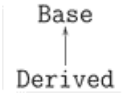
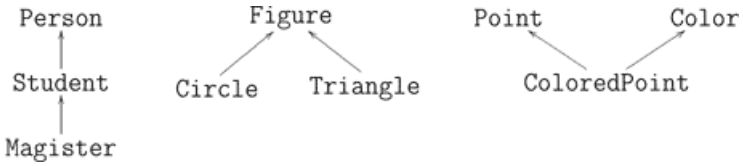
2. Наследование

В такой иерархии:



класс `Base` называют *базовым классом*, *суперклассом* или *предком*. Класс `Derived` соответственно называют *производным классом (подклассом)*, *субклассом* или *потомком*.

(П) Примеры иерархий наследования:



Что наследуется: все данные члены.

Что не наследуется: конструкторы, деструктор, `operator=`, дружественность (друг моего друга — не мой друг!)

(!) Напомним, что если явно не определены четыре особые функции — конструктор по умолчанию, конструктор копий, деструктор и `operator=`, то генерируются предопределённые.

Принцип подстановки (ПП). Если `B` — подкласс `A` (возможно, не прямой), то в любой ситуации, где используется объект класса `A`, может быть использован объект класса `B` с сохранением функциональности.

(П) Сын рудокопа

Принцип подстановки при наследовании крайне желателен, но выполняется не всегда!

Что добавляется и замещается. Данные-члены добавляются, функции-члены могут добавляться и *замещаться*. Замещение также называют *переопределением* (overridding) — сравни с *перегрузкой* (overloading). *Замещающая функция* либо полностью, либо частично замещает одноименную функцию предка.

(П) Переопределение функции

```
class A {
  void f() {...}
} ;

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

Пусть функция `B::f()` *уточняет* поведение соответствующей функции предка. Тогда существует три варианта определения функции `B::f()`:

```
void B::f()  void B::f()  void B::f()
{            {            {
  A::f();      ...          ...
```

```
  ...          A::f();      A::f();
}            }              ...
                          }
```

Явный вызов замещённой функции:

```
B b;
b.A::f();
```

(!) Отметим, что при переопределении параметры функций могут не совпадать. Т. е. замещающая функция переопределяет исходную, но с другим списком параметров. Перегрузка при этом не происходит, т. к. перегрузка возможна только в одном пространстве имён, а производный класс вводит новое пространство имён.

(!) Принцип „Открыт—закрыт“

Принцип „Открыт—закрыт“: закрыт для изменения кода, открыт для модификации поведения через наследование.

Ситуации, в которых используется наследование

(по Бадду, с. 146—151)

1. Порождение подкласса для *специализации*.
B — специализированная форма A; удовлетворяет всем свойствам A и добавляет новые.

(П) `Window ← EditWindow`

Принцип подстановки выполняется.
Это наиболее „чистый“ способ использования наследования.

2. Порождение подкласса для *реализации объявленного поведения* (реализации объявленного, но не определённого в базовом классе поведения).
Некоторые функции в базовом классе могут быть неопределены; их реализация доверяется производным классам.

```
class Figure {
  virtual void draw() = 0; // абстрактный базовый класс (поведение не реализовано)
} ;
```

```
class Circle : public Figure {
  void draw() { /*реализация*/ }
} ;
```

Класс, где все функции объявлены, но не определены, а данные-члены отсутствуют, называется *интерфейсом*.

Принцип подстановки выполняется.
Также наиболее „чистый“ способ использования наследования.

В Java для этих целей введено ключевое слово `interface`.

3. Порождение класса *с целью конструирования*. Подкласс, наследуя поведение базового класса, изменяет имена методов или список параметров. Принцип подстановки *нарушается* (но ведь никто и не собирался использовать производный класс вместо базового!)

(П) Здесь может, и даже должно использоваться включение:

```
class list {          class Stack : private List
  void add(int i);    { //          -------
  int last();           void push(int i) {add(i);}
  ...                   int top() {return last();}
} ;                   } ;
```

(П)

```
class File {                      class IntFile : public File {
  void write(char* p, int bytes);   void write(int i) {
} ;                                   File::write(&i, sizeof int);
                                    }
                                  } ;
```

(!) В последнем примере перегрузки не произойдёт! Функция `write()` подкласса переопределит `write()` базового класса с другим набором параметров! После этого функцией `write()` базового класса будет нельзя пользоваться, по крайней мере неявно. Причина: перегрузка возможна только в одном пространстве имён, а каждый класс представляет собой отдельное пространство имён.

4. Порождение класса для *ограничения*. Какие-то операции базового класса запрещаются в произвольном классе. Принцип подстановки нарушается!

(П) 

5. Порождение подкласса для *варьирования*.

```
class Mouse {         class Plotter : public Mouse {
  MoveTo(x, y);         MoveTo(x, y);
} ;                   } ;
```

Потомок не является разновидностью базового класса! Используется когда два класса имеют сходную реализацию, но нет видимой иерархической связи между понятиями, ими представляемыми.

Лучшее решение — вынести общее в базовый класс (см. пункт 2 — специализация), но оно не всегда возможно: например, `Mouse` был написан ранее и код его недоступен.

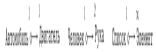
6. Порождение подкласса для *комбинирования* (множественное наследование интерфейсов).

(П) 

7. *Подмешивание*.

(П) 

Сравнение отношений „Быть экземпляром“ (is-a) и „Включать как часть“ (has-a).

(П) 

(!) Сравнение наследования и композиции см. GoF, с. 32, п. „Механизмы повторного наследования“.

Наследование интерфейса и наследование реализации. (GoF, с. 28—32).

Интерфейс — множество сигнатур всех определённых для объекта опреаций.

Тип — имя, используемое для обозначения конкретного интерфейса.

Класс — это интерфейс + *реализация* + *внутреннее состояние*.

У одного объекта может быть несколько типов. Сильно отличающиеся объекты могут разделять общий тип.

Тип `A` является *подтипом* типа `B`, если интерфейс типа `A` содержит интерфейс типа `B`. Для подтипов действует принцип подстановки!

При наследовании класса реализация объекта определяется в терминах реализации другого объекта.

При наследовании интерфейса (порождение подтипов) определяется, когда один объект можно использовать вместо другого.

В большинстве языков программирования (и в C++) различие между наследованием интерфейса и наследованием реализации не поддерживается на уровне языка. В C++ интерфейс моделируется абстрактным классом, в котором все функции — *чисто виртуальные*.

Недостатки наследования.
1) Нельзя изменить унаследованную реализацию во **NIY** (неразборчиво).
2) Наследование нарушает инкапсуляцию, т. к. подклассу доступны некоторые детали реализации базового класса.

Преобразование типов в иерархии `Base ← Derived`.

```
struct B {
  int a, b;
} ;

struct D : B {
  int c;
} ;

B b;
D d;
b = d; // усечение: b.a = d.a; b.b = d.b; точнее, b = (B)d;
B* pb = &d;
B& rb = d;
```

Таким образом, неявно преобразовываются `D` к `B`, `D&` к `B&` и `D*` к `B*`.

(П) `Person ← Student`

Смотри листинг в файле `inheritance.cpp`.

```
#include <stdio.h>
#include <iostream>
#include <string.h>

using namespace std;

char* mystrdup (const char* s)
{
  return strcpy(new char[strlen(s)+1], s);
}

class Person {
  char* _name;
  int _age;
  void init(char* nm, int ag) {
    _name = mystrdup(nm);
    _age = ag;
  }
  void destroy() {delete[] _name;}
public:
  Person(char* nm, int ag) {init(nm, ag);}
  ~Person() {destroy();}
  Person(const Person& p) {init(p._name, p._age);}
  Person& operator=(const Person& p);
  const char* name() const {return _name;}
  int age() const {return _age;}
  void print() const;
} ;

class Student: public Person {
  char* _univ;
  int _course, _group;
  Person chief;
  void init(char* un, int cr, int gr) {
    _univ = mystrdup(un);
    _course = cr;
    _group = gr;
  }
  void destroy() {delete[] _univ;}
public:
  Student(char* nm, int ag, char* un, int cr, int gr, char* cnm, int cag) :
      Person(nm, ag), chief(cnm, cag) {init(un, cr, gr);}
  ~Student() {destroy();}
  Student(const Student& s) : Person(s), chief(s.chief) {
    init(s._univ, s._course, s._group);
  }
  Student& operator=(const Student& s);
  const char* univ() const {return _univ;}
  int course() {return _course;}
  int group() {return _group;}
  void print() const;
} ;

Person& Person::operator=(const Person& p) {
  if (this != &p) {
    destroy();
    init(p._name, p._age);
  }
  return *this;
}
```

```
void Person::print() const {
  cout << _name << " " << _age;
}

Student& Student::operator=(const Student& s) {
  if (this != &s) {
    *static_cast<Person*>(this) = s; // Person::operator=(s); – так лучше
    destroy();
    init(s._univ, s._course, s._group);
    chief = s.chief;;
  }
  return *this;
}

void Student::print() const {
  Person::print();
  cout << endl << _univ << endl << _course << "  " << _group << endl;
  chief.print();
}

void main() {
  Student s("Ivanov", 20, "RSU", 3, 1, "Mikhalkovich", 36);
  s.print();
}
```

(!) Обратите внимание на конструкторы и `operator=` в классах-потомках.

Порядок вызова конструкторов и деструкторов:
1. Конструктор базового класса (вне зависимости от порядка в списке инициализации)
2. Конструктор подобъектов (вне зависимости от порядка в списке инициализации)
3. Конструктор потомка

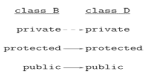
Деструкторы вызываются в порядке, обратном порядку вызрва конструкторов. Если мы не определяем деструктор, он генерируется и вызывает деструкторы подобъектов, затем деструктор базового класса.

Вид доступа `protected`: `public` для наследников и их `friend`-функций, `private` для остальных.

Public-, protected- и private-наследование. (См. Страуструп, 15.3 „Управление доступом“.)

```
          | public    |    // по умолчанию для struct
class D : | protected | B
          | private   |    // по умолчанию для class
```

Public-наследование



Protected-наследование

Private-наследование

Преобразование типов в иерархии `Base` ← `Derived`. Продолжение

1. При private- и protected-наследовании нельзя преобразовывать `D&` в `B&` и `D*` в `B*`. Точнее, можно, но только в функциях-членах класса `D` и производных от него. А преобразовывать `D` в `B`? Тоже, по идее, нет!

2. Раннее связывание.

```
class A {
  void f();
} ;

class B {
  void f();
  void g();
} ;

  ...
  B b;
  A* a = &b;
  a->f();          // вызывается A::f()
  a->g();          // ошибка!
  static_cast<B*>(a)->g(); // нужно вот так
```

(?) Последняя строчка — пример так называемого „downcast“. Что такое „downcast“ и „upcast“?

(?) Что такое раннее и позднее связавние? Связываются имя метода и код функции.

Множественное наследование

Диаграммы наследования

(П) Множественное наследование

Смотри файл `ColorCircle.doc`.

```
#include <iostream.h>
#include <string.h>
class Point
{
  int x, y;
public : Point(int x, int y) {
  cout < endl < "Конструктор Point";
  this->x=x;
  this->y=y;
}
  ~Point() { cout << endl < "Деструктор Point"; }
} ;

class ColorPoint: public Point
{
  int c;
public:
  ColorPoint(int x, int y, int c) : Point(x,y)
  {
    cout << endl << "Конструктор ColorPoint";
    this->c=c;
```

```
  }
  ~ColorPoint() { cout << endl < "Деструктор ColorPoint"; }
} ;

class Circle: public Point
{
  int r;
public:
  Circle(int x, int y, int r): Point(x,y)
  {
     cout << endl << "Конструктор Circle";
     this->r=r;
  }
  ~Circle() { cout << endl << "Деструктор Circle"; }
};

class CircleWithColorPoint: public ColorPoint, public Circle
{
public:
  CircleWithColorPoint(int cx, int cy, int cr,
                       int px, int py, int pc) :
      ColorPoint(px,py,pc), Circle(cx,cy,cr)
  { cout << endl << "Конструктор CircleWithColorPoint"; }
  ~CircleWithColorPoint()
  { cout << endl << "Деструктор CircleWithColorPoint"; }
} ;

void main()
{
  CircleWithColorPoint CC(1,2,3,4,5,6);
}
```

```
Конструктор Point
Конструктор ColorPoint
Конструктор Point
Конструктор Circle
Конструктор CircleWithColorPoint
Деструктор CircleWithColorPoint
Деструктор Circle
Деструктор Point
Деструктор ColorPoint
Деструктор Point
```

(П) Множественное наследование с виртуальным базовым классом

Смотри файл `ColorCircle.doc`.

```
#include <iostream.h>
#include <string.h>
class Point {
  int x, y;
public:
  Point(int x, int y) {
    cout << endl << "Конструктор Point";
    this->x=x; this->y=y;
  }
  ~Point() { cout << endl << "Деструктор Point"; }
} ;

class ColorPoint : virtual public Point
{
  int c;
public:
  ColorPoint(int x, int y, int c): Point(x,y) {
    cout << endl << "Конструктор ColorPoint";
    this->c=c;
```

```
  }
  ~ColorPoint() { cout << endl << "Деструктор ColorPoint"; }
} ;

class Circle : virtual public Point
{
  int r;
public:
  Circle(int x, int y, int r): Point(x,y) {
    cout << endl << "Конструктор Circle";
    this->r=r;
  }
  ~Circle() { cout << endl << "Деструктор Circle"; }
} ;

class ColorCircle : public ColorPoint, public Circle
{
public:
  ColorCircle(int x, int y, int r, int c):
  ColorPoint(5,5,c), Circle(5,5,r), Point(x,y)
  { cout << endl << "Конструктор ColorCircle"; }
  ~ColorCircle()
  { cout << endl << "Деструктор ColorCircle"; }
} ;

void main()
{
  ColorCircle CC(1,2,3,4);
}
```

```
Конструктор Point
Конструктор ColorPoint
Конструктор Circle
Конструктор ColorCircle
Деструктор ColorCircle
Деструктор Circle
Деструктор ColorPoint
Деструктор Point
```