

Derywacja klas

Nie mnóż bytów ponad potrzebę
— William Ockham

- Wprowadzenie
- Klasy pochodne
 - Funkcje składowe; Konstruktory i destruktory
- Hierarchie klas
 - Pola typów; Funkcje wirtualne; Bezpośrednia kwalifikacja; Kontrola przesłaniania;
Używanie składowych klasy bazowej; Rozluźnienie zasady dotyczącej typów zwrotnych
- Klasy abstrakcyjne
- Kontrola dostępu
 - Składowe chronione; Dostęp do klas bazowych; Deklaracje using i kontrola dostępu
- Wskaźniki do składowych
 - Wskaźniki do funkcji składowych; Wskaźniki do zmiennych składowych; Składowe bazy
i klasy pochodnej
- Rady

20.1. Wprowadzenie

Klasy i hierarchie klas w języku C++ zostały zapożyczone z języka Simula. Ponadto zaczerpnięto też pomysł, aby klasy służyły do modelowania koncepcji ze świata programisty i aplikacji. W języku C++ dostępne są wszystkie konstrukcje potrzebne do bezpośredniego stosowania tych praktyk projektowych. Ich używanie pozwala na efektywne wykorzystanie możliwości tego języka. Jeśli ktoś korzysta z nich tylko jak z notacyjnych udogodnień pozwalających stosować tradycyjne techniki programowania, to nie wykorzystuje najmocniejszych stron języka C++.

Zadna koncepcja (pomysł, pojęcie itd.) nie jest zawieszona w przeszłości, tylko współistnieje z innymi koncepcjami i z relacją z nimi czerpie swoją siłę. Spróbuj na przykład wyjaśnić, czym jest samochód. Aby to zrobić, musisz wprowadzić pojęcia koła, silnika, kierowcy, pieszego, ciężarówki, ambulansu, szosy, oleju, mandatu za przekroczenie prędkości, motelu itd. Jako że posługujemy się klasami, powstaje pytanie, jak reprezentować relacje między pojęciami. Ale w języku programowania nie można bezpośrednio wyrażać dowolnych relacji. A gdyby nawet można było, to i tak byśmy nie chcieli tego robić. Jeśli klasa ma być przydatna, musi być zdefiniowana ścisłe i bardziej precyzyjnie niż przedmioty, które spotykamy w realnym życiu.

Pojęcie klasy pochodnej i związane z nim konstrukcje językowe służą do wyrażania relacji hierarchicznych między klasami, tzn. wyrażania wspólnych cech różnych klas. Na przykład pojęcia koła i trójkąta łączy to, że są kształtami, a więc bycie kształtem to ich wspólna cecha. W związku z tym możemy zdefiniować klasy Trójkąt i Koło, których część wspólną będzie reprezentować klasa Kształt. W takim przypadku wspólna klasa Kształt nazywa się **klasą bazową** lub **nadkąską**, a klasy z niej derywowane, Koło i Trójkąt, to **klasy pochodne** lub **podklasty**. Gdyby klasy reprezentujące koło i trójkąt zdefiniowano bez posłużenia się pojęciem kształtu, utraccono by coś ważnego. W tym rozdziale opisuję właśnie znaczenie tej prostej koncepcji, która stanowi podstawę techniki powszechnie nazywanej **programowaniem obiektowym**. Język umożliwia budowanie nowych klas z istniejących:

- **Dziedziczenie implementacji** — pozwala oszczędzić na pracy implementacyjnej poprzez wykorzystanie części wspólnej w postaci klasy bazowej.
- **Dziedziczenie interfejsu** — umożliwia naprzemienne używanie klas pochodnych poprzez interfejs dostarczony przez wspólną klasę bazową.

Dziedziczenie interfejsu często nazywa się **polimorfizmem czasu działania** (albo **polimorfizmem dynamicznym**). Dla odróżnienia posługiwania się klasami nie powiązanymi przez dziedziczenie określone przez szablony (3.4, rozdział 23.) często nazywa się **polimorfizmem czasu kompilacji** (albo **polimorfizmem statycznym**).

Opis hierarchii klas jest podzielony na trzy rozdziały:

- **Derywacja klas** (rozdział 20.) — podstawowe narzędzia językowe służące do programowania obiektowego. W rozdziale tym opisane są klasy bazowe i pochodne, funkcje wirtualne oraz sposoby kontroli dostępu.
- **Hierarchie klas** (rozdział 21.) — w tym rozdziale opisane są sposoby organizacji kodu na podstawie hierarchii klas złożonych z klas bazowych i pochodnych. Większość treści dotyczy technik programowania, ale można też znaleźć techniczny opis wielodziedziczenia (klas mających więcej niż jedną klasę bazową).
- **Informacje o typach w czasie działania programu** (rozdział 22.) — opis technik poruszania się w obrębie hierarchii klas. Przedstawione są operacje konwersji typów `dynamic_cast` i `static_cast` oraz operacja `typeid` do sprawdzania typów obiektów na podstawie jednej z ich klas bazowych.

Krótkie wprowadzenie do podstawowych zasad hierarchicznej organizacji typów znajduje się w rozdziale 3. — klasy bazowe i pochodne (3.2.2) oraz funkcje wirtualne (3.2.3). Natomiast w tym i kolejnych dwóch rozdziałach znajduje się bardziej szczegółowy opis tych podstawowych technik oraz związanych z nimi metod programowania i projektowania.

20.2. Klasy pochodne

Wyobraź sobie, że masz napisać program do zarządzania kadrami firmy. Struktura danych takiego programu mogłaby być następująca:

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    //...
};
```

Następnie możemy zdefiniować menedżera:

```
struct Manager {
    Employee emp;           // rekord pracownika, którym jest menedżer
    list<Employee*> group; // podwładni
    short level;
    ...
};
```

Menedżer jest także pracownikiem. Jego dane pracownicze są przechowywane w składowej `emp` obiektu typu `Manager`. Dla człowieka — zwłaszcza uważnego czytelnika kodu — jest to oczywiste, ale kompilator ani inne narzędzie nie mogą wiedzieć, że menedżer (`Manager`) to także pracownik (`Employee`). `Manager*` to nie `Employee*`, przez co nie można użyć jednego typu w miejscu, w którym potrzebny jest drugi. W szczególności nie można umieścić menedżera na liście pracowników bez napisania specjalnego kodu. Możemy zastosować jawną konwersję typu `Manager*` albo umieścić adres składowej `emp` na liście pracowników. Jednak oba te rozwiązania są nieeleganckie i dość niejasne. Najlepiej byłoby wprost zaznaczyć, że menedżer jest pracownikiem mającym kilka dodatkowych cech:

```
struct Manager : public Employee {
    list<Employee*> group;
    short level;
    ...
};
```

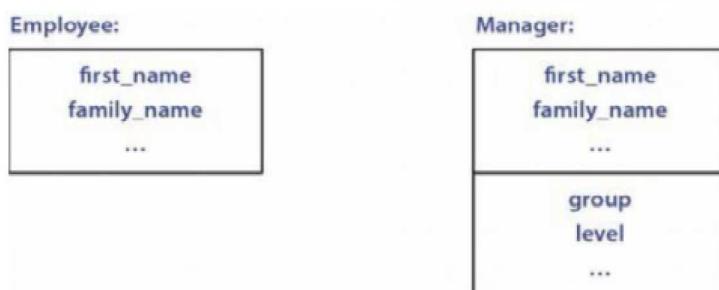
Klasa `Manager` jest *pochodną* klasy `Employee`, a klasa `Employee` jest *klasą bazową* klasy `Manager`. Klasa `Manager` zawiera wszystkie składowe klasy `Employee` (`first_name`, `department` itd.) oraz dodatkowo kilka własnych składowych (`group`, `level` itd.).

Derywację często oznacza się strzałką prowadzącą od klasy pochodnej do bazowej, co oznacza, że klasa pochodna odnosi się do swojej bazy (a nie odwrotnie):



Mówiąc też, że klasa pochodna dziedziczy właściwości klasy bazowej i dlatego relację tę nazywa się także **dziedziczeniem**. Klasa bazowa jest czasami nazywana **nadklassą**, a klasa pochodna **podklassą**. Jednak nazwy te są mylące dla tych, którzy zauważają, że dane znajdujące się w obiekcie klasy pochodnej są nadzbiorem danych znajdujących się w obiekcie klasy bazowej. Klasa pochodna jest z reguły większa (i nigdy nie mniejsza) od klasy bazowej, bo zawiera więcej danych i funkcji.

W popularnej i wydajnej implementacji pojęcia klasy pochodnej obiekt tej klasy reprezentowany jest jako obiekt klasy nadzującej zawierający dodatkowe informacje na końcu. Na przykład:



Derywacja klas nie powoduje nadmiernego zwiększenia zużycia pamięci. Zajmowane jest tylko tyle miejsca, ile potrzeba do przechowywania składowych.

Derywacja klasy `Manager` z klasy `Employee` sprawia, że klasa `Manager` jest podtypem klasy `Employee`, a więc jej obiekt może być użyty wszędzie tam, gdzie można użyć obiektu typu `Employee`. Na przykład teraz bez trudu możemy utworzyć listę pracowników, wśród których znajdują się menedżerowie:

```
void f(Manager m1, Employee e1)
{
    list<Employee*> elist {&m1,&e1};
    //...
}
```

Menedżer jest również pracownikiem, a więc `Manager*` można używać jako `Employee*`. Podobnie `Manager&` można używać jako `Employee&`. Z drugiej strony, pracownik nie musi być menedżerem, a więc `Employee*` nie można użyć jako `Manager*`. Ogólnie rzecz biorąc, jeśli klasa Pochodna ma publiczną klasę bazową (20.5) Baza, to `Pochodna*` można przypisać do zmiennej typu `Baza*` bez stosowania jawnnej konwersji typów. Natomiast konwersja w drugą stronę, z `Baza*` na `Pochodna*`, musi być jawną. Na przykład:

```
void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;           // OK: każdy menedżer jest pracownikiem
    Manager* pm = &ee;            // błęd: nie każdy pracownik jest menedżerem

    pm->level = 2;               // katastrofa: ee nie ma składowej level

    pm = static_cast<Manager*>(pe); // pomyłkowe rozwiązanie: działa, bo pe wskazuje
                                    // obiekt mm klasy Manager

    pm->level = 2;               // w porządku: pm wskazuje obiekt mm klasy Manager mający składową level
}
```

Innymi słowy: obiekt klasy pochodnej można traktować jak obiekt klasy bazowej, jeśli posługuje się nim poprzez wskaźniki i referencje. Przeciwna sytuacja jest jednak niedozwolona. Opis operacji `static_cast` i `dynamic_cast` znajduje się w podrozdziale 22.2.

Użycie klasy jako bazy jest równoznaczne ze zdefiniowaniem (nienazwanego) obiektu tej klasy. Aby więc klasy można było używać jako bazowej, musi ona być zdefiniowana (8.2.2):

```
class Employee;           // sama deklaracja, bez definicji

class Manager : public Employee { // błęd: klasa Employee nie jest zdefiniowana
    //...
};
```

20.2.1. Funkcje składowe

Proste struktury danych, takie jak klasy `Employee` i `Manager`, tak naprawdę nie są zbyt interesujące i rzadko znajdują zastosowanie. Musimy napisać porządnego typu z odpowiednim zestawem operacji i jednocześnie uniknąć związania się ze szczegółami jakiejkolwiek konkretnej reprezentacji. Na przykład:

```
class Employee {
public:
    void print() const;
    string full_name() const { return first_name + ' ' + middle_initial + ' ' + family_name; }
    ...
private:
    string first_name, family_name;
    char middle_initial;
    ...
};

class Manager : public Employee {
public:
    void print() const;
    ...
};
```

Składowa klasy pochodnej może używać publicznych — i chronionych (20.5) — składowych klasy bazowej, tak jakby były w niej zadeklarowane. Na przykład:

```
void Manager::print() const
{
    cout << "Imię i nazwisko: " << full_name() << '\n';
    ...
}
```

Nie ma natomiast klasa pochodna dostępu do prywatnych składowych klasy bazowej:

```
void Manager::print() const
{
    cout << " Nazwisko: " << family_name << '\n'; // blq!
    ...
}
```

Ta wersja funkcji `Manager::print()` nie da się skompilować, bo składowa `family_name` jest niedostępna.

Niektórzy są tym zaskoczeni, ale pomyślmy, co by było, gdyby funkcja składowa klasy pochodnej miała dostęp do prywatnych składowych klasy bazowej. Tworzenie prywatnych składowych straciłoby wtedy sens, bo programista mógłby uzyskać dostęp do prywatnej części klasy, tworząc jej podkласę. Ponadto nie dałoby się już znaleźć wszystkich przypadków użycia prywatnej nazwy klasy, patrząc tylko na funkcje zadeklarowane jako jej składowe i przyjaciele. Konieczne byłoby przeszukiwanie całego kodu źródłowego programu w celu znalezienia klas pochodnych, przejrzenie wszystkich ich funkcji itd. W najlepszym przypadku byłoby to niepraktyczne i żmudne. W uzasadnionych przypadkach można używać składowych chronionych zamiast prywatnych (20.5).

Zazwyczaj najlepszym rozwiązaniem jest używanie w klasie pochodnej tylko publicznych składowych klasy bazowej. Na przykład:

```
void Manager::print() const
{
    Employee::print(); // drukuje informacje ogólnopracownicze
    cout << level;    // drukuje informacje specyficzne dla menedżera
    ...
}
```

Zwróć uwagę, że konieczne jest użycie operatora `::`, ponieważ funkcja `print` została w klasie `Manager` przeddefiniowana. Taki sposób używania nazw w klasach jest typowy. Nieuważny programista mógłby napisać coś takiego:

```
void Manager::print() const
{
    print(); // ups!
    // druk informacji specyficznych dla menedżera
}
```

Ten kod zawiera wywołanie rekurencyjne, które doprowadzi do awarii programu.

20.2.2. Konstruktory i destruktory

Konstruktory i destruktory jak zawsze są bardzo istotne:

- Obiekty są tworzone od podstaw (baza przed składową i składowa przed pochodną) i usuwane od góry (pochodna przed składową i składowa przed bazą) — 17.2.3.
- Każda klasa może inicjować swoje składowe i bazy (ale nie bezpośrednio składowe lub bazy swoich baz) — 17.4.1.
- Destruktory w hierarchii zwykle muszą być wirtualne — 17.2.5.
- Konstruktorów kopiujących klas należących do hierarchii należy używać ostrożnie (jeśli w ogóle), aby uniknąć pocięcia — 17.5.1.4.
- Rozwiązywanie wywołania funkcji wirtualnej, operacji `dynamic_cast` lub funkcji `typeid()` w konstruktorze albo destruktorze odzwierciedla etap konstrukcji i destrukcji (a nie typ obiektu, który ma zostać dopiero ukończony) — 22.4.

W komputerach określenia typu „od dołu” i „od góry” mogą być bardzo mylące. W tekście programu definicje klas bazowych muszą znajdować się przed definicjami klas pochodnych. W niewielkich przykładach kod klas bazowych znajduje się po prostu nad kodem klas pochodnej na ekranie. Ponadto drzewa z reguły rysujemy do góry korzeniami. Kiedy jednak piszę o tworzeniu obiektów od dołu, mam na myśli rozpoczęcie pracy od podstaw (np. klas bazowych) i budowanie wszystkiego, co zależy od nich (np. klas pochodnych). Budujemy od korzeni (klas bazowych) do liści (klas pochodnych).

20.3. Hierarchie klas

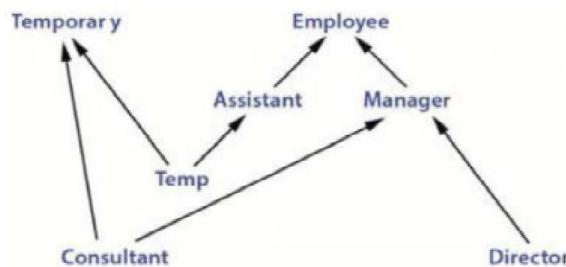
Klasa pochodna może być jednocześnie klasą bazową. Na przykład:

```
class Employee { /* */;
class Manager : public Employee { /* */;
class Director : public Manager { /* */;
```

Takie zbiory powiązanych ze sobą klas nazywa się **hierarchiami klas**. Hierarchie najczęściej są drzewami, ale mogą też reprezentować bardziej ogólne struktury grafowe. Na przykład:

```
class Temporary { /* */;
class Assistant : public Employee { /* */;
class Temp : public Temporary, public Assistant { /* */;
class Consultant : public Temporary, public Manager { /* */;
```

Reprezentacja graficzna:



To oznacza, że w języku C++, jak wyjaśniono w podrozdziale 21.3, można wyrazić skierowany niecykliczny graf klas.

20.3.1. Pola typów

Aby klasy pochodne były czymś więcej niż uproszczeniem składni deklaracji, trzeba rozwiązać następujący problem: jeśli dany jest wskaźnik typu *Baza**, to do którego typu pochodnego należy wskazywany obiekt? Wyróżnia się cztery sposoby rozwiązania tego problemu:

1. Dopilnowanie, aby wskazywane były obiekty tylko jednego typu (3.4, rozdział 23.).
2. Umieszczenie pola typu w klasie bazowej, aby mogło być sprawdzane przez funkcje.
3. Użycie operacji `dynamic_cast` (22.2, 22.6).
4. Użycie funkcji wirtualnych (3.2.3, 20.3.2).

Jeśli nie użyto słowa kluczowego `final` (20.3.4.2), rozwiązanie przedstawione w punkcie 1. wymaga większej wiedzy o typach, niż ma kompilator. Ogólnie rzecz biorąc, lepiej jest nie próbować przechytrzyć systemu typów, ale (zwłaszcza w połączeniu z szablonami) pierwsze rozwiązanie można zastosować w celu implementacji jednolitych kontenerów (np. standardowych typów `vector` i `map`) o niezrównanej wydajności. Rozwiązania 2., 3. i 4. można zastosować do budowy niejednorodnych list, tzn. list (wskaźników do) obiektów różnych typów. Rozwiązanie 3. jest wspomaganym przez język wariantem rozwiązania 2. Rozwiązanie 4. jest specjalnym bezpiecznym typowo wariantem rozwiązania 2. Szczególnie ciekawe i przydatne są kombinacje rozwiązań 1. i 4. Przy ich zastosowaniu prawie zawsze uzyskuje się klarowniejszy kod niż przy zastosowaniu rozwiązań 2. i 3.

Najpierw przestudiujemy możliwości zastosowania prostego rozwiązania z polem typu, aby dowiedzieć się, dlaczego lepiej go unikać. Przykład z menedżerem i pracownikiem można przeddefiniować następująco:

```

struct Employee {
    enum Empl_type { man, empl };
    Empl_type type;

    Employee() : type{empl} { }

    string first_name, family_name;
    char middle_initial;

    Date hiring_date;
    short department;
    //...
};
  
```

```
struct Manager : public Employee {
    Manager() { type = man; }

    list<Employee*> group; // podwładni
    short level;
    ...
};
```

Możemy napisać następującą funkcję drukującą informacje o każdym pracowniku:

```
void print_employee(const Employee* e)
{
    switch (e->type) {
        case Employee::empl:
            cout << e->family_name << '\t' << e->department << '\n';
            ...
            break;
        case Employee::man:
            cout << e->family_name << '\t' << e->department << '\n';
            ...
            const Manager* p = static_cast<const Manager*>(e);
            cout << " poziom " << p->level << '\n';
            ...
            break;
    }
}
```

Listę pracowników można teraz wydrukować tak:

```
void print_list(const list<Employee*>& elist)
{
    for (auto x : elist)
        print_employee(x);
}
```

Ten kod działa, zwłaszcza w niewielkim programie zarządzanym przez jedną osobę. Ale jego wielką wadą jest to, że programista zarządza w nim typami w sposób całkowicie niekontrolowany przez kompilator. Problem ten pogarsza jeszcze fakt, że funkcje takie jak `print_employee()` często wykorzystują wspólnotę cech klas:

```
void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    ...
    if (e->type == Employee::man) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " poziom " << p->level << '\n';
        ...
    }
}
```

Znalezienie wszystkich takich testów pola typu w dużej funkcji posługującej się wieloma klasami pochodnymi jest bardzo trudne. A nawet gdy się to uda, trudno jest zrozumieć, o co w takim kodzie chodzi. Ponadto dodanie nowego rodzaju pracownika sprawia, że trzeba zmienić wszystkie kluczowe funkcje systemu — zawierające testy pola typu. Po wprowadzeniu zmiany programista musi zastanowić się nad każdą funkcją, która mogłaby wymagać testu pola typu.

To oznacza potrzebę uzyskiwania dostępu do krytycznego kodu źródłowego i powstanie narzutu spowodowanego testowaniem kodu. Widoczne użycie jawnej konwersji typów stanowi jednak podpowiedź, że problem da się lepiej rozwiązać.

Inaczej mówiąc, używanie pola typu to podatna na błędy technika, która sprawia wiele problemów w utrzymaniu kodu. Skala problemu rośnie wraz z rozmiarem programu, ponieważ pola typu powodują złamanie zasad modularności i ukrywania danych. Każda funkcja używająca pola typu musi znać reprezentację i inne szczegóły implementacji każdej klasy pochodnej klasy zawierającej to pole typu.

Ponadto wydaje się, że obecność wspólnych danych dostępnych w każdej klasie pochodnej, takich jak pole typu, stanowi zachętę dla programistów, aby dodawać więcej takich danych. Wówczas wspólna baza staje się magazynem rozmaitych „przydatnych informacji”. To z kolei prowadzi do splątania implementacji bazy i klas pochodnych w bardzo niekorzystny sposób. W dużej hierarchii klas dane dostępne (nie prywatne) we wspólnej klasie bazowej stają się „zmiennymi globalnymi” hierarchii. Dla uproszczenia projektu i ułatwienia utrzymania kodu należy rozdzielić koncepcje i wystrzegać się wzajemnych zależności.

20.3.2. Funkcje wirtualne

Funkcje wirtualne umożliwiają rozwiązywanie problemu z polami typów poprzez deklarowanie w klasie bazowej funkcji, które można przedefiniować w każdej klasie pochodnej. W takim przypadku kompilator i konsolidator zapewniają poprawną relację między obiektami i stosowanymi do nich funkcjami. Na przykład:

```
class Employee {
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    ...
private:
    string first_name, family_name;
    short department;
    ...
};
```

Słowo kluczowe `virtual` oznacza, że funkcja `print()` może służyć jako interfejs do funkcji `print()` zdefiniowanej w tej klasie i jej klasach pochodnych. Jeśli funkcja ta jest zdefiniowana w klasach pochodnych, kompilator zawsze wybiera odpowiednią wersję dla danego obiektu klasy `Employee`.

Aby funkcja wirtualna mogła służyć jako interfejs do funkcji zdefiniowanych w klasach pochodnych, typy argumentów tej funkcji w klasie pochodnej muszą być takie same jak typy argumentów w klasie bazowej, a typ zwrotny może różnić się tylko nieznacznie (20.3.6). Wirtualna funkcja składowa jest czasami nazywana **metodą**.

Funkcja wirtualna *musi* być zdefiniowana w klasie, w której znajduje się jej pierwsza deklaracja (chyba że jest to funkcja czysto wirtualna — 20.4). Na przykład:

```
void Employee::print() const
{
    cout << family_name << '\t' << department << '\n';
    ...
}
```

Funkcji wirtualnej można używać nawet wtedy, gdy nie ma ani jednej klasy pochodnej. Ponadto nie ma obowiązku definiowania własnej wersji funkcji wirtualnej w klasie pochodnej, jeśli nie jest ona potrzebna. Tworząc klasę pochodną, po prostu definiuje się odpowiednią funkcję w razie potrzeby. Na przykład:

```
class Manager : public Employee {
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    ...
private:
    list<Employee*> group;
    short level;
    ...
};

void Manager::print() const
{
    Employee::print();
    cout << "poziom " << level << '\n';
    ...
}
```

Funkcja w klasie pochodnej mająca taką samą nazwę i taki sam zestaw typów argumentów jak funkcja wirtualna w klasie bazowej **przesłania** tę funkcję. Ponadto można przesłonić funkcję wirtualną klasy bazowej przy użyciu pochodnego typu zwrotnego (20.3.6).

Nie licząc przypadków bezpośredniego wyboru wersji funkcji wirtualnej (jak w wywołaniu `Employee::print()`), z funkcji przesyłających wybierana jest ta, która najlepiej odpowiada obiekowi, którego dotyczy wywołanie. Mechanizm wyboru funkcji wirtualnych wybiera zawsze tę samą funkcję, niezależnie od tego, której klasy bazowej (interfejsu) użyjemy.

Globalna funkcja `print_employee()` (20.3.1) jest teraz już niepotrzebna, bo jej rolę przejęły funkcje składowe `print()`. Listę pracowników można teraz wydrukować tak:

```
void print_list(const list<Employee*>& s)
{
    for (auto x : s)
        x->print();
}
```

Dane każdego pracownika zostaną wydrukowane zgodnie z jego typem. Na przykład:

```
int main()
{
    Employee e {"Kowalski",1234};
    Manager m {"Kwiatkowski",1234,2};
    print_list({&m,&e});
}
```

Wynik:

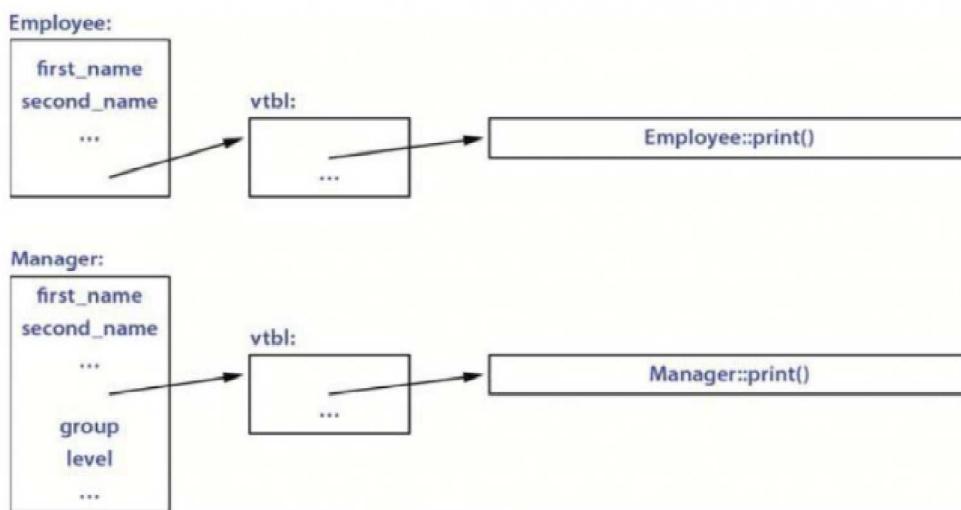
```
Kowalski 1234
poziom 2
Kwiatkowski 1234
```

Ten kod zadziałałby, nawet gdyby funkcja `print_list()` została napisana i skompilowana, zanim byśmy nawet pomyśleli o klasie pochodnej `Manager`! Jest to kluczowa cecha klas. Jeśli są używane poprawnie, stanowią fundament obiektowego projektu programu i zapewniają mu dużą stabilność podczas jego rozwoju.

Sprawienie, że funkcje klasy `Employee` zachowują się „poprawnie” bez względu na to, jaki konkretnie rodzaj obiektu reprezentującego pracownika jest używany, nazywa się **polimorfizmem**. Typ zawierający funkcje wirtualne nazywa się **typem polimorficznym**, a dokładniej mówiąc **typem polimorficznym czasu działania**. Aby skorzystać z polimorfizmu czasu działania w języku C++, wywoływane funkcje muszą być wirtualne, a obiekty muszą być manipulowane poprzez wskaźniki lub referencje. Gdy obiekt jest używany bezpośrednio (nie zaś poprzez wskaźnik lub referencję), kompilator zna jego typ, więc polimorfizm czasu wykonywania nie jest potrzebny.

Domyślnie funkcja przesłaniająca funkcję wirtualną sama staje się wirtualna. W klasie pochodnej można powtórzyć słowo kluczowe `virtual`, ale nie ma takiego obowiązku. Ja nie zalecam tego robić. Jeśli chcesz coś podkreślić, to lepiej użądź słowa kluczowego `override` (20.3.4.1).

Implementacja polimorfizmu wymaga, aby kompilator przechowywał w każdym obiekcie klasy `Employee` pewne informacje o typach, przy użyciu których może wybrać odpowiednią wersję funkcji wirtualnej `print()`. W typowej implementacji zajęta na ten cel przestrzeń jest wystarczająca do zapisania wskaźnika (3.2.3): najczęściej stosowana technika implementacyjna polega na zamianie przez kompilator nazwy funkcji wirtualnej na indeks do tablicy wskaźników do funkcji. Tablica ta nazywa się **tablicą funkcji wirtualnych** (ang. *virtual function table*) albo w skrócie `vtbl`. Tablica taka jest tworzona dla każdej klasy zawierającej funkcje wirtualne. Spójrz na poniższą ilustrację:



Dzięki funkcjom znajdującym się w tablicy `vtbl` obiektu można używać poprawnie nawet mimo nieznajomości przez wywołującego jego rozmiaru ani układu jego danych. Wywołujący musi tylko znać lokalizację tablicy `vtbl` w klasie `Employee` oraz indeks każdej z funkcji wirtualnych. Ten mechanizm wywoływanego funkcji wirtualnych może być prawie tak samo wydajny jak „normalny mechanizm wywoływanego funkcji” (do 25%), a więc nie należy rezygnować z użycia funkcji wirtualnej z powodów wydajnościowych w przypadkach, w których użycie zwykłej funkcji byłoby akceptowalne. Jeśli chodzi o pamięć, to tworzony jest jeden dodatkowy wskaźnik w każdym obiekcie klasy zawierającej funkcję wirtualną oraz jedna tablica `vtbl` dla każdej takiej klasy. Narzut ten występuje tylko w przypadku obiektów klas zawierających funkcje wirtualne. Godzisz się na niego tylko wtedy, gdy chcesz skorzystać z oferowanych przez nie udoskonalień. Gdybyś zastosował opisane wcześniej rozwiązanie z polami typów, to podobną ilość pamięci trzeba było przeznaczyć na te pola.

Wywołanie funkcji wirtualnej w konstruktorze lub destruktory świadczy, że obiekt jest częściowo zbudowany lub zniszczony (22.4). Dlatego wywoływanie tych funkcji w konstruktorze i destruktory jest zazwyczaj złym pomysłem.

20.3.3. Bezpośrednia kwalifikacja

Wywołanie funkcji przy użyciu operatora zakresu, `::`, jak w przykładzie `Manager::print()`, powoduje wyłączenie mechanizmu wywoływania funkcji wirtualnych:

```
void Manager::print() const
{
    Employee::print(); // to nie jest wywołanie wirtualne
    cout << "\tpoziom " << level << '\n';
    //...
}
```

W przeciwnym przypadku funkcja `Manager::print()` wpadłaby w nieskończoną rekurencję. Użycie nazwy z kwalifikatorem ma też pewną inną zaletę. Jeśli funkcja wirtualna jest dodatkowo `inline` (co zdarza się dość często), to wywołania z użyciem operatora `::` mogą być poddawane inliningowi. To pozwala programiście sprawnie poradzić sobie z ważnymi specjalnymi przypadkami, w których jedna funkcja wirtualna wywołuje inną na tym samym obiekcie. Przykładem tego jest funkcja `Manager::print()`. Dzięki temu, że typ obiektu jest określany przy wywoływaniu funkcji `Manager::print()`, nie trzeba go określać dynamicznie po raz kolejny dla wywołania `Employee::print()`.

20.3.4. Kontrola przesłaniania

Jeśli w klasie pochodnej zadeklaruje się funkcję o takiej samej nazwie i typie jak funkcja wirtualna w klasie bazowej, to funkcja w klasie pochodnej przesłoni funkcję z klasy bazowej. Jest to bardzo prosta i niezwykle efektywna zasada. Ale w dużych hierarchiach klas czasami można się pogubić w ocenie, czy przesłaniana jest ta funkcja, co potrzeba. Na przykład:

```
struct B0 {
    void f(int) const;
    virtual void g(double);
};

struct B1 : B0 { /* */ };
struct B2 : B1 { /* */ };
struct B3 : B2 { /* */ };
struct B4 : B3 { /* */ };
struct B5 : B4 { /* */ };

struct D : B5 {
    void f(int) const; // przesłania f() z klasą bazową
    void g(int);       // przesłania g() z klasą bazową
    virtual int h();   // przesłania h() z klasą bazową
};
```

W kodzie tym przedstawione są trzy błędy, które bardzo trudno wykryć w prawdziwej hierarchii klas, w której klasy `B0 – B5` mają wiele składowych i są rozproszone w kilku plikach nagłówkowych:

- Funkcja `B0::f()` nie jest wirtualna, więc nie można jej przesłonić, a jedynie ukryć (20.3.5).
- Funkcja `D::g()` ma inny typ argumentu niż funkcja `B0::g()`, a więc jeśli coś przesłania, to na pewno nie jest to funkcja wirtualna `B0::g()`. Najprawdopodobniej ukrywa tylko funkcję `B0::g()`.
- W klasie `B0` nie ma funkcji o nazwie `h()`, więc jeśli funkcja `D::h()` cokolwiek przesłania, nie jest to funkcja z klasy `B0`. Najprawdopodobniej jest to po prostu wprowadzenie nowej funkcji wirtualnej.

Nie pokazałem kodu źródłowego klas `B1 – B5`, więc może znajdują się w nich deklaracje coś jeszcze zmieniają. Osobiście nie używam niepotrzebnie słowa kluczowego `virtual` do oznaczania funkcji przeznaczonych do przesłaniania. W małych programach (zwłaszcza gdy używa się kompilatora zgłaszającego przyzwoite ostrzeżenia na temat typowych błędów) nie jest trudno poprawnie zastosować zasady przesłaniania. Ale w większych hierarchiach konieczne może być użycie dodatkowych elementów kontrolnych:

- `virtual` — funkcja może być przesłaniana (20.3.2).
- `=0` — funkcja musi być wirtualna i przesłonięta (20.4).
- `override` — funkcja przesłania funkcję wirtualną z klasy bazowej (20.3.4.1).
- `final` — funkcja nie jest przeznaczona do przesłaniania (20.3.4.2).

Jeśli nie ma ani jednego z tych kontrolerów, niestatyczna funkcja składowa jest wirtualna, ale pod warunkiem że przesłania funkcję wirtualną z klasy bazowej (20.3.2).

Kompilator może ostrzegać o niespójnym stosowaniu kontrolerów przesłaniania. Przykładowo deklaracja klasy zawierająca słowo kluczowe `override` w siedmiu z dziewięciu funkcji wirtualnych może być bardzo myląca dla innych programistów.

20.3.4.1. override

Można bezpośrednio zaznaczyć swój zamiar co do przesłaniania:

```
struct D : B5 {
    void f(int) const override; // błąd: B0::f() nie jest funkcją wirtualną
    void g(int) override; // błąd: B0::g() przyjmuje argument typu double
    virtual int h() override; // błąd: nie ma funkcji h() do przesłonięcia
};
```

Przy takiej definicji (i założeniu, że pośrednie klasy bazowe `B1 – B5` nie zawierają odpowiednich funkcji) wszystkie trzy deklaracje są błędne.

W dużych i skomplikowanych hierarchiach klas zawierających wiele funkcji wirtualnych słowa kluczowego `virtual` najlepiej jest używać tylko do wprowadzania nowych funkcji wirtualnych, a słowa `override` do oznaczania wszystkich funkcji mających służyć do przesłaniania innych funkcji. Użycie słowa kluczowego `override` zwiększa objętość kodu, ale pozwala wyraźnie przedstawić zamiar programisty.

Specyfikator `override` wpisuje się na ostatnim miejscu w deklaracji:

```
void f(int) const noexcept override; // OK (jeśli istnieje odpowiednia funkcja f() do przesłonięcia)
override void f(int) const noexcept; // błąd składni
void f(int) override const noexcept; // błąd składni
```

Tak, to, że specyfikator `virtual` jest przedrostkiem, a `override` przyrostkiem, jest nielogiczne. Ale jest to cena za zgodność i stabilność, którą płacimy już od dziesięcioleci.

Specyfikator `override` nie jest częścią typu funkcji i nie powinien być używany w definicjach poza klasą. Na przykład:

```
class Derived : public Base {
    void f() override;           // OK, jeśli Base ma wirtualną funkcję f()
    void g() override;           // OK, jeśli Base ma wirtualną funkcję g()
};

void Derived::f() override // błąd: słowo kluczowe override poza klasą
{
    //...
}

void Derived::g()           // OK
{
    //...
}
```

Co ciekawe, `override` nie jest zwykłym słowem kluczowym, tylko tzw. **kontekstowym słowem kluczowym**, czyli w niektórych kontekstach ma specjalne znaczenie, a w innych może być używane jako identyfikator. Na przykład:

```
int override = 7;

struct Dx : Base {
    int override;

    int f() override
    {
        return override + ::override;
    }
};
```

Nie należy jednak tak wykorzystywać tego słowa, bo to tylko utrudnia utrzymanie kodu. Jedynym powodem, dla którego `override` jest kontekstowym słowem kluczowym, jest to, że przez dziesięciolecia używano go jako zwykłego identyfikatora w różnych programach. Drugie takie słowo to `final` (20.3.4.2).

20.3.4.2. final

Deklarując funkcję składową, możemy wybrać, czy ma być wirtualna, czy nie (domyślnie). Funkcje wirtualne definiuje się po to, by twórcy klas pochodnych mogli je definiować lub przedefiniowywać. Wybór rodzaju funkcji zależy od znaczenia (semantyki) klasy:

- Czy przewidujemy, że mogą być tworzone klasy pochodne?
- Czy projektant klasy pochodnej musi przedefiniować funkcję, aby osiągnąć cel?
- Czy poprawne przesłonięcie funkcji jest łatwe (tzn. czy względnie łatwo można w funkcji przesłaniającej dostarczyć semantykę funkcji wirtualnej)?

Jeśli na wszystkie powyższe pytania odpowiesz „nie”, to możesz pozostawić funkcję jako nie-wirtualną. Dzięki temu trochę uprościsz program i w niektórych przypadkach nieznacznie zyskasz na wydajności (głównie dzięki inliningowi). Biblioteka standardowa jest pełna przykładów takich funkcji.

Rzadziej zdarzają się sytuacje, w których początkowa hierarchia klas zawiera funkcje wirtualne, ale po zdefiniowaniu zestawu klas pochodnych odpowiedź na jedno z pytań zmienia się na „nie”. Można sobie na przykład wyobrazić abstrakcyjne drzewo składniowe dla języka, w którym wszystkie konstrukcje językowe zostały zdefiniowane jako konkretne klasy węzłowe pochodne od kilku interfejsów. Utworzenie nowej klasy pochodnej może być potrzebne, tylko gdy zmieni się język. W takim przypadku możemy chcieć uniemożliwić użytkownikom przesłanianie funkcji wirtualnych, bo jedynym skutkiem takich przesłonięć byłaby zmiana semantyki naszego języka. Innymi słowy, czasami trzeba zamknąć projekt przed dalszymi modyfikacjami. Na przykład:

```
struct Node { // klasa interfejsowa
    virtual Type type() = 0;
    ...
};

class If_statement : public Node {
public:
    Type type() override final; // uniemożliwienie dalszego przesłaniania
    ...
};
```

W realnej hierarchii klas ogólny interfejs (tutaj: `Node`) od klasy pochodnej reprezentującej konkretną konstrukcję językową (tutaj: `If_statement`) oddzielałoby kilka klas pośrednich. Ale w przykładzie tym chodzi przede wszystkim o pokazanie, że funkcja `Node::type()` ma być przesłonięta (dlatego została zadeklarowana jako wirtualna), a przesłaniająca ją funkcja `If_statement` nie może (dlatego zadeklarowano ją jako `final`). Funkcja składowa zawierająca w deklaracji słowo `final` nie może być przesłonięta, a próba jej przesłonięcia oznacza błąd. Na przykład:

```
class Modified_if_statement : public If_statement {
public:
    Type type() override; // błąd: funkcja If_statement::type() jest finalna
    ...
};
```

Można też sprawić, by wszystkie funkcje wirtualne klasy były finalne. W tym celu należy dodać słowo `final` po nazwie klasy. Na przykład:

```
class For_statement final : public Node {
public:
    Type type() override;
    ...
};

class Modified_for_statement : public For_statement { // błąd: klasa For_statement jest finalna
    Type type() override;
    ...
};
```

Dodatkowa deklaracja klasy jako finalnej sprawia, że nie można tworzyć jej klas pochodnych. Niektórzy próbują z użycia słowa kluczowego `final` uzyskać korzyści wydajnościowe — nie-wirtualna funkcja jest szybsza od wirtualnej (o około 25% w nowoczesnej implementacji) i łatwiej ją poddać inliningowi (12.1.5). Jednak nie należy ślepo używać tego słowa jak optymalizatora. Ma ono wpływ na projekt hierarchii klas (często negatywny), a zwiększenie wydajności

jest zazwyczaj tylko nieznaczne. Zanim stwierdzisz, że program działa szybciej, dokładnie go przetestuj. Słowa kluczowego `final` używaj w sytuacjach, gdy jego użycie nie budzi wątpliwości co do poprawności projektu. Innymi słowy, używaj go w celu zaspokojenia potrzeb semantycznych.

Specyfikator `final` nie jest częścią typu funkcji i nie może znajdować się w definicji funkcji poza klasą. Na przykład:

```
class Derived : public Base {
    void f() final;      // OK, jeśli klasa Base zawiera wirtualną funkcję f()
    void g() final;      // OK, jeśli klasa Base zawiera wirtualną funkcję g()
    ...
};

void Derived::f() final // błąd: specyfikator final poza klasą
{
    ...
}

void Derived::g()      // OK: brak słowa final
{
    ...
}
```

Podobnie jak `override` (20.3.4.1), również `final` jest kontekstowym słowem kluczowym, tzn. ma specjalne znaczenie w niektórych miejscach, ale może też być używane jako zwykły identyfikator. Na przykład:

```
int final = 7;

struct Dx : Base {
    int final;

    int f() final
    {
        return final + ::final;
    }
};
```

Nie należy jednak tak wykorzystywać tego słowa, bo to tylko utrudnia utrzymanie kodu. Jedynym powodem, dla którego `final` jest kontekstowym słowem kluczowym, jest to, że przez dziesięciolecia używano go jako zwykłego identyfikatora w różnych programach. Drugie takie słowo to `override` (20.3.4.1).

20.3.5. Używanie składowych klasy bazowej

Przesłanianie funkcji nie działa między zakresami dostępności (12.3.3). Na przykład:

```
struct Base {
    void f(int);
};

struct Derived : Base {
    void f(double);
};
```

```
void use(Derived d)
{
    d.f(1); // wywołuje Derived::f(double)
    Base& br = d
    br.f(1); // wywołuje Base::f(int)
}
```

Niektórzy są tym zaskoczeni i czasami chcielibyśmy, aby została wybrana najlepsza funkcja składowa zgodnie z zasadami przeciążania. Jeśli chodzi o przestrzeń nazw, to dowolną funkcję do zakresu można dodać przy użyciu deklaracji `using`. Na przykład:

```
struct D2 : Base {
    using Base::f; // dodaje wszystkie f z Base do D2
    void f(double);
};

void use2(D2 d)
{
    d.f(1); // wywołuje D2::f(int), tzn. Base::f(int)
    Base& br = d
    br.f(1); // wywołuje Base::f(int)
}
```

Jest to prosta konsekwencja traktowania klas jako przestrzeni nazw (16.2).

Za pomocą kilku deklaracji `using` można dodać nazwy z kilku klas bazowych. Na przykład:

```
struct B1 {
    void f(int);
};

struct B2 {
    void f(double);
};

struct D : B1, B2 {
    using B1::f;
    using B2::f;
    void f(char);
};

void use(D d)
{
    d.f(1); // wywołuje D::f(int), tzn. B1::f(int)
    d.f('a'); // wywołuje D::f(char)
    d.f(1.0); // wywołuje D::f(double), tzn. B2::f(double)
}
```

Do zakresu klasy pochodnej można przenieść konstruktory — 20.3.5.1. Dostępność nazwy przeniesionej do zakresu klasy pochodnej za pomocą deklaracji `using` jest określona przez miejsce występowania tej deklaracji — 20.5.3. Za pomocą dyrektywy `using` nie można przenieść wszystkich składowych klasy bazowej do pochodnej.

20.3.5.1. Dziedziczenie konstruktorów

Powiedzmy, że potrzebujemy typu wektorowego podobnego do `std::vector`, ale gwarantującego sprawdzanie zakresu. Możemy spróbować czegoś takiego:

```
template<typename T>
struct Vector : std::vector<T> {
    using size_type = typename std::vector<T>::size_type; // użycie typu rozmiaru wektora

    T& operator[](size_type i) { return this->at(i); } // użycie dostępu kontrolowanego
    const T& operator[](size_type i) const { return this->at(i); }
};
```

Niestety szybko się przekonamy, że ta definicja jest niekompletna. Na przykład:

```
Vector<int> v { 1, 2, 3, 5, 8 }; // błąd: brak konstruktora z listą inicjalizacyjną
```

Klasa `Vector` nie dziedziczyła po `std::vector` żadnych konstruktorów.

Zasada ta nie jest pozbawiona sensu: gdyby klasa dodawała zmienne składowe do klasy bazowej lub stosowała bardziej restrykcyjny niezmiennik, to dziedziczenie konstruktorów byłoby katastrofą. Ale w klasie `Vector` nic podobnego nie miało miejsca.

Problem ten można łatwo rozwiązać poprzez zaznaczenie, że konstruktory mają być dziedziczone:

```
template<typename T>
struct Vector : std::vector<T> {
    using size_type = typename std::vector<T>::size_type; // użycie typu rozmiaru wektora

    using std::vector<T>::vector; // dziedziczenie konstruktorów wektora

    T& operator[](size_type i) { return this->at(i); } // użycie kontrolowanego dostępu
    const T& operator[](size_type i) const { return this->at(i); }
};

Vector<int> v { 1, 2, 3, 5, 8 }; // OK: użycie konstruktora z listą inicjalizacyjną z std::vector
```

Przedstawiony tu sposób użycia deklaracji `using` niczym nie różni się od użycia ich dla zwykłych funkcji (14.4.5, 20.3.5).

Jeśli chcesz, możesz strzelić sobie w stopę, dziedzicząc konstruktory w klasie pochodnej, w której definiujesz nowe zmienne składowe wymagające jawniej inicjalizacji:

```
struct B1 {
    B1(int) {}
};

struct D1 : B1 {
    using B1::B1; // niejawnie deklaruje D1(int)
    string s; // string ma domyślny konstruktor
    int x; // „zapomnieliśmy” zainicjować x
};

void test()
{
    D1 d {6}; // ups: składowa d.x jest niezainicjowana
    D1 e; // błąd: D1 nie ma domyślnego konstruktora
}
```

Składowa D1::s jest inicjowana, a D1::x nie, ponieważ odziedziczenie konstruktora jest równoważne z użyciem konstruktora po prostu inicjującego bazę. W tym przypadku równie dobrze mogliśmy napisać taki kod:

```
struct D1 : B1 {
    D1(int i) : B1(i) { }
    string s; // typ string ma domyślny konstruktor
    int x;    // „zapomnieliśmy” zainicjować x
};
```

Jednym z rozwiązań problemu jest dodanie inicjatora składowej wewnątrz klasy (17.4.4):

```
struct D1 : B1 {
    using B1::B1; // niejawnie deklaruje D1(int)
    int x {0};    // uwaga: składowa x jest zainicjowana
};

void test()
{
    D1 d {6}; // d.x wynosi zero
}
```

W większości przypadków lepiej jest nie wymyślać sprytnych rozwiązań, tylko ograniczyć dziedziczenie konstruktorów do prostych przypadków, gdy nie są dodawane żadne zmienne składowe.

20.3.6. Rozluźnienie zasady dotyczącej typów zwrotnych

Reguła nakazująca, że typ zwrotny funkcji przesłaniającej musi być taki sam jak typ zwrotny przesłanianej funkcji wirtualnej, może być rozlużniona. To znaczy, że jeśli oryginalnym typem zwrotnym jest B*, to typem zwrotnym funkcji przesłaniającej może być D*, pod warunkiem że B jest publiczną bazą D. Podobnie typ zwrotny B& może zostać zamieniony na D&. Czasami nazywa się to zasadą **kowariantnych typów zwrotnych**.

Rozlużniona zasada dotyczy tylko typów zwrotnych będących wskaźnikami lub referencjami, nie zaś „inteligentnych wskaźników”, takich jak unique_ptr (5.2.1). Zasada ta nie dotyczy też typów argumentów, bo prowadziłoby to do łamania typów.

Zastanówmy się nad hierarchią klas reprezentującą różne rodzaje wyrażeń. Oprócz operacji do manipulowania wyrażeniami klasa bazowa Expr zawierałaby narzędzia do tworzenia nowych obiektów wyrażeń i różnych typów wyrażeń:

```
class Expr {
public:
    Expr();           // konstruktor domyślny
    Expr(const Expr&); // konstruktor kopiąjący
    virtual Expr* new_expr() =0;
    virtual Expr* clone() =0;
    ...
};
```

Funkcja new_expr() ma tworzyć domyślny obiekt typu wyrażenia, a clone() powinna wykonywać kopię obiektu. Obie te funkcje zwracają obiekt konkretnej klasy będącej pochodną klasy Expr. Nie mogą zwrócić obiektu samej klasy Expr, bo ta została celowo i słusznie zadeklarowana jako abstrakcyjna.

W klasie pochodnej można przesłonić funkcje `new_expr()` i `clone()`, aby zwracały obiekty własnego typu:

```
class Cond : public Expr {
public:
    Cond();
    Cond(const Cond&);
    Cond* new_expr() override { return new Cond(); }
    Cond* clone() override { return new Cond(*this); }
    ...
};
```

To oznacza, że mając dany obiekt klasy `Expr`, użytkownik może utworzyć nowy obiekt „tego samego typu”. Na przykład:

```
void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    ...
}
```

Wskaźnik przypisany do `p2` może wskazywać „czysty obiekt klasy `Expr`”, ale będzie wskazywał obiekt typu pochodnego tej klasy, np. `Cond`.

Typem zwrotnym funkcji `Cond::new_expr()` i `Cond::clone()` jest `Cond*`, nie zaś `Expr*`. Dzięki temu można klonować obiekty klasy `Cond` bez utraty informacji o typie. Analogicznie klasa `Addition` zawierałaby funkcję `clone()` zwracającą `Addition*`. Na przykład:

```
void user2(Cond* pc, Addition* pa)
{
    Cond* p1 = pc->clone();
    Addition* p2 = pa->clone();
    ...
}
```

Gdybyśmy użyli funkcji `clone()` klasy `Expr`, to wiedzielibyśmy jedynie, że wynik jest typu `Expr*`:

```
void user3(Cond* pc, Expr* pe)
{
    Cond* p1 = pc->clone();
    Cond* p2 = pe->clone(); // błąd: funkcja Expr::clone() zwraca typ Expr*
    ...
}
```

Ponieważ funkcje `new_expr()` i `clone()` są wirtualne, a jednocześnie (pośrednio) tworzą obiekty, są one przedstawicielami tzw. **konstruktorów wirtualnych**. Każda z nich wykorzystuje jakiś konstruktor w celu utworzenia odpowiedniego obiektu.

Aby utworzyć obiekt, konstruktor musi znać dokładny typ obiektu, który ma utworzyć. W związku z tym konstruktor nie może być wirtualny. Ponadto konstruktor nie jest zwykłą funkcją, bo współpracuje z procedurami zarządzania pamięcią w sposób, w jaki nie robią tego zwykłe funkcje. Dlatego nie można utworzyć wskaźnika do konstruktora, aby przekazać go do funkcji tworzącej obiekty.

Oba te ograniczenia można obejść poprzez zdefiniowanie funkcji wywołującej konstruktor i zwracającej utworzony obiekt. Jest to korzystne, bo tworzenie obiektu bez znajomości jego typu to często przydatna możliwość. Przykładem klasy specjalnie stworzonej do tego celu jest `Ival_box_maker` (21.2.4).