Język C++

część 5
dziedziczenie

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- dziedziczenie - wprowadzenie i podstawy

Motywacja:

- bardzo wiele opisywanych w pisanym programie aspektów rzeczywistości ma w naturalny sposób charakter hierarchiczny
- mechanizmy dziedziczenia dostarczają więc "naturalnego sposobu" na opisanie rzeczywistości, w której działamy

Kilka podstawowych faktów technicznych:

- dziedziczenie pozwala utworzyć nową klasę (klasa pochodna) w oparciu o klasę już istniejącą (klasa bazowa)
- klasa pochodna automatycznie dziedziczy właściwości swojej klasy bazowej (pola i metody)
- dodając nowe pola i /lub metody do klasy pochodnej, uzupełniamy możliwości klasy bazowej
- utworzenie klasy pochodnej nie zmienia w żaden sposób klasy bazowej

- dziedziczenie - opis zadania

Załóżmy, że potrzebny jest program zarządzający pracownikami firmy X. Założenia szczegółowe:

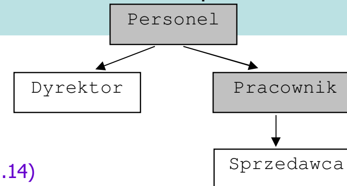
W programie należy zapamiętać nazwisko (`nazwisko`) pracownika i numer pokoju (`biuro`) w którym pracownik przebywa.
Na podstawie numeru pokoju, odpowiednia funkcja ma wyświetlić numer telefonu do pracownika (baza danych numer pokoju / telefon może być „zaszyta" w kodzie programu).

Firma X zatrudnia trzy rodzaje pracowników opłacanych w różny sposób:

Pracownicy pracujący na akord. Wypłata pracownika (pensja) równa się liczbie przepracowanych w miesiącu godzin (`godziny`) razy stawka godzinowa (`stawka`).

Sprzedawcy. Wypłata równa się stałej pensji (wyliczonej jako ustalona stawka razy ustalona liczba godzin – tak jak przy Pracowniku) + dodatek wyliczony w oparciu o procent (`procent`) zrealizowanej w danym miesiącu sprzedaży (`sprzedaz`).

Dyrektorzy. Wypłata równa się stałej pensji (`pensja`) + premia (`premia`) uzależniona od liczby podległych mu pracowników (`ile_pracowników`). Uwaga: pensja i premia Dyrektora jest ustalana z góry i nie zależy od godzin, stawki, procentów jak w przypadku Pracowników i Sprzedawców zdefiniowanych wyżej.

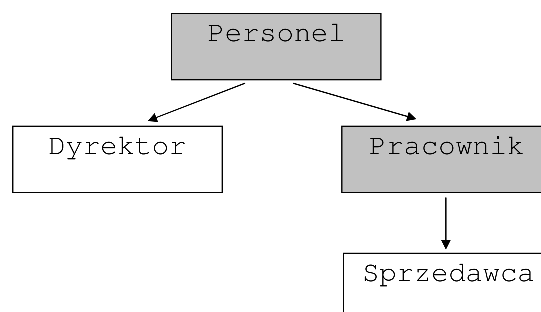


- dziedziczenie - rozwiązanie zadania
  - Na następnych stronach przedstawiono fragment kodu programu, zawierający:

    1. przykładową klasę `Personel`, z polami odnoszącymi się do wszystkich pracowników czyli pola `nazwisko` i `biuro`. Umieszczone w klasie funkcje pozwalają wykonywać założone czynności – wyświetlić informacje o pracowniku i uzyskać jego numer telefonu.

    2. przykładową klasę `Pracownik`, dziedziczącą po klasie `Personel` zawierającą pola i funkcje stosowne do opisu pracownika pracującego na akord.

    3. przykładową funkcję main(), w której korzysta się z opisanych wyżej klas.

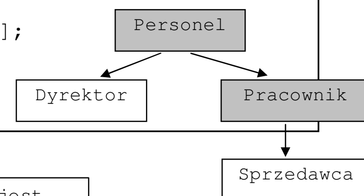


- dziedziczenie - rozwiązanie zadania

```
class Personel
{
public:
    Personel( const char* _nazwisko, short _biuro = 0 );
    ~Personel() { delete [] nazwisko; }
    void print() const;
    void set_biuro( short _biuro ) { biuro = _biuro; }
    const char* get_telefon() const;
private:
    char* nazwisko;
    short biuro;
};

Personel::Personel( const char* _nazwisko, short _biuro)
                  : biuro( _biuro )
{
    nazwisko = new char [ strlen( _nazwisko) + 1 ];
    strcpy( nazwisko, _nazwisko );
}
```

porównaj uwagi z wykładu #4 dotyczące funkcji const

lista inicjalizacyjna nie jest ograniczona jedynie do pól const (patrz poprzedni wykład)



- dziedziczenie - rozwiązanie zadania

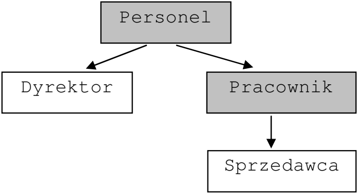
```
void Personel::print() const
{
    cout << "Pracownik " << nazwisko << " :" << endl;
    cout << "biuro = " << biuro << ", "
         << "tel. = " << get_telefon() << endl;
}

const char* Personel::get_telefon() const
{
   static const char* tel[] = {"0000", "1111", "2222", "3333", "4444"};
   // symulacja uzyskiwania numeru telefonu
   return tel[ biuro ];
}
```

- dziedziczenie - rozwiązanie zadania

nadpisana funkcja

dziedziczenie publiczne (najczęściej stosowane). Dziedziczenie prywatne i chronione też jest możliwe. Będzie omawiane później.

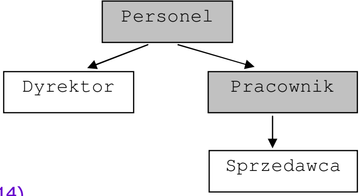
```
class Pracownik : public Personel
{
public:
    Pracownik( const char* _nazwisko, int _biuro = 0,
               float _stawka = 0.0, float _godziny = 0.0 );
    void print() const;
    void set_stawka( float _stawka ) { stawka = _stawka; }
    void set_godziny( float _godziny ) { godziny = _godziny; }
    float oblicz_place() const;
private:
    float stawka;
    float godziny;
 };
```



- dziedziczenie - rozwiązanie zadania

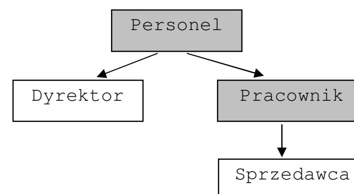
```
Pracownik::Pracownik( const char* _nazwisko, int _biuro,
                      float _stawka, float _godziny )
                    : Personel( _nazwisko, _biuro ),
                      stawka( _stawka ), godziny( _godziny )
{ }

void Pracownik::print() const
{
    Personel::print();
    cout << "Stawka godzinowa = " << stawka << ", "
         << "Liczba przepracowanych godzin = " << godziny << endl;
}

float Pracownik::oblicz_place() const
{
    return stawka * godziny;
}
```

zwykle (ale niekoniecznie) wywoływana jest w takim miejscu nadpisana funkcja z klasy nadrzędnej

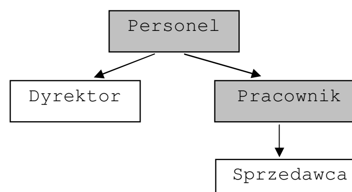


- dziedziczenie - rozwiązanie zadania

```
void main()
{
    Personel p1( "Kowalski", 2 );

    p1.print();             // Personel::print()
    p1.set_biuro( 5 );      // Personel::set_biuro()
    p1.print();             // Personel::print()

    Pracownik e1( "Malinowski", 3, 70.0, 169.0 );

    e1.print();             // Pracownik::print()
    e1.set_biuro( 4 );      // Personel::set_biuro()    // 3 -> 4
    e1.set_stawka( 75.0 ); // Pracownik::set_stawka() // 70.0 -> 75.0
    e1.print();             // Pracownik::print()
}
```



- dziedziczenie - analiza rozwiązania
  - klasa `Pracownik`, definiując własne pola (`stawka, godziny`) i metody (`set_stawka(), set_godziny(), oblicz_place()`) uzupełnia podstawowy opis pracownika i podstawowe operacje wykonywane na obiekcie `Personel`
  - metoda `print()` klasy `Pracownik` zastępuje metodę `print()` z klasy `Personel`
  - w metodzie `Pracownik::print()` korzystamy z metody `Personel::print()`

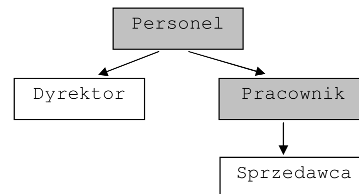


- dziedziczenie - analiza rozwiązania
  - każdy obiekt klasy pochodnej ma dostęp do wszystkich elementów publicznych swojej klasy bazowej (z reguły będą to metody, rzadziej pola)

```
Pracownik e1( "Kowalski", 3, 70.0, 169.0 );
e1.print();              // Pracownik::print()
e1.set_biuro( 4 );       // Personel::set_biuro()
```

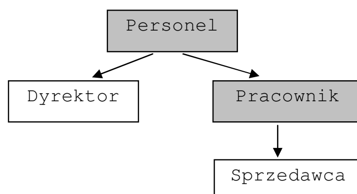
  - klasa pochodna nie może korzystać z elementów prywatnych swojej klasy bazowej. Musi "zadowolić się" zawartością sekcji publicznej (czyli interfejsem !)

```
void Pracownik::print() const
{
    // Personel::print(); tak jest teraz...

    // a tak będzie źle...
    // BLAD: Personel::nazwisko jest w sekcji private
    cout << "Pracownik" << nazwisko << " :" ,, endl;
}
```

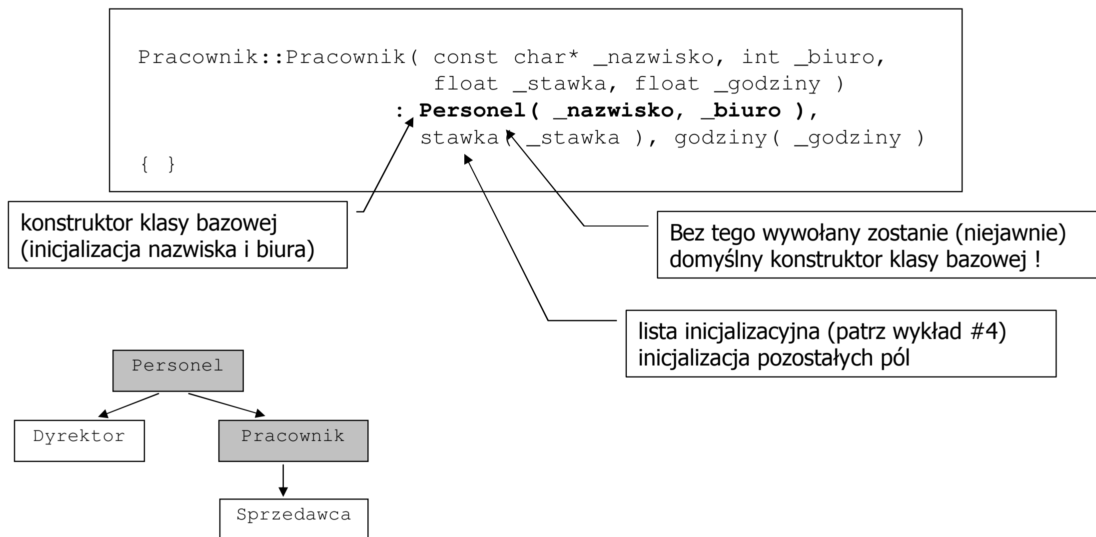


- dziedziczenie - analiza rozwiązania
  - użycie zamiast `Personel::print()` samego `print()` spowoduje zapętlenie się programu. Operator zakresu `::` jest więc niezbędny.
    Taka sytuacja zachodzi gdy klasa pochodna redefiniuje metodę z klasy bazowej

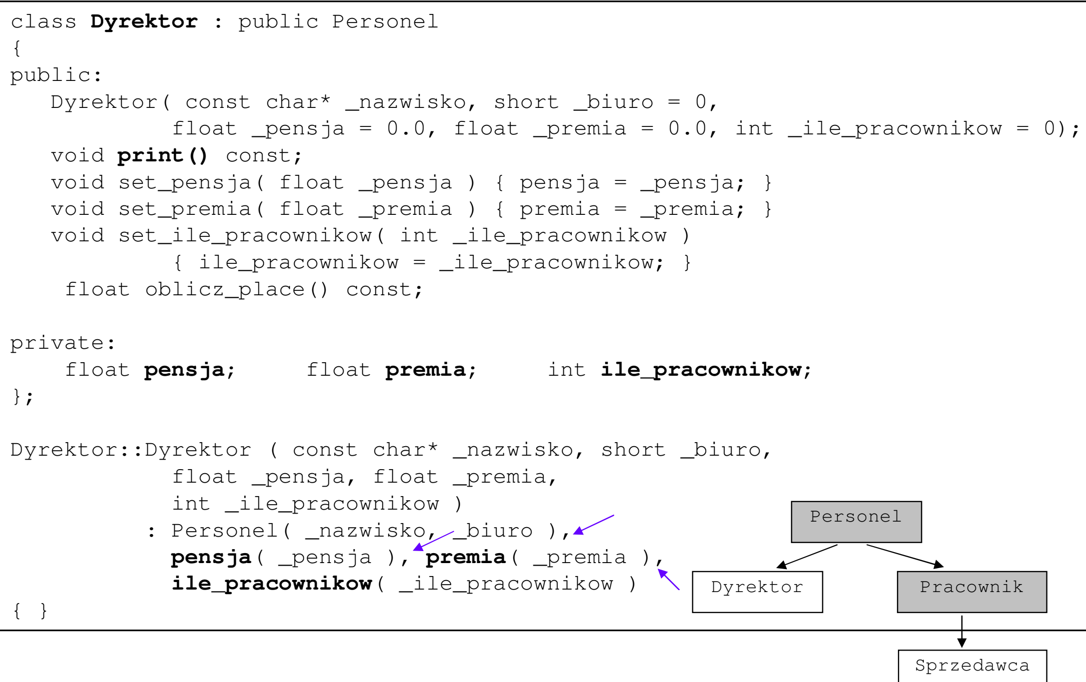
```
void Pracownik::print() const
{
    Personel::print(); // OK
    print();           // BLAD, wywołujemy Pracownik::print();
}
```



- dziedziczenie - analiza rozwiązania
  - jeśli klasa bazowa definiuje własny konstruktor, to musi on zostać wywołany w czasie tworzenia obiektu klasy pochodnej (wtedy poprawnie zainicjalizują się dane, które obiekt dziedziczy)
  - należy więc w konstruktorze klasy pochodnej umieścić wywołanie konstruktora klasy bazowej (przekazując mu ewentualne parametry)

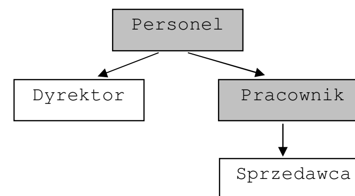


- pozostałe klasy przykładu



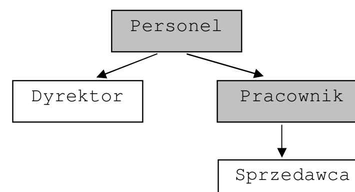
- pozostałe klasy przykładu

```
void Dyrektor::print() const
{
    Personel::print();
    cout << "pensja = " << pensja << ", "
    << "premia = " << premia << endl;
    cout << "Liczba nadzorowanych pracownikow = " << ile_pracownikow << endl;
}

float Dyrektor::oblicz_place() const
{
    return pensja + premia;
}
```



- pozostałe klasy przykładu

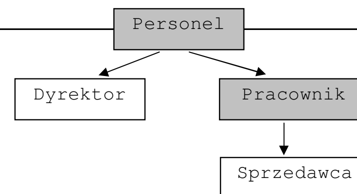
```
class Sprzedawca : public Pracownik
{
public:
    Sprzedawca( const char* _nazwisko, short _biuro = 0,
                float _stawka = 0.0, float _godziny = 0.0,
                float _procent = 0.0, float _sprzedaz = 0.0 );
    void print() const;
    void set_procent( float _procent ) { procent = _procent; }
    void set_sprzedaz( float _sprzedaz ) { sprzedaz = _sprzedaz; }
    float oblicz_place() const;
private:
    float procent;
    float sprzedaz;
};

Sprzedawca::Sprzedawca( const char* _nazwisko, short _biuro,
        float _stawka, float _godziny,
        float _procent, float _sprzedaz )
      : Pracownik( _nazwisko, _biuro, _stawka, _godziny ),
        procent( _procent ), sprzedaz( _sprzedaz )
{ }

void Sprzedawca::print() const {
    Pracownik::print();
    cout << "procent = " << procent << ", "
    << "od sprzedazy = " << sprzedaz << endl;
}

float Sprzedawca::oblicz_place() const {
    return Pracownik::oblicz_place() + procent * sprzedaz;
}
```



- program główny

Sprzedawca ma bezpośredni dostęp do funkcji `Personel::set_biuro()` a `set_biuro()` **NIE** została przedefiniowana ani w klasie `Pracownik` ani w klasie `Sprzedawca`

```
void main()
{
    Sprzedawca s1( "Kowalski" );

    s1.set_biuro( 1 );            // Personel::set_biuro()
    s1.set_stawka( 90.0 );        // Pracownik::set_stawka()
    s1.set_godziny( 169.0 );      // Pracownik::set_godziny()
    s1.set_procent( .05 );        // Sprzedawca::set_procent()
    s1.set_sprzedaz( 10000.0 );   // Sprzedawca::set_sprzedaz()
    s1.print();                   // Sprzedawca::print()
    cout << "Placa = " << s1.oblicz_place() << endl;

    Dyrektor d1( "Malinowski" );

    d1.set_biuro( 2 );             // Personel::set_biuro()
    d1.set_pensja( 15000.0 );      // Dyrektor::set_pensja()
    d1.set_premia( 1500.0 );       // Dyrektor::set_premia()
    d1.set_ile_pracownikow( 25 );  // Dyrektor::set_ile_pracownikow()
    d1.print();                    // Dyrektor::print()
    cout << "Placa = " << d1.oblicz_place() << endl;
}
```



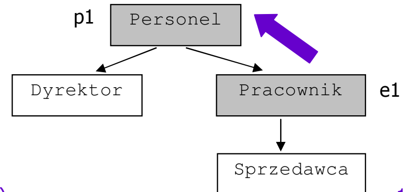
- dziedziczenie - wybrane zagadnienia pokrewne

- konwersja obiektów w hierarchii klas
  - pola poszczególnych klas:
    `Personel -  nazwisko, biuro`
    `Pracownik - nazwisko, biuro, stawka, godziny`
  - metody klasy `Pracownik` nie mają bezpośredniego dostępu do pól prywatnych, <u>które dziedziczą</u> z klasy `Personel`.
  - każdy obiekt typu `Pracownik` pola te jednak przechowuje w pamięci
  - musi więc istnieć sposób konwersji obiektu typu `Pracownik` na obiekt typu `Personel`
    (konwersja obiektu klasy pochodnej na obiekt klasy bazowej)

```
void main() {
    Personel  p1 ( "Malinowski" );
    Pracownik e1 ( "Kowalski", 2, 75.0, 169.0 );

    e1.print();
    p1 = e1;     // OK : Konwersja Pracownik -> Personel.
    p1.print(); // to samo osiągniemy gdy napiszemy: e1.Personel::print();
}
```

<u>Wynik działania programu:</u>

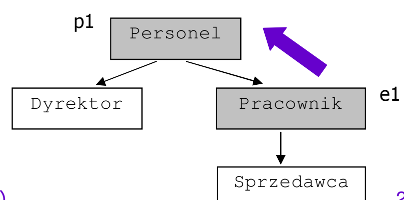
```
Pracownik Kowalski
Biuro = 2, tel. = 2222
Stawka godzinowa = 75, Liczba przepr. godzin = 169

Pracownik Kowalski
Biuro = 2, tel. = 2222
```



- konwersja obiektów w hierarchii klas
  - pola poszczególnych klas:
    `Personel -  nazwisko, biuro`
    `Pracownik - nazwisko, biuro, stawka, godziny`
  - konwersja polega na <u>skopiowaniu</u> do pól obiektu `p1` odpowiednich pól obiektu `e1`
  - wykonano więc <u>niejawną</u> konwersję obiektu klasy pochodnej na obiekt klasy bazowej
  - automatyczna (niejawna) konwersja <u>w przeciwna stronę</u> nie jest możliwa

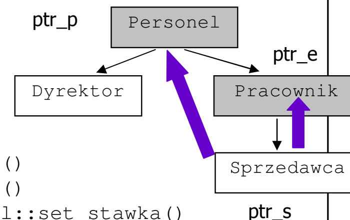
```
void main() {
    Pracownik e1( "Kowalski", 2, 75.0, 169.0 );
    Personel p1( "Malinowski" );

    e1.print();
    p1 = e1;     // OK : Konwersja Pracownik -> Personel.
    e1 = p1;     // BLAD : Konwersja Personel -> Pracownik jest niemożliwa

    p1.print();
}
```

Konwersja w kierunku
klasa bazowa --> klasa pochodna
nie pozwoliłaby poprawnie inicjować
dodatkowych pól definiowanych w klasie
pochodnej.



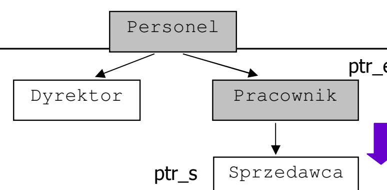
- konwersja wskaźników do obiektów w hierarchii klas
  - podobnie wygląda sytuacja przy konwersji wskaźników do obiektów
  - wskaźnik do obiektu klasy pochodnej podlega automatycznej konwersji na wskaźnik do obiektu klasy bazowej
  - ew. błędy wynikają NIE z niemożliwości wykonania konwersji, ale z efektów wykonania owej konwersji

```
void main() {
   Sprzedawca s( "Wincenty", 2, 75.0, 169.0, 0.05, 10000 );

   Sprzedawca* ptr_s = &s;
   Pracownik*  ptr_e = ptr_s;   // OK : konwersja (Sprzedawca*) -> (Pracownik*)
   Personel*   ptr_p = ptr_s;   // OK : konwersja (Sprzedawca*) -> (Personel*)


   ptr_s->set_biuro( 3 );   // OK : Personel::set_biuro()
   ptr_e->set_biuro( 4 );   // OK : Personel::set_biuro()
   ptr_p->set_biuro( 5 );   // OK : Personel::set_biuro()

   ptr_s->set_stawka( 85.0 );  // OK : Pracownik::set_stawka()
   ptr_e->set_stawka( 95.0 );  // OK : Pracownik::set_stawka()
   ptr_p->set_stawka( 98.0 );  // BLAD: nie istnieje Personel::set_stawka()

   ptr_s->set_sprzedaz( 15000 );  // OK : Sprzedawca::set_sprzedaz()
   ptr_e->set_sprzedaz( 20000 );  // BLAD: nie istnieje Pracownik::set_sprzedaz()
   ptr_p->set_sprzedaz( 30000 );  // BLAD: nie istnieje Personel::set_sprzedaz()   }
```



- konwersja wskaźników do obiektów w hierarchii klas
  - konwersja w drugą stronę (konwersja wskaźnika do obiektu klasy bazowej (nadrzędnej) na wskaźnik do obiektu klasy pochodnej) wymaga jawnego polecenia programisty (konwersja NIE jest automatyczna)

```
void main()
{
    Sprzedawca s( "Wincenty", 2, 75.0, 169.0, 0.05, 1000 );
    Pracownik* ptr_e = &s;     // OK : konwersja (Sprzedawca*) -> (Pracownik*)


    Pracownik p( "Jan", 3, 70.0, 169.0);
    Sprzedawca* ptr_s = &p;    // BLAD: nie jest wykonywana automatyczna konwersja
                                 (Pracownik*)->(Sprzedawca*)

    ptr_s -> sprzedaz = 2000; // BLAD: ptr_s wskazuje na p, ktory nie zawiera pola
                                 "sprzedaz"


    ptr_s = (Sprzedawca*) ptr_e;  // OK: jawna konwersja
    ptr_s -> sprzedaz = 2000;     // OK

}
```



- sekcja (kwalifikator) protected

  - do tej pory używane były tylko kwalifikatory `public` i `private`. W dziedziczeniu używany jest również kwalifikator `protected`
  - pola zawarte w sekcji `protected` klasy zachowują się jak pola prywatne. Jednak funkcje składowe klasy od niej pochodnej (**tylko funkcje !**) uzyskują dostęp do tych pól jakby były one publiczne
  - składowe chronione wykorzystywane są do nadawania przywilejów klasom pochodnym i jednoczesnym odebraniu ich innym funkcjom nie będącym ani składowymi ani funkcjami zaprzyjaźnionymi z dana klasą
  - co osiągamy ?
    kwalifikator `protected` pozwala, aby dana klasa posiadała inny interfejs wobec programu zewnętrznego, aniżeli w stosunku do klas od niej pochodnych.
  - kiedy używać `protected` ?
    Raczej w klasach stanowiących podstawę hierarchii klas pochodnych

- sekcja (kwalifikator) protected

  - wada 1: ochrona nie tak pewna jak z użyciem kwalifikatora `private`. Poziom zabezpieczenia pomiędzy `public` i `private`
  - wada 2 (chyba istotniejsza): po modyfikacji klasy bazowej, zmienić trzeba wszystkie klasy pochodne, które bezpośrednio korzystają z jej komponentów `protected`

- sekcja (kwalifikator) protected

```
class Bazowa {
  public:        void f();  int publ;
  protected:     int prot;
  private:       int priv;  };

void Bazowa::f() {
  publ = 1;     // OK : komponent publiczny
  prot = 2;     // OK : f() to funkcja skladowa klasy Bazowa
  priv = 3;     // OK : f() to funkcja skladowa klasy Bazowa    }
  ---------------------------------------------------------------------------
class Pochodna : public Bazowa {
  public:       void g(); };

void Pochodna::g() {
  publ = 1;     // OK :  komponent publiczny
  prot = 2;     // OK :  dozwolony dostep do komponentu protected
  priv = 3;     // BLAD: niedozwolony dostep do komponentu private. }
  ---------------------------------------------------------------------------
void main() {
  Bazowa b;

  b.publ = 1;  // OK :  komponent publiczny
  b.prot = 2;  // BLAD: niedozwolony dostep do komponentu protected poprzez obiekt typu Bazowa
  b.priv = 3;  // BLAD: niedozwolony dostep do komponentu private.

  Pochodna d;

  d.publ = 1;  // OK :  komponent publiczny
  d.prot = 2;  // BLAD: niedozwolony dostep do komponentu protected poprzez obiekt typu Pochodna
  d.priv = 3;  // BLAD: niedozwolony dostep do komponentu private.
}
```

OK, dostęp przez funkcję klasy bazowej

OK, dostęp przez funkcję klasy pochodnej

Błąd, dostęp bezpośredni, zamiast przez funkcję klasy bazowej lub pochodnej

- dziedziczenie, zagadnienia zaawansowane

- dziedziczenie prywatne (private)
  - do tej pory używając mechanizmu dziedziczenia umieszczaliśmy słowa `public` przed nazwa klasy bazowej
  - takie dziedziczenie jest najczęściej (**prawie zawsze !**) używane, ale ...
  - w dziedziczeniu `public` wszystkie komponenty `public` i `protected` klasy bazowej są dostępne dla jej klasy pochodnej, a komponenty `private` nie są
  - istnieją przypadki, gdy takie zachowanie (w sensie ochrony danych) jest nieodpowiednie (patrz ramki na następnym slajdzie)

```
class Bazowa {
  public:
    void f();
    int  a, b;
  private:
    float c, d;
 };

class Pochodna : public Bazowa  {
  public:
    void g();
};

void main() {
  Pochodna d;
  d.a = 1;         // dostęp do elementów publicznych klasy bazowej
  d.b = 2;
  d.f();
}
```

Bazowa → public → Pochodna

Komponenty publiczne klasy `Bazowa` pozostają publiczne także w klasie `Pochodna`

Program zewnętrzny może te komponenty bezpośrednio przetwarzać za pośrednictwem obiektu utworzonego w oparciu o klasę `Pochodna`

- dziedziczenie prywatne (private)
  - dziedziczenie `private` udostępnia dodatkowy stopień zabezpieczenia
  - dziedziczenie `private` pozwala precyzyjniej kontrolować dostępność komponentów klasy bazowej z poziomu klasy pochodnej
  - w dziedziczeniu `private` wszystkie komponenty klasy bazowej zachowują się tak jak komponenty `private` klasy pochodnej
  - komponenty klasy bazowej są więc dostępne wyłącznie za pomocą funkcji składowych klasy pochodnej (bezpośredni dostęp nie jest więc możliwy - i o to chodzi)

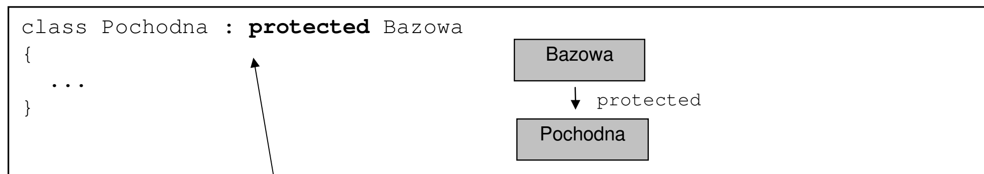
```
class Bazowa {
  public:
    void f1();
    void f2();
    int  a, b;
  private:
    float c, d;
 };

class Pochodna : private Bazowa  {
  public:
    Bazowa::f1();  // jawna autoryzacja dostępu do wybranych
    Bazowa::a;     // komponentów klasy Bazowa (robimy to w sekcji public !)
};

void main() {
  Pochodna d;
  d.a = 1;   // OK
  d.b = 2;   // BŁĄD
  d.f1();    // OK
  d.f2();    // BŁĄD
}
```

Bazowa → private → Pochodna

Brak bezpośredniego dostępu do komponentów klasy bazowej

Można ręcznie autoryzować dostęp do wybranych komponentów klasy bazowej

- dziedziczenie chronione (protected)
  - w dziedziczeniu `protected`, wszystkie komponenty `public` i `protected` klasy bazowej zachowują się jak komponenty `protected` klasy pochodnej (są niedostępne dla programu zewnętrznego)
  - klasy wyprowadzone od tej klasy pochodnej uzyskują normalny dostęp zarówno do jej danych jak i funkcji składowych

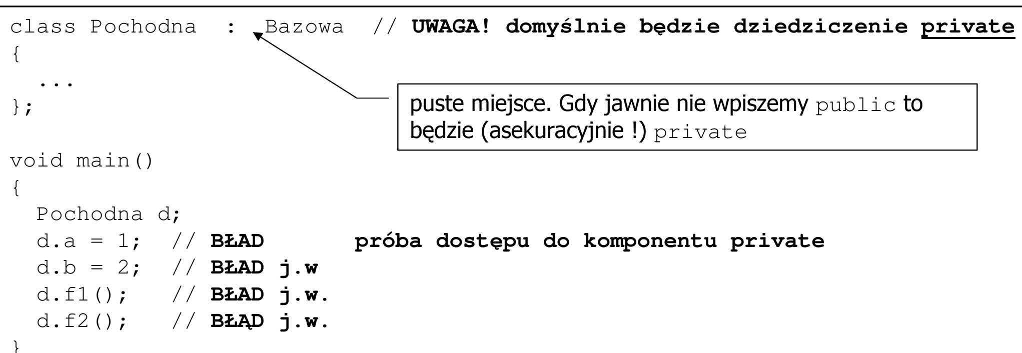


co uzyskujemy:
program zewnętrzny i/lub inne klasy wywiedziane od tej klasy pochodnej mają dostęp jedynie do jej interfejsu.
Czy jest to jednak często niezbędne ???

- dziedziczenie chronione (protected)
  - a tak dla dziedziczenia `private`

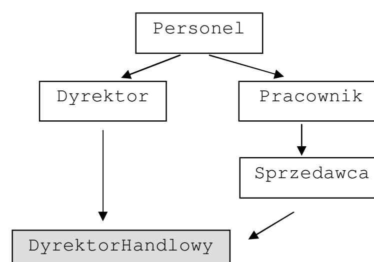
```
class Pochodna  : Bazowa  // UWAGA! domyślnie będzie dziedziczenie private
{
  ...
};
```

puste miejsce. Gdy jawnie nie wpiszemy `public` to będzie (asekuracyjnie !) `private`


```
void main()
{
  Pochodna d;
  d.a = 1;   // BŁAD      próba dostępu do komponentu private
  d.b = 2;   // BŁAD j.w
  d.f1();    // BŁAD j.w.
  d.f2();    // BŁĄD j.w.
}
```

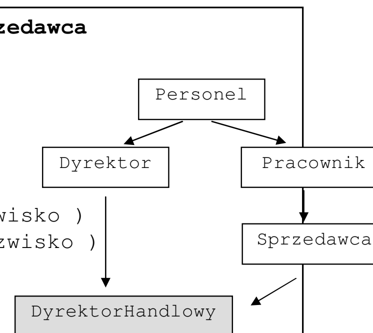
- sekcje i dziedziczenie `public, private, protected`
  uwagi na zakończenie
  - porównaj samodzielnie i znajdź ew. analogie pomiędzy omawianymi wcześniej dwoma zagadnieniami:
  - w deklaracji klasy - sekcje private, public, protected
  - w sposobie dziedziczenia - private, public, protected
  - jednolite nazewnictwo nie jest przypadkowe !

- dziedziczenie wielokrotne
  - czasami celowe jest utworzenie klasy dziedziczącej cechy kilku klas
  - dyrektor handlowy
    - zarządza pewna ilością pracowników (cecha z klasy `Dyrektor`)
    - otrzymuje pensje oraz premię uzależniona od ilości sprzedaży (cecha z klasy `Sprzedawca`)
  - każdy obiekt typu `DyrektorHandlowy` ma dostęp do komponentów (publicznych) klas `Dyrektor` i `Sprzedawca`



- dziedziczenie wielokrotne

```
class DyrektorHandlowy : public Dyrektor, public Sprzedawca
{
public:
    DyrektorHandlowy( const char* _nazwisko );
    // ... ewentualnie pozostale funkcje
};

DyrektorHandlowy::DyrektorHandlowy( const char* _nazwisko )
            : Dyrektor( _nazwisko ), Sprzedawca( _nazwisko )
{ }

void main()
{
    DyrektorHandlowy jk( "Jan Kowalski" );

    jk.set_pensja( 15000.0 );        // Dyrektor::set_pensja()
    jk.set_premia( 2000.0 );         // Dyrektor::set_premia()
    jk.set_ile_pracownikow( 5 );     // Dyrektor::set_ile_pracownikow()

    jk.set_procent( 0.001 );         // Sprzedawca::set_procent()
    jk.set_sprzedaz( 500000.0 );     // Sprzedawca::set_sprzedaz()
}
```



- dziedziczenie wielokrotne, problemy
  - dziedziczenie wielokrotne jest jednak podatne na błędy
  - klasa `Personel` jest <u>dwa razy</u> pośrednią klasa bazową dla klasy `DyrektorHandlowy`
  - obiekt typu `DyrektorHandlowy` posiada więc <u>dwie</u> kopie pól należących do `Personel`

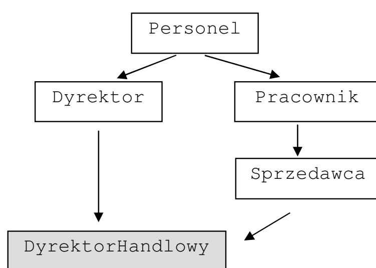
```
void main()
{
    DyrektorHandlowy jk( "Jan Kowalski" );

    jk.set_pensja( 15000.0 );          // Dyrektor::set_pensja()
    jk.set_premia( 2000.0 );           // Dyrektor::set_premia()
    jk.set_ile_pracownikow( 5 );       // Dyrektor::set_ile_pracownikow()

    jk.set_procent( 0.001 );           // Sprzedawca::set_procent()
    jk.set_sprzedaz( 500000.0 );       // Sprzedawca::set_sprzedaz()

    jk.set_biuro( 2 );                 // BLAD : instrukcja niejednoznaczna
    cout << jk.get_telefon() << endl;  // BLAD : instrukcja niejednoznaczna

    jk.Dyrektor::set_biuro( 2 );                  // OK.
    cout << jk.Dyrektor::get_telefon() << endl;   // OK.

    jk.print();     // BLAD: Sprzedawca::print() czy Dyrektor::print() ?
}
```

Użycie tu operatora zakresu, choć wyeliminuje błąd, to nie spowoduje wyświetlenia wszystkich informacji o obiekcie `jk`, a jedynie częściowe informacje w zależności od tego, czy wybrano `Sprzedawca::print()` czy `Dyrektor::print()`

- dziedziczenie wielokrotne, problemy
  - problemy przy konwersji
  - w przykładzie poniżej dokonujemy konwersji klasy pochodnej w stosunku do klasy bazowej

```
Personel* pp = &jk;   // BLAD : instrukcja niejednoznaczna

// wskazujemy żądaną gałąź dziedziczenia
Personel* pp = (Dyrektor*) &jk  ;
```

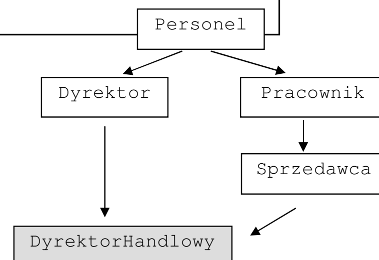
Na którą z kopii pól klasy Personel ma wskazywać wskaźnik pp ?



- dziedziczenie wielokrotne - wirtualna klasa bazowa
  - każdy obiekt typu `DyrektorHandlowy` posiada dwa zestawy pól dziedziczonych z klasy `Personel` (dwa nazwiska, dwa biura). Nie jest to oczywiście zgodne z rzeczywistością.
  - użycie słowa `virtual` przy tworzeniu klas pochodnych bezpośrednio od klasy `Personel` usuwa tą niedogodność
  - słowo `virtual` nie pozwala tu na duplikowanie pól klasy, w stosunku do której go użyto (tu w stosunku do klasy `Personel`)
  - instrukcje niejednoznaczne pokazane wcześniej są teraz OK

```
class Pracownik : public virtual Personel
{
... };

class Dyrektor : public virtual Personel
{
... };
```

`virtual` użyte tylko w stosunku do pierwszych klas wyprowadzonych z klasy `Personel`



- dziedziczenie wielokrotne - wirtualna klasa bazowa
  - przy dziedziczeniu wirtualnym, każda z tworzonych klas powinna jawnie wywołać konstruktor swojej wirtualnej klasy bazowej (nawet gdy nie jest jej bezpośrednim potomkiem)

```
class Pracownik : public virtual Personel
{ ... };

class Dyrektor : public virtual Personel
{ ... };

class DyrektorHandlowy : public Dyrektor, public Sprzedawca
{
public:
    DyrektorHandlowy( const char* _nazwisko );
    void print() const
      {
       Sprzedawca::print();
       Dyrektor::print();
      }
    ...
};

DyrektorHandlowy::DyrektorHandlowy( const char* _nazwisko )
            : Dyrektor( _nazwisko ), Sprzedawca( _nazwisko ),
              Personel( _nazwisko )
{}
```


Wywołanie `DyrektorHandlowy::print()` spowoduje tu dwukrotne wyświetlenie informacji pochodzących z `Personel::print()`.
Aby to zmienić, należałoby zmodyfikować wszystkie funkcje `print()` w całej hierarchii klas !!!

tego nie było poprzednio

- dziedziczenie wielokrotne, uwagi
  - dziedziczenie wielokrotne jest podatne na błędy
  - gdy można, lepiej go unikać (w wielu przypadkach użycie dziedziczenia wielokrotnego jest nieuzasadnione)
  - implementując klasę `DyrektorHandlowy`, można jej opis uzupełnić (zamiast dziedziczyć) o fragmenty kodu implementującego założenie o dodatku do pensji w postaci premii uzależnionej od ilości sprzedaży