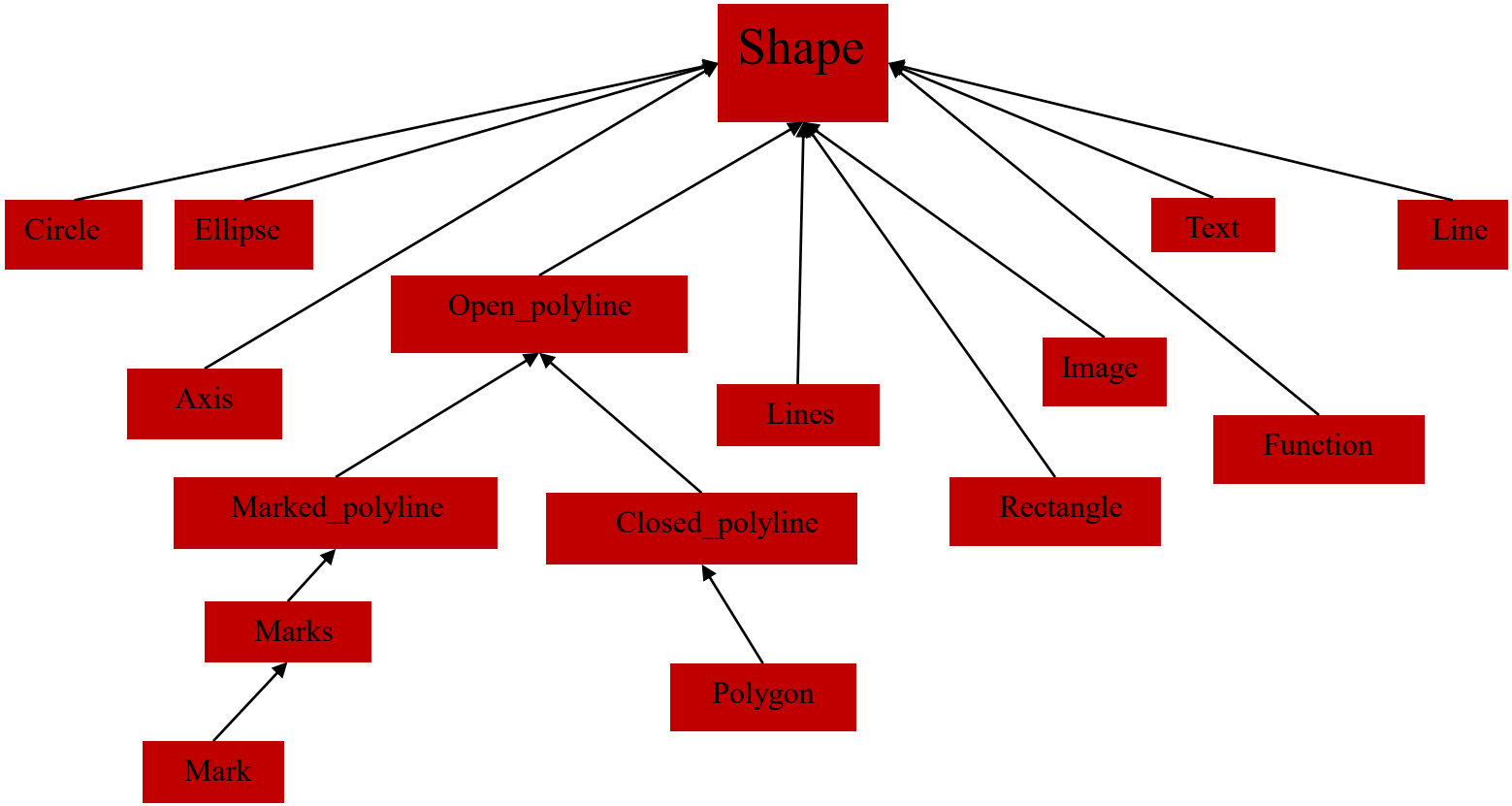
Идеали

- Идеал програмирања је да се концепти из домена примене изразе директно у коду.
  - То би значило да ако разумете домен примене, разумећете и код, и обрнуто. На пример:
    - **Window**
    - **Line**
    - **Point**
    - **Color**
    - **Shape** – оно што је заједничко за све облике у нашем моделу графичког света
- **Shape** се разликује од осталих јер је генерализација.
  - Да ли би могли направити објекат који је само **Shape**?

Класа Shape

- Сви облици су „засновани на“ Shape класи



Класа Shape

- **Window** барата објектима класе **Shape**
  - А сви графички објекти су врста класе **Shape**
- У класи **Shape** се регулише стил линије и боја
  - Поседује **Color** и **Line_style** атрибуте
- **Shape** садржи тачке (објекте класе **Point**)
- **Shape** поседује основну функцију за исцртавање
  - Напросто повезује све тачке које га чине

Класа Shape

- Рад са бојом и стилом:

```
    void set_color(Color col);
    Color color() const;
    void set_style(Line_style sty);
    Line_style style() const;
    // …
  private:
    // …
    Color line_color;
    Line_style ls;
```

Класа Shape

- **Shape** садржи тачке (**Point**s)

```
Point point(int i) const;
int number_of_points() const;
// …
protected:
      void add(Point p);
      // …
private:
vector<Point> points;
```

Класа Shape

- **Shape** приступа тачкама директно:

```
void Shape::draw_lines() const
{
  if (color().visible() && 1 < points.size())
    for (int i = 1; i < points.size(); ++i)
      fl_line(points[i-1].x, points[i-1].y, points[i].x, points[i].y);
}
```

- Остале класе (укључујући и наследнице) користе **point()** и **number_of_points()**. Нпр.:

```
void Lines::draw_lines() const // црта линију за сваки пар тачака
{
  for (int i = 1; i < number_of_points(); i += 2)
    fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

Класа Shape (уопштени алгоритам цртања)

```
void Shape::draw() const
  // Срце класе Shape
  // позива се само од стране класе Window
{
  // … сачувај тренутну боју и стил …
  // … постави боју и стил облика …

  // … нацртај посебности конкретног облика …
  // … напомена: ово јако варира од облика до облика …
  // … нпр. Text, Circle, Closed_polyline

  // … врати сачувану боју и стил …
}
```

Класа Shape (имплементација цртања)

```
void Shape::draw() const
{
  Fl_Color oldc = fl_color();
  // не постоји згодан начин да се добави тренутни стил :(
  fl_color(line_color.as_int());
  fl_line_style(ls.style(), ls.width());

  draw_lines();
     // али сваки конкретан облик има своју функцију draw_lines()

  fl_color(oldc);
  fl_line_style(0);
}
```

Виртуалне функције

- **Shape**

```
virtual void draw_lines() const;
```

- **Circle**

```
void draw_lines() const override { /* draw the Circle */ }
```

- **Text**

```
void draw_lines() const override { /* draw the Text */ }
```

- **Circle**, **Text**, и остале класе изведене из **Shape** могу преклопити **draw_lines()** функцију.

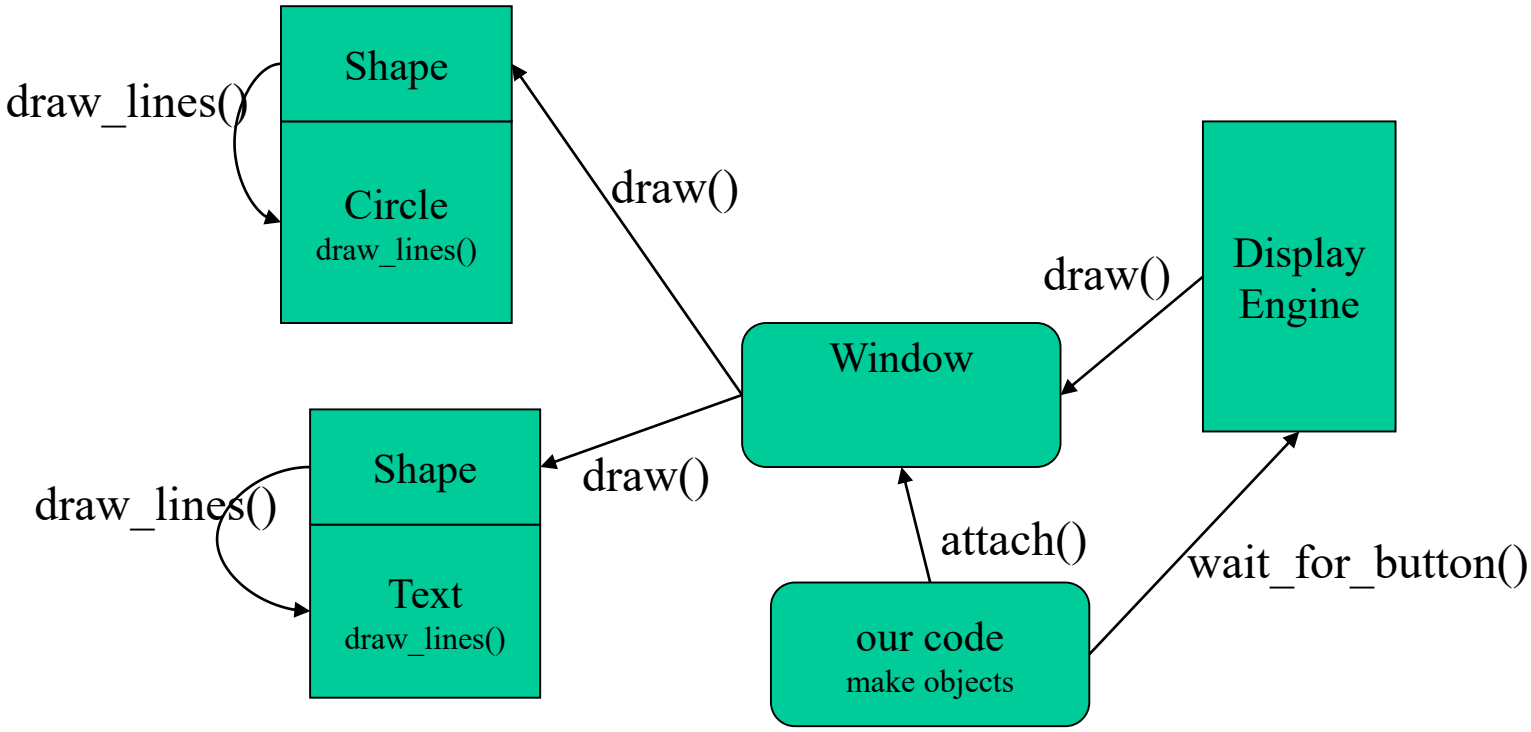
```
class Shape  {
public:
  void draw() const;              // управља бојама и стилом и зове draw_lines()
  virtual void move(int dx, int dy);   // помери облик +=dx и +=dy

  void set_color(Color col);
  int color() const;
  // … остале функције за приступ бојама и стилу …

  Point point(int i) const;       // приступ тачкама у read-only режиму
  int number_of_points() const;
protected:
  Shape();                  // да не би могли направити објекат класе Shape
  void add(Point p);
  virtual void draw_lines() const;
private:
  vector<Point> points;    // не користе га их сви облици
  Color  lcolor;
  Line_style  ls;
  Color  fcolor;

  // … prevent copying …
};
```

Комплетан модел графичког приказа



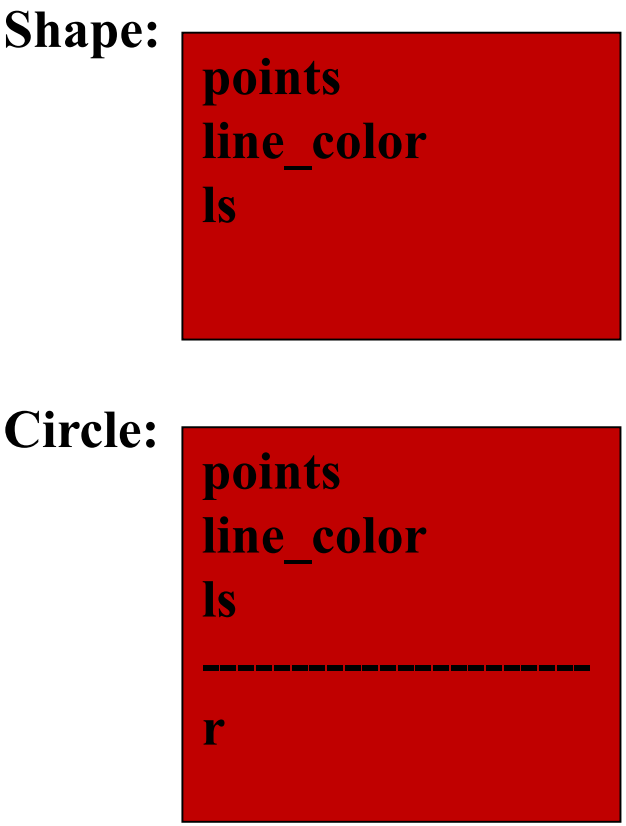
ООП

ООП == наслеђивање + полиморфизам + енкапсулација (скривање података)

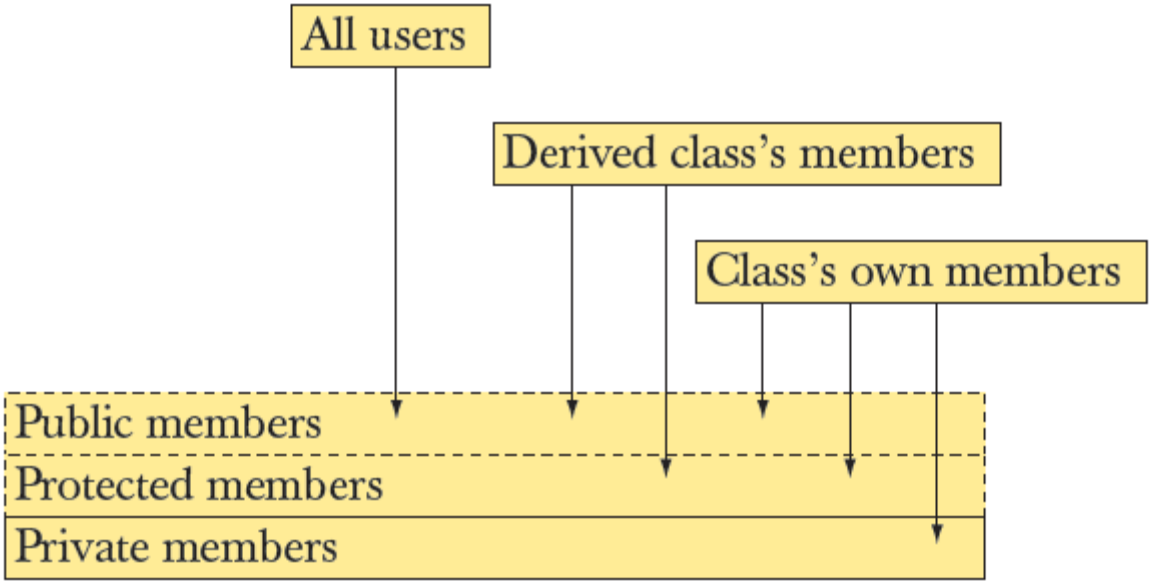
- Базна и изведена класа - наслеђивање
  - `struct Circle : Shape { … };`
- Виртуалне функције - полиморфизам
  - `virtual void draw_lines() const;`
- Private, protected и public – скривање података
  - `protected: Shape();`
  - `private: vector<Point> points;`

Изглед објекта

- Подаци изведене класе се напросто додају на крај базне класе.



Модел приступа



Преклапање

- За преклапање виртуалне функције потребно је
  - декларација виртуалне функције
  - да име буде исто
  - да **тип функције** буде исти

```
struct B {
  void f1();
  virtual void f2(char);
  virtual void f3(char) const;
  virtual void f4(int);
};
```

Тип функције представља број и тип њених параметара и тип повратне вредности.

```
struct D : B {
  void f1();      // не преклапа
  void f2(int);   // не преклапа
  void f3(char);  // не преклапа
  void f4(int);   // преклапа
};
```

Преклапање

```
struct B {
 virtual void f2(char);
 virtual void f3(char) const;
 virtual void f4(int);
};
```

```
 struct D : B {
   void f2(int); // не преклапа и не пријављује грешку
   void f3(char); // не преклапа и не пријављује грешку
   void f4(int);  // преклапа
 };
```

Преклапање

```
struct B {
 virtual void f2(char);
 virtual void f3(char) const;
 virtual void f4(int);
};
```

```
 struct D : B {
   void f2(int) override; // пријављује грешку
   void f3(char) override; // пријављује грешку
   void f4(int) override; // преклапа и даље
 };
```

Преклапање

- За позив виртуалне функције потребан је
  - показивач

```
// за сада само оволико (учићемо мало детаљније ускоро):
B* bptr = &d1;
bptr->f4(2); // позива D::f4(2) над d1 јер bptr показује на неко D

// или на пример
void foo(B* bptr) {
  bptr->f4(2);
}
foo(&d1); // позива D::f4(2) над d1
```

  - или референца

```
void foo(B& bref) {
  bref.f4(2);
}
D d1;
foo(d1); // позива D::f4(2) над d1
```

Чисте виртуалне функције

- Често се дешава да неке функције базне класе немају смислену имплементацију
  - Нпр. неопходни подаци су скривени у изведеним класама
  - Изведена класа мора имплементирати ту функцију
- Класа која има бар једну чисту виртуалну функцију назива се „чистом спрегом“ (енгл. „pure interface“)

```
struct Engine {
 virtual double increase(int i) =0;
 // …
 virtual ~Engine();
};
Engine eee;  // error
```

Конструкторе и деструкторе ћемо детаљније радити мало касније

Чисте виртуалне функције

```
Class M123 : public Engine {
  // ...
public:
  M123();
  double increase(int i) override { /* … */ }// преклапа
  // …
  ~M123();
};

M123 window3_control;  // OK
```