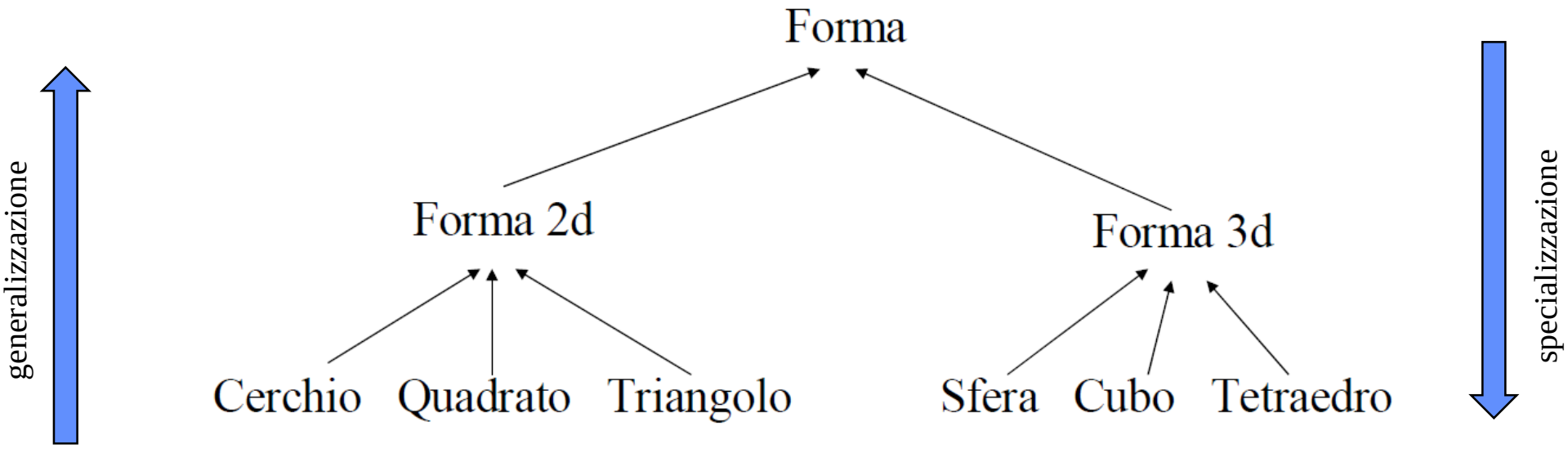
Ereditarietà

- L'ereditarietà consente di definire nuove classi per specializzazione di classi esistenti
- L'ereditarietà è utile per indurre una organizzazione gerarchica nel software e per la creazione di software riutilizzabile
- Quando si crea una nuova classe si può fare in modo che questa erediti (acquisisca) i dati membro e le funzioni membro da una classe già definita precedentemente, la classe derivata può aggiungere caratteristiche nuove o modificare caratteristiche pre-esistenti
- Le classi derivate (sotto-classi) realizzano comportamenti specializzati rispetto alle classi base
- La classe precedente prende il nome di *classe base*
- La classe che eredita prende il nome di *classe derivata*

Ereditarietà

- E’ possibile continuare il procedimento di ereditarietà creando una classe che eredita a sua volta da una classe derivata
- Questo procedimento crea una gerarchia
- Siamo interessati all’ereditarietà *singola*: quando una classe derivata eredita da una sola classe base



Ereditarietà

- la dichiarazione di una classe derivata deve includere il nome della classe base da cui deriva ed, eventualmente, uno specificatore d'accesso indicante il tipo di ereditarietà (*public*, *private* o *protected*) secondo la seguente sintassi:

```
class ClasseDerivata : specific_accesso_opz ClasseBase {
    membri;
};
```

- Se si omette lo specificatore di accesso viene utilizzato per default il tipo di eredità *private*
- La più utilizzata è l'eredità di tipo *public*
- Un oggetto di una classe derivata può essere assegnato ad un oggetto della classe padre (upcasting) con perdita dei dati membro definiti nella classe derivata. L'assegnamento contrario non è consentito.

Ereditarietà

- Siamo maggiormente interessati ad una eredità di tipo *public:* **significa che una classe derivata ha accesso agli elementi pubblici e protetti della sua classe base**
- **I membri public** della classe base diventano membri **public** della classe derivata, i membri **private** della classe base non sono accessibili direttamente nella classe derivata, sono accessibili indirettamente chiamando membri public della classe base, i membri **protected** della classe base sono accessibili alla classe derivata ma *non* all'esterno

```
class BaseClass{
  //dichiarazione
};

class DerivedClass: public BaseClass {
  //dichiarazione
};
```

Ereditarietà

- Ereditarietà private:

```
class ClasseDerivata : private ClasseBase {
public:
    // sezione pubblica
protected:
    // sezione protetta
private:
    // sezione privata
};
```

- i membri pubblici e protetti della classe base diventano membri privati della classe derivata

Ereditarietà

- Ereditarietà protected: i membri pubblici e protetti della classe base diventano membri protetti della classe derivata ed i membri privati della classe base diventano inaccessibili

```
class ClasseDerivata : protected ClasseBase {
public:
   // sezione pubblica
protected:
   // sezione protetta
private:
   // sezione privata
};
```

- In nessun tipo di ereditarietà una classe derivata può accedere a variabili e funzioni *private* della sua classe base

Ereditarietà

- Schema riassuntivo

Tipo di ereditarietà	Accesso a membro classe base	Accesso a membro classe derivata
`public`	`public`	`public`
	`protected`	`protected`
	`private`	inaccessibile
`protected`	`public`	`protected`
	`protected`	`protected`
	`private`	inaccessibile
`private`	`public`	`private`
	`protected`	`private`
	`private`	inaccessibile

Ereditarietà

- I costruttori non si ereditano e vanno ridefiniti nelle classi derivate
- Si può chiamare esplicitamente il costruttore della classe base per inizializzare i dati membri *condivisi* con la classe base
- il costruttore della classe derivata si deve occupare di inizializzare i dati membri aggiuntivi, quelli cioè che sono introdotti in più rispetto ai dati membro della classe base

```
NomeClassDeriv(T prm_bas,T prm_drv):NomeClassBase(prm_bas){
  //init con prm_drv
}
```

- In mancanza di una chiamata esplicita al costruttore della classe base, nella inizializzazione, il compilatore inserisce automaticamente la chiamata al costruttore di default della classe base

Ereditarietà

```
#include <iostream>
using namespace std;
// Base class
class Shape {
 public:
    void setWidth(int w) { width = w; }
    void setHeight(int h) { height = h; }
    //private: errore
    //protected: ok
      int width; int height;
};
// Derived class
class Rectangle : public Shape {
    public: int getArea() { return (width * height); }
};
int main(void) {
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7);
    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;
    return 0;
}
```

Output:
Total area: 35

Ereditarietà

```
#include <iostream>
using namespace std;
class Base {
  public: int i1;  protected: int i2;  private: int i3; };
class D1: private Base {
  void f() { i1 = 0; i2 = 0; // i3 = 0; // errore } };
class D2 : protected Base { void g() { i1 = 0; i2 = 0; //i3 = 0; // errore } };
class D3 : public Base { void g() { i1 = 0; i2 = 0; //i3 = 0; // errore } };

int main(void) {
  Base b; b.i1 = 1;
  //b.i2 = 1; //errore
  //b.i3 = 1; //errore
  D1 d1;
  //d1.i1 = 1; //errore
  //d1.i2 = 1; //errore
  //d1.i3 = 1; //errore
  D2 d2;
  //d2.i1 = 1; //errore
  //d2.i2 = 1; //errore
  //d2.i3 = 1; //errore
  D3 d3;
  d3.i1 = 1;
  //d3.i2 = 1; //errore
  //d3.i3 = 1; //errore return 0; }
```

Ereditarietà

- i distruttori non si ereditano
- i distruttori si scrivono esplicitamente normalmente solo quando un corrispondente costruttore ha assegnato spazio in memoria con allocazione dinamica
- Se non è presente un distruttore scritto esplicitamente dal programmatore viene generato un distruttore di default
- La chiamata (automatica) di un distruttore di una classe derivata produce automaticamente la chiamata di tutti i distruttori delle sue superclassi. I distruttori sono chiamati secondo l'ordine che si ottiene risalendo nella gerarchia delle classi.
- Pertanto il distruttore di una classe derivata deve occuparsi esclusivamente delle operazioni di pulizia relative ai nuovi dati membro introdotti dalla classe derivata

Polimorfismo

- il polimorfismo consente di accedere alle funzioni membro di un oggetto di una classe derivata anche tramite un riferimento (o puntatore) ad un oggetto di una classe base.
- si ottiene con l'ereditarietà e le funzioni virtuali
- Una chiamata a una funzione membro causa l'esecuzione di funzioni diverse in base al tipo di oggetto che chiama la funzione membro
- classe base Forma
- classe derivata Punto
- classe derivata Cerchio
- classe derivata Cilindro
- Si vuole poter invocare una unica funzione **disegna** per tutte le classi derivate tramite un puntatore alla classe base Forma

(*ptr).disegna()

Polimorfismo

```
#include <iostream>
using namespace std;
class Shape {
  protected:
    int width, height;
  public: Shape(int a = 0, int b = 0) {
    width = a; height = b; }
  int area() {
    cout << endl << "Parent class area :";
    return 0; }
};

class Rectangle : public Shape {
   public: Rectangle(int a = 0, int b = 0) :Shape(a, b) { }
   int area() {
     cout << endl << "Rectangle class area :";
     return (width * height); }
};

class Triangle : public Shape {
   public: Triangle(int a = 0, int b = 0) :Shape(a, b) { }
   int area() {
     cout << endl << "Triangle class area :";
     return (width * height / 2); }
};
```

Polimorfismo

```
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10, 7);
   Triangle tri(10, 5);
   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   cout << shape->area();
   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   cout << shape->area();
   return 0;
}
```

Output:

Parent class area : 0

Parent class area : 0

Viene chiamata la funzione membro area() della classe base (**early binding**).

Polimorfismo

```
class Shape {
  protected:
   int width, height;
  public:
   Shape(int a = 0, int b = 0) {
     width = a;
     height = b; }
  virtual int area() {
   cout << "Parent class area :" << endl; return 0; }
};
```

Output
Rectangle class area :70
Triangle class area :25

Definendo virtual la funzione area() nella classe base il compilatore a tempo di esecuzione guarda al contenuto del puntatore shape, invece che al suo tipo, e quindi viene chiamata la funzione membro della classe (**late binding**)

Polimorfismo

- Ricapitolando: l'effetto delle funzioni virtuali si nota quando si utilizzano i puntatori alla classe base per riferirsi agli oggetti delle classi derivate
- In pratica accedendo ad un oggetto di una classe derivata tramite un puntatore di tipo classe base nel caso ordinario si accede ai membri della classe base nel caso virtual si accede ai membri della classe derivata

Polimorfismo

- Talvolta non è possibile definire un comportamento significativo per una funzione in una classe base
- Esempio: la classe base Forma può avere un metodo Stampa() ma questo è definibile con precisione solo dalle classi derivate Cerchio e Cilindro che specificano i propri attributi

- E' possibile definire delle *classi astratte*
- Una classe astratta è una classe che contiene almeno una *funzione virtuale pura*
- Sintassi:

virtual retType FuncName(argType)=0;

- Non è possibile istanziare alcun oggetto di una classe astratta, ma possono essere definiti dei puntatori a tale classe

Polimorfismo

- il livello più alto di una gerarchia di classi generalmente è realizzato come classe astratta
- Si specifica così l'interfaccia necessaria per tutte le classi derivate

```
// abstact class
class Shape {
  protected:
  int width, height;
  public: Shape(int a = 0, int b = 0) {
   width = a; height = b; }
  // pure virtual function
  virtual int area() = 0;
};
```

Polimorfismo

- Cosa accade quando si chiama il distruttore tramite puntatore ad una classe base?
- Se si allocano dinamicamente oggetti con il new e si deallocano tramite l'operatore delete... se si sta usando un puntatore alla classe base per riferirsi ad un oggetto di una classe derivata...allora viene chiamato il distruttore della classe base e non quello della classe derivata
- per chiamare correttamente il distruttore della classe derivata si deve dichiarare il distruttore come **virtual**
- Per classi con funzioni virtuali si consiglia di creare sempre distruttori virtuali anche se non strettamente necessari. In questo modo le classi derivate invocheranno i distruttori in modo appropriato
- ESEMPIO: si veda il codice della classe astratta list delle prossime lezioni
- **NOTA:** i costruttori non possono essere virtuali

Template

- In C++, tramite i *template,* è possibile definire delle funzioni o classi dette *generiche* ovvero che abbiano come *parametro* il *tipo* di dato
- il template è un meccanismo che consente di creare uno schema (o modello) generico di funzione o di classe, che non fa esplicito riferimento a nessun particolare tipo di dati
- Utile quando si devono scrivere più funzioni o classi che si comportano allo stesso modo ma lavorano su dati di tipo diverso
- il template consente di progettare solo uno schema generale per realizzare una particolare operazione

Template

- Una funzione (classe) generica si chiama funzione (classe) template
- il codice di definizione e di implementazione dei metodi di una classe template va scritto nei file header!
- Codice generato al momento della compilazione (e conseguente rilevazione di errori al tempo della compilazione)
- Un template è un codice parametrico

Template

- il template di funzione (*function template*) è uno schema di funzione generica, che non fa esplicito riferimento a nessun particolare tipo di dati input
- consente di progettare un'unica funzione per realizzare una particolare operazione/algoritmo
- Esistono algoritmi logicamente identici ma che possono operare su tipi diversi (esempio: ordinamento, scambio, min…)
- Con le funzioni generiche è possibile definire la natura dell'algoritmo indipendentemente dai dati
- Con un'unica definizione si hanno a disposizione versioni diverse della funzione in grado di operare su differenti tipi di dati

Template

- La forma tipica della definizione di una funzione generica è:

```
template <typename tipo>
  tipo-restituito nome-funzione (parametri) {
    // corpo della funzione
}
```

- ‘’tipo’’ non è altro che un simbolo, un “segnaposto” che il compilatore sostituirà col vero tipo dei dati durante la creazione di una versione specifica della funzione
- Nella definizione di un template si deve specificare la lista dei parametri, tra <>
- Una funzione template può essere parametrica rispetto al tipo del valore di ritorno, ai tipi dei suoi parametri, ai tipi delle sue variabili locali

Template

- Esempio di due funzioni che eseguono la somma degli elementi di un array:

```
int sumVector(int *array, int n) {
 int sum(0);
 for (int i = 0; i < n; ++i)
  sum += array[i];
 return (sum);
}
```

```
double sumVector(double *array, int n) {
 double sum(0);
 for (int i = 0; i < n; ++i)
  sum += array[i];
 return (sum);
}
```

- Vediamo una soluzione generica con template
- Un parametro può essere un tipo (parametro di tipo, anche classi e strutture dati) preceduto dalle parole ‘’class’’ o ‘’typename’’ oppure un’espressione costante (parametro di valore) preceduta dal tipo del risultato dell’espressione

Template

- Soluzione generica con funzione template

```
#include<iostream>
using namespace std;
template <typename T>
   T sumVector(T *array, int n) {
      T sum(0);
      for (int i = 0; i < n; ++i)
         sum += array[i];
      return (sum); }

#include "esempio_1_32.h"
int main() {
   double darray[3] = { 1.0, 2.2, 3.2 };
   double dsum = sumVector(darray, 3);
   cout << "dsum=" << dsum << endl;
   int iarray[5] = { 1,5,6,2,8 };
   int isum = sumVector(iarray, 5);
   cout << "isum=" << isum << endl;
return 0; }
```

Output:
dsum=6.4
isum=22

Template

- Esempio:

```
#include<iostream>
using namespace std;
template <typename X>
   void scambia(X & a, X & b) {
     X temp;
     temp = a;
     a = b;
     b = temp;
   }
```

```
#include "esempio_1_33.h"
int main() {
  int i = 10, j = 20;
  char a = 'A', b = 'B';
  scambia(i, j);
  cout << i << " " << j << endl;
  scambia(a, b);
  cout << a << " " << b << endl;
}
```

Output
20 10
B A

Template

- E’ possibile definire più di un tipo di dati generico e indicare esplicitamente il valore di un parametro del template (tra <>)

```
#include<iostream>
using namespace std;
template <typename T1, typename T2>
void myfunc(T1 x, T2 y) {
  cout << x << " " << y << endl;
}
```

```
#include “esempio_1_34.h”
void main() {
  int a = 7;
  char b = 'F';
  myfunc(a, "prova");
  myfunc<float,char>(10.2, b);
}
```

Output
7 prova
10.2 F

Template

- Una classe generica è in grado di contenere algoritmi per i quali il tipo dei dati da manipolare è specificato al momento della creazione di ogni elemento della classe

```
template <typename tipo> class nome-classe {
  // definizione della classe
}
```

- ''tipo'' è un segnaposto a cui verrà sostituito il nome del tipo al momento della creazione di un'istanza della classe
- E' possibile dichiarare una classe generica che utilizza più tipi generici mediante un elenco separato da virgole: <typename T1, typenameT2>
- Per le classi si deve sempre esplicitare il tipo nella dichiarazione
- Non ci sono meccanismi di deduzione automatica

nome-classe<tipo> oggetto;

Template

- Esempio di class template

```
#ifndef ESEMPIO_1_35
#define ESEMPIO_1_35
#include<iostream>
using namespace std;

template <class T> class myclass {
   T value1, value2;
   public: myclass(T first, T second) { value1 = first; value2 = second; }
   T getmax() {
        T retval;
        retval = value1>value2 ? value1 : value2;
        return retval; } };
#endif

#include "esempio_1_35.h"
int main() {
    myclass<int> myobject(100, 75);
    cout << myobject.getmax();
    return 0;
}
```

Output:
100

Template

- soluzione equivalente

```
#ifndef ESEMPIO_1_35
#define ESEMPIO_1_35
#include<iostream>
using namespace std;
template <class T> class myclass {
   T value1, value2;
   public: myclass(T first, T second) { value1 = first; value2 = second; }
   T getmax();
};

template <class T> T myclass<T>::getmax() {
    T retval;
    retval = value1>value2 ? value1 : value2;
    return retval; }
#endif

#include "esempio_1_35.h"
int main() {
    myclass<int> myobject(100, 75);
    cout << myobject.getmax();
return 0;
}
```

Output:
100

Template

- altro esempio

```
#ifndef ESEMPIO_1_36
#define ESEMPIO_1_36
#include<iostream>
using namespace std;
template<class T> class Item {
    T Data;
    public:
     void SetData(T nValue) {
        Data = nValue; }
     T GetData() const { return Data; }
     void PrintData() { cout << Data; } };
#endif

#include "esempio_1_36.h"
int main() {
   Item<int> item1;
   item1.SetData(120);
   item1.PrintData();
   Item<float> item2;
   item2.SetData(0.5);
   float n = item2.GetData();
   cout << endl << n; return 0;
}
```

Output:
120
0.5

Template

- altro esempio

```
#ifndef ESEMPIO_1_37
#define ESEMPIO_1_37
#include<iostream>
using namespace std;
template<typename T> class Item {
  private:
     T element;
  public:
     Item(T n) { element = n; }
     void Add(T n) {
               element = element + n;
              }
     T getItem() { return element; } };
#endif

#include "esempio_1_37.h"
#include <string> int main() {
         Item<int> item1(3); Item<string> item2("pippo");
         item1.Add(5);
         item2.Add("topolino");
         cout << item1.getItem() << endl;
         cout << item2.getItem() << endl;
         return 0;
}
```