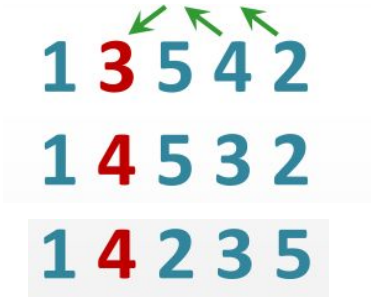
Objects layout in memory

Permutation with using standard function

```
class Solution {
 public:
     vector<vector<int>> permute(vector<int>& nums)
 {
     vector<vector<int>>result;
     std::sort(nums.begin(), nums.end());
     do
     {
         result.push_back(nums);
     }while(std::next_permutation(nums.begin(), nums.end()));

     return result;
 }
 };
```

Next Lexicographic permutation

```cpp
//next lexicographically permutation
bool nextPermutation(std::vector<int>& nums)
{
  int indexShouldChenge = 0;
  for(int i=nums.size(); i > 0; i--)
  {
      if(nums[i-1] < nums[i])
      {
          indexShouldChenge = i-1;
          std::sort(nums.begin()+i,nums.end());
          for(int j = i; j < nums.size(); j++)
          {
              if(nums[indexShouldChenge] < nums[j])
              {
                  swap(nums,indexShouldChenge,j);
                  return true;
              }
          }
      }
  }
  return false;
}
```



Upgraded solution

```
typedef pair<int, int> perm_element;

while (true)
{
    //  finding i: (perm[i] < perm[i + 1]) AND (perm[k]>=perm[k+1], k=(i+1,...,n-2))
    int i = n-2;
    for (; i >= 0; i--)
    {
        if (perm[i].first < perm[i+1].first) break;
    }
    if (i == -1) break; // all permutes are generated;
                        //perm[0]=n-1, perm[1]=n-2, ..., perm[n-1]=0

    // k = max{k: n-1 >= k >= i+1; perm[k] >= perm[i]}
    int k = i + 1;
    for (; k <= n-1; k++)
    {
        if (perm[k].first < perm[i].first) break;
    }
    k--;

    swap(perm[k], perm[i]);

    // swap the order of numbers - get the minimum number out of "digits"
    // {perm[i+1], perm[i+2],..., perm[n-1]}
    for (int k = 0; k < ((n-1)-(i+1)+1) / 2; k++)
        swap(perm[i+1 + k], perm[n-1 - k]);

    addPerm(perm, permutes);
}
```

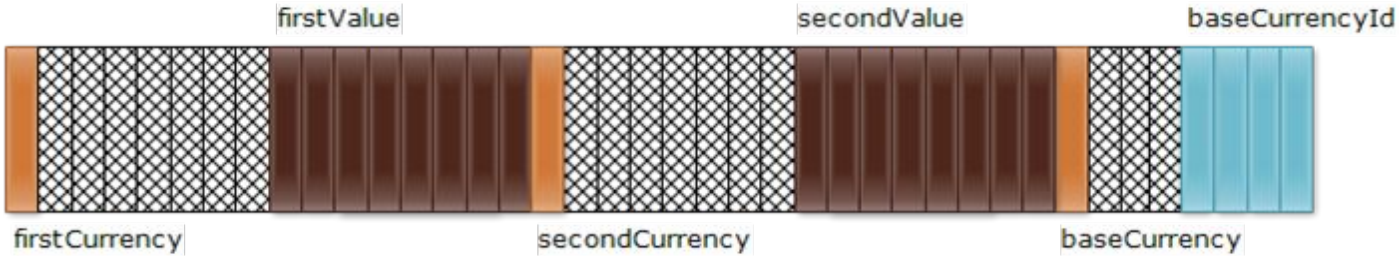
Get permutation by sequence number

```
std::vector<int> getNPernutation(std::vector<int> nums, int position, int n)
{
  int k = position;
  int nPost = n;  // how many ellements after i-ellement
  for(int i = 0; i < n; i++)
  {
      std::sort(nums.begin()+i,nums.end());
      for(int j=1;j<nPost+1;j++)
      {
          if(k <= j*factorial(nPost-1))
          {
              swap(nums,i,i+j-1);
              if(k >= (j-1)*factorial(nPost-1))
                k = k - (j-1)*factorial(nPost-1);
                      nPost -= 1;
              break;
          }
      }
  }
  return nums;
}

```

Objects layout in memory

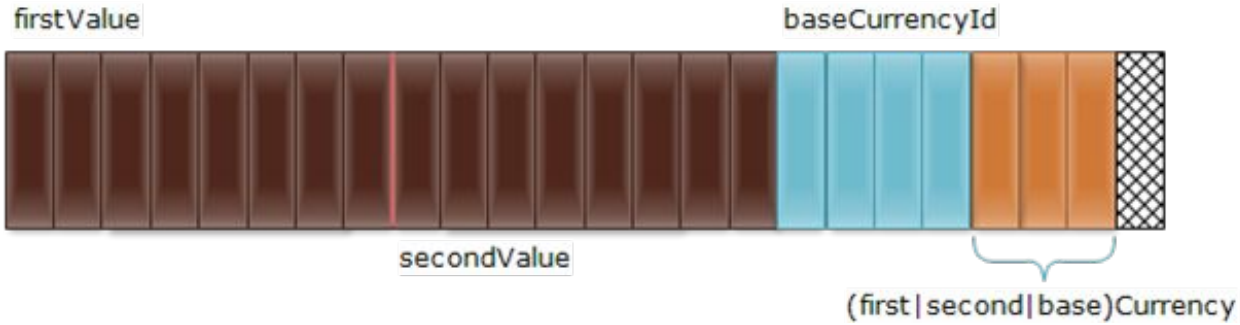
Alignment

```
struct Currency
{
   char firstCurrency;
   double firstValue;
   char secondCurrency;
   double secondValue;
   char baseCurrency;
   int baseCurrencyId;
};
```



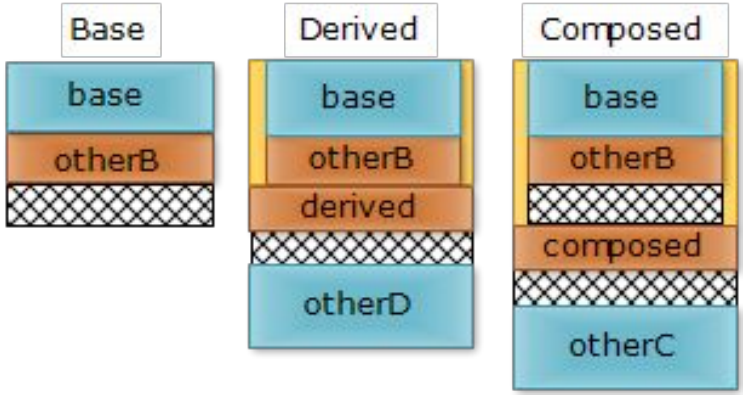
Alignment

```
struct Currency
{
   double firstValue;
   double secondValue;
   int baseCurrencyId;
   char firstCurrency;
   char secondCurrency;
   char baseCurrency;
};
```



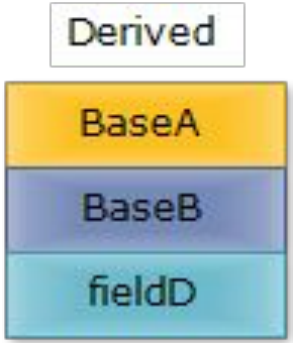
Inheritance

```
class Base
{
   int base;
   char otherB;
};

class Derived : public Base
{
   char derived;
   int otherD;
};

class Composed
{
   Base base;
   char composed;
   int otherC;
};
```



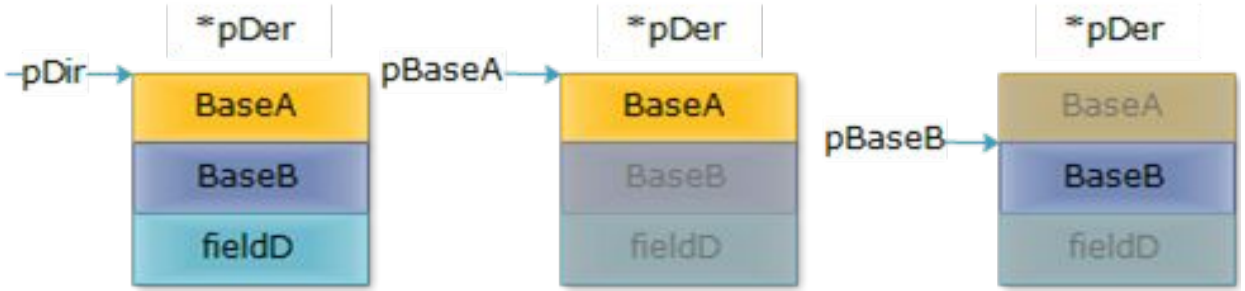
Multiple Inheritance

```
class BaseA
{
   int fieldA;
};

class BaseB
{
   int fieldB;
};

class Derived : public BaseA, public
BaseB
{
   int fieldD;
};
```



Multiple Inheritance

```
Derived* pDer = new Derived;
BaseA* pBaseA = pDer;
BaseB* pBaseB = pDer;
```



Multiple Inheritance

```
1  class Top
2  {
3      public:
4          int a;
5  };
6
7  class Left :  public Top
8  {
9      public:
10         int b;
11 };
12
13 class Right :  public Top
14 {
15     public:
16         int c;
17 };
18
19 class Bottom : public Left, public Right
20 {
21     public:
22         int d;
23 };
24
25 int main()
26 {
27
28 }
```

VIRTUAL TABLE

Non-virtual multiple inheritance

```
1  class Top
2  {
3      public:
4          int a;
5  };
6
7  class Left :  public Top
8  {
9      public:
10         int b;
11 };
12
13 class Right :  public Top
14 {
15     public:
16         int c;
17 };
18
19 class Bottom : public Left, public Right
20 {
21     public:
22         int d;
23 };
24
25 int main()
26 {
27
28 }
```

-fdump-class-hierarchy

```
1  Class Top
2     size=4 align=4
3     base size=4 base align=4
4  Top (0x0x7fc14eb204e0) 0
5
6  Class Left
7     size=8 align=4
8     base size=8 base align=4
9  Left (0x0x7fc14eb140d0) 0
10   Top (0x0x7fc14eb20540) 0
11
12 Class Right
13    size=8 align=4
14    base size=8 base align=4
15 Right (0x0x7fc14eb14138) 0
16   Top (0x0x7fc14eb205a0) 0
17
18 Class Bottom
19    size=20 align=4
20    base size=20 base align=4
21 Bottom (0x0x7fc15626cee0) 0
22   Left (0x0x7fc14eb141a0) 0
23     Top (0x0x7fc14eb20600) 0
24   Right (0x0x7fc14eb14208) 8
25     Top (0x0x7fc14eb20660) 8
26
```

Bottom
Left::Top::a
Left::b
Right::Top::a
Right::c
Bottom::d

```
Top* top = bottom;
error: `Top' is an ambiguous base of `Bottom'

/* The two possibilities can be disambiguated using */
Top* topL = (Left*) bottom;
Top* topR = (Right*) bottom;
```

Virtual Inheritance

```
1  class Top
2  {
3  public:
4     int a;
5  };
6
7  class Left : virtual public Top
8  {
9  public:
10    int b;
11 };
12
13 class Right : virtual public Top
14 {
15 public:
16    int c;
17 };
18
19 class Bottom : public Left, public Right
20 {
21 public:
22    int d;
23 };
24
25 int main()
26 {
27     Bottom* bottom = new Bottom();
28     Left* left = bottom;
29     int p = left->a;
30
31     /*
32     movl  left, %eax        # %eax = left
33     movl  (%eax), %eax      # %eax = left.vptr.Left
34     movl  (%eax), %eax      # %eax = virtual base offset
35     addl  left, %eax        # %eax = left + virtual base offset
36     movl  (%eax), %eax      # %eax = left.a
37     movl  %eax, p           # p = left.a
38     */
39
40     Bottom* bottom = new Bottom();
41     Right* right = bottom;
42     int p = right->a;
43 }
```

-fdump-class-hierarchy

```
42 Vtable for Bottom
43 Bottom::_ZTV6Bottom: 6u entries
44 0     32u
45 8     (int (*)(...))0                                    left
46 16    (int (*)(...))(& _ZTI6Bottom)
47 24    16u
48 32    (int (*)(...))-16
49 40    (int (*)(...))(& _ZTI6Bottom)
50
51 Construction vtable for Left (0x0x7f37e104b1a0 instance) in Bottom
52 Bottom::_ZTC6Bottom0_4Left: 3u entries
53 0     32u
54 8     (int (*)(...))0
55 16    (int (*)(...))(& _ZTI4Left)
56
57 Construction vtable for Right (0x0x7f37e104b208 instance) in Bottom
58 Bottom::_ZTC6Bottom16_5Right: 3u entries
59 0     16u
60 8     (int (*)(...))0
61 16    (int (*)(...))(& _ZTI5Right)
62
63 VTT for Bottom
64 Bottom::_ZTT6Bottom: 4u entries
65 0     ((& Bottom::_ZTV6Bottom) + 24u)
66 8     ((& Bottom::_ZTC6Bottom0_4Left) + 24u)
67 16    ((& Bottom::_ZTC6Bottom16_5Right) + 24u)
68 24    ((& Bottom::_ZTV6Bottom) + 48u)
69
70 Class Bottom
71    size=40 align=8
72    base size=32 base align=8
73 Bottom (0x0x7f37e87a3ee0) 0
74     vptridx=0u vptr=((& Bottom::_ZTV6Bottom) + 24u)
75   Left (0x0x7f37e104b1a0) 0
76       primary-for Bottom (0x0x7f37e87a3ee0)
77       subvttidx=8u
78     Top (0x0x7f37e1057600) 32 virtual
79         vbaseoffset=-24
80   Right (0x0x7f37e104b208) 16
81       subvttidx=16u vptridx=24u vptr=((& Bottom::_ZTV6Bottom) + 48u)
82     Top (0x0x7f37e1057600) alternative-path
```

```
class A
{
  virtual ~A() {}

  virtual void foo()  { cout << "A::foo()" << endl; }
  virtual void bar()  { cout << "A::bar()" << endl; }
  void baz()  { cout << "A::baz()" << endl; }
};

class B : public A
{
  virtual void foo()  { cout << "B::foo()" << endl; }
  void bar()  { cout << "B::bar()" << endl; }
  void baz()  { cout << "B::baz()" << endl; }
};

class C : public B
{
  virtual void foo()  { cout << "C::foo()" << endl; }
  void bar()  { cout << "C::bar()" << endl; }
  void baz()  { cout << "C::baz()" << endl; }
};
```

```
int main()
{
  cout << "pA is B:" << endl;
  A * pA = new B;
  pA->foo();
  pA->bar();
  pA->baz();
  delete pA;

  cout << "\npB is C:" << endl;
  B* pB = new C;
  pB->foo();
  pB->bar();
  pB->baz();
  delete pB;

  return 0;
}
```

Is it right output?

pA is B:
B::foo()
B::bar()
A::baz()

pA is C:
C::foo()
B::bar()
A::baz()

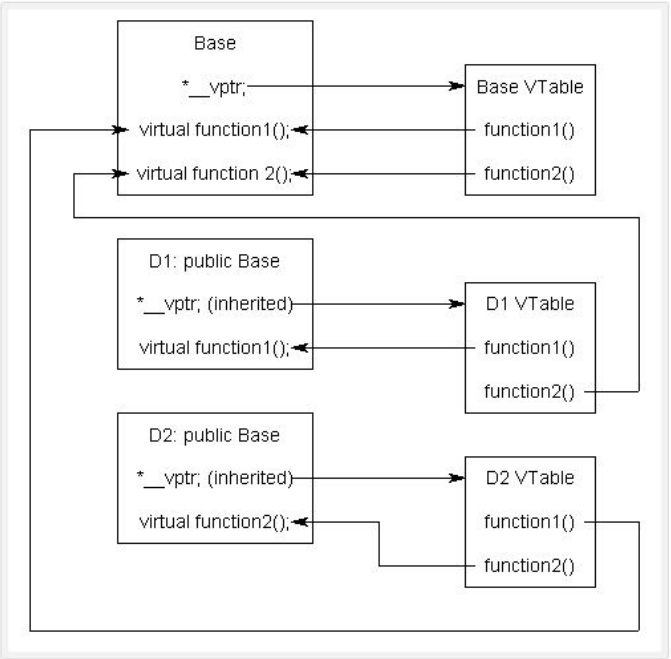
Right output:

```
pA is B:
B::foo()
B::bar()
A::baz()

pB is C:
C::foo()
C::bar()
A::baz()
```

Virtual table

C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

```
class Base
{
  virtual ~Base() {}

  virtual void function1() const { cout << "A::foo()" << endl; }
  virtual void function2() const { cout << "A::bar()" << endl; }
};

class D1 : public Base
{
  virtual void function1() const { cout << "B::foo()" << endl; }
};

class D2 : public Base
{
  virtual void function2() const { cout << "C::bar()" << endl; }
};
```

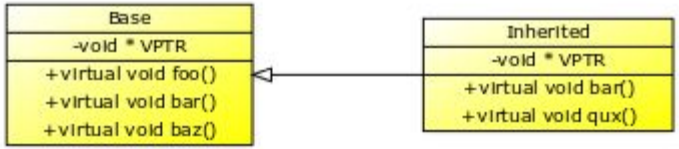


Proof:

g ++ -fdump-class-hierarchy option

```
Vtable for Base
Base::_ZTV4Base: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI4Base)
16    (int (*)(...))Base::~Base
24    (int (*)(...))Base::~Base
32    (int (*)(...))Base::function1
40    (int (*)(...))Base::function2

Class Base
   size=8 align=8
   base size=8 base align=8
Base (0x0x7f00526d0360) 0 nearly-empty
    vptr=((& Base::_ZTV4Base) + 16u)

Vtable for D1
D1::_ZTV2D1: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI2D1)
16    (int (*)(...))D1::~D1
24    (int (*)(...))D1::~D1
32    (int (*)(...))D1::function1
40    (int (*)(...))Base::function2

Class D1
   size=8 align=8
   base size=8 base align=8
D1 (0x0x7f00526ede38) 0 nearly-empty
    vptr=((& D1::_ZTV2D1) + 16u)
  Base (0x0x7f00526d03c0) 0 nearly-empty
      primary-for D1 (0x0x7f00526ede38)

Vtable for D2
D2::_ZTV2D2: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI2D2)
16    (int (*)(...))D2::~D2
24    (int (*)(...))D2::~D2
32    (int (*)(...))Base::function1
40    (int (*)(...))D2::function2

Class D2
   size=8 align=8
   base size=8 base align=8
D2 (0x0x7f00526edea0) 0 nearly-empty
    vptr=((& D2::_ZTV2D2) + 16u)
  Base (0x0x7f00526d0420) 0 nearly-empty
      primary-for D2 (0x0x7f00526edea0)
```

Virtual table



Base	
0	Base::foo()
1	Base::bar()
2	Base::baz()

и

Inherited	
0	Base::foo()
1	Inherited::bar()
2	Base::baz()
3	Inherited::qux()

```cpp
class A
{
  virtual ~A() {}

  virtual void foo()  { cout << "A::foo()" << endl; }
  virtual void bar()  { cout << "A::bar()" << endl; }
  void baz()  { cout << "A::baz()" << endl; }
};

class B : public A
{
  virtual void foo()  { cout << "B::foo()" << endl; }
  void bar()  { cout << "B::bar()" << endl; }
  void baz()  { cout << "B::baz()" << endl; }
};

class C : public B
{
  virtual void foo()  { cout << "C::foo()" << endl; }
  void bar()  { cout << "C::bar()" << endl; }
  void baz()  { cout << "C::baz()" << endl; }
};
```

```
Vtable for A
A::_ZTV1A: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1A)
16    (int (*)(...))A::~A
24    (int (*)(...))A::~A
32    (int (*)(...))A::foo
40    (int (*)(...))A::bar

Class A
   size=8 align=8
   base size=8 base align=8
A (0x0x7f7296d14360) 0 nearly-empty
    vptr=((& A::_ZTV1A) + 16u)

Vtable for B
B::_ZTV1B: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1B)
16    (int (*)(...))B::~B
24    (int (*)(...))B::~B
32    (int (*)(...))B::foo
40    (int (*)(...))B::bar

Class B
   size=8 align=8
   base size=8 base align=8
B (0x0x7f7296d31e38) 0 nearly-empty
    vptr=((& B::_ZTV1B) + 16u)
  A (0x0x7f7296d143c0) 0 nearly-empty
      primary-for B (0x0x7f7296d31e38)

Vtable for C
C::_ZTV1C: 6u entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI1C)
16    (int (*)(...))C::~C
24    (int (*)(...))C::~C
32    (int (*)(...))C::foo
40    (int (*)(...))C::bar

Class C
   size=8 align=8
   base size=8 base align=8
C (0x0x7f7296d31ea0) 0 nearly-empty
    vptr=((& C::_ZTV1C) + 16u)
  B (0x0x7f7296d31f08) 0 nearly-empty
      primary-for C (0x0x7f7296d31ea0)
    A (0x0x7f7296d14420) 0 nearly-empty
        primary-for B (0x0x7f7296d31f08)
```

Dynamic_cast

Dynamic_cast

Syntax:

```
dynamic_cast < new_type > ( expression )
```

Example:

```
class Base{
virtual void Print()  { cout << "Base::print"; }
void SpecificPrint() { cout << "Specific function of the Base class"; } };

class Derived : Base{
void Print() { cout << "Derived::print"; }
void SpecificPrint() { cout << "Specific function of the Derived class"; } };

//downcast
Base* base = new Derived;
Derived* derived = dynamic_cast<Derived*>(base);        or        Derived derived = dynamic_cast<Derived&>(*base);
derived->SpecificPrint();  // call Derived::SpecificPrint()

//upcast
Derived* derived = new Derived;
Base* base = static_cast<Base*>(derived);
```

Thanks for attention