Jakub Wach,
Michał Pelczar

C++ od kuchni,
czyli rola kompilatora w modelu obiektu języka C++

1. Wprowadzenie

Języki umożliwiające programowanie metodą OO posiadają bardzo silnie rozbudowaną funkcjonalność obiektową. Funkcjonalność ta musi zostać w odpowiedni sposób zaimplementowana. Programista, operując na wyższym poziomie abstrakcji z reguły nie jest zainteresowany stroną implementacji, co jest poważnym błędem. Z reguły bowiem kod, jaki powstaje po kompilacji, przedstawia się zupełnie inaczej niż kod przez nas zamierzony. Może to prowadzić do pomyłek zwłaszcza, gdy programujemy wykorzystując bardzo zaawansowane elementy składni – im większy stopień skomplikowania kodu, tym mniejsza pewność, co tak naprawdę się w nim dzieje. Dodatkowym problemem jest istnienie różnych standardów i dialektów języka i szerokiego zakresu kompilatorów, na których konfigurację mamy bardzo mały wpływ i które niezbyt chętnie informują o **koncepcyjnych** zmianach dokonywanych w kodzie.

Zagadnienia omówimy na przykładzie języka C++ który podobnie jak większość stosowanych obecnie, łączy w sobie paradygmat proceduralny i obiektowy. Zaprezentowany pseudokod języka C++ opiera się na kompilatorze **cfront**.

2. Konstruktory

2.1.Konstruktory domyślne

Standardy mówią, że konstruktory domyślne generowane są "tam, gdzie to potrzebne" i gdzie są one **nietrywialne**, co obejmuje następujące przypadki:

- składowy obiekt klasy z konstruktorem domyślnym

```
class Foo (public: Foo(), .... }
class Bar { public: Foo foo; }
... Bar bar; ...
```

W tej sytuacji, należy stworzyć konstruktor, który wywoła konstruktor Foo:Foo().

Aby jednak synteza nie spowodowała konfliktów w przypadku wspólnej kompilacji kilku plików tworzących obiekty bar, konstruktor musi zostać zsyntetyzowany jako funkcja inline:

```
inline Bar:Bar() {
     foo.Foo::Foo(); *
}
```

która zawsze jest łączona statycznie.

Co więcej, linijka ta jest dopisywana we wszystkich istniejących konstruktorach.

- klasa podstawowa z kontruktorem domyślnym

Sytuacja analogiczna, konstruktor musi zostaje niejawnie wywołany.

- klasa z funkcją wirtualną

Należy zainicjalizować wskaźnik **vptr**.

- klasa z wirtualną klasą podstawową

Należy zainicjalizować wskaźniki do wirtualnych klas podstawowych, przykładowo dla klasy wirtualnej X wskaźnik __vbX.

2.2.Kontruktory kopiujące.

Dla przypomnienia, ogólna postać konstruktora kopiującego jest następująca: Konstruktor kopiujący X::X(const X& x)

Domyślą metodą kopiowania obiektu jest kopiowanie bit po bicie - i wówczas konstruktor w ogóle nie musi istnieć.

Jest on natomiast syntetyzowany, jeśli klasa nie ujawnia semantyki kopiowania bit po bicie:

```
class Square { char *str; int cnt; }

class Square { String str; int cnt; }

inline Square:Square( const Square* wd) {
     str.String::String(wd.str);
     cnt = wd.cnt;
}
```

Ponieważ, jak widać, niektóre atrybuty muszą zostać zainicjalizowane w szczególny sposób.

2.3.Optymalizacja NRV

Podstawowa metoda używana przy zwracaniu obiektów jako rezultaty wykonania funkcji. W miejscu wywołania metody tworzony jest obiekt. Referencja do tego obiektu jest potajemnie przesyłana do wnętrza funkcji, która pracuje na orginale. Żadne kopiowanie obiektu nie jest więc potrzebne. Schematycznie jest to przedstawione poniżej:

```
X bar() {
     X xx;
     //  operacje na xx
     return xx;
}


void bar(X& __result) {
     __result.X::X();
     // operacje na __result
     return;
}
```

Zasadniczą wadą tej optymalizacji jest to, że wykonuje się ona milcząco nie mamy jasności, czy została wykonana. Ponadto, dla skomplikowanych funkcji jest ona trudna do realizacji.

3. Semantyka danych

3.1. Rozmiar obiektu

Rozważmy następującą hierachę klas:

```
class X {};
class Y : public virtual X {};
class Z : public virtual X {};
class A : public Y, public Z {};
```

Występuje tutaj jak widać dziedziczenie wielokrotne.

Wielkości klas mogą, w zależności od kompilatora, być na przykład następujące:

sizeof(X)	1B
sizeof(Y)	8B
sizeof(Z)	8B
sizeof(A)	12B

Klasa pusta nie ma zerowego rozmiaru.
Kompilator wstawia składową typu char, co pozwala przydzielać obiektom różne adresy w pamięci.

Ponadto, narzut na rozmiar pozostałych klas jest wypadkową czynników:

- wsparcie językowe

  Wskaźnik do podobiektu wirtualnej klasy podstawowej albo tablicy zawierającej jego adres lub przesunięcie.

- rozmieszczenie w pamięci

  Wyrównanie do pełnej liczby bajtów, co umożliwia efektywniejsze ładowanie i zapisywanie.

- optymalizacja rozpoznanych przez kompilator przypadków szczególnych

W praktyce, pusta wirtualna klasa podstawowa jest używana w C++ jako wirtualny interfejs.
Nowe kompilatory traktują taką klasę, jakby nakładała się na początek obiektu klasy pochodnej. Nie zajmuje ona żadnej dodatkowej pamięci. Jest to przykład ewolucyjnego charakteru modelu obiektu C++. Model dotyczy przypadku ogólnego. W miarę, jak wykrywane są pewne przypadki szczególne, wprowadza się heurystyki umożliwiające ich optymalizację

3.2.Organizacja danych w pamięci, konsekwencje na przykładzie procedury przypisania

Dziedziczenie bez polimorfizmu

Obiekt klasy pochodnej jest w istocie konkatenacją jej składowych ze składowymi klas podstawowych (standard nie określa kolejności części klasy pochodnej i podstawowej).

Dane znajdują się w jednym miejscu, ale korzystać z nich i operować na nich mogą dwie lub więcej powiązanych ze sobą abstrakcji. Dlatego też **dziedziczenie nie dokłada do reprezentacji żadnego narzutu na pamięć oraz czas dostępu**.

Dziedziczenie z polimorfizmem

Aspekty implementacji dziedziczenia z polimorfizmem są następujące:

- Wprowadzenie tablicy wirtualnej zawierającej adresy wszystkich funkcji wirtualnych zadeklarowanych w klasie. Jej rozmiar jest proporcjonalny do liczby zadeklarowanych funkcji oraz miejsca do rozpoznawania typów w czasie wykonania.

- Wprowadzenie do każdego obiektu wskaźnika **vptr**. Pozwala on efektywnie sięgać do tablicy wirtualnej.

- Rozszerzenie konstruktora o inicjowanie wskaźnika vptr obiektu adresem tablicy wirtualnej jego klasy. Możliwe są tutaj optymalizacje.

- Rozszerzenie destruktora polegające na zmianie wskaźnika vptr na adres związanej z klasą tablicy wirtualnej. Optymalizacja może wyeliminować część z tych przypisań.

Nowsze kompilatory umieszczają wskaźnik **vptr na początku** klasy. Pozwala na to na efektywniejsze wywoływanie funkcji wirtualnych. Gdyby przyjąć inne rozwiązanie, w czasie wykonania trzeba by udostępniać nie tylko przemieszczenie początku klasy, ale i położenie wskaźnika **vpt**r tej klasy.

Zasadniczo, przypisanie wskaźnika do obiektu klasy pochodnej do wskaźnika do obiektu klasy bazowej:

```
Vehicle veh;
Bike*bike = &veh;
```

nie wymaga żadnych interwencji kompilatora ani modyfikacji adresu, jest zatem w pełni efektywne.
Istnieje jeden przypadek, w którym takie rozlokowanie wskaźnika wymusza ingerencję kompilatora przy dokonaniu przypisaniu. Jeśli klasa podstawowa nie ma funkcji wirtualnych, natomiast klasa pochodna je zawiera. Należy wówczas skorygować adres o rozmiar wskaźnika **ptr**. Niektóre kompilatory również na tym polu stosują odpowiednie optymalizacje.

Wielodziedziczenie

W wielodziedziczeniu operacje podstawienia obiektów muszą przebiegać w mniej nienaturalny sposób.

Przypisanie adresu obiektu podlegającemu wielodziedziczeniu do wskaźnika do obiektu jego pierwszej klasy podstawowej nie różni się niczym od przypadku dziedziczenia pojedynczego, ponieważ oba obiekty zaczynają się w tym samym miejscu.:

```
class A {};
class B {};
class AB : public A, B {};

AB ab;
A* aptr = &ab;
```

Przypisanie do drugiej albo dalszych klas wymaga już modyfikacji adresu, polegającej na dodaniu rozmiaru leżących “po drodze” obiektów klas podstawowych.
Tak więc podstawienie:

```
AB ab;
B* bptr = &ab;
```

musi być zamienione przez kompilator na postać:

```
bptr = (B*)(((char*)&ab) + sizeof(A));
```

Zauważmy, że takie rozwinięcie nie będzie działać prawidłowo dla podstawienia wskaźników:

```
AB* abptr;
B* bptr = abptr;

bptr = (B*)(((char*)&ab) + sizeof(A));
```

ponieważ gdyby wskaźnik ab był równy 0 , to w bptr znalazłaby się wartość sizeof(A).

Tak więc, pseudokod w wersji finalnej powinien mieć postać:

```
bptr = ab ? (B*)(((char*)&ab) + sizeof(A)) : 0;
```

Te przykłady uświadamiają nam, jak wiele operacji może kryć się za zwykłym podstawieniem adresów, zarówno na etapie kompilacji, jak i na etapie wykonania.

Dziedziczenie wirtualne

Klasę zawierającą jeden lub więcej podobiektów wirtualnych klas podstawowych dzieli się na dwa obszary: **niezmienny** i **wspólny**. Dane w obszarze niezmiennym zachowują ustalone przesunięcia względem początku obiektu niezależnie od późniejszych dziedziczeń, można więc odnosić się do nich bezpośrednio. Obszar wspólny reprezentuje podobiekty wirtualnych klas podstawowych - zmienia się on przy każdym dziedziczeniu. Różne implementacje modelu różnią się pod względem metody owego pośredniego dostępu.
Ogólna strategia polega na umieszczeniu w pamięci najpierw obszaru niezmiennego a potem wspólnego - należy jednak jeszcze zapewnić dostęp do wspólnego obszaru klasy.

Pierwotnym rozwiązaniem było wstawienie do obiektów pochodnych wskaźników do wszystkich wirtualnych klas podstawowych. Tak więc każde odniesienie do atrybutu klasy wirtualnej:

```
class P {int x;};
class B : public P { x = 10; };
```

jest przez kompilator zaimplementowane jako:

```
vbcP->x = 10;
```

Natomiast konwersja z klasy pochodnej do podstawowej:

```
B* Bptr;
P* Pptr = Bptr;
```

przyjmie postać:

```
P* Pptr = Bptr ? Bptr->__vbcP : 0;
```

Przy takim podejściu, liczbą wywołań pośrednich rośnie wraz z długością łańcucha dziedziczenia wirtualnego. Dla n poziomów – przejście przez n składników a pamiętajmy, że jest to odniesienie do zwykłego obiektu składowego. Czas dostępu nie jest stały. Sposobem na rozwiązanie tego problemu jest umieszczenie kopii wskaźników do wszystkich zagnieżdżonych klas wirtualnych w każdym pochodnym obiekcie.
Pożądany byłby również stały względem liczby klas wirtualnych narzut związany z jednym wskaźnikiem w obiekcie klasy. Dwa sposoby rozwiązania tego problemu zostaną zaprezentowane w dalszej części.

4. Semantyka funkcji

4.1.Wprowadzenie

Nieodłączną cechą definicji obiektu w języku C++ są funkcje składowe klas, czyli metody. Wynikają one rzecz jasna bezpośrednio z hermetyzacji.
Tak jak wszystkie inne składowe klas również metody muszą być reprezentowane w kodzie strukturalnie. Jak to się dzieje zobaczmy zaraz na przykładzie tłumaczenia do strukturalnego psedokodu C++.

4.2.Kodowanie nazw

Deklarując metody w obiektowym kodzie C++ nie musimy zupełnie przejmować się takimi samymi metodami w różnych klasach, ponieważ za ich rozróżnianiem stoi kwalifikator <nazwa klasy>::, ani też przeciążonymi metodami tej samej klasy, gdyż rozróżnia je lista argumentów.
W kodzie strukturalnym powstaje jednak problem – wszystkie metody będą należeć do tej samej przestrzeni nazw, co grozi potencjalnymi konfliktami. Oczywistym rozwiązaniem jest kodowanie nazw. Przykładowe rozwiązanie, wprowadzone przez autora języka C++ przedstawia się następująco:

```
<przedrostek>_<nazwa klasy><parametry>
```

gdzie:

`<przedrostek>` związany jest ze specyfiką metody (np metoda operatorowa)
`<parametry>` zaś kodowane są w sposób zachowujący ich typ, rodzaj oraz kwalifikatory (np `rcf` oznacza stałą-const referencję & do danej float). Każdy kwalifikator i typ wbudowany posiada własny kod, typy uzytkownika (klasy, struktury) identyfikowane są poprzez pełną nazwę.

W ten sposób odwzorowuje się jednoznacznie każdą metodę i gwarantuje że każde wywołanie niewłaściwej (np inaczej zdefiniowanej niż zadeklarowanej) wersji zostanie wykryte w czasie łączenia. Nazywa się to *łączeniem bezpiecznym pod względem zgodności typów.*

4.3.Niestatyczne funkcje składowe

Każda metoda niestatyczna podlega wewnętrznemu przekształceniu na równoważną wersję nie będącą składową (zwyką funkcję). Jest to związane z kryterium projektowania C++ które zakładało, iż każda metoda musi być przynajmniej tak samo

efektywna jak jej odpowiednik nie będący składową.
Przekształcenie to nazywa się *przekształceniem z niejawnym wskaźnikiem this* i składa z następujących etapów:

- rozszerza się nagłówek metody o koljny argument-niejawne *this* - <klasa>* const this
- każde bezpośrednie odwołanie do niestatycznej danej składowej zastępuje się odwołaniem poprzez nowy argument
- metodę koduje się wg przyjętej zasady oraz zamienia na zwykłą funkcję
- wszystkie odwołania do danej metody w programie zostają zamienione w sposób uwzględniający uprzednie zmiany

4.4 Statyczne funkcje składowe

W przypadku składowych statycznych kompilator ma jeszcze mniej pracy niż w przypadku metod niestatycznych, ponieważ *this* nie jest potrzebne do wewnętrznych odwołań. Składnia metod statycznych gwarantuje bowiem iż nie będą one sięgać do składowych niestatycznych oraz że nie będą deklarowane z kwalifikatorami *const, volatile* ani *virtual*.
Cała praca wykonywana dla metod satycznych sprowadza się do zakodowania nazwy oraz wyrzucenia poza deklarację klasy (staje się normalną funkcją).

4.5 Wirtualne funkcje składowe

Z punktu widzenia twórcy kompilatora, najciekawsze i wymagające zarazem największego nakładu pracy są zagadnienia dotyczące mechanizmów *polimorfizmu* i *dziedziczenia wielobazowego* a w szczególności ich połączenie.
W języku C++ polimorfizm wprowadzony został poprzez dodanie kwalifikatora *virtual*.
Naturalną konsekwencją takiego stanu rzeczy jest, iż każda klasa posiadająca przynajmniej jedną metodę wirtualną powinna być traktowana przez kompilator jako możliwie polimorficzna.
Zagadnienie to omówimy w dwóch podpunktach.

4.5.1.Funkcje wirtualne przy dziedziczeniu pojedynczym

Problemem, na jaki napotyka kompilator w przypadku wywołania funkcji wirtualnej np postaci *wskaznik->metoda()* jest identyfikacja, znalezienie i wywołanie odpowiedniej wersji wirtualnej funkcji *metoda()*. Informacja ta dostępna jest w czasie wykonania, potrzebne jest więc jakieś rozwiązanie ktore tą informację dostarczy.
Ponieważ w jezyku C++ zbiór funkcji wirtualnych jest znany w czasie kompilacji i niezmienny w czasie wykonania, kompilator może wygenerować pewien rodzaj tablicy funkcji wirtualnych, z której brane będą odpowiednie informacje w czasie wykonania.
Ogólny model implementacji metod wirtualnych jest zatem następujący:

- każda klasa polimorficzna posiada własną tablicę wirtualną, która zawiera adresy jej metod wirtualnych oraz informację o typie.
- Każda instancja danej klasy posiada wskaźnik do tablicy wywołań wirtualnych, odpowiadającej jej typowi
- Każdej funkcji wirtualnej przypisywany jest ustalony indeks w tablicy wirtualnej

Dzięki takiemu modelowi wywołanie metody wirtualnej

```
wskaznik->metoda()
```

zostaje wewnętrznie przez kompilator przekształcone na:

```
(*wskaznik->vptr[offset])(wskaznik)
```

gdzie drugie wystąpienie *wskaznik* reprezentuje niejawne *this* (tak jak w przypadku wszystkich metod niestatycznych).

Sama tablica, generowana osobno dla każdej klasy, zawiera wszystkie wskaźniki do funkcji wirtualnych które:

- definiowane są w klasie zastępując ewetualne wersje z superklasy
- dziedziczone są z superklasy o ile nie zostały zastąpione w klasie pochodnej
- są funkcjami czysto wirtualnymi (abstrakcyjnymi) – wszystkie zostają zastapione jednym wywołaniem bibliotecznym *pure_virtuall_called()* która ma za zadanie obslugiwać wyjątek czasu wykoania jakim byłoby wywołanie metody abstrakcyjnej

W przypadku omawianego tutaj dziedziczenia pojedynczego klasy pochodnej z klasy bazowej posiadającej funkcje wirtualne może zachodzić jeden z trzech przypadków (dla każdej z metod wirtualnych klasy pochodnej):

- klasa pochodna dziedziczy wersję funkcji z superklasy – adres jest kopiowany z tablicy superklasy do tablicy klasy pochodnej
- klasa pochodna wprowadza swoją wersję funkcji – nowa tablica zawiera adres ostatniej wersji
- klasa pochodna wprowadza nową funkcję wirtualną – tablica powiększa się o jedno miejsce w którym umieszcza się adres nowej funkcji (należy zwrocić uwagę na fakt iż zostaje dodana na końcu, gdyż każdy indeks związany raz z daną funkcją pozostaje z nią związany w całej hierarchii dziedziczenia).

4.5.2.Funkcje wirtualne i dziedziczenie wielobazowe

W porównaniu z realizacją funkcji wirtualnych przy dziedziczeniu pojedynczym mechanizm obowiązujący przy dziedziczeniu wielobazowym jest znacznie trudniejszy. Jego złożoność bierze się z obecności drugiej (i ewentualnie kolejnych) superklas i koncentruje wokół konieczności uaktualniania wskaźnika *this* w czasie wykonania. Skąd bierze się taka konieczność? Przykładowo wystarczy, że instancję klasy *Pochodna : public Superklasa1, public Superklasa2* przypiszemy wskaźnikowi typu *Superklasa2*. Jeżeli nie przesunęlibyśmy wskaźnika *this* na podobiekt typu *Superklasa2* (czyli w tym przypadku o *sizeof(Superklasa1)* bajtów) żadne niepolimorficzne, odnoszące się do składowych z Superklasa2 odwołanie nie byłoby poprawne!
Jak widać zatem, kompilator musi wygenerować rozmiar przesunięcia oraz kod który będzie je dodawał. Wyróżnia się dwa typowe podejścia do tego problemu:

- oryginalne podejście zastosowane przez Bjarne'a Stroustrupa, ktore polegało na rozszerzeniu tablicy funkcji wirtualnych o ewentualne przesunięcie dla każdej metody. Wadą jest oczywiście dodatkowy nakład pamięciowy i obliczeniowy niezależny od tego czy jest potrzeba poprawki czy też nie.
- Zastosowanie tzw *mikroprocedury thunk.* Polega ona na dodaniu niewielkiego kodu który przesuwa wskaźnik *this* i wywołuje metodę już z przesunięciem. Rzecz jasna procedura dodawana jest tylko tam, gdzie zachodzi konieczność modyfikacji *this*.

Ponieważ, jak to już zostało wcześniej napisane, indeksy w tablicy zostają przypisane do metod w całej hierarchii dziedziczenia na stałe, klasy które dziedziczą wielobazowo muszą posiadać wiele tablic wirtualnych wskazywanych przez wiele wskaźników **vptr** (wiele oznacza dokładnie tyle ile klas bazowych). Pierwsza tablica jest wspólna dla kalsy pochodnej oraz pierwszej superklasy (dlatego też przy dziedziczeniu pojedynczym jest tylko jedna tablica wirtualna). Należy tutaj zuwazyć fakt, iż współczesne kompilatory łączą wiele tablic w jedną, kolejne wskaźniki generując poprzez dodanie

przesunięcia do adresu tablicy połączonej.

Na początku tego podpunktu wspomniałem o konieczności poprawianiu wskaźnika *this* w przypadku wejścia do metody. Zdażyć się jednak może również konieczność poprawienia również wskaźnika zwracanego (np dla metody, która zwraca aktualną instancję ale jest wywołana ze wskaźnika na superklasę). Taka sytuacja jest jak najbardziej dopuszczalna przez mechanizm polimorfizmu i wymaga aby zwrócić instancję superklasy. W tym celu kompilator korzysta ze strategii *"funkcji podwójnych"* – metoda generowana jest w dwóch wersjach realizujących ten sam algorytm, przy czym w drugiej wersji przed zwróceniem wartości *this* jest aktualizowany o stosowne przesunięcie. Dodane przesunięcie zależy od oczekiwanego typu (dla wskaźnika na konkretną superklasę zwracamy właściwy jej podobiekt).

5. Semantyka konstruowania

W języku C++ zakłada się że każda deklaracja nowej zmiennej danego typu pociąga za sobą wywołanie konstruktora, nawet jeżeli nie dokonuje się w danym miejscu inicjalizacji (poprzez listę argumentów konstruktora lub inny obiekt danego typu). Dla każdego typu zatem kompilator jest zobowiązany syntetyzować trywialny konstruktor. Rzecz jasna, jest to wymóg czystko formalny, gdyż rezerwacja pamięci jest wykonywana w innym miejscu, tak jak w języku C, zatem większość kompilatorów w praktyce nie generuje ani nie wywołuje trywialnych konstruktorów.
Oczywiście sam mechanizm wywołania konstruktora jest prosty jedynie w najbardziej trywialnych przypadkach, tj wtedy gdy kompilator nie ma do czynienia z polimorfizmem lub dziedziczeniem.Kiedy w grę wchodzą wymienione mechanizmy języka, kompilator musi zadbać o odpowiednie zmiany w kodzie. W najogólniejszym przypadku ich lista przedstawia się następująco:

(1) Jeżeli klasa jest polimorficzna (posiada funkcje wirtualne), a co za tym idzie należy inicjować **vptr** (wskaźnik do tablicy wirtualnej) należy zsynetyzować odpowiedni konstruktor kopiujący oraz operator kopiowania. Operator naiwny kopiujący bit po bicie mógłby bowiem nadac wskaźnikowi **vptr** złą wartość w przypadku kopiowania z instancji klasy pochodnej. Nowozsyntetyzowany operator (i konstrutor kopiujący) musi nadać nowemu wskaźnikowi **vptr** wartość odpowiadającą adresowi tablicy wirtualnej klasy podstawowej, a nie pochodnej, jak by to zrobił jego naiwny odpowiednik.
(2) Reszta zmian zachodzi w obrębie samego konstruktora:
- Najpierw należy zadbać o wywołanie konstruktorów wszystkich wirtualnych klas podstawowych w kolejności przeszukiwania od lewej do prawej i w głąb grafu hierarchii dziedziczenia
  - jeżeli klasa znajduje się na liście inicjalizacji składowych wywołania konstruktora to należy przekazać jawne argumenty. W przeciwym wypadku wywołuje się konstruktor domyślny
  - należy zapewnić również dostęp w RT do przesunięć wszystkich podobiektów wirtualnych superklas w tej klasie
  - konstruktory superklas wirtualnych wywoływane są tylko gdy obiekt danej klasy jest ostatnim w hierarchii dziedziczenia, o co dba osobny mechanizm
- Następnie należy wywołać konstruktory bezpośrednich superklas w kolejności jak deklaracja.
  - Jeżeli klasa znajduje się na liście inicjalizacji składowych to należy przekazać stosowne argumenty. W przeciwnym wypadku wywołuje się konstruktor domyślny – o ile taki jest zadeklarowany.
  - Jeżeli superklasa jest drugą lub następną w kolejności to należy tradycyjnie

poprawić *this*

- Potem wskaźniki **vptr** muszą zostać zainicjalizowane adresmi odpowiednich tablic wirtualnych odpowiednich superklas.
- Obiekty składowe inicjowane na liście inicjowania składowych wywołania konstruktora należy umieścić wewnątrz konstruktora dokładnie w kolejności ich deklaracji.
- Obiekty składowe, których nie ma na liście inicjalizacji składowych muszą zostać zainicjalizowane poprzez wywołanie ich domyślnych konstruktorów (z poprzedniego punktu wiemy że dzieje się tak tylko dla istniejących konstruktorów domyślnych).

Kilka słów należy powiedzieć o wspomnianym mechaniźmie kompilatora dbającym o odpowiednie wywołania konstruktorów przy dziedziczeniu wirtualnym.

- Pierwsze rozwiązanie tego problemu opierało się na wprowadzeniu do konstruktora (dokładniej rzecz ujmując wszystkich konstruktorów w hierarchii) dodatkowego parametru – flagi, wskazującej czy dana klasa jest "ostatnia" i konstruktor wirtualnej superklasy powinien być wywołany. Ciało konstruktora rozszerzane było o kod testujący flagę i ewentualnie wywołujący żądany konstruktor. Sam zas kompilator zajmował się wyszukaniem ostatniej w hierarchii klasy i przekazaniu jej konstruktorowi flagi ustawionej na **true** oraz przesłaniu flagi **false** w górę hierarchii w razie ewentualnych wywołań.
- Późniejsze, wydajniejsze rozwiązania wprowadziły podwajanie konstruktorów – każdy posiada dwie wersje: dla ukończonego obiektu (gdy konstruowana instancja jest danego typu) oraz dla podobiektu (gdy konstruowana instancja jest typu dziedziczącego z typu konstruktora). Ta pierwsza bezwarunkowo wywołuje konstruktory superklas wirtulanych oraz inicjalizuje **vptr** (o tym za chwilę). Ta druga natomiast nie wywołuje konstruktorów wirtualnych superklas i może nie inicjować **vptr**.

Ostatnia kwestia, którą musi zająć się kompilator w kontekście konstrukcji obiektu jest inicjalizacja "demonicznych" wskaźników **vptr**. Opisany powyżej mechanizm, zakładający ich inicjalizację po fazie wywołań konstruktorów superklas działa dobrze, z wyjątkiem sytuacji gdy któryś konstruktor superklasy zarząda wywołania funkcji wirtualnej.
Najprostsze nasuwające się rozwiązanie - przesunięcie inicjalizacji przed wołania konstrutorów superklas niestety również byłoby błędne, gdyż zgodnie z mechanizmem polimorfizmu wywołane zostałyby wersje ostatnie. Język zaś nakłada obowiązek wołania w konstruktorach danej klasy wersji przynależnej tej własnie klasie (a nie klasie pochodnej na przykład!).
Kolejne rozwiązanie, wypełniające tym razem nakazy języka, zakłada ustawienie w konstruktorze specjalnej flagi która w odpwiednich przypadkach nakazywałaby wywołania statyczne (typu <klasa>::<metoda>). Jest to dosyć nieeleganckie, dlatego też konstruktorzy kompilatorów preferują inne, prostsze i zdecydowanie elegantsze.
Otóż każdy konstruktor klasy podstwowej ustawia wskaźnik **vptr** konstruowanego obiektu na adres tablicy wirtualnej swojej klasy. W ten sposób, na czas działania danego konstruktora obiekt staje się instancją tej klasy, a wszystkie wywołania wirtualne realizowane są poprawnie.

6. Semantyka destrukcji

Destrukcja, czyli niszczenie obiektów w C++ odbywa się podobnie jak konstrukcja, przy czym główną różnicą jest odwrotna kolejność oraz wołanie destruktorów.

Podobnie jak to było w przypadku konstruktorów również teraz zachodzi przypadek, gdy kompilator musi stworzyć konstruktor jeżeli nie został on zadeklarowany przez użytkownika. Dzieje się tak jeżeli dana klasa ma superklasę lub klasę składową posiadającą destruktor.
W ogólnym przypadku każdy (również tworzony przez kompilator) destruktor rozszerzany jest o następujące elementy:

- jeżeli obiekt jest typu polimorficznego (zawiera vptr) to tenże wskaźnik przyjmuje wartość adresu tablicy wirtualnej związanej z daną klasą
- wykonywany jest kod uzytkownika (o ile takowy jest)
- wołane są destruktory składowych obiektów klasy w kolejności odwrotnej do ich deklaracji
- wołane są destruktory bezpośrednich niewirtualnych superklas w kolejności odwortnej do ich deklaracji
- jeżeli dana klasa jest ostatnia w hierarchii to wołane sa destruktory wirtualnych superklas w kolejności odwrotnej niż przy konstruowaniu

Dokładnie tak jak w przypadku konstruktorów nowsze podejście zakłada tworzenie podwójnych konstruktorów – w wersji dla pełnego obiektu oraz w wersji dla podobiektu z klasy podstawowej, w celu zwiększenia efektywności i usunięcia kodu zaleznego od stanu przekazywanych w argumentach flag.

7. Zagadnienia semantyki czasu wykonania

Specyfika języka C++ sprawia, że w czasie wykonania zazwyczaj dzieje się o wiele więcej, niż spodziewa się programista piszący kod. Czasem dodatkowego kodu genererowanego przez kompilator może być więcej, niż kodu napisanego przez programistę i to w konstrukcjach sprawiających na pierwszy rzut oka wrażenie trywialnych.

- Przykładem takiego wyrażenia jest na przykład porównanie obiektow dwóch różnych klas, np

```
klasa1 fst;
klasa2 snd;
if (fst == snd) {...}
```

Taki prosty z punktu widzenia kod zostanie (przy założeniu iż zdefiniowane są żądane operatory porównania i konwersji) zamieniony na coś w rodzaju:

```
{
   klasa1 temp1 = snd.operator klasa1();
   int temp2 = fst.operator==(temp1);

   if (temp2) {..}

   temp1.klasa1::~klasa1();
}
```

Jak widać kodu dodatkowego jest całkiem sporo.

- To samo dotyczy prostych sytuacji tworzenia noweg obiektu. Dla przykładu jeżeli przed instrukcją `switch` umieszczoną w osobnym bloku zadeklarujemy obiekt pewnej klasy, to wywołanie destruktora znajdzie się w każdej gałęzi `case`. Dzieje się tak, ponieważ język gwarantuje wywołanie destruktora przed wyjściem z zasięgu.

- Kolejnym przykładem jest konstrukcja tablicy instancji. Język wymaga, aby dla każdej

instancji w tablicy wywoływany był domyślny (bezparametrowy) konstruktor. Przy wyjściu z zasięgu natomiast dla każdej pozycji w tablicy musi zostać wywołany destruktor. Aby sobie z tym poradzić kompilatory wyposażone zostają w specjalne funkcje biblioteczne, osobne dla konstrukcji i destrukcji tabel. Niektóre implementacje dublują jeszcze liczbę tych funkcji, tworząc osobne wersje dla klas z wirtualnym dziedziczeniem. Swego rodzaju ciekawostką jest iż funkcje te, aby obsłużyć wszystkie klasy pobierają jako jeden z arguemntów adres konstruktora, co na poziomie "zwykłego śmiertelnika" jest surowo zabronione ;-) Dodatkowo niektóre kompilatory w celu obsługi argmentów domyślnych konstruktora tworzą wewnętrzny bezargumentowy konstruktor z argumentami domyślnymi podanymi jawnie, co również stoi w sprzeczności z regułami języka C++.

Zagadnień związanych ze zmianami w kodzie, dokonywanymi przez kompilator bez wiedzy programisty jest całe mnóstwo i liczba ich rośnie wraz z kolejnymi wprowadzanymi optymalizacjami (np dotyczącymi strategii tworzenia obiektów tymczasowych). Dlatego też patrząc na napisany właśnie piękny, świeży kod w C++ warto czasem zastanowić się nad aspektami dodającymi kompilatorowi dużo pracy a pomijalnymi z punktu widzenia implementowanego algorytmu.