

Sommario:

Introduction:	2
Classes and objects	6
Coding style guidelines for classes	19
Inheritance	20
Generic programming & templates	39
STL.....	53
Exceptions.....	70
Resource Management.....	79
Design patterns.....	85
Adapter.....	87
Observer.....	91
Factory	99

Introduction:

Perchè uno sviluppo verso l'OO?

Una struttura migliorata del software è + semplice per :

- Capire
- Mantenere
- Valorizzare
- Il riutilizzo di component software e schemi(modules)
 - Potenziare un componente già esistente
 - Creare nuovi componenti più generalizzabili.

Programmazione strutturata vs approccio OO

- L'analisi e il design della programmazione strutturata si concentra principalmente su una struttura procedurale e, in quanto tale la funzionalità relativa agli stessi dati è stata spesso diffusa attraverso il sistema di software.
- L'approccio object oriented si concentra su lo strutturare il sistema intorno ai dati incapsulandoli all'interno di oggetti in modo tale che tutte le funzionalità in relazione con tali dati sono, almeno in teoria, contenuti all'interno degli oggetti stessi.

Concetti chiave dell'OO

- L'idea principale: incapsulare i dati all'interno di oggetti

1. Data Abstraction: poichè ora i dati sono visti non più in termini di puri "data values" ma diversamente in termini delle operazioni che possiamo effettuare su di essi.

- ADT's(Abstract Data Type), information hiding: poichè i dettagli dell'implementazione dei data format e il modo in cui tali operazioni sono effettivamente implementate sono nascoste da altro codice che utilizza un oggetto.

- C++ classes: più ovvia facilità usata per implementare delle "data abstraction"

2. Ereditarietà (generalizzazione)

- La gerarchia delle classi e l'ereditarietà: alcune classi di oggetti sono generalizzazioni di altre classi più specifiche di oggetti e così ha senso porre funzionalità generali in ciò che noi chiamiamo classe base e permettere funzionalità più specifiche da poter essere messe in classi derivate dalla classe base. Si dice che le classi derivate ereditano(o estendono) le funzionalità della classe base.

3. Il polimorfismo: possiamo usare differenti tipi di oggetti nello stesso modo è ciò che il software system, in questo caso il C++, elaborerà per noi quale funzionalità dovrebbe in verità essere invocata in realtà a seconda dell'effettivo tipo di oggetto che è coinvolto per un particolare utilizzo.

- Le stesse operazioni su differenti classi/oggetti:

Operatori aritmetici che possono essere applicati sia con interi che con float, sebbene la reale implementazione dell'aritmetica è abbastanza diversa.

- Funzioni virtuali

- L'operatore overloading

In C++ il polimorfismo è sostenuto dall'uso di oggetti che sono chiamate funzioni virtuali come pure per mezzo di un sovraccarico ulteriore di operatori standard.

Oggetti

• Un oggetto è come una variabile comune: è come una reale istanza di una entità software che detiene reali informazioni, ad esempio riguardo a qualcosa del mondo reale.

- Contiene informazioni
- È la realizzazione Software di alcune “cose” del mondo.
- Si possono applicare delle operazioni su di essi.

Classi

• Una classe è (come) un tipo:

• Una astrazione di una serie di oggetti che si comportano in modo identico: tiene insieme l'implementazione dei dati e le operazioni.

• Definisce l'implementazione interna, le operazioni.

• Si possono creare (istanziare) oggetti da una classe, esattamente come per un built-in type (int o float), un tipo definito (di) dall'utente usando la facilitazione della classe può allora essere istanziato in oggetti o variabili.

• Le classi aiutano a organizzare il codice e a ragionare sui programmi.

• Una classe è la rappresentazione di un'idea, un concetto, in forma di codice.

Un oggetto di una classe rappresenta un particolare esempio dell'idea nel codice.

• Senza le classi, un lettore del codice dovrebbe indovinare/supporre le relazioni tra i data-items e le funzioni – le classi rendono tali relazioni esplicite e comprensibile dai compilatori.

Con le classi, gran parte della struttura di alto livello del tuo programma si riflette nel codice, non semplicemente nei commenti.

I metodi dell'OO.

• I metodi dell'OO sono una serie di tecniche per analizzare, decomporre e modularizzare l'architettura dei software systems.

• In generale, i sistemi si evolvono e le funzionalità cambiano, ma gli oggetti (e le classi degli oggetti) tendono a rimanere stabili nel tempo.

• Il paradigma dell'OO influenza l'intero processo di sviluppo del software.

Procede attraverso una serie di fasi, sebbene i limiti siano spesso sfocati/confusi.

OO Software Development

• **OO Analysis:**

• La funzionalità, classi e le loro relazioni richieste nell'identificazione:

spesso usa (l' **Analysis**) strumenti (tools) presi dall'UML (Unified Modelling Language) come ad esempio i diagrammi di classe e l'uso di “cases”. (such as class diagrams and use cases)

• **OO Design:**

• Specificare la gerarchia tra le classi, le interfacce delle classi e il loro comportamento.

Gli UML tools come diagrammi di classe, di iterazione e di stato possono essere usati nel Design di tipo OO.

• **OO Programming:**

• Implementare un OO design in un linguaggio di programmazione OO: in questa fase il codice è effettivamente scritto, testato e integrato con altro codice.

OOA

• Object-oriented analysis (OOA) applica le tecniche di object-modeling per analizzare le esigenze funzionali del sistema.

• L'OOA guarda al centro del problema, con il fine di produrre un modello concettuale dell'informazione che c'è nella zona che è stata analizzata e ne capisce le necessità.

• I modelli dell'Analysis non considerano nessun impedimento di implementazione che potrebbe esistere.

OOD

- Object-oriented design (OOD) elabora i modelli di analisi per produrre una descrizione specifica dell'implementazione.
- Durante lo sviluppo dell'OOD si creano astrazioni e meccanismi necessari per incontrare le richieste di comportamento del sistema(systems') fissati durante l'OOA.
- I concetti nei modelli dell'analisi sono mappati nelle classi di implementazione. Il risultato è un modello della(del dominio della) soluzione, una dettagliata descrizione di come il system debba essere costruito..
- OOD è relativamente indipendente dal linguaggio di programmazione utilizzato.
- OOA si concentra su ciò che il sistema esegue, OOD invece su come il sistema lo esegue

Gli obiettivi del Design

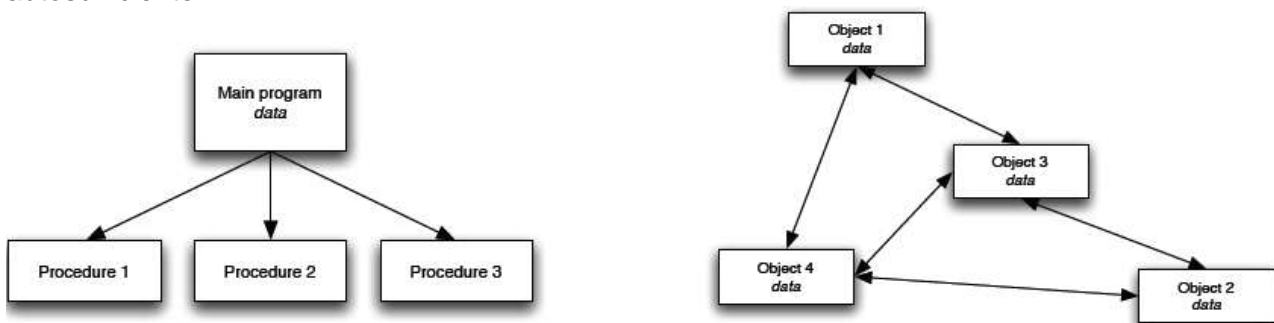
- Creare software che sia semplice da cambiare ed estendere, ovvero flessibile.
- Decoprire il sistema in moduli, determinando le relazioni tra essi, per esempio identificando le dipendenze e le forme di comunicazione.
 - Le classi, il loro uso, l'ereditarietà, la composizione.
 - Raggiungere un Software altamente coeso, poco accoppiato.

Object Oriented Programming

- Object-oriented programming (OOP) è concentrato principalmente sul linguaggio di programmazione e nelle questioni di implementazione del software.
- OOP è un paradigma di programmazione che usa "oggetti" – strutture dati che consistono di dati e metodi insieme alle loro interazioni – per progettare applicazioni e programmi per il computer.
- Un programma object-oriented può esser visto come un insieme di oggetti che interagiscono, in opposizione al modello strutturale della programmazione, in cui un programma è visto come una lista di compiti(task/ subroutines) da eseguire.

Procedural Programming vs. OOP

- Un programma può esser visto come una sequenza di chiamate procedurali. Il main program è responsabile di passare dati alle varie singole chiamate, i dati sono processati da delle procedure.
- Un programma può esser visto come una rete di oggetti interagenti tra loro, ciascuno autosufficiente.



History of C++

- C++ fu inventato da Bjarne Stroustrup, circa negli anni 80'.
- C++ non è semplicemente un linguaggio di programmazione Object-Oriented, è un multi-paradigm
- ISO Standard in 1998 (C++98), aggiornato nel 2003

C++ and C

- C++ è un diretto discendente del C che conserva quasi tutto il C come sottoinsieme.
- C++ fornisce un controllo maggiore sui tipi rispetto al C e supporta un più ampio range di stili di programmazione di quanto non faccia il C.

C++ sub-languages

- C++ può essere considerato come una federazione di linguaggi; i principali sotto linguaggi sono:
- C: poiché il C++ supporta tutte le tecniche di programmazione supportate dal C
- Object Oriented C++: con classi, incapsulazione, ereditarietà, polimorfismo, etc.
- Template C++: generic programming in C++

Makefiles

- Progetti più ampi richiederanno l'uso di Makefile.
- Un makefile è un file di testo a cui si fa riferimento per mezzo del comando make che descrive la costruzione di obiettivi (eseguibili e librerie), e contiene informazioni come le dipendenze source-level e le dipendenze dell'ordine di costruzione.
- Eclipse è un IDE che tratta questi makefiles.

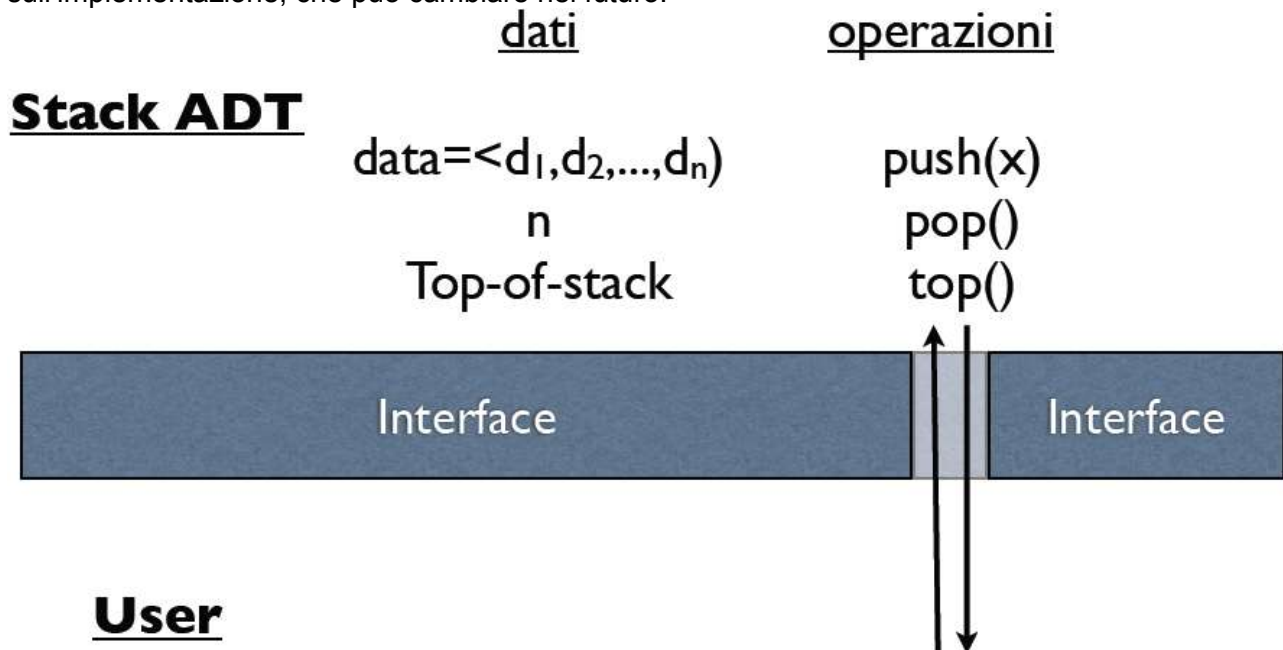
Classes and objects

Perchè astrarre?

- Aiuta a dare un modello al problema, separando i dettagli necessari da quelli inutili.
- Vogliamo ottenere una separazione tra:
 - le operazioni eseguite sui dati
 - La rappresentazione delle strutture dati e l'implementazione degli algoritmi.
- L'astrazione è lo strutturare un problema nebuloso e indefinito in precisi schemi definendo i dati e le operazioni(accoppiate).

ADT (Abstract Data Type)

- Un ADT è un'individuazione di un set di dati e di un set di operazioni (chiamata interfaccia dell'ADT) che possono essere eseguite sui dati.
- È un'astrazione nel senso che è indipendente dalle varie concrete implementazioni.
- Quando un programma è realizzato, l'ADT è rappresentato da un'interfaccia, che maschera una corrispondente implementazione. Gli utenti di un ADT sono concentrati sull'interfaccia, non sull'implementazione, che può cambiare nel futuro.



Why encapsulation ?

- Il principio di nascondere le strutture dati usate e di fornire solo una meglio definita interfaccia è detto incapsulazione.
- La separazione di strutture dati e operazioni e le restrizioni per cui si può accedere alla struttura dato solo attraverso l'interfaccia(ben definita), permette di scegliere la struttura dati appropriata per l'ambiente dell'applicazione.

Why classes?

- Una classe è una reale rappresentazione di un ADT: essa fornisce dettagli implementativi per la struttura dati utilizzata e le operazioni.
- Richiama l'importante distinzione tra una classe e un'oggetto:
 - Una classe è una rappresentazione astratta di una serie di oggetti che si comportano identicamente.

- Oggetti(ovvero variabili) sono istanziati dalle classi.
- Le classi definiscono le proprietà e il comportamento di un insieme di oggetti.

Classes and objects

- Una classe è un'implementazione di un ADT. Definisce gli attributi e i metodi che implementano la struttura dati e le operazioni dell'ADT rispettivamente.
- Un **oggetto** è un'**istanza** di una **classe**. Può essere identificato univocamente col suo **nome** e definisce uno **stato**, il quale è rappresentato dai valori dei suoi attributi in un particolare momento.
- Il comportamento di un oggetto è definito da un insieme di metodi che possono essere applicati su di esso.

Procedural programming

- C'è una divisione tra i dati e le operazioni su di essi.
- Il centro della programmazione procedurale è di scomporre i compiti di un programma in una serie di variabili, strutture dati e subroutines.
- Quando si programma in C ci concentriamo su le strutture dati e le funzioni.

OO programming

- Nella programmazione object-oriented ci si concentra nello scomporre un programma in oggetti e le iterazione tra gli oggetti.
- Un oggetto è associato a dati e operazioni su tali dati, ad esempio:
 - L'oggetto "oven" ha un dato interno che rappresenta la temperatura e un'operazione attraverso la quale è possibile impostare una nuova temperatura.

Why C++ classes ?

- Una classe C++ può prevedere l'information hiding:
 - Nasconde la rappresentazione interna dei dati
 - Nasconde i dettagli implementativi delle operazioni.
- La classe agisce come una scatola nera, fornendo un servizio ai propri clienti, senza rivelare il proprio codice che potrebbe essere usato in modo errato.

Il principio dell'Open-Closed

- L'incapsulazione è una tecnica chiave per seguire il principio dell'Open-Closed:
 - Le classi dovrebbero essere aperte all'estensione ma chiuse per la modifica.
- Vogliamo permettere cambiamenti al sistema, ma senza richiedere una modifica del codice esistente.
- Se una classe ha un particolare comportamento, codificato nella maniera che preferiamo, se nessuno può cambiare il codice della classe l'abbiamo chiuso a modifiche.
- Ma, se per qualche ragione, dobbiamo estendere tale comportamento, possiamo lasciar estendere la classe di cui dobbiamo sovrascrivere il metodo e fornirle nuove funzionalità. La classe è aperta all'estensione.
- Vedremo come l'ereditarietà e la composizione ci aiuterà a seguire tale principio.

Class identification

- Identificare oggetti reali o entità del mondo come potenziali classi di oggetti software.
- L'approccio consueto è quello di pensare agli oggetti del mondo reale che esistono nel dominio dell'applicazione che sta per essere programmata. Invece di pensare ai processi che devono essere eseguiti, come facevamo solitamente nella programmazione procedurale, pensiamo invece alle cose del mondo reale.
- Identificare gruppi di oggetti che si comportano similmente, che possono essere implementati come classi.
- Le classi sono descrizioni per gli oggetti.

- Ritardare le decisioni circa i dettagli implementativi, come per esempio ciò per cui i dati e le operazioni si applicheranno agli oggetti fino a quando non abbiamo una chiara idea di quali classi di oggetti saranno richieste.
- L'iniziare a modellare una classe identificando le classi candidate - una lista iniziale di classi dalle quali emergeranno le classi del vero Design.
- Una regola generale per identificare le classi candidate: identificare il nome e le frasi sostantivo, i verbi(azioni),e gli aggettivi(attributi) dai casi e dalla descrizione del problema.
 - Ci sono metodi più formali per identificare(ad esempio CRC cards, use cases) e rappresentare(e.g. UML) le classi.

Single responsibility principle

- Ogni oggetto nel suo sistema dovrebbe avere una singola responsabilità, e tutti i servizi dell'oggetto dovrebbero essere concentrati sul esternare questa unica responsabilità.
- Una classe dovrebbe avere solo un motivo per cambiare.
- Una responsabilità può essere definita come una ragione buona per cambiare.
- È un concetto collegato alla coesione.
- Per esempio considera un modulo che compila e stampa un report: il contenuto del report può cambiare, il formato del report può cambiare.
- Il principio della singola responsabilità dice che questi due aspetti del problema sono davvero due responsabilità separate, e dovrebbero dunque stare in classi separate o moduli.
- Non accoppiare due cose che cambiano per ragioni differenti in momenti differenti.

SRP: example

- Ecco un semplice test per capire se una classe segue l'SRP:
per ogni metodo della classe scrivi una riga che dice

The class name write method here itself.

- Adatta la grammatica e la sintassi e leggila ad alta voce. Ha senso?
- Se non lo ha probabilmente il metodo appartiene ad un'altra classe. Use common sense!

- Applica il metodo alla classe Automobile: →
- Siamo ancora molto lontani dall'avere macchine che si guidano da sole(abbiamo bisogno di un autista)
- Sicuramente non si autocambieranno le gomme o si autolaveranno (ma avranno bisogno di un meccanico)
- Pensa bene al significato dei metodi: getOil significa semplicemente che la macchina ha un sensore.

Automobile
start()
stop()
changeTires(Tire[])
drive()
wash()
checkOil()
getOil() : int

C++ Classes

- Una classe C++ estende il concetto di strutture C
- Mette insieme gruppi di variabili(attributi o dati membro) che possono essere referenziati usando un nome collettivo e un identificatore simbolico.
- Può avere funzioni (metodi o funzioni membro) che operano nel contesto della classe
- Definisce un tipo di dato: possiamo crearne un'istanza(detto oggetto).

Class definition

- Use the keyword class, ad esempio:

```
class Stack {
bool push(data value);
bool pop(data* pValue);
void init(int size);
```



```
int TOS;
data* buffer;
int size;
}; // do NOT forget the ; !
```

A C stack implementation

```
struct stack {
    int TOS;
    data *buffer;
    int size; };
```

```
bool push(struct stack *ptr, data value) {...}
```

```
bool pop(struct stack *ptr, data *pValue) {...}
```

Access level

- Tutti i membri di una struct sono visibili non appena c'è un riferimento alla struttura, mentre in una classe è possibile differenziare il tipo di accesso come pubblico, privato o protetto (l'accesso di default è impostato su privato).
- Possiamo progettare meglio l'“interfaccia” della classe, cioè decidere ciò che può essere nascosto e ciò che può essere visibile nella classe (incapsulazione). Possiamo disaccoppiare le classi.

Access levels

- **Public:** un membro pubblico è visibile a **tutti** quanti i quali abbiano l'indirizzo o il riferimento all'oggetto.
- **Private:** un membro privato è visibile **solo** ai metodi della **classe in cui è definito**.
- **Protected:** un membro protetto è visibile soltanto ai **metodi** della **classe in cui è definito**, e in **tutte** le classi **derivate** (fino in fondo col meccanismo dell'ereditarietà).

Example:

```
class Stack {
public:
    bool push(data value);
    bool pop(data* pValue);
    void init(int size);

private:
    int TOS;
    data* buffer;
    int size;
};
```

Access levels: regola generale

- Usa sempre un controllo di accesso esplicito
- Non avere dati membro (attributi) pubblici
 - Utilizza metodi pubblici per set/get i loro valori (dei dati membro).

- Diversi IDEs possono creare questi metodi automaticamente.

Method implementation

- Di solito i metodi sono definiti(implementati) nei files .cpp:
Aggiungi il nome della classe davanti al metodo ad esempio:

```
bool stack::push(data value) {  
    // code to implement the method  
}
```

- Possiamo implementarli anche nell'header(inline), ma di solito questo viene fatto solo se questi sono molto corti(ad esempio 5 o 7 righe)

Attributes

- Un metodo può accedere agli attributi della classe: gli attributi sono visibili all'interno dei metodi.
 - Ciò riduce di molto la complessità delle interfacce del C: paragona l'implementazione C++ con quella del C.
- Gli attributi mantengono lo "stato" degli oggetti
- Gli attributi sono una sorta di contesto condiviso dai metodi(è per questo che le interfacce sono più semplici)
- Comunque, i metodi sono più accoppiati con gli attributi.
- Vale la pena pagare questo prezzo se le classi sono state progettate per avere responsabilità comuni.

Argument passing

- In C il meccanismo di passaggio avviene "per valore": il valore dei parametri attuali(arguments) viene copiato nei parametri formali.
 - Una funzione usa la copia dei valori per effettuare i suoi calcoli.
 - Noi dobbiamo usare puntatori per simulare un passaggio "per riferimento".
- In C++ possiamo passare parametri anche per riferimento.

Pass by reference

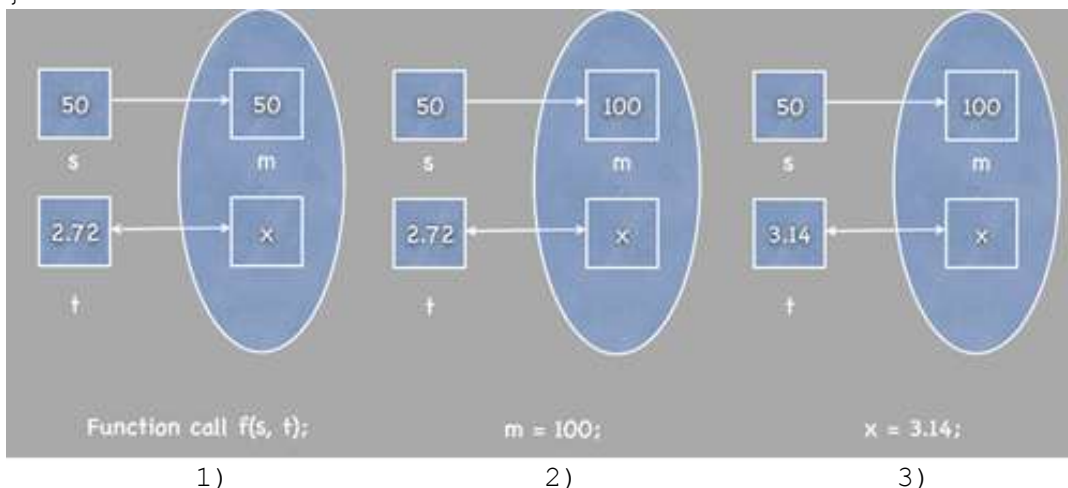
- Un riferimento è essenzialmente un sinonimo in senso che non c'è una copia del dato passata nei parametri attuali(actual argument).
- Si indica con il simbolo e commerciale & che segue il tipo del parametro attuale(base).
- In C++, la chiamata per riferimento e la chiamata simulata in stile C usando puntatori sono simili, ma non ci sono puntatori coinvolti esplicitamente: non c'è bisogno di deferenziare il parametro attuale(argument).

```
void add(int a, int b, int& sum) {  
    sum = a + b;}  
  
int main() {  
    int i = 1;  
    int j = 2;  
    int total = 0;  
  
    cout << "total: " << total << endl;  
    add( i, j, total);  
    cout << "total: " << total << endl;  
}
```

```

void f(int m, double& x) {
m = 100;
x = 3.14
}
int main() {
int s = 50;
double t = 2.72;
f(s,t);
return 0;
}

```



- Un riferimento può essere specificato come `const`: la funzione/metodo non può modificare il contenuto della variabile.
- Passa grandi strutture dati che non devono essere modificate come riferimento costante (è veloce)

```

42 void add(int a, int b, const int& sum) {
43     sum = a + b;
44 }
45

```

Reference variables

- È possibile avere un riferimento variabile, ma deve sempre tenere un riferimento valido, e quindi deve essere inizializzato quando viene creato.

```

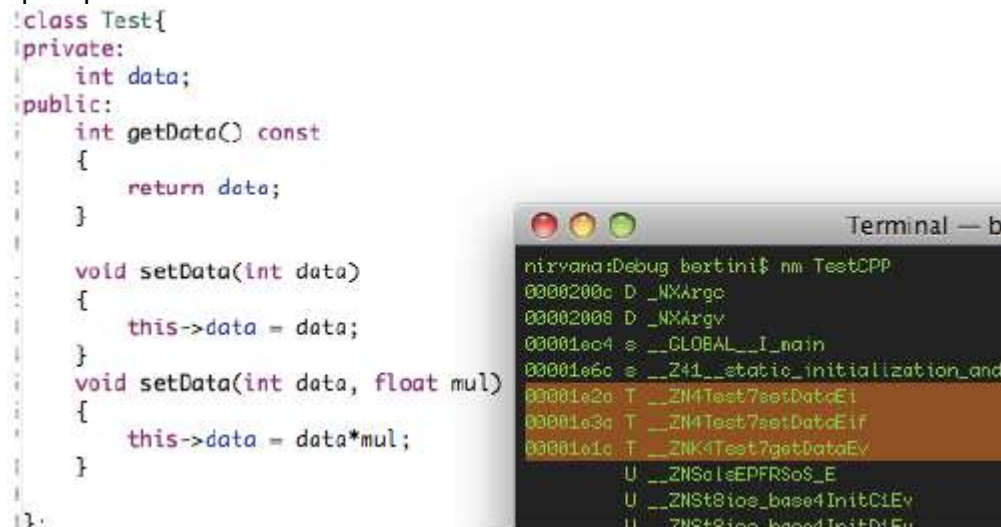
int x;
int& y=x; // reference
y=2; // also x is modified
int& z; // Error: doesn't compile ! Why ?
int *z; // pointer
z = &x; // & on the left is different from & on the right of =
*z = 3; // x is modified

```

Overloading

- Possiamo definire più di un metodo con lo stesso nome e tipo di ritorno, ma con differenti (in numero e tipo) parametri (signature). Tale metodo è detto sovraccaricato (overloaded).
- Comunemente usato per fornire una versione alternativa di funzioni per essere usata in differenti situazioni.
- La stessa operazione astratta può avere differenti implementazioni concrete.

- Il compilatore creerà un differente segmento di codice e un simbolo (attraverso lo storpiamento (mangling) del nome) ottenuto estendendo il nome del metodo con suffissi relativi ai tipi di parametri.



- **Non possiamo semplicemente cambiare il valore di ritorno:** il compilatore dovrebbe sempre controllare il tipo della variabile dove mettiamo il valore.. e che succede se noi ce ne liberiamo?

Operator overloading

- È possibile sovraccaricare anche gli operatori, non solo i metodi (nella vita reale + è un operatore usato per interi, numeri reali, numeri complessi..)
- Sovraccaricare gli operatori solo quando ha davvero senso
 - Ad esempio sovraccaricare l' '=' per comparare stringhe, (do not overload * for strings...)
- Alcuni linguaggi OO non permettono il sovraccaricamento degli operatori, in particolare il Java.
- Gli operatori sono sovraccaricati nel senso che gli oggetti si comportano come tipi primitivi. Non possono essere creati nuovi operatori, possono essere modificate solo le funzionalità di operatori già esistenti.
- Se tu sovraccarichi il + non devi aspettarti che il += sia dato automaticamente! Va ridefinito anche tale operatore.
- Spesso gli operatori sono semplicemente friend... (more later)

```

class Array {
public:
    Array(int size); // constructor
    bool operator==(const Array& right) const; //the method can't modify
                                                //anything

    // ... other members
private:
    int size;
    int* data; // pointer to first element of array};

    bool Array::operator==(const Array& right) const {
        if ( size != right.size ) // start checking the size of the arrays
            return false;
        // then check the whole content of arrays
        for ( int i=0; i < size; i++ ) {
            if ( data[i] != right.data[i] )
                return false;
        }
        return true; // both size and content are equal
    }
}
  
```

Type checking

- C++ ha un controllo molto più severo sui tipi rispetto al C: dipende dal cast dei parametri (depending on the parameter cast you determine the method that is executed !)
- Ad esempio: Fai il cast dei puntatori a vuoto quando li assigni ad altri puntatori (in C questo compila lo stesso). (cast void pointers when assigning them to other pointers, in C it compiles).

```
29     int aInt = 3;
30     void* ptr = &aInt;
31     int* pInt;
32     bInt = ptr;
33     pInt = (int*)ptr;
34
```

Object creation

- Una volta che una classe è definita possiamo creare le istanze (oggetti) da essa, come si fa per le variabili dei tipi base.
- La creazione può essere statica (sullo stack) o dinamica (sull'heap)
- Il codice dei metodi è rappresentato nel segmento di codice, condiviso tra tutte le istanze di una classe.
- Ogni oggetto ha un indirizzo della funzione che implementa i metodi.

Dynamic object creation

- È molto simile all'utilizzo delle funzioni di malloc/free, ma sintatticamente semplificato, usando il new e il delete:

```
stack* sPtr;
sPtr = new stack;
...
delete sPtr;
```

Constructors

- È una funzione membro che sarà invocata quando un **oggetto** di tale classe è **creato**. Non restituisce valore.
 - Ha sempre lo **stesso nome** della **classe**. In generale i costruttori compiono una sorta di inizializzazione di un oggetto. Se non viene definito un costruttore ne viene generato uno di default, senza alcun parametro.
 - È **comune sovraccaricare** una funzione **costruttore** (cioè fornirne più versioni) cosicché l'oggetto possa essere creato in differenti maniere.
- Considera come oggetti di un nuovo tipo possano essere creati e quali costruttori sono necessari.
- Se non è definito un costruttore il compilatore ne genera uno di default che non prende parametri.
- Il **costruttore di default** è **solitamente invocato senza parentesi**, ad esempio nell'esempio precedente:

```
sPtr = new stack;
```

- Se una classe ha qualche costruttore ma non ne ha uno di default la sua creazione sarà costretta a situazioni gestite dai costruttori, ad esempio

```
class B {
public:
    B(int i) { ... }
};
B b1; // illegal
B b3(123); // ok
```

- **C'è una maniera compatta e compiler-friendly per inizializzare gli attributi in un costruttore:**

```
class stack {
    protected:
        int TOS;
        data *buffer;
        int size;
    public:
        stack(int s) : TOS(0), size(s), buffer(new data[s]) {...}
```

- I costruttori sono pubblici (di solito, ma non necessariamente)
- Se non vogliamo che una classe sia istanziata possiamo dichiarare un costruttore come protetto. Possiamo istanziare le classi derivate (se il loro costruttore è pubblico).
- In altri casi possiamo dichiarare un costruttore come private.
- Il suo utilizzo è tipicamente collegato ai metodi static.

Explicit constructors

- I **costruttori** in C++ che hanno solo un parametro eseguono implicitamente una **conversione del tipo**, ad esempio: se passi un intero quando un costruttore si aspetta un parametro puntatore a stringa, il compilatore aggiungerà il codice che deve convertire l'intero in un puntatore a stringa.
- **Puoi aggiungere esplicitamente al costruttore una declaration per prevenire queste conversioni implicite.**
- Dichiarare un costruttore che ha più parametri che devono essere espliciti, poichè in tal caso questi costruttori non possono prendere parte alle conversioni implicite.
- Comunque, l'explicitare avrà un effetto se un costruttore ha più parametri e tutti eccetto uno dei parametri hanno assegnato un valore di default.

Explicit constructors: example

```
class A {
public:
    A();
};
class B {
public:
    explicit B(int x=0, bool b=true);
};
class C {
public:
    explicit C(int x);
};
```

```
void doSomething(B objB);
B objB1;
```

```
doSomething( bObj1 ); // OK
B objB2( 28 ); // OK, b arg is set to default
doSomething(28); // BAD: we need a B obj, and we do not allow implicit
// conversion
doSomething(B(28)); // OK
doSomething("foo"); // BAD, thanks the compiler for not allowing it
```

- **È preferibile utilizzare costruttori espliciti** (c'è anche una Google C++ guideline for it)
- Mentre progetti un tipo (ovvero una classe) pensa quali conversioni dovrebbero essere permesse:

Dovresti scrivere un tipo di funzione di conversione o un costruttore non esplicito(con un solo parametro)?

Destructors

- È un metodo con lo stesso nome della classe preceduto da ~, ad esempio: ~stack();
- Niente parametri, niente valori di ritorno, nessun overload
- Viene chiamato automaticamente quando un oggetto viene distrutto.
- Dovrebbe mantenere un sistema che lavori in modo corretto(housekeeping).

C'tor and D'tor

```
class stack {
    protected:
        int TOS;
        data* buffer;
        int size;
    public:
        stack(int s);
        ~stack();
//..}

// C'tor allocates memory
stack::stack(int s)
{
    TOS=0;
    size=s;
    buffer = new data[size];
}

// D'tor has to release memory
stack::~~stack()
{
    delete(buffer);
}
```

Come si usano metodi e attribute?

- I membri di una classe possono essere referenziati analogamente ai membri di uno struct:

```
<var>.member_name
<expr_addr>->member_name
```

Ma tenendo conto della loro visibilità, definita dal livello di accesso. Ad esempio:

```
stack S;
stack *pS;
...
pS = &S;
...
S.push(3);
...
pS->push(8);
```

Self reference

- Un **oggetto** può far riferimento a se stesso tramite la parola **this**.
- Un **oggetto** usa implicitamente **this** quando si riferisce a un **metodo** o ad un **attributo**.

```

stack::stack(int s)
{
TOS=0;
size=s;
buffer = new data[size];
}

```

```

stack::stack(int s)
{
this->TOS=0;
this->size=s;
this->buffer = new data[this->size];
}

```

- L'uso di this è essenziale quando un oggetto deve passare un riferimento a se stesso ad un altro oggetto.

- Una tipica applicazione è il callback(richiamo): l'oggetto A dà un riferimento a se stesso che sarà utilizzato dall'oggetto B per invocare un metodo sull'oggetto A.

- **This è usato per implementare schemi di inversione di responsabilità:**

l'oggetto A non chiama l'oggetto B per eseguire un'operazione ma lascia che l'oggetto B chiami l'oggetto A.

```

class observer;
class subject;

```

```

class observer {

    public:
        void update(subject* pSubj);
        int getState() {return state;}

    private:
        int state;
};

```

```

class subject {
    public:
        subject(observer* pObs);
        void setState(int aState);
        int getState() {return state;}

    private:
        int state;
        observer* pObs;
};

```

```

observer::update(subject* pSubj) {
if ( ... ) // possible condition that starts an update
this->state = pSubj->getState();
}

```

```

subject::subject(observer* pObs) {
this->pObs = pObs;
}

```

```

subject::setState(int aState) {
this->state = aState;
this->pObs->update( this );
}

```

```

int main() {
subject* pSubj;

```



```

observer* pObs;
pObs = new observer;
pSubj = new subject( pObs );
// ...
pSubj->setState( 10 );
cout << "subj state: " << pSubj->getState << endl;
cout << "obs state: " << pObs->getState() << endl;
}

```

Static members

- Un membro statico è associato con la classe, non con l'oggetto(un'istanza della classe), cioè c'è un'unica copia del membro per tutte le istanze
 - Estende le variabili statiche del C.
- Dati membro statici: una copia della variabile
- **Funzioni membro statiche: possono essere invocate senza necessitare di un oggetto**

Static data members

```

class Point {

    public:
        Point()      {x=y=0;
                      n++;}

        ~Point()     {n--;}

        int count() const { return n;}
// ...

    private:
        int x,y;
        static int n; // declaration
}
// definition: must be in namespace scope
int Point::n = 0;

int main() {
    Point a,b;
    cout << "n: " << p.count() << endl;
}

```

Static method members

```

class Point {
    public:
        Point()      {x=y=0;
                      n++;}

        ~Point()     {n--;}

    static int n;
    static float distance(const Point a, const Point b) {
//...calc distance }
// ...
}

```

```

    private:
        int x,y;
}

// definition: must be in namespace scope
int Point::n = 0;
int main() {
    // access static members even before the creation of any instance of the
    class
    cout << "n: " << Point::n << endl;
    Point a,b;
    // set a and b coordinates
    Point::distance(a,b);
}

```

Friend

- Una classe può permettere l'accesso ai suoi membri (anche se privati) dichiarando che le funzioni del livello superiore (top-level functions) sono amiche (friends).
- Friends dovrebbero essere usate solo in situazioni molto speciali, ad esempio I/O operator di overloads dove non è desiderabile fornire funzioni membro accessorie
 - Rende difficile l'incapsulazione:

```

class Point{

    private:
        int x,y;

    public:
        friend bool operator ==(Point a, Point b);
        Point() : x(0), y(0) {};
//... }

```

```

bool operator ==(Point a, Point b) {
    if ( ( a.x != b.x ) || ( a.y != b.y ) )
        return false;
    else
        return true;}

```

```

int main() {
    Point p, q;
    //...
    if (p == q)
        std::cout << "p and q are equal" << endl;

    return 0;}

```

Coding style guidelines for classes

Classes and Objects

- I nomi che rappresentano i tipi(cioè le classi) i casi(maiscolo/minuscolo) devono essere mixati partendo dal maiuscolo, ad esempio:

`Line, SavingsAccount`

- I nomi delle variabili devono essere invece mixati partendo dal minuscolo, ad esempio:

`line, savingsAccount`

- Le parti di una classe devono essere ordinate da public a protected a private.

Tutte le sezioni devono essere identificate esplicitamente. Le sezioni non applicabili dovrebbero essere lasciate da parte.

- Una classe dovrebbe essere dichiarata nel file di header e definita in un file sorgente dove il nome dei files corrispondono al nome della classe.

- Tutte le definizioni dovrebbero risiedere nei files sorgente. Quest'ultima regola è particolarmente difficile da seguire: Eclipse CDT crea i metodi getter/setter all'interno della dichiarazione della classe...

Methods

- I nomi che rappresentano i metodi o le funzioni devono essere verbi e scritti in case mixato partendo dal minuscolo(come nel Java), ad esempio:

`getName(), computeTotalWidth()`

- Il nome degli oggetti è implicito e dovrebbe essere evitato nel nome di un metodo, ad esempio:

`line.getLength();` // NOT: (così e non come qui sotto:)

`line.getLineNumber();`

Attributes

- Le variabili private di una classe dovrebbero avere un suffisso underscore, ad esempio:

```
class SomeClass {  
private:  
int length_  
}
```

- Questa regola è molto discussa. Altri approcci corretti sono: il prefisso di underscore, m_ prefix, o alcun suffisso o prefisso.

Inheritance

Why inheritance ?

- Per il riutilizzo del software: riutilizzare una classe esistente nelle nuove classi è una specializzazione per la classe base.
- Una nuova classe è derivata da quella base e eredita le strutture della classe base.
 - Una classe derivata può essere a sua volta la base di un'ulteriore ereditarietà – forma una classe di gerarchia
 - La classe derivata **estende** le funzionalità della classe base.
- L'ereditarietà è il secondo concetto più importante nella programmazione OO – il primo è l'abstract data type.
- L'ereditarietà **ci permette di evitare la duplicazione di codice** o di funzioni prendendo tutte le caratteristiche di un'altra classe semplicemente nominandola in una ereditarietà.
 - Poi, se i dati privati o il codice necessario per implementare qualunque delle comuni caratteristiche ha bisogno di esser cambiato, viene cambiata solo nella classe base e non nelle classi derivate che ottengono i cambiamenti automaticamente.

When to use inheritance

- Usare l'ereditarietà come un dispositivo specifico.
- “Gli esseri umani astraggono cose in due dimensioni: parte-di e tipo-di. Una Ford Taurus è un tipo-di Macchina, e una Ford Taurus ha un Motore, Gomme, etc.. La parte della gerarchia è stata una parte del software da quando lo stile ADT è diventata rilevante; l'ereditarietà aggiunge “l'altra” maggiore dimensione della decomposizione.”

Da: C++ FAQ Lite - [19.2]

Using inheritance

- Il progetto delle gerarchie di classe è un'abilità chiave nel OODesign.
- Usa solo l'ereditarietà quando c'è una chiara relazione di “is a” tra le classi derivate e la classe base.
- L'ereditarietà esprime le naturali relazioni tra le cose tra, per esempio, “un bus is a/è un veicolo”
- Una istanza di una classe derivata potrebbe sostituire un'istanza di una classe base (derived is_a base).

Inheritance in C++ - example

```
class Person {  
    public:  
        const string& GetName() const;  
    // ...    };
```

```
class Student : public Person { // ... };  
class Staff : public Person { // ... };  
class Permanent : public Staff { // ... };  
class Casual : public Staff { // ... };
```

- Dopo ogni classe derivata c'è un due punti “:” seguito dalla parola “public” e poi il nome della classe da cui si eredita. Due punti rappresenta l'ereditarietà.
- La keyword public dopo i due punti dice che stiamo utilizzando una ereditarietà pubblica. Questo è il modo più comune di ereditare sebbene sia possibile avere ereditarietà di tipo protetto o privato.

- Questi differenti tipi di ereditarietà si riferiscono a se i membri pubblici della classe base saranno o meno accessibili agli utenti della classe derivata. Con l'ereditarietà di tipo pubblico i membri della classe base diventano effettivamente membri pubblici della classe derivata.

Inheritance access specifiers

```
class D : public B {};  
class D : protected B {};  
class D : private B {};
```

```
class B {  
public:  
void pub();  
protected:  
void prot();  
private:  
void priv();  
};
```

		Base class member access		
		public	protected	private
Derived class inheritance access	public	whatever function	methods of D friends of D classes derived from D	not accessible
	protected	methods of D friends of D classes derived from D	methods of D friends of D classes derived from D	not accessible
	private	methods of D friends of D	methods of D friends of D	not accessible

Inheritance access specifiers

- Una classe derivata **pubblicamente** eredita i membri pubblici e protetti della classe base **mantenendo il loro livello di accesso**.
- Una classe derivata in maniera **protetta** eredita i membri pubblici e protetti della classe base ma li esibisce come **membri protetti**.
- Una classe derivata in maniera **privata** esibisce i membri pubblici e privati della classe base come **membri privati**.

Class interface

- Una classe ha due distinti set di interfacce:
 - Ha una interfaccia pubblica che serve per le classi non imparentate(non in relazione).
 - Ha un interfaccia protetta che serve le classi derivate.

Access control

- I membri privati di una classe rimangono solo privati! Una classe derivata non può accedere ai membri privati della classe base, sebbene li abbia ereditati (sono inclusi in un oggetto della classe derivata).
- I membri privati sono accessibili solo tramite i metodi pubblici della classe base. Non sono accessibili direttamente dagli utenti della classe derivata e non possono neppure essere accessibili dai metodi della classe derivata.

Protected Members

- Come facciamo se vogliamo che un membro della classe sia visibile ai metodi di una classe derivata ma e che non sia visibile nemmeno agli utenti della classe base o di quella derivata?
 - I membri protetti del C++
- Se ci sono livelli di ereditarietà indiretta attraverso una gerarchia della classe, i membri protetti saranno accessibili attraverso tutta la gerarchia della classe.

```
class baseClass{
public:
    void method1();
protected:
    void method2();};
```

```
class derivedClass : public baseClass {
public:
    void method3 () { method2(); // OK};};
```

```
derivedClass d;
d.method1(); // OK
d.method3(); // OK      lui può chiamare method2 perchè pubblico
d.method2(); // ERROR!  method2 is protected
                        //Puoi accederci appunto solo tramite un metodo della classe
                        // derivata, d.method3 () che richiama la funzione method2()
```

Access control hint(indizio)

- Dichiarare i tuoi dati membro della classe base come privati e utilizzare funzioni di accesso inline *protette* (use protected inline access functions) attraverso le quali le classi derivate accederanno ai dati privati nella classe base. In tal modo le dichiarazioni dei dati privati possono cambiare, ma il codice della classe derivata non si romperà (ammeno che tu non cambi l'accesso protetto alle funzioni)

Da: C++ FAQ Lite - [19.7]

Accessing Base Class Members

- Un oggetto di una classe derivata eredita i membri della classe base, ad esempio:

Where is it implemented? Looks like implemented in Casual class

```
• Casual cas;
cout << "Name: " << cas.GetName() << endl;
```

- Il vero potere dell'ereditarietà viene quando non sappiamo il momentaneo tipo di un'oggetto, ad esempio:

p is a pointer to the base class
FindPerson may return derived
classes, but we can invoke methods
of the base class without knowing
what p has become

- ```
Person *p;
p = FindPerson(...);
cout << "Name: " << p->GetName() << endl;
```

  
*//in pratica anche se p punta una classe di cui non conosciamo bene il  
//nome ma di cui sappiamo che in qualche modo eredita dalla classe base  
//possiamo comunque invocare i metodi della classe base*
- Questo è un'esempio di polimorfismo

## Inheritance vs. Composition

- Perché non questo?
    - ```
Class Student {  
public:  
    Person details;  
    // ...  
};
```
 - Questa è la composizione. Viene usata quando gli oggetti di una classe contengono o comprendono uno o più oggetti di un'altra classe:
 - ```
Student s;
cout << "Name: " << s.details.GetName();
```
- Notice access using two levels of member selection  
Nota l'accesso usando due livelli della selezione del membro (il primo punto per entrare negli attributi di class → entri in un oggetto Person → secondo punto per selezionare un metodo dell'oggetto (details)).

- Usa l'ereditarietà per le relazioni di "is a", di "ha\_un" o di "contiene o di "è compreso in" ("has\_a" or "contains" or "is comprised of").

• Considera il caso in cui ci siano più istanze di una classe all'interno di un'altra classe, ad esempio:

```
class Person {
public:
 Address home;
 Address office;
 // ...};
```

- Questo con l'ereditarietà non si può fare!

## Composition and relationships

- Quando un oggetto contiene un altro oggetto ci potrebbe essere una relazione che è una forma differente di has\_a, ed è molto più una relazione di è implementata in termini di, ad esempio: Quando una classe dipende molto dal comportamento di una classe contenuta, modificandone alcune delle sue caratteristiche.

# Using derived classes

È possibile usare un oggetto inizializzato da una classe derivata in qualsiasi momento, è possibile usare un oggetto inizializzato dalla classe base(perchè un oggetto derivato è\_un oggetto base):

```
class Employee {
 string first_name, family_name;
 Date hiring_date;
 short department;
 // ...};
```

```
class Manager : public Employee {
 set<Employee*> group;
 short level;
 // ...
};
```

```
void paySalary(Employee* e)
{ //... code to pay salary }
```

```
//...
Employee *e1;
Manager *m1;
//...
paySalary(e1);
paySalary(m1);
```

## Public inheritance and is\_a

- Se S estende pubblicamente P allora S è\_un P e qualunque funzione che esegue un P(o puntatore a P o riferimento a P) sarà fattibile anche per un S(o puntatore a S o riferimento a S) (ovvero Derived=Student, Base=Person)

```
class Person {...};
class Student : public Person {...};
void eat(const Person& p);
void study(const Student& s);
Person p;
Student s;
eat(p); // OK
eat(s); //OK: s is_a p
study(s); // OK
study(p); // bad: p is not an s
```

- Ma attenzione alla **progettazione**:

```
class Bird {
 public:
 virtual void fly();
 ...};

class Penguin : public Bird { ... };

Penguin p;
p.fly(); // but penguinsdo not fly !
```



- Forse sarebbe meglio avere:

```
class Bird { ... };

class FlyingBird : public Bird {
public:
 virtual void fly();}

class Penguin : public Bird { ... };
```

Quindi generare una sottoclasse dove ci sono anche gli uccelli che non volano.

- L'ereditarietà pubblica asserisce che tutto ciò che è applicabile agli oggetti base è applicabile anche agli oggetti derivati, sta a te progettare correttamente la classe base, così che i pinguni non volino!

## Private inheritance

- Il comportamento è un po' differente quando si eredita privatamente: non abbiamo più una relazione di `is_a`, il compilatore non convertirà le classi derivate alla classe base:

```
class Student : private Person { ... };
void eat(const Person& p);
Student s;
eat(s); // error: now a Student is not a Person !
```

- Tutto ciò che è ereditato diventa private: è un dettaglio implementativo
- L'ereditarietà privata significa che la classe derivata D è implementata in termini della classe base B, ma non che D è un B.
- Utilizza l'ereditarietà **privata** se vuoi **ereditare solo l'implementazione** della **classe base**, utilizza l'ereditarietà **pubblica** se vuoi **ereditare anche l'interfaccia**.
- Ricorda che anche la composizione permette di implementare una classe in termini di un'altra classe (composta)
- Usa la composizione ogni volta che puoi e l'ereditarietà private quando necessita, ad esempio quando hai bisogno accedere a parti protette di una classe o di ridefinire metodi virtuali(...).

## Constructors and inheritance

- Quando un oggetto di una classe derivata è creato, i costruttori(se ci sono) di ogni classe ereditata sono invocati in ordine dalla classe base verso quella derivata. È un processo dal fondo verso l'alto(**bottom-up**)
- I costruttori di default sono invocati automaticamente.
- Se una classe base non ha un costruttore di default, qualunque altro costruttore deve essere invocato esplicitamente dal costruttore della classe derivate nella sua lista di inizializzazione.

```
class Derived: public Base {
 private:
 int d;
 public:
 Derived();
 Derived(int a, int b, int c, int d);
```

```
void print();};
```

```
Derived::Derived() { d=0;}
Derived::Derived(int a=0, int b=0, int c=0, int d=0) : Base(a,b,c)
{ this->d = d; } // Use a,b,c as parameters to the c'tor of Base
Derived::Derived(int a=0, int b=0, int c=0, int d=0) : Base(a,b,c) , d(d)
{}
```

## Destructors and Inheritance

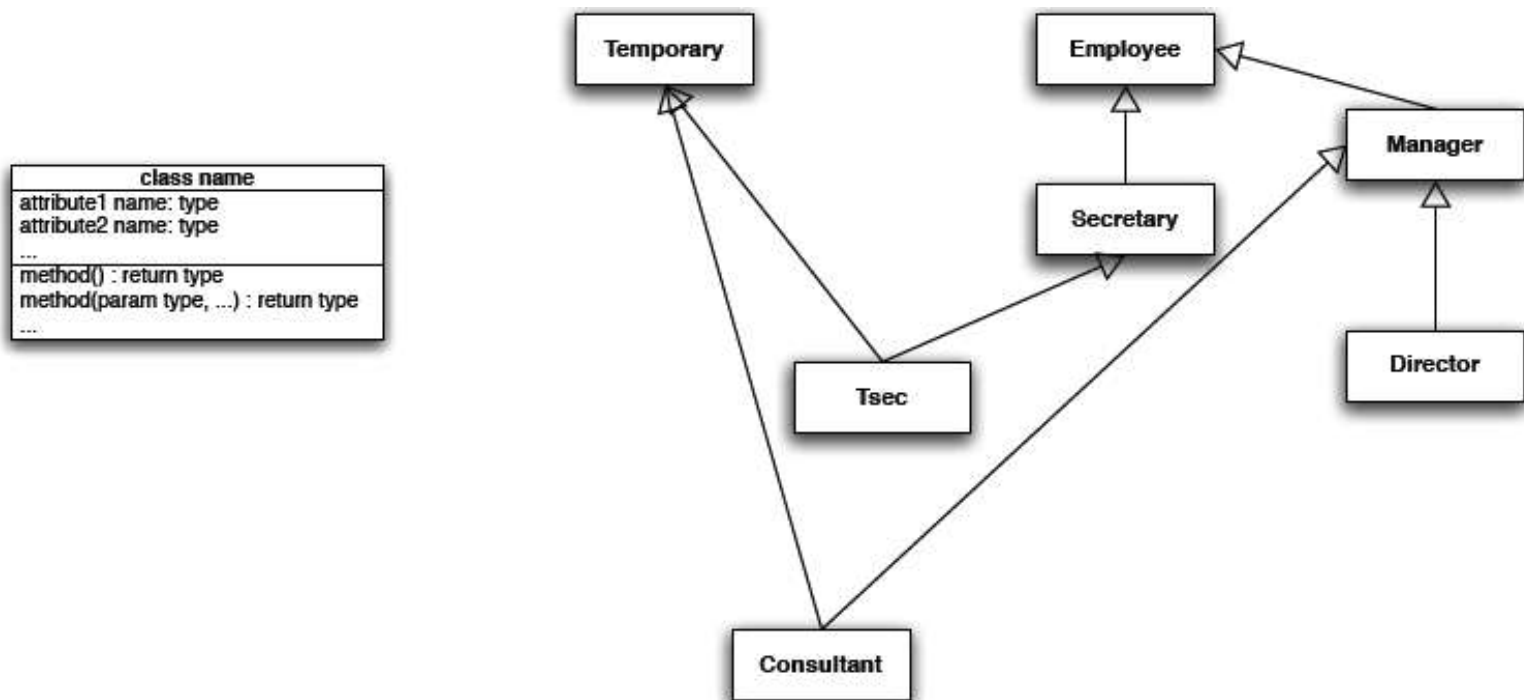
- Proprio come per i costruttori, solo che l'ordine viene invertito! È un processo dalla cima verso il basso(top-down)
- Quando una classe derivata viene distrutta, il costruttore di classe derivata(se c'è) sarà invocato prima e poi sarà invocato il distruttore della classe base(se c'è).  
(per prima cosa viene invocato il distruttore della derivata dopodichè sarà invocato il distruttore della classe base).
- I distruttori non sono sovraccaricati o invocati esplicitamente così non avremo la confusione su quale distruttore viene invocato!

## Multiple inheritance

- Una classe può derivare da diverse classi base
- Riporta semplicemente tutte le classi base dopo il “:”, e stabilisci il livello di accesso, ad esempio:

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
class Temporary { /* ... */ }; //provvisorio
class Secretary : public Employee { /* ... */ };
class Tsec : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

## A bit of UML class diagram



# Polymorphism

- Una classe derivata può fare l'*override* di un metodo ereditato da una classe base
  - La classe dovrebbe semplicemente includere una dichiarazione del metodo(e fornire un'implementazione)
  - Il metodo interessato dall'override spesso aggiunge alcune funzionalità in accordo con specializzazione della classe derivata(may upcall the base method)
- Il metodo è polimorfo perchè ha una differente implementazione che dipende da se esso è invocato nella classe base o nella classe derivata.

## Override vs. overload

- Il metodo dell'overload: significa dare lo stesso nome a un metodo ma con diversi parametri(quindi nella stessa classe)
- Metodo dell'override: stesso nome e stessi parametri in una gerarchia di classe

## Late binding

- La caratteristica dell'override permette che differenti implementazioni di un metodo esistano: ciò introduce il prolema di legare(il binding) l'invocazione di un metodo a una particolare implementazione:
- La decisione si basa sul tipo di classe usata per riferirsi a un metodo:
  - `<var>.op()` usa l'`op()` della classe della `<var>`
  - `<addr_expr>->op()` usa l'`op()` della classe dell'`<addr_expr>` che può essere differente dalla classe dell'oggetto inizializzato

Ad esempio:

```
class Base {
public:
 Base();
 virtual ~Base();
 void foo() { std::cout << "Base::foo" << std::endl;};
 int foo2() { std::cout << "Base::foo2" << std::endl;
 return -1;};
};

class Derived1: public Base {
public:
 Derived1();
 virtual ~Derived1();

 void foo() {
 Base::foo(); // upcall
 std::cout << "Derived1::foo" << std::endl;};

 int foo2() {std::cout << "Derived1::foo2()" << std::endl;
 return 1;};
};
```

```
Base *pBase; //nonostante ci metta un indirizzo a classe derivata
Derived1 aD1; //il puntatore è a classe base... quindi se chiamato..
 // ottieni un metodo della classe base..
```

```

cout << "pBase = &D1" << endl;
pBase = &aD1; // Base pointer to derived class
pBase->foo(); // Base::foo() because of static bind
cout << "D1::foo()" << endl;
aD1.foo(); // Derived1::foo()

// cast to call the method of derived class
((Derived1 *)pBase)->foo2(); // Derived1::foo2()

// se fai il cast del puntatore, ovvero lo trasformi in un puntatore a
// classe derivata invece quando vai a chiamare i metodi ottieni i metodi
// della classe derivata.

```

## Virtual methods

- I metodi virtuali evitano il bisogno per un cliente di una classe di sapere il tipo concreto della istanza che è usata
  - Nell'esempio precedente abbiamo dovuto convertire un puntatore a base per usare un metodo overridden nella classe derivata.
- Uno o più metodi di una classe derivata può essere dichiarato virtual aggiungendo la keyword (appunto virtual) nella dichiarazione.
- Un **metodo virtuale nella classe base rimane virtuale nelle classi derivate** (anche se la dichiarazione virtuale non è espressamente usata).
- La dichiarazione virtuale modifica il legame (binding): l'implementazione che è usata è sempre quella della classe istanziata.

Ad esempio

```

class Base {
public:
 Base();
 virtual ~Base();
 void foo() {std::cout << "Base::foo" << std::endl;};
 virtual int bar(int i)
 {std::cout << "Base::bar" << i << std::endl;
 return (i);};
};

class Derived1: public Base {
public:
 Derived1();
 virtual ~Derived1();
 void foo() {Base::foo(); // upcall
 std::cout << "Derived1::foo" << std::endl;};
 virtual int bar(int i)
 {std::cout << "Derived1::bar" << i << std::endl;
 return (i+1);};
};

Base *pBase;
Derived1 aD1;
cout << "pBase = &D1" << endl;
pBase = &aD1; // Base pointer to derived class
pBase->foo(); // Base::foo()
cout << "D1::foo()" << endl;
aD1.foo(); // Derived1::foo(), NO need to cast the pointer: it's a virtual
pBase->bar(1); // Derived1::bar() method

```

# Why virtual methods ?

- L'uso dei metodi virtuali riduce di molto l'accoppiamento tra un cliente e la gerarchia delle classi sviluppato dalla classe base.
  - Un puntatore di un tipo di classe base non necessita di sapere che tipo stia puntando: il legame ritardato(late binding) risolverà il problema!
- I **metodi virtuali** sono l'**attrezzatura chiave per il polimorfismo**: la funzione che è invocata usando un puntatore a classe base(o un riferimento) può avere diverse forme, a seconda del momentaneo tipo di oggetto che si sta usando.

## Rules for Virtual Functions

- Una funzione virtuale deve essere precisata virtual nella classe base
- Una funzione in una classe derivata con la stessa signature di una funzione virtuale nella classe base sarà virtuale anche se non viene precisata tale(virtuale). Comunque sia precisalo ogni volta.
- Una definizione separata(cioè non all'interno della dichiarazione della classe) di una funzione non è considerata virtuale.
- Le funzioni top-level non possono essere virtuali. Non avrebbe alcun senso..
- Le funzioni statiche(marked static) della classe non possono essere virtuali. Non avrebbe senso..

## Constructors and Destructors

- I **costruttori NON possono essere virtuali**: un costruttore è invocato su un tipo esplicito, non c'è bisogno di polimorfismo per essere considerati.
- I **distruttori** possono essere virtuali. Farli **virtuali assicura** che **siano chiamati i corretti distruttori** se l'oggetto è identificato da un riferimento a classe base o da un suo puntatore.
  - Nota che il wizard dell'Eclipse genera sempre distruttori virtuali!

## Virtual destructors

- Ricorda di dichiarare i distruttori virtuali in classi base polimorfiche(cioè in quelle che hanno almeno un metodo virtuale)

```
class TimeKeeper {
public:
 TimeKeeper();
 ~TimeKeeper(); ←
 virtual getCurrentTime();
 ..};

class AtomicTimeKeeper :public TimeKeeper {...};
class WristWatch : public AtomicTimeKeeper {...};
```

```
• TimeKeeper* getTimeKeeper();
...
TimeKeeper* ptk = getTimeKeeper(); // get it
... // use it
delete ptk; // release it
```

The derived part of the object will not be released leaking resources

Solve the issue declaring a virtual destructor

- Linee guida: se una classe non contiene un metodo virtuale allora probabilmente non è fatta per essere una classe base (o è una classe base non utilizzabile polimorficamente)
- Linee Guida: non è utile dichiarare un distruttore virtuale se non ci sono altri metodi virtuali nella classe:
  - Sprechiamo memoria per la creazione della tavola virtuale usata per gestire le funzioni virtuali.
- Cosa succede se derivi da una classe senza alcun distruttore virtuale?

```
class SpecialString : public std::string {...}; // std::string
 // has no virtual
 // destructor

SpecialString* pss = new SpecialString("Problems are coming");
std::string* ps;
...
ps = pss; // SpecialString is_a std::string
...
delete ps; // Ouch! We use the std::string destructor, any
 // resource managed by SpecialString is leaked
```

## Factory

- Un modo per esplorare a fondo il polimorfismo ottenuto usando i metodi virtuali è tramite l'uso di una classe factory che istanzi oggetti che dipendono da certe condizioni, ad esempio:

```
class Factory {
public:
 Base* getInstance();
 ...}

Base* Factory::getInstance() {
if (...)
return new Base;
else
return new Derived;}

int main() {
Base* pBase;
Factory *pFactory;
...
pBase = pFactory->getInstance();
...
pBase->aVirtualMethod();
...
}
```

## Name hiding

- Se una **classe base** dichiara una **funzione** membro e una **classe derivata** dichiara una **funzione** membro con lo **stesso nome** ma con **differenti tipi** di **parametri** e/o costranti, **allora il metodo base è nascosto** (hidden) piuttosto che overloaded o overridden (anche se il metodo è virtuale)

Ad esempio:

```
class Base {
 public:
 void f(double x); // doesn't matter whether or not this is virtual
};

class Derived : public Base {
 public:
 void f(char c); // doesn't matter whether or not this is virtual
};

int main() {

 Derived* d = new Derived();
 Base* b = d;
 b->f(65.3); // okay: passes 65.3 to f(double x)
 d->f(65.3); // bizarre: converts 65.3 to a char ('A' if ASCII)
 // and passes it to f(char c); does NOT call f(double x)!!

 delete d;
 return 0;
}
```

**Solution:**

```
class Derived : public Base {
 public:
 using Base::f; // This un-hides Base::f(double x)
 void f(char c);
};

or otherwise:
class Derived : public Base {
 public:
 // a redefinition that simply calls Base::f(double x)
 void f(double x) { Base::f(x); }
 void f(char c);
};
```

- La razionalità di tale comportamenti è ciò che evita che ci sia un sovraccarico ereditato accidentalmente da una classe base distante, quando crei una nuova classe, ad esempio in una libreria

- se hai bisogno di quegli overloads usa la dichiarazione *using* che hai visto prima.
- è qualcosa di simile al name hiding delle variabili:

```
double x;
void someFunc() {
 int x; // hides the global variable
 ...
}
```

- Il Name hiding e l'ereditarietà pubblica non stanno bene insieme:

ricorda che un oggetto Derivato *is\_a* oggetto Base, ma il nascondere(hiding) non lo rende necessariamente vero!

- Se erediti pubblicamente da una classe e ridefinisci un metodo forse dovresti aver dichiarato il metodo come virtual, accedendo alla classe derivata attraverso un puntatore a classe base, possiamo chiamare un metodo della classe base invece di ridefinirne uno.

```
class B {
public:
 void mf();
 ...};

Class D: public B {
public:
 void mf(); // hides B::mf()
 ...};

D x;
B* pB = &x;
D* pD = &x;
pD->mf(); // calls D::mf()
pB->mf(); // calls B::mf(), should have been virtual to call D::mf()
```

Questa ereditarietà pubblica non si comporta come una relazione di is\_a! Questo name hiding non è un buona idea. (This name hiding is bad design !)

## Fragile base class

- I linguaggi come C++ e Java soffrono di un problem anche è conosciuto come fragili per le classi base (fragile base classes). **Le classi base sono definite fragili quando l'aggiungere nuovi elementi ad una classe base conduce (l'aggiungere) a rompere (breaking) le classi derivate esistenti.**
- Quando l'aggiungere un metodo nuovo virtuale ad una classe base, i metodi esistenti con uno stesso nome in classi derivate faranno l'override sul nuovo metodo automaticamente. Se le semantiche del nuovo metodo non vanno bene con il metodo esistente nella classe derivata, la qual cosa quasi certamente non accadrà allora emerge il problema. Questo problema accade perché in C++ e Java l'utilizzatore non può specificare il loro intento per quanto riguarda l'overriding, così l'overriding accade in modo silenzioso per default.

## RTTI

Run-time type identification

## Why RTTI ?

- Una volta che abbiamo ottenuto un puntatore a un oggetto, è possibile usarlo per invocare una funzione polimorfica senza necessariamente sapere il tipo dell'oggetto.
  - **Il C++ late binding assicurerà che la funzione corretta (virtual) sia chiamata in accordo con il tipo momentaneo dell'oggetto.**
- Ma che accade se ci sono operazioni che sono uniche a un particolare tipo? Ma se noi abbiamo il tipo sbagliato allora non serve a nulla invocare una funzione che non esiste! Una possibile soluzione a tale problema è essere in grado di determinare esplicitamente il tipo degli oggetti puntati in ogni momento.

## How RTTI works



- Abbiamo un puntatore a classe base, possiamo allora trasformarlo in un puntatore a una specifica classe derivata e poi controllare di vedere se il cast funziona o meno.
  - Se l'oggetto attuale è del tipo desiderato, allora il cast può funzionare, se non lo è allora fallirà. Un tale cast è detto cast dinamico(dynamic cast).
- Usiamo il cast dinamico per tentare di trasformare un puntatore a classe base in un puntatore a classe derivata.

## C++ dynamic\_cast

- Il cast dinamico è usato per controllare in qualunque momento se un cast type è sicuro.(...)
- **Non è legale su un tipo polimorfico, cioè una classe che ha almeno un metodo virtual. Più specificatamente:**
  - Il source type (tra parentesi tonde) deve essere un puntatore o un riferimento a un tipo polimorfico.
  - Il target type (tra parentesi angolari) deve essere un puntatore o un riferimento, ma non necessariamente deve essere polimorfico.
- Stiamo lavorando sui puntatori, perciò un fallimento risulta in un puntatore a 0(cotrolla sempre se abbiamo uno 0 come risultato!)

## dynamic\_cast example

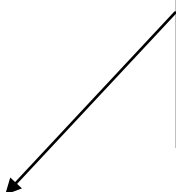
```
class B {
public:
 virtual void f() {...}
};
```

```
class D1 : public B {
public:
 virtual void f() {...}
 void D1specific() {...}
};
```

```
class D2 : public B {
public:
 ...
};
```

```
B* bp;
D1* dp;
bp = new D1;
dp = dynamic_cast<D1>(bp);
if (dp != 0) {dp->D1specific();}
bp = new D2;
dp = dynamic_cast<D1>(bp);
if (dp != 0) {dp->D1specific();}
```

More realistically: when using a Factory to get the instances, we do not know what is the real type of the object



## dynamic\_cast to reference

- Se utilizziamo il dynamic\_cast per referenziare non possiamo verificare uno 0, perchè un riferimento deve sempre essere valido

- Il C++ ha un meccanismo diverso di gestione degli errori lo vedremo in una lettura successiva: eccezioni:

```
try {
T& tref = dynamic_cast<T&>(xref); } catch (bad_cast) {
// ...
}
```

## typeid

- L'operatore di typeid restituisce un'identificazione del tipo di un tipo base, una classe, una variabile o una qualsiasi espressione.

Può essere utile per (to store objects to file, recording the type of each object).

- Necessita della direttiva al preprocessore: `#include<typeinfo>`.
- Il typeid in realtà restituisce un riferimento a un oggetto nella classe `system type_info`.
- Non hai bisogno di sapere i dettagli interni, ad esempio per verificare se una variabile è un particolare tipo:

```
if(typeid(x) == typeid(float)) { // ... }
```

## Multiple inheritance

- È più complessa di una ereditarietà singola: la gerarchia dell'ereditarietà non è più semplicemente una gerarchia ad albero, ma piuttosto diventa una rete (o un grafico).
- C'è una relazione di IS\_A tra una classe derivata e le sue classi base, ad esempio: Un tutor IS A studente e un tutor IS A impiegato provvisorio

## Multiple Inheritance Rules

- Non ci sono effettivi cambiamenti dalla ereditarietà singola a quella multipla.
- La classe derivata eredita tutti i dati membro e i metodi dalle classi base.
- I membri protetti delle classi base possono essere accessibili dalla classe derivata, come prima.
- Conflitti tra i nomi possono risultare nei membri delle classi base pur avendo lo stesso nome (risolvi tale problema attraverso l'uso di dichiarazioni o tramite la qualificazione completa dei nomi).
- I costruttori di ogni classe base (se c'è) saranno similmente invocati prima dei costruttori delle classi derivate (se ci sono). I distruttori faranno lo stesso ma con l'ordine opposto.

## Multiple Inheritance characteristics

- I costruttori Base sono chiamati nell'ordine della dichiarazione della classe, ad esempio:

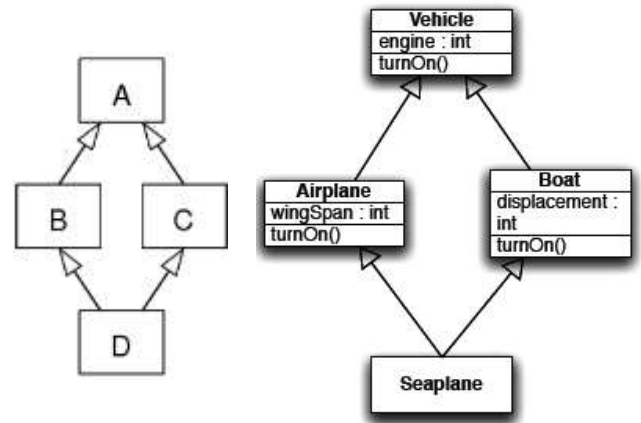
```
class Bat : public Mammal, public Winged
{ ... Bat(); // the Mammal() c'tor is called before Winged() }
```

- Elimina le ambiguità usando lo scope resolution, ad esempio:

```
Bat aBat;
aBat.Mammal::eat(); // if both Mammal and Winged
// have a eat() method
```

## Diamond problem

- Il problema del diamante è una ambiguità che sorge con l'ereditarietà multipla quando due classi B e C ereditano da A e la classe D eredita sia da B che da C.
- Il risultato sarà la replica della classe base nella classe derivata che usa l'ereditarietà multipla.
- **Se un metodo in D chiama un metodo definito in A (senza l'override del metodo), e B e C hanno overridden tale metodo diversamente, allora da quale classe eredita: dalla B o dalla C?**



## Virtual inheritance

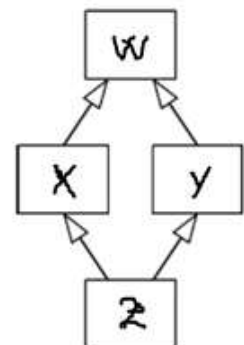
- L'ereditarietà virtual è un tipo di ereditarietà che risolve alcuni dei problemi causati dall'ereditarietà multipla (in particolare il problema del “diamante”) chiarificando l'ambiguità circa quali metodi della classe antenata usare.
- Una classe base ereditata multiplamente è denotata virtual con la keyword virtual.

## Pointer conversions

- Le conversioni (sia esplicite che implicite) da un puntatore a classe derivata o un riferimento a classe base o un riferimento deve riferirsi inequivocabilmente allo stesso oggetto accessibile della classe base, ad esempio:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };

int main () {
 Z z;
 X* pX = &z; // valid
 Y* pY = &z; // valid
 W* pW = &z; // error, ambiguous reference to class W
}
```



## Abstract classes

## Why abstract classes ?

- Ci sono molte situazioni per cui la classe base in una gerarchia di classe dovrebbe essere una classe astratta, cioè, **nessun oggetto può essere istanziato da essa**.
  - **Include dichiarazioni speciali di metodi virtuali ma non la loro implementazione**
- Una classe astratta è una base da cui definire altre classi concrete.
- Una classe puramente astratta non ha l'implementazione dei suoi metodi.
- Un client può contare su l'interfaccia fornita da una classe astratta senza il bisogno di sapere i dettagli sulle classi che la implementano.
  - È una tecnica che disaccoppia gli oggetti, specialmente quando considerando classi puramente astratte che NON tengono conto dell'ereditarietà dell'implementazione ma permettono il meccanismo della sostituzione.

## Abstract classes: how

- Una classe base astratta è una classe che ha almeno una funzione puramente virtual.
- Una funzione puramente virtuale è dichiarata usando la sintassi speciale:  

```
virtual void Method1() = 0;
```
- La precedente funzione (sopra) non ha bisogno di essere definita, poiché non esisterà in realtà e non sarà mai chiamata!
- Una classe derivata da una classe base astratta deve fare l'override di tutte le sue funzioni puramente virtuali o anch'essa diventerà una classe base astratta.

## Class Hierarchy example

```
class Vehicle {
public:
 virtual intNumWheels() const = 0;
};

class MotorCycle: public Vehicle {
public:
 virtual intNumWheels() const { return 2; };
};

class Car : public Vehicle {
public:
 virtual intNumWheels() const {return 4; };
};

class Truck : public Vehicle {

public:
 Truck(int w = 10) : wheels(w) {}
 virtual intNumWheels() const {return wheels;}

private:
 int wheels;
};
```

## Pure virtual destructor

- Se vuoi creare una classe base astratta ma non hai metodi che siano puramente virtuali dichiara il distruttore come puramente virtuale!

Osserva il trucco:

```
class AWOV { // Abstract W/O Virtuals
public:
 virtual ~AWOV() = 0;
 ...
};

AWOV::~~AWOV() {} // REMIND: you HAVE to define the
 // pure virtual destructor !
```

We have declared pure virtual but the compiler needs a destructor that is called when it reaches the base class. Forget it and the linker will complain.

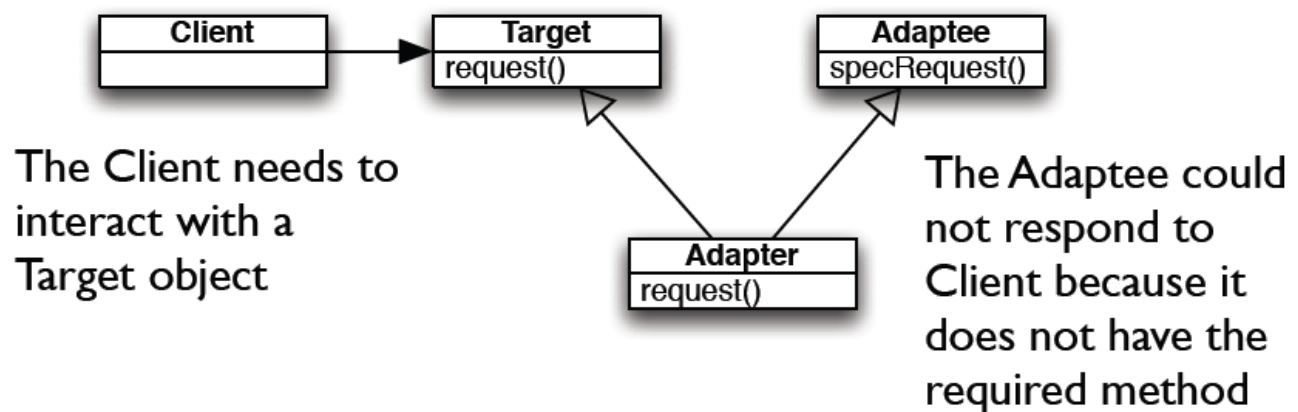
# Class adapter

I metodi virtuali, ereditarietà privata, le classi astratte e l'ereditarietà multipla, tutte insieme.

## Use of multiple inheritance

- Nell'esempio che segue viene mostrato un interessante uso di ereditarietà multipla, insieme con classi astratte, metodi virtuali e ereditarietà privata.
- Una classe(adapter) adatta l'interfaccia di un'altra classe(adaptee) al cliente, usando l'interfaccia descritta che ci si aspetta in una classe astratta(target)
  - Questa è il modello della cosiddetta "class adapter" (pattern): permette di lavorare insieme alle classi che altrimenti non potrebbero per via delle interfacce compatibili.

## "Class Adapter" UML class Diagram



The Adapter lets the Adaptee to respond to request of a Target by extending both Target and Adaptee

- 1) Il client ha bisogno di interagire con un oggetto target
- 2) L'adapter permette all'adaptee di rispondere alla richiesta di un target estendendo sia il target che l'adaptee.
- 3) L'adaptee non potrebbe rispondere al client perchè non ha il metodo richiesto.

## Class Adapter example

```
class Adaptee {
 public:
 getAlpha() {return alpha;};
 getRadius() {return radius;};
 private:
 float alpha;
 float radius;
};
```

```
class Target {
 public:
```

```
 virtual float getX() = 0;
 virtual float getY() = 0;
};
```

```
class Adapter : private Adaptee, public Target{
public:
 virtual float getX();
 virtual float getY();
};
```

```
float Adapter::getX() {return Adaptee::getRadius()*cos
(Adaptee::getAlpha());}
```

```
float Adapter::getY() {return Adaptee::getRadius()*sin
(Adaptee::getAlpha());}
```

**Il cliente non può accedere ai metodi dell'adaptee perchè l'adapter li ha ottenuti usando un'ereditarietà privata.**

# Generic programming

## What is generic programming ?

- La programmazione generica è uno stile di programmazione in cui gli algoritmi sono scritti in termini di tipi che si specificheranno poi, essi sono poi istanziati quando necessari per specifici tipi che sono forniti come parametri.
- La programmazione generica fa riferimento a caratteristiche di certi linguaggi di programmazione statici, che permettono a del codice di evitare effettivamente la necessità dei tipi statici, ad esempio, in C++, un **template** è una **routine** in cui **alcuni parametri sono qualificati da un tipo variabile**. Poiché la generazione del codice in C++ dipende dai tipi concreti, i template sono specializzati per ogni combinazione di tipi di parametri che occorrono(occur) in pratica.

## Generic programming

- Gli algoritmi sono scritti indipendentemente dai dati
    - I tipi di dati sono riempiti durante l'esecuzione
    - Le funzioni o classi istanziate con i tipi di dati
    - Formalmente conosciuta come specializzazione
  - Un numero di algoritmi sono indipendenti dai tipi di dato.
- Ad esempio il sorting, searching, finding n-th largest, swapping, etc.

## Identical tasks for different data types

- **Approcci per funzioni che implementano identici compiti per differenti tipi di dati.**
  - Naïve Approach
  - Function Overloading
  - Function Template

### • Naïve approach

- Creare funzioni univoche con nomi univoci per ogni combinazione di tipi di dati.  
(e.g. atoi(), atof(), atol() )
  - Difficile tener d'occhio i nomi molteplici delle funzioni.
  - Conduce ad errori di programmazione.

### • Function overloading

- L'uso di stessi nomi per differenti funzioni C++, distinguendole tra loro attraverso la loro lista di parametri.
  - Elimina la necessità di avanzare con molti nomi differenti che compiono stessi compiti(Naïve).
  - Riduce la possibilità di conseguenze inattese causate da un errato uso del nome della funzione.
  - La **duplicazione del codice rimane**: è necessario scrivere il codice per ogni singola funzione per(fatta per) ogni singolo tipo.

### • Function template

- Un costrutto del linguaggio C++ che permette al compilatore di generare versioni multiple di una funzione permettendo di parametrizzare i tipi di dato.
  - Una definizione di funzione (una funzione template).

- Il compilatore genera funzioni individuali.

# Generic programming & templates

## Why templates ?

• I templates C++ sono usati principalmente per classi e funzioni che possono essere applicate a tipi differenti di dati.

- Esempi comuni sono i container STL classi e algoritmi.

Ad esempio: algoritmi come l'ordinamento sono programmati per essere in grado di ordinare quasi qualunque tipo di oggetto.

## Generic programming in C++

- Templates = generic programming
- Due tipi:
  - **function templates**: funzioni speciali che possono operare su tipi generici.
  - **class templates**: possono avere membri che usano parametri templates come tipi.

## Templates & Inheritance

- L'ereditarietà opera tutt'uno con i Template e supporta:
  - 1) Una classe template basata su una classe template
  - 2) Una classe template basata su una classe non template
  - 3) Una classe non template basata su una classe template

## Coding C++ templates

- Un template inizia con l'intestazione:  
`template<class T>`
- Il tag T è utilizzato ovunque che un tipo base sia necessario. Usa qualunque lettera o combinazione di lettere che preferisci.
- In generale, le classi template (e le funzioni) possono avere "tipi" di parametri multipli e può anche avere parametri "senza tipo".
- Le funzioni membro separate sono un template separato.

## Function Templates

- Funzioni speciali utilizzando i tipi template.
- Un parametro template è un tipo speciale di parametro usato per passare un tipo come parametro(argument).
- Semplicemente come parametri di funzioni regolari, ma passano i tipi alla funzione.  
(just like regular function parameters, but pass types to a function)

## Function templates declaration format

- `template <class identifier>`



function\_ declaration;

- template <typename identifier>

function\_ declaration;

- Stesse funzionalità, differenti parole chiave.
  - use <typename ...>
  - I compilatori più vecchi usavano soltanto il formato <class...>
- template keyword deve apparire dopo una funzione o classe template.
- ...seguita dalla lista di tipi generic o template
- Puoi avere più di un tipo generic.
- ... seguita da la dichiarazione di funzione e/o dalla definizione.

## Function template example

Funzione che prende 2 myType e restituisce un myType(il maggiore tra i due)

- template <typename myType>

```
myType getMax (myType a, myType b){
 return (a>b?a:b);
}
```

- o anche meglio, utilizzando riferimenti e il const-ness:

- template <typename myType>

```
const myType& getMax (const myType& a, const myType& b) {
 return (a>b?a:b);
}
```

## Function template usage example

```
int main() {
 int i=5, j=6, k;
 long l=10,m=5,n;
 k=getMax<int>(i,j); // OK
 n=getMax<long>(l,m); //OK
 cout << k << endl;
 cout << n << endl;
 return 0;
}
```

```
int main() {
 int i=5, j=6, k;
 long l=10,m=5,n;
 k=getMax(i,j); // OK
 n=getMax(l,m); // OK
 cout << k << endl;
 cout << n << endl;
 k=getMax(i,l); // Wrong: can't mix types !
 return 0;
}
```

## Multiple template types

- Definisci tutti i tipi template necessari dopo la template keyword, ad esempio:
- `template <typename T1, typename T2>`  
`const T1& getMax (T1& a, T2& b) {`  
`return (a>(T1)b?a:(T1)b);`  
`}`

# Instantiating a function

## Template

- Quando il compilatore istanzia un template, sostituisce i parametri(argument) del template per i parametri del template attraverso la funzione template.  
 (When the compiler instantiates a template, it substitutes the template argument for the template parameter throughout the function template)
- Template function call(chiamata):  
`Function <TemplateArgList> (FunctionArgList)`

## Function template specialization example

- La specializzazione della funzione template permette di specializzare il codice per certi specifici tipi, ad esempio:
- `template<typename T> inline std::string stringify(const T& x){`  
`std::ostringstream out;`  
`out << x;`  
`return out.str();`  
`}`
- `template<> inline std::string stringify<bool>(const bool& x) {`  
`std::ostringstream out;`  
`out << std::boolalpha << x;`  
`return out.str();`  
`}`

## Class templates

- Le classi possono avere membri che usano i parametri template come tipi, ad esempio, una classe che salva due elementi di un qualunque tipo valido:
- `template <class T>`  
`class mypair {`  
`private:`  
`T values [2];`  
`public:`  
`mypair (T first, T second) {`  
`values[0]=first; values[1]=second;`  
`}`  
`};`

# Class template: non-inline definition

- Per definire una funzione membro fuori la dichiarazione della classe template, fai sempre precedere quella definizione con il prefisso `template <...> :`

```
template <class T>
```

```
class mypair {
private:
 T values [2];
public:
 mypair(T first, T second) {
 values[0]=first;
 values[1]=second;}
```

```
 T getMax(); };
```

```
template <class T>
T mypair<T>::getMax() {
 T retval;
 retval = (a>b? a : b);
 return retval;
}

int main () {
 mypair<int> myobject (100, 75);
 cout << myobject.getMax();
 return 0;
}
```

## Templates and polymorphism

- Nella programmazione OO la scelta del metodo che deve essere invocato su un oggetto può essere selezionato in runtime(a runtime), ad esempio, il polimorfismo con metodi virtuali(virtualmethods).
- Nella programmazione generica è scelto al tempo di compilazione, quando si istanzia un template.

## C++ templates: so many Ts !

- Ci sono tre T in questa dichiarazione del metodo (quale metodo?):
  - la prima è il parametro template.
  - la seconda T si riferisce al tipo restituito dalla funzione
  - la terza T (quella tra <> in grassetto) specifica che questo parametro template della funzione è anche un parametro della classe template.

Esempio scritto da me: `T mypair<T>::getMax() {T retval; /* ... */ }`

## Default parameters

- Le classi template possono avere tipi di parametri(arguments) di default.

- Come con i parametri (arguments) di default di una funzione, il tipo di default di un template dà al programmatore molta più flessibilità nello scegliere il tipo ottimale per una particolare applicazione. Ad esempio:

```
template <class T, class S = size_t > class Vector { /*..*/};
Vector <int> ordinary; //second argument is size_t
Vector <int, unsigned char> tiny(5);
```

## Non-type parameters

- I template possono anche avere scritti(typed) parametri, simili a quelli che trovavamo nella funzioni

- template <class T, int N>

```
class mysequence {
T memblock [N];
public:
void setMember (int x, T value);
T getMember (int x);
};
```

- template <class T, int N>

```
T mysequence<T,N>::getMember (int x) {
return memblock[x];
}
```

```
int main () {
mysequence <int,5> myints;
mysequence <double,5> myfloats;
myints.setMember (0,100);
myfloats.setMember (3,3.1416);
cout << myints.getMember(0) << '\n';
cout << myfloats.getMember(3) << '\n';
return 0;
}
```

## Class template specialization

- È utilizzata per definire una differente implementazione per un template quando un tipo specifico è passato come un parametro template.
- Dichiara esplicitamente una specializzazione di un template, ad esempio: una classe con una sorte di metodo che ordina gli interi, i char, i double, i float e che necessita anche di ordinare le stringhe basandosi sulla lunghezza(della stringa), ma l'algoritmo è differente(non un ordinamento lexicographico).
- È necessario creare esplicitamente una specializzazione per il metodo di ordinamento quando una stringa è passata come tipo.

## Class template specialization example

```
template <class T>
class MyContainer {
private:
```

```

T element[100];
public:
MyContainer(T* arg) {...};
void sort() { // sorting algorithm}
};

// class template specialization:
template <>
class MyContainer <string> {
string element[100];
public:
MyContainer (string *arg) {...};
void sort() {
// use a string-length
// based sort here
}
};

```

## Instantiating a class template

- I parametri(arguments) della classe template devono essere espliciti.
- Il compilatore genera distinti tipi di classi chiamate classi template o classi generate(generated classes).
- Instanziando un template, un compilatore sostituisce il parametro(argument) template con il parametro template attraverso la classe template.
  - Per efficienza, il compilatore usa una politica di “istanza su richiesta”( “instantiate on demand”) solamente di quei metodi che sono richiesti.
- Per creare liste di differenti tipi di dati da una GList template class:

```

// Client code
GList<int> list1;
GList<float> list2;
GList<string> list3;
list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");

```



The compiler generates 3 distinct class types:

```

GList_int list1;
GList_float list2;
GList_string list3;

```

## Class template

## A complete example

```

1 // Fig. 22.3: tstack1.h
2 // Class template Stack
3 #ifndef TSTACK1 H
4 #define TSTACK1 H
5
6 template< class T >
7 class Stack {
8 public:
9 Stack(int = 10); // default constructor (stack size 10)
10 ~Stack() { delete [] stackPtr; } // destructor
11 bool push(const T&); // push an element onto the stack
12 bool pop(T&); // pop an element off the stack
13 private:
14 int size; // # of elements in the stack
15 int top; // location of the top element
16 T *stackPtr; // pointer to the stack
17
18 bool isEmpty() const { return top == -1; } // utility
19 bool isFull() const { return top == size - 1; } // functions
20 };
21
22 // Constructor with default size 10
23 template< class T >
24 Stack< T >::Stack(int s)
25 {
26 size = s > 0 ? s : 10;
27 top = -1; // Stack is initially empty
28 stackPtr = new T[size]; // allocate space for elements
29 }

```

- Class template definition
- Function definitions
- Stack constructor
- push
- pop

```

30
31 // Push an element onto the stack
32 // return 1 if successful, 0 otherwise
33 template< class T >
34 bool Stack< T >::push(const T &pushValue)
35 {
36 if (!isFull()) {
37 stackPtr[++top] = pushValue; // place item in Stack
38 return true; // push successful
39 }
40 return false; // push unsuccessful
41 }
42
43 // Pop an element off the stack
44 template< class T >
45 bool Stack< T >::pop(T &popValue)
46 {
47 if (!isEmpty()) {
48 popValue = stackPtr[top--]; // remove item from Stack
49 return true; // pop successful
50 }
51 return false; // pop unsuccessful
52 }
53
54 #endif
55 // Fig. 22.3: fig22_03.cpp
56 // Test driver for Stack template
57 #include <iostream>
58
59 using std::cout;
60 using std::cin;
61 using std::endl;
62
63 #include "tstack1.h"
64
65 int main()
66 {
67 Stack< double > doubleStack(5);
68 double f = 1.1;
69 cout << "Pushing elements onto doubleStack\n";
70
71 while (doubleStack.push(f)) { // success true returned
72 cout << f << ' ';
73 f += 1.1;
74 }
75
76 cout << "\nStack is full. Cannot push " << f
77 << "\n\nPopping elements from doubleStack\n";
78
79 while (doubleStack.pop(f)) // success true returned

```

- Class template definition
- Function definitions
- Stack constructor
- push
- pop
- Include header
- Initialize doubleStack
- Initialize variables
- Function calls
- Function calls
- Output

```

80 cout << f << ' ';
81
82 cout << "\nStack is empty. Cannot pop\n";
83
84 Stack< int > intStack;
85 int i = 1;
86 cout << "\nPushing elements onto intStack\n";
87
88 while (intStack.push(i)) { // success true returned
89 cout << i << ' ';
90 ++i;
91 }
92
93 cout << "\nStack is full. Cannot push " << i
94 << "\n\nPopping elements from intStack\n";
95
96 while (intStack.pop(i)) // success true returned
97 cout << i << ' ';
98
99 cout << "\nStack is empty. Cannot pop\n";
100 return 0;
101 }

```

- Include header
- Initialize doubleStack
- Initialize variables
- Function calls
- Function calls
- Output

## Static members and variables

- Ogni classe template o funzione generata da un template ha le sue proprie copie di ogni variabile static o membro static.
- Ogni istanza di una funzione template ha la sua propria copia di tutte le variabili static definite nella visibilità(lo scope) della funzione..

## Template constraints

- Le operazioni eseguite all'implementazione di un template costringono implicitamente i tipi di parametri; questa è chiamata "constraints through use":

template <typename T>

.. // some code within a class template..

.. T t1, t2; // implies existence of default ctor

.. t1 + t2 // and plus operator

..

- Il codice implica che + dovrebbe essere supportato dal tipo T:
  - vero per tutti quelli costruiti su tipi numerici.
  - Si può definire per user-defined types (classes)
- Se mancante, genera un errore al tempo di compilazione ed è riportata immediatamente dal compilatore.
- Un template è controllato(cheked) al momento della definizione.
- I parametri template dipendenti dal codice possono essere controllati solo quando il template diventa specifico alla sua istanziazione.  
(template parameter dependent code can be checked only when the template becomes specified at its instantiation)
- Il codice può funzionare per diversi tipi di parametri(argument), e fallire per diversi altri tipi di parametri(argument) al momento della compilazione.
- Le costrizioni implicite di una classe/funzione template sono richieste solo se il template viene inizializzato(al momento della compilazione).



- E tutti i template sono istanziati solo quanto veramente necessari: un oggetto è creato o la sua particolare operazione è chiamata.
- Nota che tutti i tipi di parametro non necessitano di soddisfare tutte le richieste date implicite dall'intera definizione del template – poiché solo alcune funzioni membro possono essere in realtà necessarie e chiamate per qualche codice (called for some code)

## C++ templates: source code organization

- Le classi template sono codificate nei file di header.
  - Molti pochi compilatori supportano la codifica delle funzioni template in un file sorgente separato e l'utilizzo della keyword `export` per renderli disponibili in altre unità di compilazione.
- Gran parte dei compilatori NON lo supportano, e inoltre i template possono essere visti principalmente come una sostituzione del testo.
- È possibile tenere l'implementazione di una funzione o dei metodi di una classe in un file di estensione `.cpp` e includere (con `include`) questo file nel file di header che contiene la loro dichiarazione. ( Il codice del cliente includerà lo header, the client code will include the header).

```
// header file compare.h
#ifndef _COMPARE_H_
#define _COMPARE_H_
template<class > int compare(const T& a, const & b);
// other declarations...
#include "compare.cpp" //contains implementation
#endif // _COMPARE_H_

// implementation file compare.cpp
template<class T> int compare(const T& a, const T& b) {
if(a < b) return -1;
if(b < a) return 1;
return 0;
}
// other definitions...
```

## Templates and base classes

- Ci possono essere alcune questioni con l'ereditarietà e i template, ad esempio se una classe base template è specializzata e la specializzazione non ha la stessa interfaccia del template generale (ricorda: nei templates le interfacce sono "implicite")

```
class CompanyA {
public:
void sendCleartext(const string& msg);
void sendEncrypted(const string& msg);
//...};

class CompanyZ {
public:
void sendEncrypted(const string& msg);
//...
};
```

This template does not work with CompanyZ: the sendClear requires a working sendCleartext that is not available !

```
class MsgInfo {...};
template<typename Company>
class MsgSender {
public:
//...
void sendClear(const MsgInfo&info)
{
string msg;
//create msg from info;
...
Company c;
c.sendCleartext(msg);
}
void sendSecret(const MsgInfo& info) { ... };
//...
};
template<typename Company>
class LoggingMsgSender : public
MsgSender<Company> {
public:
//...
void sendClearMsg(const MsgInfo& info) {
logMsg(...);
sendClear(info); // does NOT compile !
logMsg(...);
}
// ...
};
```

- Per risolvere il problema crea una versione specializzata di MsgSender, che non ha un metodo sendClear:

```
template<>
class MsgSender<CompanyZ> {
public:
//...
void sendSecret(const MsgInfo& msg) { //... }
//...};
```

- Tuttavia non è abbastanza: dobbiamo dire al compilatore di guardare alla classe base per controllare se l'interfaccia è implementata completamente:

- La classe base di prefazione si chiama con this ->:

```
void sendClearMsg(const MsgInfo& info)
{
logMsg(...);
this->sendClear(info); // OK: assumes that it will be inherited
logMsg(...);
}
```

- Usa una dichiarazione using, per forzare il compilatore a cercare la base class scope:

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
using MsgSender<Company>::sendClear; // OK: tell compilers it's in base class
```

# Code bloat

- Poichè i template sono gestiti tramite sostituzione testuale, più istanze dello stesso template con differenti tipi risulterà in più istanze del codice.
- Resi peggiori dalla richiesta comune di porre tutte le funzioni membro inline (risultante da copie multiple addizionali).
  - Ogni singola chiamata a una classe template o a una funzione membro di una classe template sarà inlined, risultando potenzialmente in numerose copie dello stesso codice.

## Reducing code bloat

- Dai un codice alla classes template con una classe wrapper che fa relativamente poco, ma può ereditare da una classe non template le cui funzioni membro possono poi essere codificate in un modulo compilato separatamente.
- Questa tecnica è utilizzata in STL per molti contenitori standard e algoritmi laddove la classe base non template ha a che fare con un un tipo generico di puntatore a void (void\* pointer type).

La classe template dà un'interfaccia sicura dal punto di vista dei tipi (type-safe) ad una classe base non sicura.

## Why use templates ?

- Aggiungi un controllo extra type per le funzioni che altrimenti prenderebbero puntatori a vuoto (void pointers): i template sono type-safe.
- Poiché i tipi su cui i template agiscono sono conosciuti al momento della compilazione, il compilatore può operare un controllo di tipo prima che accadano gli errori.
- Crea una classe di collezione type-safe (per esempio uno stack) che può operare su dati di qualsiasi tipo.
  - Crea solo una versione generic della tua classe o funzione invece di specializzazioni create manualmente.
  - Code understanding: I template possono essere compresi con facilità, poichè essi possono dare un modo immediato per astrarre un'informazione tipo (type information)

## Templates vs. Macros

- I C++ templates si assomigliano ma non sono macro:
- Il primo nome istanziato identifica la stessa istanza di classe generate in ogni luogo.
- Il compilatore tipicamente rappresenta la classe con qualche nome interno generato e pone l'istanziamento in qualche deposito interno per l'uso futuro.
- Qualsiasi nome libero (indipendente dal parametro) dentro un template è legato al punto della definizione del template.

```
#define min(i, j) (((i) < (j)) ? (i) : (j))
```

vs.

```
template<class T> T min (T i, T j) { return ((i < j) ? i : j) }
```

- Ecco alcuni problemi con i macros:
- Non c'è modo per il compilatore di verificare che i parametri macro sono del tipo compatibile. Il macro è espanso senza alcun controllo sui tipi speciale.
- I parametri i e j sono valutati due volte. Per esempio, se ciascun parametro ha una variabile incrementata dopo (esempio: min(i++,j)), l'incremento è operato due volte.

- Poichè i macros sono espansi per mezzo del preprocessore, i messaggi di errore del compilatore si riferiranno al macro espanso, piuttosto che alla definizione macro stessa. Poi, il macro mostrerà in una forma espansa durante il debug(debugging).

## Templates vs. void pointers

- Molte funzioni che ora sono implementate con i puntatori a vuoto, possono essere implementate con i template.

I puntatori a vuoto sono spesso usati per permettere alle funzioni di operare su dati di un tipo sconosciuto. Quando si **usano i puntatori a vuoto, il compilatore non riesce a distinguere i tipi, così non può operare il controllo sui tipi** o il comportamento specifico dei tipi come per esempio usando operatori specifici di tipo, operatori di overloading, o costruttori o distruttori.

- Con i template noi possiamo creare funzioni e classi che operano su tipi di dati(typed data). Il tipo sembra astratto nella definizione template. Comunque nel momento della compilazione il compilatore crea una versione separata della funzione per ciascun tipo specificato. Questo permette al compilatore di trattare le classi e le funzioni template come se agissero su tipi specifici. I template possono inoltre migliorare la chiarezza nella codificazione, poiché non c'è bisogno di creare casi speciali per tipi complessi come le strutture.

# STL

Standard Template Library

## STL history

- Negli ultimi anni 70 Alexander Stepanov per primo osservò che alcuni algoritmi non dipendono da alcuna particolare implementazione di una struttura dati, ma solamente da poche proprietà semantiche fondamentali della struttura.
- The Standard Template Library (STL) fu sviluppata da Alex Stepanov, originariamente implementata per Ada (80's - 90's)
- Nel 1997, STL fu accettata dal comitato ANSI/ISO C++ Standards come parte della library standard C++.
  - L'utilizzo dell'STL influenza pure ampiamente vari elementi del linguaggio del C++, soprattutto gli elementi offerti dai template.

## What is STL?

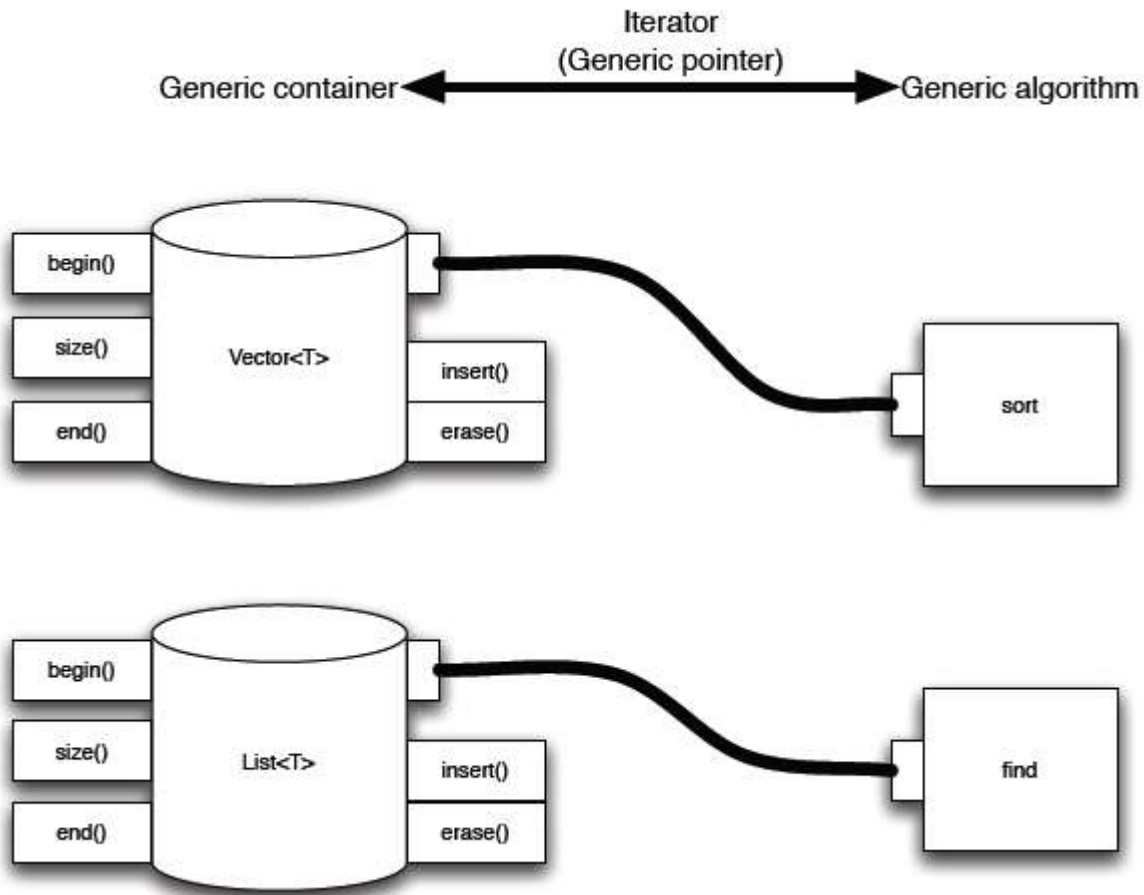
- È una libreria(un intento generale) di algoritmi generici e di strutture di dati; supporta tipi base di dati come vettori, liste, contenitori associativi(mappe,sets), e algoritmi come sorting e searching..
- Efficiente e compatibile con il modello di computazione C++
- Non object-oriented: molte operazioni(algoritmi) sono definite come funzioni che stanno da sole(stand alone)
- Usa i template per un'ulteriore riutilizzazione(reusability)

## Basic principles of STL

- I contenitori STL(raccolte) sono template di tipo parametrizzato(type-parameterized), piuttosto che classi con legami dinamici e di ereditarietà.
  - Non c'è una classe base comune per tutti i contenitori.
  - Non ci sono funzioni virtuali e late binding.
  - Comunque i containers implementano una (qualche) interfaccia uniforme di contenitore con operazioni simili.
- La stringa standard era definita(o doveva definirsi) indipendentemente ma più tardi estesa a coprire i servizi e le interfacce simili a STL.

## What's in STL

- STL (Standard Template Library) garantisce tre componenti base per supportare l'ADTs:
  1. I contenitori, per mantenere e possedere raccolte omogenee di valori; un contenitore stesso gestisce la memoria per i suoi elementi.
  2. Gli iteratori sono sintatticamente e semanticamente simili ai puntatori del C; containers differenti danno iteratori differenti(ma con interfacce simili).
  3. Gli algoritmi operano sui vari containers per mezzo degli iteratori; gli algoritmi prendono diverse specie di iteratori come (generici) parametri; per eseguire un algoritmo su un container l'algoritmo e il container devono supportare iteratori compatibili.



## STL example

```
#include <vector> // get std::vector
#include <algorithm> // get std::reverse, std::sort, etc.
//...
int main () {
 std::vector<double> v; // vector (STL container) for input data
 double d;
 while (std::cin >> d) // read elements using IO stream
 v.push_back(d); // method to append data to the vector
 if (!std::cin.eof ()) { // check how input failed
 std::cerr << "format error\n"; // IO stream used for error messages
 return 1; // error return
 }

 std::cout << "read " << v.size() << " elements\n"; // get size of
 // container
 std::reverse(v.begin(), v.end()); // STL algorithm (with two STL
 // iterators)
 std::cout << "elements in reverse order:\n";
 for (int i = 0; i < v.size (); ++i)
 std::cout << v [i] << '\n';
}
```

## Basic concepts of STL

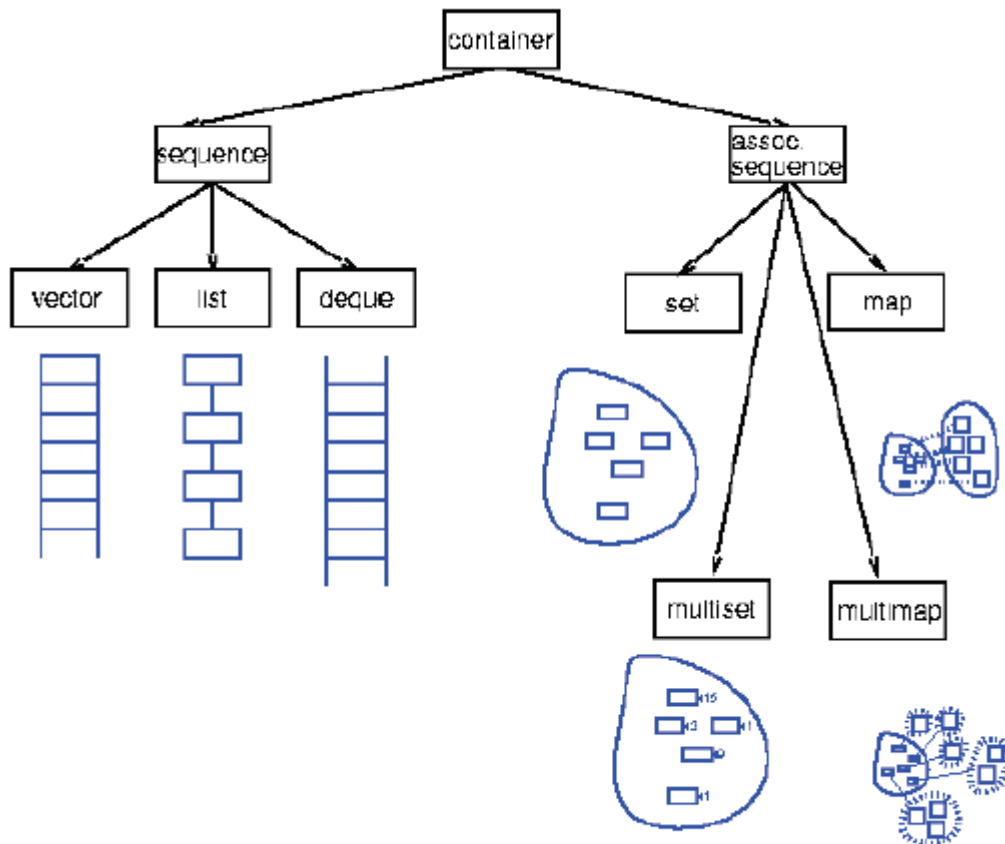
- I containers sono classi di template parametrizzati (parameterized class templates); essi cercano di fare le assunzioni minime circa il tipo di elementi che essi posseggono – hanno bisogno di alcune operazioni per esempio per copiare elementi, aggiungerne o rimuoverne...

- Gli iteratori sono astrazioni, compatibili ai puntatori, che garantiscono accesso a elementi all'interno di un particolare contenitore.
  - Gli iteratori sono usati sia per leggere che per modificare gli elementi del contenitore- ci sono differenti tipi di iteratori, con differenti capacità.
  - Gli algoritmi sono funzioni di template parametrizzate(parameterized function templates); esse non sanno il reale tipo dei contenitori su cui operano
  - Gli algoritmi sono intenzionalmente scollegati dai contenitori, e essi usano sempre gli iteratori per accedere agli elementi del container.
  - Gli algoritmi dell'STL hanno una complessità di tempo associate, implementata per l'efficienza(costante,lineare o logaritmica)
  - Sono template di funzione, parametrizzati da iteratori, per accedere ai contenitori su cui essi operano:
- ```
std::vector<int> v;
.. // initialize v
std::sort( v.begin(), v.end() ); // instantiate
std::deque<double> d; // double-ended queue
.. // initialize d
std::sort( d.begin(), d.end() ); // again
```
- Se un algoritmo generale come per esempio l'ordinamento non è disponibile per un contenitore specifico(gli iteratori non sono compatibili), allora è dato come un'operazione membro(ad esempio: std::list)

Containers

- Un container è una classe i cui oggetti contengono una raccolta omogenea di valori
- `Container<T> c; // initially empty`
- Quando inserisci un oggetto in un contenitore, in verità inserisci una copia di valore di questo oggetto
- `c.push_back(value); // grows dynamically`
- L'elemento di tipo T deve supportare una copia del costruttore(che opera una copia corretta e sufficientemente profonda dei dati dell'oggetto)
- Raccolte eterogenee sono rappresentate come contenitori che contengono puntatori a una classe base
 - Questo richiede di gestire tutti i problemi del management pointer/memory (quando svuotando un container, copiando(deep copying) etc).
- I contenitori STL in verità usano due parametri di tipo-data(data-type).
 - Il data type per gli oggetti nel contenitore
 - L'allocatore, gestisce l'allocazione di memoria per un container
- L'allocatore di default(un oggetto di allocatore classe che usa `new` e `delete`) è sufficiente per molti usi, e sarà omesso nei seguenti casi.
- Contenitori di sequenza(Sequence containers), ciascun elemento è posto in una particolare posizione relativa: come primo secondo etc:
 - `vector<T>` vettori, sequenze di lunghezza variabile
 - `deque<T>` dequeues, file che finiscono doppie(con operazione a ciascun termine)
 - `list<T>` liste unite doppiamente
- I containers associative usati per cercare elementi usando una chiave
 - `set <KeyType>` sets with unique keys(imposta con chiavi uniche)
 - `map <KeyType, ValueType>` maps with unique keys
 - `multiset <KeyType>` sets with duplicate keys
 - `multimap <KeyType, ValueType>` maps with duplicate keys
- Container adaptors, sono usati per adattare i contenitori per l'uso delle interfacce specifiche per esempio questi successive sono adattatori di sequenze(sequences):
 - Stack LIFO (last in first out)
 - Queue FIFO (first in first out)
 - `priority_queue` items with higher priority

Containers taxonomy



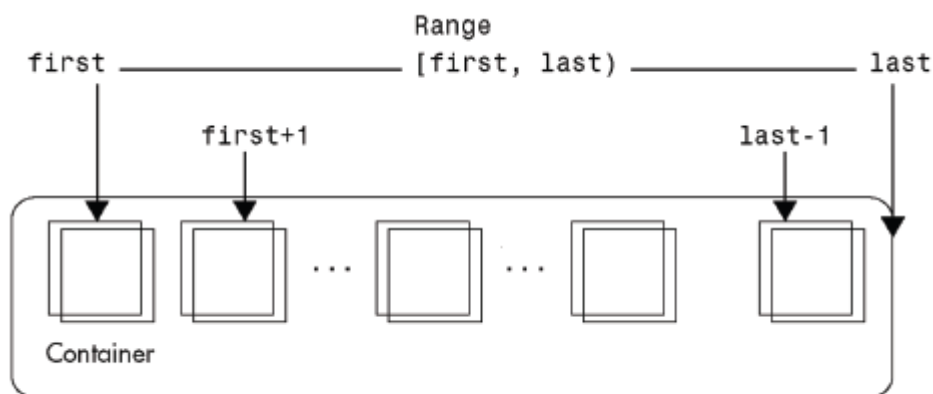
Restrictions on contained types

- I tipi nei contenitori STL devono avere un(qualcosa) accessibile (defaults OK quando applicabile)
 - default constructor
 - destructor
 - assignment operator
 - copy constructor
- Alcune cose richiedono operatori di inequality/equality

Iterators

- Ciascun template(container) definisce un nome di tipo pubblico(public type name) chiamato iteratore che può essere usato per iterazioni di oggetti nel contenitore.
- Nell'STL un iteratore è una generalizzazione di un puntatore(puntatore generico).
- Pensa a un iteratore come a un "puntatore" a un qualunque oggetto nel contenitore in un certo momento. L'operatore asterisco * (di dereferenziazione) è definito per restituire il vero elemento che stava puntando in quel momento.
- Decouples element access from structure(l'accesso dell'elemento decouples dalla struttura)
- Per gli iteratori unidirezionali, ++ è definito per avanzare all'elemento successivo.
- Per operatori bidirezionali -- è definito per andare indietro all'elemento precedente.

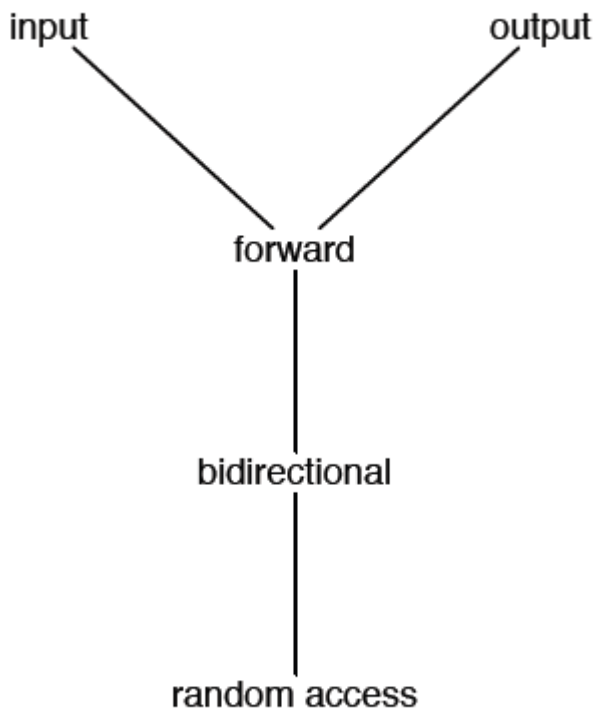
- Qualsiasi contenitore ha funzioni di membro definite `begin()` e `end()` che rispettivamente indicano il primo elemento e l'ultimo.
- Un iteratore garantisce l'accesso agli oggetti immagazzinati in un container (credo il sogg=iteratore punta un elemento); ogni iteratore che esso deve supportare (every iterator it has to support):
 - `*it it->` per accedere all'elemento puntato dall'iteratore
 - `++it` per spostarsi all'elemento successivo del contenitore
 - `it == it1` per comparare due iteratori per l'uguaglianza tra puntatori
 - `it != it1` per comparare due iteratori per l'ineguaglianza tra
- Ogni tipo di container fornisce uno o più iteratori in modo uniforme come nomi tipo standardizzati:
 - `std::vector<std::string>::iterator` // typedef
 - `std::vector<std::string>::const_iterator`
- `begin()` restituisce un iteratore che punta al primo element
returns an iterator pointing to the first element
- `end()` restituisce un iteratore che punta dopo la fine; questo serve da sentinella, vale a dire come un marcatore di fine.



- `C::iterator first = c.begin(), last = c.end();`
- Un contenitore è un discreto set di valori del genere `value_type` (of type `value_type`)
- Un iteratore può puntare sia ad un element del suo contenitore o, semplicemente al di là di esso, usare il valore special oltre la fine `c.end()`
- Può essere deferenziato usando l'operatore `*` (ad esempio `*it`), e l'operatore `->` (ad esempio, `it->op()`).
- Una sequenza di valori consecutive in un contenitore è determinate da un range di iteratore, definite da due iteratori, per esempio: `[first, last)`
 - si suppone che `last` sia raggiungibile in primo luogo usando l'operatore `++`, e tutti gli operatori, includendo `first` ma escludendo `last`, possono essere dereferenziati (gli operatori)
 - Due iteratori possono essere comparati per uguaglianza e disuguaglianza
 - Sono considerate uguali se puntano allo stesso element del contenitore (o entrambi puntano proprio dietro all'ultimo valore)
 - Il compilatore non controlla la validità dei ranges, ad vale a dire che gli itereatori per davvero si riferiscono allo stesso contenitore.
- Le operazioni di iteratore sono sufficient per accedere a un container:


```
Container c; ...
Container::iterator it;
for ( it = c.begin(); it != c.end (); it++) {
.. it->op (); .. std::cout << *it; ..
}
```
- Il `for` può essere rimpiazzato dall'algoritmo `for_each`

- Gli iteratori non-const supportano semantiche di overwrite: modificare o sovrascrivere gli elementi già stoccati in un container.
- Ci sono adattatori di iteratori che supportano semantiche di inserzione, (per esempio aggiungendo nuovi elementi nello stesso punto)
- La validità degli iteratori e dei puntatori non è garantita (come di solito in C/C++)
 - In particolare, la modifica nell'organizzazione di un contenitore spesso invalida tutti gli iteratori e le referenze (dipende dal genere di contenitore e dal genere di modifica).
- Per strutture simili ad array, gli iteratori sono (di solito) implementati come puntatori natii (native) (C-Style) verso elementi dell'array (ad esempio, vector)
 - Molto efficient: usa l'aritmetica dei puntatori
 - Hanno gli stessi problemi di sicurezza come altri puntatori native.
 - Alcune librerie possono fornire speciali iteratori controllati
- Gli iteratori di accesso casuale sono disponibili per operazioni su vettori e deque: `it+=i` , `it-=i` , `it+i` , `it-i` , `it[i]` (access element at it+i) , `<` , `<=` , `>` , `>=`
- Le funzionalità degli iteratori possono essere rappresentate da una gerarchia (non è una gerarchia di classe). Muovendosi in basso gli iteratori aggiungono le funzionalità (gli iteratori del fondo sono più potenti).
 - Input Iterator: `..=*it ++`
 - Output Iterator: `*it=.. ++`
 - Forward Iterator: multipass
 - Bidirectional Iterator: `--`
 - Random Access Iterator: `[] it+i it-i`



Algorithms

- L'STL ha alcuni algoritmi comuni (~70 operations) per: insert, get, search, sort, other math operations (e.g. permute)
- Generic w.r.t. data types and also w.r.t. containers (in reality they are generic w.r.t. the iterator types)
- Basati sull'overload (usa lo stesso nome ma parametric differenti)

- Non necessita di relazioni di ereditarietà
 - I tipi sostituiti non hanno bisogno di avere una comune classe base
 - Hanno solo bisogno di essere modelli del concetto di algoritmo
- Implementazioni in C++:
 - Fa affidamento sui template, e sul polimorfismo basato su interfacce
 - Gli algoritmi sono implementati come template di funzione
 - Usa tipi che modellano i concetti di iteratore
 - L'iteratore a sua volta dà accesso ai contenitori
- The `<algorithm>` header file contains:
 - Non-modifying sequence operations:
 - Do some calculation but don't change sequence itself
 - Examples include `count`, `count_if`
 - Mutating sequence operations:
 - Modify the order or values of the sequence elements
 - Examples include `copy`, `random_shuffle`
 - Sorting and related operations
 - Modify the order in which elements appear in a sequence
 - Examples include `sort`, `next_permutation`
- The `<numeric>` header file contains
 - General numeric operations
 - Scalar and matrix algebra, especially used with `vector<T>`
 - Examples include `accumulate`, `inner_product`

Algorithms example

```
#include <algorithm>
sort( v.begin(), v.end() ); /* sort all of v */
vector<int>::iterator it;
it = find( v.begin(), v.end(), 14 );
/* it is an iterator with elements == 14 in v */
```

Nota che `sort` & `find` prendono gli iteratori

Iterator = (Container + position)

... esattamente il bisogno dell'informazione `sort/find` (exactly the info `sort/find` need)

Gli iteratori forniscono un'interfaccia molto generica

Function objects

- Un oggetto di funzione o Functor (i due termini sono sinonimi) è semplicemente un qualsiasi oggetto che può essere chiamato come se fosse una funzione.
- Una funzione ordinaria è un oggetto di funzione e dunque un puntatore a funzione; più generalmente, così è un oggetto di una classe che definisce `operator()`.
- Molti algoritmi generici (alcuni contenitori) possono richiedere un functor.

Why function objects ?

- Possono essere sviluppati inline
- Potrebbero usare attributi dell'oggetto, per immagazzinare uno stato (invece di usare variabili statiche in una funzione)
- Potrebbe usare un costruttore per definire i dati associati (attributi)

Functor example

```
class IntGreater {
public:
    bool operator()(int x, int y) const {
```

```

return x>y;
}
};
IntGreater intGreater;
int i,j;
//...
bool result = intGreater( i, j );
//... container and iterators...
sort( itrBegin, itrEnd, IntGreater() );

```

Functor example

```

template<class T>
class Summatory {
public:
Summatory(T sum=0) : _sum(sum) {}
void operator()(T arg) { _sum += arg; }
T getSum() const { return _sum; }
private:
T _sum;
};
list<int> li;
Summatory<int> s;
for_each( li.begin(), li.end(), s);
cout << s.getSum() << endl;

```

Sequences

Vector, List, Deque

Sequences

- I contenitori STL forniscono molte specie di sequenze
- Vettori quando:
 - ci sono operazioni di accesso casuali
 - molte aggiunte e rimozioni sono alla fine del container
- deque quando:
 - ci sono aggiunte frequenti e cancellazioni su entrambi i lati
 - ci sono operazioni di accesso casuali
- lists quando:
 - ci sono frequenti inserzioni e cancellazioni in posizioni diverse dalla fine
 - ci sono poche operazioni di accesso casuali(ad eccezione dell'accesso sequenziale)
 - vuoi garantire che gli iteratori sono validi dopo modifiche strutturali

Sequences example

```

std::deque<double> d(10, 1.0); // deque with 10 values
(1.0)
std::vector<Integer> v(10); // vector with 10 Integers;
each with default value
std::list<Integer> s1; // empty list
// store some elements:
s1.push_front( Integer(6) );
s1.insert( s1.end(), Integer(13) ); ..
// create list s2 that is a copy of s1

```

```
std::list<Integer> s2( s1.begin(), s1.end() );
// reinitialize all elements to Integer(2)
s2.assign( s2.size() - 2, Integer(2) ); // two fewer
```

Sequences: some methods

Constructor (copy): `Sequence(size_type n, const T& v = T())`
 create n copies of v. If the type T does not have a no-arg constructor, then use explicit call to the constructor

Re-construction: `assign(first, last)`

copy the range defined by input iterators first and last, dropping all the elements contained in the vector before the call and

replacing them by those specified by the parameters

`assign(size_type n, const T& v = T())`

assign n copies of v

Access: `reference front()`

first element. A reference type depends on the container; usually it is T&.

`reference back()`

last element

Insertions and deletions: `iterator insert(iterator p, T t)`

insert a copy of t before the element pointed to by p and return the iterator pointing to the inserted copy

`void insert(iterator p, size_type n, T t)`

insert n copies of t before p

`void insert(iterator p, InputIterator i, InputIterator j)`

insert copies of elements from the range [i,j) before p

`iterator erase(iterator p)`

remove the element pointed to by p, return the iterator pointing to the next element if it exists; `end()` otherwise

`iterator erase(iterator i, iterator j)`

remove the range [i,j), return the iterator pointing to the next element if it exists; `end()` otherwise

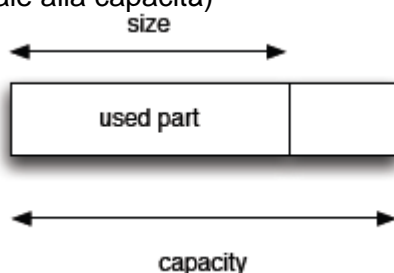
`clear()`

remove all elements

Vectors

- Una sequenza che supporta l'accesso casuale agli elementi
- Gli elementi possono essere inseriti o rimossi all'inizio alla fine e nel mezzo
- L'accesso random può essere ottenuto in tempo costante
- Operazioni usate comunemente:
 - `begin()`, `end()`, `size()`, `[], push_back(...)`, `pop_back()`, `insert(...)`, `empty()`

- La classe di vettore template rappresenta un array flessibile e resizable.
- La capacità è il numero Massimo di elementi che può essere ottenuta senza una riallocazione e copia degli elementi(`allocated by reserve ()`)
- Size è il numero corrente di elementi veramente immagazzinati nel vettore(sempre minore o uguale alla capacità)



- Quando inserisci un nuovo element e non c'è più spazio(room), cioè la grandezza già equaglia la capacità, allora il vettore viene riallocato.
- Aggiunte alla fine di un vettore sono ottenibili in un tempo costante(mentre un'aggiunta individuale potrebbe essere lineare nella ampiezza corrente)
- Nella riallocazione, ogni iteratore o riferimento sono invalidati(are invalidated)
- Nota che le operazioni di overwriting non riallocano i vettori così il programmatore deve evitare ogni corruzione o fuoriuscita della memoria.

Vectors: some methods

Capacity

`capacity()`

current capacity

`reserve(n)`

allocate space for n elements

`resize(n, t = T())`

If $n > \text{size}$ then add new n-size elements; otherwise decrease the size

Accessors

reference `operator[]`

reference `at()` `throw(out_of_range)`

checked access

Modifiers

`push_back()`

Insert a new element at the end; expand vector if needed

`pop_back()`

remove the last element; undefined if vector is empty

Vector example

```
// Instantiate a vector
vector<int> V;
V.reserve(100); // allocate space for 100 int
// Insert elements
V.push_back(2); // v[0] == 2, constant
time!
// after insert: V[0] == 3, V[1] == 2
V.insert( V.begin(), 3 ); // linear time!
// Random access
V[0] = 5; // V[0] == 5
cout << V[1] << endl;
// Test the size
int size = V.size(); // size == 2
vector<int> Vcopy(V); // use copy constructor
```

Vector and iterator example

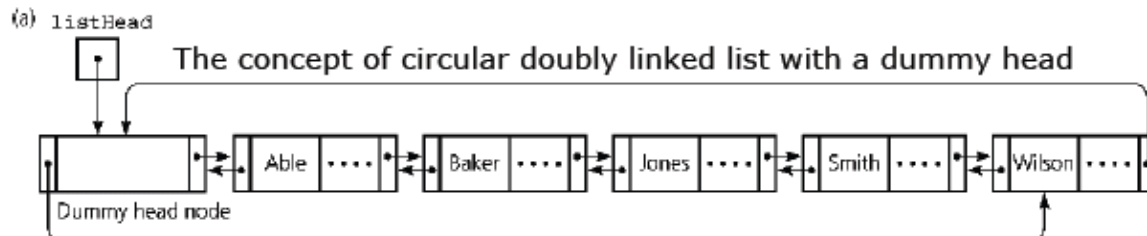
```
// the iterator type is inside vector<int> !
vector<int>::iterator it = myVect.begin();
while (it != myVect.end()) {
    int x = *it;
    cout << "Current thing is " << x << endl;
    it++;
}
```

Deque

- Deques sono simili ai vettori
- Gli iteratori deque sono ad accesso casuale
- Inoltre due operazioni per inserire o rimuovere gli elementi difronte:
 - `push_front()` aggiungi il nuovo primo elemento
 - `pop_front()` rimuovi il primo elemento
- deque non hanno operazioni `capacity()` e `reserve()`

Lists

- La lista di classe STL è tipicamente implementata come una lista circolare connessa doppiamente.
 - With a dummy head node.
- `ll begin()` restituisce un iteratore al primo elemento della lista
- `ll end()` restituisce un iteratore al dummy head node (to the dummy head node in the list)



Lists: some methods

Modifiers

`push_front(t)`

insert at back

`pop_front()`

delete from front

Auxiliary (specialized for lists)

`sort()`

to sort the list

`sort(cmp)`

to sort the list using the comparison object function cmp

`reverse()`

to reverse a list

`remove(const T& value)`

uses `==` to remove all elements equal to v

`remove_if(pred)`

uses the predicate pred

`unique()`

remove consecutive duplicates using `==`

`unique(binpred)`

remove consecutive duplicates using the binary predicate binpred

`head.splice(i_head, head1)`

move the contents of head1 before iterator i_head, which must point to a position in head, and empty the list head1

`head.merge(list& head1)`

merge two sorted lists into head, empty the list head1.

List example

```
list<char> s; // empty list
```

```

s.insert ( s.end(), 'a');
s.insert ( s.end(), 'b'); // s has (a, b)
list <char> s1; // empty list
// copy s to s1:
s1.insert ( s1.end(), s.begin(), s.end() );
s.clear ();
assert( s1.front() == 'a' );
s1.erase ( s1.begin() ); // remove first
element
assert( s1.front () == 'b' );

```

Associative containers

Set, Multiset, Map, Multimap

Overview

- I containers associativi sono una generalizzazione di sequenze. Le sequenze hanno un indice intero(are indexed by Integers); i contenitori associativi possono essere messi sotto indice con qualsiasi tipo.
- Il tipo più comune da usarsi come chiave è una string; puoi avere un insieme di stringhe o una mappa dalle stringhe agli operatori(to employees), e così via
- è spesso utile avere altri tipi come chiave; per esempio se voglio conservare traccia dei nomi di tutti i Widgets in una applicazione, io potrei usare una mappa da Widgets a Strings.
- Permettono di aggiungere o di cancellare elementi, una query(Il termine query, in informatica viene utilizzato per indicare l'interrogazione di un database in modo da ottenere dei dati contenuti in uno o più database) chiedere una membership(query for membership), o iterare attraverso il set.
- Multisets sono simili ai set con l'eccezione che è possibile avere diverse copie dello stesso elemento(queste sono spesso chiamate bags)
- Le mappe rappresentano una mappatura da un tipo(il tipo chiave) a un altro tipo(il tipo valore). È possibile associare un valore con una chiave o trovare il valore associato con una chiave, molto efficientemente; le maps possono iterare attraverso tutte le chiavi
- Le multimaps sono simili alle mappe ad eccezione che una chiave può essere associata con diversi valori.

Sets

- Gli elementi contenuti nel set sono ordinate basandosi su una funzione oggetto Compare(default <operator)
- Nessun accesso casuale, solo avanti e indietro(forward and reverse)
- La classe fornisce inserzione/cancellazione/ricerca/metodi di conteggio
- Usa gli algoritmi STL per l'unione/l'intersezione/la differenza..

Sets example

```

set<int> s;
int a[]={0,1,2,3,4,5,6,7,8,9};
s.insert( a, a+10 );
cout << s.count(5); // number of elements == 5
// search the first element >= 5
cout << s.lower_bound(5);

```


Maps

- Il concetto più importante qui è che una mappa consente la gestione di una key-value pair.
- La sua dichiarazione dunque ti permette di specificare tipi come chiave e valore(the “key” and the “value”).
- Chiavi uniche sono valori mappati
- Un valore è recuperato usando la sua unica chiave
- Può specificare una funzione di comparazione per le chiavi(gli elementi sono ordinati usando la funzione)

Maps example

```
#include <map>
map<string, int> mp;
mp["Jan"] = 1;
mp["Feb"] = 2;
mp["Mar"] = 3;
//...
cout << "Mar is month " << mp["Mar"] <<
endl;
```

Maps example 2

```
map<string, int> m;
m.insert( make_pair("Wallace", 9999) );
m.insert( make_pair("Gromit", 3343) );
map<string, int>::iterator p;
p = m.find("Wallace");
if( p != m.end() )
cout << "Wallace's extension is: " p->second <<
endl;
else
cout << "Key not found." << endl;
m["Wallace"] = 1679;
cout << "New value is: " << m["Wallace"] << endl;
```

Cleaning up containers of pointers

From “Thinking in C++” - Bruce Eckel

Motivation

- Fai attenzione a pulire un container di puntatori: devi chiamare i distruttori appropriati per rilasciare memoria e evitare perdite.
- Usa le funzioni template suggerite da Bruce Eckel per pulire(purge) i containers.
 - Fai attenzione che un puntatore ad oggetto sia ordinato in due container per evitare la doppia cancellazione.

```
/*
 * Thinking in C++ 2nd Ed.
 * Bruce Eckel, chap. 15
```

```

*
*/
#ifndef __PURGE_H__
#define __PURGE_H__
#include <algorithm>
using namespace std;
template<class Seq> void purge(Seq& c) {
typename Seq::iterator i; // typename keyword says that Seq::iterator is
a
type
for (i = c.begin(); i != c.end(); i++ ) {
delete *i;
*i = 0; // a double purge will do no harm: delete 0 is OK
}
}
template<class InpIt> void purge(InpIt begin, InpIt end) {
while (begin != end) {
delete *begin;
*begin = 0; // a double purge will do no harm: delete 0 is OK
begin++;
}
}
}
#endif

```

The typename keyword

- Usa la keyword `typename` se hai un nome qualificato che si riferisce a un tipo e dipende da un parametro template. Usa solamente la keyword `typename` in dichiarazioni template e definizioni. L'esempio che segue illustra l'uso di una keyword `typename`.

- `template<class T> class A {`
`typedef char C;`
`A::C d; // WRONG: use typename A::C d;`
`}`

- Lo statement `A::C d;` è mal formato. La classe `A` si riferisce anche a `A<T>` e così dipende da un parametro template. Devi aggiungere la keyword `typename` all'inizio di questa dichiarazione.
- Usa la parola chiave `typename` per dire al compilatore che il successivo identificatore è un tipo e NON un membro classe(class member).

Algorithms

A few examples

Algoritmi non modificabili

- L'algoritmo `count`
 - Si muove attraverso un range iteratore
 - Controlla ogni posizione per l'uguaglianza(Checks each position for equality)
 - Accresce il conteggio se è uguale

```

#include <iostream>
#include <vector>
#include <algorithm>

```

```

using namespace std;

```

```

int main (int, char * [])
{

```

```

vector<int> v;
v.push_back(1); v.push_back(2);
v.push_back(3); v.push_back(2);

int i = 7;
cout << i << " appears "
      << count(v.begin(), v.end(), i)
      << " times in v" << endl;

i = 2;
cout << i << " appears "
      << count(v.begin(), v.end(), i)
      << " times in v" << endl;

return 0;
}

```

Usando un oggetto di funzione

- L'algoritmo `count_if`
 - Generalizza l'algoritmo di conteggio
 - Invece di comparare per l'eguaglianza a un valore (Instead of comparing for equality to a value)
 - Applica un dato oggetto di funzione predicato (functor)
 - Se il risultato di un functor è vero accresce il conteggio

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T>
struct odd {
    bool operator() (T t) const
    {
        return (t % 2) != 0;
    }
};

int main (int, char * []) {

    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(2);

    cout << "there are "
          << count_if(v.begin(), v.end(),
                      odd<int>())
          << " odd numbers in v" << endl;

    return 0;
}

```

Usando un algoritmo di ordinamento

- L'algoritmo di `sort`
 - Riordina on dato range
 - Può anche inserire (pug in) un funto per cambiare la funzione ordinatrice

- L'algoritmo della `next_permutation`
 - Genera uno specifico genere di riordino, chiamato "permutazione"
 - Si può usare per generare ogni possibile ordine di una data sequenza

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

int main (int, char * []) {

    string s = "asdf";
    cout << "original: " << s << endl;

    sort (s.begin(), s.end());
    cout << "sorted: " << s << endl;

    string t(s);
    cout << "permutations:" << endl;

    do {
        next_permutation (s.begin(), s.end());
        cout << s << " ";
    } while (s != t);

    cout << endl;

    return 0;
}
```

Usando gli algoritmi numerici

- L'algoritmo di `accumulate`
 - Somma gli elementi in un range(basato su un valore di somma iniziale)
- L'algoritmo di `inner_product`
 - Calcola il prodotto interno dei due vettori(pure conosciuto come "dot"): somma dei prodotti dei loro rispettivi elementi.

Computes the inner

```
#include <iostream>
#include <vector>
#include <numeric>

using namespace std;

int main (int, char * []) {

    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(2);

    cout << "v contains ";
    for (size_t s = 0; s < v.size(); ++s) {
        cout << v[s] << " ";
    }
}
```

```
}
cout << endl;
cout << "the sum of the elements in v is "
    << accumulate (v.begin(), v.end(), 0)
    << endl;
cout << "the inner product of v and itself is "
    << inner_product (v.begin(), v.end(),
                      v.begin(), 0)
    << endl;

return 0;
}
```

Exceptions

What are exceptions ?

- Le eccezioni sono un meccanismo per gestire un errore durante l'esecuzione.
- Una funzione può indicare che un errore è accaduto gettando un'eccezione.
- Si dice che il codice che tratta dell'eccezione lo gestisce (handle it).

Why use exceptions ?

- Puoi separare il code dove l'errore accade e la codifica nel trattamento dell'errore.
(where the error occurs and code to deal with the error can be separated)
- Si possono usare eccezioni con i costruttori e con altri operatori /funzioni che non possono restituire un codice di errore.
- Eccezioni opportunamente implementate conducono a un codice migliore.

How to use exceptions ?

- **try**
 - Prova ad eseguire alcuni blocchi di codice
 - Guarda se c'è un errore
- **throw**
 - Una condizione di errore capitata
 - Lancia un'eccezione per registrar il fallimento
- **catch**
 - Gestisce un'eccezione gettata in un try-block

How exceptions work ?

- Il flusso normale del controllo del programma è fermato
 - Nel punto in cui un'eccezione è lanciata
- The program call stack "unwinds"
 - Stack frame of each function in call chain "pops"
 - Variables in each popped frame are destroyed
 - Proceede fino a quando non è raggiunto il campo di azione che include il try/catch
(Goes until an enclosing try/catch scope is reached)
- Il controllo passa al primo blocco catch di incontro (che combacia matching)
 - Può gestire l'eccezione e continuare da lì
 - Può liberare alcune risorse e gettare nuovamente l'eccezione

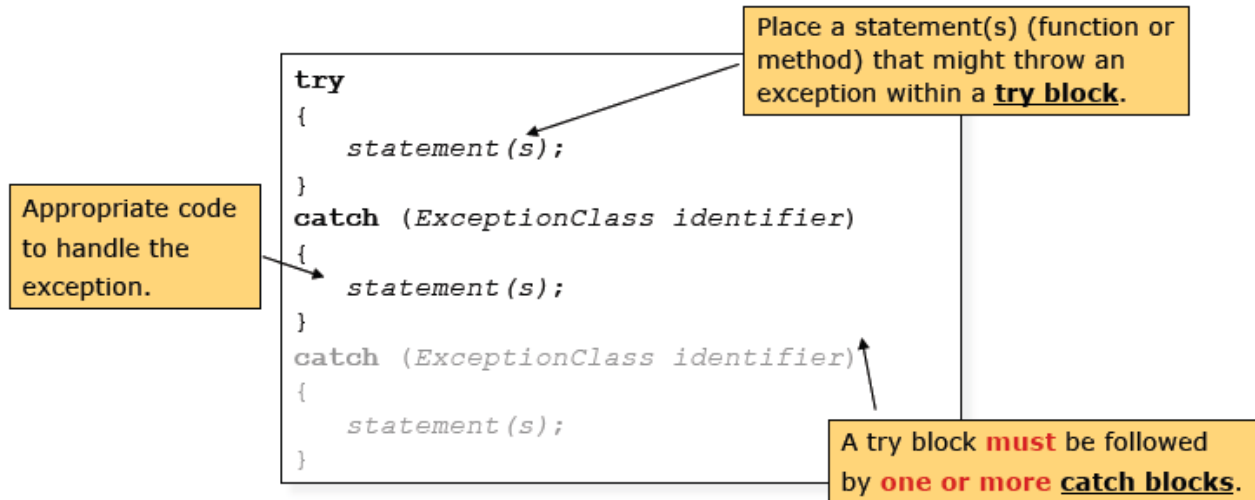
What's right about exceptions

- Non può essere ignorato in modo silente: se non c'è alcun blocco catch applicabile per una eccezione il programma termina.
- Si propaga automaticamente attraverso il campo di azione (dovuto allo stack unwinding)
- La gestione fuori dal principale flusso di controllo, il codice che implementa l'algoritmo non è corrotto (polluted)

Exceptions syntax

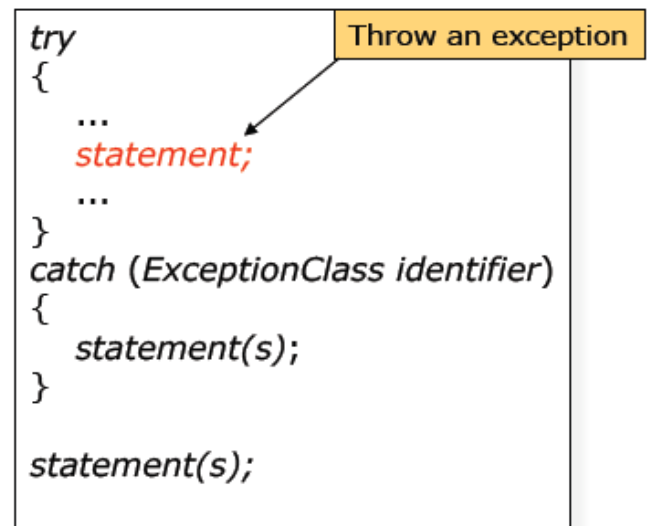
C++ exceptions syntax

- Usa i blocchi `try-catch` per catturare un'eccezione



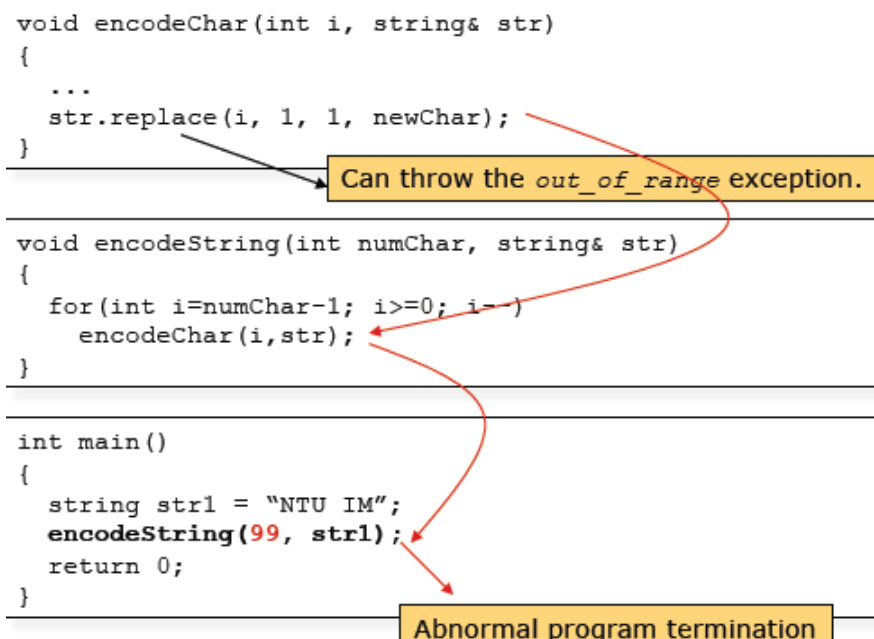
C++ exception flow

- Quando uno statement(funzione o metodo) causa un'eccezione in un try-block:
 - Il resto del try-block è ignorato
 - Il controllo passa al catch-block che corrisponde all'eccezione
 - Dopo che un catch block esegue, il controllo passa a uno statement dopo l'ultimo catch block associato con il try block.



C++ exception flow

- Un esempio più complesso di flusso di eccezione:



Catching the exception

- Due esempi di come catturare un'eccezione:

Primo:

```
void encodeChar(int i, string& str)
{
    try
    {
        ...
        str.replace(i, 1, 1, newChar);
    } catch (out_of_range e) {
        cout << "No character at " << i << endl;
    }
}
```

```
void encodeString(int numChar, string& str)
{
    for(int i=numChar-1; i>=0; i--)
        encodeChar(i, str);
}
```

```
int main()
{
    string str1 = "NTU IM";
    encodeString(99, str1);
    return 0;
}
```

No character at 98
No character at 97
...

Secondo:

```
void encodeChar(int i, string& str)
{
    ...
    str.replace(i, 1, 1, newChar);
}
```

```
void encodeString(int numChar, string& str)
{
    try
    {
        for(int i=numChar-1; i>=0; i--)
            encodeChar(i, str);
    } catch (out_of_range e) {
        cout << "Something wrong" << endl;
    }
}
```

```
int main()
{
    string str1 = "NTU IM";
    encodeString(99, str1);
    return 0;
}
```

Something wrong

Handlers

- Un handler può gettare di nuovo l'eccezione che è stata passata:
 - fa proseguire l'eccezione
 - Uso: `throw;` // no operand
 - dopo la pulizia dell'endler locale resterà in esercizio l'endler corrente (after the local handler cleanup it will exit the current handler)

- Un handler può gettare un'eccezione di tipo diverso
 - Traduce l'eccezione

Catching multiple exceptions

- L'ordine delle clausole catch è importante:
 - Soprattutto con le classi di eccezione relative all'ereditarietà
 - Poni più specifici catch block prima di quelli più generali
 - Poni i catch blocks per classi di eccezioni più derivate prima dei catch blocks per le loro classi base rispettive.
- `catch(...)` cattura ogni tipo

Catching multiple exceptions example

```
try {
    // can throw exceptions
} catch (Derived &d) {
    // Do something
} catch (Base &d) {
    // Do something else
} catch (...) {
    // Catch everything else
}
```

Throwing exceptions

- Quando scopri un errore all'interno di un metodo, puoi gettare un'eccezione usando un throw statement
- Il codice rimanente all'interno della funzione non esegue
- Syntax(sintassi):

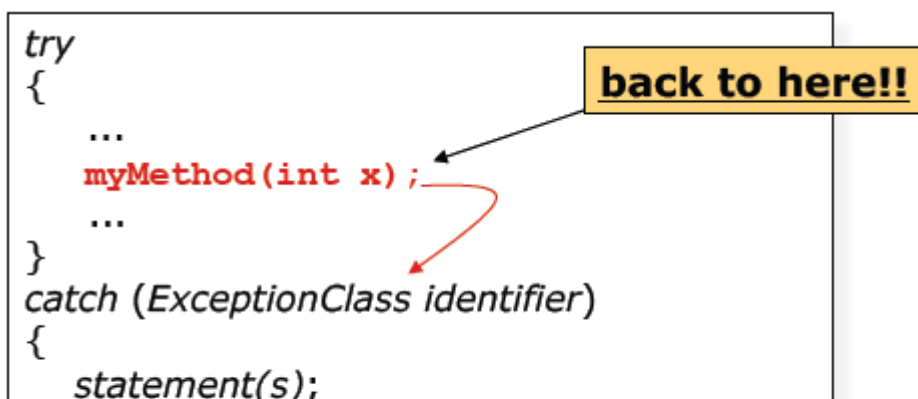
```
throw ExceptionClass(stringArgument);
```

type of the exception

more detailed information

```
void myMethod(int x) throw(MyException)
{
    if (...)
        throw MyException("MyException: ...");
    ...
} // end myMethod
```

- L'eccezione è propaga indietro nel punto in cui è stata chiamata la funzione.



Specifying exceptions

- Le funzioni che gettano un'eccezione hanno una clausola throw, per restringere le eccezioni che una funzione può gettare.
 - Permette un tipo di controllo più forte rafforzato dal compilatore
 - Per default, una funzione può gettare qualsiasi cosa voglia
- Una clausola throw in una function's signature:
 - Limita cioè che può essere lanciato
 - Una promessa alle funzioni di richiamo (A promise to calling function)
- Una clausola throw senza tipi:
 - Dice che non sarà gettato nulla
- Può fare la lista di tipi multipli, comma separated (separati da una virgola)

Specifying exceptions examples

```
// can throw anything
void Foo::bar();
// promises not to throw
void Foo::bar() throw();
// promises to only throw int
void Foo::bar() throw(int);
// throws only char or int
void Foo::bar() throw(char,int);
```

Destructors and exceptions

Destructors and exceptions

- Evita che le eccezioni lascino i distruttori:
un prematuro termine di programma o un comportamento non definito può essere il risultato di distruttori che emettono eccezioni
 - durante lo stack unwinding che risulta dal processo dell'eccezione, i distruttori degli oggetti locali sono chiamati e uno può avviare un'altra eccezione.

How to behave: example

```
class DBConnection {
public:
    //...

    // return a DBConnection object
    static DBConnection create();

    void close(); // close connection and
                 // throws exception if
                 // closing fails
};

// class to manage DBConnection
class DBConnMgr {
public:
    //...
    DBConnMgr(DBConnection dbc);
    ~DBConnMgr() {
        dbc.close(); // we're sure it
                    // gets closed
    }
};

// client code
{
    DBConnMgr dbc( DBConnection::create() );
    //... use DBConnection through DBConnMgr interface
} // DBConnMgr obj is automatically destroyed, calling the close
```

If close() throws the
destructor propagates the
exception

```
Private:
    DBConnection db;    → Ditetro l' "if close()... exception"
};
```

(Not so good) solutions

- Terminate the program:

```
DBConnMgr::~~DBConnMgr() {
    try{ db.close(); }
    catch (...) {
        // log failure and...
        std::abort();
    }
}
```

- Swallow/Copre the exception:

```
DBConnMgr::~~DBConnMgr() {
    try{ db.close(); }
    catch (...) {
        // just log the error
    }
}
```

With this solution we're
hiding the problem

A better strategy

```
// class to manage DBConnection
```

```
class DBConnMgr {
```

```
public:
```

```
    //...
```

```
    DBConnMgr(DBConnectio dbc);
```

```
    void close() {
```

```
        db.close();
```

```
        closed = true;
```

```
    }
```

```
    ~DBConnMgr() { // we're sure it gets closed
```

```
        if( !closed ) {
```

```
            try {
```

```
                db.close();
```

```
            } catch (...) {
```

```
                // log and... terminate or swallow
```

```
            }
```

```
        }
```

```
    }
```

```
private:
```

```
    DBConnection db;
```

```
    bool closed;
```

```
};
```

Client code should use
this method...

...but if it doesn't
there's the destructor

Defining exceptions classes

Syntax and example

Defining exceptions classes

- La C++ Standard Library fornisce un numero di classi di eccezione
 - Ad esempio, `exception`, `out_of_range`, ... etc.
- Forse vuoi anche definire la tua propria classe di eccezione
 - Dovresti ereditare da quelle classi di eccezione predefinite per una eccezione standardizzata che opera con interfaccia. (working interface)
- Syntax:

```
#include <stdexcept>
using namespace std;
```

Defining exceptions classes

```
#include <exception>
#include <string>
using namespace std;

class MyException : public exception
{
public:
    MyException(const string & Message = "")
        : exception(Message.c_str()) {}
} // end class
```

```
try
{
    ...
}
catch (MyException e)
{
    cout << e.what();
}
```

```
throw MyException("more detailed information");
```

A full example

- Una implementazione di una lista ADT usando le eccezioni:
 - fuori dall'indice di lista bloccato (out-of-bound list index)
 - tentativo di inserirsi in una lista completa (attempt to insert into a full list)

Define two exception classes

```
#include <stdexcept>
#include <string>
using namespace std;
```

```
class ListIndexOutOfRangeException : public out_of_range {
public:
    ListIndexOutOfRangeException(const string& message = "")
        : out_of_range(message.c_str()) {}
}; // end ListException
```

```

class ListException : public logic_error {
public:
    ListException(const string & message = "")
        : logic_error(message.c_str()) {}
}; // end ListException

```

Declare the throw

```

#include "MyListExceptions.h"
...
class List
{
public:
    ...
    void insert(int index, const ListItemType& newItem)

        throw(ListIndexOutOfRangeException,
              ListException);

    ...
} // end List

```

Method implementation

```

void List::insert(int index, const ListItemType& newItem)
    throw(ListIndexOutOfRangeException, ListException) {
    if (size >= MAX_LIST)
        throw ListException("ListException: List full on insert");
    if (index >= 1 && index <= size+1) {
        for (int pos = size; pos >= index; --pos)
            items[translate(pos+1)] = items[translate(pos)];
        // insert new item
        items[translate(index)] = newItem;
        ++size; // increase the size of the list by one
    } else // index out of range
        throw ListIndexOutOfRangeException(
            "ListIndexOutOfRangeException: Bad index on insert");
} // end insert

```

Good Programming Style with C++ Exceptions

- Non usare le eccezioni per un flusso di programma normale
 - Usale solo dove non sia possibile il normale flusso
- Non permettere alle eccezioni di lasciare il main o i costruttori (Don't let exceptions leave main or constructors)
 - Viola un'inizializzazione o un termine "normale".
- Getta sempre qualche tipo
 - Così l'eccezione può essere catturata
- Usa le specificazioni dell'eccezione in modo ampio
 - Aiuta il chiamante a conoscere possibili eccezioni da catturare

Exception-safe functions

- Exception-safe functions (le funzioni salva eccezioni) offrono una delle tre garanzie:
 - garanzia base (basic guarantee): se un'eccezione è gettata, ogni cosa nel programma rimane in un valido stato
 - forte garanzia (strong guarantee): se un'eccezione è gettata lo stato del programma non è cambiato. La chiamata alla funzione è atomica (atomic)
 - Garanzia nothrow (nothrow guarantee): la promessa di non gettare mai un'eccezione: le funzioni fanno sempre ciò che promettono. Tutte le operazioni di tipo in costruzione (on built-in) sono notaro. (they always do what they promise. All operations on built-in types are nothrow.)

Exception-safe code

- Quando un'eccezione è gettata, le funzioni exception safe:
 - non perdono alcuna risorsa (ad esempio new-ed objects, handles, etc.)
 - non permettono alle strutture dei dati di corrompersi (ad esempio, un puntatore che doveva puntare a un nuovo oggetto è stato lasciato puntare a NULL (was left pointing to nowhere))

Resource Management

Memory, `auto_ptr<>` and RAI

- La risorsa più comunemente usata nei programmi C++ è la memoria
 - ci sono file handles, mutexes, database e connections, etc.
- è importante rilasciare una risorsa dopo che è stata usata

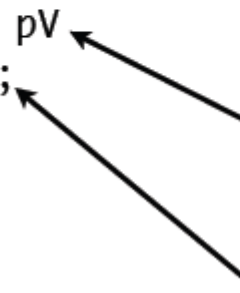
An example

```
class Vehicle { ... }; // root class of a hierarchy
```

```
Vehicle* createVehicle(); /* return a pointer to root  
class but may create any other object in the hierarchy.  
The caller MUST delete the returned object */
```

```
void f() {  
    Vehicle* pV = createVehicle();  
    //... use pV  
    delete pV;  
}
```

If there's a premature
return or an exception we
may never reach the
delete !



A solution

- Poni la risorsa restituita da `createVehicle` dentro un oggetto il cui distruttore automaticamente rilascia la risorsa quando il controllo lascia `f()`.
 - Le chiamate del distruttore sono automatiche
- Con questi oggetti che gestiscono le risorse:
 - le risorse sono acquisite e immediatamente ritornate agli oggetti che gestiscono la risorsa RAI (resources are acquired and immediately turned over to resource-managing objects (RAI))
 - Questi oggetti usano i loro distruttori per assicurare che le risorse sono rilasciate

RAI

Resource Acquisition Is Initialization

What is RAI

- Questa tecnica è stata inventata da Stroustrup per gestire la de allocazione della risorsa in C++ e per scrivere un codice sicuro dal punto di vista delle eccezioni: il solo codice che può essere garantito come eseguibile dopo che un'eccezione è gettata sono i distruttori di oggetti che si trovano nello stack.

- Questa tecnica permette di rilasciare risorse prima di permettere alle eccezioni di propagarsi (per evitare le perdite di risorsa)
- Le risorse sono legate alla durata di vita degli oggetti adatti. Essi sono acquisiti durante l'inizializzazione, quando non c'è opportunità per loro di essere usate prima di essere disponibili. Esse sono rilasciate con la distruzione degli stessi oggetti, che è garantita cosicché abbia luogo anche in casi di errori.

RAII example

```
#include <cstdio>
#include <stdexcept> // std::runtime_error

class file {
public:
    file (const char* filename) : file_(std::fopen(filename, "w+")) {
        if (!file_) {
            throw std::runtime_error("file open failure");
        }
    }
    ~file() {
        if (std::fclose(file_)) {
            // failed to flush latest changes.
            // handle it
        }
    }
    void write (const char* str) {
        if (EOF == std::fputs(str, file_)) {
            throw std::runtime_error("file write failure");
        }
    }
private:
    std::FILE* file_;
    // prevent copying and assignment; not implemented
    file (const file &);
    file & operator= (const file &);
};

void example_usage() {
    // open file (acquire resource)
    file logfile("logfile.txt");
    logfile.write("hello logfile!");
    // continue using logfile ...
    // throw exceptions or return without
    // worrying about closing the log;
    // it is closed automatically when
    // logfile goes out of scope
}
```

auto_ptr

- `auto_ptr` è un oggetto simile a un puntatore (a smart pointer) il cui distruttore automaticamente chiama la cancellazione su ciò a cui esso punta
 - è nella C++ standard library:
- esistono altri smart pointers (per esempio, Boost)

auto_ptr: an example

- Riconsidera la funzione `f()` usando `auto_ptr`:

```
void f() {

    std::auto_ptr<Vehicle> pV(createVehicle());

    // use pV as before...
```



```
} /* the magic happens here: automatically
deletes pV via the destructor of auto_ptr,
called because it's going out of scope */
```

auto_ptr: another example

- In generale ecco come trasformare un codice non sicuro in un codice sicuro:

```
// Original code
void f() {
    T* pt( new T );
    /*...more code...*/
    delete pt;
}

//Safe code, with auto_ptr
void f() {
    auto_ptr<T> pt( new T );
    /*...more code...*/
} /* pt's destructor is called
as it goes out of scope, and
the object is deleted
automatically */
```

auto_ptr characteristics

- poiché l' auto_ptr cancella automaticamente ciò a cui punta quando è distrutto, non ci dovrebbero essere due auto_ptr che puntano ad un oggetto:
 - o l'oggetto può essere cancellato due volte: è un comportamento non definito, se siamo fortunati il programma semplicemente chashes.
- Per evitare ciò, auto_ptr ha una caratteristica speciale: copiandoli(ovvero il costruttore di copia o l'operatore di assegnamento) li pone a null(puntano a null) e copiando, il puntatore assume la proprietà dell'oggetto(the ownership of the object)

auto_ptr characteristics: example

```
// pV1 points to the created object
std::auto_ptr<Vehicle> pV1(createVehicle());

std::auto_ptr<Vehicle> pV2( pV1 );
/* now pV2 points to the object and pV1 is
null ! */

pV1 = pV2;
/* now pV1 points to the object and pV2 is
null ! */
```

- Se l'obiettivo dell'auto_ptr tiene un qualche oggetto esso è liberato (If the target auto_ptr holds some object, it is freed)
- Questo comportamento nella copia significa che tu non puoi creare un container STL di auto_ptr!
 - Ricorda: I container STL vogliono oggetti con comportamenti di copia normale
 - I moderni compilatori con moderni STL emettono errori di compilazione.

- Se non vuoi allentare la proprietà(ownership) usa l'idioma const auto_ptr idiom:
(If you do not want to loose ownership use the const auto_ptr idiom)

```
const auto_ptr<T> pt1( new T );
    // making pt1 const guarantees that pt1 can
    // never be copied to another auto_ptr, and
    // so is guaranteed to never lose ownership
```

```
auto_ptr<T> pt2( pt1 ); // illegal
auto_ptr<T> pt3;
pt3 = pt1; // illegal
pt1.release(); // illegal
pt1.reset( new T ); // illegal
```

- semplicemente permette la deferenza

- gli auto_ptr usano cancellare il suo distruttore così non usarlo coin arrays allocati dinamicamente:
(use delete in its destructor so do NOT use it with dynamically allocated arrays)

```
std::auto_ptr<std::string>
aPS(new std::string[10]);
```

- Usa un vettore invece di un array

auto_ptr methods

- Usa get() per ottenere un puntatore ad un oggetto gestito da auto_ptr, o ottenere uno 0 se sta puntando a null.
- usa release() per impostare il puntatore interno auto_ptr a puntatore nullo(il che indicache esso non punta nessun oggetto) senza distruggere l'oggetto al momento puntato da auto_ptr.
- usa reset() per deallocare l'oggetto puntato e imposta un nuovo valore(è come creare un nuovo auto_ptr)

```
auto_ptr<int> p (new int);
*p.get() = 100;
cout << "p points to " << *p.get() << endl;
auto_ptr<int> auto_pointer (new int);
int * manual_pointer;
*auto_pointer=10;
manual_pointer = auto_pointer.release();
cout << "manual_pointer points to " <<
*manual_pointer << "\n";
// (auto_pointer is now null-pointer auto_ptr)
delete manual_pointer;
```

```

auto_ptr<int> p;
p.reset (new int);
*p=5;
cout << *p << endl;

p.reset (new int);
*p=10;
cout << *p << endl;

```

auto_ptr methods - cont.

- L'operatore `*`() e l'operatore `->`() sono stati sovraccaricati(overloaded) e restituiscono l'elemento puntato dall'oggetto `auto_ptr` per avere accesso a a uno dei suoi membri

```

auto_ptr<Car> c(new Car);
c->startEngine();
(*c).getOwner();

```

Scope guard

- Qualche volta non vogliamo rilasciare risorse se non è gettata alcuna eccezione ma noi vogliamo per davvero rilasciarle se l'eccezione è gettata. Lo "scope guard" è una variazione del RAI

- | | |
|---|---|
| <ul style="list-style-type: none"> • <pre> Foo* createAndInit() { Foo* f = new Foo; auto_ptr<Foo> p(f); init(f); // may throw // exception p.release(); return f; } </pre> | <ul style="list-style-type: none"> • <pre> int run () { try { Foo *d = createAndInit(); return 0; } catch (...) { return 1; } } </pre> |
|---|---|

- ```
Foo* createAndInit() {
 Foo* f = new Foo;
 auto_ptr<Foo> p(f);
 init(f); // may throw
 // exception
 p.release();
 return f;
}
```

Use `auto_ptr` to guarantee that an exception does not leak

When we are safe release the `auto_ptr` and return the pointer

# Design patterns

## What are design patterns ?

- Nell'ingegneria del software un design pattern è una soluzione generale riutilizzabile per un problema che accade comunemente nel design del software.
- Un design pattern non è un design finito che può essere trasformato direttamente in un codice
- è una descrizione o un template su come risolvere un problema che può essere usato in molte situazioni differenti
- I patterns sono soluzioni ricorrenti ai problemi del design


## When DPs were developed ?

- L'idea ha origine da un libro che organizzava una conoscenza implicita su come le persone risolvessero problemi ricorrenti nel costruire le cose:  
"ciascun modello descrive un problema che accade più volte nel nostro ambiente, e poi descrive il cuore della soluzione del problema, in modo tale che tu puoi usare questa soluzione più di un milione di volte, senza mai doverla rifare due volte" - Prof. Charles Alexander

## Why using design patterns ?

- I design patterns possono velocizzare il processo di sviluppo fornendo paradigmi di sviluppo testati
- I design patterns forniscono soluzioni generali, documentate in un format che non richiede specifiche legate ad un particolare problema
- I design patterns permettono agli sviluppatori di comunicare usando nomi conosciuti e ben compresi per le interazioni del software

## Design patterns classification

- I design patterns sono stati originariamente raggruppati nelle categorie Creational patterns, Structural patterns, e Behavioral patterns.
- DPs sono state descritte usando concetti di:
  - **delegation**: è il concetto di gestire un compito su un'altra parte del programma. Nella programmazione OO è usato per descrivere la situazione dove un oggetto sposta un impegno verso un altro oggetto, conosciuto come (wherein one object defers a task to another object, known as the delegate) 
  - **aggregation**: è un modo per combinare semplici oggetti o data types in più complessi. Compositoid objects sono spesso riferiti come una "ha una" relazione. Nella composizione, quando l'oggetto possedente è distrutto, così lo sono gli oggetti contenuti (when the owning object is destroyed, so are the contained objects). Nell'aggregazione, questo non è necessariamente vero.
  - **consultation**: è quando l'implementazione di un metodo di un oggetto consiste.. boh.. (is when an object's method implementation consists of a message send of the same message to another constituent object)

## Types of design patterns - 1

### Creational design patterns

- Questo design pattern tratta dell'istanziazione di classe. Questo modello può essere ulteriormente di viso in modelli di class-creation patterns e di object-creational patterns.
- Mentre i modelli class-creation patterns usano l'ereditarietà validamente nel processo di istanziazione, i modelli di object-creation usano la delega validamente per realizzare il lavoro.

# Types of design patterns - 2

## Structural design patterns


- Questi design patterns trattano della composizione Classe e Oggetto. Modelli strutturali di class-creation usano l'ereditarietà per comporre interfacce.
- Gli structural object-patterns forniscono i modi per comporre gli oggetti per ottenere una nuova funzionalità.

# Types of design patterns - 3

## Behavioral design patterns

- Questi design patterns riguardano tutta la comunicazione degli oggetti di classe. I behavioral patterns sono quei modelli che concernono in modo più specifico la comunicazione tra oggetti.

## “Gang of Four” Pattern Structure

- Gang of Four (GoF): Gamma, Johnson, Helm, Vlissides
- Gli autori del libro popolare “Design Patterns”
- Un pattern ha un **nome**
  - Ad esempio il Command pattern
- pattern documents, un **problema** ricorrente 
  - ad esempio, emettere richieste verso oggetti senza sapere in anticipo che cosa deve essere richiesto e quale oggetto
- Un pattern destrinse il cuore di una **soluzione**
  - ad esempio ruoli di classe, relazioni e interazioni
  - Importante: questo è differente dal descrivere un design
- Un pattern considera le **conseguenze** del suo uso
  - Trade-offs, unresolved forces, e altri patterns da usare

## What patterns are not

- Design patterns ...
- ... non sono ristretti all'OOP
- ... non sono principi euristici o astratti
- ... non sono soluzioni nuove e/o non testate
- ... non sono una soluzione specifica ad uno specifico problema
- ... non sono un “silver bullet”(proiettile d'argento)

## The iterator design pattern

- Abbiamo già visto un esempio di modello di design comportamentale (behavioural design pattern) gli iteratori di STL.
- L'idea base di un iteratore è che permette l'inversione di un contenitore (come un puntatore che si muove attraverso un array). Comunque per arrivare all'elemento successivo di un container non hai bisogno di sapere nulla su come il contenitore è costruito. Questo è il lavoro degli iteratori. Usando semplicemente le funzioni del membro fornito da un iteratore ti puoi muovere nell'ordine voluto dal container, dal primo elemento all'ultimo

## Programming idioms

- Gli idiomi sono soluzioni che capitano più volte nei problemi di programmazione comune
- Gli idiomi sono low-level patterns specifici per un linguaggio di programmazione. I design patterns sono di alto livello e indipendenti dal linguaggio.
- Durante l'implementazione cerchi gli idiomi. Durante il design cerchi i modelli.

# We have already seen them...

- include guard, RAII, const auto\_ptr, sono idiomi di programmazione
- Un altro esempio: la classe interfaccia
  - Se siamo più interessati all'ereditarietà dell'interfaccia piuttosto che all'ereditarietà dell'implementazione disegniamo l'interfaccia usando una classe composta solo da puri metodi pubblici virtuali.

## Idiom: interface class

In Java we would use an interface.

```
class Shape // An interface class
{
public:
 virtual ~Shape();
 virtual void move_x(int x) = 0;
 virtual void move_y(int y) = 0;
 virtual void draw() = 0;
//...
};

class Line : public Shape
{
public:
 virtual ~Line();
 virtual void move_x(int x); // implements move_x
 virtual void move_y(int y); // implements move_y
 virtual void draw(); // implements draw
private:
 point end_point_1, end_point_2;
//...
};
```

Remind the virtual destructor !

The classes that extend Shape are not dependent each other

## Adapter pattern

Class and Object Adapter

### *Adapter*

## Class and Object Adapter

- Riguarda il pattern che abbiamo già visto sulle slides dell'ereditarietà: la classe adapter. Vedi un'altra versione del pattern, l'object adapter.
- L'Adapter pattern converte l'interfaccia di una classe in un'altra interfaccia che i clienti si aspettano. L'adapter permette alle classe di lavorare insieme, classi che non potrebbero diversamente a causa delle interfacce incompatibili.

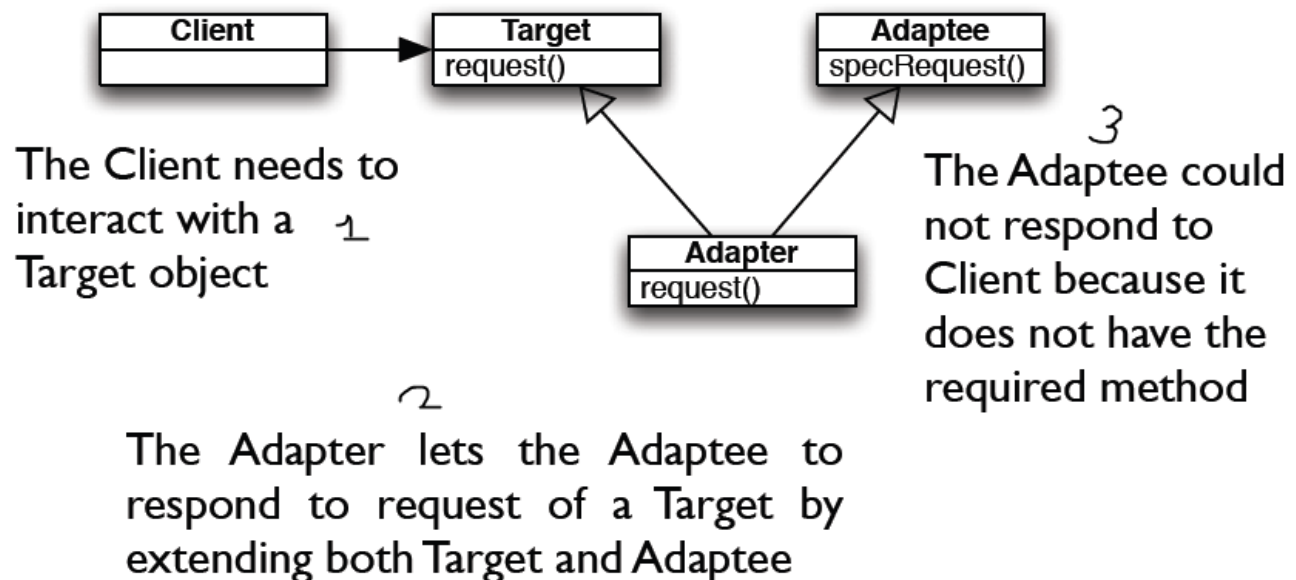
## Adapter pattern

- Problema
  - Avere un oggetto con un'interfaccia che è vicina, ma non esattamente ciò di cui abbiamo bisogno
- Context
  - Vuole riutilizzare una classe esistente
  - Non può mutare la sua interfaccia
  - Scomodo per una gerarchia estesa di classe più generalmente.
- Soluzione
  - Racchiudi una particolare classe o oggetto con l'interfaccia necessaria (due forme: class form and object forms)
- Conseguenze
  - L'implementazione che ti è data realizza l'interfaccia che vuoi.

## Class Adapter

- Una classe (Adapter) adatta l'interfaccia di un'altra classe (Adaptee) a un cliente, usando l'interfaccia descritta in una classe astratta (Target).
- Usa un'ereditarietà multipla insieme con una classe astratta, metodi virtuali e ereditarietà privata.

## Class Adapter UML class



- 1) Il cliente ha bisogno di interagire con un Target Object
- 2) L'adapter permette all'adaptee di rispondere alla richiesta di un Target estendendo sia il target che l'adaptee.
- 3) L'adaptee non potrebbe rispondere al cliente perché non ha il metodo richiesto

## Class Adapter example



```

class Adaptee {
public:
 getAlpha() {return alpha;};
 getRadius() {return radius;};
private:
 float alpha;
 float radius;
};

class Target {
public:
 virtual float getX() = 0;
 virtual float getY() = 0;
};

```

```

class Adapter : private Adaptee, public Target
{
public:
 virtual float getX();
 virtual float getY();
};

float Adapter::getX() {
 return Adaptee::getRadius()*cos
 (Adaptee::getAlpha());
}

float Adapter::getY() {
 return Adaptee::getRadius()*sin
 (Adaptee::getAlpha());
}

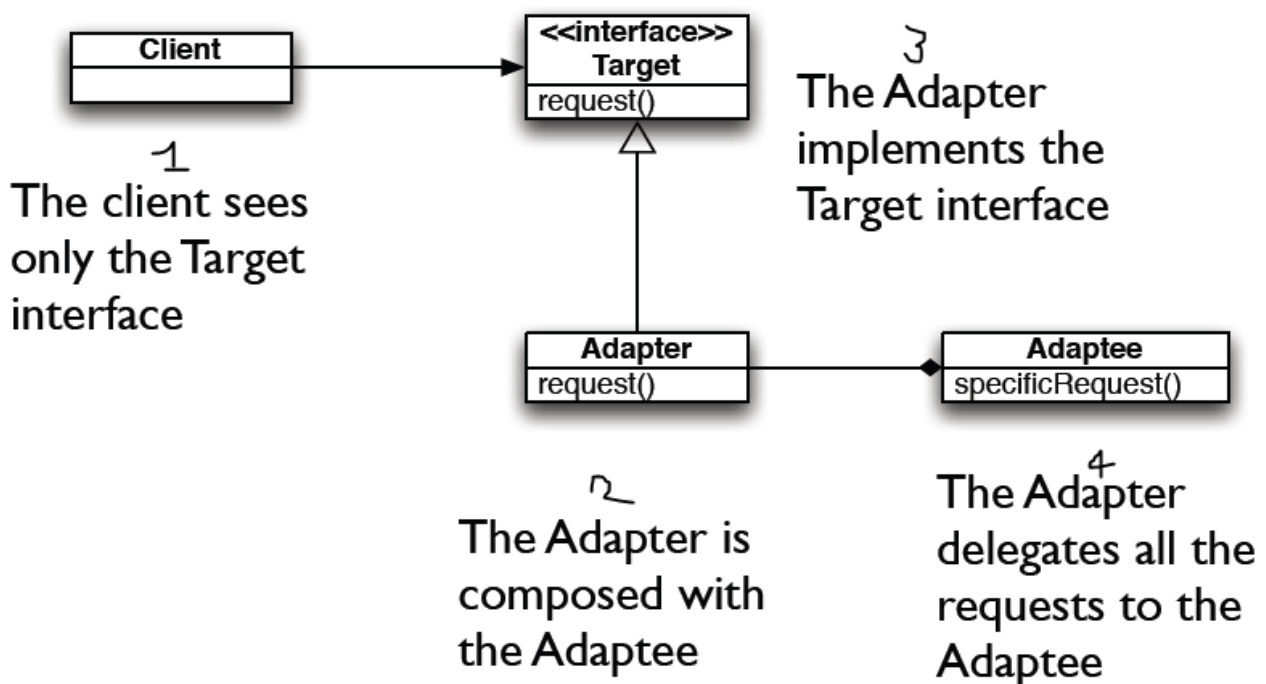
```

Il cliente non può accedere ai metodi dell'Adaptee perché l'Adapter li ha ottenuti usando l'ereditarietà privata.

## Object Adapter

- Non usa l'ereditarietà multipla ma usa la composizione per adattare la classe adaptee:
  - Può anche adattare sottoclassi dell'adaptee (diversamente dalla classe adapter)
  - Necessita della reimplementazione dell'adaptee e così richiede più codificazione (diversamente dalla classe adapter)
- L'adapter delega all'adaptee la richiesta dei clienti

## Object Adapter UML class Diagram



- 1) Il cliente vede solo l'interfaccia Target
- 2) L'adapter è composto con l'adaptee
- 3) L'adapter implementa l'interfaccia target

4) L'adaptee delega tutte le richieste all'adaptee

## Object Adapter example

```
class Duck {
 public: virtual ~Duck() = 0 { }
 public: virtual void fly() const = 0;
 public: virtual void quack() const = 0;
};

class MallardDuck : public Duck {
 public: void fly() const {
 cout << "I'm flying" << std::endl;
 }
 public: void quack() const {
 cout << "Quack" << std::endl;
 }
};

class Turkey {
 public: virtual ~Turkey() = 0 { }
 public: virtual void gobble() const = 0;
 public: virtual void fly() const = 0;
};

class WildTurkey : public Turkey {
 public: void fly() const {
 cout << "I'm flying a short
distance" << endl;
 }
 public: void gobble() const {
 cout << "Gobble gobble" << endl;
 }
};

class TurkeyAdapter : public Duck {
 private: const Turkey* _turkey;
 public: TurkeyAdapter(const Turkey* turkey) : _turkey(turkey) { }
 public: void fly() const {
 for(int i = 0; i < 5; i++) {
 _turkey->fly();
 }
 }
 public: void quack() const {
 _turkey->gobble();
 }
};

void testDuck(const Duck* duck) {
 duck->quack();
 duck->fly();
}

...
MallardDuck* duck = new MallardDuck();
WildTurkey* turkey = new WildTurkey();
Duck* turkeyAdapter = new TurkeyAdapter(turkey);
testDuck(duck);
testDuck(turkeyAdapter);
```

## Some patterns

- Observer - behavioral – Un modo per notificare il cambiamento a un numero di classi
- Factory - creational – Crea un esempio di parecchie famiglie di classi
- Singleton - creational – Una classe di cui solo un singolo esempio può esistere
- Adapter - structural – Mette insieme(Match) interfacce di classi diverse
- Facade - structural – Una singola classe che rappresenta un intero sottosistema
- Decorator - structural – Aggiunge responsabilità agli oggetti in modo dinamico

# Design pattern

## Observer

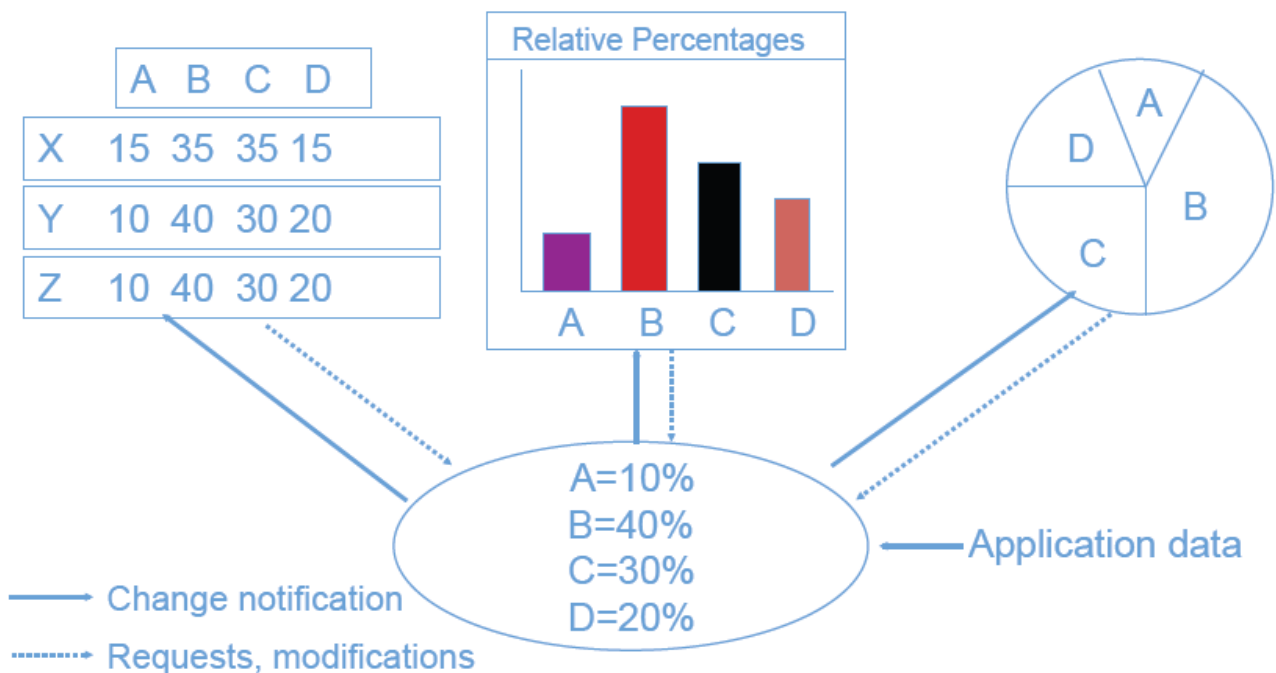
### Some motivations

- In molti programmi quando cambia lo stato di un oggetto, altri oggetti possono dover essere notificati
- Questo pattern risponde alla domanda: come fare per notificare quegli oggetti quando il soggetto cambia?
- E che succede se la lista di quegli oggetti cambia in tempo reale(run-time)?

### Some examples

- Esempio: quando si muove una macchina in un gioco
  - La graphics engine ha bisogno di sapere così può rendere di nuovo l'item.
  - Le routine nella computazione del traffic hanno bisogno di contare di nuovo il modello di traffic
  - Gli oggetti che l'auto contiene hanno bisogno di sapere pure che si stanno muovendo
- Un altro esempio: i dati in un foglio elettronico cambiano
  - Il display deve essere aggiornato
  - Possibilmente grafici multipli che utilizzano quei dati hanno bisogno di ridisegnarsi

### Another example



### Observer Pattern

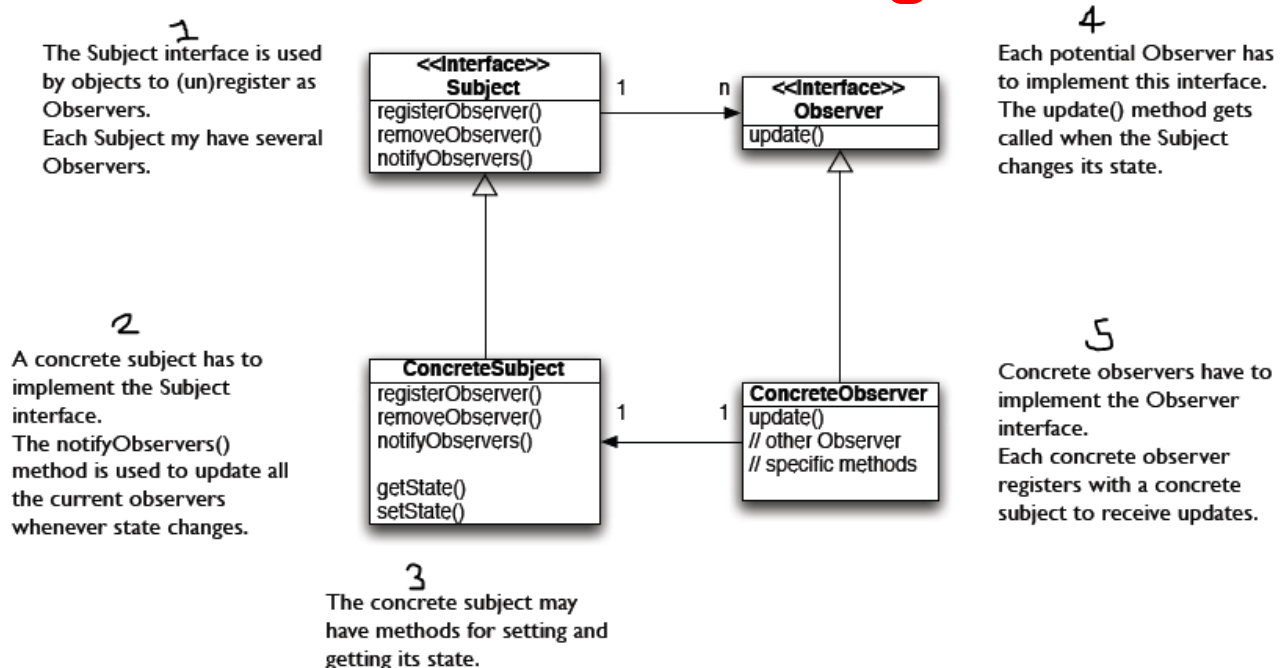
- Problema
  - Bisogno di aggiornare(update) oggetti multipli quando cambia lo stato di un oggetto(uno verso molte dipendenze)
- Context(contesto)
  - Oggetti multipli dipendono dallo stato di un oggetto

- Un insieme di oggetti dipendenti possono mutare in tempo reale
- Soluzione
  - Permettere agli oggetti dipendenti di registrar con oggetti di interesse , notificare loro di aggiornamenti(udates) quando lo stato cambia
- Conseguenze
  - Quando l'oggetto osservato cambia gli altri vengono notificati
  - Utile per la programmazione dell'interfaccia, alter applicazioni.

## Participants

- I partecipanti chiave in questo modello sono:
- Il Subject, che fornisce una interfaccia(virtual) per attaccare e staccare gli osservatori
- L' Observer, che definisce l'intefaccia che si aggiorna(virtuale) udating
- Il ConcreteSubject, che è la classe che eredita e stende o implement ail soggetto
- Il ConcreteObserver, che è la classe che eredita estende o implementa l'Observer
- Quest modello è anche conosciuto come dependents or publish-subscribe

## Observer UML class diagram



- 1) L'interfaccia soggetto è usata da oggetti per (non)registrare come osservatori. Ciascun soggetto può avere molteplici osservatori.
- 2) Un soggetto concreto deve implementare l'interfaccia soggetto per `notifyObservers()` è usato il metodo di aggiornare tutti gli osservatori correnti ogni qual volta lo stato cambia
- 3) Il coggett oconcreto può avere metodi per impostare e ottenere il suo stato
- 4) Ciascun osservatore potenziale deve implementare questga interfaccia. Il metodo `update()` è chiamato quando iul soggetto cambia il suo stato
- 5) Gli osservatori concreti devono implementare l'interfaccia dell'osservatore. Ogni osservatore concreto registra con un soggetto concreto per ricevere aggiornamenti.

## Some interesting points

- Nel modello dell'osservatore quando lo stato di un oggetto cambia tutti i suoi dipendenti sono notificati:
  - il soggetto è l'unico proprietario di quei dati, gli osservatori sono dipendenti dal soggetto per aggiornarli quando i dati cambiano
- è un disegno più pulito che permette a molti oggetti di controllare gli stessi dati

## Loose coupling

- L'Observer pattern fornisce un modello in cui i soggetti e gli osservatori sono accoppiati debolmente (minimizzando l'interdipendenza tra gli oggetti):
  - l'unica cosa che il soggetto conosce circa un osservatore è che esso implementa un'interfaccia
  - gli osservatori possono essere aggiunti o rimossi in qualsiasi momento (anche in run-time)
  - Non c'è bisogno di modificare il soggetto per aggiungere nuovi tipi di osservatori (essi hanno solo bisogno di implementare l'interfaccia)
  - mutamenti verso il soggetto o gli osservatori non influenzano l'altro (fino a quando essi implementano l'interfaccia richiesta)

## Observer example

```
class Subject {
 protected: virtual ~Subject() = 0;
};


 public: virtual void
registerObserver(Observer* o) = 0;

 public: virtual void removeObserver
(Observer* o) = 0;

 public: virtual void
notifyObservers() const = 0;
};
```

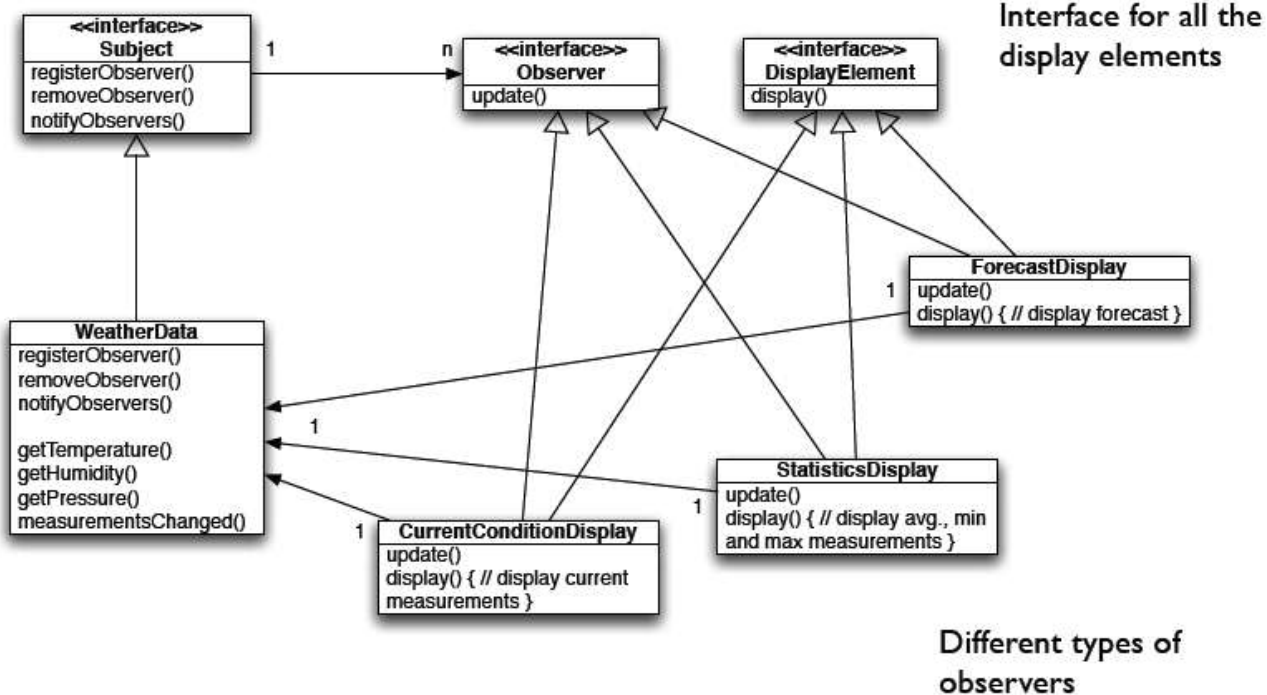
```
class Observer {
 protected: virtual ~Observer() = 0
 { };

 public: virtual void update(float
temp, float humidity, float pressure)
= 0;
};
```



The update method gets the state values from the subject: they'll change depending on the subject, in this example is a weather station

1) Il metodo update ottiene i valori di stato dal soggetto: essi cambieranno dipendendo dal soggetto, in questo esempio è una stazione meteorologica.



```

class DisplayElement {
public: virtual void display() const = 0;
protected: virtual ~DisplayElement() = 0 {
};
};

```

### Implementing the Subject Interface

```

class WeatherData : public Subject {
private: list< Observer* > _observers;
private: float _temperature;
private: float _humidity;
private: float _pressure;
public: WeatherData() : _temperature(0.0),
_humidity(0.0), _pressure(0.0) { }
public: void
registerObserver(Observer* o) {
_observers.push_back(o);
}
public: void
removeObserver(Observer* o) {
_observers.remove(o);
}
public: void notifyObservers() const {
for(list< Observer* >::iterator itr =
_observers.begin(); _observers.end() != itr; ++itr) {
Observer* observer = *itr;
observer->update(_temperature, _humidity,
_pressure);
}
}
}

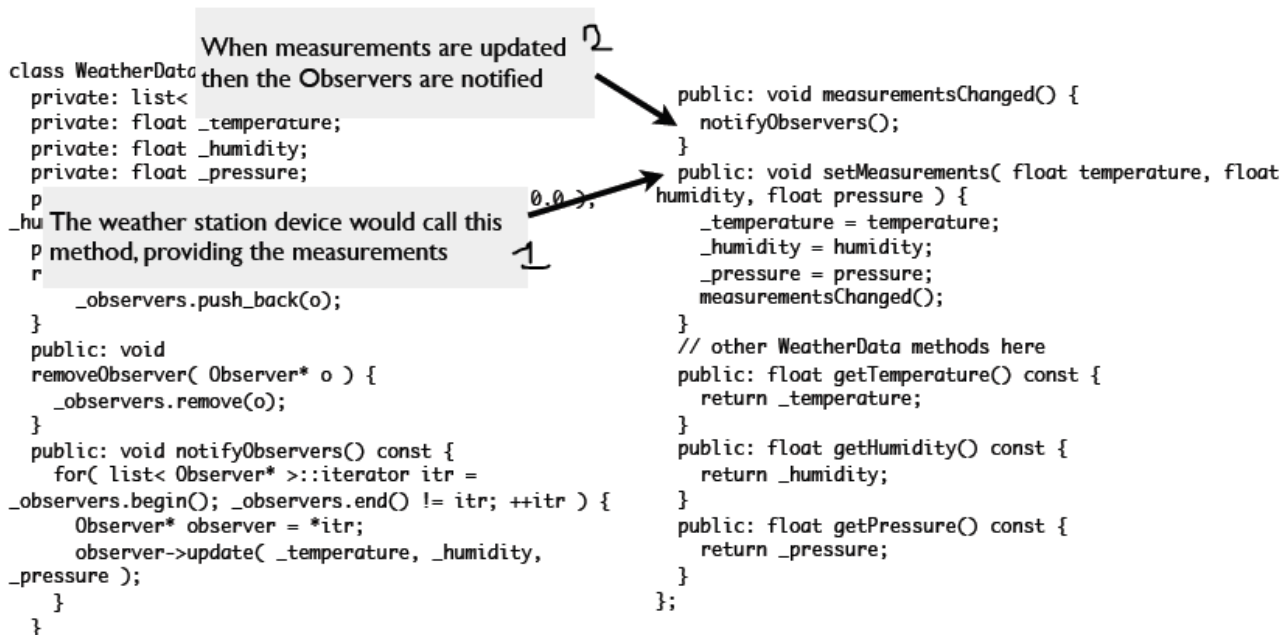
```

```

public: void measurementsChanged() {
notifyObservers();
}
public: void setMeasurements(float temperature, float
humidity, float pressure) {
_temperature = temperature;
_humidity = humidity;
_pressure = pressure;
measurementsChanged();
}
// other WeatherData methods here
public: float getTemperature() const {
return _temperature;
}
public: float getHumidity() const {
return _humidity;
}
public: float getPressure() const {
return _pressure;
}
}
};

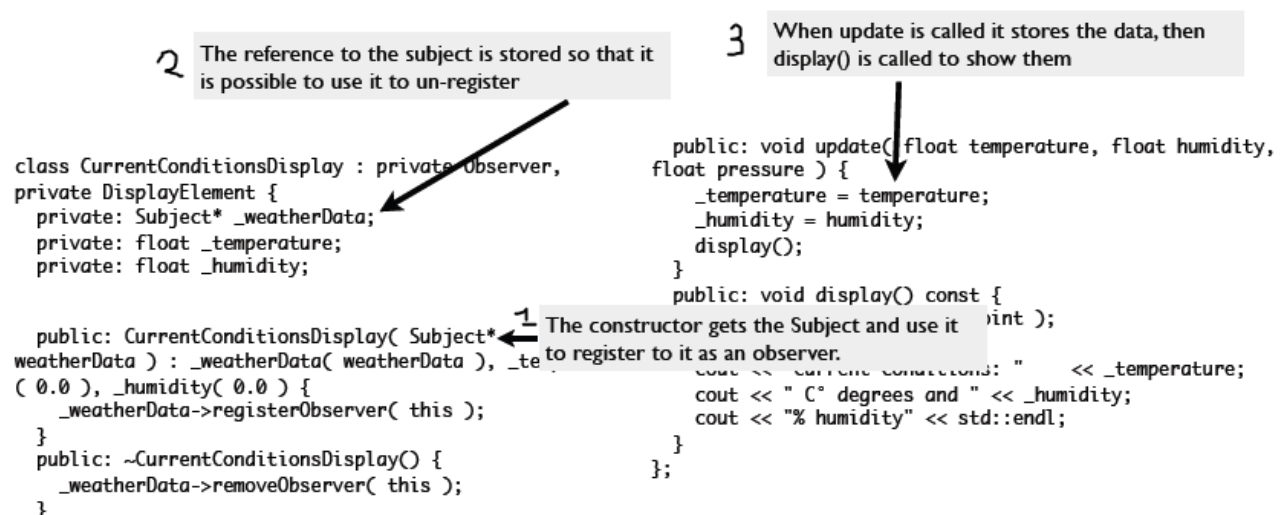
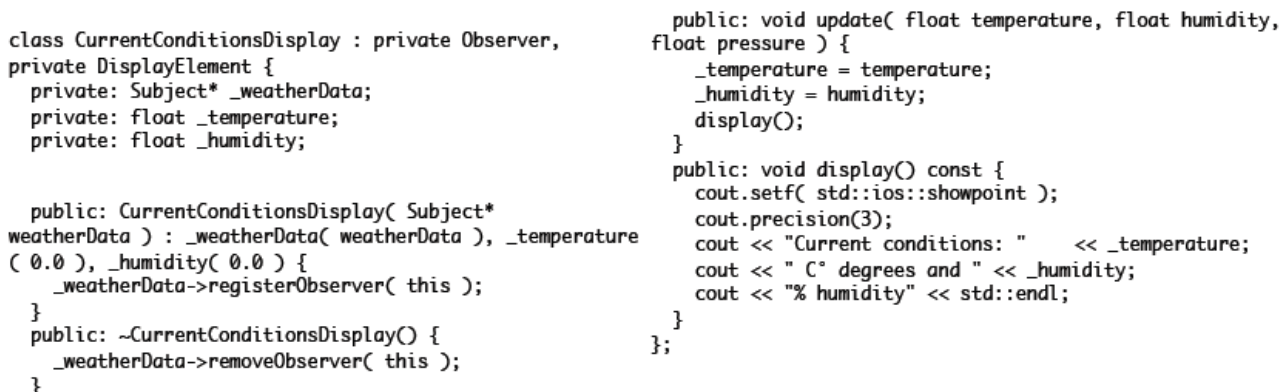
```





- 1) L'espedito della stazione metereologica chiamerebbe questo metodo, fornendo le misure
- 2) Quando le misure sono aggiornate allora sono notificati gli osservatori

## Implementing the DisplayElement interface



- 1) Il costruttore ottiene il soggetto e lo usa per registrar a esso come un osservatore
- 2) Il riferimento al soggetto è immagazzinato cosicchè sia possibile usarlo per una non-registrazione
- 3) Quando si richiama l'aggiornamento esso immagazzina i dati poi si chiama display() per mostrarle

## Test the pattern

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

```
 WeatherData weatherData;
```

Create the  
concrete subject

Create the displays  
and pass the  
concrete subject

```
 CurrentConditionsDisplay currentDisplay(&weatherData);
```

```
 StatisticsDisplay statisticsDisplay(&weatherData);
```

```
 ForecastDisplay forecastDisplay(&weatherData);
```

```
 weatherData.setMeasurements(80, 65, 30.4f);
```

```
 weatherData.setMeasurements(82, 70, 29.2f);
```

```
 weatherData.setMeasurements(78, 90, 29.2f);
```

Simulate  
measurements

```
 return 0;
```

```
}
```

## Push or pull ?

- Nell'implementazione precedente lo stato è spinto dal soggetto all'osservatore
- Se il soggetto qualche metodo getter pubblico, l'observer può trarne lo stato quando esso è notificato di un cambiamento (Observer may pull the state when it is notified of a change)
- Se lo stato è modificato non c'è bisogno di modificare l'update(), cambiare i metodi getter

## Pull example



The update() method in the Observer interface now is decoupled from the state of the concrete observer

```
public: void update() {
 _temperature = _weatherData->getTemperature();
 _humidity = _weatherData->getHumidity();
 display();
}
```

We just have to change the implementation of the update() in the concrete observers

- 1) Il metodo update() nell'interfaccia observer ora è disaccoppiato dallo stato dell'osservatore concreto
- 2) Dobbiamo semplicemente cambiare l'implementazione dell'update() negli osservatori concreti

## Flexible updating

•Per avere più flessibilità nell'aggiornamento degli observers, il soggetto può avere un metodo setChanged() che permette al notifyObservers() di avviare l'update()

```
setChanged() {
 _changed = true;
}
public: void notifyObservers() const {
 if(_changed) {
 for(list< Observer* >::iterator itr = _observers.begin();
_observers.end() != itr; ++itr) {
 Observer* observer = *itr;
 observer->update(_temperature, _humidity, _pressure);
 }
 _changed = false;
 }
}
```

- 1) chiama il metodo setChanged() quando lo stato è cambiato abbastanza per dirlo agli osservatori
- 2) Controlla la bandiera per cominciare le notifiche

## The observer pattern and

# GUIs

- L'observer pattern è molto spesso associato con il paradigma model-view-controller (MVC).
- Nell' MVC, l'observer pattern è usato per creare un'accoppiamento debole tra il modello e il view (between the model and the view)

Tipicamente, una modifica nel modello avvia la notificazione dei model observers che in realtà sono le views

# Design pattern

## **Factory**

## **Some motivations**

- Considera un kit di toolkit per una user interface per supportare standard multipli look-and-feel(standards):
  - for portability an application must not hard code its widgets for one look and feel.
- L'uso del factory pattern permette:
  - generazione di esempi diversi di una classe, usando gli stessi tipi di parametri.
  - di accrescere la flessibilità del sistema – il codice può usare un oggetto di un'interfaccia (tipo) w/o sapendo a quale classe implementazione esso appartiene

## **Factory pattern**

- Il problema
  - Vuoi una classe per creare una classe collegata(related) polimorficamente
- Context
  - Ogni classe conosce a quale versione della classe collegata essa dovrebbe creare
- Soluzione
  - Dichiarare un metodo astratto di cui le classi derivate fanno l'override(declare abstract method that derived classes override)
- Conseguenze
  - Type created matches type(s) it's used with

## **Factory pattern**

- Factory: una classe il cui solo lavoro è di creare facilmente e restituire istanze di altre classi:
  - è un creational pattern; rende più facile costruire oggetti complessi, creare oggetti individuali in situazioni in cui il costruttore da solo è inadeguato
  - invece di chiamare un costruttore, usa un metodo static in una factory class per impostare(set up)l'oggetto

## **Pattern intent**

- Definisci una interfaccia per creare un oggetto ma lascia che le sottoclassi decidano quali classi instanziare
- Lascia che una classe posponga l'instanziazione alle sottoclassi
- Vedremo alcune variazioni del tema della factory

## **The problem with *new***

- In alcuni casi non c'è bisogno di instanziare da vicino classi collegate(related) (ovvero derivate da una base comune) che dipendono da alcuni criteri, ad esempio:

```
• Duck duck;
 if (picnic) {
 duck = new MallardDuck();
 } else if(decorating) {
 duck = new DecoyDuck();
 } else if(inBathTub) {
 duck = new RubberDuck();
 }
```



What happens if we  
have to add another  
duck ?

## Simple Factory

### Goal

- Incapulare la creazione di una classe collegata in una classe: dovremo modificare solamente quella classe quando cambia l'implementazione.
- La factory gestirà i dettagli della creazione dell'oggetto
- La Simple Factory non è un vero design pattern, è più un idiomma di programmazione.

## Design Patterns and Programming Idioms

- Secondo Alexander, un pattern:
  - Descrive un problema ricorrente
  - Descrive il cuore di una soluzione
  - è capace di generare molti designs distinti
- Un idiomma più ristretto
  - Comunque descrive un problema ricorrente
  - Fornisce una soluzione più specifica, con meno variazioni
  - Si applica solo a un contest più stretto
  - Ad esempio il linguaggio C++

## Simple Factory example

```

Pizza* orderPizza(string type) {
 Pizza* pizza = 0;

 if (type.compare("4cheeses") == 0)
 pizza = new FourCheesesPizza();
 else if (type.compare("zucchini") == 0)
 pizza = new ZucchiniPizza();
 else if (type.compare("ham_mushrooms") == 0)
 pizza = new HamMushroomsPizza();

 pizza->prepare();
 pizza->bake();
 pizza->box();
 return pizza;
}

```

Adding new types of pizzas will require to change this code

← This part of code will remain the same

## Encapsulating object creation

```

class SimplePizzaFactory {
public: Pizza* createPizza(string type) const {

 Pizza* pizza = 0;

 if (type.compare("4cheeses") == 0)
 pizza = new FourCheesesPizza();
 else if (type.compare("zucchini") == 0)
 pizza = new ZucchiniPizza();
 else if (type.compare("ham_mushrooms") == 0)
 pizza = new HamMushroomsPizza();

 return pizza;
}
};

```

## Using the Simple Factory

```

class PizzaStore {
 private: SimplePizzaFactory* _factory;

 public: PizzaStore(SimplePizzaFactory* factory)
 _factory(factory) { }

 public: Pizza* orderPizza(string type) {
 Pizza* pizza;
 pizza = _factory->createPizza(type);
 pizza->prepare();
 pizza->bake();
 pizza->box();

 return pizza;
 }
};

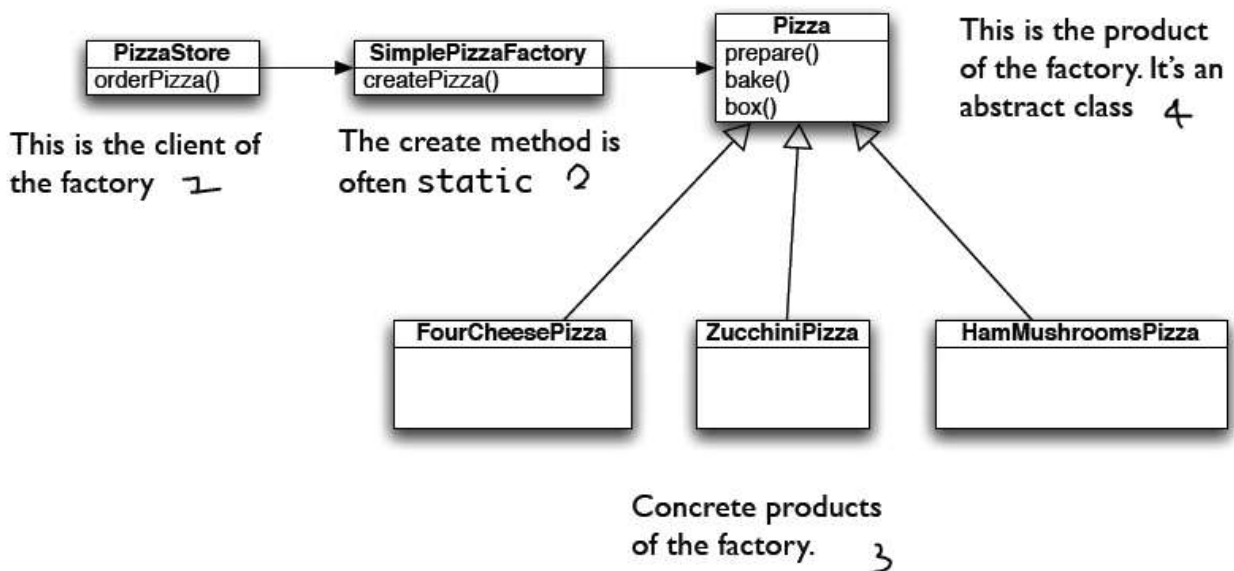
```

Hold a reference to a Simple Factory

Get the factory passed in the constructor

Use the factory with the create() method instead of using a new

## Simple Factory UML class Diagram



- 1) Questo è il cliente della factory
- 2) Il metodo creato è spesso statico
- 3) Prodotti reali della factory
- 4) Questo è il prodotto della factory. È una classe astratta

## Factory Method

Class creational

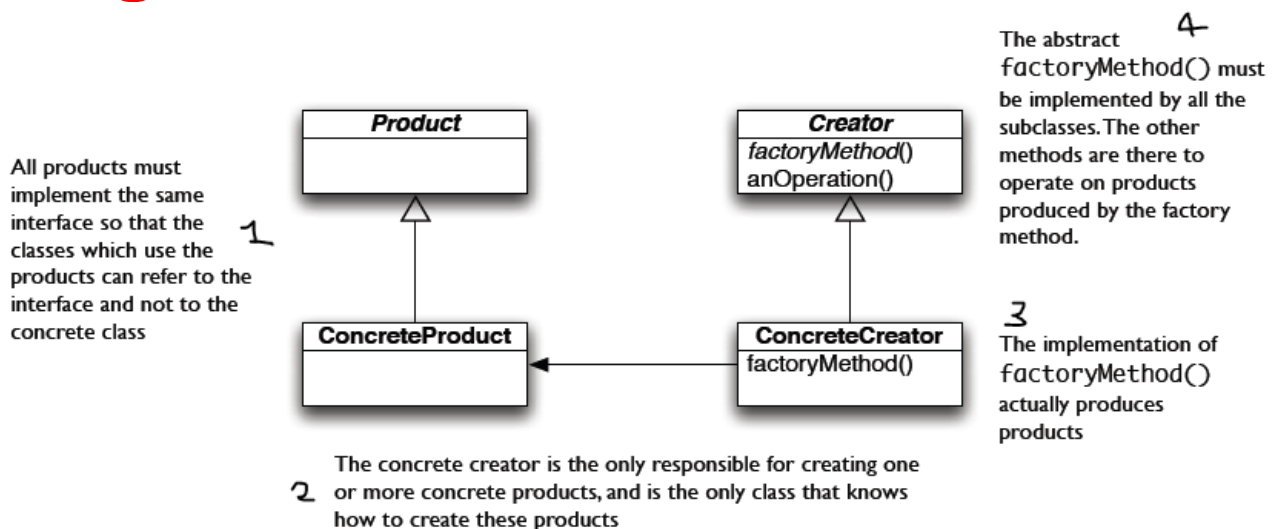
# Some motivations

- Usa il Factory Method pattern quando:
  - una classe non riesce ad anticipare la classe degli oggetti che essa deve creare
  - una classe vuole le sue sottoclassi per specificare l'oggetto che essa crea
  - le classi delegano la responsabilità a molteplici sottoclassi di aiuto e tu vuoi localizzare la conoscenza di quel sottoclasse di aiuto è la delegata.

## Factory Method

- Il problema
  - Vuoi una classe per creare una classe collegata(related) polimorficamente
- Context
  - Ogni classes conosce a quale versione della classe collegata essa dovrebbe creare
- Soluzione
  - Dichiarare un metodo astratto di cui le classi derivate fanno l'override(declare abstract method that derived classes override)
- Conseguenze
  - Type created matches type(s) it's used with

## Factory method UML class Diagram

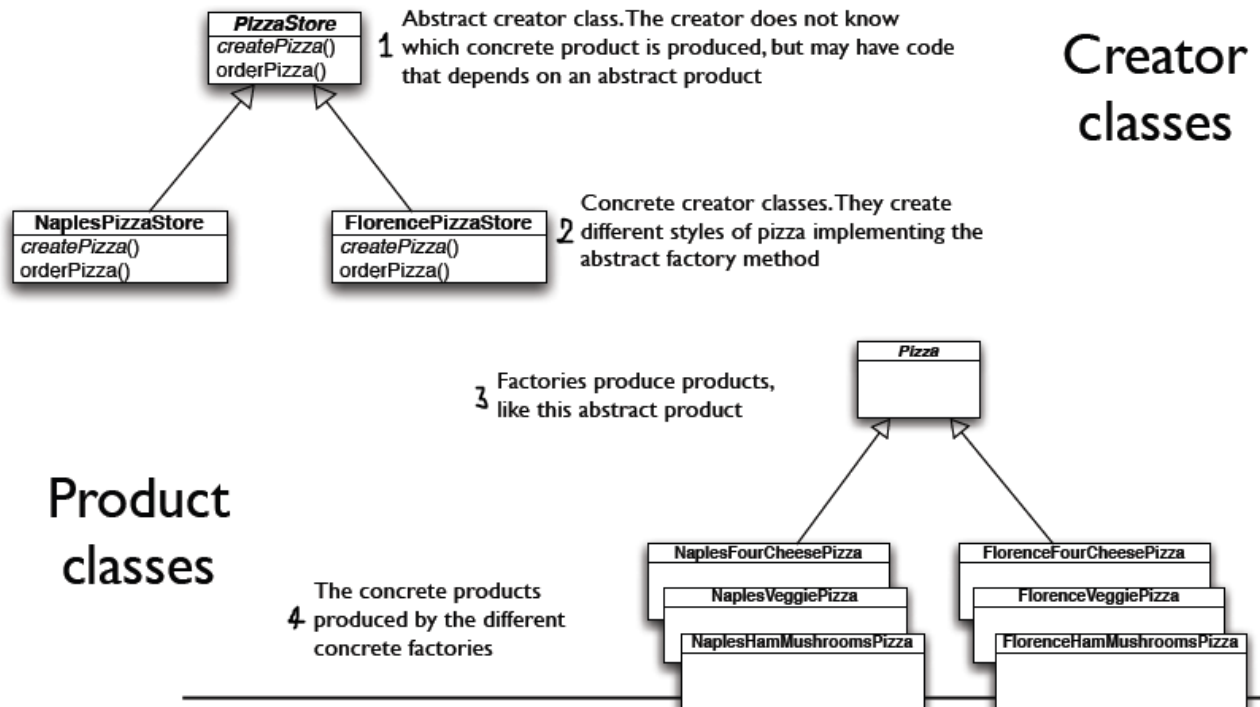


- 1) Tutti i prodotti devono implementare la stessa interfaccia cosicchè le calssi che usano i prodotti possano riferirsi all'interfaccia e non alla classe reale
- 2) Il creatore concreto è il solo responsabile nella creazione di uno o più prodotti ed è la sola classe che conosce come creare questi prodotti
- 3) L'implementazione del `factoryMethod()` in realtà produce prodotti
- 4) Il `factoryMethod()` astratto deve essere implementato da tutte le sottoclassi. Gli altri metodi sono lì per operare su prodotti che sono stati prodotti dal factory method.

Definisci un interfaccia per creare un oggetto ma lascia che le sottoclassi decidano quali classi instanziare. Il factory method di posticipare l'instanziazione alle sottoclassi.

## Factory Method example

# UML class diagram



- 1) L'abstract creator class. Il creatore non sa quale prodotto concreto è prodotto ma può avere un codice che dipende da un prodotto astratto.
- 2) Concrete creator classes. Creano differenti stili di pizza implementando lo abstract factory method
- 3) Le factories producono prodotti, come questo prodotto astratto
- 4) I prodotti concreti prodotti da diverse concrete factories.

## Participants

- **Product**: definisce l'interfaccia degli oggetti che il factory method crea
- **ConcreteProduct**: implementa l'interfaccia del prodotto
- **Creator**: dichiara il metodo della factory che restituisce un oggetto del tipo prodotto. Il creatore può anche definire un'implementazione default del factory method che restituisce un oggetto default **ConcreteProduct**. Può chiamare il metodo factory per creare un oggetto **Prodotto**.
- **ConcreteCreator**: fa l'override del factory method per restituire un'istanza di un **ConcreteProduct**

## Factory Method example



```

class PizzaStore {
protected: PizzaStore() { }
public: virtual ~PizzaStore() = 0 { }

public: Pizza* orderPizza(string type) const {
 Pizza* pizza;

 pizza = createPizza(type);

 cout << "- Making a " << pizza->getName() << " -" << endl;
 pizza->prepare();
 pizza->bake();
 pizza->cut();
 pizza->box();
 return pizza;
}

public: virtual Pizza* createPizza(string type) const = 0;
};

```

```

class PizzaStore {
protected: PizzaStore() { }
public: virtual ~PizzaStore() = 0 { }

public: Pizza* orderPizza(string type) const {
 Pizza* pizza;

 pizza = createPizza(type);

 cout << "- Making a " << pizza->getName() << " -" << endl;
 pizza->prepare();
 pizza->bake();
 pizza->cut();
 pizza->box();
 return pizza;
}

public: virtual Pizza* createPizza(string type) const = 0;
};

```

The createPizza() is back into the PizzaStore object rather than in a factory object

The factory object has been moved to this method

The factory method is abstract in the PizzaStore

```

class NaplesPizzaStore : public PizzaStore {

 public: Pizza* createPizza(string type) const {
 if(type.compare("fourcheese") == 0) {
 return new NaplesStyleFourCheesePizza();
 } else if(type.compare("veggie") == 0) {
 return new NaplesStyleVeggiePizza();
 } else if(type.compare("clam") == 0) {
 return new NaplesStyleClamPizza();
 } else if(type.compare("hammushrooms") == 0) {
 return new NaplesStyleHamMushroomsPizza();
 } else return 0;
 }
};

```

```

class NaplesPizzaStore : public PizzaStore {

```

```

 public: Pizza* createPizza(
 if(type.compare("fourcheese") == 0) {
 return new NaplesStyleFourCheesePizza();
 } else if(type.compare("veggie") == 0) {
 return new NaplesStyleVeggiePizza();
 } else if(type.compare("clam") == 0) {
 return new NaplesStyleClamPizza();
 } else if(type.compare("hammushrooms") == 0) {
 return new NaplesStyleHamMushroomsPizza();
 } else return 0;
 }
};

```

The createPizza() of the Naples pizza store ensures that pizzas are created as in Naples: thick, large crust and using only buffalo mozzarella cheese

Each subclass of PizzaStore overrides the abstract createPizza() method, while all subclasses use the orderPizza() method defined in PizzaStore.

## Decoupling

- Il `PizzaStore::orderPizza()` è definito nella classe astratta `PizzaStore`, non nelle sottoclassi: il metodo non sa quale sottoclasse sta gestendo il codice e facendo le pizze.
- è disaccoppiato da quel codice
- Quando `orderPizza()` chiama `createPizza()` una delle sottoclassi è chiamata in azione dipendendo dalla sottoclasse `PizzaStore`
- non è una decisione in real time da parte della sottoclasse

# The factory method

- Il factory method gestisce la creazione dell'oggetto e lo incapsula in una sottoclasse. Questo disaccoppia il codice cliente nella superclasse(ad esempio un codice come orderPizza()) dalla creazione dell'oggetto nella sottoclasse.
- Il metodo factory deve essere virtuale e possibilmente anche puramente virtuale(ma un'implementazione default può essere fornita, per ottenere flessibilità): sottoclassi possono override come esso sono create
- Il factory method può essere parametrizzato(oppure no) per scegliere tra le variazioni del prodotto(ad esempio utile per la de serializzazione)

## How to get a pizza

- Get a pizza store:

```
PizzaStore* mergellinaStore = new NaplesPizzaStore();
```

- Take an order:

```
mergellinaStore->orderPizza("veggie");
```

- The orderPizza() method calls the createPizza() method implemented in the subclass:

```
Pizza* pizza = createPizza("veggie");
```

- The orderPizza() finished preparing it:

```
pizza->prepare();
```

```
pizza->bake();
```

```
...
```

## Implementing pizzas

```
class Pizza {
protected: string _name;
protected: string _dough;
protected: string _sauce;
protected: list< string > _toppings;
protected: Pizza() { }
public: virtual ~Pizza() = 0 { }
public: virtual void prepare() const {
 cout << "Preparing " << _name.c_str() << endl;
 cout << "Tossing dough..." << endl;
 cout << "Adding sauce..." << endl;
 cout << "Adding toppings: " << endl;
 for(list< string >::iterator itr = _toppings.begin();
 _toppings.end() != itr; ++itr) {
 cout << " " << itr->c_str() << endl;
 }
}
public: virtual void bake() const {
 cout << "Bake for 25 minutes at 350" << endl;
}
// void bake(); void cut(); void box(); string getName(); ...
```

## Abstract class (it has abstract methods)

```
class Pizza {
protected: string _name;
protected: string _dough;
protected: string _sauce;
protected: list< string > _toppings;
protected: Pizza() { }
public: virtual ~Pizza() = 0 { }
public: virtual void prepare() const {
 cout << "Preparing " << _name.c_str() << endl;
 cout << "Tossing do
 cout << "Adding sau
 cout << "Adding top
 for(list< string >
 _toppings.end(
 cout << " " <<
 }
}
public: virtual void bake() const {
 cout << "Bake for 25 minutes at 350" << endl;
}
// void bake(); void cut(); void box(); string getName(); ...
}
```

The class provides some basic default methods for preparing, baking, cutting,... They are virtual and can be overridden by the subclasses

```
class NaplesStyleVeggiePizza : public Pizza {

public: NaplesStyleVeggiePizza() {

 _name = "Naples Style Veggie Pizza";
 _dough = "Thick Crust Dough";
 _sauce = "Marinara Sauce";

 _toppings.push_back("Buffalo Mozzarella Cheese");

 _toppings.push_back("Garlic");
 _toppings.push_back("Onion");
 _toppings.push_back("Mushrooms");
 _toppings.push_back("Friarelli");

}

public: virtual void bake() const {
 cout << "Bake for 20 minutes at 350" << endl;
}
};
```

```
class NaplesStyleVeggiePizza : public Pizza {
```

```
public: NaplesStyleVeggiePizza() {
```

```
 _name = "Naples Style Veggie Pizza";
```

```
 _dough = "Thick Crust Dough";
```

```
 _sauce = "Marinara";
```

```
 _toppings.push_back(
```

```
 _toppings.push_back(
```

```
 _toppings.push_back("Buffalo",
```

```
 _toppings.push_back("Mushrooms");
```

```
 _toppings.push_back("Friarelli");
```

```
}
```

```
public: virtual void bake() const {
```

```
 cout << "Bake for 20 minutes at 350" << endl;
```

```
}
```

```
};
```

The Naples style pizza has its thick crust, marinara sauce, *friarelli* veggie and uses buffalo mozzarella cheese

The Naples style pizza is baked less time, to make a soft crust

## Putting everything together

```
PizzaStore* mergellinaStore = new NaplesPizzaStore();
```

```
Pizza* pizza = mergellinaStore->orderPizza("veggie");
```

Questo approccio è utile anche se c'è solo un creatore concreto giacchè il metodo factory disaccoppia l'implementazione del prodotto dal suo uso.

Il metodo factory e il creatore non hanno bisogno di essere astratti, essi possono fornire una qualche implementazione base

L'implementazione di ciascun store concreto somiglia alla simple factory, ma in questo precedente approccio la factory è un altro oggetto composto con PizzaStore, ecco una sottoclasse che si estende ad una classe astratta

- non è una soluzione in un sol colpo, stiamo usando una frame work che permette alle sottoclassi di decidere quale implementazione sarà usata
- Il factory method può anche cambiare i prodotti creati: è più flessibile

## Lazy initialization

- Il costruttore semplicemente inizializza il prodotto a 0, la creazione è delegata al metodo, la creazione è delegata a metodo accessorio/di accesso (controlla pure il metodo Singleton):

```
class Creator {
```

```
public: Creator() { _product = 0; };
```

```
public: Product* getProduct();
```

```
protected: virtual Product* createProduct();
```

```
private: Product* _product;
```

```
}
```

```
Product* Creator::getProduct() {
```

```
if (_product == 0) {
```

```
_product = createProduct();
```

```
}
```

```
return _product;
}
```

# Abstract Factory

Object creational

## Motivation

- Considera un kit di toolkit per una user interface per supportare standard multipli look-and-feel(standards):
  - for portability an application must not hard code its widgets for one look and feel.
- Come disegnare l'applicazione cosicchè incorporando nuove richieste look and feel sarà facile?

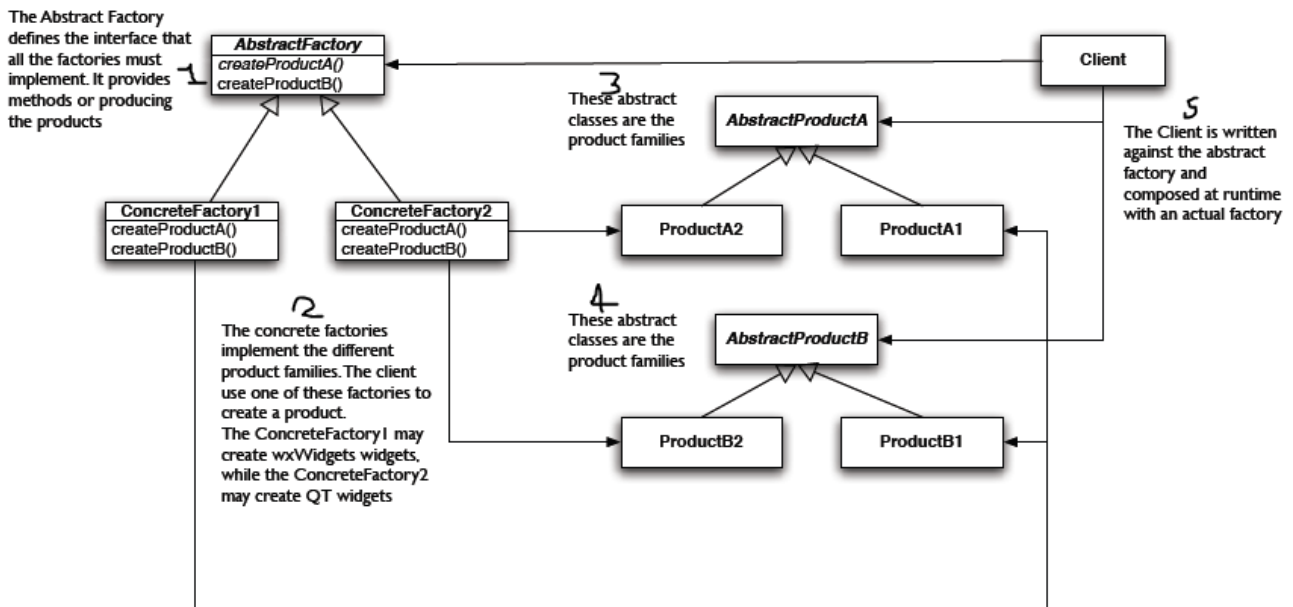
## Solution

- Definire una classe astratta **WidgetFactory**.
- Questa classe dichiara una interfaccia per creare differenti generi di widgets
- C'è una sola classe astratta per ciascun genere di Widgets e sottoclassi concrete implementano widget per standard diversi.
- **WidgetFactory** offre un'operazione per restituire un nuovo oggetto widget per ciascuna classe astratta widget. I clienti chiamano queste operazioni per ottenere istanza di widget senza essere consapevoli delle classi concrete che essi usano.

## Intent and applicability

- Fornisce un'interfaccia per creare famiglie creatrici di oggetti collegati o dipendenti w/o specificanti le loro classi concrete
- Questo modello può essere applicato quando:
  - un sistema dovrebbe essere indipendente di come i suoi prodotti sono creati composti e rappresentati
  - un sistema dovrebbe essere configurato con uno o più famiglie multiple di prodotti
  - una famiglia di oggetti di prodotti collegati e disegnata per essere usata insieme
  - è necessario fornire una libreria di classe dei prodotti che riveli le loro interfacce e non la loro implementazione.

## Abstract Factory UML class Diagram



## Participants

- **AbstractFactory**: dichiara un'interfaccia per le operazioni che crea abstract product objects
- **ConcreteFactory**: implementa le operazioni per creare concrete product objects
- **AbstractProduct**: dichiara un'interfaccia per un tipo di product object
- **ConcreteProduct**: definisce un prodotto per  
defines a product to be object  
created by the corresponding concrete factory,  
implementing the AbstractProduct interface
- **Client**: usa solo le interfacce create dall'AbstractXXX classes

## Collaborations

- Normalmente una una singola istanza di una classe ConcreteFactory è creata in run-time. Questa factory crea oggetti che hanno una particolare implementazione, per creare differenti oggetti usa factory differenti. Ciò promuove  
This promotes  
consistency among products: products of a  
whole family are created.
- **AbstractFactory** defers creation to the ConcreteFactory classes. It insulates the client from implementation classes.

## Implementation

- An application typically needs only one instance of a factory: these are implemented using the Singleton pattern
- Often the concrete factories are built using the Factory Method pattern for each product
- The AbstractFactory usually defines a different operation for each kind of product; these products are encoded in the operation signatures, thus adding a new kind of product requires changing the interface.



# Abstract Factory: example

```
// Abstract Factory
class PizzaIngredientFactory {
public:
 virtual Dough* createDough() const = 0;
 virtual Sauce* createSauce() const = 0;
 virtual Cheese* createCheese() const =
0;
 virtual std::vector< Veggies* >
 createVeggies() const = 0;
 virtual Clams* createClam() const = 0;
 virtual ~PizzaIngredientFactory() = 0 {}
};
```

```
// Abstract Factory
class PizzaIngredientFactory {
public:
 virtual Dough* createDough() const = 0;
 virtual Sauce* createSauce() const = 0;
 virtual Cheese* createCheese() const =
0;
 virtual std::vector< Veggies* >
 createVeggies() const = 0;
 virtual Clams* createClam() const = 0;
 virtual ~PizzaIngredientFactory() = 0 {}
};
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
 public: Dough* createDough() const {
 return new ThickCrustDough();
 }
 public: Sauce* createSauce() const {
 return new MarinaraSauce();
 }
 public: Cheese* createCheese() const {
 return new BuffaloMozzarellaCheese();
 }
 public: std::vector< Veggies* >
 createVeggies() const {
 std::vector< Veggies* > veggies;
 veggies.push_back(new Friarelli());
 veggies.push_back(new Onion());
 veggies.push_back(new Mushroom());
 veggies.push_back(new RedPepper());
 return veggies;
 }
 public: Clams* createClam() const {
 return new FreshClams();
 }
};
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
 public: Dough* createDough() const {
 return new ThickCrustDough();
 }
};
```

We have many classes:  
one for each ingredient.  
If there's need for a  
common functionality in  
all the factories  
implement a method  
here.

```
 return veggies;
 }
 public: Clams* createClam() const {
 return new FreshClams();
 }
};
```



```
// We are creating a
// specific version of
// ingredient for each
// factory.
// Some ingredients may be
// shared by different
// factories, though.
```

```
class NaplesPizzaIngredientFactory :
public PizzaIngredientFactory {
 public: Dough* createDough() const {
 return new ThickCrustDough();
 }
 public: Sauce* createSauce() const {
 return new MarinaraSauce();
 }
 public: Cheese* createCheese() const {
 return new BuffaloMozzarellaCheese();
 }
 public: std::vector< Veggies* >
 createVeggies() const {
 std::vector< Veggies* > veggies;
 veggies.push_back(new Friarelli());
 veggies.push_back(new Onion());
 veggies.push_back(new Mushroom());
 veggies.push_back(new RedPepper());
 return veggies;
 }
 public: Clams* createClam() const {
 return new FreshClams();
 }
};
```

## Abstract Factory: example

```
class Pizza {
private: std::string _name;
protected:
 Dough* _dough;
 Sauce* _sauce;
 std::vector< Veggies* > _veggies;
 Cheese* _cheese;
 Clams* _clam;
 Pizza() { }
public: virtual void prepare() const = 0;
 virtual ~Pizza() {
 for(std::vector< Veggies* >::iterator itr = _veggies.begin();
 _veggies.end() != itr; ++itr) {
 delete *itr;
 }
 _veggies.clear();
 }
 virtual void bake() const {
 std::cout << "Bake for 25 minutes at 350"
 << std::endl;
 }
 virtual void box() const {
 std::cout << "Place pizza in official
 PizzaStore box" << std::endl;
 } //...all the other methods...
};
```

The pure virtual prepare method will collect all the ingredients from the ingredient factory

## Abstract Factory: example

- The concrete product classes get their ingredients

from the ingredient factories: there's no more need for specific classes for the regional versions

```
class ClamPizza : public Pizza {
private: PizzaIngredientFactory* _ingredientFactory;
public: ClamPizza(PizzaIngredientFactory*
ingredientFactory) :
_iningredientFactory(ingredientFactory) {
}
void prepare() const {
std::cout << "Preparing " << getName().c_str() << std::endl;
_dough = _ingredientFactory->createDough();
_sauce = _ingredientFactory->createSauce();
_cheese = _ingredientFactory->createCheese();
_clam = _ingredientFactory->createClam();
}
};
```

## Abstract Factory: example

```
class NaplesPizzaStore : public PizzaStore {
```

```
public: Pizza* createPizza(std::string item) const { 1
 Pizza* pizza = 0;
```

The store is composed with the regional ingredient factory.

```
 PizzaIngredientFactory* ingredientFactory = new NaplesPizzaIngredientFactory
();
```

```
 if(item.compare("cheese") == 0) {
 pizza = new CheesePizza(ingredientFactory);
 pizza->setName("Naples Style Cheese Pizza");
 } else if(item.compare("veggie") == 0) {
 pizza = new VeggiePizza(ingredientFactory);
 pizza->setName("Naples Style Veggie Pizza");
 } else if(item.compare("clam") == 0) {
 pizza = new ClamPizza(ingredientFactory);
 pizza->setName("Naples Style Clam Pizza");
 } else if(item.compare("pepperoni") == 0) {
 pizza = new PepperoniPizza(ingredientFactory);
 pizza->setName("Naples Style Pepperoni Pizza");
 }
 return pizza;
}
```

2 For each type of product we pass the factory it needs, to get the ingredients from it. The factory (built according to Abstract Factory pattern) creates a family of products

## Putting everything together

```
PizzaStore* nStore = new NaplesPizzaStore();
```

```
Pizza* pizza = nStore->orderPizza("cheese");
```

```
std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

```
pizza = nStore->orderPizza("clam");
```

```
std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

```
PizzaStore* nStore = new NaplesPizzaStore();
```

```
Pizza* pizza = nStore->orderPizza("cheese");
```

The orderPizza()  
method calls the  
createPizza()  
method 1

```
std::cout << "Just ordered a " << pizza->toString() << std::endl;
```

When the createPizza() method is  
called the factory gets involved 2

```
pizza = nStore->prepare();
```

When prepare() method is called the  
factory creates the ingredients 3

```
std::cout << "Just ordered a " << pizza->toString() << std::endl;
```