

# Il linguaggio C++

---

Prima parte

1



## In Questa Lezione

---

- Note sul C++
- Paradigmi di Programmazione
- Astrazione, Incapsulamento, Modularità, Gerarchia
- Oggetti e Classi
- Istanziamento ed Ereditarietà
- Generalizzazione/Specializzazione
- Classi Astratte
- Polimorfismo ed Ereditarietà Multipla

2



## Che cosa è il C++

---

- Il “padre” del C++ è Bjarne Stroustrup, che ha cominciato a sviluppare nel 1979 il linguaggio che è stato recentemente (1998) standardizzato
- Il C++ è un linguaggio di programmazione ibrido (programmazione procedurale, programmazione OO, programmazione generica)

3



## La Programmazione Procedurale

---

E' basata sull'idea di *decomposizione funzionale*:

- si scompone la funzionalità principale del sistema da sviluppare in funzionalità più semplici
- si termina la scomposizione quando le funzionalità individuate sono così semplici da permetterne una diretta implementazione come funzioni

4



## La programmazione OO

- S'individuano le *classi di oggetti* (entità del mondo reale o concettuale) che caratterizzano il dominio applicativo;
- Si individuano le modalità secondo cui gli oggetti devono interagire per realizzare le diverse funzionalità dell'applicazione
- Ogni classe è descritta da un'*interfaccia* che specifica il comportamento degli oggetti della classe
- L'interazione tra gli oggetti avviene attraverso uno scambio di messaggi

5



## La programmazione generica

- La *programmazione generica* consente di definire una funzione o una classe senza specificare il tipo di una o più delle sue entità (parametri, membri).
- Ad esempio, una funzione di ricerca in grado di operare di volta in volta su elementi di tipo diverso. Oppure una classe che descrive contenitori (lista, insieme, coda, ...) in grado di contenere oggetti di tipi qualsiasi.

6



## Astrazione

---

- L'*astrazione* è il processo che porta a estrarre le proprietà rilevanti di un'entità, ignorando i dettagli inessenziali
  - le proprietà estratte definiscono una vista dell'entità
  - una stessa entità può dar luogo a viste diverse
- Esempio: un'automobile
  - vista dal venditore
    - prezzo, durata della garanzia, colore, ...
  - vista dal meccanico
    - Numero di Km percorsi, usura dei pneumatici, ...

7



## Incapsulamento

---

- *Incapsulamento* significa nascondere e proteggere alcune informazioni di una certa entità (tipicamente un oggetto).
- L'accesso alle informazioni dell'entità incapsulata è possibile attraverso l'*interfaccia*
- Esempio: il driver di una stampante
  - L'interfaccia della stampante è costituita dal suo driver che la mette in grado di dialogare con il menu stampa di vari programmi. I dettagli di come funziona sono nascosti, per installarla e usarla non è necessario conoscere alcunché della sua struttura interna, è sufficiente il driver.

8



## La Gerarchia

- Una *gerarchia* deriva da un processo di classificazione e ordinamento di entità sulla base di una qualche relazione che sussiste tra di esse.
- Le gerarchie aiutano a comprendere i sistemi complessi. Ad esempio: la gerarchia di una società aiuta i dipendenti a comprendere la struttura della società e le posizioni al suo interno.

9



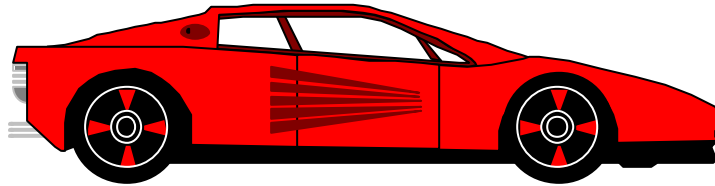
## Cosa è un oggetto

- Un oggetto è uno strumento attraverso cui è possibile modellare entità del mondo reale. Esso è caratterizzato da un nome (*un identificatore unico*) e da alcune *proprietà*, queste ultime si distinguono in:
  - *Dati (variabili)*: ne descrivono lo "stato";
  - *Metodi (funzioni)*: ne descrivono il "comportamento";
- Più oggetti interagiscono mediante uno scambio di messaggi. L'interazione si concretizza in una modifica dello stato o nell'esecuzione di un servizio

10



## Un esempio di Oggetto



### Funzioni

- Avviati
- Fermati
- Accelera
- ...

### Dati:

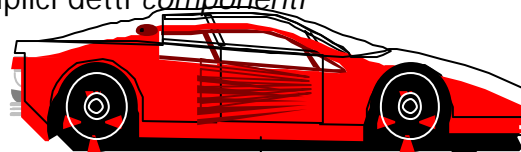
- Targa
- Colore
- Velocità
- Livello benzina
- ...

11



## Gli Oggetti sono composti da altri Oggetti

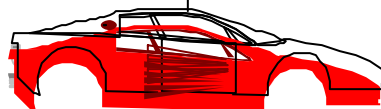
- Un oggetto è generalmente composto di oggetti più semplici detti *componenti*



automobile



ruota



carrozzeria



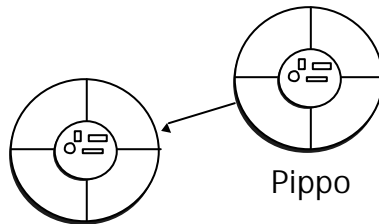
motore

12



## Gli oggetti ed i Messaggi

- Per ottenere funzionalità di una certa utilità e comportamenti complessi è necessaria l'interazione tra oggetti.
- Un messaggio può essere inteso come l'invocazione di un metodo di un oggetto da parte di un altro oggetto con relativo scambio di parametri

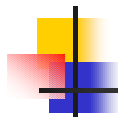


La bicicletta di Pippo

### I componenti di un Messaggio:

- L'oggetto a cui inviare il messaggio (*bicicletta*)
- Il nome del metodo da eseguire (*cambia marcia*)
- I parametri di cui il metodo ha bisogno (*marcia più bassa*)

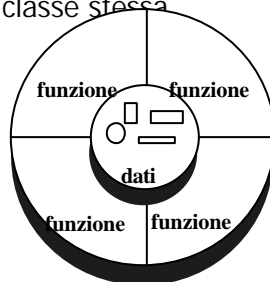
13



## Cosa è una Classe

- Un oggetto è una **istanza** di una **classe** di oggetti che condividono lo stesso stato e lo stesso comportamento (*N.B. lo stato di un'istanza è indipendente dallo stato di un'altra istanza*)
- La più importante caratteristica distintiva di una classe è l'insieme dei messaggi che possono essere inviati agli oggetti della classe stessa

*Alcuni elementi (dati e funzioni) sono pubblici, cioè visibili "all'esterno"*



*Altri elementi sono privati dell'oggetto, e quindi "nascosti" all'esterno*

14



## Un esempio di classe

- La classe Automobile
  - *parte pubblica*:
    - dà accesso a “ciò che l'auto può fare”
      - volante
      - blocchetto di accensione
      - pedale dell'acceleratore
      - ...
  - *parte privata*:
    - specifica “come l'auto fa ciò che può fare”
      - meccanica dello sterzo
      - elettromeccanica dell'avviamento
      - sistema di alimentazione e accensione
      - ...

15



## Una classe ha un'interfaccia

L'*interfaccia* specifica i *membri pubblici* che sono essenzialmente *funzioni membro* destinate ad elaborare i messaggi che possono essere inviati agli oggetti della classe

Quindi quello che prima avevamo indicato con parte pubblica lo indicheremo adesso con INTERFACCIA PUBBLICA

16



## Una classe ha un'implementazione

- L'*implementazione* di una classe descrive:
  - la rappresentazione concreta degli oggetti della classe
  - gli algoritmi delle funzioni membro
- Per usare una classe non occorre conoscerne l'implementazione, che può quindi essere nascosta e inaccessibile
- Quindi ciò che prima avevamo indicato con parte privata lo indicheremo adesso con IMPLEMENTAZIONE PRIVATA

17



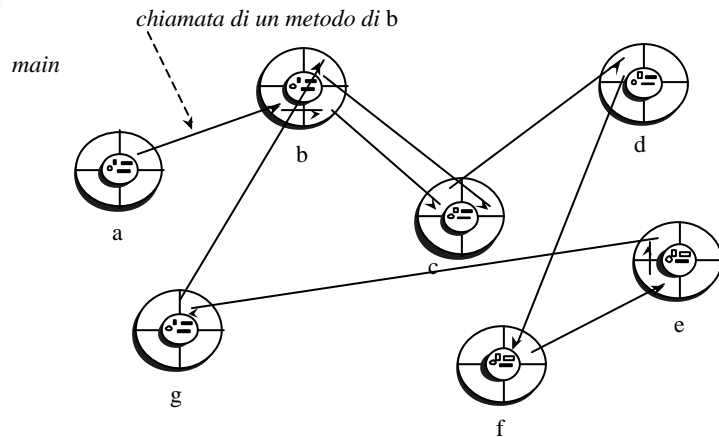
## I Sistemi ad Oggetti

- Un sistema ad oggetti è costituito da un insieme di oggetti che evolve durante l'esecuzione (nuovi oggetti sono creati, oggetti esistenti sono distrutti o modificati)
- Gli oggetti comunicano e interagiscono mediante scambio di messaggi:
  - un messaggio in arrivo attiva la corrispondente funzione membro dell'oggetto ricevente
  - questa può produrre:
    - Il cambiamento dello stato
    - l'esecuzione di un servizio
    - La restituzione di una informazione

18



## I Sistemi ad Oggetti - 2



19



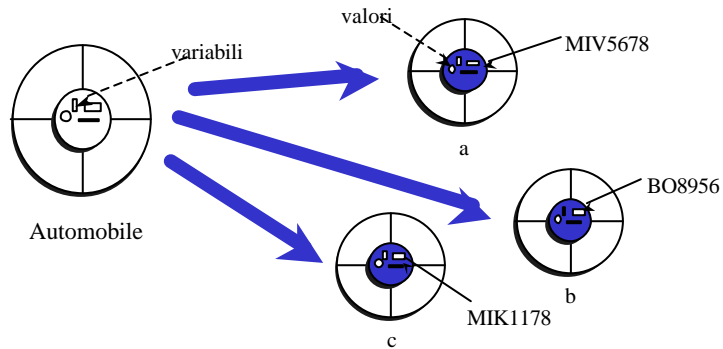
## Istanziamento ed Ereditarietà

- Una classe può essere usata per:
  - Creare direttamente oggetti della classe (istanziamento);
  - Definire nuove classi che si poggiano in qualche modo su classi esistenti. In particolare, attraverso l'ereditarietà è possibile costruire gerarchie di classi.

20

## Istanziamento

- Un oggetto è una *istanza* ("esemplare") di una classe
- Due esemplari della stessa classe sono distinguibili soltanto per il loro stato (i valori dei dati membro), mentre il comportamento (le funzioni membro) è sempre identico



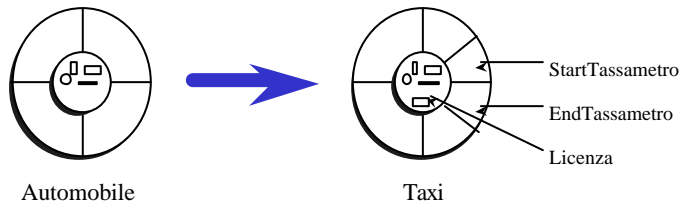
21

## L'Ereditarietà - 1

- È il meccanismo che consente di derivare una *sottoclasse* da una classe data (*superclasse*) attraverso l'*aggiunta e/o occultamento e/o la ridefinizione* dello stato e/o dei metodi della superclasse
- Esprime ciò che gli oggetti hanno in comune
- Rende possibile il riuso del codice
- Evidenzia le relazioni di generalizzazione/specializzazione

22

## L'Ereditarietà - 2



23

## L'Ereditarietà - 3

- Il codice che implementa la classe erede vede la parte privata della classe base?
- La soluzione spesso adottata introduce un altro livello di visibilità e le seguenti regole:
  - **parte pubblica** - accessibile a tutti (classe stessa, classi derivate e utenti)
  - **parte protetta** - accessibile solo alla classe stessa e a quelle derivate
  - **parte privata** - accessibile solo alla classe stessa

24



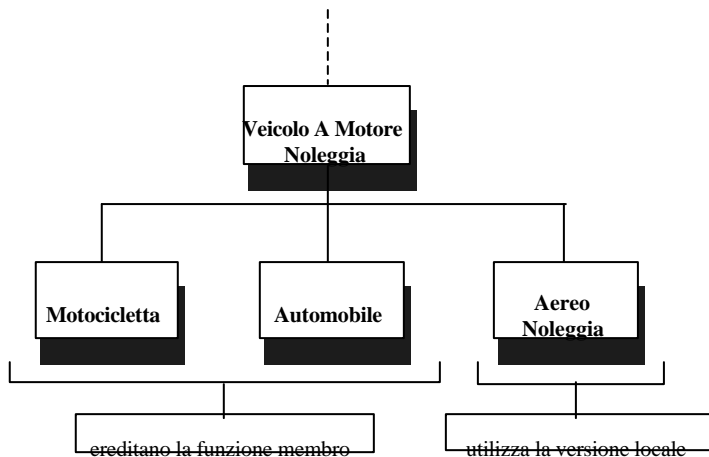
## L'Ereditarietà - 4

- Classi (*sottoclassi* o *discendenti*) possono essere definite a partire da classi preesistenti (*superclassi* o *antenate*).
- Ogni sottoclasse *eredita* stato (nella forma di dichiarazione di variabili) e metodi della superclasse.
- Può aggiungere variabili e metodi a quelli forniti dalla superclasse e/o sovrascrivere (*override*) metodi ereditati fornendone implementazioni specializzate.

25



## L'Ereditarietà - 5



26



## Le Classi Astratte

- Una classe è *astratta* quando non fornisce una completa implementazione della propria interfaccia
  - una classe astratta può avere tutte le funzioni membro non implementate
  - una classe astratta non può essere istanziata
- Le classi astratte servono come punto di partenza per la costruzione di altre classi
  - specificano l'interfaccia comune ad un insieme di classi

27



## Polimorfismo - 1

- La parola *polimorfismo* significa "avere molte forme"
- Un riferimento ad un oggetto è detto polimorfico se può essere associato a run-time ad oggetti di tipi differenti
  - il *tipo statico* è la classe dichiarata del riferimento
  - il *tipo dinamico* è la classe dell'oggetto dinamicamente riferito, sottoclasse del tipo statico
- L'invocazione di una funzione membro attraverso un riferimento polimorfico produce l'attivazione della funzione membro definita dal tipo dinamico (*binding dinamico*)

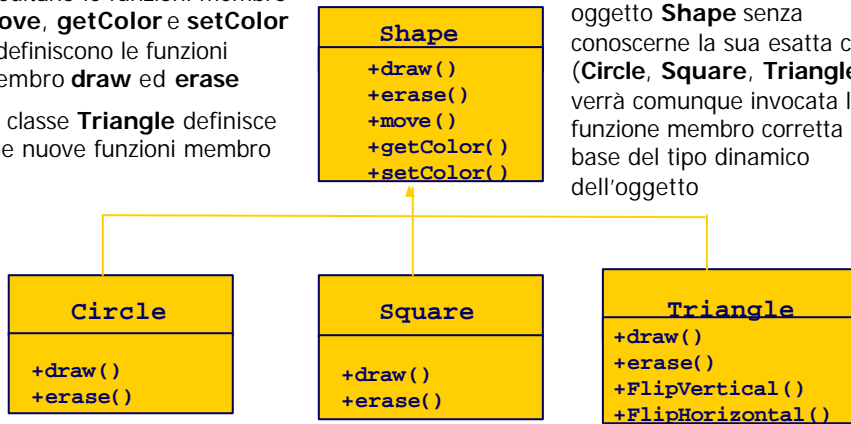
28



## Polimorfismo - 2

Ereditano le funzioni membro **move**, **getColor** e **setColor**  
Ridefiniscono le funzioni membro **draw** ed **erase**

La classe **Triangle** definisce due nuove funzioni membro



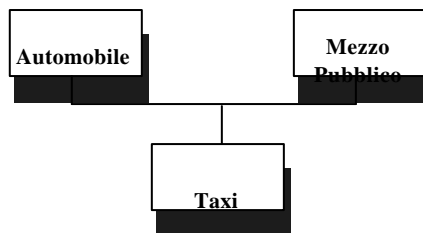
29

**Polimorfismo:** Il messaggio **draw()** può essere inviato a un oggetto **Shape** senza conoscerne la sua esatta classe (**Circle**, **Square**, **Triangle**); verrà comunque invocata la funzione membro corretta sulla base del tipo dinamico dell'oggetto




## Ereditarietà multipla

- Meccanismo che consente di derivare sottoclassi da due o più classi



- Soffre del problema degli *omonimi* (se si ereditano da due o più superclassi membri aventi lo stesso nome)
- C++ implementa l'ereditarietà multipla, a differenza di altri linguaggi orientati agli oggetti (Java, Smalltalk)

30



# Il linguaggio C++

---

Seconda parte

31



## In questa lezione

---

- Hello world
- Tipi e istruzioni principali
- Array e Puntatori
- Funzioni
- Header Files
- Regole di Scope
- Gestione della Memoria

32



# Hello World

```
// introduce un commento che termina con la linea
// Si aggiunge alla forma /* ... */
// File: hello.cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world" << endl;
    return 0;
}
```

Senza .h

I programmi che usano librerie standard dovrebbero includere `using namespace std;`

Standard output stream   Insertion operator (put to)   end-of-line

**main** deve restituire un **int**

Se **return 0** manca è generata dal compilatore, che può segnalare un warning

Lo **0** indica che tutto è andato bene



# Hello World – 2

```
// File: hello2.cpp
#include <iostream>
using namespace std;

void Hello();

int main() {
    Hello();
    return 0;
}

void Hello() {
    cout << "Hello world" << endl;
}
```

Prototipo della funzione  
Può essere omesso se si definisce **Hello** prima di **main**

Svuota il buffer di output  
In alternativa, la sequenza escape `"\n"`



# Keywords

- Salvo che per alcuni dettagli, un programma C è un programma C++ legale
  - ad esempio, le parole riservate del C sono un sottoinsieme di quelle del C++:

asm	default	for	private	struct	unsigned
auto	delete	friend	protected	switch	using
bool	do	goto	public	template	virtual
break	double	if	register	this	void
case	dynamic_cast	inline	reinterpret_cast	throw	volatile
catch	else	int	return	true	wchar_t
char	enum	long	short	try	while
class	explicit	mutable	signed	typedef	
const	extern	namespace	sizeof	typeid	
const_cast	false	new	static	typename	
continue	float	operator	static_cast	union	

35



# Tipi del linguaggio

- Alcuni dei tipi fondamentali (primitivi) del C++:

```
bool // boolean, valori possibili sono true e false
char // character, per esempio, 'a', 'z', e '9'
int // integer, per esempio, 1, 42, e 1216
float // numero floating-point in singola precisione
double // numero floating-point in doppia precisione
```

- Nelle assegnazioni e nelle operazioni aritmetiche, i tipi base possono essere mescolati liberamente; inoltre il C++ esegue automaticamente tutte le conversioni significative:

```
void some_function n() {
    double d = 2.2;
    int i = 7;
    d = d + i;
    i = d * i;
}
```

36



## Istruzioni del linguaggio

```
if ( expression ) statement
if ( expression ) statement else statement
switch ( expression ) case/default statements
while ( expression ) statement
do statement while ( expression )
for (statement expression expression )
case constant expression : statement
default : statement
break;
continue;
return expression
goto identifier ;
try { statements } catch statements
catch { exception declaration } { statements }
```

37



## Il controllo (le selezioni) Istruzioni di Selezione

Standard  
output  
stream

Standard  
input  
stream

```
bool accept(){
    char answer = 0;
    ▶cout <<<"Do you want to proceed (y or n)?\n";
    ▼cin >> answer;
    if (answer == 'y') return true;
    return false; }
    Extraction operator (get from)
```

put to

```
bool accept2() {
    char answer = 0;
    cout << "Do you want to proceed (y or n)?\n";
    cin >> answer;
    switch (answer) {
        case 'y': return true;
        case 'n': return false;
        default : cout << "I'll take that for a no.\n";
                  return false;
    }
}
```

38



## Istruzioni Cicliche

```
bool accept3() {
    char answer = 0; int tries = 1;
    while (tries < 4) {
        cout << "Do you want to proceed (y or n)?\n";
        cin >> answer;
        switch (answer) {
            case 'y': return true;
            case 'n': return false;
            default : cout << "Sorry, I don't understand that.\n";
                       tries++;
        }
    }
    cout << "I'll take that for a no.\n"; return false;
}
```

39



## Exception handling

```
try {
    //Code that may generate exceptions
}
catch(type1 id1){
    //Handles exceptions of type1
}
catch(type2 id2){
    //Handles exceptions of type2
}
catch(...){
    //Handles any other exception
}
```

40

# Gli array

Un array è una sequenza di locazioni di memoria:

```
int v[10];    // an array of 10 ints
v[3] = 1;    // assign 1 to v[3]
int x = v[3]; // read from v[3]
```

```
// Compute average of an array of data values
double average (double data[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += data[i];
    }
    return sum/n;
}
```

*i è locale al corpo del for*

41

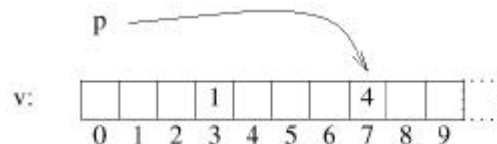
# I Puntatori

- Un puntatore memorizza l'indirizzo di una locazione di memoria

```
int *p, y; // p is a pointer to an int, y is an int
p = &v[7]; // assign the address of v[7] to p
*p = 4;    // write to v[7] through p
y = *p;    // read from v[7] through p
```

\* consente di riferire  
l'elemento puntato

\* denota un puntatore



La costante 0 (non **NULL**) indica il  
puntatore che non riferisce alcun dato<sub>2</sub>



## I Puntatori Void

- Se un puntatore ha tipo `void*` è possibile assegnargli l'indirizzo di qualsiasi dato
- Prima di potere usare il puntatore è necessario effettuare un *type casting*

```
int main() {  
    void *vp;  
    char c;  
    float f;  
    double d;  
    int i = 99;  
    // The address of ANY type can be  
    // assigned to a void pointer:  
    vp = &c;  
    vp = &f;  
    vp = &d;  
    vp = &i;  
    // Can't dereference a void pointer:  
    // *vp = 3; // Compile-time error  
    // Must cast back to int before dereferencing:  
    *((int*)vp) = 3;  
}
```

43



## Funzioni che ritornano puntatori

*se l'elemento esiste ne restituisce il puntatore*

```
int *find(int v[], int vsize, int val) { // find val in v  
    for(int i = 0; i < vsize; i++)  
        if (v[i] == val) return &v[i];  
    return &v[vsize-1];  
}
```

*se l'elemento non esiste restituisce il puntatore alla posizione che segue l'ultimo elemento*

```
int count [] = {2, 3, 1, 9, 7, 3, 3, 0, 2};  
int count_size = 9;  
void f() {  
    int *p = find(count, count_size, 7) ; // find 7 in count  
    int *q = find(count, count_size, 0) ; // find 0 in count  
    *q = 4;  
    ...  
}
```

44



# Puntatori e Array - 1

- Il nome di un array è un puntatore:

```
int *find(int *v, int vsize, int val) {  
    for(int *p = v; p < v + vsize; p++)  
        if (*p == val) return p;  
    return v + vsize;}  
}
```

- Quando si passa un array a una funzione, in realtà si passa un puntatore al primo elemento dell'array (*passaggio per riferimento*):

```
int count [] = {2, 3, 1, 9, 7, 3, 3, 0, 2};  
int count_size = 9;  
void f() {  
    int *p = find(count, count_size, 7) ; // find 7 in count  
    int *q = find(count, count_size, 0) ; // find 0 in count  
    *q = 4;  
    ...  
}
```

45



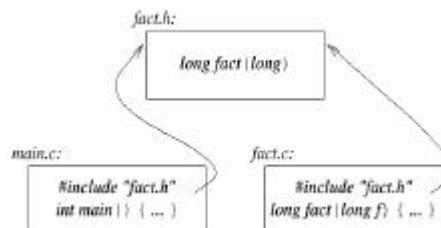
# Compilazione Separata

- Un programma C++ consiste di più file sorgente che sono individualmente compilati in file oggetto
- Questi sono poi collegati insieme per produrre la forma eseguibile del programma

```
// File: fact.h:  
long fact(long);
```

```
// File: main.cpp:  
#include "fact.h"  
#include <iostream>  
using namespace std;  
int main(){  
    cout << "factorial(7) is "  
        << fact(7) << '\n';  
    return 0;  
}
```

```
// File: fact.cpp:  
#include "fact.h"  
long fact(long f) {  
    if (f <= 1) return 1;  
    return f * fact(f-1);  
}
```





## Perché separare un programma

- La struttura di un programma medio-grande è più chiara se il programma è suddiviso in più file sorgente
- Più programmatori possono lavorare allo stesso tempo su file diversi
- Non occorre ricompilare l'intero programma ogni volta che si effettua una piccola modifica
- Si possono usare librerie senza doverne includere fisicamente il codice sorgente nel programma (in effetti, spesso il sorgente non è disponibile)

47



## Come separare un programma

- Un header file va incluso non più di una volta in ogni translation unit:
  - Le definizioni multiple sono errori
  - Esiste il rischio di inclusioni circolari (A include B che a sua volta include A)
- Il problema si risolve con l'uso delle direttive `#ifndef`, `#define`, `#endif`, secondo lo schema seguente:

```
// File: Name.h
#ifndef _Name_H
#define _Name_h
...
#endif
```

*Alla prima inclusione, il simbolo **\_Name\_H** è definito e il file è incluso*  
*Alle successive inclusioni, il simbolo è già definito e questo fa "saltare" il preprocessore alla linea finale del file **#endif***

48



## Scope - 1

---

- Le *regole di "scope"* dicono quando un nome è valido e può essere usato e qual è la dichiarazione/definizione associata ad ogni suo uso
- In generale, lo *scope* di un nome si estende dal punto dove esso è dichiarato/definito alla parentesi chiusa appaiata con la più vicina parentesi aperta che precede la dichiarazione/definizione
- Un nome può essere usato solo nel suo scope e in ogni scope innestato { { } }

49



## Scope - 2

---

- Il compilatore associa a ciascun uso di un nome la dichiarazione ad esso "più vicina"
- Ridichiarare in uno scope innestato un nome rende inutilizzabile quello più esterno (*shadowing*)
- Per forzare l'accesso a un nome globale si può usare l'operatore **::** (**N.B. solo ad un nome globale !**)

50

## Scope – 3

```
// How variables are scoped
int main() {
    int a;           // a visible here
    {               // a still visible here
        //...
        int b;      // b visible here
        //...
        {           // a & b still visible here
            //...
            int c;  // a, b & c visible here
            //...
        }          // <-- c destroyed here
                  // c not available here
                  // a & b still visible here
        //...
    }             // <-- b destroyed here
                  // c & b not available here
                  // a still visible here
    //...
}                // <-- a destroyed here
```

51

## Scope - 4

```
// File: scope.cpp
#include <iostream>
using namespace std;
int main() {
    int first = 10;
    if (first > 0) {
        int first = 100;
        first = first/10;
        cout << "The value of first is " << first << endl;
    }
    cout << "The value of first is " << first << endl;
}
return 0;
}
```

Output

```
The value of first is 100
The value of first is 10
```

52



## Definizioni al volo

```
int main() {
    ...
    {
        // Begin a new scope
        int q = 0; // C requires definitions here
        ...
        for(int i = 0; i < 100; i++) { // Define at point of use
            q++; // q comes from a larger scope
            int p = 12; // Definition at the end of the scope
        }
        int p = 1; // A different p
    } // End scope containing q & outer p
    cout << "Type characters:" << endl;
    while (char c = cin.get() != 'q') {
        //...
        if(char x = c == 'a' || c == 'b')
            ...
    }
    switch(int i = cin.get()) {
        ...
    }
}
```

53



## I namespace

- Utilizzo dei namespace
- Esempio
- La dichiarazione dei namespace
- L'importazione dei nomi

54



## I namespace - utilizzo

- Pur potendo innestare i nomi all'interno delle classi, comunque i nomi di variabili globali e classi restano visibili in un unico namespace globale. Questo spesso porta alla creazione di nomi lunghi o poco significativi.
- Il meccanismo dei namespace permette di suddividere un unico programma in diversi spazi di nomi

```
// MyLib.cpp
namespace MyLib { ← Dichiarazione del namespace
    class MyClass {...}
    ...
    // Declarations
}
int main() {
    MyLib::MyClass *c; ← Riferimento a entità di un namespace
}
```

55



## I namespace - esempio

```
namespace Math {
    int factorial(int n);
    int fibonacci(int n);
}
int Math::factorial(int n) {
    int product = 1;
    for(int k = 1; k <= n; k++){
        product *= k;
    }
    return product;
}
int Math::fibonacci(int n) {
    int f = 1, s = 1;
    for(int k = 1; k <= n; k++){
        int fib = f + s;
        f = s; s = fib;
    }
    return f;
}
```

```
#include <iostream>
int main() {
    int n;
    std::cout << "enter n ";
    std::cin >> n;
    std::cout << n << "! = "
        << Math::factorial(n)
        << std::endl;
    std::cout << "F_(" << n << ") = "
        << Math::fibonacci(n)
        << std::endl;
    return 0;
}
```

56



## I namespace - dichiarazione

- Un namespace può essere costruito incrementalmente in file diversi...

```
// C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
    extern int x;
    void f();
    // ...
}
#endif //HEADER1_H
```

```
// C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { //NOT a redefinition
    extern int y;
    void g();
    // ...
}
#endif //HEADER2_H
```

57



## I namespace – importazione dei nomi

- La direttiva **using** consente di importare nomi in uno scope (es. `using Int::sign`);
- La forma **using namespace** consente di importare i nomi dell'intero namespace;

```
namespace Int{
    enum sign {positive, negative};
    class Integer {
        sign s;
    public:
        ...
        void setSign(sign sgn){s=sgn;};
    };
    ...
namespace Math{
    using namespace Int;
    Integer a,b;
    Integer divide(Integer, Integer);}
```

```
#include "NamespaceInt.h"

void arithmetic()
{
    using namespace Int;
    Integer x;
    x.setSign(positive);
}
```

58



## La memoria dati di un programma C++

---

- La memoria dati di un programma si articola nelle seguenti aree:
  - **Memoria *statica***
    - il dato esiste per tutta la durata del programma
    - esempi: variabili globali e variabili dichiarate **static** nelle funzioni
  - **Memoria *automatica***
    - è un'area di memoria gestita a *stack*
    - serve per gli argomenti e le variabili locali delle funzioni
    - la memoria automatica di una funzione (*frame*, *record di attivazione*) è automaticamente creata alla chiamata e distrutta al ritorno

59



## La memoria dati di un programma C++

---

- **Memoria *dinamica (heap)***
  - contiene i dati allocati nel corso dell'esecuzione per mezzo dell'operatore **new** e ancora non distrutti per mezzo dell'operatore **delete**

60

# Variabili Globali

```
// File: Global.cpp
// Demonstration of global variables
#include <iostream>
using namespace std;
int globe;
void func();
int main() {
    globe = 12;
    cout << globe << endl;
    func();           // Modifies globe
    cout << globe << endl;
}
```

*Il tempo di vita di una  
variabile globale è l'intero  
tempo di esecuzione del  
programma  
Cosa succede se dichiaro  
globe all'interno del main?*

```
// File: Global2.cpp
// Accessing external global variables
extern int globe;
// The linker resolves the reference
void func() {
    globe = 47;
}
```

*Una variabile globale può  
essere riferita in un file  
diverso da quello di  
definizione per mezzo di una  
dichiarazione **extern***

61

# Variabili Globali Static

- Se una variabile globale è definita **static** essa è accessibile solo nel file in cui è definita
- Se si tenta di riferire la variabile in un altro file, il linker segnala l'errore

```
// File: Static.cpp
// File scope demonstration
// File scope means only available in this file
static int fs;
int main() {
    fs = 1;
}
```

Errore in fase di linking

```
// File: Static2.cpp
// Trying to reference fs
extern int fs;
void func() {
    fs = 100;
}
```

62



## Variabili Automatiche

- Il *tempo di vita* di una variabile locale automatica coincide con il tempo che intercorre tra la chiamata della funzione in cui è dichiarata e il ritorno dalla funzione stessa
- Il tentativo di usare una variabile automatica dopo il ritorno dalla funzione è un errore, in quanto la variabile è stata deallocata

```
// WARNING - The program contains an error
char *readALine () {
    char buffer[1000]; // declare a buffer for the line
    ...
    return buffer;    // return text of line
}
```

- L'ingombro di una variabile automatica deve essere noto a tempo di compilazione, quando la struttura e l'ingombro del record di attivazione della funzione sono fissati

63



## Variabili Locali Static

- Se una variabile locale è dichiarata **static** essa:
  - conserva il suo valore tra le successive chiamate della funzione
  - è inizializzata solo la prima volta che la funzione è chiamata

```
// Using a static variable in a function
#include <iostream>
using namespace std;
void f() {
    static int i = 0;
    cout << "i = " << ++i << endl;
}
int main() {
    for(int x = 0; x < 10; x++) f();
}
```

*Ogni volta che f è chiamata scrive un diverso valore*

*La variabile i non è visibile al di fuori del suo scope  
Questa è la differenza rispetto a una variabile globale*

64



## Variabili Dinamiche

```
char * buffer;  
int newSize;  
...  
// newSize is given some value  
buffer = new char[newSize]; // create an array of the given size  
...  
delete[] buffer; // delete buffer when no longer being used
```

- Tipici errori quando si usa l'allocazione dinamica sono:
  - dimenticare di allocare un dato nello heap e usare il puntatore come se lo stesse riferendo (produce un effetto imprevedibile)
  - dimenticare di restituire la memoria inutilizzata allo heap manager (*memory leak*)
  - tentare di usare dati dinamici dopo che sono stati restituiti allo heap manager
  - invocare **delete** più di una volta sullo stesso dato

```
AnObject *a = new AnObject();  
...  
a = new AnObject(); // leak, old reference is now lost
```

65



## Le regole

- Classi, variabili globali e funzioni possono essere dichiarate quante volte si vuole in una translation unit, a condizione che le dichiarazioni siano consistenti:

```
class Matrix; // declare (but don't define) a class  
...  
class Matrix; // declare (but don't define) the class again
```

- Le definizioni invece non possono essere ripetute:

```
class Matrix {... }; // define class  
class Matrix {... }; //error: duplicate definition
```

- Una definizione di classe può apparire al più una volta in una translation unit;
- Una definizione di funzione o di variabile globale o static può apparire al più una volta in ogni programma eseguibile.

66