(Średnio)zaawansowane programowanie w C++

Wykład 5 - polimorfizm, funkcje wirtualne. Powtórzenie

Robert Nowak

23Z

Plan wykładu

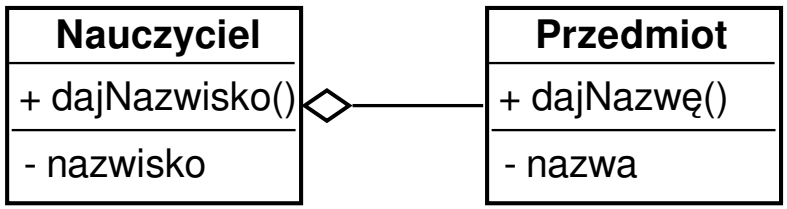
- Dzieczienie, funkcje wirtualne
- Dynamiczna informacja o typie
- Bibioteki, wtyczki
- Powtórzenie

Programowanie przyrostowe. Tworzenie nowych klas

Budowa klas na podstawie już istniejących, ponowne wykorzystanie kodu:

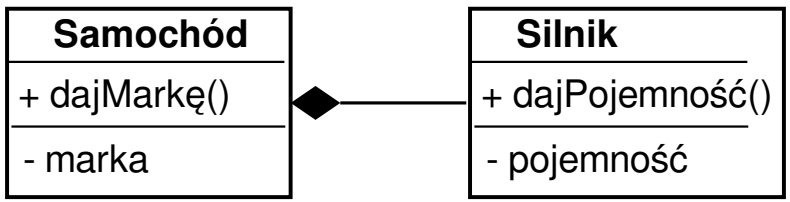
- agregacja,
- dziedziczenie.

Agregacja („posiada”) - gdy budowa z mniejszych kawałków większej całości.



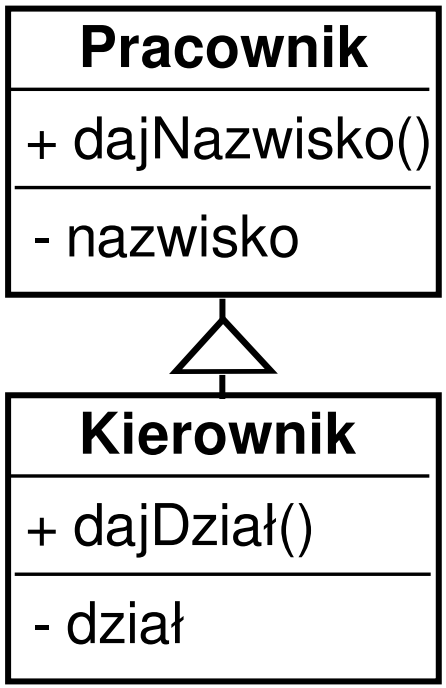

```
class Przedmiot { /* ...  */ };
class Nauczyciel {
private:
  std::vector<Przedmiot*> przedm_;
};
```

Kompozycja - agregacja, obiekt składowy nie istnieje bez obiektu głównego

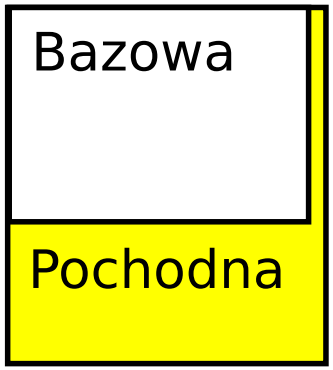



```
class Silnik { /* ...  */ };
class Samochod {
private:
  Silnik silnik_;
};
```

Dziedziczenie - relacja „może być traktowany jako”




```
//klasa bazowa
class Pracownik {
  //...
};
//klasas pochodna
class Kierownik : public Pracownik {
  //...
};
```



Dziedziczenie wprowadza powiązanie pomiędzy typami.

Wycinanie

Działanie konstruktora kopiującego lub operatora przypisania:

```
class Pracownik { ... };
class Kierownik : public Pracownik { ... };

Kierownik k(...);
Pracownik p = k;
```

- kopiuje tylko część klasy,
- źródło niespodzianek i błędów,
- rozwiązanie: przekazywanie wskaźników lub referencji do obiektów.

Problem typu dla obiektów

```
class Osoba {
public:
  void drukuj(ostream& os) const { os << "osoba"; }
};
class Pracownik : public Osoba {
public://Przedefiniowanie metody (redefining)
  void drukuj(ostream& os) const { os << "pracownik"; }
};
void drukujOsobe(const Osoba& f) {
  f.drukuj(cout);
}
Osoba o;
Pracownik p;
p.drukuj( cout );//woła metodę Pracownik::drukuj
drukujOsobe(o);//wołana metoda Osoba::drukuj
drukujOsobe(p);//błąd!  wołana metoda Osoba::drukuj
```

Polimorfizm za pomocą pola typu (bardzo złe rozwiązanie)

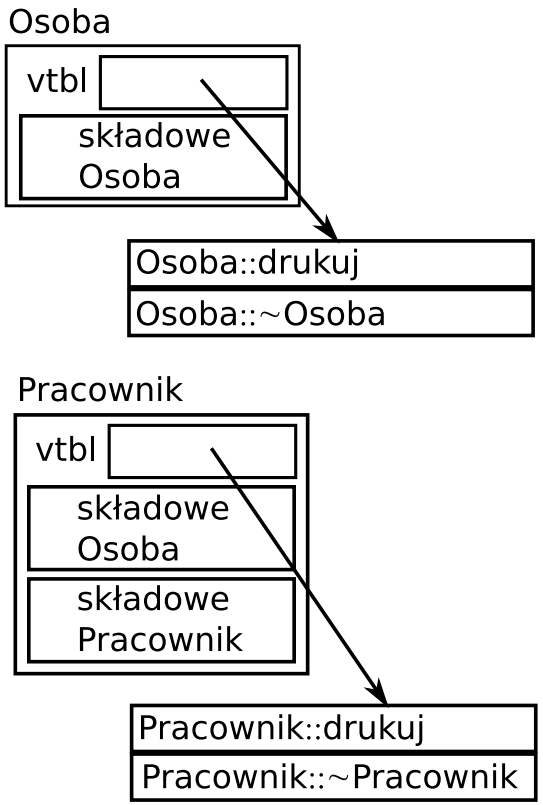
```
class Osoba {
public:
  enum Typ { OSOBA, PRACOWNIK, KLIENT };
  const Typ typ_;//Pole pamięta typ obiektu
  Osoba(Typ t = OSOBA) : typ_(t) {}
};
class Pracownik : public Osoba {
public:
  Pracownik() : Osoba(PRACOWNIK) {}
};
void drukuj(const Osoba& o) {
  switch(o.typ)
  //...
}
```

- kompilator nie potrafi sprawdzić poprawności
- kod staje się nieczytelny i trudno modyfikowalny

Polimorfizm z wykorzystaniem funkcji wirtualnych

wspierane przez kompilator: późne wiązanie

```
class Osoba {
  virtual void drukuj(ostream& os) const
  { os << "osoba"; };
  virtual ~Osoba() { /* ...  */ }
};
class Pracownik : public Osoba {
public://Nadpisywanie metody (overriding)
  virtual void drukuj(ostream& os) const
  { os << "pracownik"; }
  virtual ~Pracownik() { /* ...  */ }
};
```



narzuty pamięciowe:

- wskaźnik w obiekcie (`vtbl`),
- tablica wskaźników dla klas dostarczających funkcji wirtualnych

narzuty czasowe: dodatkowe adresowanie pośrednie

Typy klas

Klasa wartość (klasa autonomiczna):

- brak metod wirtualnych
- konstruktor, konstruktor kopiujący, operator przypisania, destruktor
- najczęściej obiekt automatyczny lub składowa klasy
- może być przekazywana przez wartość

Klasa bazowa dla hierarchii klas:

- używa metod wirtualnych, powinna mieć wirtualny destruktor
- najlepiej gdy abstrakcyjna albo prywatny konstruktor kopiujący i prywatny operator przypisania (zapobiega wycinaniu)
- najczęściej obiekt na stercie
- przekazywana przez wskaźnik lub referencję

Metody - możliwości modyfikacji w klasach pochodnych

```
class Base {
public:
  //metoda
  void a() { /* ...  */ }
  //metoda wirtualna
  virtual void b() { /* ...  */ }
  //metoda czysto wirtualna
  virtual void c() = 0;
};
```

- metoda - kod nie może być zmieniany w klasach pochodnych
- metoda wirtualna - kod może (ale nie musi) być dostarczony w klasach pochodnych, domyślna implementacja w klasie bazowej
- metoda czysto wirtualna - kod musi być dostarczony w klasach pochodnych

Interfejs bez funkcji wirtualnych (NVI idiom)

Publiczna funkcja wirtualna { część interfejsu (metoda publiczna); część implementacji (nadpisywanie) }

```
//NVI - demonstracja
class Better {
public:
  int calculate() const {
    //czynności przed
    int ret = doCalculate();
    //czynności po
    return ret;
  }
private:
  //Prywatna metoda może być nadpisywana!
  virtual int doCalculate() const;
};
```

Wyjątki: destruktory

Alternatywa dla funkcji wirtualnych - wskaźnik do funkcji

```
//Przykładowa funkcja
int defaultCalcFun(const Calc& c){ /* ...  */ }
class Calc {
public:
  using CalcFun = int (*)(const Calc&); //Sygnatura (typ) funkcji

  Calc(CalcFun cf = defaultCalcFun ) : calcFun_(cf) {}
  int calculate() const { return calcFun_(*this); } //Funkcja polimorficzna
private:
  CalcFun calcFun_; //Wskaźnik do funkcji
};
```

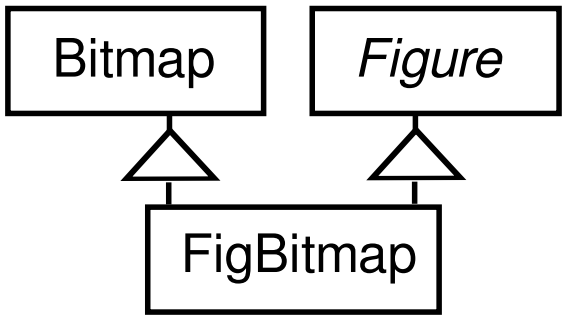
Polimorfizm za pomocą uchwytu (wskaźnika):

- zachowanie może być różne nawet dla obiektów tej samej klasy,
- zachowanie może się zmienić w czasie działania,
- funkcja nie jest metodą, ma dostęp tylko do interfejsu klasy.

Dziedziczenie wielobazowe (multiple inheritance, MI)

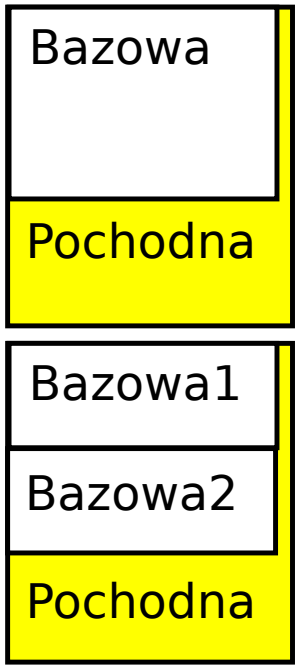
Wprowadzone, ponieważ może istnieć wiele niezależnych hierarchii klas.

W C++ nie ma klasy bazowej (o nazwie np. `Object`) dla wszystkich innych klas.




```
class FigBitmap
: public Figure, public Bitmap
{ /* ...  */ };
```

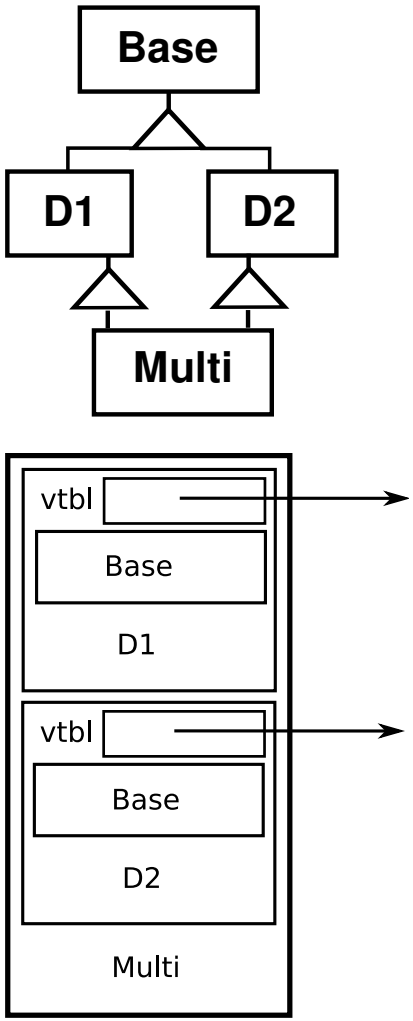
Obiekty przy wielo-dziedziczeniu




```
class Bazowa { /*   */ };
class Pochodna : public Bazowa { /*   */ };
class B1 {
    public: void f();
};
class B2 {
    public: void f();
};
class P2 : public B1, public B2 { /*   */ };
```

```
P2 d; // Rzutowanie w górę może zmienić adres
B1* pb1 = &d;//pb1 to ten sam adres co &d
B2* pb2 = &d;//pb2 jest innym adresem niż pb1
//Wołanie metody może być niejednoznaczne
d.f(); //Błąd kompilacji
static_cast<B1&>(d).f(); //OK, woła B1::f()
```

Wielokrotne wystąpienie klasy podstawowej



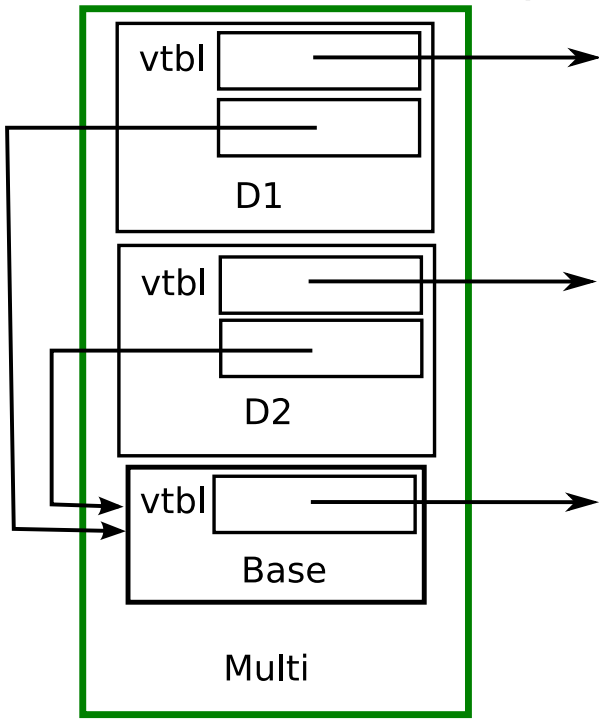

```
class Base { /* ...  */ };

class D1 : public Base { /* ...  */ };

class D2 : public Base { /* ...  */ };

class Multi : public D1, public D2
{ /* ...  */ };
```

Obiekty Klasy `Multi` zawierają dwa obiekty klasy `Base`

- niejednoznaczność
- nie zawsze jest to pożądane (zasoby)

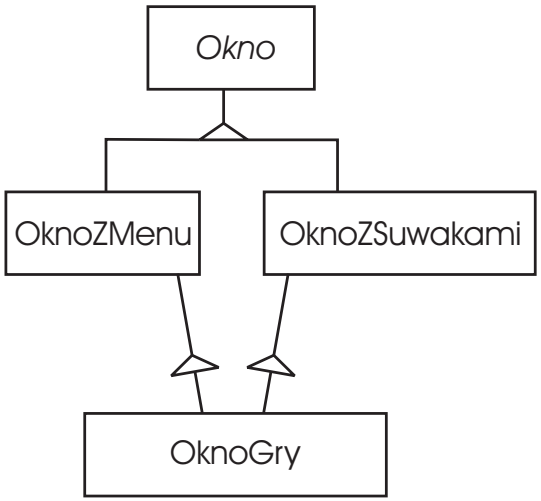
Wirtualne klasy bazowe

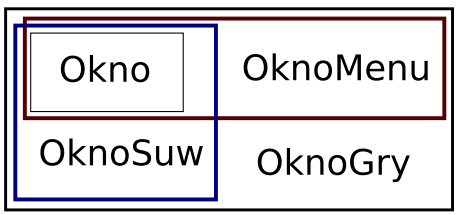
Zapewnia, że tylko jedna instancja obiektu klasy bazowej będzie umieszczana w klasie pochodnej




```
class Base { /* ...  */ };

class D1 : virtual public Base
{ /* ...  */ };
class D2 : virtual public Base
{ /* ...  */ };
class Multi : public D1, public D2
{ /* ...  */ };
```

Dziedziczenie wirtualne (przykład)






```
class Okno { /* ...  */ };

class OknoZMenu : virtual public Okno
{ /* ...  */ };

class OknoZSuw : virtual public Okno
{ /* ...  */ };

class OknoGry
: public OknoZMenu, public OknoZSuw
{ /* ...  */ };
```

Metody wirtualne a konstruktory

Mechanizm funkcji wirtualnych nie działa w konstruktorach i destruktorach.

```
class Base {
public:
  Base(){ f(); }//Wołana Base::f()
  virtual void f(){ /* ...  */ }
};
class Derived : public Base {
public:
  Derived() : Base() { }
  virtual void f(){ /* ...  */ }
};
Derived d; //Wołana Base::f() w konstruktorze
Base* b = new Derived; //Wołana Base::f() w konstruktorze
b->f(); //Wołana Derived::f()
```

Dziedziczenie prywatne i chronione

```
class D: public B { ... };//Dziedziczenie publiczne
class D: protected B { ... };//Dziedziczenie chronione
class D: private B { ... };//Dziedziczenie prywatne
```

Ochrona dla różnych rodzajów dziedziczenia:

rodzaj	Dostęp do		
	konwersji $D^* \rightarrow B^*$	składowych publicznych B	składowych chronionych B
publiczne	wszystkie funkcje		metody D oraz klas pochodnych po D
chronione	metody D oraz klas pochodnych po D		
prywatne	metody D		

dynamiczna informacja o typie (RTTI)

typeid - operator, zwraca informację o typie

```
const type_info& typeid(nazwa_typu) throw();
const type_info& typeid(wyrazenie) throw(bad_typeid);
```

Rzutowanie klasy bazowej na pochodną (w dół, downcasting)

```
class B1 { /* ...  */ };
class B2 { /* ...  */ };
class P : public B1, public B2 { /* ...  */ };

B1* b = new P;
//wskaźniki:  dla odp. typu zwraca wskaźnik, inaczej nullptr
P* p = dynamic_cast<P*>(b);
//referencje:  dla błędnego typu wyjątek 'bad_cast'
P& r = dynamic_cast<P&>(b);
//Rzutowanie skrośne (do klasy siostrzanej, crosscasting)
B2* pb2 = dynamic_cast<B2*>(b);
```

RTTI - koszty

- wielkość kodu
  - struktura `type_info` dla każdej klasy
  - wskaźnik do `type_info` w tablicy funkcji wirtualnych
  - lista referencji do struktur `type_info` do klas bazowych (dla klas, które dziedziczą)
- czas wykonania
  - wołanie `typeid` - uzyskanie wskaźnika z tablicy funkcji wirtualnych
  - wołanie `dynamic_cast` rekurencyjne przeszukanie referencji do `type_info` do klas bazowych (przechowywane w strukturze `type_info`)
  - badanie typu w `catch` - jak `dynamic_cast`

Translacja w C++

- obróbka wstępna (preprocessing)
- kompilacja
- konsolidacja

Preprocesor (unikać wykorzystania preprocesora)

Preprocesora należy używać jedynie do:

- dołączania plików nagłówkowych (`#include`)
- zabezp. nagłówków przed wielokrotnym dołączaniem
- kompilacji warunkowej

Makrodefinicje (**#define**)

- zamiast prostych stosować stałe
- zamiast złożonych szablony funkcji

```
const int MAX = 20; //stała globalna
class Foo {
  static const int SIZE = 30; //stała globalna, dostępna dla metod
};
template<typename T>
inline void max(const T& a, const T& b) {
  if(a > b) return a;
  else return b;
}
```

Biblioteki

- biblioteki dostępne w formie źródeł
  - wykorzystują szablony
  - brak problemów z łączeniem (konsolidacją)
- biblioteki binarne
  - muszą być zgodne z aplikacją
  - kod może być ukryty

Biblioteki binarne:

- statyczne (Windows: .lib, Linux: .a): - skompresowane wyniki kompilacji + tablica symboli
- dynamiczne(Windows: .dll, Linux: .so): - kod PIC (ang. *position independent code*) + tablica symboli
  - ładowane podczas startu aplikacji
  - ładowane podczas pracy aplikacji (np. wtyczki)

Biblioteki ładowane dynamicznie

- możliwe wykorzystywanie typów, które nie są dostępne w czasie kompilacji
- interfejs musi być znany w czasie kompilacji

Biblioteka upraszczająca wykorzystanie wtyczek zawiera:

- zarządca wtyczek (rejestracja, inicjacja, finalizacja)
- wtyczka
- klasy pomocnicze (np. klasa reprezentująca bibliotekę dynamiczną)

Przykłady: QT (plugin), Boost.DLL, Boost.Extensions, Plugg, Gigi Sayfan framework (Dr Dobb's Journal).

Przykład - zarządca wtyczek

```
class PluginManager { //fabryka obiektów
public:
  static PluginManager& getInstance(); //singleton
  bool loadAll(const std::string& directory); //ładuje i inicjuje
  IPlugin* getObject(const std::string& id); //dostarcza uchwyt
};
```

Zadania:

- przechowuje uchwyty do dostępnych wtyczek
- pozwala ładować/zwalniać wtyczki
- w zależności od platformy wykorzystuje różne funkcje do ładowania bibliotek dynamicznych

Reprezentacja wtyczki - bez użycia refleksji

```
class Selectable {
public:
  virtual bool isSelectable() = 0; //czy obiekt może być wyróżniany
  virtual void select() = 0; //wyróżnia obiekt
  virtual void unselect() = 0; //kasuje wyróżnienie obiektu
};
```

Wada: typy, które nie dostarczają interfejsu muszą go implementować

```
class Background : public Selectable {//typ bez możliwości wyróżnienia
public:
  virtual bool isSelectable() { return false; } //nie można wybierać
  virtual void select() {} //wymagana pusta implementacja
  virtual void unselect() {} //wymagana pusta implementacja
};
```

Reprezentacja wtyczki - wykorzystanie refleksji

Mechanizmy refleksji : rzutowanie dynamiczne

- umożliwiają badanie obecności interfejsu
- upraszczają obsługę typów dostarczanych przez biblioteki dynamiczne

```
class Selectable {//interfejs dla typów dających możliwość wyróżniania
  public:
  virtual void select() = 0;
  virtual void unselect() = 0;
};
class Button : public Selectable { //implementuje interfejs
  public:
  virtual void select() { /* wyróżnienie obiektu */ }
  virtual void unselect() { /* kasowanie wyróżnienia */ }
};
class Background { //nie implementuje interfejsu
  //...
};
```

Wykorzystywanie rzutowania dynamicznego do badania interfejsu

```
Object* obj = /* inicjacja obiektu */ ;
Selectable* sel = dynamic_cast<Selectable*>(obj);
if(sel) //badanie, czy obiekt dostarcza interfejsu
    sel->select(); //wyróżnienie obiektu
```

- klasy, które nie dostarczają interfejsu nie muszą go implementować
- badanie następuje za pomocą mechanizmów języka
- interfejs musi być znany
- typy mogą być implementowane wewnątrz bibliotek dynamicznych

Powtórzenie

Powtórzenie

- polimorfizm, funkcje wirtualne, dziedziczenie wielobazowe
- dynamiczna informacja o typie
- różne strategie obsługi błędów, mechanizm wyjątków
- sprytne wskaźniki:
  - std::unique_ptr,
  - std::shared_ptr,
  - std::weak_ptr,
  - boost::intrusive_ptr
- referencja do r-wartości w C++11 (r-value reference)
- Rust: podstawy, zarządzanie zasobami

Przykładowe zadania

Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {
  for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )
    if( *i == name )
       return (*i).addr;
  return "";
}
```

Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {
  for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )
    if( *i == name )
       return (*i).addr;
  return "";
}
 string FindAddr( const list<Employee>& l, const string& name ) {
   string addr;
   for( list<Employee>::iterator i = l.begin(); i != l.end(); ++i )
     if( i->name() == name ){
        addr = (*i).addr; break; }
   return addr;
}
```

Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {
  for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )
    if( *i == name )
       return (*i).addr;
  return "";
}
 string FindAddr( const list<Employee>& l, const string& name ) {
   string addr;
   for( list<Employee>::iterator i = l.begin(); i != l.end(); ++i )
     if( i->name() == name ){
        addr = (*i).addr; break; }
   return addr;
}
  list<Employee>::const_iterator it =
find_if(l.begin(), l.end(), bind(&Employee::name,_1) == name);
```

Zadanie (Guru of The Week, 2), niepotrzebne obiekty

```
string FindAddr( list<Employee> l, string name ) {
  for( list<Employee>::iterator i = l.begin(); i != l.end(); i++ )
    if( *i == name )
       return (*i).addr;
  return "";
}
 string FindAddr( const list<Employee>& l, const string& name ) {
   string addr;
   for( list<Employee>::iterator i = l.begin(); i != l.end(); ++i )
     if( i->name() == name ){
        addr = (*i).addr; break; }
   return addr;
}
  list<Employee>::const_iterator it =
find_if(l.begin(), l.end(), bind(&Employee::name,_1) == name);
  auto it =
find_if(l.begin(), l.end(), [&](Employee& e){ return e.name == name;});
```

Zadanie, brak zwalniania obiektów

```
class List {
public:
  using PNode = shared_ptr<Node>;
  struct Node {
    Node(const string& s):s_(s){}
    string s_;
    PNode prev_;
    PNode next_;
  };
  void push_back(const string& s);
private:
  PNode head_;
};
```

```
void List::push_back(const string& s)
  if( head_ ) { //not empty
    PNode tail( new Node(s) );
    tail->prev_ = head_->prev_;
    tail->next_ = head_;
    head_->prev_->next_ = tail;
    head_->prev_ = tail;
  }
  else { //empty
    head_ = PNode(new Node(s));
    head_->prev_ = head_;
    head_->next_ = head_;
  }
}
```

Zadanie, brak zwalniania obiektów - rozwiązanie

```
class List {
public:
  using PNode = shared_ptr<Node>;
  using PWNode = weak_ptr<Node>;
  struct Node {
    Node(const string& s):s_(s){}
    string s_;
    PWNode prev_; //zmiana
    PNode next_;
  };
  void ~List() { //zmiana
    //usuniecie zal.  cyklicznej
    if( head_ )
      head_->prev_->next_.reset();
  }
  void push_back(const string& s);
private:
  PNode head_;
};
```

```
void List::push_back(const string& s)
  if( head_ ) { //not empty
    PNode tail( new Node(s) );
    tail->prev_ = head_->prev_;
    tail->next_ = head_;
    head_->prev_.lock()->next_ = tail;
    head_->prev_ = tail;
  }
  else { //empty
    head_ = PNode(new Node(s));
    head_->prev_ = head_;
    head_->next_ = head_;
  }
}
```

Dziękuję