

# Metody Realizacji Języków Programowania

Realizacja funkcji, procedur i metod

Marcin Benke

MIM UW

14–21 listopada 2016

## Podprogramy

W wielu językach programowania podstawowy mechanizm abstrakcji stanowią podprogramy (funkcje, procedury, metody).

Wywołanie podprogramu wymaga kilku kroków:

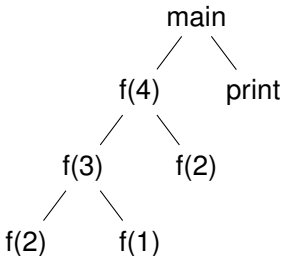
- Wyznaczenie (adresu) podprogramu (ten krok jest często trywialny, acz nie zawsze, np.  $s[i].f(x)$  w C, a tym bardziej dla metod wirtualnych w C++).
- Zapewnienie wyrównania stosu
- Przygotowanie argumentów
- Skok ze śladem
- Po powrocie ewentualne uprzątnięcie argumentów
- Odczytanie wyniku

# Przygotowanie argumentów

- dla argumentów przekazywanych przez wartość, obliczenie wartości
- dla argumentów przekazywanych przez zmienną, obliczenie adresu
- dla argumentów przekazywanych przez nazwę/potrzebę, przygotowanie domknięcia (thunk)
- Umieszczenie argumentów w przewidzianym protokołem miejscu (w rejestrach, na stosie, etc.)

## Drzewo aktywacji

Wykonanie programu można przedstawić jako *drzewo aktywacji*. Korzeniem jest program główny a węzeł  $F$  ma synów  $G_1 \dots G_n$  jeśli wcielenie funkcji  $F$  wywołało  $G_1$ , później  $G_2$  itd.



Funkcja jest rekurencyjna, jeśli istnieje ścieżka, na której występuje wiele jej wcieleń.

# Rekord aktywacji

- Z każdym wcieleniem funkcji wiążemy pewne informacje. Obszar pamięci, w którym są zapisywane, nazywamy *rekordem aktywacji* albo *ramką* (ang. *frame*).
- W większości języków potrzebne są tylko rekordy dla aktywnych wcieleń na aktualnej ścieżce w drzewie aktywacji.
- Gdyby nie rekurencja, dla każdej funkcji moglibyśmy z góry zarezerwować obszar pamięci na jedną ramkę (wczesny Fortran).
- W językach z rekurencją rekordy aktywacji alokujemy przy wywołaniu funkcji a zwalniamy, gdy funkcja się skończy.
- Rekordy aktywacji przechowujemy więc na stosie.

## Zawartość rekordu aktywacji

Informacje pamiętane w rekordzie aktywacji zależą m. in. od języka. Mogą tam być:

- parametry
- zmienne lokalne i zmienne tymczasowe
- ślad powrotu
- kopia rejestrów (wszystkich, niektórych lub żadnego)
- łączy dynamiczne (DL, ang. dynamic link) – wskaźnik na poprzedni rekord aktywacji; ciąg rekordów połączonych wskaźnikami DL tworzy tzw. łańcuch dynamiczny.
- łączy statyczne (SL, ang. static link)
- miejsce na wynik

Postać rekordu aktywacji nie jest sztywno określona — projektuje ją autor implementacji języka.

## Adresowanie rekordu aktywacji

- Adres ramki jest zwykle przechowywany w wybranym rejestrze (FP = frame pointer, BP = base pointer).
- Pola rekordu aktywacji są adresowane przez określenie ich przesunięcia względem adresu w FP.
- Każde wcielenie funkcji, niezależnie od położenia rekordu aktywacji, w ten sam sposób może korzystać z jego zawartości, a więc wszystkie wcielenia mają wspólny kod.
- Adresem rekordu aktywacji nie musi być adres jego początku. Czasem wygodniej przyjąć adres jednego z pól w środku tego rekordu.

## Adresowanie rekordu aktywacji

- Jeśli w języku są funkcje ze zmienną liczbą argumentów (jak np. w C), lepiej adresować względem środka ramki.
- Przy adresowaniu względem początku, przesunięcia zależą od liczby parametrów, a więc nie są znane podczas kompilacji.
- Rozwiązanie: adresowanie ramki względem miejsca pomiędzy parametrami a zmiennymi lokalnymi funkcji.
- Parametry zapisujemy od ostatniego do pierwszego, dzięki czemu przesunięcie K-tego parametru nie zależy od liczby parametrów, tylko od stałej K.
- Wynik funkcji często w rejestrach zamiast na stosie.
- Dla architektur z dużą liczbą rejestrów (np x86\_64) niektóre argumenty też w rejestrach.



# Protokół wywołania (i powrotu z) funkcji

- Protokół wywołania opisuje czynności, które ma wykonać wołający (zarówno przed przekazaniem sterowania do wołanego, jak i po powrocie) oraz to, co wołany (czyli każda funkcja) ma robić na początku i na końcu.
- Podstawowym zadaniem jest zbudowanie rekordu aktywacji oraz usunięcie go.
- Niektóre czynności musi wykonywać wołający (np. liczenie parametrów), inne wołany (np. zarezerwowanie miejsca na zmienne lokalne), a jeszcze inne może wykonać zarówno wołany jak i wołający.

# Rejestry

Protokół może określać, które rejestry “należą” do wołającego, a które do wołanego.

Rejestry, które zachowują swoją wartość po wywołaniu funkcji — ich zachowanie jest obowiązkiem wołanego (callee-save)

dla x86/libc: EBP, EBX, ESI, EDI

dla x86\_64: RBP, RBX, R12–R15

Rejestry, które nie zachowują wartości po wywołaniu funkcji (caller-save)

## Protokół wywołania (i powrotu z) funkcji

Zaprojektujemy protokół wywołania i powrotu z funkcji. Założymy przy tym następującą postać rekordu aktywacji:

- miejsce na wynik
- parametry
- ślad
- DL
- zmienne

Będziemy używać abstrakcyjnego procesora, w którym wskaźnik/int zajmuje jedno słowo, z rejestrami

- SP — wskaźnik stosu
- BP — wskaźnik ramki
- A,B,C,D — rejestry uniwersalne
- S — link statyczny (o czym za chwilę)

Notacja [*adres*] oznacza odwołanie do pamięci

# Protokół wywołania (i powrotu z) funkcji

Wskaźnikiem rekordu aktywacji będzie BP zawierający adres DL.

wołający

```
PUSH 0                # miejsce na wynik
<parametry na stos>
CALL adres_wołanego
SP += n               # n - łączny rozmiar parametrów
                     # wynik zostaje na stosie
```

wołany

```
PUSH BP              # DL na stos
BP = SP              # aktualizacja wskaźnika ramki
SP -= k              # k - łączny rozmiar zmiennych
...                  # tłumaczenie treści funkcji
SP = BP              # przywracamy wskaźnik stosu
POP BP               # przywracamy wskaźnik ramki
RET                  # powrót do wołającego
```

# Mechanizm przekazywania parametrów

Przekazywanie parametru przez wartość:

- wołający umieszcza w rekordzie aktywacji wartość argumentu;
- wołany może odczytać otrzymaną wartość, ew. zmieniać ją traktując parametr tak samo, jak zmienną lokalną;
- ewentualne zmiany nie są widziane przez wołającego.

Jesli argumenty są przekazywane w rejestrach, wołany musi zwykle zapisać je w swojej części rekordu aktywacji.

## Przekazywanie parametru przez zmienną

- wołający umieszcza w rekordzie aktywacji adres zmiennej;
- wołany może odczytać wartość argumentu sięgając pod ten adres,
- może też pod ten adres coś wpisać, zmieniając tym samym wartość zmiennej będącej argumentem.

Przekazywanie parametru przez wartość-wynik:

- wołający przekazuje wartość, a po powrocie odczytuje wynik.

## Przykład 1

W programie, którego fragment wygląda następująco:

```
void p() {  
    int a,b;  
    a = q(b,b);  
}  
  
int q(int x, int& y) {  
    int z;  
    result = x + y;  
    y = 7;  
}
```

Przyjmujemy, że przypisanie na `result` ustawia wynik funkcji.

rekord aktywacji procedury p wyglądałby tak (w nawiasach podano przesunięcia poszczególnych pól względem pola DL wskazywanego przez rejestr BP):

( +1 )	ślad
( 0 )	DL
( -1 )	a
( -2 )	b

a rekord aktywacji funkcji q tak:

( +4 )	miejsce na wynik
( +3 )	x
( +2 )	adres y
( +1 )	ślad
( 0 )	DL
( -1 )	z



**kod wynikowy dla procedury p miałby postać:**

```
# void p() { int a,b; a = q(b,b); }
```

PUSH BP	# DL do ramki
BP = SP	# nowy adres ramki do BP
SP -= 2	# miejsce na dwie zmienne
PUSH 0	# miejsce na wynik
PUSH [BP-2]	# wartość b
A = BP-2	# adres zmiennej b do A (LEA)
PUSH A	
CALL q	# skok ze śladem do q
SP += 2	# usuwamy parametry
POP [BP-1]	# przypisujemy wynik na a
SP = BP	# usuwamy zmienne
POP BP	# wracamy do poprzedniej ramki
RET	# i do miejsca wywołania

a kod funkcji q:

```
# int q(int x, int&y) { result = x+y; y = 7; }
```

PUSH BP	# DL do ramki
BP = SP	# nowy adres ramki do BP
SP -= 1	# miejsce na jedną zmienną
A = [BP+3]	# wartość x
B = [BP+2]	# adres y do B
A += [B]	# dodajemy x i y
[BP+4] = A	# zapamiętujemy wynik
A = [BP+2]	# adres zmiennej y
[A] = 7	# przypisanie na y
SP = BP	# usuwamy zmienną
POP BP	# wracamy do poprzedniej ramki
RET	# i do miejsca wywołania

# Środowisko w językach ze strukturą blokową

- Wiele języków programowania (n.p. Pascal) pozwala na zagnieżdżanie funkcji i procedur. Języki te nazywamy językami ze *strukturą blokową*.
- Kod funkcji ma dostęp nie tylko do jej danych lokalnych, ale także do danych funkcji, w której jest zagnieżdżona itd. aż do poziomu globalnego.
- Działanie funkcji jest określone nie tylko przez jej kod oraz parametry, lecz także przez środowisko, w którym ma się wykonać.

## Wiązanie statyczne i dynamiczne

- Postać środowiska jest w Pascalu wyznaczona statycznie — z kodu programu wynika, do której funkcji należy rekord aktywacji, w którym mamy szukać zmiennej nielokalnej.
- Mówimy, że w Pascalu obowiązuje statyczne wiązanie zmiennych.
- Istnieją również języki (n.p. Lisp), w których obowiązuje wiązanie dynamiczne — w przypadku odwołania do danej nielokalnej, szukamy jej w rekordzie aktywacji wołającego itd. w górę po łańcuchu dynamicznym.

## Łącze statyczne

- By korzystać z danych nielokalnych, działająca funkcja musi mieć dostęp do swojego środowiska.
- Moglibyśmy przekazać jej wszystkie potrzebne dane znajdujące się w jej środowisku jako dodatkowe parametry. Rozwiązanie takie stosuje się często w językach funkcyjnych.
- W językach imperatywnych najczęściej stosowanym rozwiązaniem jest powiązanie w listę ciągu ramek, które są na ścieżce w hierarchii zagnieżdżenia.
- Każda ramka zawiera łącze statyczne (SL, static link) – wskaźnik do jednego z rekordów aktywacji funkcji otaczającej daną.
- Rekord ten nazywamy poprzednikiem statycznym, a ciąg rekordów połączonych SL to łańcuch statyczny.

## Wyliczanie SL

- SL musi być liczony przez wołającego, bo do jego określenia trzeba znać zarówno funkcję wołaną jak i wołającą.
- Środowisko dla funkcji wołanej zależy od środowiska wołającej – jeśli obie widzą zmienną  $x$ , jej wartość ma być dla nich równa.
- Jeśli funkcja  $F$  znajdująca się na poziomie zagnieżdżenia  $F_p$  wywołuje  $G$  z poziomu zagnieżdżenia  $G_p$ , w pole SL wpisze adres rekordu, który odnajdzie przechodząc po własnym łańcuchu statycznym o  $F_p - G_p + 1$  kroków w górę.
- SL dla  $G$  ma wskazywać na rekord aktywacji funkcji na poziomie zagnieżdżenia  $G_p - 1$ , o  $\delta$  kroków w łańcuchu aktywacji od ramki  $F$ :

$$G_p - 1 = F_p - \delta$$

$$\delta = F_p - G_p + 1$$

## Wyliczanie SL

- Jeśli funkcja  $F$  znajdująca się na poziomie zagnieżdżenia  $F_p$  wywołuje  $G$  z poziomu zagnieżdżenia  $G_p$ , w pole SL wpisze adres rekordu, który odnajdzie przechodząc po własnym łańcuchu statycznym o  $F_p - G_p + 1$  kroków w górę.
- Jeśli np.  $G$  jest funkcją lokalną  $F$  (czyli  $G_p = F_p + 1$ ), funkcja  $F$  w pole SL dla  $G$  wpisze adres swojego rekordu aktywacji ( $F_p - G_p + 1 = 0$ )
- Jeśli  $F$  i  $G$  są na tym samym poziomie zagnieżdżenia, w polu SL dla  $G$  będzie to, co w SL dla  $F$  ( $F_p - G_p + 1 = 1$ )

## Dostęp do danych nielokalnych

- Dane lokalne funkcji są w jej rekordzie aktywacji a dane globalne w ustalonym miejscu w pamięci — można do nich sięgać za pomocą adresów bezwzględnych.
- Dostęp do danych nielokalnych przez SL; W funkcji  $F$  o poziomie  $F_p$  sięgamy do zmiennej z funkcji  $G$  o poziomie  $G_p$  przechodząc  $F_p - G_p$  kroków w górę po SL.
- Adres zmiennej jest wyznaczony przez poziom zagnieżdżenia i pozycję w rekordzie.
- Adresy rekordów z łańcucha statycznego można też wpisać do tablicy (tzw. display). Dzięki temu unikniemy chodzenia po łańcuchu statycznym, ale za to trzeba będzie stale aktualizować tablicę.



## Przykład 2

- Przyjmijmy, że w rekordzie aktywacji SL będzie się znajdował pomiędzy parametrami a śladem powrotu.
- Zakładamy też, że protokół wywołania i powrotu z funkcji jest prawie taki sam, jak w poprzednim przykładzie.
- Jedyną różnicą będzie dodanie po stronie wołającego obliczenia SL przed wywołaniem i usunięcia go ze stosu po powrocie sterowania.

## Przykład 2

W programie, którego fragment wygląda następująco:

```
procedure p;  
var a : integer;  
  procedure q;  
    var b : integer;  
    procedure r;  
      var c : integer;  
        procedure s; begin ... end {s};  
    begin  
      a:=b+c;  
      s;  
    q  
    end {r};  
  begin ... end {q};  
begin ... end {p};
```

## Przykład 2 — rekord aktywacji p

rekord aktywacji procedury p wyglądałby tak (w nawiasach podano przesunięcia poszczególnych pól względem pola DL wskazywanego przez rejestr BP):

( +2 )	SL
( +1 )	ślad
( +0 )	DL
( -1 )	a

## Przykład 2 — rekordy aktywacji q,r

ramka q:

( +2 ) SL  
( +1 ) ślad  
( 0 ) DL  
( -1 ) b

ramka r:

( +2 ) SL  
( +1 ) ślad  
( 0 ) DL  
( -1 ) c

Tablica symboli przy tłumaczeniu procedury r:

nazwa	poziom	$\delta$	offset
a	1	2	-1
b	2	1	-1
c	3	0	-1
p	1	2	
q	2	1	
r	3	0	
s	4	-1	

## Przykład 2 — kod procedury r

PUSH BP	# DL do rekordu aktywacji
BP = SP	# nowy adres ramki do BP
SP -= 1	# miejsce na zmienną c
S = [BP+2]	# SL - ramka q
B = [S-1]	# wartość b
B += [BP-1]	# dodaj wartość c
S = [S+2]	# SL - ramka p
[S-1] = B	# zapisz do a
PUSH BP	# SL dla s na stos
CALL s	# skok ze śladem do s
SP += 1	# usuwamy SL
S = [BP+2]	# adres ramki q
PUSH [S+2]	# adres ramki p - SL dla q
CALL q	# skok ze śladem do q
LEAVE	# powrót do poprzedniej ramki
RET	# i do miejsca wywołania

## Funkcje jako argumenty funkcji

- W językach bez zagnieżdżania funkcji, każda funkcja ma dostęp do zmiennych globalnych oraz własnych zmiennych lokalnych.
- W takiej sytuacji wystarczy przekazać adres kodu funkcji.
- W językach ze strukturą blokową, funkcja może mieć dostęp do danych nielokalnych, który realizowany jest przy pomocy SL.
- Jak ustawić SL przy wywołaniu funkcji otrzymanej jako parametr? Własny SL niekoniecznie jest tu dobrym rozwiązaniem. . .

## Przykład

```
procedure t (procedure p);  
begin  
  p;  
end {t}  
function f : int;  
  var a : int;  
  procedure x; begin a := 17 end  
begin {f}  
  t(x); f := a  
end {f}
```

- W momencie wywołania procedury *x* (przekazanej jako parametr *p*) wewnątrz *t*, SL dla *x* musi być ustawiony na *f*.
- Odpowiedni SL musi być zatem przekazany razem z adresem procedury.

## Przykład - kod

Ramka t:

```
( +4 )  p - SL
( +3 )  p - adres
( +2 )  SL
( +1 )  ślad
(  0 )  DL
```

Kod t:

```
# proc t(proc p) begin p; end
```

```
PUSH BP
```

```
BP = SP
```

```
PUSH [BP+4]      # SL
```

```
CALL [BP+3]
```

```
LEAVE
```

```
RET
```



## Przykład - kod

Ramka f:

( +3 )    wynik  
( +2 )    SL  
( +1 )    ślad  
(  0 )    DL  
( -1 )    a

Kod f:

```
# t(x); f := a

PUSH BP
BP = SP
SP -= 1    # a
PUSH BP    # x-SL
PUSH x     # x-adres
CALL T
SP += 2
A = [BP-1]
[BP+3] = A
```

## Protokół wywołania i386

Istnieje wiele wariantów, tu zajmiemy się protokołem używanym przez GCC+libc (aka "cdecl").

- przy wywołaniu na stosie argumenty od końca, ślad powrotu
- wołający zdejmuję argumenty
- przy powrocie wynik typu int/wskaźnik w EAX
- rejestry EBP,ESI,EDI,EBX muszą być zachowane

Standardowy prolog:

```
pushl    %ebp
movl     %esp, %ebp
subl     $x, %esp    /* zmienne lokalne */
```

Standardowy epilog:

```
movl     %ebp, %esp /* pomijane jesli nop */
popl     %ebp
ret
```

## Protokół wywołania x86-64

- Liczby całkowite przekazywane w EDI,ESI,EDX,ECX,R8D,R9D
- wskaźniki przekazywane w RDI,RSI,RDX,RCX,R8,R9
- jeśli więcej argumentów, lub większe niż 128-bitowe, to na stosie
- przy powrocie wynik typu int w EAX; wskaźnik w RAX
- rejestry RBP, RBX i R12 do R15 muszą być zachowane
- $RSP \equiv 0 \pmod{16}$  (przed CALL, czyli 8 po CALL)

Standardowy prolog:

```
pushl    %rbp
movl     %rsp, %rbp
subl     $x, %rsp    /* zmienne lokalne */
```

Standardowy epilog:

```
movl     %rbp, %rsp /* pomijane jeśli nop */
popl     %rbp
ret
```

## Pomijanie wskaźnika ramki

- Aktualny rekord aktywacji zawsze znajduje się na wierzchołku stosu; można do jego adresowania użyć wskaźnika stosu.
- Zalety: oszczędzamy jeden rejestr i kilka instrukcji na każde wywołanie.
- Wady: wierzchołek stosu przesuwa się podczas obliczeń, powodując zmiany przesunięć pól rekordu aktywacji; podatne na błędy w generowaniu kodu.
- W GCC opcja `-fomit-frame-pointer`
- Wymaga dodatkowych zabiegów przy obsłudze wyjątków

## Przykład

```
int sumto(int n)
{
    int i, sum;
    i = 0;
    sum = 0;
    while (i<n) {
        i = i+1;
        sum = sum+i;
    }
    return sum;
}
```

## Z ramką stosu

```
sumto:  pushq    %rbp
        movq     %rsp, %rbp
        testl    %edi, %edi
        jle      .L4
        movl     $0, %eax
        movl     $0, %edx

.L3:
        addl     $1, %edx
        addl     %edx, %eax
        cmpl     %edi, %edx
        jne      .L3
        jmp      .L2

.L4:
        movl     $0, %eax

.L2:
        popq     %rbp
        ret
```

## Bez ramki stosu

```
sumto:  testl    %edi, %edi
        jle     .L4
        movl    $0, %eax
        movl    $0, %edx

.L3:
        addl    $1, %edx
        addl    %edx, %eax
        cmpl    %edi, %edx
        jne     .L3
        rep ret      # tweak AMD

.L4:
        movl    $0, %eax
        ret
```

# Realizacja konstrukcji języków obiektowych

- W obiektowych językach programowania każdy obiekt posiada pewną wiedzę, przechowywaną w zmiennych instancyjnych (zmiennych obiektowych), ma również określone umiejętności reprezentowane przez metody.
- To, jakie zmienne instancyjne i jakie metody posiada obiekt danej klasy, wynika z definicji tej klasy oraz z definicji klas, z których dziedziczy bezpośrednio lub pośrednio.
- Omówimy teraz realizację mechanizmów obiektowości dla języka programowania z pojedynczym dziedziczeniem.



## Zmienne instancyjne

- Każdy obiekt posiada zmienne zdefiniowane w jego klasie, a także zmienne odziedziczone z nadklas.
- Reprezentacja obiektów jest analogiczna do rekordów — w obszarze pamięci zajęтым przez obiekt znajdują się wartości jego zmiennych instancyjnych.
- Kolejność tych zmiennych ma być zgodna z hierarchią dziedziczenia — zmienne zdefiniowane w klasie obiektu muszą się znaleźć na końcu, przed nimi są zmienne z klasy dziedziczonej itd. w górę hierarchii dziedziczenia.

## Zmienne instancyjne

Np. w programie zawierającym definicje klas:

```
struct A {  
    int w;  
    void writeA() { print(w) }  
};  
struct B : A {  
    int x,y;  
    void writeB() { print(w,x,y) }  
};  
struct C : B {  
    int z;  
    void writeC() { print(w,x,y,z) }  
};
```

metody `write...` wypisują wartości wszystkich zmiennych obiektu danej klasy.

## Zmienne instancyjne

obiekty klasy A będą zawierały jedną zmienną:

w
---

obiekty klasy B trzy zmienne w kolejności:

w	x	y
---	---	---

obiekty klasy C cztery zmienne w kolejności:

w	x	y	z
---	---	---	---

taka kolejność umożliwia metodom danej klasy prawidłowe działanie zarówno dla obiektów tej klasy, jak i jej podklas.

W obiekcie dziedziczącym zmienną znajduje się ona w tym samym miejscu, co w obiektach klasy dziedziczonej.

W naszym przypadku, zarówno w obiektach klasy B jak i C, zmienne `x`, `y` są odpowiednio na 2 i 3 pozycji.

Metoda `writeB` wie pod jakim przesunięciem te zmienne się znajdują niezależnie od rzeczywistej klasy obiektu

## Przesłanianie atrybutów

Ktoś mógłby zapytać *“a co jeśli podklasa redefiniuje atrybut?”*

W rzeczywistości nie jest to problem. Rozważmy definicje w C++

```
struct A {  
    int a;  
};  
struct B : A {  
    int b, a;  
    int sum() {A::a = a; return a-b;}  
};
```

W którym miejscu jest atrybut a?

Struktura obiektu klasy B jest następująca:

A::a	B::b	B::a
------	------	------

## Alternatywne rozwiązanie

Można dopuścić dowolną kolejność fragmentów odpowiadających poszczególnym klasom, np. 

C	B	A
---	---	---

Analizator typów przepisze metodę `B::writeB` mniej więcej tak:

```
void B::writeB(B *this){  
    write(this->Aptr->w, this->Bptr->x, this->Bptr->y);  
}
```

Jest to jak widać bardziej złożone i mniej efektywne, jednak daje większą elastyczność: teraz fragmenty A, B, C nie muszą nawet być obok siebie (ten fakt za chwilę nam się przyda).

Znane muszą być tylko przesunięcia `Aptr`, `Bptr`

# Metody wirtualne

Wywołanie metody w językach obiektowych różni się od wywołania funkcji/procedury w językach proceduralnych dwoma elementami:

- metoda otrzymuje jako dodatkowy ukryty parametr obiekt, dla którego ma się wykonać.
- w niektórych językach występuje mechanizm metod wirtualnych: wybór metody zależy od rzeczywistej (raczej niż deklarowanej) klasy obiektu i jest dokonywany podczas wykonania programu, a nie podczas kompilacji.

Reakcja obiektu na komunikat zależy od jego klasy. Jeśli w tej klasie jest metoda o nazwie takiej, jak nazwa komunikatu, wywołujemy ją, jeśli nie, to szukamy w nadklasie itd.

## Tablica metod wirtualnych

- Powszechnie stosowanym rozwiązaniem jest wyposażenie obiektu w tablicę metod wirtualnych, zawierającą adresy kodu odpowiednich metod.
- W językach z typami statycznymi, dopuszczalne komunikaty są znane podczas kompilacji. Można je ponumerować i reprezentować tablicę metod wirtualnych za pomocą zwykłej tablicy  $V$ , gdzie  $V[k]$  zawiera adres metody, którą należy wykonać w odpowiedzi na komunikat numer  $k$ .
- Wysłanie komunikatu  $k$  wymaga skoku ze śladem pod adres  $V[k]$ .
- Wszystkie obiekty danej klasy mogą mieć wspólną tablicę metod wirtualnych. W samym obiekcie umieszczamy jedynie adres tej tablicy.

## Budowa tablic metod wirtualnych

- Budowa tablic metod wirtualnych oraz numerowanie komunikatów odbywa się podczas kompilacji, na etapie analizy kontekstowej.
- Tablice metod wirtualnych dla poszczególnych klas budujemy w kolejności przejścia drzewa dziedziczenia z góry na dół".
- Tablica metod wirtualnych dla podklasy powstaje z tablicy dla nadklasy przez dodanie adresów metod zdefiniowanych w tej klasie.
- Jeśli metoda była już zdefiniowana "wyżej" w hierarchii, czyli jest redefiniowana, jej adres wpisujemy na pozycję metody redefiniowanej.
- Jeśli metoda pojawia się na ścieżce dziedziczenia pierwszy raz, jej adres wpisujemy na pierwsze wolne miejsce w tablicy metod wirtualnych.



# Wielodziedziczenie

Rozważmy klasy

```
struct A {int a;}  
struct B:A {int b;}  
struct C:A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy *D*?

Rozważmy klasy

```
struct A {int a;}  
struct B:A {int b;}  
struct C:A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy  $D$ ?

B::A::a	B::b	C::A::a	C::c
---------	------	---------	------

Obiektu klasy  $D$  możemy bez problemu używać jako obiektu klasy  $B$   
Konwersja  $(D^*) \mapsto (B^*)$  jest identycznością

Rozważmy klasy

```
struct A {int a;}  
struct B:A {int b;}  
struct C:A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy  $D$ ?

B::A::a	B::b	C::A::a	C::c
---------	------	---------	------

Obiektu klasy  $D$  możemy bez problemu używać jako obiektu klasy  $B$   
Konwersja  $(D^*) \mapsto (B^*)$  jest identycznością

Konwersja  $(D^*) \mapsto (C^*)$  wymaga przesunięcia wskaźnika o rozmiar  $B$

## Wirtualne nadklasy

```
struct A {int a;}  
struct B: virtual A {int b;}  
struct C: virtual A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy *D*?

# Wirtualne nadklasy

```
struct A {int a;}  
struct B: virtual A {int b;}  
struct C: virtual A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy *D*?

A::a	B::b	C::c
------	------	------

Obiektu klasy *D* możemy bez problemu używać jako obiektu klasy *B*  
Konwersja  $(D^*) \mapsto (B^*)$  jest identycznością

## Wirtualne nadklasy

```
struct A {int a;}  
struct B: virtual A {int b;}  
struct C: virtual A {int c;}  
struct D:B,C {}
```

Jak wygląda obiekt klasy  $D$ ?

A::a	B::b	C::c
------	------	------

Obiektu klasy  $D$  możemy bez problemu używać jako obiektu klasy  $B$   
Konwersja  $(D^*) \mapsto (B^*)$  jest identycznością

Żadna arytmetyka na wskaźnikach nie zapewni  $(D^*) \mapsto (C^*)$ .  
Trzeba użyć mechanizmu analogicznego do metod wirtualnych.  
Patrz “*Alternatywne rozwiązanie*” kilka slajdów wcześniej.