

20.4. Klasy abstrakcyjne

Wiele klas jest podobnych do klasy `Employee` pod tym względem, że są przydatne same w sobie, jako interfejsy dla klas pochodnych oraz jako część implementacji klas pochodnych. Do tworzenia takich klas wystarczą techniki opisane w sekcji 20.3.2. Jednak nie wszystkie klasę mają taki wachlarz zastosowań. Niektóre, np. `Shape`, reprezentują abstrakcyjne koncepcje, dla których nie ma obiektów. Klasa `Shape` może być tylko bazą dla klas pochodnych. Można się o tym przekonać, próbując podać w niej sensowne definicje klas wirtualnych:

```
class Shape {
public:
    virtual void rotate(int) { throw runtime_error("Shape::rotate"); } //nielegantne
    virtual void draw() const { throw runtime_error("Shape::draw"); }
    ...
};
```

Tworzenie kształtów tego nieokreślonego typu jest bezsensowne, ale dozwolone:

```
Shape s; // bez sensu: bezkształtny kształt
```

Nie ma to sensu, bo każda operacja na obiekcie `s` spowoduje błąd.

Dlatego lepiej jest funkcje wirtualne klasy `Shape` zadeklarować jako **funkcje czysto wirtualne**. Służy do tego „pseudoinicjator” =0:

```
class Shape { // klasa abstrakcyjna
public:
    virtual void rotate(int) = 0;           //funkcja czysto wirtualna
    virtual void draw() const = 0;          //funkcja czysto wirtualna
    virtual bool is_closed() const = 0;     //funkcja czysto wirtualna
    ...
    virtual ~Shape();                     //wirtualny
};
```

Klasa zawierająca przynajmniej jedną funkcję czysto wirtualną jest **klassą abstrakcyjną**, a więc bez możliwości tworzenia jej obiektów:

```
Shape s; // błąd: zmienna abstrakcyjnej klasy Shape
```

Klasa abstrakcyjna służy jako interfejs do obiektów, do których dostęp uzyskuje się poprzez wskaźniki i referencje (w celu zachowania polimorfizmu). Dlatego też klasę takie zwykle muszą mieć destruktor wirtualny (3.2.4, 21.2.2). Jako że interfejsu dostarczanego przez klasę abstrakcyjną nie można użyć do tworzenia obiektów za pomocą konstruktora, klasę takie zazwyczaj nie mają konstruktorów.

Klasy abstrakcyjne można użyć tylko jako interfejsu do innych klas. Na przykład:

```
class Point { /* */};

class Circle : public Shape {
public:
    Circle(Point p, int r);

    void rotate(int) override { }
    void draw() const override;
    bool is_closed() const override { return true; }
    ...
};
```

```
private:
    Point center;
    int radius;
};
```

Funkcja czysto wirtualna, której nie zdefiniowano w klasie pochodnej, pozostaje czysto wirtualna, w związku z czym klasa pochodna jest abstrakcyjna. Dzięki temu implementacje można budować etapami:

```
class Polygon : public Shape { // klasa abstrakcyjna
public:
    bool is_closed() const override { return true; }
    //...funkcje draw i rotate nie są przesłonięte...
};

Polygon b {p1,p2,p3,p4}; // błęd: deklaracja obiektu abstrakcyjnej klasy Polygon
```

Klasa `Polygon` też jest abstrakcyjna, bo nie przesłoniliśmy w niej funkcji `draw()` i `rotate()`. Dopiero gdy to zrobimy, otrzymamy klasę, której obiekty będzie można tworzyć:

```
class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    Irregular_polygon(initializer_list<Point>);

    void draw() const override;
    void rotate(int) override;
    ...
};

Irregular_polygon poly {p1,p2,p3,p4}; // założenie, że p1-p4 to zdefiniowane gdzieś obiekty klasy Point
```

Klasa abstrakcyjna dostarcza interfejs bez ukazywania szczegółów implementacyjnych. Na przykład system operacyjny może ukrywać szczegóły swoich sterowników urządzeń za klasą abstrakcyjną:

```
class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...) = 0; // sterowanie urządzeń wejścia i wyjścia

    virtual ~Character_device() { } // destruktor wirtualny
};
```

Teraz sterowniki poszczególnych urządzeń można definiować jako klasy pochodne klasy `Character_device` i manipulować różnymi sterownikami poprzez ten interfejs.

Styl projektowania oparty na klasach abstrakcyjnych nazywa się **dziedziczeniem interfejsu**, w odróżnieniu od **dziedziczenia implementacji** opartego na klasach bazowych ze stanem i zdefiniowanymi funkcjami składowymi. Można też stosować kombinacje tych dwóch technik. Czyli można na przykład zdefiniować klasę bazową mającą zarówno stan, jak i funkcje czysto wirtualne i jej używać. Jednak takie zabawy mogą być mylące i wymagają wytężonej uwagi.

Od kiedy poznaliśmy klasy abstrakcyjne, mamy podstawowy zestaw narzędzi do napisania kompletnego programu zbudowanego na bazie klas.

20.5. Kontrola dostępu

Składowa klasy może być prywatna (`private`), chroniona (`protected`) lub publiczna (`public`):

- Jeśli składowa jest prywatna, jej nazwy mogą używać tylko funkcje składowe i zaprzyjaźnione klasy, w której jest zadeklarowana.
- Jeśli składowa jest chroniona, jej nazwy mogą używać tylko funkcje składowe i zaprzyjaźnione klasy, w której jest zadeklarowana, oraz funkcje składowe i zaprzyjaźnione klas pochodnych tej klasy (zobacz 19.4).
- Jeśli składowa jest publiczna, może być używana przez wszystkie funkcje.

Powysze zasady odzwierciedlają zasadę, że są trzy rodzaje funkcji dostępu do klas: funkcje implementacji klasy (składowe i zaprzyjaźnione), funkcje implementacji klas pochodnych (składowe i zaprzyjaźnione klas pochodnych) oraz inne funkcje. Można to przedstawić graficznie:



Znaczenie specyfikatorów dostępu jest dla wszystkich nazw takie samo, tzn. to, do czego odnosi się nazwa, nie jest ważne pod tym względem. Oznacza to, że można tworzyć prywatne funkcje, typy, stałe itd. oraz prywatne zmienne składowe. Na przykład wydajna i nieintruzyjna klasa listy często musi zawierać struktury danych do zapanowania nad elementami. Lista jest **nieintruzyjna**, gdy nie wymaga modyfikowania swoich elementów (np. poprzez wymóg, aby typy elementów zawierały pole łącza). Informacje i struktury danych służące do organizacji takiej listy mogą być prywatne:

```

template<typename T>
class List {
public:
    void insert(T);
    T get();
    //...
private:
    struct Link { T val; Link* next; };

    struct Chunk {
        enum { chunk_size=15};
        Link v[chunk_size];
        Chunk* next;
    };

    Chunk* allocated;
    Link* free;
    Link* get_free();
    Link* head;
};
  
```

Definicje funkcji publicznych są proste:

```
template<typename T>
void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
    lnk->next = head;
    head = lnk;
}

template<typename T>
T List<T>::get()
{
    if (head == nullptr)
        throw Underflow{}; // Underflow to klasa wyjątków

    Link* p = head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}
```

Jak zwykła definicja funkcji pomocniczych (tutaj prywatnych) jest trochę trudniejsza:

```
template<typename T>
typename List<T>::Link* List<T>::get_free()
{
    if (free == nullptr) {
        //... alokacja nowego fragmentu pamięci i umieszczenie łącza w liście free...
    }
    Link* p = free;
    free = free->next;
    return p;
}
```

Do zakresu `List<T>` wchodzimy poprzez napisanie `List<T>::` w definicji funkcji składowej. Jednak jako że typ zwrotny funkcji `get_free()` jest określony przed miejscem pojawienia się nazwy `List<T>::get_free()`, musi zostać użyta pełna nazwa `List<T>::Link` zamiast skrótu `Link`. Alternatywą jest użycie notacji przyrostkowej dla typów zwrotnych (12.1.4):

```
template<typename T>
auto List<T>::get_free() -> Link*
{
    ...
}
```

Funkcje nie będące składowymi (z wyjątkiem zaprzyjaźnionych) nie mają takiego dostępu:

```
template<typename T>
void would_be_meddler(List<T>* p)
{
    List<T>::Link* q=0; // błąd: List<T>::Link jest prywatna
    ...
    q = p->free; // błąd: List<T>::free jest prywatna
    ...
}
```

```

if (List<T>::Chunk::chunk_size > 31) { // blqđ: List<T>::Chunk::chunk_size jest prywatna
    ...
}
}

```

Składowe klasy są domyślnie prywatne. Natomiast składowe struktury są domyślnie publiczne (16.2.4).

Oczywistą alternatywą dla używania typu składowego jest umieszczenie typu w otaczającej przestrzeni nazw. Na przykład:

```

template<typename T>
struct Link2 {
    T val;
    Link2* next;
};

template<typename T>
class List {
private:
    Link2<T>* free;
    ...
};

```

Struktura Link jest parametryzowana niejawnie parametrem T z List<T>. W przypadku struktury Link2 musielibyśmy to zrobić jawnie.

Jeśli typ składowy nie zależy od wszystkich parametrów klasy szablonowej, lepsza może być wersja nieskładowa — zobacz 23.4.6.3.

Jeśli zagnieżdżanie jest niepożądane, ale zagnieżdzona klasa nie jest sama w sobie specjalnie przydatna, dobrym pomysłem może być zadeklarowanie jej jako zaprzyjaźnionej (19.4.2) z tą, która miała ją zawierać:

```

template<typename T> class List;

template<typename T>
class Link3 {
    friend class List<T>; // tylko List<T> ma dostęp do Link3<T>
    T val;
    Link3* next;
};

template<typename T>
class List {
private:
    Link3<T>* free;
    ...
};

```

Kompilator może zmienić kolejność części klasy oznaczonych specyfikatorami dostępu (8.2.6). Na przykład:

```

class S {
public:
    int m1;
public:
    int m2;
};

```

Kompilator może wstawić `m2` przed `m1` w układzie obiektu klasy `S`. To może zaskoczyć programistę i jest całkowicie zależne od implementacji, dlatego wielu specyfikatorów dostępu do składowych należy używać tylko wtedy, gdy ma się ku temu dobry powód.

20.5.1. Składowe chronione

Przy projektowaniu hierarchii klas czasami tworzy się funkcje przeznaczone do użytku przez implementatorów klas pochodnych, nie zaś zwykłych użytkowników. Przykładowo możemy napisać (wydajną) funkcję niekontrolowanego dostępu dla implementatorów klas pochodnych i (bezpieczną) kontrolowaną funkcję dostępową dla pozostałych użytkowników. Pomyśl ten można zrealizować, deklarując niekontrolowaną wersję jako chronioną (`protected`). Na przykład:

```
class Buffer {
public:
    char& operator[](int i); // dostęp kontrolowany
    ...
protected:
    char& access(int i); // dostęp niekontrolowany
    ...
};

class Circular_buffer : public Buffer {
public:
    void reallocate(char* p, int s); // zmienia lokalizację i rozmiar
    ...
};

void Circular_buffer::reallocate(char* p, int s)// zmienia lokalizację i rozmiar
{
    ...
    for (int i=0; i!=old_sz; ++i)
        p[i] = access(i); // bez niepotrzebnego sprawdzania
    ...
}

void f(Buffer& b)
{
    b[3] = 'b'; // OK (kontrolowane)
    b.access(3) = 'c'; // bląd: funkcja Buffer::access() jest chroniona
}
```

Jeszcze jeden przykład znajduje się w sekcji 21.3.5.2 (`Window_with_border`).

Klasa pochodna ma dostęp do składowych chronionych klasy bazowej tylko dla obiektów swojego typu:

```
class Buffer {
protected:
    char a[128];
    ...
};

class Linked_buffer : public Buffer {
    ...
};

class Circular_buffer : public Buffer {
```

```

//...
void f(Linked_buffer* p)
{
    a[0] = 0;      // OK: dostęp do składowej chronionej klasy Circular_buffer
    p->a[0] = 0; // błąd: dostęp do chronionej składowej innego typu
}

```

To chroni nas przed subtelnymi błędami, które mogą powstawać, gdy jedna klasa pochodna uszkodzi dane należące do innej klasy pochodnej.

20.5.1.1. Używanie składowych chronionych

Prosty model prywatnych i publicznych zmiennych składowych wspomaga techniki tworzenia typów konkretnych (16.3). Ale gdy tworzone są klasy pochodne, wyróżniamy dwa rodzaje użytkowników klas: właśnie klasy pochodne i „ogólne społeczeństwo”. Składowe i funkcje zaprzyjaźnione stanowiące implementację klasy operują na jej obiektach w imieniu tych użytkowników. Model prywatnych i publicznych składowych umożliwia rozróżnienie implementatorów i ogólnych użytkowników, ale nie pomaga w korzystaniu z klas pochodnych.

Składowe chronione są znacznie bardziej narażone na nadużycia od składowych prywatnych. Zwłaszcza błędem projektowym jest deklaracja jako chronionych danych składowych. Umieszczenie większej ilości danych we wspólnej klasie, aby wszystkie klasy pochodne miały do nich dostęp, jest proszeniem o ich zniszczenie. Co gorsza, danych chronionych podobnie jak publicznych nie można łatwo restrukturyzować, bo nie da się znaleźć wszystkich miejsc, w których są używane. W ten sposób dane chronione stają się ciężarem.

Na szczęście nie trzeba używać danych chronionych. Domyślnie w klasach składowe są prywatne i w większości przypadków nie trzeba tego zmieniać. Z doświadczenia wiem, że zawsze jest inne rozwiązanie niż umieszczenie dużej ilości informacji we wspólnej klasie bazowej, aby mogły z nich korzystać klasy pochodne.

Ale powyższe rozważania nie dotyczą *funkcji* chronionych, które tworzy się w celu definiowania operacji do użytku w klasach pochodnych. Przykładem takiej funkcji jest `Ival_slider` z sekcji 21.2.2. Gdyby w tamtym przykładzie klasa implementacyjna była prywatna, dalsza derywacja byłaby niemożliwa. Z drugiej strony, publiczne udostępnianie szczegółów implementacyjnych w klasie bazowej to zaproszenie dla błędów i pomyłek.

20.5.2. Dostęp do klas bazowych

Klasę bazową, podobnie jak składową, można zadeklarować jako prywatną (`private`), chronioną (`protected`) lub publiczną (`public`). Na przykład:

```

class X : public B { /*...*/};
class Y : protected B { /*...*/};
class Z : private B { /*...*/};

```

Każdy specyfikator dostępu służy w projekcie klasy do czegoś innego:

- Derywacja publiczna sprawia, że klasa pochodna staje się podtypem swojej klasy bazowej. Na przykład `X` jest rodzajem `B`. Jest to najczęściej spotykana forma derywacji.
- Bazy prywatne są najczęściej używane do definiowania klas poprzez ograniczenie interfejsu do klasy bazowej w celu zapewnienia silniejszej gwarancji. Na przykład `B` jest szczegółem implementacyjnym `Z`. Dobrym przykładem jest szablon wektora wskaźników wzbogacający swoją bazę `Vector<void*>` o sprawdzanie typów, opisany w podrozdziale 25.3.

- Bazy chronione są przydatne w hierarchiach klas, w których dalsza derywacja jest oczywista. Derywacja chroniona, podobnie jak prywatna, jest wykorzystywana do reprezentowania szczegółów implementacyjnych. Dobrym przykładem jest `Ival_slider` z sekcji 21.2.2.

Specyfikator dostępu klasy bazowej można opuścić i wówczas domyślnie stosowany jest specyfikator `private` (w przypadku klas) lub `public` (w przypadku struktur). Na przykład:

```
class XX : B { /*...*/}; // B jest bazą prywatną
struct YY : B { /*...*/}; // B jest bazą publiczną
```

Wielu programistom wydaje się, że baza powinna być publiczna (aby wyrażać relację podtypu) i brak specyfikatora dostępu dla bazy może ich zaskoczyć w przypadku klas, ale raczej nie, jeśli chodzi o struktury.

Specyfikator dostępu klasy bazowej kontroluje dostęp do jej składowych oraz konwersję wskaźników i referencji z typu klasy pochodnej na typ klasy bazowej. Przyjmijmy na przykład, że klasa `D` jest pochodną klasy `B`:

- Jeżeli `B` jest bazą prywatną, jej publiczne i chronione składowe mogą być używane tylko przez funkcje składowe i zaprzyjaźnione klasy `D`. Tylko funkcje zaprzyjaźnione i składowe `D` mogą konwertować `D*` na `B*`.
- Jeżeli `B` jest bazą chronioną, jej publiczne i chronione składowe mogą być używane tylko przez funkcje składowe i zaprzyjaźnione klasy `D` oraz przez funkcje składowe i zaprzyjaźnione klas pochodnych klasy `D`. Tylko funkcje zaprzyjaźnione i składowe klasy `D` oraz przyjaciele i składowe klas pochodnych klasy `D` mogą konwertować `D*` na `B*`.
- Jeżeli `B` jest bazą publiczną, jej publicznych składowych mogą używać wszystkie funkcje. Ponadto jej składowych chronionych mogą używać składowe i przyjaciele klasy `D` oraz składowe i przyjaciele klas pochodnych klasy `D`. Każda funkcja może dokonać konwersji `D*` na `B*`.

Jest to zasadniczo powtórzenie zasad dotyczących dostępu do składowych (20.5). Projektując klasę, poziom dostępu do bazy wybiera się tak samo jak dla składowych. Zobacz na przykład `Ival_slider` w sekcji 21.2.2.

20.5.2.1. Wielodziedziczenie a kontrola dostępu

Jeśli do nazwy klasy bazowej można dotrzeć różnymi ścieżkami w strukturze wielodziedziczenia (21.3), to nazwa ta jest dostępna, gdy można uzyskać do niej dostęp poprzez którykolwiek z tych ścieżek. Na przykład:

```
struct B {
    int m;
    static int sm;
    ...
};

class D1 : public virtual B { /*...*/};
class D2 : public virtual B { /*...*/};
class D12 : public D1, private D2 { /*...*/};

D12* pd = new D12;
B* pb = pd;           // OK: dostępna przez D1
int i1 = pd->m; // OK: dostępna przez D1
```

Nawet jeśli do obiektu można dotrzeć wieloma ścieżkami, to i tak da się do niego odnosić w sposób jednoznaczny. Na przykład:

```
class X1 : public B { /*...*/};
class X2 : public B { /*...*/};
class XX : public X1, public X2 { /*...*/};

XX* pxx = new XX;
int i1 = pxx->m; // błąd niejednoznaczności: XX::X1::B::m czy XX::X2::B::m?
int i2 = pxx->sm; // OK: jest tylko jedna składowa B::sm w obiekcie klasy XX (sm jest składową statyczną)
```

20.5.3. Deklaracje using i kontrola dostępu

Deklaracji `using` (14.2.2, 20.3.5) nie można wykorzystywać w celu zdobycia dostępu do dodatkowych informacji. Jest to tylko technika umożliwiająca uzyskiwanie dostępu do już dostępnych informacji w wygodniejszy sposób. Z drugiej strony, gdy jest dostęp, można go przypisać innym użytkownikom. Na przykład:

```
class B {
private:
    int a;
protected:
    int b;
public:
    int c;
};

class D : public B {
public:
    using B::a; // błąd: składowa B::a jest prywatna
    using B::b; // sprawia, że składowa B::b będzie publicznie dostępna przez D
};
```

Używając deklaracji `using` w połączeniu z derywacją prywatną lub chronioną, można określić interfejsy do niektórych, ale nie wszystkich elementów zawartości klasy. Na przykład:

```
class BB : private B { // udostępnia B::b i B::c, ale nie B::a
public:
    using B::b;
    using B::c;
};
```

Zobacz również sekcję 20.3.5.

20.6. Wskaźniki do składowych

Wskaźnik do składowej jest rodzajem offsetowej konstrukcji umożliwiającej pośrednie odwoływanie się do składowej klasy. Operatory `->*` i `.*` są prawdopodobnie najbardziej specjalistycznymi i najrzadziej używanymi operatorami w języku C++. Przy użyciu operatora `->` możemy uzyskać dostęp do składowej klasy `m`, wymieniając jej nazwę, np. `p->m`. Przy użyciu operatora `->*` możemy uzyskać dostęp do składowej, której nazwa (konsepcyjnie) jest przechowywana we wskaźniku do składowej (`ptom`): `p->*ptom`. W ten sposób można uzyskiwać dostęp do składowych przy użyciu ich nazw przekazywanych jako argument. W obu przypadkach `p` musi być wskaźnikiem do obiektu odpowiedniej klasy.

Wskaźnika do składowej nie można przypisać do `void*` ani żadnego innego zwykłego wskaźnika. Wskaźnik pusty (np. `nullptr`) można przypisać do wskaźnika do składowej i wówczas oznacza on „brak składowej”.

20.6.1. Wskaźniki do funkcji składowych

Wiele klas udostępnia proste i bardzo ogólne interfejsy przeznaczone do użytku na kilka różnych sposobów. Na przykład w wielu „obiektowych” interfejsach zdefiniowane są zestawy żądań, na które każdy obiekt reprezentowany na ekranie powinien móc zareagować. Ponadto żądania takie w programach mogą być prezentowane bezpośrednio lub pośrednio. Na przykład:

```
class Std_interface {
public:
    virtual void start() = 0;
    virtual void suspend() = 0;
    virtual void resume() = 0;
    virtual void quit() = 0;
    virtual void full_size() = 0;
    virtual void small() = 0;

    virtual ~Std_interface() {}
};
```

Działanie każdej operacji zależy od obiektu, na którym zostanie wywołana. Często między osobą lub programem wysyłającym żądanie a obiektem będącym jego adresatem występuje dodatkowa warstwa oprogramowania. Najlepiej żeby takie pośrednie warstwy nie musiały niczego wiedzieć o poszczególnych operacjach, jak `resume()` czy `full_size()`. Gdyby było inaczej, trzeba by je było zmieniać przy każdej zmianie operacji. A zatem warstwy pośrednie tylko przekazują dane reprezentujące operacje do wywołania od źródła do adresata.

Jednym z prostych sposobów na realizację tego pomysłu jest wysyłanie łańcucha reprezentującego operację, która ma zostać wywołana. Aby na przykład wywołać funkcję `suspend()`, można by było wysłać łańcuch "suspend". Ale ktoś musi utworzyć ten łańcuch, a ktoś inny musi go rozszyfrować, aby dowiedzieć się, której operacji dotyczy — jeśli w ogóle którejś. To często jest żmudne i nieeleganckie rozwiązanie. Dlatego do reprezentacji operacji można użyć liczby całkowitej. Na przykład cyfra 2 mogłaby oznaczać `suspend()`. Ale liczby są wygodne tylko dla maszyn, nie zaś dla ludzi. Poza tym i tak trzeba napisać kod rozszyfrowujący 2 jako reprezentację funkcji `suspend()` i ją wywołujący.

Alternatywnie możemy pośrednio odwoływać się do składowych klas przy użyciu wskaźnika. Zastanówmy się nad klasą `Std_interface`. Jeśli chcę wywołać funkcję `suspend()` na jakimś obiekcie, nie wymieniając jej nazwy bezpośrednio, to potrzebuję wskaźnika odnoszącego się do składowej `Std_interface::suspend()`. Dodatkowo potrzebuję wskaźnika lub referencji do obiektu. Spójrz na poniższy prosty przykład:

```
using Pstd_mem = void (Std_interface::*)(); // typ wskaźnik do składowej

void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend; // wskaźnik do suspend()
    p->suspend(); // bezpośrednie wywołanie
    p->*s(); // wywołanie poprzez wskaźnik do składowej
}
```

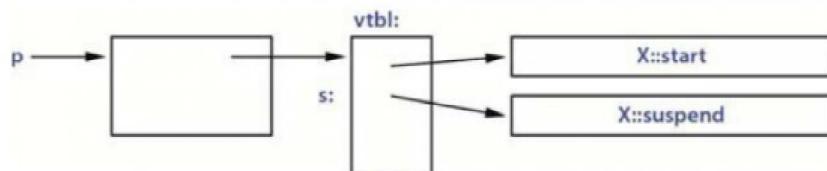
Wskaźnik do składowej można uzyskać poprzez zastosowanie operatora adresowania & do pełnej nazwy składowej, np. `&Std_interface::suspend`. Zmienną typu „wskaźnik do składowej klasy X” deklaruje się przy użyciu deklatora w postaci `X::*`.

Typowe jest stosowanie aliasu w celu zrekompensowania nieczytelności składni tego deklatora w stylu języka C. Mimo to zwrócić uwagę, że deklator `X::*` dokładnie odpowiada tradycyjnemu deklatorowi `*`.

Wskaźnika do zmiennej `m` można też użyć w kombinacji z obiektem, bo pozwalają na to operatory `->*` i `.*`. Na przykład `p->*m` wiąże `m` z obiektem wskazywanym przez `p`, a `obj.*m` wiąże `m` z obiektem `obj`. Wyniku można używać zgodnie z typem `m`. Nie można zapisać wyniku operacji `a->*` lub `a.*` do późniejszego użytku.

Oczywiście gdybyśmy wiedzieli, którą składową chcemy wywołać, wywołalibyśmy ją bezpośrednio, zamiast bawić się wskaźnikami do składowych. Wskaźniki do funkcji składowych, podobnie jak wskaźniki do zwykłych funkcji, służą do wywoływania funkcji bez znajomości ich nazw. Jednak wskaźnik do funkcji składowej nie wskazuje fragmentu pamięci, jak jest w przypadku wskaźnika do zmiennej lub funkcji. Wskaźnik ten bardziej przypomina określenie miejsca w strukturze lub indeks w tablicy, ale implementacja oczywiście uwzględnia różnice między danymi składowymi, funkcjami wirtualnymi, funkcjami niewirtualnymi itd. Kiedy wskaźnik do składowej zostanie użyty w połączeniu ze wskaźnikiem do obiektu odpowiedniego typu, wynikiem jest coś, co identyfikuje konkretną składową konkretnego obiektu.

Wywołanie `p->*s()` można przedstawić graficznie w następujący sposób:



Jako że wskaźnik do wirtualnej składowej (`s` w tym przykładzie) jest rodzajem offsetu, nie jest zależny od lokalizacji obiektu w pamięci. W związku z tym wskaźnik taki można przekazywać między różnymi przestrzeniami adresowymi, pod warunkiem że w każdej używany jest obiekt o takim samym układzie. Podobnie jak wskaźniki do zwykłych funkcji, również wskaźniki do niewirtualnych funkcji składowych nie mogą być przekazywane między różnymi przestrzeniami adresowymi.

Funkcja wywoływana przez wskaźnik do funkcji może być wirtualna. Gdy na przykład wywołujemy funkcję `suspend()` poprzez wskaźnik, otrzymujemy wywołanie funkcji `suspend()` dla obiektu, do którego ten wskaźnik do funkcji składowej jest zastosowany. Jest to podstawowa właściwość wskaźników do funkcji składowych.

Pisząc interpreter, możemy używać wskaźników do składowych w celu wywoływania funkcji prezentowanych jako łańcuchy:

```

map<string,Std_interface*> variable;
map<string,Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var]->*operation[oper])(); // var.oper()
}
  
```

Statyczna składowa nie jest związana z konkretnym obiektem, a więc wskaźnik do takiej składowej jest zwykłym wskaźnikiem. Na przykład:

```
class Task {
    //...
    static void schedule();
};

void (*p)() = &Task::schedule;           // OK
void (Task::*pm)() = &Task::schedule;    // błąd: zwykły wskaźnik przypisany
                                         // do wskaźnika do składowej
```

Opis wskaźników do danych składowych znajduje się w sekcji 20.6.2.

20.6.2. Wskaźniki do danych składowych

Oczywiście pojęcie wskaźnika do składowej dotyczy danych składowych oraz funkcji składowych z argumentami i typami zwrotnymi. Na przykład:

```
struct C {
    const char* val;
    int i;

    void print(int x) { cout << val << x << '\n'; }
    int f1(int);
    void f2();
    C(const char* v) { val = v; }
};

using Pmfi = void (C::*)(int); // wskaźnik do funkcji składowej struktury C przyjmującej int
using Pm = const char*C::*;

void f(C& z1, C& z2)
{
    C* p = &z2;
    Pmfi pf = &C::print;
    Pm pm = &C::val;
    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);
    pf = &C::f1; // błąd: inny typ zwrotny
    pf = &C::f2; // błąd: inny typ argumentu
    pm = &C::i; // błąd: różne typy
    pm = pf;    // błąd: różne typy
}
```

Typ wskaźnika do funkcji jest sprawdzany tak jak każdy inny typ.

20.6.3. Składowe bazy i klasy pochodnej

Klasa pochodna zawiera przynajmniej składowe odziedziczone po klasach bazowych. Często ma ich więcej. To oznacza, że można bezpiecznie przypisać wskaźnik do składowej klasy bazowej do wskaźnika do składowej klasy pochodnej, ale nie odwrotnie. Właściwość tę często nazywa się **kontrawariancją**. Na przykład:

```
class Text : public Std_interface {  
public:  
    void start();  
    void suspend();  
    //...  
    virtual void print();  
private:  
    vector s;  
};  
  
void (Std_interface::*pmi)() = &Text::print; // błęd  
void (Text::*pmt)() = &Std_interface::start; // OK
```

Wydaje się, że reguła kontrawariancji stoi w sprzeczności z regułą głoszącą, że można przypisać wskaźnik do klasy pochodnej do wskaźnika do jej klasy bazowej. Jednak w istocie obie te reguły mają za zadanie pomóc w zachowaniu podstawowej gwarancji, że wskaźnik nie może wskazywać obiektu, który nie ma przynajmniej wszystkich właściwości, jakie ten wskaźnik sugeruje. W tym przypadku `Std_interface*` można zastosować do dowolnego obiektu klasy `Std_interface` i większość z takich obiektów z pewnością nie jest typu `Text`. W związku z tym nie mają one składowej `Text::print`, przy użyciu której próbowaliśmy zainicjować `pmi`. Odmawiając inicjacji, kompilator ratuje nas przed błędem w czasie działania programu.

20.7. Rady

1. Unikaj pól typów — 20.3.1.
2. Uzyskuj dostęp do obiektów polimorficznych przez wskaźniki i referencje — 20.3.2.
3. Używaj klas abstrakcyjnych, aby projektować programy na bazie klarownych interfejsów — 20.4.
4. Używaj słowa kluczowego `override` do oznaczania przesłaniania w dużych hierarchiach klas — 20.3.4.1.
5. Słowa `final` używaj oszczędnie — 20.3.4.2.
6. Używaj klas abstrakcyjnych do określania interfejsów — 20.4.
7. Używaj klas abstrakcyjnych, aby oddzielić szczegóły implementacji od interfejsów — 20.4.
8. Klasa zawierająca funkcję wirtualną powinna zawierać też destruktor wirtualny — 20.4.
9. W klasie abstrakcyjnej zwykle nie jest potrzebny konstruktor — 20.4.
10. Najlepiej żeby szczegóły implementacyjne były prywatne — 20.5.
11. Najlepiej żeby interfejsy były publiczne — 20.5.
12. Składowych chronionych używaj ostrożnie i tylko wtedy, gdy naprawdę są potrzebne — 20.5.1.1.
13. Nie deklaruj jako chronionych danych składowych — 20.5.1.1.