

# Dzisiejszy wykład

- # Klasy pochodne

- # Tekstowy system okienek

- # Zarządzanie zasobami

- Technika "zdobywanie zasobów jest inicjalizacją"
- Wzorzec *auto\_ptr*

# Klasy pochodne

⌘ Pojęcia nie istnieją w izolacji

⌘ Próba opisu pojęcia "samochód" prowadzi do wprowadzenia pojęć:

- kół
- silników
- kierowców
- pieszych
- ciężarówek
- ambulansów
- dróg
- benzyny
- mandatów

⌘ Do reprezentowania pojęć używamy klas. Jak reprezentujemy związki między pojęciami?

- dziedziczenie służy do wyrażania związków hierarchicznych, tj. wspólnych cech klas

# Klasy pochodne

- ✚ Rozważmy problem bazy danych pracowników zatrudnionych w pewnej firmie

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

```
struct Manager {
    Employee emp;
    // manager's employee record
    set<Employee*> group; // people managed
    short level;
    // ...
};
```

- ✚ Kierownik jest też pracownikiem, dane typu *Employee* są zawarte w polu *emp* obiektu typu *Manager*. Z punktu widzenia kompilatora zależność taka jednak nie istnieje. Poprawne podejście polega na jawnym zapisaniu, że kierownik **jest** pracownikiem i dodaniu pewnych informacji

```
struct Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
};
```

# Klasy pochodne

- ❏ Klasa *Manager* **pochodzi** (ang. **is derived**) z klasy *Employee* oraz *Employee* jest klasą **podstawową** (ang. **base class**) klasy *Manager*
- ❏ *Pochodzenie* często reprezentuje się graficznie strzałką od klasy pochodnej do jej klasy podstawowej. Pochodzenie jest również nazywane *dziedziczeniem*



- ❏ Popularną i efektywną implementacją pojęcia klas pochodnych jest przedstawienie obiektu klasy pochodnej jako obiektu klasy podstawowej z dodaną na końcu informacją specyficzną dla klasy pochodnej

**Employee :**

```
first_name
family_name
...
```

**Manager :**

```
first_name
family_name
...
group
level
...
```

# Klasy pochodne

- ✚ Kierownik jest pracownikiem, więc można użyć wskaźnika obiektu klasy *Manager* wszędzie tam, gdzie akceptowalny jest wskaźnik do *Employee*

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist;
    elist.push_front(&m1) ;
    elist.push_front(&e1) ;
    // ...
}
```

- ✚ Konwersja w drugą stronę musi być jawna

```
void g(Manager mm, Employee ee)
{
    Employee* pe= &mm;           // ok: every Manager is an Employee
    Manager* pm= &ee;           // error: not every Employee is a Manager
    pm->level = 2;                // disaster: ee doesn't have a 'level'
    pm = static_cast<Manager*>(pe) ; // brute force: works because pe points
                                   // to the Manager mm
    pm->level = 2; // fine: pm points to the Manager mm that has a 'level'
}
```

# Metody

- ✚ Proste struktury danych, jak *Employee* czy *Manager*, nie są zbyt interesujące ani szczególnie użyteczne. Musimy dostarczyć informacje w postaci typu, z odpowiednim zestawem operacji prezentujących implementowane pojęcie, bez wiązania sobie rąk szczegółami konkretnej reprezentacji

```
class Employee {
    string first_name, family_name;
    char middle_initial;
    // ...
public:
    void print() const;
    string full_name() const
    { return first_name+ " " +middle_initial+ " " + family_name; }
    // ...
};

class Manager : public Employee {
    // ...
public:
    void print() const;
    // ...
};
```

# Metody

- ✚ W klasie pochodnej można używać publicznych i chronionych składowych jej klasy podstawowej, tak jak gdyby były zadeklarowane w klasie pochodnej

```
void Manager::print() const
{
    cout << "name is" << full_name() << '\n';
    // ...
}
```

- ✚ Klasa pochodna nie może jednak używać prywatnych nazw klasy podstawowej, poniższy fragment programu nie skompiluje się

```
void Manager::print() const
{
    cout << " name is" << family_name << '\n'; // error!
    // ...
}
```

# Metody

- ⚡ Najczystszy rozwiązaniem jest używanie przez klasę pochodną jedynie publicznych składowych jej klasy podstawowej

```
void Manager::print() const
{
    Employee::print() ;      // print Employee information
    cout << level; // print Manager-specific information
    // ...
}
```

- ⚡ Operator zasięgu :: jest tu konieczny, bez niego otrzymalibyśmy nieskończone wywołanie rekurencyjne funkcji

```
void Manager::print() const
{
    print() ;      // oops!
    // print Manager-specific information
}
```

# Konstruktory i destruktory

- # Dla niektórych klas pochodnych są potrzebne konstruktory. Jeśli klasa podstawowa ma konstruktor, to należy go wywołać, jeśli dla konstruktora są potrzebne argumenty, to należy je dostarczyć

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& n, int d) ;
    // ...
};

class Manager : public Employee {
    set<Employee*> group; // people managed
    short level;
    // ...
public:
    Manager(const string& n, int d, int lvl) ;
    // ...
};
```

# Konstruktory i destruktory

- Argumenty dla konstruktora klasy podstawowej podaje się w definicji konstruktora klasy pochodnej

```
Employee::Employee(const string& n, int d) : family_name(n) , department(d)
// initialize members
{
// ...
}
Manager::Manager(const string& n, int d, int lvl)
: Employee(n,d) , // initialize base
  level(lvl) // initialize members
{
// ...
}
```

- Konstruktor klasy pochodnej może specyfikować inicjatory tylko dla swoich składowych i bezpośrednich klas podstawowych, nie może bezpośrednio inicjować składowych klasy podstawowej

```
Manager::Manager(const string& n, int d, int lvl):
family_name(n) , // error: family_name not declared in Manager
department(d) , // error: department not declared in Manager
level(lvl)
{
// ...
}
```

- Obiekty będące klasami konstruuje się metodą wstępującą: najpierw klasa podstawowa, potem składowe, a następnie sama klasa pochodna. Niszczy się je w odwrotnej kolejności: najpierw samą klasę pochodną, następnie składowe, a potem podstawową. Klasy składowe i podstawowe konstruuje się w kolejności deklaracji w klasie, a niszczy w kolejności odwrotnej.

# Konstruktory i destruktory

- ❏ Kopiowanie obiektów klasy definiuje się za pomocą konstruktora kopiującego i przypisania

```
class Employee {  
    // ...  
    Employee& operator=(const Employee&) ;  
    Employee(const Employee&) ;  
};  
void f(const Manager&m)  
{  
    Employee e =m;    // construct e from Employee part of m  
    e = m;           // assign Employee part of m to e  
}
```

- ❏ Ponieważ funkcje kopiujące pracowników nie zawierają informacji o kierownikach, więc kopiują jedynie część pracowniczą klasy *Manager*. Powszechnie nazywa się to **wycinaniem** (ang. **slicing**).
- ❏ Jeżeli nie zdefiniuje się operatora przypisania kopiującego, to wygeneruje go kompilator. Oznacza to, że nie można odziedziczyć operatorów kopiowania.
- ❏ W domyślnie wygenerowanym operatorze przypisania najpierw przypisywana jest klasa podstawowa (z użyciem jej operatora przypisania), a następnie składowe, pole po polu.

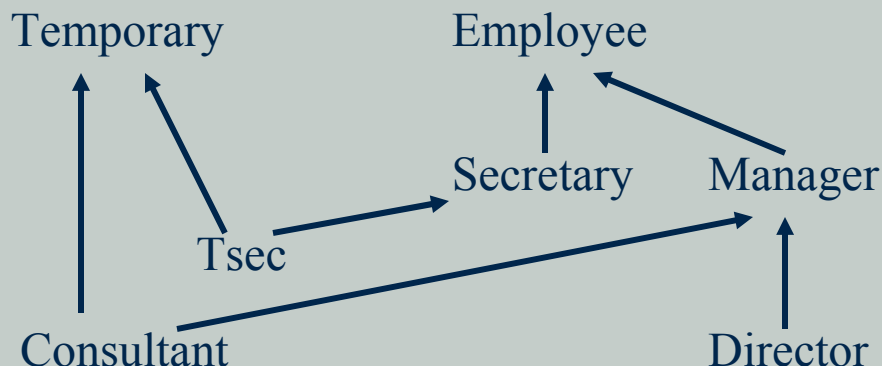
# Hierarchie klas

- ⚡ Klasa pochodna sama może być klasą podstawową

```
class Employee{ /* ... */ };  
class Manager : public Employee{ /* ... */ };  
class Director : public Manager{ /* ... */ };
```

- ⚡ Taki zbiór klas pokrewnych tradycyjnie nazywa się hierarchią klas. Hierarchia jest najczęściej drzewem, lecz może mieć bardziej ogólną strukturę skierowanego grafu acyklicznego

```
class Temporary{ /* ... */ };  
class Secretary : public Employee{ /* ... */ };  
class Tsec : public Temporary, public Secretary{ /* ... */ };  
class Consultant : public Temporary, public Manager{ /* ... */ };
```



# Pola typu

- ⚡ Wskaźniki do klas podstawowych często stosuje się w projektach kolekcji, takich jak zbiór, wektor czy lista.

```
void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
        (*p)->print(); //oops! Prints only the Employee part
}
```

- ⚡ Na liście mogą znajdować się wskaźniki do pracowników i managerów. Powyższa funkcja wypisze jedynie informacje o części pracowniczej. Pole *level* nie zostanie wydrukowane dla managerów.
- ⚡ Co zrobić, żeby funkcja działała zgodnie z naszymi oczekiwaniami?
- ⚡ Rozwiązanie problemu sprowadza się do odpowiedzi na pytanie: na obiekt jakiego typu w rzeczywistości wskazuje wskaźnik?

# Pola typu

- ⚡ Jeżeli mamy wskaźnik typu *Base\**, to do jakiego typu pochodnego należy w rzeczywistości wskazywany obiekt? Istnieją cztery zasadnicze rozwiązania problemu odpowiedzi na to pytanie:
  - Zapewnienie, by wskaźniki wskazywały zawsze tylko na obiekty jednego typu
  - Umieszczenie w klasie podstawowej pola typu, które będzie badane przez funkcje
  - Użycie operatora *dynamic\_cast*
  - Użycie funkcji wirtualnych
- ⚡ Wskaźniki do klas podstawowych często stosuje się w projektach kolekcji, takich jak zbiór, wektor czy lista. Rozwiązanie pierwsze dostarcza listy jednorodne, czyli listy obiektów tego samego typu; drugie, trzecie i czwarte może służyć do budowy list niejednorodnych. Rozwiązanie trzecie jest wariantem rozwiązania drugiego mającym wsparcie językowe, a czwarte bezpiecznym ze względu na typ wariantem drugiego.

# Pola typu

# Dlaczego należy unikać umieszczania w klasach pola typu? Rozważmy przykład:

```
struct Employee {
    enum Empl_type {M,E };
    Empl_type type;
    Employee() : type(E) { }
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
struct Manager : public Employee {
    Manager() { type =M; }
    set<Employee*> group; // people managed
    short level;
    // ...
};
```

# Pola typu

- ✚ Napiszmy funkcję, która umożliwi drukowanie informacji o każdym pracowniku

```
void print_employee(const Employee* e)
{
    switch (e->type) {
        case Employee::E:
            cout << e->family_name << '\t' << e->department << '\n';
            // ...
            break;
        case Employee::M:
            { cout << e->family_name << '\t' << e->department << '\n';
              // ...
              const Manager* p = static_cast<const Manager*>(e) ;
              cout << " level" << p->level << '\n';
              // ...
              break;
            }
    }
}
```

- ✚ Możemy ją teraz wykorzystać do drukowania listy pracowników

```
void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p!=elist.end(); ++p)
        print_employee(*p) ;
}
```

# Pola typu

- # Rozwiązanie sprawdza się w małym programie napisanym przez jedną osobę
- # Zależy od programisty posługującego się typami w sposób, którego kompilator nie może sprawdzić
- # Znalezienie wszystkich instrukcji testujących pole typu, umieszczonych w dużej funkcji, która obsługuje wiele klas pochodnych, może być trudne
- # Dodanie nowego typu pracownika powoduje zmianę wszystkich kluczowych funkcji w systemie

# Funkcje wirtualne

- ❏ Funkcje wirtualne umożliwiają przezwycięzenie problemów, które powstały w rozwiązaniu z zastosowaniem pola typu.
- ❏ Pozwalają one na przedefiniowanie w każdej klasie pochodnej funkcji zadeklarowanych w klasie podstawowej
- ❏ Kompilator wywołuje funkcję wirtualną zdefiniowaną w klasie, na jaką rzeczywiście wskazuje wskaźnik do obiektu

```
class Employee {  
    string first_name, family_name;  
    short department;  
    // ...  
public:  
    Employee(const string& name, int dept) ;  
    virtual void print() const;  
    // ...  
};
```

- ❏ Aby deklaracja funkcji wirtualnej mogła pełnić rolę interfejsu do funkcji zdefiniowanych w klasach pochodnych, typy argumentów podane w deklaracji funkcji w klasie pochodnej nie mogą się różnić od typów argumentów zadeklarowanych w klasie podstawowej, a typ wyniku może się zmienić minimalnie

# Funkcje wirtualne

- ❑ Funkcja wirtualna musi być zdefiniowana dla klasy, w której po raz pierwszy została zadeklarowana

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    // ...
}
```

- ❑ Klasa pochodna, która nie potrzebuje specjalnej wersji funkcji wirtualnej, nie musi jej dostarczać. Wyprowadzając klasę pochodną, wystarczy po prostu dostarczyć właściwą funkcję, jeżeli jest potrzebna

```
class Manager : public Employee {
    set<Employee*> group;
    short level;
    // ...
public:
    Manager(const string& name, int dept, int lvl) ;
    void print() const;
    // ...
};
void Manager::print() const
{
    Employee::print() ;
    cout << "\tlevel" << level << '\n';
    // ...
}
```

# Funkcje wirtualne

- Globalna funkcja `print_employee()` nie jest już potrzebna, gdyż jej miejsce zajęły metody `print()`. Listę pracowników można teraz wydrukować tak:

```
void print_list(set<Employee*>& s)
{
    for (set<Employee*>::const_iterator p = s.begin() ; p!=s.end() ; ++p)
        (*p)->print() ;
}
```

- Albo jeszcze krócej:

```
void print_list(set<Employee*>& s)
{
    for_each(s.begin() ,s.end() ,mem_fun(&Employee::print)) ;
}
```

- Następujący fragment kodu

```
int main()
{
    Employee e("Brown",1234) ;
    Manager m("Smith",1234,2) ;
    set<Employee*> empl;
    empl.insert(&e) ;
    empl.insert(&m) ;
    print_list(empl) ;
}
```

- Wypisze

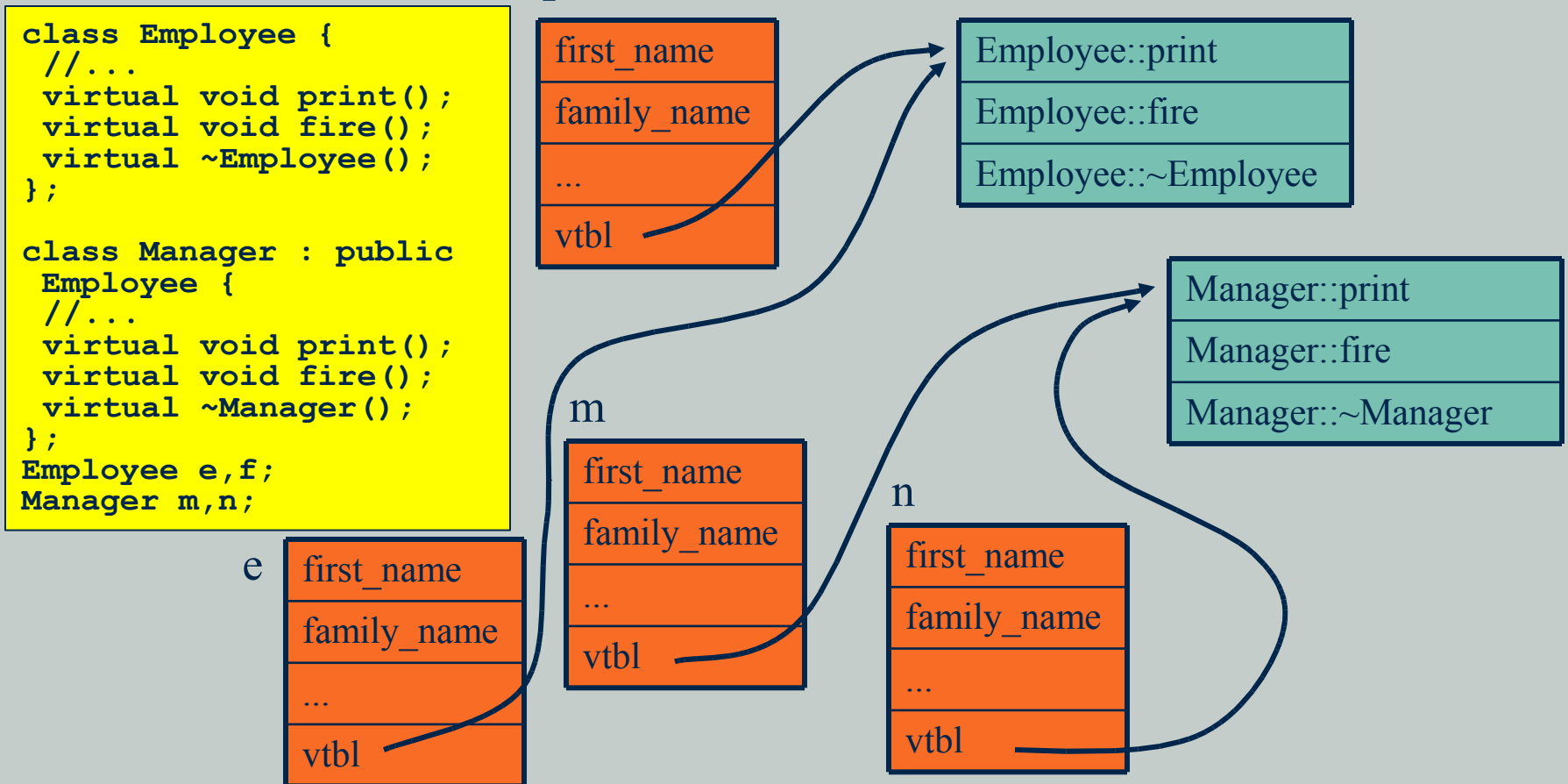
```
Smith 1234
    level 2
Brown 1234
```

# Polimorfizm

- ✚ Uzyskanie "właściwego" zachowania funkcji obsługujących pracownika, zależnie od rzeczywistego typu obiektu, nazywa się **polimorfizmem**. Typ z funkcjami wirtualnymi nazywa się **typem polimorficznym**.
- ✚ Aby uzyskać w C++ polimorficzne zachowanie, wywoływane metody muszą być wirtualne, a do obiektów trzeba się odwoływać przez wskaźniki lub referencje. Jeżeli istnieje bezpośredni dostęp do obiektu, to kompilator zna dokładnie jego typ i polimorfizm czasu wykonania jest zbędny.
- ✚ Do implementacji polimorfizmu wymaga się od kompilatora zapamiętania pewnego rodzaju informacji o typie w każdym obiekcie klasy *Employee*. Zwykle jest to pamięć wystarczająca na przechowanie wskaźnika.
- ✚ Jeżeli funkcję wywołuje się z użyciem operatora zasięgu ::, to nie używa się mechanizmu wirtualnego.

# Polimorfizm

- Typowa implementacja metod wirtualnych polega na dodaniu do każdego obiektu klasy zawierającej co najmniej jedną metodę wirtualną wskaźnika do tablicy metod wirtualnych
- Tablica ta zawiera wskaźniki do wszystkich metod wirtualnych klasy, do której należy dany obiekt



# Klasy abstrakcyjne

- ⚡ Niektóre klasy, takie jak *Employee*, są użyteczne same z siebie, jak i jako klasy podstawowe klas pochodnych
- ⚡ Niektóre klasy reprezentują pojęcia abstrakcyjne. Klasa *Shape* ma sens tylko jako klasa podstawowa jakiejś klasy pochodnej, ponieważ nie można dostarczyć sensownej definicji jej funkcji wirtualnych

```
class Shape {  
    public:  
    virtual void rotate(int) { error("Shape::rotate") ; } // inelegant  
    virtual void draw() { error("Shape::draw") ; }  
    // ...  
};
```

- ⚡ Tworzenie figury takiego niewyspecyfikowanego rodzaju jest niezbyt rozsądne, ale dopuszczalne

```
Shape s; // silly: ``shapeless shape``
```

- ⚡ Każda operacja na *s* spowoduje błąd

# Klasy abstrakcyjne

- ✚ Lepszym rozwiązaniem jest zadeklarowanie funkcji wirtualnych klasy *Shape* jako **czystych funkcji wirtualnych**. Funkcja wirtualna staje się **czysta**, jeśli inicjator ma postać  $=0$ .

```
class Shape{ // abstract class
public:
    virtual void rotate(int) = 0; // pure virtual function
    virtual void draw() = 0; // pure virtual function
    virtual bool is_closed() = 0; // pure virtual function
    // ...
};
```

- ✚ Klasa z jedną lub wieloma czystymi funkcjami wirtualnymi jest klasą abstrakcyjną. Nie można tworzyć żadnych obiektów takiej klasy

```
Shape s; // error: variable of abstract class Shape
```

# Klasy abstrakcyjne

- Klasy abstrakcyjnej można użyć jedynie jako interfejsu i jako klasy podstawowej dla innej klasy

```
class Point{ /* ... */ };
class Circle : public Shape {
public:
    void rotate(int) { }           // override Shape::rotate
    void draw() ;                 // override Shape::draw
    bool is_closed() { return true; } // override Shape::is_closed
    Circle(Point p, int r) ;
private:
    Point center;
    int radius;
};
```

# Klasy abstrakcyjne

- # Czysta funkcja wirtualna, która nie jest zdefiniowana w klasie pochodnej, pozostaje czystą funkcją wirtualną, a więc klasa pochodna jest również klasą abstrakcyjną.
- # Dzięki temu można budować implementacje etapami

```
class Polygon : public Shape{ // abstract class
public:
    bool is_closed() { return true; }          // override Shape::is_closed
                                              // ... draw and rotate not overridden ...
};
Polygon b; // error: declaration of object of abstract class Polygon
class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    void draw() ;                            // override Shape::draw
    void rotate(int) ;                        // override Shape::rotate
    // ...
};
Irregular_polygon poly(some_points) ; // fine (assume suitable constructor)
```

# Klasy abstrakcyjne

- ✚ Klasy abstrakcyjne przydają się do tworzenia interfejsów bez odsłaniania szczegółów implementacyjnych
- ✚ W systemie operacyjnym można ukryć szczegóły funkcji obsługi urządzeń w klasie abstrakcyjnej

```
class Character_device {  
public:  
    virtual int open(int opt) = 0;  
    virtual int close(int opt) = 0;  
    virtual int read(char* p, int n) = 0;  
    virtual int write(const char* p, int n) = 0;  
    virtual int ioctl(int ...) = 0;  
    virtual ~Character_device() { } // virtual destructor  
};
```

- ✚ W klasach pochodnych można teraz specyfikować funkcje obsługi urządzeń i przez ten interfejs posługiwać się nimi
- ✚ Każda klasa posiadająca co najmniej jedną funkcję wirtualną powinna posiadać wirtualny destruktor

# Przykład - tekstowy system okienkowy

- # System okienek pracujący w trybie tekstowym w oparciu o bibliotekę *ncurses*
- # Biblioteka *ncurses* użyta jest w celu umożliwienia sterowania pozycją kursora w sposób przenośny
- # W programie użyto małego podzbioru funkcji oferowanych przez *ncurses*

```
//screen.h
void init_screen ();
void done_screen ();
void gotoyx (int y, int x);
int ngetch ();
void getscreensize (int &y, int &x);
```

```
//ncurses.h
int printf(char *fmt [, arg] ...);
int refresh(void);
```

# Przykład - tekstowy system okienkowy

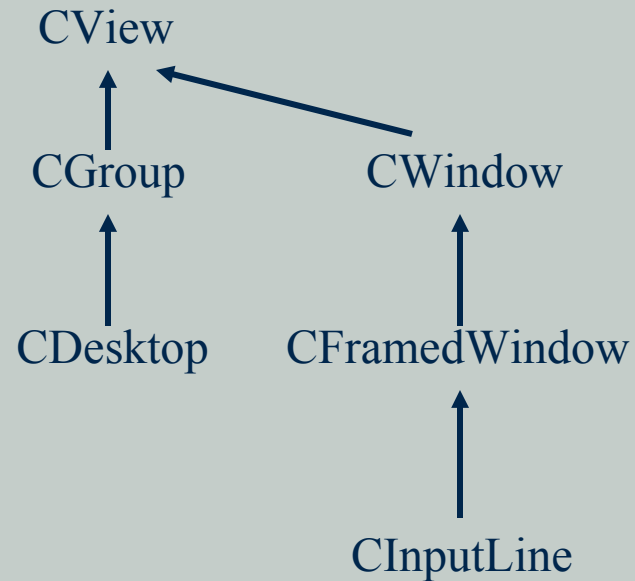
- # W pliku *cpoint.h* zdefiniowano klasy pomocnicze: *CPoint* i *CRect*

```
struct CPoint
{
    int x;
    int y;
    CPoint(int _x=0, int _y=0): x(_x), y(_y) {};
    CPoint& operator+=(const CPoint& delta)
    {
        x+=delta.x;
        y+=delta.y;
        return *this;
    };
};

struct CRect
{
    CPoint topleft;
    CPoint size;
    CRect(CPoint t1=CPoint(), CPoint s=CPoint()): topleft(t1), size(s) {};
};
```

# Przykład - tekstowy system okienkowy

## # Struktura klas w programie



# Przykład - tekstowy system okienkowy

## # *CView* - obiekt widoczny na ekranie

- Pole *geom* opisujące wymiary i położenie widoku
- Funkcja *paint* umożliwiająca wydrukowanie zawartości okna
- Funkcja *handleEvent* obsługująca zdarzenia
- Wirtualny destruktor

```
class CView
{
protected:
    CRect geom;
public:
    CView (CRect g):geom (g)
    {
    };
    virtual void paint () = 0;
    virtual bool handleEvent (int key) = 0;
    virtual void move (const CPoint & delta)
    {
        geom.topleft += delta;
    };
    virtual ~CView () {};
};
```

# Przykład - tekstowy system okienkowy

- # Środowisko sterowane zdarzeniami
- # Reaguje jedynie na polecenia zewnętrzne
  - wciśnięcie klawisza
  - polecenie narysowania wnętrza okna
- # Żaden z obiektów nie robi nic "z własnej inicjatywy"

# Przykład - tekstowy system okienkowy

## # *CWindow* - ruchome okno

```
class CWindow:public CView
{
protected:
    char c;
public:
    CWindow (CRect r, char _c = '*'):CView (r), c (_c) {};
    void paint ()
    {
        for (int i = geom.topleft.y; i < geom.topleft.y + geom.size.y; i++)
            {
                gotoyx (i, geom.topleft.x);
                for (int j = 0; j < geom.size.x; j++)
                    printf ("%c", c);
            };
    };
    bool handleEvent (int key)
    {
        switch (key)
        {
            case KEY UP:
                move (CPoint (0, -1));
                return true;
            case KEY DOWN:
                move (CPoint (0, 1));
                return true;
            case KEY RIGHT:
                move (CPoint (1, 0));
                return true;
            case KEY LEFT:
                move (CPoint (-1, 0));
                return true;
        };
        return false;
    };
};
```

# Przykład - tekstowy system okienkowy

## # *CFramedWindow* - okno z ramką

```
class CFramedWindow: public CWindow
{
public:
    CFramedWindow (CRect r, char _c = '\\'):CWindow (r, _c) {};
    void paint ()
    {
        for (int i = geom.topleft.y; i < geom.topleft.y + geom.size.y; i++)
        {
            gotoyx (i, geom.topleft.x);
            if ((i == geom.topleft.y) || (i == geom.topleft.y + geom.size.y - 1))
            {
                printw ("+");
                for (int j = 1; j < geom.size.x - 1; j++)
                    printw ("-");
                printw ("+");
            }
            else
            {
                printw ("|");
                for (int j = 1; j < geom.size.x - 1; j++)
                    printw ("%c", c);
                printw ("|");
            }
        }
    };
};
```

# Przykład - tekstowy system okienkowy

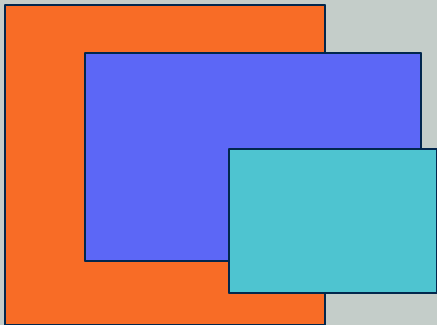
## # *CInputLine* - okienko do wprowadzania tekstu

```
class CInputLine:public CFramedWindow
{
    string text;
public:
    CInputLine (CRect r, char _c = ','):CFramedWindow (r, _c) {};
    void paint ()
    {
        CFramedWindow::paint ();
        gotoyx (geom.topleft.y + 1, geom.topleft.x + 1);
        for (unsigned j = 1, i = 0; (j + 1 < (unsigned) geom.size.x) &&
            (i < text.length ()); j++, i++)
            printf ("%c", text[i]);
    };
    bool handleEvent (int c)
    {
        if (CFramedWindow::handleEvent (c))
            return true;
        if ((c == KEY_DC) || (c == KEY_BACKSPACE))
        {
            if (text.length () > 0)
            {
                text.erase (text.length () - 1);
                return true;
            };
        }
        if ((c > 255) || (c < 0))
            return false;
        if (!isalnum (c) && (c != ' '))
            return false;
        text.push_back (c);
        return true;
    }
};
```

# Przykład - tekstowy system okienkowy

## # *CGroup* - grupa obiektów

```
class CGroup:public CView
{
    list < CView * >children;
public:
    CGroup (CRect g):CView (g) {};
    void paint ()
    {
        for (list < CView * >::iterator i = children.begin ();
            i != children.end (); i++)
            (*i)->paint ();
    };
    //...
};
```



# Przykład - tekstowy system okienkowy

## # *CGroup* - grupa obiektów (cd.)

```
class CGroup:public CView
{
    list < CView * >children;
public:
    CGroup (CRect g):CView (g) {};
    void paint ();
    bool handleEvent (int key)
    {
        if (!children.empty () && children.back ()->handleEvent (key))
            return true;
        if (key == '\t')
        {
            if (!children.empty ())
            {
                children.push_front (children.back ());
                children.pop_back ();
            };
            return true;
        }
        return false;
    }
    //...
};
```

# Przykład - tekstowy system okienkowy

## # *CGroup* - grupa obiektów (cd.)

```
class CGroup:public CView
{
    list < CView * >children;
public:
    CGroup (CRect g):CView (g) {};
    void paint ();
    bool handleEvent (int key);
    void insert (CView * v)
    {
        children.push_back (v);
    };
    ~CGroup ()
    {
        for (list < CView * >::iterator i = children.begin ();
            i != children.end (); i++)
            delete (*i);
    };
};
```

# Przykład - tekstowy system okienkowy

## # *CDesktop* - cały ekran

```
class CDesktop:public CGroup
{
public:
CDesktop ():CGroup (CRect ())
{
    int y, x;
    init_screen ();
    getscreensize (y, x);
    geom.size.x = x;
    geom.size.y = y;
};
~CDesktop ()
{
    done_screen ();
};
void paint()
{
    for (int i = geom.topleft.y;
        i < geom.topleft.y + geom.size.y; i++)
    {
        gotoyx (i, geom.topleft.x);
        for (int j = 0; j < geom.size.x; j++)
            printf (".");
    };
    CGroup::paint();
}
```

```
int getEvent ()
{
    return ngetch ();
};
void run ()
{
    int c;
    paint ();
    refresh ();
    while (1)
    {
        c = getEvent ();
        if (c == 27)
            break;
        if (handleEvent (c))
        {
            paint ();
            refresh ();
        };
    };
};
```

# Przykład - tekstowy system okienkowy

## # Funkcja *main*

```
int main ()
{
    CDesktop d;
    d.insert (new CInputLine (CRect (CPoint (5, 7), CPoint (15, 15))));
    d.insert (new CWindow (CRect (CPoint (2, 3), CPoint (20, 10)), '#'));
    d.run ();
    return 0;
};
```

# Zarządzanie zasobami

- ❑ Kiedy funkcja alokuje zasób - otwiera plik, zamyka semafor, blokuje dostęp, alokuje pamięć - zwykle dla późniejszej poprawnej pracy systemu ważne jest poprawne zwolnienie zasobu. Często funkcja, w której przydzielono zasób, jest również odpowiedzialna za jego zwolnienie

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"w") ;
    // use f
    fclose(f) ;
}
```

- ❑ Powyższe rozwiązanie nie zadziała w przypadku wcześniejszego powrotu z funkcji, np. podczas wystąpienia wyjątku. Poniżej pierwsze podejście do rozwiązania problemu

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"w") ;
    try {
        // use f
    }
    catch (...) {
        fclose(f) ;
        throw;
    }
    fclose(f) ;
}
```

# Zarządzanie zasobami

- ⚡ Rozwiązanie na poprzedniej stronie jest rozwlekłe i nużące, a więc podatne na błędy. Na szczęście istnieje rozwiązanie bardziej eleganckie. Problem w ogólnej postaci wygląda następująco:

```
void acquire()
{
    // acquire resource 1
    // ...
    // acquire resource n

    // use resources

    // release resource n
    // ...
    // release resource 1
}
```

- ⚡ Zwykle wymagane jest zwalnianie zasobów w kolejności odwrotnej do ich przydzielania. Przypomina to zachowanie obiektów lokalnych, tworzonych przez konstruktory i kasowanych przez destruktory.

# Technika zdobywanie zasobów jest inicjalizacją

- # Zdefiniujmy klasę *File\_ptr*, która zachowuje się jak *FILE\**:

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n,a) ; }
    File_ptr(FILE* pp) { p = pp; }
    ~File_ptr() { fclose(p) ; }
    operator FILE*() { return p; }
};
```

- # Program skraca się znacząco

```
void use_file(const char* fn)
{
    File_ptr f(fn,"r") ;
    // use f
}
```

# Zdobywanie zasobów i konstruktory

# Podobną technikę można zastosować w konstruktorach

```
class X {
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x, const char* y)
      : aa(x, "rw") ,          // acquire 'x'
        bb(y)                  // acquire 'y'
    {}
    // ...
};
```

# Często alokowanym w podobny sposób zasobem jest pamięć

```
class Y {
    int* p;
    void init() ;
public:
    Y(int s) {p = new int[s]; init() ; }
    ~Y() { delete[] p; }
    // ...
};
```

```
class Z {
    vector<int> p;
    void init() ;
public:
    Z(int s) : p(s) { init() ; }
    // ...
};
```

# Zwykle lepiej zastosować standardowy wzorzec *vector*

# Wzorzec *auto\_ptr*

- # Wzorzec *auto\_ptr* jest wzorcem z biblioteki standardowej implementującym technikę "zdobywanie zasobów jest inicjalizacją" dla obiektów alokowanych dynamicznie

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb)
    // remember to delete pb on exit
{
    auto_ptr<Shape> p(new Rectangle(p1,p2)) ; // p points to a rectangle
    auto_ptr<Shape> pbox(pb) ;
    p->rotate(45) ; // use auto_ptr<Shape> exactly as a Shape*
    // ...
    if (in_a_mess) throw Mess() ;
    // ...
}
```

# Wzorzec *auto\_ptr*

# *auto\_ptr* jest zadeklarowany w pliku nagłówkowym `<memory>`

```
template<class X> class std::auto_ptr {
template <class Y> struct auto_ptr_ref{ /* ... */ }; // helper class
    X* ptr;
public:
    typedef X element_type;
    explicit auto_ptr(X* p =0) throw() { ptr=p; }
    auto_ptr(auto_ptr& a) throw() { ptr=a.release(); } // note: not const auto_ptr&
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw() { ptr=a.release(); }
    auto_ptr& operator=(auto_ptr& a) throw() { reset(a.release()); return *this;}
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw()
                                                                    { reset(a.release()); return *this;}
    ~auto_ptr() throw() { delete ptr; }
    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; } // extract pointer
    X* release() throw() { X* t = ptr; ptr=0; return t; } // relinquish ownership
    void reset(X* p =0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }
    auto_ptr(auto_ptr_ref<X>) throw() ; // copy from auto_ptr_ref
    template<class Y> operator auto_ptr_ref<Y>() throw() ; // copy from auto_ptr_ref
    template<class Y> operator auto_ptr<Y>() throw() ; // destructive copy from auto_ptr
};
```

# Wzorzec *auto\_ptr*

- ✚ konstruktor wzorca i przypisanie wzorca umożliwiają niejawną konwersję tak samo jak dla wskaźników

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2 = pc; // now p2 is responsible for deletion
    auto_ptr<Circle> p3 = p2;
        // now p3 is responsible for deletion (and p2 isn't)
    p2->m = 7; // programmer error: p2.get()==0
    Shape* ps = p3.get() ; // extract the pointer from an auto_ptr
    auto_ptr<Shape> aps = p3; // transfer of ownership and convert type
    auto_ptr<Circle> p4 = pc;
        // programmer error: now p4 is also responsible for deletion
}
```

- ✚ *auto\_ptr* ze względu na destrukcyjną semantykę kopiowania nie nadaje się do przechowywania w standardowych kolekcjach

```
void h(vector< auto_ptr<Shape*> >& v)
        // dangerous: use of auto_ptr in container
{
    sort(v.begin() ,v.end()) ;
        // Don't do this: The sort will probably mess up v
}
```

# Wzorzec *auto\_ptr*

- # *auto\_ptr* nie nadaje się do przechowywania wskaźników do tablic

```
void g()
{
    auto_ptr<int> p=new int[100];
        // error, delete instead of delete[] invoked on exit from g
}
```

- # do tego celu można stosować wzorzec *vector*

```
void g()
{
    vector<int> a(100);
    int* p=&(a[0]); // guaranteed to point to array of 100 integers
}
```