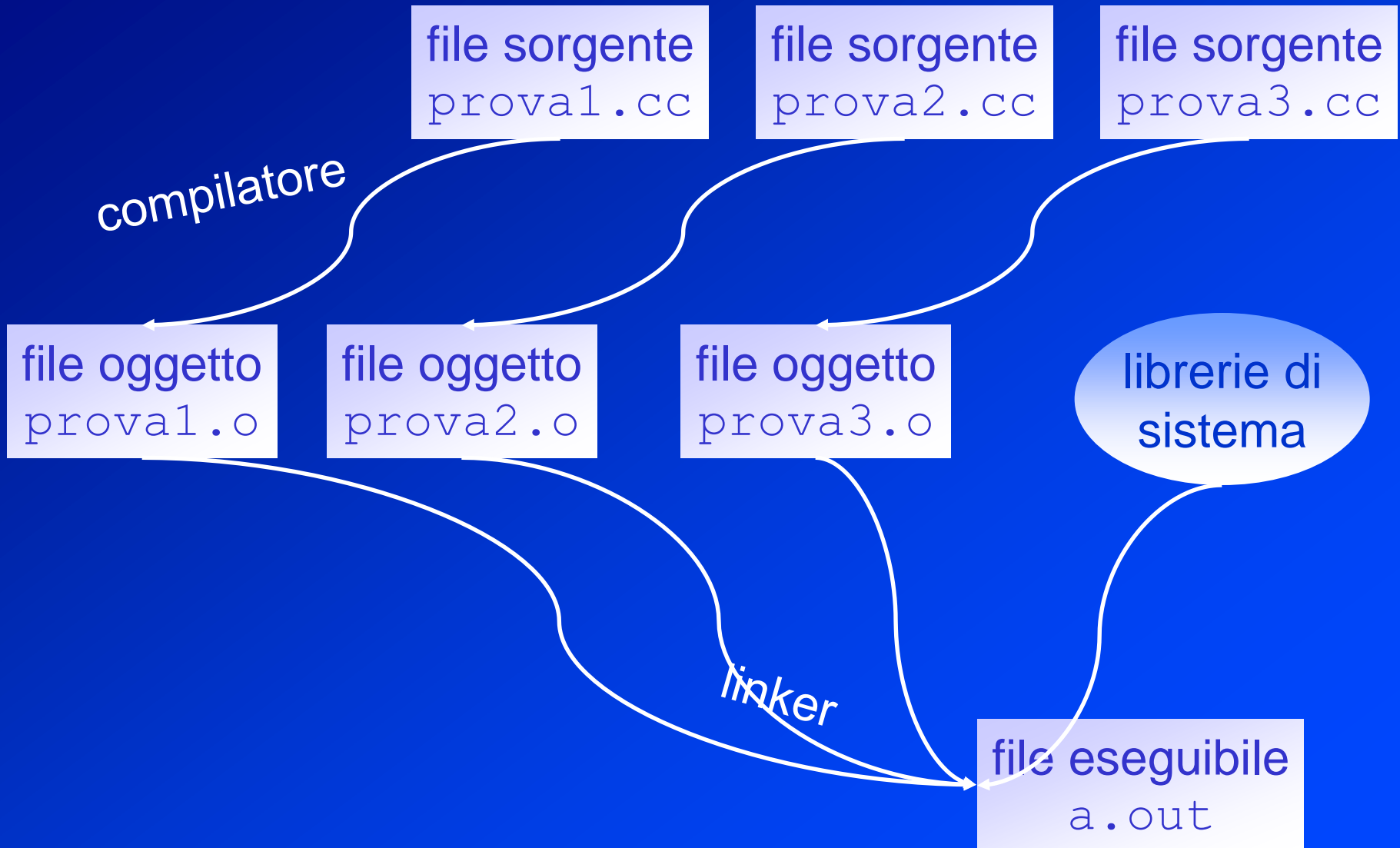
```
// Questo e' il primo programma del corso di C++
#include <iostream> // inclusione di file
int fact (int n) {    // definizione funzione fact
    int i, v;          // dichiarazione di variabili
    v=1;               // inizializzazione
    for (i=1; i<=n; i++) // ciclo
        v = v*i;       // corpo del ciclo
    return v;          // valore di ritorno
}

int main() { // definizione procedura princ.
    int x;
    cout << "Dammi un numero: "; // output
    cin >> x;                      // input
    cout << " Il fattoriale di " << x << " e': "
    << fact(x);                    // invocazione di fact
    return 0;
}
```

# Compilazione di un programma (1)



# Compilazione di un programma (2)



# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Elementi di base: sequenze di parole

Le parole possono essere costituite da

- lettere

\_ a b c d e f g h i j k l m n o p q r s  
t u v w x y z A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z

- cifre (digit)

0 1 2 3 4 5 6 7 8 9

- caratteri speciali

! % ^ & \* ( ) - + = { } | ~ [ ] \ ; `  
: " < > ? , . /

# Elementi di base: spaziature

Le spaziature possono essere costituite da

- spazi
- tabulazioni
- caratteri di nuova riga
- commenti
  - racchiusi fra `\/*'` e `\*/'`  
esempio: `/* Questo commento e'`  
`su piu' righe */`
  - tra `//` e fine riga  
esempio: `// su una sola riga`

# Identificatori (*Identifiers*)

- le entità di un programma C++ devono avere dei nomi che ne consentano l'individuazione
- nel caso più semplice i nomi sono degli *identificatori* liberamente scelti
- un identificatore è una parola iniziante con una lettera

## Note:

- il carattere '\_' è una lettera
- il C++ distingue tra maiuscole e minuscole (si dice che è *case sensitive*) es. **F**act è diverso da **f**act

# Parole chiave (*keywords*)

Le parole chiave del C++ sono un insieme di simboli il cui significato è stabilito dal linguaggio e non può essere ridefinito in quanto il loro uso è riservato.

asm	auto	break	case	catch
char	class	const	continue	default
delete	do	double	else	enum
extern	float	for	friend	goto
if	inline	int	long	new
operator	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	throw
try	typedef	union	virtual	void
volatile	while			



# Espressioni letterali

- le ***espressioni letterali*** denotano valori costanti, spesso sono chiamati solo *letterali* (o *costanti* senza nome)
- possono essere
  - costanti carattere (denotano singoli caratteri)
  - costanti stringa (denotano sequenze di caratteri)
  - costanti numeriche intere
  - costanti numeriche reali

# Rappresentazione costanti carattere

- una costante carattere viene rappresentata racchiudendo il corrispondente carattere fra apici: per esempio la costante ' a ' rappresenta il carattere 'a'
- i caratteri di controllo vengono rappresentati da combinazioni speciali dette **sequenze di escape** che iniziano con una barra invertita (backslash '\')

# Sequenze di escape

Nome	Abbreviazione	Sequenza di escape
nuova riga	NL (LF)	<code>\n</code>
tabulazione orizzontale	HT	<code>\t</code>
tabulazione verticale	VT	<code>\v</code>
spazio indietro	BS	<code>\b</code>
ritorno carrello	CR	<code>\r</code>
avanzamento modulo	FF	<code>\f</code>
segnale acustico	BEL	<code>\a</code>
barra invertita	<code>\</code>	<code>\\</code>
apice	<code>'</code>	<code>\'</code>
virgolette	<code>"</code>	<code>\"</code>

# Rappresentazione costanti stringa

- una costante stringa è una lista di caratteri compresa fra una coppia di virgolette "
- esempio: `"Hello!"`
- possono esservi anche delle sequenze di escape: per esempio `"Hello, world!\n"`
- due costanti stringa separate da spaziature sono equivalenti ad una sola costante stringa
- `"Hello, "`  
`"world\n"`  
equivale a: `"Hello, world\n"`

# Rappresentazione numeri

- un ***numero intero*** viene rappresentato da una sequenza di cifre, che vengono interpretate
  - in base decimale (default)
  - in base ottale se inizia con uno zero
  - in base esadecimale se inizia con '0x' (o '0X')
- un ***numero reale*** (in virgola mobile) viene rappresentato facendo uso del punto decimale e della lettera 'e' (o 'E') per separare la parte in virgola fissa dall'esponente; ad esempio:  
12.356e2 rappresenta 1235,6

# Operatori

Alcuni caratteri speciali e loro combinazioni sono usati come ***operatori***, cioè servono a denotare certe operazioni nel calcolo delle espressioni. Esempio:  $2*3+4$

+	-	*	/	%	^	&		~
!	=	<	>	+=	--	*=	/=	%=
^=	&=	=	<<	>>	>>=	<<=	==	!=
<=	>=	&&		++	--	,	->*	->
.*	::	()	[]	?:				

# Proprietà operatori

- la **posizione** rispetto ai suoi operandi (o argomenti): un operatore si dice
  - prefisso se precede gli argomenti
  - postfisso se segue gli argomenti
  - infisso se sta fra gli argomenti
- il **numero di argomenti** (o arità)
- la **precedenza** (o priorità) nell'ordine di esecuzione; ad esempio l'espressione  $1+2*3$  viene calcolata come  $1+(2*3)$  e non come  $(1+2)*3$
- la **associatività**, cioè l'ordine in cui vengono eseguiti operatori della stessa priorità (gli operatori possono essere associativi a destra o a sinistra)

# Separatori

- i ***separatori*** sono simboli di interpunzione, che indicano il termine di una istruzione, separano elementi di liste, raggruppano istruzioni o espressioni, ecc.
- sono ( ) , ; : { }
- alcuni caratteri, come la virgola, possono essere sia separatori che operatori, a seconda del contesto
- le parentesi devono essere sempre usate in coppia, come nel normale uso matematico



# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Variabili e Costanti

Per memorizzare un ***valore*** in un'area di memoria si utilizzano entità chiamate **variabili** o **costanti**.

Le **variabili** permettono la modifica del loro valore durante l'esecuzione del programma.

L'area di memoria corrispondente è identificata da un ***nome***, che ne individua l'*indirizzo di memoria*.

# Le Variabili

Le variabili sono caratterizzate da una quadrupla:

- **nome (identificatore)**
- **tipo**
- **locazione di memoria (l-value)**
- **valore (r-value)**

# Dichiarazione e Definizione di Variabili

## Definizione:

*tipo identificatore;*

Esempio:    **int x;**

## Definizione con inizializzazione:

*tipo identificatore=exp;*

Esempio:    **int x=3\*2;**

## Dichiarazione:

**extern** *tipo identificatore;*

Esempio:    **extern int x;**

# Definizione e Dichiarazione

- **Definizione:** quando il compilatore incontra una definizione di una variabile, esso predispone l'allocazione di un'area di memoria in grado di contenere la variabile del tipo scelto
- **Dichiarazione:** specifica solo il tipo della variabile e presuppone dunque che la variabile venga definita in un'altra parte del programma

# Dichiarazione di Costanti

## Sintassi:

**const** *tipo identificatore* = *exp*;

*exp* deve essere una espressione il cui valore deve poter essere calcolato in fase di compilazione

## Esempi:

```
const int kilo = 1024;
```

```
const double pi = 3.14159;
```

```
const int mille = kilo - 24;
```

# Rappresentazione binaria dei numeri

- La rappresentazione binaria dei numeri avviene tramite sequenze di bit (uni e zeri).
- Distinguiamo la rappresentazione per:
  - numeri interi positivi
  - numeri interi con segno
  - numeri reali

# Numeri interi positivi

- Una sequenza di bit  $b_{n-1}...b_1b_0$  rappresenta il numero:  $2^{n-1} \cdot b_{n-1} + ... + 4 \cdot b_2 + 2 \cdot b_1 + 1 \cdot b_0$
- Dati  $n$  bit è possibile rappresentare numeri da 0 a  $2^n - 1$
- Esempio (con 8 bit): 00001001 rappresenta il numero 9 ( $2^3 + 1 = 8 + 1$ )



# Interi con segno

- I numeri interi con segno sono rappresentati tramite diverse codifiche
- Le **codifiche** più usate sono:
  - codifica **segno-valore**
  - codifica **complemento a 1**
  - codifica **complemento a 2**

# Segno-valore

- Il primo bit rappresenta il segno, gli altri il valore
- Esempio (con 8 bit): 10001001 rappresenta -9
- Comporta una doppia rappresentazione dello zero: 00000000 e 10000000
- I numeri rappresentati appartengono all'intervallo  $-(2^{n-1}-1) \div +(2^{n-1}-1)$
- Occorre usare due diversi algoritmi per la somma a seconda che il segno degli addendi sia concorde o discorde

# Complemento a 1

- Un numero negativo è il “complemento” del corrispondente numero positivo
- Esempio: 11110110 rappresenta -9 (dove 00001001 rappresenta 9)
- Comporta una doppia rappresentazione dello zero: 00000000 e 11111111
- I numeri rappresentati appartengono all'intervallo  $-(2^{n-1}-1) \div +(2^{n-1}-1)$

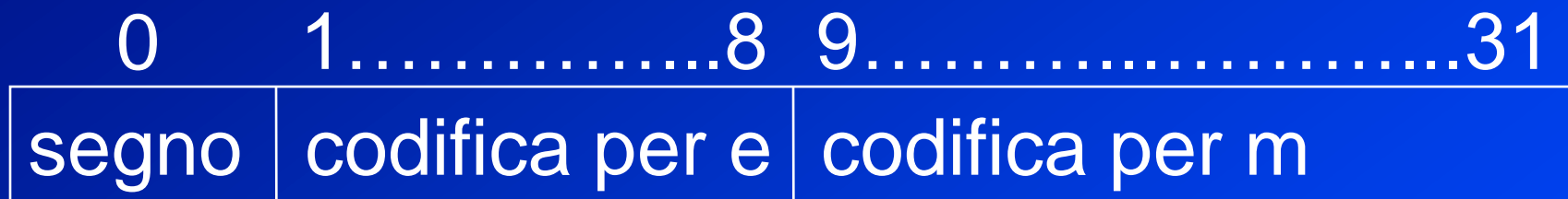
## Complemento a 2

- Un numero negativo è ottenuto calcolando il suo complemento e poi aggiungendo 1
- Esempio: 11110111 rappresenta -9
- Comporta un'unica rappresentazione dello zero che è 00000000
- I numeri rappresentati appartengono all'intervallo  $-2^{n-1} \div +(2^{n-1}-1)$

# Numeri Reali

Rappresentazione in virgola mobile  $m \times 10^e$

- $m$  è la mantissa ed  $e$  è l'esponente



- è necessaria una rappresentazione per lo zero
- intervallo di rappresentazione (con 32 bit)  
 $-1.7 \times 10^{+38} \div -0.29 \times 10^{-38}$  e  
 $0.29 \times 10^{-38} \div 1.7 \times 10^{+38}$
- La codifica è dipendente dal numero di bit e dalla particolare rappresentazione binaria

# I Tipi

- Oggetti dello stesso tipo utilizzano
  - lo stesso spazio in memoria
  - la stessa codifica
- Vantaggi sull'uso dei tipi:
  - correttezza semantica
  - efficiente allocazione della memoria dovuta alla conoscenza dello spazio richiesto in fase di compilazione

# Tipi Fondamentali e Derivati

- Nel C++ i tipi sono distinti in
  - **tipi fondamentali**, che servono a rappresentare informazioni semplici, come i numeri interi o i caratteri (**int**, **char**, ...)
  - **tipi derivati**, che permettono di costruire strutture dati complesse, a partire dai tipi fondamentali (puntatori, array, ...)

# I Tipi Fondamentali

- I tipi fondamentali del C++ sono:

- `int`, `short`, `long`, tipi interi

- `enum`, tipi enumerati

- `char`, tipo carattere

- `float`, `double`, tipi reali

Tipi

discreti



## Il tipo `int`

- il tipo `int` è costituito dai numeri interi compresi tra due estremi, a seconda dell'implementazione
- con N bit impiegati per la rappresentazione e assumendo una codifica in complemento a 2, si ha:

N	Min ( $-2^{N-1}$ )	Max ( $2^{N-1}-1$ )
16	-32768	32767
32	-2147483648	2147483647

# I tipi `short` e `long`

- Il tipo `short` tipicamente non ha più di 16 bit

Esempio:

```
short x=1232;
```

```
short int x=-132;
```

- Il tipo `long` tipicamente ha almeno 32 bit

Esempio:

```
long y=-34213332;
```

```
long int x=-6767899;
```

## Il tipo `unsigned`

- Il tipo `unsigned` rappresenta numeri interi **non negativi** di varie dimensioni
- Esempi:

```
unsigned int x=1232;
```

```
unsigned short int x=567;
```

```
unsigned long int x=878678687;
```

# Operatori aritmetici sugli interi

Operatore	Significato
<b>+</b>	<b>addizione</b>
<b>-</b>	<b>sottrazione</b>
<b>*</b>	<b>moltiplicazione</b>
<b>/</b>	<b>divisione intera</b>
<b>%</b>	<b>resto della divisione intera</b>

# Operatori bit a bit

Op.	Es.	Significato
<b>&gt;&gt;</b>	<b><math>x \gg n</math></b>	<b>shift a destra di <math>n</math> posizioni</b>
<b>&lt;&lt;</b>	<b><math>x \ll n</math></b>	<b>shift a sinistra di <math>n</math> posizioni</b>
<b>&amp;</b>	<b><math>x \&amp; y</math></b>	<b>AND bit a bit tra <math>x</math> e <math>y</math></b>
<b> </b>	<b><math>x   y</math></b>	<b>OR bit a bit tra <math>x</math> e <math>y</math></b>
<b>^</b>	<b><math>x \wedge y</math></b>	<b>XOR bit a bit tra <math>x</math> e <math>y</math></b>
<b>~</b>	<b><math>\sim x</math></b>	<b>NOT, complemento bit a bit</b>

## Esempi operazioni bit a bit

Supponiamo **x** e **y** di tipo **unsigned short**,  
rappresentato su otto bit

**x:** 00001100

**y:** 00001010

**x|y:** 00001110

**x&y:** 00001000

**x^y:** 00000110

**~x:** 11110011

**x<<2:** 00110000

**x>>2:** 00000011

Nota:

- Nello shift a sinistra i bit liberati sulla destra sono riempiti con degli zeri.
- Nello shift a destra i bit liberati a sinistra sono riempiti con il valore del bit più a sinistra

# Operatore di assegnazione

La sintassi dell'operatore di *assegnazione semplice*  
***exp1 = exp2***

***exp1*** deve essere un'espressione dotata di l-value  
***exp1*** e ***exp2*** devono essere di tipo compatibile

Un'assegnazione può essere usata all'interno di un'altra espressione. Il valore denotato da un'espressione di assegnazione è il valore di *exp2*. L'operazione di assegnazione, '=', associa a destra.

Esempio:

```
int a,b,c,d;  
a = b = c = d = 5;
```

è equivalente a:

```
(a=(b=(c=(d=5))));
```

# Operatori aritmetici e assegnazione

Forma compatta	Forma estesa
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x \% = y$	$x = x \% y$



## Operatori di incremento e decremento unitario

**$x++$**  incrementa  **$x$**  e

denota il valore di  **$x$**  prima dell'incremento

**$x--$**  decrementa  **$x$**  e

denota il valore di  **$x$**  prima del decremento

**$++x$**  incrementa  **$x$**  e

denota il valore di  **$x$**  dopo l'incremento

**$--x$**  decrementa  **$x$**  e

denota il valore di  **$x$**  dopo il decremento

# Operatori Relazionali e Logici

Operatore	Significato
<b>==</b>	<b>uguaglianza</b>
<b>!=</b>	<b>diversità</b>
<b>&gt;</b>	<b>maggiore</b>
<b>&lt;</b>	<b>minore</b>
<b>&gt;=</b>	<b>maggiore o uguale</b>
<b>&lt;=</b>	<b>minore o uguale</b>
<b>&amp;&amp;</b>	<b>and</b>
<b>  </b>	<b>or</b>
<b>!</b>	<b>not</b>

# Tipo Booleano

- Il C++ prevede un tipo booleano **bool** (ma puo' usare a tal scopo anche il tipo **int**)
- il valore **FALSO** è rappresentato dalla costante **false** (equivalente a 0)
- il valore **VERO** è rappresentato dalla costante **true** (equivalente ad un **valore intero diverso da 0**)
- Esempi:

<b>x</b>	<b>y</b>	<b>!x</b>	<b>x  y</b>	<b>x&amp;&amp;y</b>
<b>FALSO</b>	<b>FALSO</b>	<b>VERO</b>	<b>FALSO</b>	<b>FALSO</b>
<b>FALSO</b>	<b>VERO</b>	<b>VERO</b>	<b>VERO</b>	<b>FALSO</b>
<b>VERO</b>	<b>FALSO</b>	<b>FALSO</b>	<b>VERO</b>	<b>FALSO</b>
<b>VERO</b>	<b>VERO</b>	<b>FALSO</b>	<b>VERO</b>	<b>VERO</b>

## Tipi reali: `float` e `double`

- I tipi reali hanno come insieme di valori un sottoinsieme dei numeri reali, ovvero quelli rappresentabili all'interno del computer in un formato prefissato
- Ne esistono tre tipi a seconda della precisione:
  - `float`
  - `double`
  - `long double`
- La precisione di ciascun tipo è maggiore o uguale a quella del tipo precedente

## Esempi di variabili con tipi reali

```
double a = 2.2, b = -14.12e-2;  
double c = .57, d = 6.;
```

```
float g = -3.4F;    //literal float  
float h = g-.89F;   //suffisso F (f)
```

```
long double i = +0.001;  
long double j = 1.23e+12L;  
// literal long double  
// suffisso L (l)
```

# Tipi enumerati

- Un **tipo enumerato** è un **insieme finito** di costanti intere, definite dal programmatore, ciascuna individuata da un identificatore
- Sintassi:  
`enum typeid { id_or_init1, ..., id_or_initn }`

## Esempi di enumerazioni

```
enum Giorno {LUN,MAR,MER,GIO,  
             VEN,SAB,DOM};
```

```
Giorno oggi = LUN;
```

```
oggi = 3; //ERRORE! enum=const int
```

```
enum boolean { FALSE, TRUE};
```

```
boolean X = FALSE;
```

```
enum colore {ROSSO=10, GIALLO=15,  
             BLU=20};
```

```
colore sfondo= GIALLO, testo= BLU;
```

# Tipo carattere

- Il tipo **char** ha come insieme di valori i caratteri di stampa (es. 'a', 'Y', '6', '+')
- Il tipo **char** è un sottoinsieme del tipo **int** che generalmente un carattere occupa 1 byte
- Il valore numerico associato ad un carattere è detto codice e dipende dalla codifica utilizzata dal computer (es. ASCII, EBCDIC, BCD, ...)
- La codifica più usata è quella ASCII ma il tipo **char** è indipendente dalla particolare codifica adottata



# Codifica dei caratteri

Qualunque codifica deve soddisfare:

- **'a' < 'b' < ... < 'z'**
- **'A' < 'B' < ... < 'Z'**
- **'0' < '1' < ... < '9'**
- continuità dei codici delle successioni
  - **'a', 'b', ..., 'z'**
  - **'A', 'B', ..., 'Z'**
  - **'0', '1', ..., '9'**

Nota: non è fissa la relazione tra maiuscole e minuscole o fra i caratteri non alfabetici

# Codifica ASCII

0	NUL	1	^A	2	^B	3	^C	4	^D	5	^E	6	^F	7	^G
8	^H	9	^I	10	^J	11	^K	12	^L	13	^M	14	^N	15	^O
16	^P	17	^Q	18	^R	19	^S	20	^T	21	^U	22	^V	23	^W
24	^X	25	^Y	26	^Z	27	^[	28	^\	29	^]	30	^^	31	^-
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	DEL

# Esempio sui caratteri

```
char c = 'f';
```

```
n = '\n';
```

```
char l = 'a';
```

```
l += 3;          // l diventa 'd'
```

```
l--;            // l diventa 'c'
```

```
l -= 'a' - 'A'; // l diventa 'C'
```

# Operazioni miste e conversioni di tipo

- Spesso si usano operandi di tipo diverso in una stessa espressione o si assegna ad una variabile un valore di ***tipo diverso*** della variabile stessa
- Esempio:  

```
int prezzo = 27500;  
double peso = 0.3;  
int costo = prezzo * peso;
```
- In ogni operazione mista è sempre necessaria una conversione di tipo che può essere ***implicita*** o ***esplicita***

# Conversioni implicite

- Le conversioni *implicite* vengono effettuate dal compilatore
- Le conversioni implicite più significative sono:
  - nell'ambito numerico, gli operandi sono convertiti al tipo di quello di dimensione maggiore  
Esempio:  $3 * 2.6$  viene calcolata come  $3.0 * 2.6$  e non come  $3 * 2$
  - nell'assegnazione un'espressione viene sempre convertita al tipo della variabile

Esempi:

```
float x = 3;    //equivale a: x = 3.0
```

```
int y = 2*3.6; //equivale a: y = 7
```

# Conversioni esplicite

- Il programmatore può richiedere una **conversione esplicita** di un valore da un tipo ad un altro (*casting*)
- Esistono due notazioni:
  - Esempio:  
`int i = (int) 3.14;`
  - Esempio:  
`double f = double(3)`

# Operatore `sizeof`

- L'operatore `sizeof`, può essere prefisso a:
  - una variabile (esempio: `sizeof x`)
  - una costante (esempio: `sizeof 'a'`)
  - al nome di un tipo (esempio: `sizeof double`)
- Restituisce un intero rappresentante la dimensione in byte
- È valutato in fase di compilazione
- È applicabile a qualsiasi tipo (non solo ai tipi fondamentali)

# Funzioni di libreria

- Una **libreria** è un insieme di funzioni precompilate. Alcune librerie C++ sono disponibili in tutte le implementazioni e con le stesse funzioni
- Una libreria è formata da una coppia di file:
  - un file di **intestazione (header)** contenente le **dichiarazioni** dei sottoprogrammi stessi e il cui nome termina con *l'estensione* **h**
  - un file contenente le **funzioni compilate**
- Per utilizzare una libreria bisogna:
  - includere il file di intestazione della libreria con la direttiva **#include** <nomelibreria>
  - indicare al linker (vedi slide 21) il file contenente le funzioni compilate della libreria



# (Principali) Funzioni aritmetiche

Nella libreria `<cstdlib>`

- **abs(n)** valore assoluto
- **rand()** numero pseudocasuale tra 0 e la costante `RAND_MAX`
- **srand(n)** inizializza la funzione rand

Nella libreria `<cmath>`

- **fabs(x)** valore assoluto di tipo float
- **sqrt(x)** radice quadrata di x
- **pow(x, y)** eleva x alla potenza di y ( $x^y$ )
- **exp(x)** eleva e alla potenza di y ( $e^x$ )
- **log(x)** logaritmo naturale di x
- **log10(x)** logaritmo in base 10 di x
- **sin(x)** e **asin(x)** seno e arcoseno trigonometrico
- **cos(x)** e **acos(x)** coseno e arcoseno trigonom.
- **tan(x)** e **atan(x)** tangente e arcotangente trig.

# Funzioni sui caratteri in `<ctype>`

Funzioni che restituiscono un `int` rappr. un booleano:

- `isalnum(c)` carattere alfabetico o cifra decimale
- `isalpha(c)` carattere alfabetico
- `iscntrl(c)` carattere di controllo
- `isdigit(c)` cifra decimale
- `isgraph(c)` carattere grafico, diverso da spazio
- `islower(c)` lettera minuscola
- `isprint(c)` carattere stampabile, anche spazio
- `isspace(c)` spazio, salto pagina, nuova riga o tab.
- `isupper(c)` lettera maiuscola
- `isxdigit(c)` cifra esadecimale

Funzioni che restituiscono un carattere:

- `tolower(c)` se `c` è una lettera maiuscola restituisce la minuscola, altrimenti `c`
- `toupper(c)` come sopra ma in maiuscolo

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Tipi derivati

- Dai tipi fondamentali, attraverso vari meccanismi, si possono derivare altri tipi
- I tipi derivati sono:
  - i **referimenti**
  - i **puntatori**
  - gli array
  - le strutture
  - le unioni
  - le classi

# Tipo riferimento (1)

- Il meccanismo dei **riferimenti** (*reference*) consente di dare nomi multipli a una variabile
- Esempio:  
`int x=1;`  
`int &y=x; // y è di tipo reference`  
`y++`
- Attenzione:  
Nelle dichiarazioni di variabili di tipo reference, l'inizializzazione è **obbligatoria**  
Esempio:  
`int &y; // errore!`

## Tipo riferimento (2)

- Non è possibile ridefinire una variabile di tipo riferimento precedentemente definita:  
`double x1,x2;`  
`double &y=x1; // ok`  
`double &y=x2; // errore! gia' definita!`
- Se viene inizializzata con un *r-value* o con una variabile di tipo diverso, viene generata una variabile temporanea di cui diviene sinonimo.

```
float &y=10.2; /* Equivalente a:  
               float T1=10.2; float &y=T1 */  
double d=3.1; //Equivalente a:  
int &z=d; //double d=3.1;  
           // int T2=int(d); int &z=T2;
```

# Puntatori

- Una variabile di tipo puntatore rappresenta l'indirizzo di un altro oggetto o di una funzione
- Hanno come valori gli indirizzi di memoria di locazioni di memoria, ovvero l'*r-value* di un puntatore è un *l-value*
- L'operatore **&** (address-of) ritorna l' *l-value* dell'espressione a cui è applicato
- Sintassi:

*tipo \*id\_or\_init*

Esempio:

```
int *px;
```

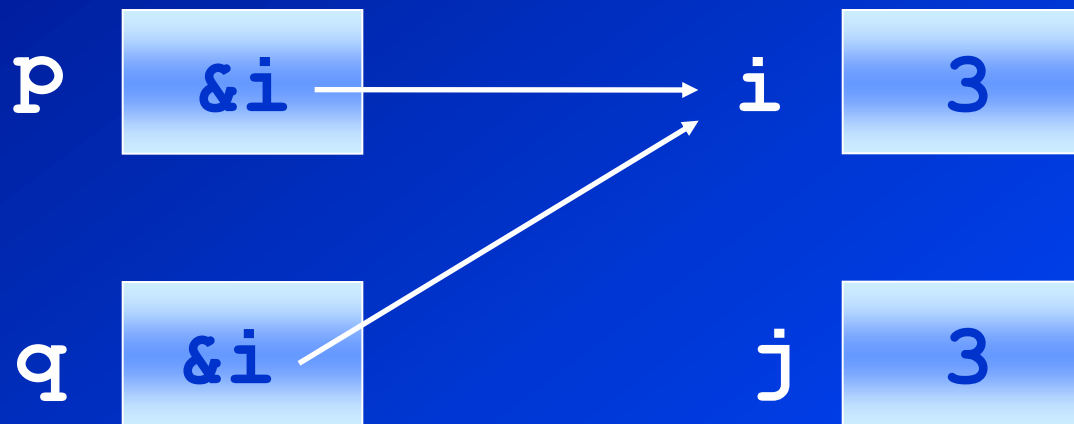
# Operatore di *dereference*

- Per accedere all'oggetto puntato da una variabile puntatore occorre applicare l'operatore di *dereference* \*

```
int x=1; // x variabile tipo int
int *px; // px variabile puntatore
px=&x;   // accede alla variabile
        // puntatore
*px=x+1; /* accede alla cella di
          memoria puntata dalla
          variabile puntatore */
```



# Esempio di utilizzo dei puntatori



```
int i, j;
```

```
int *p; int *q;
```

```
p=&i; // p=indirizzo di i
```

```
*p=3; // equivale a i=3
```

```
j=*p; // equivale a j=i
```

```
q=p; // equivale a q=indirizzo di i
```

# Memoria e puntatori

- Ad una variabile puntatore viene associata uno spazio di memoria atta a contenere un indirizzo di memoria, ma **non viene riservato spazio di memoria** per l'oggetto puntato
- Lo spazio allocato a una variabile di tipo puntatore è sempre uguale, indipendentemente dal tipo dell'oggetto puntato
- Per inizializzare una variabile puntatore ad un indirizzo costante è necessario effettuare un casting (conversione esplicita):  
`double *px=(double *) 321;`

# Puntatori a Costanti e Costanti Puntatore

Sintassi:

Puntatore a costante:

```
const tipo *id_or_init;
```

Costante puntatore:

```
tipo *const id=exp;
```

Costante puntatore a costante:

```
const tipo *const id=exp;
```

# Esempio di puntatore a costante

```
const int c1 = 3;
```

```
int c2 = 5;
```

```
const int *pc1 = &c1;    // ok
```

```
const int *pc2 = &c2;    // ok
```

```
pc2 = pc1; // ok
```

```
pc1 = &c2; // ok
```

```
c2 = 2;    // ok
```

```
*pc1 = 2;  /* errore: non è  
             possibile modificare c2  
             tramite pc1 */
```

## Esempio di costante puntatore

```
int a, b;  
int *const pa = &a;  
*pa = 3;    // ok  
pa = &b     // errore: pa e' costante
```

```
/* Esempio di costante puntatore  
   a costante */
```

```
const int b = 2;  
const int c = 3;  
const int *const a = &c;  
a = &b; // errore  
*a = 2; // errore  
c = 5;  // errore
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Struttura di un programma

- Un programma consiste in un insieme di funzioni, eventualmente suddivise in più file
- La funzione che costituisce il programma principale si deve chiamare `main`
- Contiene una lista di istruzioni di ogni tipo: semplici o strutturate

- Esempio:

```
int main() {  
    int x=2, y=6, z;  
    z=x*y;  
    return 0; }
```

} Questo è il contenuto del  
programma principale

# Istruzioni semplici

- Le istruzioni semplici sono la base delle istruzioni più complesse (istruzioni strutturate)
- Si distinguono in
  - dichiarazioni (declaration-statement) di
    - variabili, es. `int x`
    - costanti, es. `const int kilo=1024`
  - espressioni (expression-statement)
    - di assegnamento, es. `x=2`
    - aritmetiche, es. `(x-3)*sin(x)`
    - logiche, es. `x==y && x!=z`
    - costanti, es. `3*12.7`
    - **condizionali**
    - **con virgola**



# Espressione condizionale

Sintassi:

$$exp1 \text{ ? } exp2 : exp3$$

Se *exp1* è vera equivale a *exp2*, altrimenti equivale a *exp3*

Esempio:

```
prezzo = valore * peso *  
        (peso > 10) ? 0.9 : 1;
```

Se il peso è maggiore di 10 allora equivale a

```
prezzo = valore * peso * 0.9;
```

altrimenti equivale a

```
prezzo = valore * peso * 1;
```

# Operatore virgola

- Consente di inserire una sequenza di istruzioni dove il linguaggio ammette l'uso di un'unica espressione.
- L'espressione:  
`r = (a = 4, c++, x += 3);`  
equivale a:  
`a = 4; c++; r = x += 3;`
- Il risultato è dato dal valore dell'operando più a destra
- la valutazione avviene da sinistra verso destra

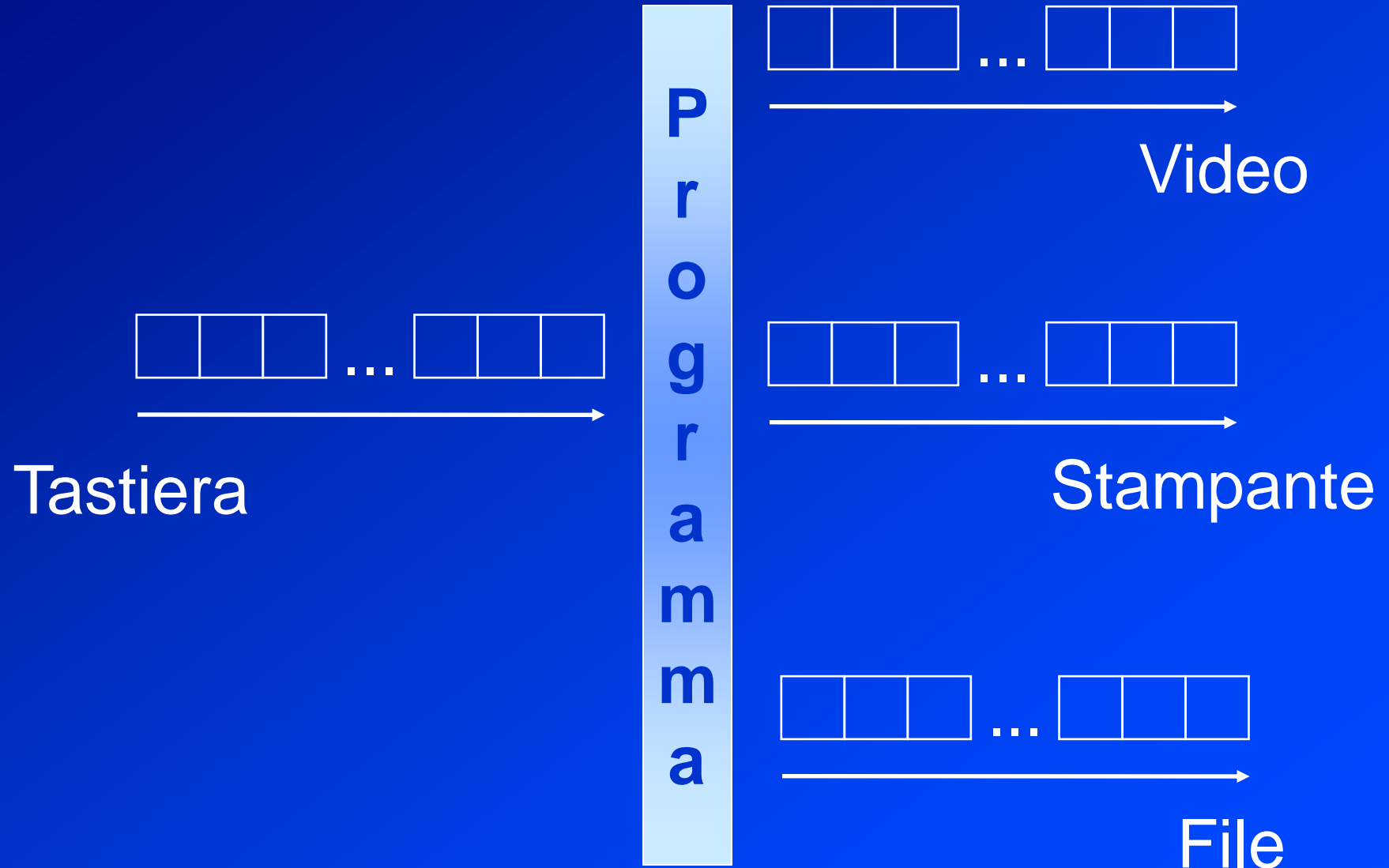
# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Concetto di stream

- Un programma comunica con l'esterno tramite uno o più **flussi** (**stream**)
- Uno **stream** è una struttura logica costituita da una **sequenza di caratteri**, in numero teoricamente infinito, terminante con un apposito carattere che ne identifica la fine
- Gli **stream** vengono associati (con opportuni comandi) ai dispositivi fisici collegati al computer (tastiera, video, stampante) o a file residenti sulla memoria di massa

# Schema funzionale degli stream



# Stream predefiniti

- In C++ esistono i seguenti **stream predefiniti**
  - **cin** (stream standard di ingresso)
  - **cout** (stream standard di uscita)
  - **cerr** (stream standard di errore)
- Le funzioni che operano su questi stream sono in una libreria di ingresso/uscita e per usarle occorre la direttiva **#include <iostream>**

# Stream di uscita

- Lo stream di uscita standard (`cout`) è quello su cui il programma scrive i dati ed è tipicamente associato allo schermo
- Per scrivere dati si usa l'istruzione di scrittura `stream << espressione;`
- L'istruzione di scrittura comporta
  - il calcolo dell'espressione
  - la conversione in una sequenza di caratteri
  - il trasferimento della sequenza nello stream

## Esempio di scrittura su cout

```
int x=10, y=29; char s=' ';  
cout << "x contiene";  
cout << s;  
cout << x;  
cout << '\n';  
cout << "y contiene";  
cout << s;  
cout << y;  
cout << '\n';
```

Apparirà sullo schermo come:

```
x contiene 10  
y contiene 29
```



# Scritture multiple

- L'istruzione di scrittura ha una forma più generale che consente scritture multiple
- Esempio:

```
cout << x << y << endl;
```

equivale a

```
cout << x;
```

```
cout << y;
```

```
cout << endl;
```

- Nota: la costante predefinita `endl` corrisponde ad un `'\n'` per cui viene iniziata una nuova linea con il cursore nella prima colonna








# Stream di ingresso

- Lo stream di ingresso standard (`cin`) è quello da cui il programma preleva i dati ed è tipicamente associato alla tastiera
- Per prelevare dati si usa l'istruzione di lettura `stream >> espressione;`  
(`espressione` dotata di `l-value`)
- L'istruzione di lettura comporta
  - il prelievo dallo stream di una sequenza di caratteri
  - la conversione di tale sequenza in un valore che viene assegnato alla variabile

# Lettura di un carattere da cin

- Con un'istruzione del tipo `cin >> x;`
- se `x` è di tipo `char` e lo stream contiene  
 .....☐☐☐☐a☐-14.53☐☐728.....  
 ↑                      →
- se il carattere selezionato dal puntatore (↑) non è una spaziatura (☐)
  - il carattere viene prelevato
  - il carattere viene assegnato alla variabile `x`
  - il puntatore si sposta alla casella successiva
- altrimenti il puntatore si sposta in avanti per trovare un carattere che non sia una spaziatura

# Lettura di un numero da cin

- Con un'istruzione del tipo `cin >> x;`
  - se `x` è di tipo `int` e lo stream contiene  
 .....-157Ciao6.28.....  
 ↑                      →
  - la variabile `x` conterrà il valore `-157` e il puntatore si sarà spostato fino al carattere seguente la cifra 7
  - Con uno stream che invece contiene  
 .....Ciao-157.....  
 ↑                      →
- l'operazione di prelievo non avviene e lo stream si porta in uno stato di errore

# Lecture multiple

- L'istruzione di ingresso ha una forma più generale che consente **lettture multiple**
- Ad esempio, con un'istruzione del tipo `cin >> x >> y >> z;`
- con uno stream contenente  

```
.....  a -14.53  728  .....
```

↑

→
- se **x**, **y** e **z** sono rispettivamente di tipo **char**, **double** e **int**
- al termine dell'esecuzione **x**, **y** e **z** avranno rispettivamente i valori **'a'**, **-14.53** e **728**

## Esempio di utilizzo cin e cout

```
#include <iostream>
int main() {
    int age, year;
    cout << "Quanti anni hai? ";
    cin >> age;
    cout << "In che anno siamo? ";
    cin >> year;
    cout << "Sei nato nel "
         << year-age << endl;
    return 0; }
```

Quanti anni hai? 25

In che anno siamo? 2000

Sei nato nel 1975

# Uso dei file

- Un'apposita libreria del C++ consente di utilizzare stream che vengono associati a file del sistema operativo
- È utilizzabile con la direttiva `#include <fstream>`
- stream di questo tipo vengono dichiarati con `fstream nomeidentificatore;`
- Ad esempio:  
`fstream ingresso, uscita;`

## Accesso ai file

- Uno stream associato ad un file può essere aperto mediante la funzione `open()` secondo le modalità, identificate dalle costanti tra ( )
  - lettura (`ios::in`)
  - scrittura (`ios::out`)
  - append (scrittura a fine file) (`ios::out|ios::app`)
- Il nome del file viene specificato come una sequenza di caratteri (es. `"file1.in"`)
- Esempio:

```
fstream ingr, usc;  
ingr.open("file1.in", ios::in);  
usc.open("file2.out", ios::out);
```



# Apertura di uno stream associato a file

- Apertura in lettura:
  - il file associato deve già essere presente
  - il puntatore si sposta all'inizio dello stream
- Apertura in scrittura:
  - il file associato se non è presente viene creato
  - il puntatore si posiziona all'inizio dello stream (eventuali dati presenti vengono perduti)
- Apertura in append
  - il file associato se non è presente viene creato
  - il puntatore si posiziona alla fine dello stream

# Chiusura di uno stream associato a file

- Uno stream, quando è stato utilizzato, può essere chiuso mediante la funzione `close()`
- Esempio:  
`ingr.close();`  
`usc.close();`
- Alla fine del programma tutti gli stream aperti vengono automaticamente chiusi
- Una volta chiuso, uno stream può essere riaperto in qualunque modalità e associato a qualunque file

# Funzioni delle librerie per gli stream (1)

- `cin.get(x)` ; legge tutti i caratteri, compresi anche quelli di spaziatura; se `x` non è di tipo `char` è equivalente a `cin>>x`;
- `cin.eof()` ; se lo stream `cin` ha raggiunto la sua fine (End Of File) viene ritornato un valore diverso da 0

## Funzioni delle librerie per gli stream (2)

- **`cin.fail()`** ; segnala uno stato di errore, assumendo un valore diverso da 0; accade quando si è tentato di leggere una sequenza di caratteri non consistente con il tipo di variabile dell'istruzione di lettura (es. si cerca di leggere un `int` e in input ci sono delle lettere)
- **`cin.clear()`** ; effettua il ripristino dello stato normale dallo stato di errore

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati

8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Istruzioni strutturate

- Le istruzioni strutturate consentono di specificare azioni complesse
- Si distinguono in
  - istruzione composta (compound-statement)
  - istruzioni condizionali (conditional-statement)
  - istruzioni iterative (iteration-statement)
  - istruzioni di salto (jump-statement)

# Istruzione composta

- L'istruzione composta consente, per mezzo della coppia di delimitatori '{' e '}', di trasformare una qualunque sequenza di istruzioni in una singola istruzione
- Esempio:

```
{ int a=4;  
  a*=6;  
  char b='c' ;  
  b+=3; }
```
- Le definizioni possono comparire in qualunque punto del blocco e sono visibili solo all'interno del blocco

# Istruzione condizionale `if`

## Sintassi:

`if (exp) istruzione1;`

Se *exp* è vera viene eseguita *istruzione1*

## Esempio:

`if (x!=0) y=1/x;`

## Sintassi:

`if (exp) istruzione1;`

`else istruzione2;`

Se *exp* è vera viene eseguita *istruzione1*  
altrimenti viene eseguita *istruzione2*

## Esempio:

`if (x<0) y=-x; else y=x;`



# Esempio di programma con if

```
#include <iostream>
// calcolo soluzioni di  $ax + b = 0$ 
int main() {
    float a,b;
    cin >> a >> b;
    if (a==0)
        if (b==0)
            cout<<"Infinite soluzioni\n";
        else
            cout<<"Non esiste soluzione\n";
    else
        cout<<"Soluzione: "<<-b/a<<endl;
    return 0;}
```

# Discriminazione fra scelte con if

```
#include <iostream>
int main() { //semplice calcolatrice
    double op1, op2; char op;
    cin >> op1 >> op >> op2;
    if (op=='+')
        cout << op1+op2;
    else if (op=='-')
        cout << op1-op2;
    else if (op=='*')
        cout << op1*op2;
    else if (op=='/')
        cout << op1/op2;
    else cout<<"Errore!";
    return 0; }
```

# Istruzione condizionale `switch`

## Sintassi:

```
switch (exp) {  
  case const-exp1: istruzione1; break;  
  case const-exp2: istruzione2; break;  
  ...  
  default: istruzione-default; }
```

L'esecuzione dell'istruzione `switch` consiste

- nel calcolo dell'espressione *exp*
- nell'esecuzione dell'istruzione corrispondente all'alternativa specificata dal valore calcolato
- se nessuna alternativa corrisponde, se esiste, viene eseguita *istruzione-default*

# Discriminazione fra scelte con switch

```
#include <iostream>
int main() {
    //semplice calcolatrice
    double op1, op2; char op;
    cin >> op1 >> op >> op2;
    switch (op)
    {
        case '+': cout << op1+op2; break;
        case '-': cout << op1-op2; break;
        case '*': cout << op1*op2; break;
        case '/': cout << op1/op2; break;
        default : cout << "Errore!";
    }
    return 0; }
```

## Scelte multiple con `switch`

- Se dopo l'ultima istruzione di un'alternativa non c'è un **`break`**, viene eseguita anche l'alternativa successiva
- Questo comportamento è **sconsigliato** ma può essere giustificato in alcuni casi

- Esempio:

```
switch (giorno)
{ case lun: case mar:
  case mer: case gio:
  case ven: ore_lavorate+=8; break;
  case sab: case dom: break;
}
```

# Istruzione iterativa **while**

## Sintassi:

**\_while** (*exp*) *istruzione*;

L'esecuzione dell'istruzione **while** comporta

- il calcolo dell'espressione *exp*
- se *exp* è vera, l'esecuzione di *istruzione* e la ripetizione dell'esecuzione dell'istruzione **while**

## Esempio di utilizzo di while

```
#include <iostream>
int main() {
    int num, ndiv2=0;
    cout << "Immetti un numero: ";
    cin >> num;
    while ( num%2 == 0 ) {
        ndiv2++;
        num/=2;
    }
    cout << "E' divisibile " << ndiv2
         << (ndiv2==1?" volta":" volte")
         << " per 2" << endl;
    return 0; }
```

# Istruzione iterativa **do**

## Sintassi:

**\_\_do** { *istruzione* } **while** (*exp*) ;

L'esecuzione dell'istruzione **do** comporta

- l'esecuzione di *istruzione*
- il calcolo dell'espressione *exp*
- se *exp* è vera, la ripetizione dell'esecuzione dell'istruzione **do**



## Esempio di utilizzo dell'istruzione do

```
#include <iostream>

int main() { //conversione di base
    unsigned int cifra, num, base;
    cin >> num >> base;
    if (base>=2 && base<=10)
        do {
            cifra = num % base;
            num /= base;
            cout << cifra;
        } while (num!=0);
    else cout << "Base non valida!";
    return 0; }
```

# Istruzione iterativa **for**

## Sintassi:

**for** ( *init-exp*; *exp1*; *exp2* ) *istruzione*;

L'esecuzione dell'istruzione **for** comporta

1. l'esecuzione di *init-exp* (che di solito è un assegnamento che inizializza una variabile, detta di controllo)
2. il calcolo dell'espressione *exp1*
3. se *exp1* è vera, viene eseguita *istruzione*, poi *exp2* (che di solito è un incremento o un decremento della variabile di controllo) e si rinizia dal passo 2.

# Esempio di utilizzo dell'istruzione for

```
#include <iostream>

int main() {
    //calcolo del fattoriale
    int fattoriale, num;
    cout << "Inserisci un numero: ";
    cin >> num;
    fattoriale=1;
    for (int i=1; i<=num; i++)
        fattoriale *= i;
    cout << "il suo fattoriale e' "
        << fattoriale;
    return 0; }
```

# Trasformazione di `for` in `while`

`for ( exp1; exp2; exp3 ) exp4;`

equivale a

`exp1; while ( exp2 ) { exp4; exp3 } ;`

Esempio:

`for (int i=1; i<10; i++) x*=2;`

equivale a

`int i=1; while (i<10) { x*=2; i++};`


# Esempio con tutte le istruzioni iterative

```
#include <iostream>
int main(){//numeri divisibili per 3
    int start, stop; char c='s';
    while (c=='s') {
        cout<<"Inserisci un intervallo: ";
        cin >> start >> stop;
        for (int i=start; i<=stop; i++) {
            if ((i % 3)==0)
                cout << i <<
                    " divisibile per 3\n"; }
        do {
            cout<<"Vuoi continuare (s/n)? ";
            cin >> c; cout << endl;
        } while (c!='s' && c!='n');
    }
    return 0; }
```

# Istruzione di salto **break**

- l'istruzione **break** termina direttamente tutto il ciclo

```
while (...) {  
    ...  
    while (...) {  
        ...  
        break;  
        ...  
    }  
    ...  
}
```



The diagram illustrates the effect of the `break` statement. It shows a nested `while` loop structure. Inside the inner `while` loop, there is a `break;` statement. A white line with an arrow originates from the `break;` statement and points back to the opening curly brace of the inner `while` loop, indicating that the loop is terminated immediately.


## Esempio di utilizzo di break

```
#include <iostream>
int main(){
    int a, b; char c;
    while (1) {
        cout<<"Inserisci due numeri: ";
        cin >> a >> b;
        cout << a << '+' << b << '='
            << a+b << '\n';
        cout<<"Vuoi continuare (s/n)? ";
        cin >> c;
        if (c=='n') break;
    }
    return 0; }
```

# Istruzione di salto `continue`

- l'istruzione `continue` termina il ciclo che attualmente è in esecuzione e passa al successivo (nel caso di ciclo `for` viene saltata l'istruzione di aggiornamento)

```
while (...) {  
    ...  
    while (...) {  
        ...  
        continue;  
        ...  
    }  
    ...  
}
```





# Esempio di utilizzo di continue

```
#include <iostream>
int main() {
    /* somma i numeri multipli di 5 o 8
       nell'intervallo da 1 a 30 */
    int i=0, somma=0;
    while (i<30) {
        i++;
        if ((i%5) !=0 && (i%8) !=0)
            continue;
        cout << i << endl;
        somma+=i;
    }
    cout << "somma = " << somma << endl;
    return 0;}
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Concetto di funzione

- Nella realizzazione di un programma
  - il codice può diventare molto lungo
  - spesso una stessa sequenza di operazioni viene ripetuta in più punti
  - È quindi opportuno e conveniente strutturare il codice raggruppandone delle sue parti in moduli autonomi, detti funzioni, che vengono eseguiti in ogni punto in cui è richiesto
- L'organizzazione a funzioni consente di isolare parti di un programma che possono essere modificate in maniera indipendente

# Definizione, Dichiarazione e Chiamata

- Definizione:  
 $\textit{tipo } id ( \textit{tipo1 } id1, \dots, \textit{tipoN } idN ) \{ \textit{corpo} \}$
- Dichiarazione:  
 $\textit{tipo } id ( \textit{tipo1 } id1, \dots, \textit{tipoN } idN ) ;$   
 $id1, \dots, idN$  sono i **parametri formali** della funzione
- Chiamata:  
 $id ( \textit{exp1}, \dots, \textit{expN} )$   
 $\textit{exp1}, \dots, \textit{expN}$  sono i **parametri attuali** della chiamata e devono essere di tipo compatibile ai tipi dei corrispondenti parametri formali

# Istruzione `return`

- Il corpo di una funzione può contenere una o più volte l'istruzione `return`
- Sintassi:  
`return expression;`
- L'istruzione `return` fa terminare la funzione e fa sì che il valore della funzione sia il valore dell'espressione *expression*
- L'espressione può mancare nel caso in cui si abbia a che fare con funzioni che non restituiscono un risultato (procedure)

## Esempio di funzione

```
/* funzione che calcola il massimo  
comune divisore tra due numeri  
interi positivi */
```

```
int mcd(int a, int b)  
{  
    int resto;  
    while (b!=0) {  
        resto = a % b;  
        a = b;  
        b = resto;  
    }  
    return a;  
}
```

# Chiamata di funzione

```
#include <iostream>
/* programma che chiama la funzione
   mcd per il calcolo del M.C.D. */
int main()
{
    int n1, n2;
    cout << "Inserisci una coppia "
           "di numeri ";
    cin >> n1 >> n2;
    cout << "Il M.C.D. fra " << n1
         << " e " << n2 << " vale "
         << mcd(n1,n2) << endl;
    return 0; }
```

# Parametri e variabili locali (1)

- Un parametro formale è una variabile cui viene associato il corrispondente parametro attuale ad ogni chiamata della funzione
- Le variabili dichiarate all'interno di una funzione sono dette *locali* a tale funzione e appartengono solo alla funzione in cui sono dichiarati (sono *visibili* solo nella funzione)



## Parametri e variabili locali (2)

- I parametri formali e le variabili locali esistono solo durante l'esecuzione della rispettiva funzione
- All'atto della chiamata viene riservata un'area di memoria per contenere i valori dei parametri formali per valore e delle variabili locali
- I parametri formali e le variabili locali vengono utilizzati per le dovute elaborazioni
- Al termine della funzione la memoria da essi occupata viene resa disponibile

# Passaggio parametri

In C++ esistono due modalità di associazione dei parametri attuali ai parametri formali:

- passaggio parametri **per valore**. Il parametro formale ha una sua propria cella di memoria in cui viene copiato il valore del parametro attuale corrispondente all'atto della chiamata.
- passaggio parametri **per riferimento**. La cella di memoria associata al parametro formale è quella del parametro attuale corrispondente (che si presuppone ne abbia una).

## Chiamata della funzione `mcd`

- La chiamata di funzione `mcd(n1, n2)` viene eseguita nel modo seguente:
  - vengono calcolati i valori dei parametri attuali `n1` e `n2` (l'ordine non è specificato)
  - i valori vengono copiati, nell'ordine, nei contenitori `a` e `b` (parametri formali)
  - viene eseguita la funzione e modificati i valori di `a`, `b` e della variabile locale `resto` (`n1` e `n2` rimangono con il loro valore originale)
  - la funzione `mcd` restituisce al programma chiamante il valore dell'espressione che appare nell'istruzione `return`

# Modello a Stack

```
void metti_iva  
  (double &val, iva){  
    double i=1+iva/100;  
    val*=i;  
  }  
int main() {  
  double aliq = 20;  
  double v = 10000;  
  metti_iva(v, aliq);  
  cout << v;  
  return 0;}
```



## Esempio di passaggio per riferimento

```
// procedura che scambia due interi
void scambia(int &n, int &m)
{
    int t;
    t = n; n = m; m = t;
}
```

```
// procedura errata!
void scambia(int n, int m)
{
    int t;
    t = n; n = m; m = t;
}
```

# Vantaggi del riferimento

- Maggiore potenzialità espressiva
- Minore carico di calcolo (soprattutto con parametri di grosse dimensioni)
- Minor bisogno di uso di variabili globali

## Svantaggi del riferimento

- Rischio di confusione nel codice (non si sa dove cambiano i valori)
- Aliasing (entità con più di un nome)
- Parametro formale e parametro attuale devono essere esattamente dello stesso tipo
- Si possono passare solo variabili (indirizzi di memoria)

# Vantaggi delle funzioni

- Localizzazione della funzionalità
- Fattorizzazione del codice
- Manutenibilità
- Leggibilità
- Verificabilità
- Correttezza semantica dei parametri



## Procedure (funzioni `void`)

- L'importanza di individuare un corpo funzionalmente rilevante all'interno di un programma può valere anche se questo corpo non ritorna un valore
- In C++ c'è perciò la possibilità di definire procedure, ovvero funzioni il cui valore di ritorno è di tipo `void`

## Esempio

```
#include <iostream>

void stampa_coppia (int a, int b)
{ cout << '(' << a << ','
  << b << ') ' ; }

int main() {
  int n1, n2;
  cout << "Inserisci due numeri ";
  cin >> n1 >> n2;
  cout << "La coppia e' ";
  stampa_coppia(n1,n2);
  return 0; }
```

# Ricorsione

- In C++ (come nella maggior parte dei linguaggi di programmazione) una funzione può invocare non solo un'altra funzione, ma anche se stessa: si ha in questo caso una funzione ricorsiva
- Si rende agevole la programmazione in tutti i quei casi in cui risulta naturale formulare il problema da risolvere in maniera ricorsiva
- Un classico esempio della matematica:  
 $0! = 1$   
 $n! = n \times (n-1)!$

## Esempio di funzione ricorsiva

```
// funzione che calcola  
// il fattoriale
```

```
int fatt(int n)  
{  
    if (n==0 || n==1) return 1;  
    return n * fatt(n-1);  
}
```

# Modello a stack

n	1
fatt	1
n	2
fatt	2
n	3
fatt	6
n	4
fatt	24

# Puntatori e Passaggio Parametri

- I puntatori consentono di simulare il passaggio parametri per riferimento
- Esempio:

```
void scambia(int *px, int *py) {  
    int t;  
    t = *px; *px = *py; *py = t; }  
int main() {  
    int a, b;  
    cin >> a >> b;  
    scambia(&a, &b);  
    cout << a << b << endl;  
    return 0;}
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Tipi e Variabili Array

- Un array è una sequenza finita di elementi omogenei (dello stesso tipo).
- Sintassi:  
*tipo id[ dim] ;*  
*tipo id[ dim] = { lista\_valori} ;*  
*tipo id[ ] = { lista\_valori} ;*
- Esempi:  
`double a[25]; //array di 25 double`  
`const int c=2;`  
`char b[2*c]={'a','e','i','o'};`  
`char d[]={ 'a','e','i','o','u'};`



# Operazioni sugli array

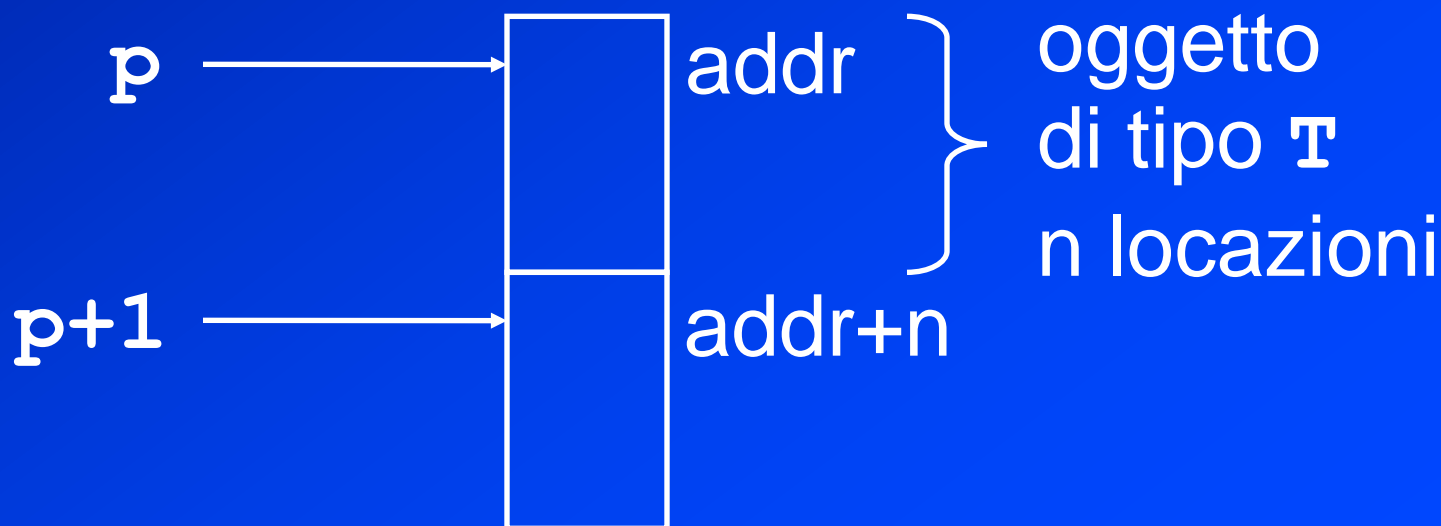
- Sugli array non sono definite né operazioni aritmetiche, né operazioni di confronto, né l'assegnamento
- L'unica operazione definita è la selezione con indice (subscripting), ottenuta nella forma:  
*identifier[expression]*
  - l'identificatore è il nome dell'array
  - il valore dell'espressione, di tipo discreto, è l'indice dell'elemento a cui si vuole accedere
- Gli elementi di un array di dimensione **N** sono numerati da **0** a **N-1**
- Esempio: **a[3]=7** assegna il quarto elemento

## Esempi con array

- `array1.cc`  
calcolo del minimo in una matrice rettangolare
- `array2.cc`  
prodotto di due matrici rettangolari

# Aritmetica dei puntatori

- I puntatori hanno un'aritmetica per cui se l'espressione  $p$  rappresenta l'indirizzo di un oggetto di tipo  $T$ , dato un intero  $i$ , allora l'espressione  $p+i$  rappresenta l'indirizzo di un oggetto, sempre di tipo  $T$ , che si trova in memoria dopo  $i$  posizioni



# Puntatori ad elementi di un array

- Se un puntatore si riferisce al primo elemento di un array, l'espressione  $p+i$  è l'indirizzo dell' $i$ -esimo elemento dell'array
- Dati due puntatori dello stesso tipo  $p1$  e  $p2$  che puntano entrambi ad elementi di uno stesso array,  $(p2-p1)$  denota il numero di elementi compresi tra  $p1$  e  $p2$

# Array e Puntatori

- Il nome di un array è una costante puntatore al primo elemento dell'array stesso
- Le espressioni `&p[i]` e `p+i` sono dunque equivalenti come lo sono anche le espressioni `p[i]` e `*(p+i)`

- **Esempio:**

```
int x[10]; int *p;  
p = x;           //indir. primo elem.  
*(p + 3) = 4;    //equivale a x[3]=4  
p = &x[5]; p += 4; //avanza di 4  
*p = 7;          //equivale a x[9]=7
```

# Array multidimensionali

- È possibile dichiarare array i cui elementi sono a sua volta degli array, generando degli array multidimensionali (matrici)
- Sintassi:  
*tipo id* [*dim*<sub>1</sub>] [*dim*<sub>2</sub>] ... [*dim*<sub>*n*</sub>] ;  
*tipo id* [*dim*<sub>1</sub>] [*dim*<sub>2</sub>] ... [*dim*<sub>*n*</sub>] = { *lista\_valori* } ;  
*tipo id* [ ] [*dim*<sub>2</sub>] ... [*dim*<sub>*n*</sub>] = { *lista\_valori* } ;
- Quindi in C++ un array multidimensionale *dim*<sub>1</sub> x *dim*<sub>2</sub> x ... x *dim*<sub>*n*</sub> può essere pensato come un array di *dim*<sub>1</sub> array multidimensionali *dim*<sub>2</sub> x ... x *dim*<sub>*n*</sub>

# Funzione con parametri di tipo array

- Una funzione può avere un argomento formale del tipo “array di T” che corrisponde ad un puntatore ad oggetti di tipo T
- Il corrispondente parametro attuale è costituito dal nome di un array di oggetti di tipo T che equivale all'indirizzo del suo primo elemento
- Quando viene passato un array ad una funzione, non vengono copiati i suoi elementi nel corrispondente parametro attuale ma viene ricopiato solo l'indirizzo del primo elemento

## Esempio funz. con param. di tipo array

- Nella dichiarazione di un parametro formale di tipo array (monodimensionale) si può omettere la specifica della dimensione per poter operare su array di dimensioni diverse

//funzione che somma i primi n elem.

```
int somma(int v[], int n) {  
    int i, s=0;  
    for (i=0; i<n; i++) s += v[i];  
    return s;  
}
```



## Esempio di passaggio array

```
#include <iostream>

int main() {
    int a[100], n;
    cout<<"Quanti numeri (max 100)?";
    cin >> n; if (n>100) return;
    for (int i=0; i<n; i++) {
        cout << endl << i+1 << ": ";
        cin >> a[i];
    };
    cout << "La media vale "
        << somma(a,n)/n << endl;
    return 0;}
```

# Passaggio array multidimensionale

- Dato che nelle chiamate di funzioni viene passato solo l'indirizzo alla prima locazione dell'array, un parametro array può essere specificato usando indifferentemente la notazione degli array o dei puntatori
- Le seguenti scritture sono equivalenti:  
`int f(int arr[dim1][dim2]...[dimN]) ;`  
`int f(int arr[][dim2]...[dimN]) ;`  
`int f(const int *arr[dim2]...[dimN]) ;`

# Funzione che restituisce un array

- Una funzione non può restituire direttamente un array ma può restituire un puntatore al primo elemento dell'array
- Un errore tipico che si commette in tale situazione è la restituzione di un puntatore ad una variabile locale che viene rimossa dalla memoria al terminare della funzione:

```
int *times(int a[10], int k) {  
    int b[10];  
    for (int i=0; i<10; i++)  
        b[i]=a[i]*k;  
    return b; } //errore b locale!
```

## Esempio di funzione con risultato array

```
//ritorna puntatore ad array int[90]
int *ruota_maxritardo
    (int ritardi[11][90])
{
    int ruota_con_max, max=-1;
    for (int r=0; r<11; r++)
        for (int i=0; i<5; i++)
            if (ritardi[r][i]>max) {
                max=ritardi[r][i];
                ruota_con_max=r; }
    return ritardi[ruota_con_max];
}
```

# Ordinamento BubbleSort di array

```
void BubbleSort (int v[], int n)
{
    for (int k=n-1; k>0; k--)
        for (int i=0; i<k; i++)
            if (v[i] > v[i+1])
                swap(v[i], v[i+1]);
}
```

# Ordinamento BubbleSort ottimizzato

```
void BubbleSortOpt (int v[], int n)
{ enum bool {FALSE, TRUE};
  bool sorted = FALSE;
  for (int k=n-1;
       k>0 && sorted==FALSE; k--)
  { sorted = TRUE;
    for (int i=0; i<k; i++)
      if (v[i] > v[i+1])
      { sorted = FALSE;
        swap(v[i], v[i+1]);
      }
  }
}
```

# Ordinamento QuickSort di array

```
void QuickSort1 (int v[],
                  int primo, int ultimo) {
    if (primo < ultimo) {
        int p=primo, u=ultimo+1, pivot=v[primo];
        do {
            while (v[++p] < pivot);
            while (v[--u] > pivot);
            if (p < u) swap(v[p], v[u]);
        } while (p < u);
        swap(v[primo], v[u]);
        QuickSort1(v, primo, u-1);
        QuickSort1(v, u+1, ultimo);
    }
}
```

```
void QuickSort (int v[], int n)
{ QuickSort1(v, 0, n-1); }
```

# Stringhe

- Una stringa è un array di **char**, il cui ultimo elemento è il carattere nullo (`'\0'`)
- L'inizializzazione di una variabile stringa si può effettuare utilizzando una costante stringa  
Esempio: `char stringa[] = "Ciao";`
- L'array stringa contiene 5 elementi: i 4 caratteri della costante stringa e il carattere nullo che viene inserito automaticamente
- Nota: dato che negli array non è definito l'assegnamento, l'uso di una costante stringa per specificare il valore di una stringa è permesso solo nell'inizializzazione



# Input e Output di Stringhe (1)

- Gli operatori di ingresso e di uscita accettano una stringa come argomento
- L'operatore di ingresso legge caratteri dallo stream di ingresso e li memorizza in sequenza finchè non incontra un carattere di spaziatura
- L'occorrenza di tale carattere, che non viene letto, causa il termine dell'operazione e la memorizzazione nella stringa del carattere nullo dopo l'ultimo carattere letto

## Input e Output di Stringhe (2)

- L'operatore di uscita scrive i caratteri della stringa (escluso il carattere nullo finale) sullo stream di uscita
- Il seguente esempio legge una stringa al massimo di 255 caratteri (più il carattere nullo finale) e la stampa:

```
char buffer[256];  
cin >> buffer;  
cout << buffer;
```

# Funzioni della libreria `<cstring>`

Nelle funzioni `s` e `t` sono stringhe e `c` è un carattere

`strcpy(s, t)` copia `t` in `s` e restituisce `s`

`strcat(s, t)` concatena `t` al termine di `s` e restituisce `s`

`strcmp(s, t)` confronta `s` e `t` e restituisce un valore negativo, nullo o positivo  
se `s` è alfabeticamente minore, uguale o maggiore di `t`

`strchr(s, c)` restituisce un puntatore alla prima occorrenza di `c` in `s`, oppure 0 se `c` non si trova in `s`

`strrchr(s, c)` come sopra ma per l'ultima occorrenza di `c` in `s`

`strlen(s)` restituisce la lunghezza di `s`

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Strutture

- Una struttura è una  $n$ -upla ordinata di elementi, detti membri o campi, ciascuno dei quali ha uno specifico tipo, nome e valore
- Sintassi (definizione di un nuovo tipo):

```
struct new_struct_id {  
    tipo1 campo1;  
    ...  
    tipon campon; } ;
```

# Definizione Variabile di tipo `struct`

Può essere definita nei seguenti modi:

- `new_struct_id var_id;`
- `struct new_struct_id {  
 tipo1 campo1;  
 ...  
 tipon campon; } var_id;`
- `struct {  
 tipo1 campo1;  
 ...  
 tipon campon; } var_id;`
  - quest'ultima è detta struttura anonima

## Esempi di Strutture

```
struct data {  
    int giorno, mese, anno;  
};
```

```
struct persona {  
    char nome[20];  
    char cognome[25];  
    char comune_nascita[25];  
    data data_nascita;  
    enum { F, M } sesso;  
};
```

# Accesso ai campi di una struttura

- Se `s` è una struttura e `field` è un identificatore di un campo, allora `s.field` denota il campo della struttura
- Se `ps` è un puntatore ad una struttura avente `field` come identificatore di campo, allora è possibile scrivere `ps->field` al posto di `(*ps).field`
- Esempio:  

```
struct complex { double re, im; };  
complex c; complex *pc = &c;  
c1.re = 2.5; pc->im = 3;
```



# Assegnazione di Strutture

- A differenza degli array, l'assegnazione è definita per le variabili di tipo **struct**
- L'assegnazione di strutture avviene per valore e quindi vengono copiati tutti i valori dei membri

- Esempio:

```
struct complex { double re, im; };  
complex c1, c2;  
c1.re = 2.5; c1.im = 3;  
c2 = c1; //assegnazione di struct
```

# Assegnazione di array tramite struct

- Per gestire gli array in modo che possano essere copiati, si può incapsulare un tipo array come membro di una **struct**

- Esempio:

```
struct int_array { int ia[3]; };  
int_array sa, sb;  
sa.ia[0]=1;sa.ia[1]=2;sa.ia[2]=3;  
sb = sa; //l'array viene copiato!  
cout << sb.ia[0] << sb.ia[1]  
      << sb.ia[2] << endl;
```

# Strutture ricorsive

- La seguente definizione non è lecita:

```
struct S
{ int value;
  S next; //definizione circolare!
};
```

- La seguente definizione è lecita:

```
struct S
{ int value;
  S *next;
};
```

Infatti ogni puntatore occupa lo stesso spazio di memoria indipendentemente dal suo tipo.

# Strutture mutuamente ricorsive (1)

- La seguente definizione non è lecita:

```
struct S1
{ int value;
  S2 *next; //S2 ancora indefinito
};
```

```
struct S2
{ int value;
  S1 *next;
};
```

- S2 non è stato ancora definito al momento della definizione di S1

## Strutture mutuamente ricorsive (2)

- Dichiarando prima S2 risulta invece lecita:  
`struct S2; // dichiarazione di S2`

```
struct S1  
{ int value;  
  S2 *next; // Ok!  
};
```

```
struct S2 // definizione di S2  
{ int value;  
  S1 *next;  
};
```

# Unioni

- Le unioni sono dichiarate e usate con la stessa sintassi delle strutture, eccetto la parola chiave `union` al posto di `struct`
- I campi di una unione sono in alternativa tra loro e rappresentano diverse “interpretazioni” di un’unica area di memoria
- L’unione occupa in memoria lo spazio richiesto dal suo membro di dimensioni maggiori, in quanto un solo membro alla volta è contenuto in memoria

## Esempio di unione

```
union numero {int i; double d;};  
numero n1, n2;  
n1.i = 10;  
n2.d = 3.14; // n2 ora ha un double  
cout << n1.i << n2.d;  
n2.i = n1.i; // n2 ora ha un int  
cout << n2.i; // ok  
cout << n2.d; /* attenzione!  
                n2 contiene un int ma viene  
                interpretato come double */
```

# Strutture varianti

- È possibile definire una struttura variante avente dei membri di tipo `union`

- Esempio:

```
union ident {  
    int matricola; char nome[10]; }  
enum idtype {MATRICOLA, NOME};  
struct identita {  
    idtype it; ident id; }  
struct compito {  
    identita studente;  
    int voto };
```



# Esempio utilizzo strutture varianti

- `structvar.cc`

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Uso memoria dinamica

- Non sempre è possibile stabilire a priori (in maniera statica) le dimensioni delle strutture dati
- In C++ è possibile gestire la memoria anche dinamicamente, ovvero durante l'esecuzione del programma
- La gestione dinamica della memoria consente di allocare porzioni di memoria nello **store**, ovvero in un area di memoria esterna allo stack di esecuzione del programma
- L'accesso avviene tramite puntatori

# Puntatore a memoria non allocata

```
int main() {  
    int a = 5; //variabile statica;  
    int *p;  
    *p = 3; /* errore! p punta ad  
              un'area non allocata */  
    p = &a; /* p punta all'area di  
             memoria riservata staticamente  
             alla variabile a */  
    *p = 4; //corretto: a ora vale 4  
    return 0; }
```

# Operatore new

- Sintassi:  
`new tipo;`  
`new tipo[dimensione] ;` (per gli array)
- L'operatore **new** alloca un'area di memoria atta a contenere un oggetto del tipo specificato e ritorna un puntatore a tale area di memoria
- Esempio:  
`int *p; char *stringa;`  
`p = new int;`  
`stringa = new char[30];`

# Operatore delete

- Sintassi:  
`delete puntatore;`  
`delete[] puntatore;` (per gli array)
- L'operatore `delete` dealloca l'area di memoria dinamica puntata dal puntatore
- Esempio:  

```
int *p; char *stringa;  
p = new int;  
stringa = new char[30];  
delete p;  
delete[] stringa;
```

# Deallocazione

- Un oggetto creato dinamicamente resta allocato finchè:
  - non viene esplicitamente deallocato con l'operatore **delete**
  - il programma non termina
- La memoria non esplicitamente deallocata con l'operatore **delete**, può risultare (a seconda del sistema operativo, es. windows) non più disponibile per altri programmi

# Vantaggi memoria dinamica

- Il meccanismo di gestione della memoria dinamica consente:
  - la gestione efficiente delle risorse di memoria in modo da allocare solo lo spazio realmente necessario
  - la creazione di strutture dati dinamiche (es. liste, alberi, ...)



# Allocazione dinamica Array

- Consente di creare a run-time array di dimensioni diverse a seconda della necessità

- Esempio:

```
#include <iostream>
int main() {
    int a[], n;
    cout<<"Quanti numeri?"; cin>>n;
    a = new int[n]; //allocazione
    for (int i=0; i<n; i++) {
        cout << endl << i+1 << ": ";
        cin >> a[i]; };
    delete[] a;    //deallocazione
    return 0;}
```

# Allocazione dinamica Stringhe

- Consente di creare stringhe opportunamente dimensionate a seconda della necessità

- Esempio:

```
#include <iostream>
#include <cstring>
int main() {
    char sa[200], sb[200] sc[];
    cin >> sa >> sb;
    sc = new char[strlen(sa)+
        strlen(sb)+1]; //allocazione
    strcpy(sc,sa); strcat(sc,sb);
    delete[] sc; //deallocazione
    return 0;}
```

## Verifica corretta allocazione

- Il buon esito nell'uso dell'operatore **new** può essere controllato dal programmatore, usando la funzione **set\_new\_handler** della libreria **<new>**
- È possibile specificare una funzione che viene chiamata in caso di fallimento nell'allocare un'area di memoria

## Esempio di utilizzo `set_new_handler`

```
#include <iostream>
#include <cstring>
void noMemory(size_t size) {
    cerr << "Memoria esaurita!\n";
    cerr << "Impossibile allocare "
         << size << " bytes\n";
}
int main() {
    set_new_handler(noMemory);
    for (int i=0; i<1024; i++) {
        char *p = new char[1024*1024];
        cout << i << endl; }
    return 0;}
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# **Strutture dati e Algoritmi**

**© Alessandro Armando, Enrico Giunchiglia &  
Roberto Sebastiani**

**Realizzazione: Daniele Zini; Roberto Sebastiani &  
Sabrina Recla**

**Versione 2.0**

# Strutture dati fondamentali

- La gestione dinamica della memoria consente di avere strutture dati realizzate in maniera efficiente, sia per l'occupazione ottimizzata della memoria e sia per l'accesso a tali dati
- Le strutture dati fondamentali più comunemente utilizzate sono:
  - le liste
  - gli alberi
  - i grafi

# Liste semplici

- Una lista è una struttura dati, formata da elementi dello stesso tipo collegati in catena, la cui lunghezza varia dinamicamente
- Ogni elemento di una lista semplice è costituito da uno o più campi contenenti informazioni e da un campo puntatore contenente l'indirizzo dell'elemento successivo

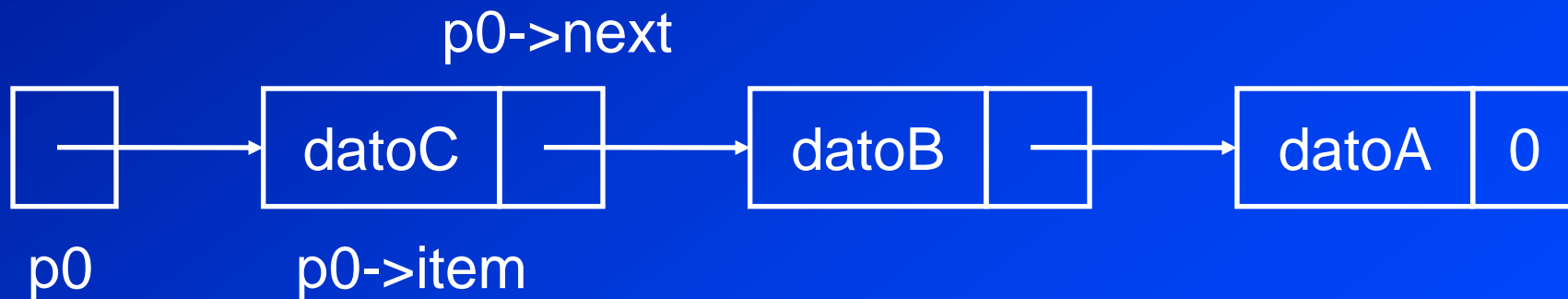
- Esempio:

```
struct node {  
    int item;  
    node *next;  
}
```



# Organizzazione di una lista

- Il primo elemento è indirizzato da una variabile puntatore, detto puntatore della lista
- Il campo puntatore dell'ultimo elemento contiene il puntatore nullo



## Creazione di una lista

```
//crea una lista di n elementi
node *crealista(int n)
{
    node *p0 = 0; node *p;
    for (int i=0; i<n; i++) {
        p = new node;
        cin >> p->item;
        p->next = p0;
        p0 = p;
    }
    return p0;
}
```

# Operazioni sulle liste

- Data una lista, si possono effettuare operazioni di estrazione e di inserimento di nuovi elementi, secondo specifiche regole di gestione della lista stessa
- Tipiche operazioni sulle liste sono:
  - inserimento ed estrazione dalla testa
  - inserimento ed estrazione dalla coda
  - inserimento in una lista ordinata
  - estrazione di un particolare elemento

## Inserimento in testa

```
void ins_testa(node *&p0, int a)
{
    node *p = new node;
    p->item = a;
    p->next = p0;
    p0 = p;
}
```

## Estrazione dalla testa

```
int estr_testa(node *&p0, int& a)
{
    node *p = p0;
    if (p0!=0) {
        a = p0->item ;
        p0 = p0->next;
        delete p;
    }
    return (p0!=0) ;
}
```

## Inserimento in coda

```
void ins_coda(node *&p0, int a)
{
    node *q; node *p;
    p = new node;
    p->item = a;
    p->next = 0;
    if (p0==0) p0 = p;
    else {
        for (q=p0;q->next!=0;q=q->next) ;
        q->next = p;
    }
}
```

## Estrazione dalla coda

```
int estr_coda(node *&p0, int& a)
{
    node *p = 0; node *q;
    if (p0!=0) {
        for (q=p0;q->next!=0;q=q->next)
            p = q;
        a = q->item;
        if (q==p0) p0 = 0;
        else p->next = 0;
        delete q;
    }
    return (p0!=0);
}
```

# Inserimento in una lista ordinata

```
void ins_ord(node *&p0, int v)
{
    node *p = 0; node *q; node *r;
    for (r = p0; r!=0 && r->item < v;
         r = r->next) p = r;
    q = new node;
    q->item = v;
    q->next = r;
    if (p==0) p0 = q;
    else p->next = q;
};
```



## Estrazione di un particolare elemento

```
int estr_elem(node *&p0, int a)
{
    node *p = 0; node *q;
    for (q = p0; q!=0 && q->item!=a;
        q = q->next) p = q;
    if (q==0) return 0;
    if (q==p0) p0 = q->next;
    else p->next = q->next;
    delete q;
    return 1;
}
```

# Liste con puntatore ausiliario

- È possibile gestire una lista con un puntatore ausiliario, avendo sia un puntatore al primo che all'ultimo elemento della lista
- In questo modo è possibile gestire una lista effettuando, senza scorrere la lista, l'inserimento e l'estrazione in coda
- Esempio:

```
struct list {  
    node *first;  
    node *last;  
}
```

## Creazione di una lista

```
//crea una lista di n elementi
list *creafifo(int n)
{
    list f = {0,0}; node *p;
    if (n>=1) {
        p = new node; cin >> p->item;
        p->next = f.first;
        f.first = p; f.last = p; }
    for (int i=2; i<=n; i++) {
        p = new node; cin >> p->item;
        p->next=f.first; f.first=p; }
    return f;
}
```

## Estrazione dalla testa

```
int estr_testa(list *&l, int& a)
{
    node *p = l.first;
    if (p == 0) return 0;
    a = p->item ;
    l.first = p->next;
    delete p;
    if (l.first == 0) l.last = 0;
    return 1;
}
```

## Inserimento in coda

```
void ins_coda(list *&l, int a)
{
    node *p = new node;
    p->item = a;
    p->next = 0;
    if (l.first==0) {
        l.first = p; l.last=p;
    }
    else {
        l.last->next = p; l.last = p;
    }
}
```

# Liste complesse

- Utilizzando variabili dinamiche è possibile costruire e gestire liste i cui elementi contengono più di un campo puntatore
- Ad esempio, si possono avere liste doppie in cui ogni elemento punta sia al precedente che al successivo, con conseguente possibilità di scorrere la lista nei due sensi
- Il tipo degli elementi può essere definito come:  

```
struct node
{
    int item;
    node *prev;
    node *next; }
```

# Alberi binari

- Un albero binario può essere realizzato in C++ associando ad ogni nodo una struttura avente un campo informazione e due campi puntatore, che puntano rispettivamente al ramo sinistro e destro

- Esempio:

```
struct node
{
    char item;
    node *left;
    node *right;
}
```

# Inserimento in un albero binario

```
node insert(node *n, char v)
{
    if (n==0) {
        n = new node; n->item = v;
        n->left = 0; n->right = 0; }
    else if (v <= n->item)
        n->left = insert(n->left, v);
    else if (v > n->item)
        n->right = insert(n->right, v);
};
```



## Visita in ordine simmetrico

- A partire dalla radice, viene visitato prima il sottoalbero sinistro, poi la radice e infine il sottoalbero destro

- Esempio:

```
void print_tree(node *t)
{
    if (t!=0) {
        print_tree(t->left);
        cout << t->item;
        print_tree(t->right);
    }
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# Scope, Visibilità e Durata

- **Scope**: è la porzione di testo nel codice sorgente in cui è attiva una dichiarazione
- **Visibilità**: in caso di blocchi nidificati, una dichiarazione di una variabile anonima può rendere localmente inaccessibile la variabile dichiarata in un blocco superiore limitandone la visibilità
- **Durata**: si intende il lasso temporale in cui la locazione di memoria associata alla variabile rimane allocata in memoria

# Scope di una variabile

- Scope a livello di file:  
la dichiarazione è attiva in tutto il file
- Scope locale:  
la dichiarazione è attiva localmente ad una funzione o ad un blocco di istruzioni

# Evidenziazione dello scope nel codice

```
const float pi=3.1415;  
// scope a livello di file  
int x;  
// scope a livello di file  
  
int f(int a, double x); // scope locale  
{ int c;                // scope locale  
    .  
    .  
}  
int main()  
{ char pi; // scope locale  
    .  
    .  
}
```

# Evidenziazione Visibilità nel codice

```
1.  const float pi=3.1415;
2.  int x;                      // 1
3.
4.  int f(int a, double x) // 1, 2, 3
5.  {  int c;                  // 1, 4
6.      .                      // 1, 4, 5
7.      .
8.  }
9.
10. int main()                 // 1, 2, 4
11. {  char pi;                // 1, 2, 4, 10
12.     .
13.     .
14. }
```

# Durata

- Static: variabili globali, funzioni e variabili dichiarate con specificatore **static**  
Durata: tutto il periodo di esecuzione del programma
- Local: variabili locali a un blocco (hanno sempre scope locale)  
Durata: solo il periodo di tempo necessario ad eseguire il blocco in cui sono dichiarate
- Dynamic: oggetti memorizzati nello *store*  
Durata: gestita dalle chiamate a **new** e **delete**

# Evidenziazione Durata nel codice

```
const float pi=3.1415;  
// static (e' globale)  
  
int x;  
// static (e' globale)  
  
int f(int a, double x)  
// f static (e' funzione)  
{ int c;  
  // a, x, c local  
  
  static double d;  
  // static con scope locale  
  ...  
}
```



## Lo specificatore `static`

- Tramite lo specificatore `static` è possibile forzare la durata di una variabile
- Nel caso di variabili `static` un'eventuale inizializzazione nella dichiarazione viene eseguita una sola volta all'atto dell'inizializzazione del programma
- Lo specificatore `static` applicato ad un oggetto di scope globale (es. funzioni, variabili, costanti globali) ha l'effetto di restringere la visibilità dell'oggetto al solo file in cui occorre la definizione

## Esempio di utilizzo variabile static

```
#include<iostream>
void inc()
{ static int n=4;
  cout << "Il valore di n e': "
        << n++ << endl; }
int main()
{ for(int i=0; i<3; i++)
  inc();
  return 0; }
```

Produce in output:

Il valore di n e': 4

Il valore di n e': 5

Il valore di n e': 6

## Lo specificatore **extern**

- Lo specificatore **extern** consente di dichiarare degli oggetti che non sono definiti nello stesso file
- Ciò consente al compilatore di verificare la ben formatezza delle espressioni che coinvolgono tali oggetti e di stabilire le dimensioni delle corrispondenti aree di memoria
- Si intende che l'oggetto deve essere definito altrove e sarà cura del collegatore (linker) di associare le variabili dichiarate alle corrispondenti definizioni

# Esempio di utilizzo di extern

```
//file main.cc
```

```
extern int x; // dichiarazione di x
```

```
float y;      // definizione di y
```

```
extern int f(float z);
```

```
// dichiarazione di f
```

```
//file funzioni.cc
```

```
int x;        // definizione di x
```

```
int f(float z){ // definizione di f
```

```
...
```

```
};
```

# Dichiarazione e Definizione

- Un oggetto può essere *dichiarato* quante volte si vuole mentre può essere *definito* una volta sola
- Un oggetto dichiarato più volte deve essere dichiarato sempre nello stesso modo
- Ogni definizione è anche una implicita dichiarazione

# Programmazione modulare

- I programmi di grosse dimensioni possono essere organizzati su più file, ovvero strutturati in maniera modulare
- Il meccanismo alla base della programmazione modulare è il meccanismo di dichiarazione (**extern**)
- E' compito del linker associare ad un oggetto dichiarato in un file l'oggetto corrispondente definito in un altro file

# Programma su un solo file

```
#include <iostream>

void f1() { ... };
void f2() { ... };

int main() {
    ...
    switch(scelta) {
        case 1: f1(); break;
        case 2: f2(); break;
        ...
    }
}
```

# Programma su più file (1)

```
//file main.cc  
#include <iostream>  
#include "fmain.h"  
  
int main() {  
    ...  
    switch(scelta) {  
        case 1: f1(); break;  
        case 2: f2(); break;  
        ...  
    }  
}
```



## Programma su più file (2)

```
//file fmain.h
```

```
void f1();
```

```
void f2();
```

```
//file fmain.cc
```

```
void f1() { ... };
```

```
void f2() { ... };
```

# Indice

1. Modello di un calcolatore
2. Sviluppo di un programma
3. Sintassi del C++
4. Variabili, Costanti e Tipi
5. Puntatori e riferimenti
6. Istruzioni semplici
7. Ingresso e uscita dei dati
8. Istruzioni strutturate
9. Funzioni
10. Array
11. Strutture e unioni
12. Gestione dinamica della memoria
13. Strutture Dati e Algoritmi
14. Visibilità e moduli
15. Astrazione dei dati

# **Astrazione dei dati**

**© Alessandro Armando, Enrico Giunchiglia &  
Roberto Sebastiani**

**Realizzazione: Daniele Zini; Roberto Sebastiani &  
Sabrina Recla**

**Versione 2.0**

# Lo stack

- Uno stack è composto da uno spazio in cui gli elementi vengono mantenuti con modalità LIFO (Last In First Out)
- Le operazioni tipiche definite su uno stack sono:
  - `init()`    inizializza lo stack
  - `push(int)`    mette elemento sullo stack
  - `pop()`    toglie l'elemento in cima allo stack
  - `top(int &)`    ritorna l'elemento in cima allo stack
- Le operazioni non fanno riferimento a nessuna particolare realizzazione della struttura dati

## File di intestazione "stack.h"

```
/* dichiarazioni per la gestione  
dello stack */
```

```
enum retval { FAIL, OK };
```

```
enum boolean { FALSE, TRUE};
```

```
void init ();
```

```
retval top (int &);
```

```
retval push (int);
```

```
retval pop ();
```

# Implementazione con array (1)

```
#include "stack.h"

static const int dim = 100;
static int indice, elem[dim];

static boolean empty ()
{ return (boolean) (indice==0) ; }

static boolean full ()
{ return (boolean) (indice==dim) ; }

void init () { indice = 0 ; }
```

## Implementazione con array (2)

```
retval top (int &n)
{ if (empty()) return FAIL;
  n=elem[indice-1]; return OK; }

retval push (int n)
{ if (fullp()) return FAIL;
  elem[indice++] = n; return OK; }

retval pop ()
{ if (empty()) return FAIL;
  indice--; return OK; }
```

# Programma che utilizza lo stack (1)

```
#include <iostream>
#include "stack.h"

int main()
{ char res;
  init();
  do
  { cout << "Operazioni possibili:\n"
    << "push (u)\npop (o)\n"
    << "top (t)\ninit (i)\n"
    << "fine (f)\n";
    cin >> res;
```



## Programma che utilizza lo stack (2)

```
switch(res)
{ int num;

  case 'u':
    cout << "Valore: ";
    cin >> num;
    if (push(num) == FAIL)
      cout << "memoria esaurita\n";
    break;

  case 'o':
    if (pop() == FAIL)
      cout << "stack vuoto\n";
    break;
```

## Programma che utilizza lo stack (3)

```
case 't':
    if (top(num) == FAIL)
        cout << "top: stack vuoto\n";
    else
        cout << "Top = "
               << num << endl;
    break;

case 'i':
    init(); break;

default : break; }

} while (res != 'f');
return 0;}
```

# Astrazione dei dati

- Dato un insieme di funzioni per gestire un dato astratto è possibile realizzare differenti implementazioni, accessibili attraverso la stessa interfaccia (dichiarata in un file di intestazione) senza modificare il programma
- Nell'esempio dello stack, basterà compilare, al posto del file con l'implementazione basata sugli array, un altro file contenente un'altra implementazione
- Il file di intestazione (stack.h) e il programma (main) rimarranno quindi inalterati

# Implementazione con struttura array (1)

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

static const int dim = 100;

struct stack
{ int indice;
  int elem[dim];
};

static stack S;
```

## Implementazione con struttura array (2)

```
static boolean empty ()
{
    return (boolean) (S.indice==0) ;
}

static boolean fullp ()
{
    return (boolean) (S.indice==dim) ;
}

void init ()
{
    S.indice = 0 ;
}
```

## Implementazione con struttura array (3)

```
retval top (int &n)
{ if (empty()) return FAIL;
  n=S.elem[S.indice-1];
  return OK; }
```

```
retval push (int n)
{ if (fullp()) return FAIL;
  S.elem[S.indice++] = n;
  return OK; }
```

```
retval pop ()
{ if (empty()) return FAIL;
  S.indice--; return OK; }
```

# Realizzazione con liste concatenate (1)

```
#include "stack.h"
struct nodo { int val; nodo *next; };
static nodo *stack;

static boolean empty ()
{ return (boolean) (stack==NULL) ; }

void init () { stack=NULL; }

retval top (int &n)
{ if (empty()) return FAIL;
  n=stack->val;
  return OK; }
```

## Realizzazione con liste concatenate (2)

```
retval push (int n)
{
    nodo * np = new nodo;
    if (np==NULL) return FAIL;
    np->val = n; np->next = stack;
    stack = np; return OK; }

```

```
retval pop ()
{
    if (empty()) return FAIL;
    nodo *first=stack;
    stack=stack->next;
    delete first;
    return OK; }

```



# ESERCIZI

- Utilizzando solamente la struttura dati stack definita, scrivere un programma che legga una sequenza di interi e la stampi in ordine inverso
- A ciascuna delle implementazioni dello stack presentate, aggiungere due routine “print()” e “print\_reverse()” che stampino il contenuto dello stack rispettivamente in ordine di immissione ed in ordine inverso, SENZA modificarne il contenuto.

## ESERCIZI (2)

- Si realizzi uno stack di elementi del tipo:

```
struct value_type {  
    int code;  
    char text[1000];}
```

e si scriva un programma di prova che legga una sequenza di stringhe in ingresso, associ un codice secondo un qualche criterio, e le stampi in ordine inverso.

NOTA: gli elementi vanno passati tramite PUNTATORE.

# La coda

- Una coda è composta da uno spazio in cui gli elementi vengono mantenuti con modalità FIFO (First In First Out)
- Le operazioni tipiche definite su uno stack sono:
  - `init()`            inizializza la coda
  - `enqueue(int)`    mette elemento nella coda
  - `dequeue(int &)`   toglie l'elemento dalla coda
- Le operazioni non fanno riferimento a nessuna particolare realizzazione della struttura dati

# File di intestazione "queue.h"

```
// dichiarazioni per la gestione  
della coda di interi
```

```
enum retval { FAIL, OK };
```

```
enum boolean { FALSE, TRUE };
```

```
void init ();
```

```
retval enqueue(int);
```

```
retval dequeue(int &);
```

# Realizzazione con array

```
#include "queue.h"

static const int dim = 100;
static int head, tail, elem[dim];

static int next(int index)
{
    return (index+1)%dim;
}
```

## Realizzazione con array (2)

```
static boolean empty()  
{  
    return (boolean) (tail==head) ;  
}
```

```
static boolean fullp()  
{  
    return (boolean)  
    (next(tail)==head) ;  
}
```

## Realizzazione con array (3)

```
void init()  
{  
    tail=head=0;  
}
```

## Realizzazione con array (4)

```
retval enqueue (int n)
{
    if (fullp())
        return FAIL;
    elem[tail] = n;
    tail = next(tail);
    return OK;
}
```



## Realizzazione con array (5)

```
retval dequeue(int & n)
{
    if (empty())
        return FAIL;
    n = elem[head];
    head = next(head);
    return OK;
}
```

# Programma che utilizza la coda (1)

```
#include <iostream>
#include "queue.h"
```

```
int main()
{
    char res;
    int num;
    init();
    do {
```

## Programma che utilizza la coda (2)

```
cout << "\nOperazioni possibili:\n"
      << "Enqueue (e)\n"
      << "Dequeue (d)\n"
      << "Fine (f)\n";

cin >> res;

switch (res) {
```

## Programma che utilizza la coda (3)

```
case 'e':  
    cout << "Valore: ";  
    cin >> num;  
    if (enqueue(num) == FAIL)  
        cout << "Coda piena\n";  
    break;
```

## Programma che utilizza la coda (4)

```
case 'd':  
    if (dequeue(num) == FAIL)  
        cout << "Coda vuota\n";  
    else  
        cout << "Val: " <<  
            num << endl;  
    break;
```

## Programma che utilizza la coda (5)

```
case 'f':  
    break;  
default:  
    cout << "Valore errato!\n";  
}  
} while (res != 'f');  
Return 0;}
```

# Realizzazione tramite lista concatenata

- Stesso header file “queue.h” della realizzazione come array
- Lista di nodi concatenata tramite puntatori
- Inserimento in coda, estrazione dalla testa
- Due puntatori:
  - head, punta al primo elemento inserito
  - tail, punta all’ultimo elemento inserito
- Il caso “coda vuota” va gestito ad hoc

# Realizzazione con lista concatenata

```
#include "queue.h"
#include <iostream>

struct node {
    int val;
    node * next;
};
```



## Realizzazione con lista concatenata (2)

```
struct queue {  
    node * head;  
    node * tail;  
};  
  
static queue Q;  
  
static boolean empty () {  
    return (boolean)  
        (Q.head == NULL) ;  
}
```

## Realizzazione con lista concatenata (3)

```
void init()
{
    Q.head = Q.tail = NULL;
}
```

```
retval enqueue(int n)
{
    node * np = new node;
    if (np==NULL)
        return FAIL;
```

## Realizzazione con lista concatenata (4)

```
np->val=n;
np->next=NULL;
if (empty())
    Q.head=Q.tail=np;
else {
    Q.tail->next=np;
    Q.tail=np;
}
return OK;
}
```

## Realizzazione con lista concatenata (5)

```
retval dequeue(int &n)
{
    node * first;
    if (empty())
        return FAIL;
    first = Q.head;
    n = first->val;
    Q.head = Q.head->next;
    delete first;
```

## Realizzazione con lista concatenata (6)

```
if (empty())  
    Q.tail = NULL;  
return OK;  
}
```

# ESERCIZI

Si realizzi una struttura dati “coda con priorit ” in cui sia possibile inserire oggetti di tipo messaggio ed estrarli in modo FIFO per classi di priorit  (prima quelli a priorit  9, poi 8, ...).

```
struct messaggio {  
    char testo[1000];  
    int priorit ; // 0,...,9  
};
```

Si scriva poi un programma di prova.

(Suggerimento: usare un array [0...9] di code.)

# Alberi binari

- Un albero binario (albero di ricerca binaria) è una struttura in cui è possibile mantenere collezioni ordinate di oggetti.
  - Accesso ai dati efficiente (sotto particolari condizioni)
  - Struttura intrinsecamente ricorsiva:  
un albero è un link che può essere vuoto oppure puntare ad un nodo che contiene un campo informazione e due alberi.
- > primitive di accesso ricorsive

## Alberi binari: tree.h

```
struct node
{ char item;
  node *left;
  node *right; };

void init(node *&);
node * insert(node *, char);
node * cerca (node *,char);
void print_ordered(node *);
void print_indented(node *,int);
```



# Alberi binari: (1)

```
#include <iostream>
#include "tree.h"

void init(node *& t)
{
    t=NULL;
}
```

## Alberi binari: (2)

```
// inserimento
node * insert(node *t, char v)
{
    if (t==NULL) {
        t = new node;
        t->item = v;
        t->left = NULL;
        t->right = NULL;
    }
}
```

## Alberi binari: (3)

```
else if (v <= t->item)
    t->left = insert(t->left, v);
else if (v > t->item)
    t->right = insert(t->right, v);
return t;
};
```

## Alberi binari: (4)

```
node * cerca (node * t, char elem)
{
    if ((t==NULL) || (elem==t->item))
        return t;
    if (elem <= t->item)
        return cerca(t->left, elem);
    else if (elem > t->item)
        return cerca(t->right, elem);
}
```

## Alberi binari: (5)

```
// stampa in ordine  
void print_ordered(node * t)  
{  
    if (t!=NULL) {  
        print_ordered(t->left);  
        cout << t->item << endl;  
        print_ordered(t->right);  
    }  
}
```

## Alberi binari: (6)

```
void print_indented(node *t,  
                    int depth)  
{ int i;  
  if (t!=NULL) {  
    print_indented(t->right,depth+1) ;  
    for (i=0;i<depth;i++)  
      cout << "  ";  
    cout << t->item << endl;  
    print_indented(t->left,depth+1) ;  
  } }
```

# Alberi binari: programma main di prova

```
#include <iostream>
```

```
#include "tree.h"
```

```
int main()
```

```
{
```

```
    char res, val;
```

```
    node * t;
```

```
    init(t);
```

```
    do {
```

## Alberi binari: programma main (2)

```
cout << "\nOperazioni:\n"
      << "Inserimento (i)\n"
      << "Ricerca (r)\n"
      << "Stampa ordinata (s)\n"
      << "Stampa indentata (e)\n"
      << "Fine (f)\n";

cin >> res;
switch (res) {
```



## Alberi binari: programma main (3)

```
case 'i':  
    cout << "Val? : ";  
    cin >> val;  
    t=insert(t,val);  
    break;  
case 's':  
    cout << "Stampa ordinata:\n";  
    print_ordered(t);  
    break;
```

## Alberi binari: programma main (4)

```
case 'r':  
    cout << "Val? : ";  
    cin >> val;  
    tmp=cerca(t,val) ;  
    if (tmp!=NULL)  
        cout << "Valore: " <<  
            val << endl;  
    else cout << "Non trovato!\n";  
    break;
```

## Alberi binari: programma main (5)

```
case 'e':  
    cout << "Stampa indentata:\n";  
    print_indented(t, 0);  
    break;  
case 'f':  
    break;  
default:  
    cout << "Optione errata\n";  
}} while (res != 'f');  
return 0; }
```

## Esercizi

- Realizzare tramite un albero binario una rubrica di persone ordinata per cognome:

```
struct persona {  
    char nome[20];  
    char cognome[20];  
    char indirizzo[100];  
    int ntel;  
}
```

(Suggerimento: usare puntatori a persona come “item”).

## Esercizi (2)

- Come nell'esercizio precedente, dove sia possibile cercare e stampare sia in ordine di nome sia di cognome.  
(Suggerimento: usare una struct di due alberi.)

# Limiti della programmazione strutturata

- Nell'implementare un dato astratto con la dichiarazione di variabili statiche, come nell'esempio precedente, si ha la limitazione di poter avere accesso ad una sola istanza del dato
- Più in generale, la programmazione strutturata non fornisce direttamente la possibilità di gestire una molteplicità di oggetti generici che forniscono un insieme di operazioni nascondendo nel contempo la struttura dati e l'implementazione
- Soluzione: programmazione object-oriented