static_cast<T>

» den Compiler überreden, verwandte Typen verträglich zu verwenden «
das Ergebnis kann ohne erneute Umwandlung verwendet werden

```
class X { ... };
class Y : public X {};

// eine Y& ist auch immer eine X&
Y o;
X& x1 = o; // implizite Anpassung der Typen
X& x2 = static_cast<X&> (o); // dasselbe

// manchmal ist eine X& auch eine Y&
Y& y1 = static_cast<Y&> (x1); // ok, weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X; Y& y2 = static_cast<Y&> (x3); // Crash ahead
```

reinterpret_cast<T>

» den Compiler überreden, nicht verwandte Typen verträglich zu verwenden «

das Ergebnis kann nur nach erneuter Rückumwandlung verwendet werden

die unveränderte Bitbelegung wird anders interpretiert; zumeist nicht portabel

```
int *pi = &someint;
void *v = reinterpret_cast<void*>(pi);
// don't use v, but:
int *p = reinterpret_cast<int*>(v);
*p = 337; // OK: sets someint
```

dynamic_cast<T>

» zur Laufzeit verwandte Typen verträglich und sicher verwenden «

```
class X { virtual void foo(); };
class Y : public X {};


// manchmal ist eine X& auch eine Y&
Y& y1 = dynamic_cast<Y&> (x1); //ok,weil x1 ein Y ref.
// aber eben nicht immer:
X& x3 = *new X;
Y& y2 = dynamic_cast<Y&> (x3); // NO Crash ahead
// exception std::bad_cast !!!
```

dynamic_cast<T>

Implementation setzt offenbar Auswertung von Laufzeittypinformationen (RTTI - run time type identification) voraus

Funktioniert für Zeiger und Referenzen polymorpher Typen (es muss virtuelle Funktionen in der Basisklasse geben!)

Ein downcast (von einem Zeiger/einer Referenz auf eine Basisklasse auf einen Zeiger/eine Referenz einer Ableitung) gelingt, wenn das referenzierte Objekt vom Typ der Ableitung oder einer Ableitung dieser ist.

Bei Zeigern liefert `dynamic_cast` den Wert 0, bei Referenzen wird die Ausnahme `std::bad_cast` geworfen, wenn die dynamische Typ nicht ausreicht

dynamic_cast<T>

```
class A {
public:  virtual void needed () {}
};
class B: public A {public: int i;};
class C: public B {public: int j;};
int main() {
        A *pa = new B;
        B *pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
        C *pc = dynamic_cast<C*>(pb);
        if (pc) pc->j = 54321; // wird nicht ausgefuehrt
        pa = new C;
        pb = dynamic_cast<B*>(pa);
        if (pb) pb->i = 12345; // ok, es ist ein B
}
```

Darüber hinaus kann man die Typidentität direkt abfragen:

dazu existiert der Operator `typeid` (wie `sizeof` vom Compiler umgesetzt und nicht überladbar), der eine (vergleichbare) Struktur des Typs `type_info` liefert, der Vergleich von `type_info` gelingt, wenn exakt der gleiche Typ vorliegt

auf `type_info` ist wiederum die Funktion `name()` definiert, die einen Klarnamen der Klasse (nicht notwendig identisch mit dem Klassennamen) erzeugt

`#include <typeinfo>` ist erforderlich

Die beteiligten Typen müssen wiederum polymorph sein, d.h. mindestens eine virtuelle Funktion in der gemeinsamen Basis besitzen

```
#include <typeinfo>
#include <iostream>
using namespace std;

class A { virtual void any () {} };
class B: public A { };
class C: public A { };
void check (A* p) {
        if (typeid(*p)==typeid(A))
                { cout << "es ist ein A\n"; return; }
        if (typeid(*p)==typeid(B))
                { cout << "es ist ein B\n"; return; }
        cout << "weder A noch B\n";
}
const char* get_name(A* p) {
        return typeid(*p).name();
}
```

```
int main()
{
        A *p;
        p = new A;
        check (p);
        cout << get_name(p) << endl;
        p = new B;
        check (p);
        cout << get_name(p) << endl;
        p = new C;
        check (p);
        cout << get_name(p) << endl;
}
```

```
C:\tmp>rtti
es ist ein A
A
es ist ein B
B
weder A noch B
C
```

dynamic_cast<T> ist manchmal nicht zu vermeiden

```
class B {
        // no functionality 'foo'
};
class D: public B {
        virtual void foo();
};
void register (B*);
B* next();
...
register(new D);
...
B* n = next();

// how to call foo ?
dynamic_cast<D*>(n)->foo();
```

RTTI nur in Ausnahmefällen explizit benutzen

statt spaghetti code

```
Shape * s;
if (typeid(*s)== typeid("Circle"))
((Circle*)s)->Circle::draw();
else
if (typeid(*s)== typeid("Rectangle"))
        ((Rectangle*)s)-> Rectangle::draw();
else ...
```

benutze

```
Shape * s;
s->draw(); // late bound virtual
```

Mehrfachvererbung (multiple inheritance)

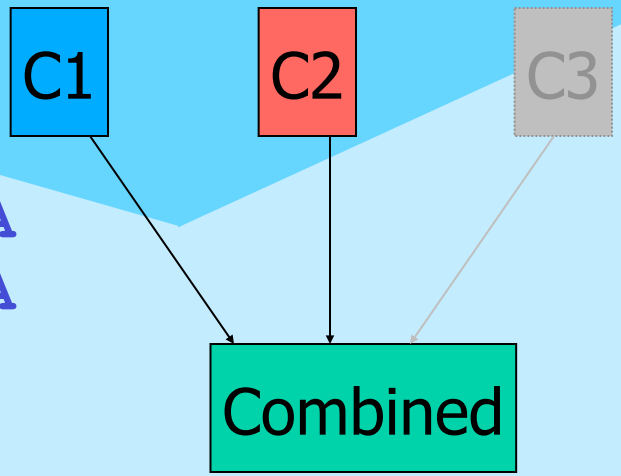
eine Klasse kann mehrere Basisklassen haben --> 'freie' Kombination von Konzepten:

```
class Combined:      public Concept1,
                     public Concept2,
                     private Concept3
{ .... };
```

jedes `Combined`-Objekt IST EIN `Concept1` und IST EIN `Concept2` (`Concept3`-Abstammung ist ein Implementationsdetail)

Mehrfachvererbung (multiple inheritance)
Polymorphie bleibt erhalten:

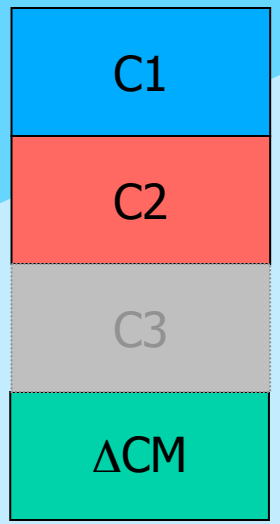
```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
// NOT c2=c1=cm;

// it's the same Object:
if (c1 == cm) // yes !
if (c2 == cm) // yes !
if (c1 == c2) // ERROR: uncomparable
```



Mehrfachvererbung (multiple inheritance)

Layout muss linearisiert werden:

```
Combined obj;
Combined * cm = &obj;
Concept1 * c1 = cm; // IS A
Concept2 * c2 = cm; // IS A
```




```
// but the (numeric) adresses may differ:
if(reinterpret_cast<void*>c1==reinterpret_cast<void*>cm)
        // yes or no!
if(reinterpret_cast<void*>c2==reinterpret_cast<void*>cm)
        // yes or no!
```

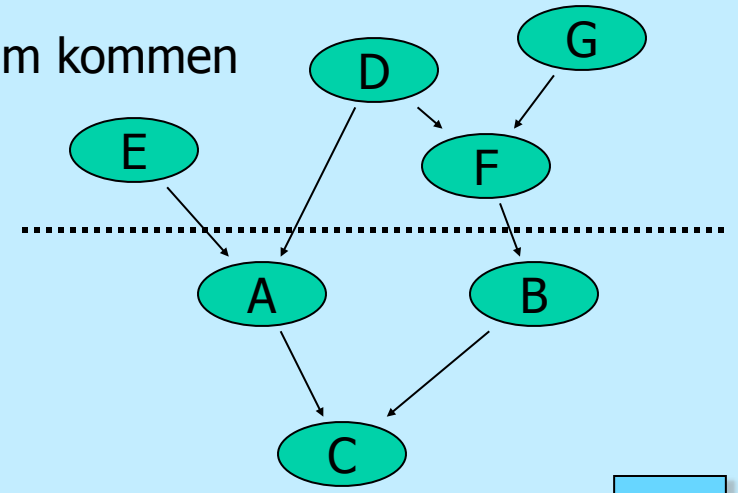
Mehrfachvererbung (multiple inheritance)

eine Klasse kann eine andere nicht direkt mehrfach einerben

```
struct A { int i; };
class B: public A, public A {   // NOT ALLOWED
       void foo(){ i = 0; /* which i ? A::i ? which A ? */ }
};
```

ansonsten kann es durchaus zu Maschen im Baum kommen

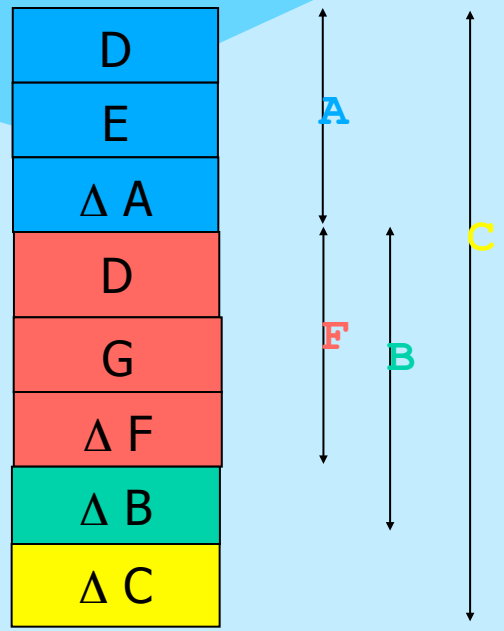
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
-----------------------------------
class C: public A, public B {...};
```



Mehrfachvererbung (multiple inheritance)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

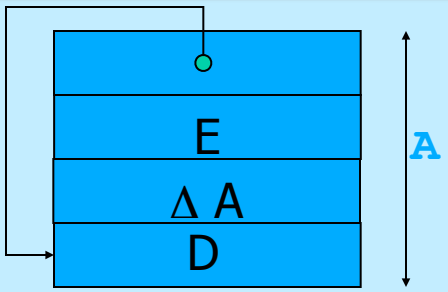
```
class A: public D, public E {...};
class F: public D, public G {...};
class B: public F {...};
class C: public A, public B {...};
```

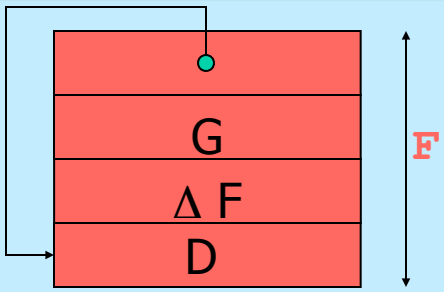


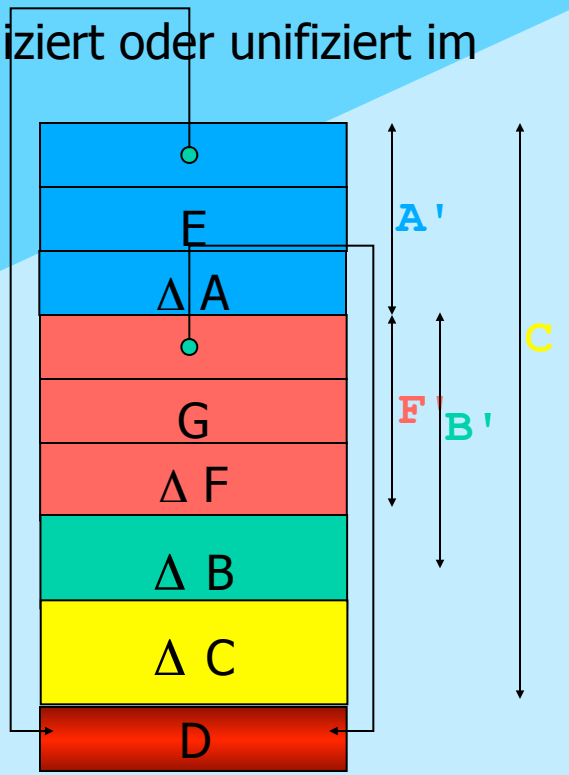
Mehrfachvererbung (multiple inheritance)

ob dabei der mehrfach eingeerbte Basisklassenanteil dupliziert oder unifiziert im resultierenden Objekt erscheint ist steuerbar:

```
class A: virtual public D,
              public E {...};
class F: virtual public D,
              public G {...};
class B: public F {...};
class C: public A, public B {...};
```







Mehrfachvererbung (multiple inheritance)

für beide Szenarien gibt es sinnvolle Anwendungen:

```
// class Listable {/* Listeneigenschaften */ };
class A: public Listable { ... };
// A's können in einer Liste erfasst werden
class B: public Listable { ... };
// B's können in einer Liste erfasst werden
class C: public A, public B { ... };
// C's können in zwei separaten Listen (als A und als B)
// erfasst werden
-------------------------------------------------------------
class Person {...};
class Angestellter: public virtual Person {...};
class Student: public virtual Person {...};
class Werkstudent: public Angestellter, public Student
{...}; // ein und dieselbe Person !!!
```

non virtual

virtual