Klasy pochodne

Na podstawie: Bjarne Stroustrup „Język C++”

- Rozważmy program dotyczący ludzi zatrudnionych w pewnej firmie. W programie mogłaby być zadeklarowana taka struktura danych:

```
struct Pracownik {
    string imię, nazwisko;
    char środkowy_inicjał;
    Data data_zatrudnienia;
    short dział;
    // ...
};
```

- Spróbujmy teraz zdefiniować kierownika:

```
struct Kierownik {
    Pracownik prac;
    set<Pracownik*> grupa;
    short poziom;
    // ...
};
```

- Kierownik jest także pracownikiem; dane typu Pracownik są zapamiętane w polu prac obiektu typu Kierownik.
- Może to być oczywiste dla czytelnika, lecz z punktu widzenia kompilatora i innych narzędzi nic nie wskazuje na to, że Kierownik to także Pracownik.

- Poprawne podejście polega na jawnym zapisaniu, że kierownik *jest* pracownikiem i dodaniu pewnych informacji:

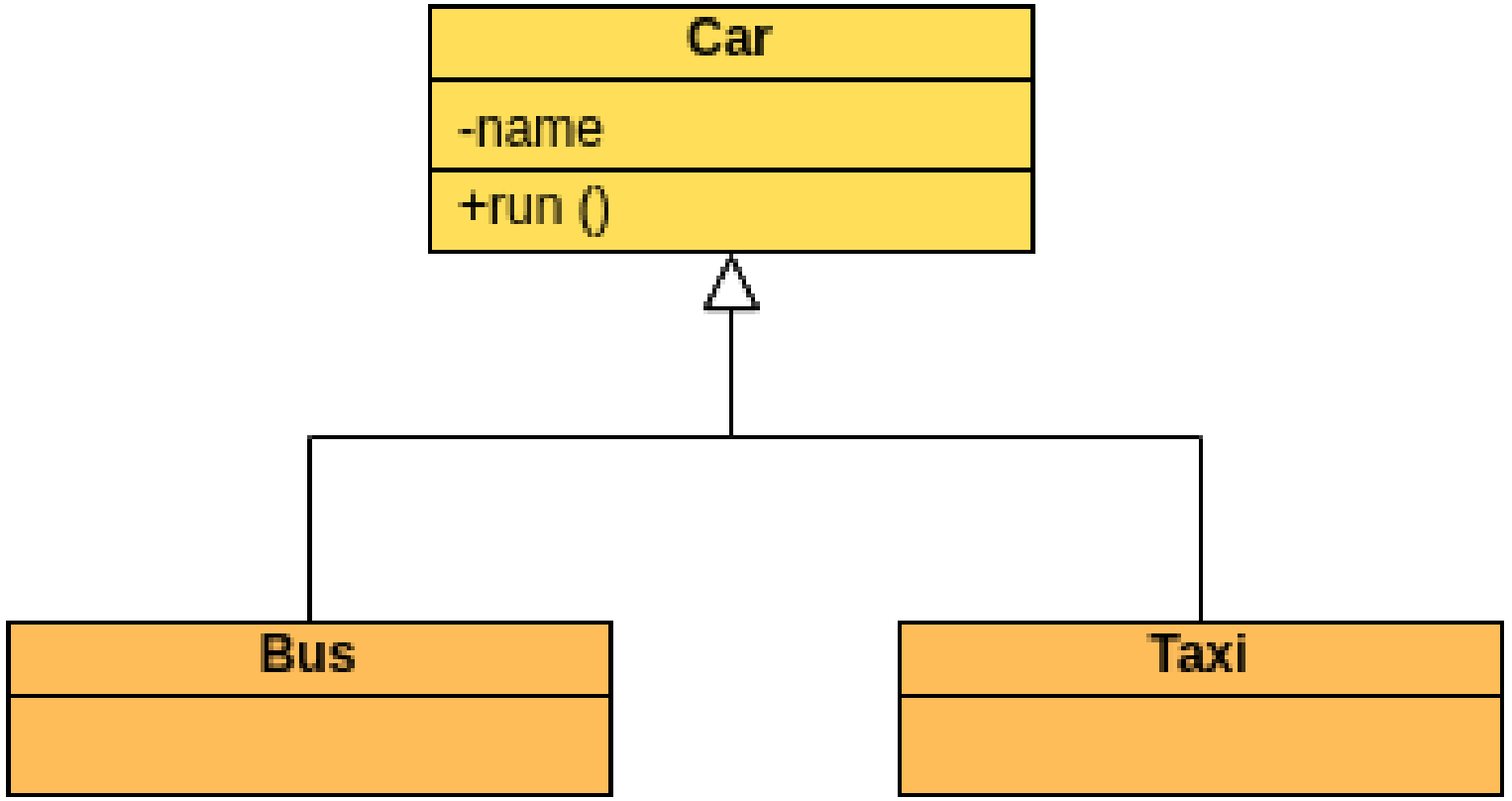
```
struct Kierownik : public Pracownik {
    set<Pracownik*> grupa;
    short poziom;
    // ...
};
```

- Klasa Kierownik ma składowe klasy Pracownik (nazwisko, dział itd.) oraz dodatkowo składową grupa i poziom.

```
struct Kierownik : public Pracownik {
    set<Pracownik*> grupa;
    short poziom;
    // ...
};
```

- Klasa Kierownik pochodzi (ang. derived) od klasy Pracownik i na odwrót Pracownik jest klasą podstawową (ang. base class) klasy Kierownik.
- Często mówi się, że klasa pochodna dziedziczy z klasy podstawowej, a więc relację nazywa się także dziedziczeniem (ang. inheritance).

Diagramy klas - UML



```
void f(Kierownik k1, Pracownik p1)
{
    list<Pracownik*> listap;

    listap.push_front(&k1);
    listap.push_front(&p1);
    // ...
}
```

- Ponieważ kierownik jest (także) pracownikiem, dlatego można używać wskaźnika **Kierownik*** jako **Pracownik***.
  Jednak nie każdy pracownik jest kierownikiem, dlatego nie można używać wskaźnika **Pracownik*** jako **Kierownik***.

- Jeżeli mamy pewność, że dany pracownik jest kierownikiem, możemy zrzutować wskaźnik **Pracownik*** na **Kierownik***.

```
k = static_cast<Kierownik*>(p);
```

Ogólnie:

- Jeśli klasa **Pochodna** ma publiczną klasę podstawową o nazwie **Podstawowa**, to wartość typu **Pochodna*** można przypisać zmiennej typu **Podstawowa*** bez użycia jawnej konwersji typu.
- Konwersja w drugą stronę, z **Podstawowa*** do **Pochodna***, musi być jawna (rzutowanie).

Uwaga:

- obiekt klasy pochodnej można traktować jak obiekt klasy podstawowej, gdy sięga się do niego za pomocą **wskaźników lub referencji**.

Metody

```
class Pracownik {
    string imię, nazwisko;
    char środkowy_inicjał;
    // ...
public:
    void drukuj() const;
    string pełne_nazwisko() const
        { return imię+' '+środkowy_inicjał+' '+nazwisko; }
    // ...
};
```

```
class Kierownik : public Pracownik {
    // ...
public:
    void drukuj() const;
    // ...
};
```

```
void Kierownik::drukuj() const
{
    cout << " Nazwisko brzmi " << pełne_nazwisko() << '\n';
    // ...
}
```

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

```
void Kierownik::drukuj() const
{
    cout << " Nazwisko brzmi " << nazwisko << '\n';
    // ...
}
```

- Powyższy kod *nie* skompiluje się!
- Klasa pochodna *nie może* używać prywatnych składowych klasy podstawowej!

- Możliwe rozwiązanie:

```
void Kierownik::drukuj() const
{
    Pracownik::drukuj();        // drukuj informację o pracowniku
    cout << poziom;             // drukuj informacje charakterystyczne dla kierownika
}
```

Konstruktory i destruktory klas pochodnych

```
class Pracownik {
    string imię, nazwisko;
    short dział;
    // ...
public:
    Pracownik(const string& n, int d);
    // ...
};
```

```
class Kierownik : public Pracownik {
    set<Pracownik*> grupa;   // podwładni
    short poziom;
    // ...
public:
    Kierownik(const string& n, int d, int poz);
    // ...
};
```

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

```
Pracownik::Pracownik(const string& n, int d)
    : nazwisko(n); dział(d)                    // inicjuj pola
{
    // ...
}

Kierownik::Kierownik(const string& n, int d, int poz)
    : Pracownik(n,d),                          // inicjuj klasę podstawową
    poziom(p)                                  // inicjuj pole
{
    // ...
}
```

- 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

Hierarchie klas

Klasa pochodna sama może być klasą podstawową, np.

```
class Pracownik { /* ... */ };
class Kierownik : public Pracownik { /* ... */ };
class Dyrektor : public Kierownik { /* ... */ };
```

Funkcje wirtualne (virtual)

```
class Pracownik {
    string imię, nazwisko;
    short dział;
    // ...
public:
    Pracownik(const string& n, int d);
    virtual void drukuj() const;
    // ...
};
```

Jeśli klasy pochodne zawierają definicje funkcji **drukuj()**, to znalezienie właściwej funkcji dla każdego wywołania **drukuj()** dla danego obiektu typu **Pracownik*** jest zadaniem kompilatora

- Funkcja wirtualna *musi* być zdefiniowana dla klasy, w której po raz pierwszy została zadeklarowana (chyba że jest zadeklarowana jako czysta funkcja wirtualna)

```
void Pracownik::drukuj() const
{
    cout << nazwisko << '\t' << dział << '\n';
    // ...
}
```

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

```
void Kierownik::drukuj() const
{
    Pracownik::drukuj();
    cout << "\tpoziom" << poziom << '\n';
    // ...
}
```

- Funkcja w klasie pochodnej z tą samą nazwą i tym samym zbiorem typów argumentów co funkcja wirtualna w klasie podstawowej przesłania (ang. override) wersję funkcji wirtualnej z klasy podstawowej.
- Funkcję zastępującą wybiera się jako tę najbardziej właściwą z punktu widzenia obiektu, dla którego sieją wywołuje, z wyjątkiem sytuacji gdy jawnie wskazuje się wersję wywoływanej funkcji wirtualnej (jak w **Pracownik: : drukuj ()**).

```
void drukuj_listę (const list<Pracownik* > & s)
{
    for (list<Pracownik* > : :const_iterator p=s .begin ( ) ; p !=s .end ( ) ; ++p)

        (*p) ->drukuj ( ) ;
}
```

```
int main ( )
{
    Pracownik p ("Brown" , 1234) ;
    Kierownik k ("Smith" , 1234 , 2) ;
    list<Pracownik* > prac ;
    prac .push_front (&p) ;
    prac .push_front (&k) ;
    drukuj_listę (prac) ;
}
```

WYNIK

```
Smith  1234
        poziom  2
Brown  1234
```

- Uzyskanie „właściwego" zachowania funkcji obsługujących pracownika, niezależnie od tego, który dokładnie Pracownik jest używany, nazywa się *polimorfizmem* (ang. polymorphism).
- Typ z funkcjami wirtualnymi nazywa się *typem polimorficznym*. Aby uzyskać w C++ polimorficzne zachowanie, wywoływane metody muszą być wirtualne, a do obiektów trzeba się dostawać przez wskaźniki lub referencje.
- Jeśli istnieje bezpośredni dostęp do obiektu (a nie przez wskaźnik lub referencję), to kompilator zna dokładnie jego typ i polimorfizm czasu wykonania jest zbędny.