



Run-Time Type Information (RTTI)

TD(ptr) tipo dinamico di puntatore polimorfo **ptr**

TD(ref) tipo dinamico di riferimento polimorfo **ref**

Run-time type information

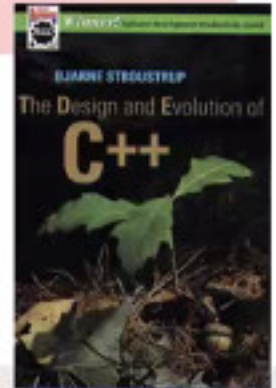
From Wikipedia, the free encyclopedia
(Redirected from [RTTI](#))

In computer programming, **RTTI** (**Run-Time Type Information**, or **Run-Time Type Identification**^[*citation needed*]) refers to a C++ mechanism that exposes information about an object's [data type](#) at [runtime](#). Run-time type information can apply to simple data types, such as integers and characters, or to generic types. This is a C++ specialization of a more general concept called [type introspection](#). Similar mechanisms are also known in other programming languages.

In the original C++ design, [Bjarne Stroustrup](#) did not include run-time type information, because he thought this mechanism was frequently misused.^[1]



intervista del 1996



CP:

C'è qualche decisione tecnica che in retrospettiva ti piacerebbe cambiare - non qualcosa che non avresti avuto modo di far accettare sin dall'inizio, ma qualcosa che l'esperienza ha dimostrato in qualche modo essere imperfetto, e che vorresti cambiare se ne avessi la possibilità. Tralasciando al momento la compatibilità con il C++ esistente, ma non con il C.

BS:

Ci sono molti dettagli con i quali mi piacerebbe giocare, ma non ci sono caratteristiche che vorrei eliminare, anche se potessi, o qualcosa di importante che vorrei aggiungere e che saprei come aggiungere. Spesso vengono mosse delle critiche all'ereditarietà multipla o al RTTI, ma devo dire che senza uno di essi il C++ sarebbe decisamente meno espressivo. Io uso entrambi a fondo e le soluzioni che dovrei utilizzare in mancanza di esse sono tutt'altro che eleganti; ci sono anche esempi in D&E...

L'operatore typeid

L'operatore `typeid` permette di determinare il tipo di una espressione qualsiasi a tempo di esecuzione.

```
#include <typeinfo> // includere sempre questo header file
#include <iostream> // per usare typeid

int main() {
    int i=5;
    std::cout << typeid(i).name() << endl;    // stampa: i(nt)
    std::cout << typeid(3.14).name() << endl; // stampa: d(ouble)
    if (typeid(i) == typeid(int)) std::cout << "Yes";
}
```


L'operatore `typeid` ha come argomento una espressione o un tipo qualsiasi e ritorna un oggetto della classe `type_info`. La definizione della classe `type_info` è nel file header `typeinfo` e rende disponibili i seguenti metodi comuni a tutte le implementazioni del compilatore.

```
class type_info {  
    // rappresentazione dipendente dall'implementazione  
private:  
    type_info();  
    type_info(const type_info&);  
    type_info& operator=(const type_info&);  
public:  
    bool operator==(const type_info&) const;  
    bool operator!=(const type_info&) const;  
    const char* name() const;  
};
```

Comportamento di typeid

[1] Se l'espressione operando di **typeid** è un riferimento **ref** ad una classe che **contiene almeno un metodo virtuale**, cioè una **classe polimorfa**, allora **typeid** restituisce un oggetto di **type_info** che rappresenta il tipo dinamico di **ref**.

[2] Se l'espressione operando di **typeid** è un puntatore **dereferenziato** ***punt**, dove **punt** è un puntatore ad un **tipo polimorfo**, allora **typeid** restituisce un oggetto di **type_info** che rappresenta il tipo **T** dove **T*** è il tipo dinamico di **punt**.

ATTENZIONE



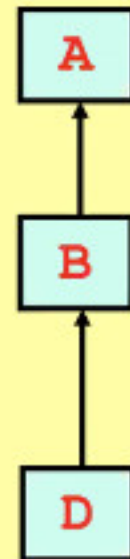
(A) Se la classe non contiene metodi virtuali allora **typeid** restituisce il tipo statico del riferimento o del puntatore dereferenziato.

(B) **typeid** su un puntatore (non dereferenziato) restituisce sempre il tipo statico del puntatore.

```
class A { public: virtual ~A() {} };  
class B : public A {};  
class D : public B {};
```

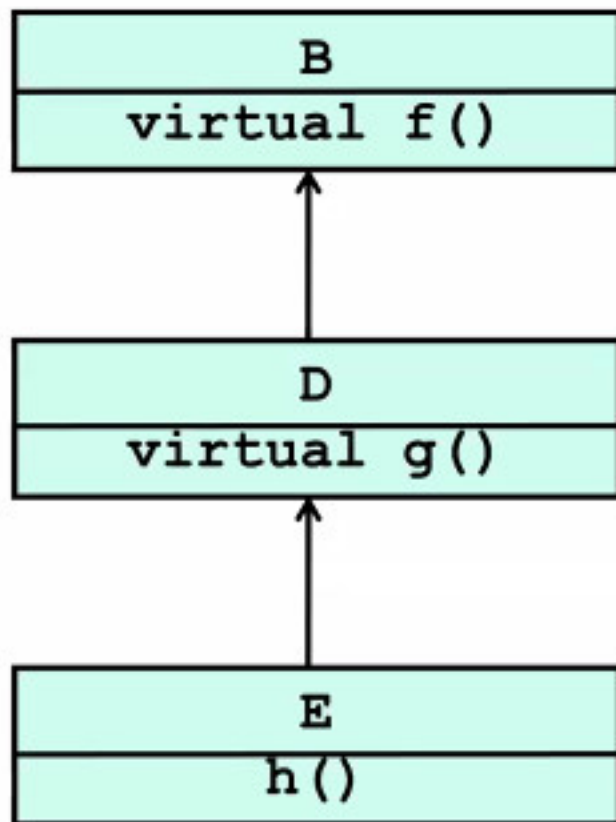
```
#include<typeinfo>  
#include<iostream>
```

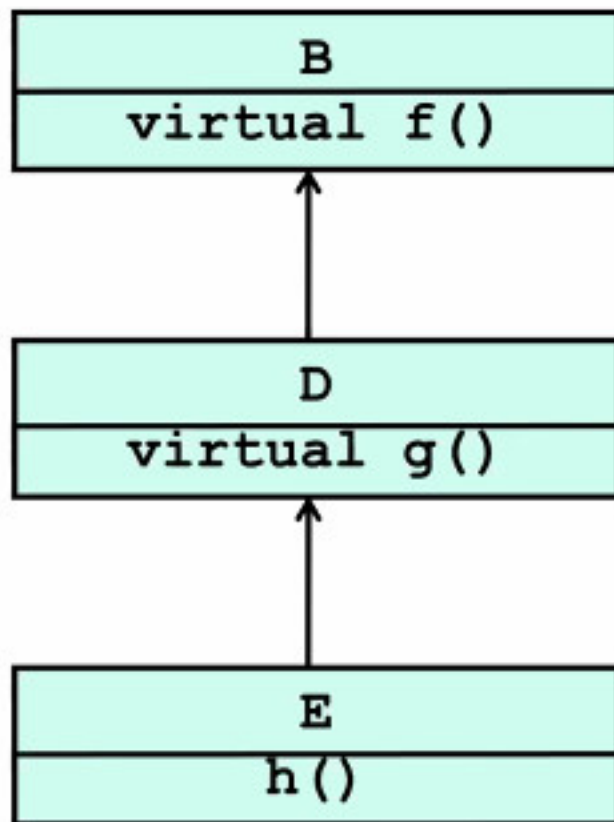
```
int main() {  
    B b; D d;  
    B& rb = d;  
    A* pa = &b;  
    if(typeid(rb) == typeid(B)) std::cout << '1';  
    if(typeid(rb) == typeid(D)) std::cout << '2';  
    if(typeid(*pa) == typeid(A)) std::cout << '3';  
    if(typeid(*pa) == typeid(B)) std::cout << '4';  
    if(typeid(*pa) == typeid(D)) std::cout << '5';  
}
```




```
class A { public: virtual ~A() {} };  
class B : public A {};  
class D : public B {};  
  
#include<typeinfo>  
#include<iostream>  
  
int main() {  
    B b; D d;  
    B& rb = d;  
    A* pa = &b;  
    if(typeid(rb) == typeid(B)) std::cout << '1';  
    if(typeid(rb) == typeid(D)) std::cout << '2';  
    if(typeid(*pa) == typeid(A)) std::cout << '3';  
    if(typeid(*pa) == typeid(B)) std::cout << '4';  
    if(typeid(*pa) == typeid(D)) std::cout << '5';  
    // stampa: 24  
}
```



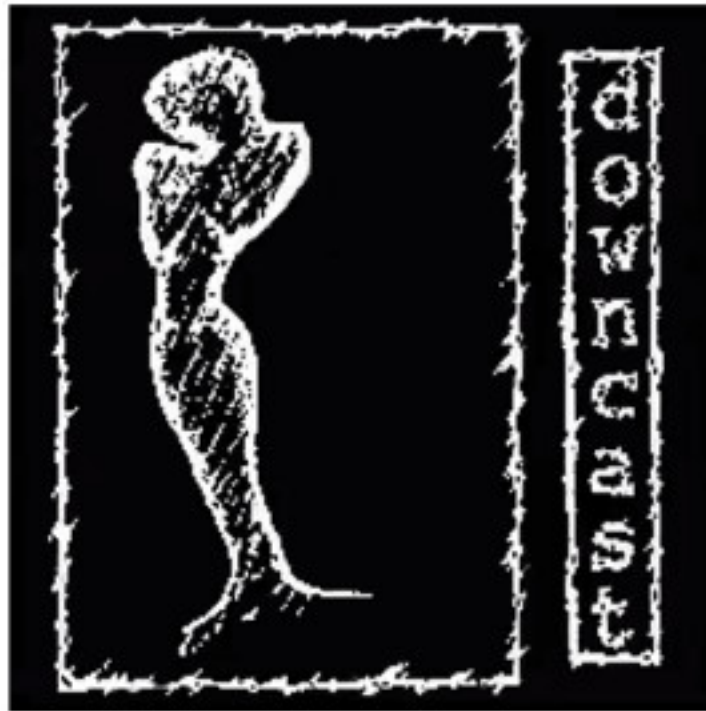




```
B b; D d; E e;  
B* p = &d;  
p->f();  
p->g(); // come fare?  
D* r = &e;  
r->g();  
r->h(); // come fare?
```

non compila

non compila



Upcasting and Downcasting

Upcasting and Downcasting

Downcasting

From Wikipedia, the free encyclopedia

(Redirected from [Downcast](#))

In [object-oriented programming](#), **downcasting** or type refinement is the act of [casting](#) a reference of a base class to one of its derived classes.

In many [programming languages](#), it is possible to check through [type introspection](#) to determine whether the type of the referenced object is indeed the one being cast to or a derived type of it, and thus issue an error if it is not the case.

In other words, when a variable of the base class ([parent class](#)) has a value of the derived class ([child class](#)), downcasting is possible.

dynamic_cast

B tipo polimorfo, $D \leq B$

B* p puntatore polimorfo

Downcast: $B^* \Rightarrow D^*$ $B\& \Rightarrow D\&$

`dynamic_cast<D*>(p) != 0`

se e solo se

$TD(p) \leq D^*$

Tipo dinamico di p compatibile con il tipo target D*

Nel caso dei riferimenti, se il `dynamic_cast` di un riferimento fallisce allora viene sollevata un'eccezione di tipo `std::bad_cast` (definito nel file header `typeinfo`).

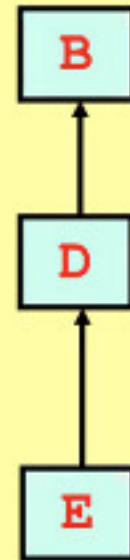
```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {} ;

#include<typeinfo>
#include<iostream>

int main() {
    D d;
    B& b = d; // upcast
    try {
        X& xr = dynamic_cast<X&>(b) ;
    } catch(std::bad_cast e) {
        std::cout << "Cast fallito!" << std::endl;
    }
}
```

Downcasting

```
class B { // classe base polimorfa
public:
    virtual void m();
};
class D : public B {
public:
    virtual void f(); // nuovo metodo virtuale
};
class E: public D {
    void g();          // nuovo metodo
};
```



```
B* fun() { /* può ritornare B*, D*, E*, ... */ }
```

```
int main() {
    B* p = fun();
    if(dynamic_cast<D*>(p)) (static_cast<D*>(p))->f();
    E* q = dynamic_cast<E*>(p);
    if(q) q->g();
}
```

downcast è possibile

downcast

downcast ha successo

Criticism [\[edit\]](#)

Many people advocate avoiding downcasting, since according to the [LSP](#), an [OOP](#) design that requires it is flawed.^{[\[citation needed\]](#)} Some languages, such as [OCaml](#), disallow downcasting altogether.^{[\[1\]](#)}

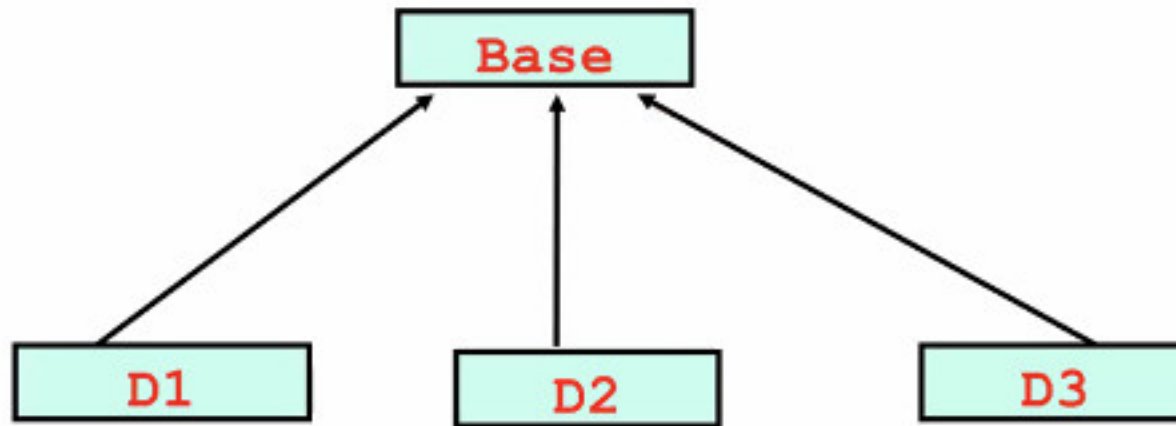
A popular example of a badly considered design is containers of [top types](#), like the [Java](#) containers before [Java generics](#) were introduced, which requires downcasting of the contained objects so that they can be used again.



Downcasting vs metodi virtuali

- usare il downcasting **solo quando necessario**
- **non fare** type checking dinamico inutile
- ove possibile **usare metodi virtuali nelle classi base** piuttosto che fare type checking dinamico

Pattern



Versione con dynamic_cast

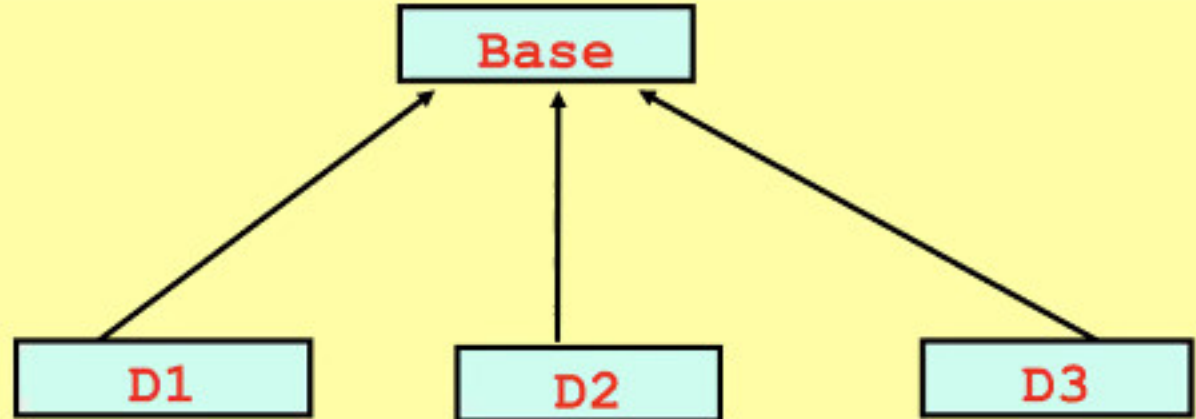
```
class Base {  
public:  
    virtual ~Base() {}  
    void do_Base_things() {}  
};
```

```
class D1: public Base {  
public:  
    void do_D1_things() {}  
};
```

```
class D2: public Base {  
public:  
    void do_D2_things() {}  
};
```

```
class D3: public Base {  
public:  
    void do_D3_things() {}  
};
```

```
void fun(Base& rb) {  
    rb.do_Base_things();  
    if (D1* p1 = dynamic_cast<D1*> (&rb))  
        p1->do_D1_things();  
    else if (D2* p2 = dynamic_cast<D2*> (&rb))  
        p2->do_D2_things();  
    else if (D3* p3 = dynamic_cast<D3*> (&rb))  
        p3->do_D3_things();  
}
```



poco estensibile

```

class Base {
public:
    virtual ~Base() {}
    void do_Base_things() {}
    virtual void do_polymorphic_things() { // metodo virtuale della Base
        do_Base_things();
    }
};

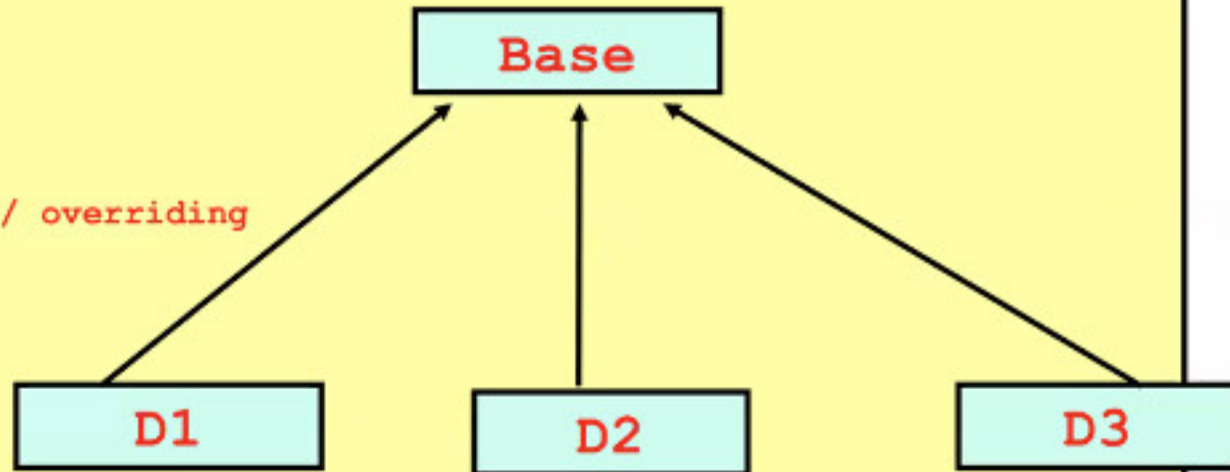
class D1: public Base {
public:
    void do_D1_things() {}
    void do_polymorphic_things() { // overriding
        do_Base_things();
        do_D1_things();
    }
};

class D2: public Base {
public:
    void do_D2_things() {}
    void do_polymorphic_things() { // overriding
        do_Base_things();
        do_D2_things();
    }
};

class D3: public Base {
public:
    void do_D3_things() {}
    void do_polymorphic_things() { // overriding
        do_Base_things();
        do_D3_things();
    }
};

void fun(Base& rb) {
    rb.do_polymorphic_things(); // chiamata polimorfa
}

```



estensibile

EXAMPLE


```
class impiegato { // classe base astratta
protected: static double stipBase;
public:      virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
    virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus; // campo dati proprio
public:
    programmatore(double b): bonus(b) {}
    virtual double stipendioBase() const {return impiegato::stipBase;}
    double getBonus() const {return bonus;}; // metodo non virtuale
};
```

```
class impiegato { // classe base astratta
protected: static double stipBase;
public:     virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
    virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus; // campo dati proprio
public:
    programmatore(double b): bonus(b) {}
    virtual double stipendioBase() const {return impiegato::stipBase;}
    double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
    static double stipendio(const impiegato& p) {
        const programmatore* q = dynamic_cast<const programmatore*>(&p);
        // se il cast ha successo,
        // cioè il tipo dinamico di &p è sottotipo di programmatore*
        if(q) return q->stipendioBase() + q->getBonus();
        // altrimenti (q==0),
        // cioè il tipo dinamico di &p non è sottotipo di programmatore*
        else return p.stipendioBase();
    }
};
```

```
class impiegato { // classe base astratta
protected: static double stipBase;
public:      virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
    virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus; // campo dati proprio
public:
    programmatore(double b): bonus(b) {}
    virtual double stipendioBase() const {return impiegato::stipBase;}
    double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
    static double stipendio(const impiegato& p) {
        const programmatore* q = dynamic_cast<const programmatore*>(&p);
        // se il cast ha successo,
        // cioè il tipo dinamico di &p è sottotipo di programmatore*
        if(q) return q->stipendioBase() + q->getBonus();
        // altrimenti (q==0),
        // cioè il tipo dinamico di &p non è sottotipo di programmatore*
        else return p.stipendioBase();
    }
};
```



```

class impiegato { // classe base astratta
protected: static double stipBase;
public:     virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
    virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus; // campo dati proprio
public:
    programmatore(double b): bonus(b) {}
    virtual double stipendioBase() const {return impiegato::stipBase;}
    double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
    static double stipendio(const impiegato& p) {
        const programmatore* q = dynamic_cast<const programmatore*>(&p);
        // se il cast ha successo,
        // cioè il tipo dinamico di &p è sottotipo di programmatore*
        if(q) return q->stipendioBase() + q->getBonus();
        // altrimenti (q==0),
        // cioè il tipo dinamico di &p non è sottotipo di programmatore*
        else return p.stipendioBase();
    }
};

```



typeid vs dynamic_cast

```
class impiegato { // classe astratta
protected: static double stipBase;
public:     virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

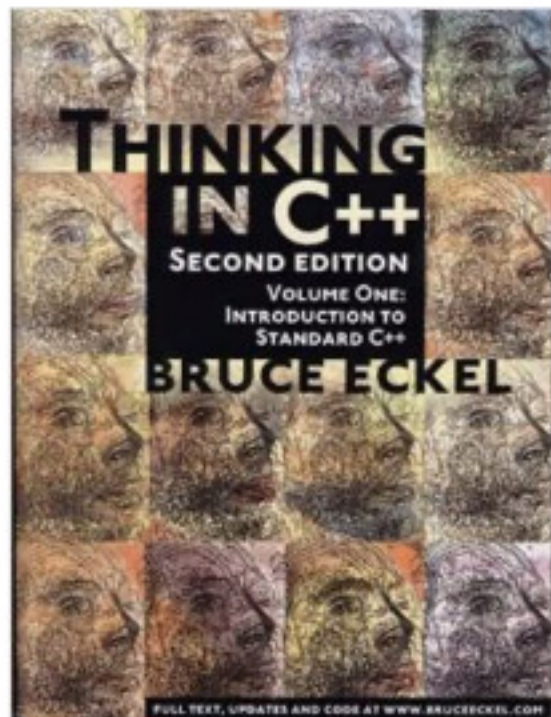
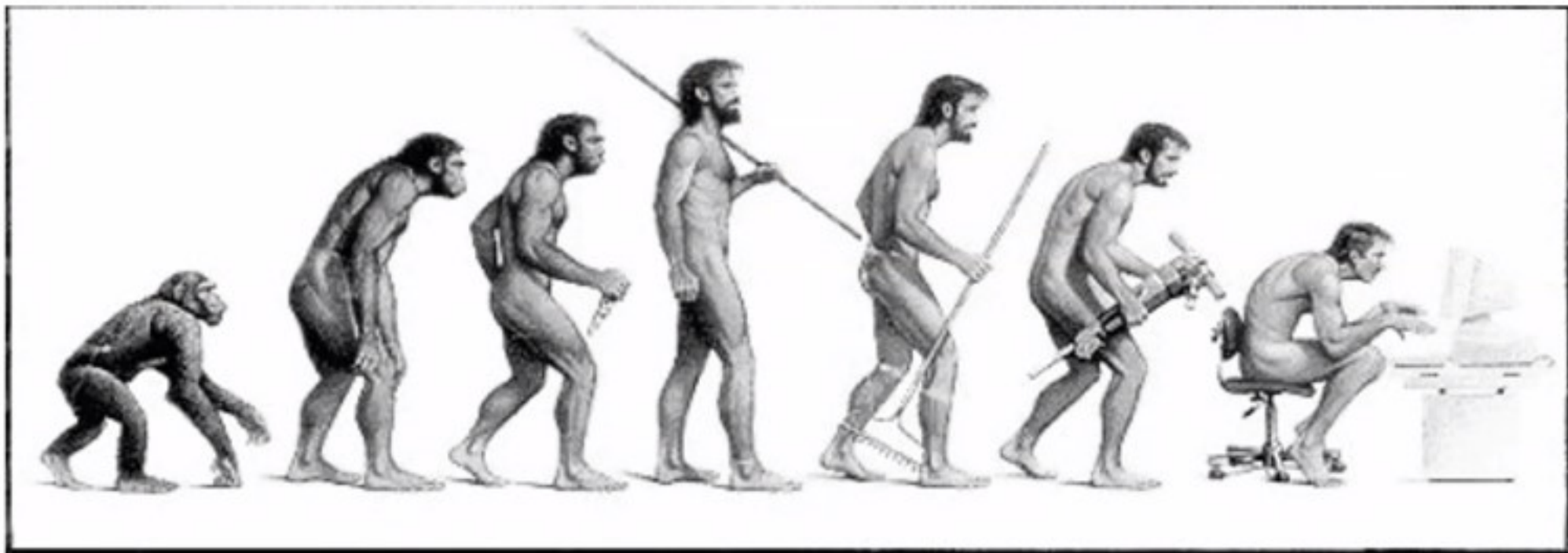
class manager : public impiegato {
public:
    virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

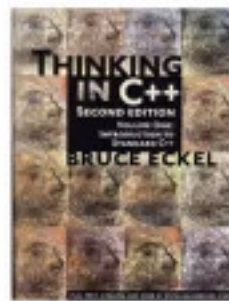
class programmatore : public impiegato {
private: double bonus;
public:
    programmatore(double b): bonus(b) {}
    virtual double stipendioBase() const {return impiegato::stipBase;}
    double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
    static double stipendio(impiegato* p) {
        if (typeid(*p)==typeid(programmatore)) { // non è codice estensibile!
            programmatore* q = dynamic_cast<programmatore*>(p);
            return q->stipendioBase() + q->getBonus();
        }
        else return p->stipendioBase();
    }
};
```



```
int main() {  
    manager m; programmatore p;  
    cout << "Il manager guadagna "  
        << GoogleAdmin::stipendio(m) << " Euro\n";  
    cout << "Il programmatore guadagna "  
        << GoogleAdmin::stipendio(p) << " Euro\n";  
}
```





Evolution of C++ programmers

C programmers seem to acquire C++ in three steps. First, as simply a “better C,” because C++ forces you to declare all functions before using them and is much pickier about how variables are used. You can often find the errors in a C program simply by compiling it with a C++ compiler.

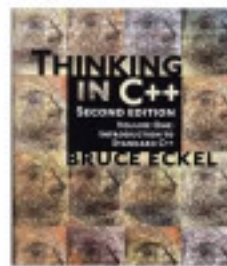
The second step is “object-based” C++. This means that you easily see the code organization benefits of grouping a data structure together with the functions that act upon it, the value of constructors and destructors, and perhaps some simple inheritance. Most programmers who have been working with C for a while quickly see the usefulness of this because, whenever they create a library, this is exactly what they try to do. With C++, you have the aid of the compiler.

You can get stuck at the object-based level because you can quickly get there and you get a lot of benefit without much mental effort. It’s also easy to feel like you’re creating data types – you make classes and objects, you send messages to those objects, and everything is nice and neat.

But don’t be fooled. If you stop here, you’re missing out on the greatest part of the language, which is the jump to true object-oriented programming. You can do this only with virtual functions.

Virtual functions enhance the concept of type instead of just encapsulating code inside structures and behind walls, so they are without a doubt the most difficult concept for the new C++ programmer to fathom. However, they’re also the turning point in the understanding of object-oriented programming. If you don’t use virtual functions, you don’t understand OOP yet.

Because the virtual function is intimately bound with the concept of type, and type is at the core of object-oriented programming, there is no analog to the virtual function in a traditional procedural language. As a procedural programmer, you have no referent with which to think about virtual functions, as you do with almost every other feature in the language. Features in a procedural language can be understood on an algorithmic level, but virtual functions can be understood only from a design viewpoint.



Why virtual functions?

At this point you may have a question: “If this technique is so important, and if it makes the ‘right’ function call all the time, why is it an option? Why do I even need to know about it?”

This is a good question, and the answer is part of the fundamental philosophy of C++: “Because it’s not quite as efficient.” You can see from the previous assembly-language output that instead of one simple CALL to an absolute address, there are two – more sophisticated – assembly instructions required to set up the virtual function call. This requires both code space and execution time.

Some object-oriented languages have taken the approach that late binding is so intrinsic to object-oriented programming that it should always take place, **that it should not be an option, and the user shouldn’t have to know about it.** This is a design decision when creating a language, and that particular path is appropriate for many languages.[\[56\]](#) However, C++ comes from the C heritage, where efficiency is critical. After all, C was created to replace assembly language for the implementation of an operating system (thereby rendering that operating system – Unix – far more portable than its predecessors). One of the main reasons for the invention of C++ was to make C programmers more efficient.[\[57\]](#) And the first question asked when C programmers encounter C++ is, “What kind of size and speed impact will I get?” If the answer were, “Everything’s great except for function calls when you’ll always have a little extra overhead,” many people would stick with C rather than make the change to C++. In addition, inline functions would not be possible, because virtual functions must have an address to put into the VTABLE. So the virtual function is an option, *and* the language defaults to nonvirtual, which is the fastest configuration. Stroustrup stated that his guideline was, “If you don’t use it, you don’t pay for it.”

Thus, the **virtual** keyword is provided for efficiency tuning. When designing your classes, however, you shouldn’t be worrying about efficiency tuning. If you’re going to use polymorphism, use virtual functions everywhere. You only need to look for functions that can be made non-virtual when searching for ways to speed up your code (and there are usually much bigger gains to be had in other areas – a good profiler will do a better job of finding bottlenecks than you will by making guesses).

5 PRINCIPLES OF DESIGN

AND WHAT THEY CAN DO FOR YOU



S.O.L.I.D

OBJECT ORIENTED DESIGN

SOLID (object-oriented design)

From Wikipedia, the free encyclopedia

In [computer programming](#), **SOLID** (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) is a [mnemonic acronym](#) introduced by Michael Feathers for the "first five principles" named by [Robert C. Martin](#)^{[1][2]} in the early 2000s^[3] that stands for five basic principles of [object-oriented programming](#) and [design](#). The principles, when applied together, intend to make it more likely that a [programmer](#) will create [a system that is easy to maintain and extend over time](#).^[3]

SOLID

Principles

[Single responsibility](#)

[Open/closed](#)

[Liskov substitution](#)

[Interface segregation](#)

[Dependency inversion](#)

V•T•E



Mantenibilità
Estensibilità

Overview [\[edit\]](#)

Initial	Stands for (acronym)	Concept
S	SRP ^[4]	Single responsibility principle a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	OCP ^[5]	Open/closed principle "software entities ... should be open for extension, but closed for modification."
L	LSP ^[6]	Liskov substitution principle "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract .
I	ISP ^[7]	Interface segregation principle "many client-specific interfaces are better than one general-purpose interface." ^[8]
D	DIP ^[9]	Dependency inversion principle one should "Depend upon Abstractions. Do not depend upon concretions." ^[8] Dependency injection is one method of following this principle.

SOLID

- Single responsibility principle
 - *A class should have only a single responsibility*



SOLID

- Open/closed principle
 - *Open for extension, but closed for modification*
 - *Alistair Cockburn: “...Identify points of predicted variation and create a stable interface around them...”*

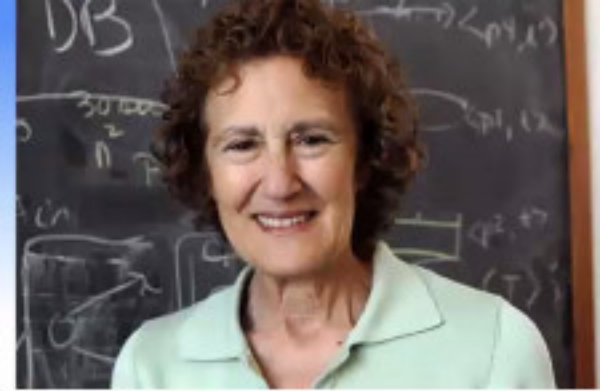
**Non servirà la chirurgia toracica
se si usa il cappotto!**



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

SOLID



- Liskov substitution principle
 - *Replace objects with instances of their subtypes without altering the correctness of that program*

Rectangle

Square



SOLID

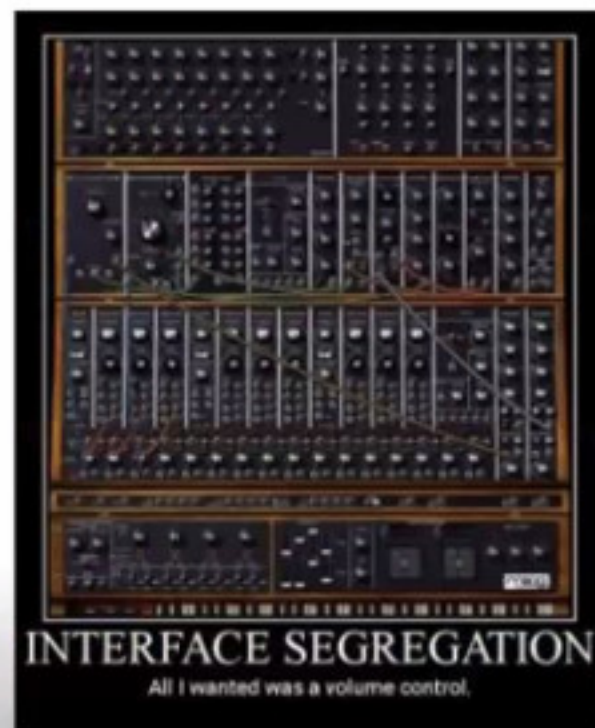
- Interface segregation principle
 - *Many client-specific interfaces are better than one general-purpose interface*

Importance in object-oriented design [\[edit \]](#)

Within [object-oriented design](#), [interfaces](#) provide layers of abstraction that facilitate conceptual explanation of the code and create a barrier preventing coupling to [dependencies](#).

According to many software experts who have signed the Manifesto for [Software Craftsmanship](#), writing well-crafted and self-explanatory software is almost as important as writing working software.^[4] Using interfaces to further describe the intent of the software is often a good idea.

A system may become so coupled at multiple levels that it is no longer possible to make a change in one place without necessitating many additional changes.^[1] Using an interface or an [abstract class](#) can prevent this side effect.



SOLID

- Dependency inversion principle
 - *Abstractions should not depend on details*
 - *Don't depend on anything concrete*
 - *Work with interfaces*

In [object-oriented design](#), the **dependency inversion principle** refers to a specific form of [decoupling](#) software [modules](#). When following this principle, the conventional [dependency](#) relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:^[1]

- A. High-level modules should not depend on low-level modules. Both should depend on [abstractions](#).
- B. Abstractions should not depend on details. Details should depend on abstractions.



