Instrukcja II

Wstęp

Zasoby

Na tych zajęciach skupimy się na zarządzaniu zasobami, życiem obiektów oraz metodach do tego służącym. Jest to temat kluczowy w C++, ponieważ język ten nie posiada automatycznego zarządzania zasobami, np. w postaci garbage collectora dostępnego w Javie czy C#. Jest to, z jednej strony, problematyczne, gdyż zmusza nas do poświęcenia uwagi oraz czasu na sprawy, które mogłyby potencjalnie zostać załatwione przez runtime programu, bez naszego udziału. Z drugiej strony, nie musimy płacić za wygodę takich rozwiązań wydajnością kodu. Jest to zgodne z filozofią C++: *don't pay for what you don't use*. Zacznijmy od (niekoniecznie ścisłej) definicji: przez zasób rozumiemy byt, który wykorzystujemy do prawidłowego funkcjonowania naszego programu i którego stworzenie i/lub uwolnienie wymaga nietrywialnego nakładu pracy. Przykładami zasobów mogą być:

- **zaalokowana pamięć**
- otwarty plik
- stworzony wątek
- otwarte połączenie sieciowe

Każdy z ww. przykładów zasobów wymaga pracy przy stworzeniu oraz zniszczeniu (np. pamięć trzeba zaalokować i zwolnić). Mamy tu na myśli zarówno pracę programistyczną (napisanie odpowiednich instrukcji) oraz pracę komputera (np. alokacja pamięci wymaga komunikacji z systemem operacyjnym, który ją przydziela). Drugi typ pracy jest nieunikniony, natomiast ten pierwszy możemy znacznie ograniczyć.

RAII

Wzorcem projektowym, który nam do tego posłuży jest RAII - *resource acquisition is initialization*. Akronim ten, pomimo dość tajemniczo brzmiącego rozwinięcia, opisuje bardzo prostą koncepcję: reprezentujemy zasoby poprzez obiekty, tworzymy zasoby w konstruktorach tych obiektów, a zwalniamy w destruktorach. Dzięki temu kod konieczny do zarządzania zasobem piszemy w jednym miejscu, a następnie wykorzystujemy go w trakcie normalnej pracy z danym obiektem. Takie podejście ma następujące zalety:

- Zwiększa poprawność kodu. Obiekty są automatycznie niszczone przy wyjściu ze scope'u, także nigdy nie zapomnimy już np. zwolnić pamięci! Znak "}" jest naszym przyjacielem.
- Zmniejsza liczbę linijek kodu, które musimy napisać.
- Pozwala zachować poprawność programu w sytuacjach wyjątkowych.

O wyjątkach mowa będzie dopiero za na ostatnich zajęciach. Na tę chwilę powiedzmy tylko, że są sytuacje, w których jakaś operacja może się nie powieść i program natychmiast przerwie wykonywanie bieżącej funkcji, funkcji, która ją zawołała, itd., aż do momentu, w którym przewidziana jest obsługa takiej wyjątkowej sytuacji. Przed opuszczeniem tych funkcji, program postara się jednak zniszczyć wszystkie obiekty, które zostały dotychczas stworzone (tzw. rozwijanie stosu, ang. *stack unwinding*). Dzięki zastosowaniu RAII zapewniamy poprawne zwolnienie zasobów, co jest szczególnie ważne, jeżeli program ma odzyskać sprawność. Przykładowo, klient może próbować połączyć się z serwerem przy użyciu funkcji, która alokuje pamięć. Serwer nie odpowiada jednak przez jakiś czas, w związku z czym klient decyduje się na porzucenie próby połączenia w trybie awaryjnym. Klient potrafi sobie jednak poradzić z taką sytuacją, np. ponownie wysyłając żądanie do serwera, albo wyświetlając odpowiednią wiadomość użytkownikowi i zajmując się czymś innym. Dzięki odpowiednio napisanemu destruktorowi, można uniknąć wycieku zaalokowanej przez klienta pamięci.

RAII jest jednak sposobem projektowania kodu, nie elementem języka C++ (RAII miało swój początek w C++, ale obecnie wykorzystywane jest także np. w językach Ada i Rust). Zapoznajmy się z konkretną funkcjonalnością oferowaną przez C++, która pozwala stosować RAII, a także inne mechanizmy służące do optymalnego i niekłopotliwego zarządzania zasobami. Skupimy się tu na pamięci, ale innymi zasobami zarządza się analogicznie. Nadzieją autorów jest, że po wykonaniu tej instrukcji czytelnik potrafić będzie pisać czysty kod, który nie powoduje wycieków oraz nie wykorzystuje 10kB tam, gdzie wystarczy 1kB.

Dynamiczna alokacja pamięci w C++

Zanim przejdziemy do zarządzania zasobami, musimy dowiedzieć się, jak je w ogóle stworzyć. Przypomnijmy, jak wyglądała alokacja pamięci w C:

```
// Poniższe 3 linijki zazwyczaj zapisywaliśmy w 1, tutaj rozbijamy ją dla celo
const size_t rozmiar_doubla = sizeof(double);
void*        wynik_alokacji = malloc(rozmiar_doubla);
```

```
double*      liczba_wsk     = (double*)wynik_alokacji;
*liczba_wsk = 42.;
// Użyj do czegoś wartości 42...
free(liczba_wsk);
```

Powyższe odwołanie do funkcji `malloc` czytamy jako "zaalokuj blok pamięci o wielkości `rozmiar_doubla` bajtów, a następnie podaj mi adres tego bloku pamięci". Dodatkowo przed przypisaniem otrzymanego adresu do zmiennej `liczba` musimy zrzutować go na typ `double*`, gdyż funkcja `malloc` otrzymuje jedynie informację o liczbie bajtów, nie tym, jakiego typu zmienne chcemy tam umieścić (zwraca typ `void*`). Jest to typowe dla języka C - operujemy na gołej pamięci i sami musimy martwić się o typ zmiennych, które w tej pamięci umieszczamy. W C++ kierujemy się fundamentalnie inną filozofią: operujemy na obiektach, których życie zaczyna się od konstruktora, a kończy na destruktorze. Jeżeli tworzymy krzesło, to chcemy na nim siedzieć, a nie przestawiać jego bity i 4 linijki kodu dalej traktować je jak stół. Zobaczmy, jak dynamiczna alokacja wygląda w C++:

```
double* liczba = new double{42.};
// Użyj do czegoś wartości 42
delete liczba;
```

Powyższy kod czytamy jako "zaalokuj dynamicznie pamięć na obiekt typu `double`, stwórz ten obiekt używając konstruktora1 z argumentem 42, a następnie przypisz adres utworzonego obiektu do zmiennej `liczba`". Dodatkowo możemy zauważyć, że `new` i `delete` są słowami kluczowymi języka, nie bazują na żadnym z nagłówków biblioteki standardowej (np. `stdlib.h`)2. Dodajmy też, że składnia

```
const unsigned int n      = 100;
double*            wektor = new double[n];
// ...
delete[] wektor;
```

ma analogiczną interpretację. Przy alokowaniu tablic musimy jedynie pamiętać o użyciu operatora `delete[]` zamiast `delete`.

Na tych zajęciach spróbujemy napisać klasę `Wektor`, reprezentującą wektor należacy do przestrzeni Rn, automatycznie dostosowujący swój rozmiar do potrzeb użytkownika (podobnie jak zachowują się wektory np. w matlabie). Zaznaczmy tu, że zachowanie tej klasy będzie inne, niż klasy `std::vector<double>` (z późniejszych instrukcji), o czym należy pamiętać.

Zadanie 1 Napisz klasę `Wektor`, przechowującą liczby typu `double`, która dynamicznie alokuje pamięć przy konstrukcji i zwalnia ją przy zniszczeniu. Jako argument konstruktora przyjmij zmienną typu całkowitego, reprezentującą ile liczb zmiennoprzecinkowych ma przechowywać wektor (jego długość). Na chwilę obecną dostęp do elementów wektora udostępnij czyniąc publicznym wskaźnik do tablicy, w której przechowujesz elementy wektora.

Zadanie 2 Zmodyfikuj konstruktor klasy `Wektor` tak, aby początkowo przypiswał elementom wektora wartość `0.`.

Zadanie 3 Dodaj do klasy `Wektor` prywatne pole `dlugosc`, przechowujące informację, ile elementów znajduje się obecnie w wektorze (inicjalizuj to pole argumentem konstruktora). Napisz getter (ale nie setter) dla tego pola.

Zadanie 4 Dodaj do klasy `Wektor` publiczną metodę `void print()` która drukuje obecną zawartość wektora (elementy od `0` do `dlugosc - 1`).

Zadanie 5 Dodaj do klasy `Wektor` prywatne pole `pojemnosc`, reprezentujące rozmiar tablicy (w znaczeniu liczby obiektów typu `double`, nie liczby bajtów) przechowującej elementy wektora (w kolejnych zadaniach `dlugosc` i `pojemnosc` staną się do pewnego stopnia niezależne). Napisz getter (ale nie setter) dla tego pola.

Zadanie 6 Dodaj do klasy `Wektor` publiczną metodę `zmienDlugosc`, która przyjmuje zmienną typu całkowitego, reprezentującą nową długość wektora.

- Jeżeli żądana długość jest mniejsza lub równa obecnej pojemności wektora oraz mniejsza lub równa obecnej długości, zmniejsz jedynie wartość pola `dlugosc`.
- Jeżeli żądana długość jest mniejsza lub równa obecnej pojemności wektora oraz większa od długości, zwiększ odpowiednio wartość pola `dlugosc` i wyzeruj elementy tablicy, które znalazły się teraz w wektorze (nie wiemy, co było tam wcześniej).
- Jeżeli żądana długość jest większa niż obecna pojemność wektora, zaalokuj nowy blok pamięci, przepisz do niego istniejące elementy wektora i wyzeruj te nowoutworzone. Nie zapomnij o skasowaniu starego bloku pamięci!
- Sytuacja, w której pojemność jest mniejsza od długości wektora, jest niemożliwa.

Zadanie 7 Przetestuj, czy stworzona przez Ciebie klasa zachowuje się zgodnie z oczekiwaniami. Przydatna do tego może być metoda `print`.

---

1 Formalnie, wbudowane typy (`char`, `int`, `float` itd.) nie mają konstruktorów, ale także możemy je inicjalizować przy użyciu nawiasów `{ }`. Prezentowany kod zadziała analogicznie dla dowolnego innego typu.

2 Ściślej mówiąc, operatory `new`, `new[]`, `delete` i `delete[]` są zdefiniowane w nagłówku `new`, ale jest on dołączany nawet bez jawnego zawołania `#include <new>`. Dla ciekawych: operatory te także można przeciążać, *vide* dokumentacja. Alokacja pamięci jest dość szerokim tematem, w który nie mamy niestety czasu się tu zagłębiać.

Referencje

Skoro wiemy już jak tworzyć i niszczyć zasoby, zastanówmy się teraz jak korzystać z nich w wydajny sposób. Bardzo często zdarza się, że chcemy wykorzystać jakiś obiekt wewnątrz funkcji (metod tej samej lub innej klasy, bądź też funkcji "wolnostojących"). Rozważmy poniższy kod, znany nam z C:

```
void print_plus1(int arg)
{
    ++arg;
    printf("%d ", arg);
}

int main()
{
    int a = 0;
    print_plus1(a);
    printf("%d\n", a);
    return 0;
}
```

Taki program wydrukuje oczywiście `1 0`, gdyż funkcja `print_plus1` robi kopię podanego argumentu, także nie zmieni ona wartości zmiennej `a` w funkcji `main`. Spójrzmy na analogiczny kod w C++:

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

void fun(T obj)
{
    // Zrób coś z obj...
}

int main()
{
    T t;
    fun(t);
}
```

Podobnie jak wyżej, funkcja `fun` będzie działała na *kopii* argumentu `t` (o tym, co dokładnie znaczy kopia w kontekście klasy powiemy za chwilę). Jak już ustaliliśmy, obiekt może być "duży", tzn. być właścicielem jakichś zasobów, mieć wiele pól, itd. Wykonywanie jego kopii może nas wtedy kosztować zarówno czas, jak i pamięć. Ponad to, podobnie jak wyżej, nie możemy bezpośrednio modyfikować jego wartości. W języku C rozwiązaniem tego problemu były wskaźniki. Wskaźniki istnieją także w C++, ale dużo bardziej eleganckim rozwiązaniem są referencje. Pozwalają one uniknąć chmary operatorów wzięcia adresu (`&`) oraz dereferencji(`*`). Referencja do obiektu typu `T` ma typ `T&`. Zauważmy, że znak `&` jest tutaj częścią typu, nie operatorem! Podkreślmy kilka cech referencji:

- Po utworzeniu, korzystamy z referencji dokładnie tak, jak z obiektu, do którego się odnosi.

```
int  a     = 0;
int& a_ref = a;

++a_ref;    // Zwiększyliśmy a o 1
a_ref = 42; // Zmieniliśmy wartość a na 42
```

- W przeciwieństwie do wskaźników, referencje nie mogą być puste. Wymusza to na nas przypisanie do referencji obiektu, do którego się odnosi. Poniższy kod się nie skompiluje:

```
int a;
int& a_ref; // błąd!
a_ref = a;
```

- Referencja nie może po utworzeniu zacząć odnosić się do innego obiektu. Wynika to z pierwszej wymienionej własności. `a_ref = b` przypisuje wartość `b` do `a`, nie "przypina" referencji do `b` do obiektu `a_ref`. Ponownie, jest to inne zachowanie, niż w przypadku wskaźników.
- Nie istnieje coś takiego jak referencja do referencji (`T&&` oznacza coś zupełnie innego, co zostanie wytłumaczone w dalszej części instrukcji). Znów jest inaczej niż przy wskaźnikach, które można zagnieżdżać dowolnie wiele razy (`int****` jest w pełni legalnym typem).

Konsekwencją wypunktowanych powyżej własności jest to, że referencje pojawiają się w praktyce prawie tylko w typach argumentów przyjmowanych oraz zwracanych przez funkcję. W przykładzie powyżej, zmienna `a_ref` jest nam w zasadzie niepotrzebna. Mamy przecież dostęp bezpośrednio do `a`. Wywołanie `fun(a_ref)` **nadal wykona kopię wartości** `a`. Bardzo ważne jest, żeby zrozumieć, dlaczego tak jest. `a_ref` jest typu `int&`, ale `fun` spodziewa się wartości typu `int`. To sygnatura funkcji decyduje o tym, jak zostaną potraktowane argumenty (czy zostaną skopiowane, czy zostanie użyta ich referencja). Aby uniknąć kopii, musimy więc postąpić następująco:

```
void print_plus2(int& arg) // Jedyna zmiana w kodzie jest tu!
{
    arg += 2;
    std::cout << arg;
}

int main()
{
    int a = 0;
    print_plus2(a);
    std::cout << a;
    return 0;
}
```

Teraz program wydrukuje 2 2. Zanim wykorzystamy referencje w praktyce, zauważmy jeszcze, że ze względu na cechy uwypuklone powyżej, referencje są nie tylko prostsze, ale też bezpieczniejsze w użyciu niż wskaźniki - trudniej jest stworzyć referencję, która do niczego się nie odnosi. Jest to trudne, lecz nie niemożliwe:

```
int& getInt()
{
    int a = 0;
    return a;
}
```

Powyższa funkcja zwraca referencję do lokalnego obiektu, który jest niszczony wraz z końcem jej wykonania. Wartość `int a = getInt()` jest nieokreślona. Szczęśliwie, kompilator powinien wykryć taką sytuację i wydrukować odpowiednie ostrzeżenie.

Zadanie 8 Uczyń tablicę, która przechowuje elementy klasy `Wektor` prywatną. Przeciąż odpowiednio operator `[]` tak, aby zwracał referencję do elementu o podanym indeksie. Przyjmij indeksowanie od 0, zauważ jednak, że nic nie stoi na przeszkodzie, aby Twój operator `[]` zaczynał indeksowanie od 1.

Zadanie 9 Sprawdź działanie operatora `[]`. Co stanie się, gdy zawołasz `wektor[0] = 42.;`? Co stanie się, gdy zawołasz `double a = wektor[0]; a++;`?

Zadanie 10 Zmodyfikuj operator `[]` tak, aby sięgnięcie po element leżący poza obecnym zakresem wektora skutkowało automatycznym zwiększeniem jego długości. Użyj napisanej wcześniej metody `zmienDlugosc`.

Zadanie 11 Sprawdź działanie klasy `Wektor`. Zauważ, że nie pozwala ona teraz sięgnąć do niedostępnych miejsc pamięci! W najgorszym wypadku wyczerpiemy dostępną pamięć RAM.

Listy inicjalizacyjne

Przyjrzyjmy się teraz następującej parze klas:

```
struct Kokardka
{
    Kokardka()      { dlugosc = 42; }
```

```
    Kokardka(int d) { dlugosc = d; }

    int dlugosc;
};

struct Prezent
{
    Prezent(int dk)
    {
        // ***
        k.dlugosc = dk;
    }

    Kokardka k;
    // Inne pola ...
};
```

Zadajmy sobie teraz pytanie: przy konstrukcji obiektu typu `Prezent`, jaką długość ma jego kokardka w linijce oznaczonej 3 gwiazdkami? Odpowiedź: 42. Wynika to z faktu, że wszystkie składowe pola klasy `Prezent`, muszą zostać zainicjalizowane przed wykonaniem ciała jego kostruktora. "Pod maską" wołamy zatem domyślny (bezargumentowy) konstruktor klasy `Kokardka`. Gdyby było inaczej, to w konstruktorze prezentu moglibyśmy odnosić się do kokardki, która nie została jeszcze stworzona, co jest logicznie niespójne i skutkowałoby błędami.

Zadanie 12 Upewnij się, że kawałek kodu przedstawiony powyżej rzeczywiście działa tak jak twierdzi jego opis (zamień `***` na odpowiednią komendę drukowania). Usuń domyślny konstruktor klasy `Kokardka`. Czy kod się teraz skompiluje?

W zadaniu 12. możemy zauważyć 2 problemy:

- Inicjalizujemy pole `dlugosc` wartością 42, która zaraz jest nadpisywana. Jest to potencjalna niewydajność - gdyby pole to było drogie w konstrukcji (np. gdyby było typu RAII), wykonywalibyśmy drogą operację, która nie byłaby do niczego potrzebna.
- Jeżeli klasa nie posiada domyślnego konstruktora, to, używając poznanych dotychczas elementów języka, nie moglibyśmy użyć jej jako pole innej klasy. Takie zachowanie byłoby niesamowicie problematyczne!

Szczęśliwie, istnieje mechanizm, który pozwala nam sterować konstrukcją składowych pól klasy: lista inicjalizacyjna. Spójrzmy, jak działa:

```
struct Kokardka
{
    Kokardka(int d) : dlugosc{d} {}

    int dlugosc;
};

struct Prezent
{
    Prezent(int dk) : k{dk} {}

    Kokardka k;
    // Inne pola ...
};
```

Jak widać, rozwiązanie to jest nie tylko bardziej wydajne, ale także zwięźlejsze w zapisie.

Uwaga: W przypadku inicjalizowania wielu pól klasy w liście inicjalizacyjnej, o kolejności decyduje kolejność deklaracji pól w ciele klasy, nie kolejność występowania w liście inicjalizacyjnej. W związku z tym dobrą praktyką jest inicjalizacja pól jedynie na podstawie argumentów konstruktora, nie innych pól zainicjalizowanych gdzie indziej w liście.

Zadanie 13 Zmień konstruktor klasy `Wektor` tak, aby korzystał z listy inicjalizacyjnej.

Szczególne metody klas

Referencje i wskaźniki pozwalają nam unikać wykonywania kopii obiektów wtedy, gdy nie jest to konieczne. Co jednak zrobić, gdy świadomie chcemy skopiować obiekt? W tej części instrukcji powiemy trochę o 2 szczególnych metodach każdej klasy, które do tego służą. Szczególnych metod jest w sumie 5. Jedną z nich - destruktor - już poznaliśmy. Teraz zaznajomimy się z konstruktorem kopiującym i kopiującym operatorem przypisania. Poniżej zamieszczono kawałek kodu ilustrujący ich definicje.

```
class T
{
    // Konstruktor kopiujący
    T(const T& t) { /* ... */ }

    // Kopiujący operator przypisania
    T& operator=(const T& t) { /* ... */ return *this; }

    // Destruktor
    ~T() { /* ... */ }
};
```

Konstruktor kopiujący

Konstruktor kopiujący to konstruktor, który tworzy obiekt na podstawie innego obiektu jego samego typu. Jest on jednoargumentowy - przyjmuje referencję do obiektu, który ma zostać skopiowany. Referencja ta jest stała (`const`), gdyż kopiując obiekt z definicji nie mamy prawa zmienić jego stanu. Zaprezentujmy to na trywialnym przykładzie:

```
struct Liczba
{
    Liczba(int w)            : wartosc{w}         {}
    Liczba(const Liczba& l) : wartosc{l.wartosc} {}

    int wartosc;
};

int main()
{
    Liczba a{1};
    Liczba b{a};  // W celu konstrukcji b wołamy konstruktor kopiujący z
    Liczba c = a; // Tutaj także wołamy konstruktor kopiujący, vide lab.
}
```

Zadanie 14 Skompiluj powyższy kod i upewnij się, że działa poprawnie. Dodaj do konstruktora kopiującego drukowanie informacji o konstrukcji. Upewnij się, że zostanie ona wydrukowana dwukrotnie.

Zadanie 15 Wykomentuj z kodu konstruktor kopiujący. Czy program się skompiluje?

W zadaniu 15. widzimy, że konstruktor kopiujący jest domyślnie tworzony przez kompilator. To dlatego właśnie mówimy, że jest on specjalną metodą. Od standardu C++11 specjalne metody możemy jawnie "zdefaultować":

```
T(const T& t) = default;
```

Jeżeli z kolei nie chcemy, aby klasa miała konstruktor kopiujący, możemy go także usunąć:

```
T(const T& t) = delete;
```

Usuwać można także inne (niespecjalne) metody oraz "wolnostojące" funkcje. Jest to dość często spotykany zabieg, zapobiegający niepoprawnemu użytkowaniu kodu, który piszemy. W przypadku zawołania usuniętej funkcji, kompilator w jasny i zrozumiały sposób zakomunikuje błąd.

Zadanie 16 Dodaj konstruktor kopiujący do klasy `Wektor`. Zwróć uwagę, że musisz zaalokować nowy blok pamięci. Zdecyduj, czy nowy wektor ma mieć pojemność równą długości, czy pojemności starego wektora. Zwróć uwagę, że język w żaden sposób nie narzuca żadnej z opcji - kopie obiektów nie muszą być wierne.

Kopiujący operator przypisania

Kolejną specjalną metodą jest kopiujący operator przypisania. Wołany jest on w momencie, w którym do istniejącego obiektu `a` próbujemy przypisać wartość istniejącego obiektu `b`. Jego zdefiniowanie pozwala w bezpieczny sposób zwolnić zasoby, które mogą być trzymane przez `a`, zanim `a` skopiuje zasoby trzymane przez `b`. Zanim przećwiczymy to, zauważmy, że operator ten zwraca referencję typu klasy, dla której jest definiowany. Konkretnie, zwraca on referencję do obiektu, do którego nastąpiło przypisanie (czyli `a`). Celem tego zabiegu jest umożliwienie łączenia przypisań w jeden ciąg, np. `a = b = c = d;`. Jest on możliwy dzięki słowu kluczowemu `this`. `this` jest wskaźnikiem do obiektu, którego metoda jest wołana i można korzystać z niego w każdej metodzie klasy. Dodajmy na koniec, że zwracanie referencji do obiektu jest kwestią konwencji (w ten sposób postępuje też kompilator, jeżeli nie zdefiniujemy tego operatora), a nie obowiązkiem.

Zadanie 17 Dodaj do klasy `Wektor` kopiujący operator przypisania. Zadbaj o to, żeby nie nastąpił wyciek pamięci. Upewnij się też, że Twój kod działa poprawnie gdy użytkownik spróbuje przypisać obiekt sam do siebie (`a = a`). Logika takiego przypisania jest wątpliwa, natomiast jest ono formalnie dopuszczalne.

Semantyka przenoszenia

> Semantyka przenoszenia jest trudnym tematem, którego zapewne nie należy poruszać na 2. zajęciach. Niemniej jednak, jest ona logicznie powiązana z zagadnieniami zarządzania zasobami i metod specjalnych. Z tego powodu, autorzy umieścili zadania jej dotyczące poniżej. Polecamy jednak czytelnikowi pozostawienie ich wykonania na ostatnie zajęcia (instrukcja nr 7 jest odpowiednio krótsza). Na chwilę obecną, najważniejsze jest zrozumienie `new`, `delete`, referencji, list inicjalizacyjnych i kopiujących metod specjalnych.

Na koniec powiedzmy jeszcze o semantyce przenoszenia (ang. *move semantics*), obecnej w języku od standardu C++11. Stanowi ona dość obszerny temat, którego nie mamy niestety czasu omówić w pełni. Warto jest jednak mieć świadomość istnienia tej funkcjonalności i rozumieć ideę, która za nią stoi. Czytelników zainteresowanych pełniejszym opisem zagadnienia odsyłamy np. tutaj lub tutaj.

Przyjrzyjmy się poniższemu kawałkowi kodu:

```
struct S{ /* ... */ };

S getS() { return S{}; }

int main()
{
    S s;
    // Zrób coś z s...
    // Nie potrzebujemy już starej wartości s, poddajemy ją "recyklingow
    s = getS();
}
```

W C++98 i C++03, instrukcja `s = getS();` wiąże się z niepotrzebną kopią. Zastanówmy się, dlaczego tak jest. Jeżeli `S` jest klasą RAII, to ma miejsce następująca sekwencja wydarzeń:

1. Stworzenie zasobów wewnątrz `getS`
2. Zasoby zostają przypisane do bezimiennego, tymczasowego wyniku `getS`
3. Kopiujący operator przypisania klasy `S` przypisuje kopię zasobów tego wyniku do obiektu `s`
4. Zasoby bezimiennego wyniku zostają zwolnione w destruktorze
5. Podejmujemy pracę z `s`

Punkty 3. i 4. są niepotrzebne - naszym zamiarem było przypisanie zasobów od razu do `s`. Semantyka przenoszenia pozwala wyeliminować w takiej sytuacji nadmiarową kopię (bez żadnej modyfikacji funkcji `main`).

lvalue vs rvalue

Kluczowa dla problemu opisanego wyżej jest tymczasowa natura wyniku inwokacji funkcji `getS`. Okazuje się, że kompilator potrafi rozróżnić tego typu obiekty od tych "namacalnych", zadeklarowanych przez programistę (np. `s` z przykładu powyżej). Tego typu "namacalne" obiekty należą do kategorii **lvalue**, a te ulotne **rvalue**. Formalna definicja (wraz z kilkoma dodatkowymi elementami taksonomii obiektów w C++) dostępna jest w dokumentacji, lecz w bieżącym tekście oprzemy się jedynie na intuicji. Lvalue i rvalue biorą swoje nazwy od strony operatora `=`, po której zazwyczaj występują (*left* lub *right*). Jeżeli będziemy umieli rozróżnić te kategorie obiektów, to będziemy mogli napisać 2 różne funkcje (w przykładzie wyżej będą to 2 różne operatory przypisania), które będą wołane w zależności od kategorii argumentu. Dla lvalue postępować będziemy tak jak zwykle (tzn. wykonujemy kopię zasobów), a dla rvalue będziemy mogli "kraść" zasoby, gdyż mamy gwarancję, że obiekt jest tymczasowy i nikt poza nami i tak nie może ich wykorzystać. Konstruktem języka C++, który na to pozwala jest referencja rvalue (ang. *rvalue reference*, dalej RVR). RVR obiektu typu `T` zapisywane jest jako `T&&`. Spójrzmy, jak wygląda to w kodzie:

```
// Przeciążenie nr 1, zwykła referencja
void print_int(int& i)  { std::cout << "Ref: " << i << '\n'; };

// Przeciążenie nr 2, RVR
void print_int(int&& i) { std::cout << "RVR: " << i << '\n'; };

int getInt() { return 42; }

int main()
```

```
{
    int liczba = 314159; // liczba to lvalue

    print_int(liczba);   // przeciążenie 1
    print_int(getInt()); // przeciążenie 2
    print_int(13);       // przeciążenie 2, bo '13' to rvalue
}
```

W drugim przeciążeniu funkcji `print_int` pracujemy z argumentem normalnie - RVR nie wymaga od nas żadnych szczególnych operacji. Mamy za to gwarancję, że argument ten nie "ucieknie" z naszej funkcji, tzn. po zakończeniu wykonania naszej funkcji nikt inny nie będzie próbował go użyć. Możemy więc zrobić z nim co chcemy, np. zwolnić jego zasoby po ich wykorzystaniu. Podkreślmy też, że usunięcie jednego z powyższych przeciążeń spowoduje błąd kompilacji.

Jeżeli piszemy funkcję, dla której nie ma znaczenia, czy argument jest RVR czy LVL, wtedy używamy stałej (`const`) referencji:

```
void print_int(const int& i) { std::cout << "const ref: " << i << '\n';
```

Jeżeli zdefiniujemy tylko powyższe przeciążenie, `main` z powyższego przykładu skompiluje się poprawnie. Dzieje się tak dzięki temu, że stałe referencje rządzą się specjalnymi zasadami przedłużania życia (ang. *lifetime extension*), destrukcja obiektu na które wskazują odwlekana jest do czasu wyjścia ze scope'u referencji. W praktyce oznacza to, że poniższy kod jest poprawny

```
const int& i = getInt(); // OK, lifetime zwróconej wartości przedłużony
```

ale ten już nie

```
int& i = getInt(); // błąd kompilacji: próba przypisania RVR do LVR
```

Podsumowując, jako argumenty funkcji (w tym metod klas) możemy przyjąć:

- Stałe referencje (`const T&`), jeżeli nie potrzebujemy modyfikować danego argumentu, a jedynie dokonać jego inspekcji. Nie wykonujemy wtedy kopii obiektu. Ta opcja potrafi obsłużyć także sytuację, w której użytkownik poda do funkcji obiekt tymczasowy.
- Referencję lvalue (`T&`), jeżeli chcemy zmodyfikować w funkcji obiekt spoza niej. Tej opcji raczej nie stosujemy, gdyż może ona prowadzić do bugów (niechcący podajemy argument, którego wcale nie chcieliśmy modyfikować). Zamiast tego korzystamy z argumentów wyjściowych. `a = fun(a);` bardziej jawnie wyraża nasze intencje niż `fun(a);`
- Referencję rvalue (`T&&`), gdy chcemy obsłużyć sytuację, w której nasza funkcja przejmuje własność nad jakimś obiektem. Często stosujemy tę opcję obok innych przeciążeń (np. obok stałej referencji) jako optymalizacja dla szczególnego przypadku.

`std::move`

W C++ istnieje także sposób, aby zamienić referencję do lvalue na referencję do rvalue. Zobaczmy, dlaczego w ogóle moglibyśmy chcieć to zrobić:

```
struct S { /* duża klasa trzymająca zasobamy */ };
void fun(const S&) { /* ... */ } // przypadek ogólny
void fun(S&&) { /* ... */ }      // optymalizacja dla RVR

int main()
{
    S s;
    // Pracujemy z s...
    fun(s);
    // Teraz nie potrzebujemy s
    // Pracujemy dalej nad czymś innym...
}
```

W powyższym przykładzie zostanie wywołane przeciążenie `fun(const S&)`, gdyż `s` jest lvalue. Po zawołaniu funkcji `fun` zmienna `s` nie jest nam już jednak potrzebna. Chcielibyśmy zatem przenieść `s` do `fun` i pozwolić tej funkcji zutylizować zasoby trzymane przez `s` w sposób, który uzna za stosowny. Właśnie do tego służy funkcja `std::move` (z nagłówka `utility`).

```
// S i fun jak wyżej

int main()
{
    S s;
```

```
    fun(std::move(s));
    // s zostało przeniesione do fun
}
```

`std::move` zamienia referencję do lvalue na referencję do rvalue, co pozwala zawołać odpowiednie przeciążenie `fun`. Podkreślmy, że sama funkcja `std::move` nie potrafi w magiczny sposób dokonać transferu zasobów, za to odpowiedzialna jest już implementacja `fun`.

Po przesunięciu obiektu do funkcji lub innego obiektu (`a = std::move(b);`) pozostaje on w nieokreślonym, ale poprawnym stanie. Oznacza to, że po zawołaniu `std::move(s)`, nie wolno nam już korzystać z `s`! Jest to logiczne, gdyż `fun` przejęła własność nad tym obiektem, a zatem `main` nie może już go dotknąć. Zasadę tę musimy jednak stosować sami, kompilator nie potraktuje tego jako błąd (może co najwyżej wydać ostrzeżenie). Używanie obiektów, które zostały przesunięte stanowi przykład nieokreślonego zachowania (ang. *undefined behavior*), tzn. operacji, której efekt nie jest określony przez standard języka C++, a zatem konsekwencje mogą być dowolne (przeważnie złe), a także różnić się w zależności od kompilatora, platformy itp. Jedyne, co wolno nam zrobić dalej ze zmienną `s` to przypisać do niej nową wartość, wtedy możemy ponownie podjąć z nią pracę.

Uważny czytelnik zauważy, że od opisanej wyżej zasady obowiązuje jeden kluczowy wyjątek. Obiekt `s` został zadeklarowany w scope'ie funkcji `main`, a zatem przed jego opuszczeniem musi zostać zawołany jego destruktor. Funkcja `fun` musi zatem zadbać o to, żeby obiekt ten został pozostawiony w "zniszczalnym" stanie, np. poprzez wyzerowanie wewnętrznych wskaźników swojego argumentu (`double* a = nullptr; delete a;` jest w pełni poprawną operacją, która po prostu nic nie zrobi, ang. *no-op*). Z tego powodu często mówi się, że w C++ obowiązuje "nieniszczące przesunięcie" (ang. *nondestructive move*).

Przećwiczmy to w praktyce. Rozważmy klasę `S`, która dynamicznie alokuje zmienną typu `int`:

```
struct S
{
    S(int li) : liczba{new int{li}} { }
    ~S() { delete liczba; }

    int* liczba;
};
```


Chcielibyśmy napisać funkcję `pow2`, która przyjmuje obiekt typu `S` i zwraca obiekt tego samego typu, którego pole `liczba` jest kwadratem pola `liczba` argumentu. Wariant dla stałej referencji wygląda następująco:

```
S pow2(const S& arg)
{
    return S{*arg.liczba * *arg.liczba};
}
```

Następujący program zwróci 4:

```
int main()
{
    S s1{2};
    S s2 = pow2(s1);
    return *s2.liczba;
}
```

Mamy tylko jeden problem: w powyższym programie dokonujemy dwukrotnie alokacji pamięci. Zobaczmy, jak przy użyciu semantyki przenoszenia możemy pozbyć się jednej z alokacji. Dodajemy do `S` pusty (lub lepiej, zdefaultowany: `S() = default`) konstruktor domyślny oraz następujące przeciążenie funkcji `pow2`, które "kradnie" zasoby argumentu:

```
S pow2(S&& arg)
{
    // niezainicjalizowany wskaźnik
    S ret_val;

    // przepisujemy wskaźnik, nie ma alokacji
    ret_val.liczba = arg.liczba;

    // podnosimy liczbę do kwadratu
    *ret_val.liczba = *ret_val.liczba * *ret_val.liczba;

    // Zerujemy wskaźnik arg
    arg.liczba = nullptr;

    return ret_val;
}
```

Teraz możemy wykorzystać `std::move`, aby zaoszczędzić jedną alokację:

```
int main()
{
    S s1{2};
    S s2 = pow2(std::move(s1));
    return *s2.liczba;
}
```

Zauważmy, że gdybyśmy nie wyzerowali w funkcji `pow2(S&& arg)` wskaźnika zmiennej `arg`, to pod koniec funkcji `main` operator `delete` zostałby zawołany dwukrotnie na wskaźnikach do tego samego adresu w pamięci (przy destrukcji `s1` i `s2`), co spowodowałoby błąd programu. W praktyce, gdy korzystamy z klas napisanych przez kogoś innego, bardzo ciężko może być nam zrozumieć w jaki sposób zwolnić lub "ukraść" zasoby danego obiektu. Zamiast tego, korzystamy ze zdefiniowanych przez autorów danej klasy specjalnych metod: konstruktora przenoszącego i przenoszącego operatora przypisania.

Konstruktor przenoszący

Jak nietrudno się domyślić, konstruktor przenoszący pozwala skonstruować obiekt poprzez "pochłonięcie" innego obiektu tego samego typu. Sygnatura takiego konstruktora dla klasy `T` to `T(T&&)`. Gdyby klasa `S` z ostatniego przykładu miała taki konstruktor, moglibyśmy uprościć funkcję `pow2`:

```
S pow2(S&& arg)
{
    // konstruktor przenoszący wykonuje za nas całą pracę
    S ret_val {std::move(arg)};

    *ret_val.liczba = *ret_val.liczba * *ret_val.liczba;
    return ret_val;
}
```

Jest to zgodne z zasadą DRY (*don't repeat yourself*) - definiujemy konstruktor przenoszący raz, a następnie korzystamy z niego w wielu różnych miejscach. Nie musimy pamiętać za każdym razem o zerowaniu wskaźników itp. Dodatkowo nasz kod jest krótszy. Zwróćmy jeszcze uwagę, że pomimo tego, że `arg` jest podane jako RVR, musimy wewnątrz `pow2` ponownie zawołać `std::move`. Argument podany z zewnątrz jako RVR funkcjonuje wewnątrz jak LVL. Może to być nieco mylące, ale łatwo zapamiętać to w następujący sposób: przenosząc obiekt do każdej kolejnej funkcji musimy zawołać `std::move`, czyli jeżeli "zanurzamy" obiekt na głębokość n funkcji, to musimy zawołać `std::move` n razy.

Zadanie 18 Dodaj do klasy `Wektor` konstruktor przenoszący. Pamiętaj, że musisz zmodyfikować także obiekt **z którego** przenosisz tak, aby jego zniszczenie nie powodowało niepożądanych skutków ubocznych.

Przenoszący operator przypisania

Ostatnią szczególną metodą klas jest przenoszący operator przypisania, o sygnaturze `T& operator=(T&&)`. Przenosi on obiekt `b` na obiekt `a` (gdzie `a` i `b` nie muszą być różne).

Zadanie 19 Dodaj do klasy `Wektor` przenoszący operator przypisania. Pamiętaj o odpowiedniej modyfikacji obiektu **z którego** przenosisz oraz wykryciu przypadku, w którym obiekt przenoszony jest sam na siebie (`v = std::move(v)` nie powinno skutkować błędami).

Zasada 0, Zasada 5

Znamy już 5 szczególnych metod klas:

- konstruktor kopiujący
- kopiujący operator przypisania
- konstruktor przenoszący
- przenoszący operator przypisania
- destruktor

Jednymi z elementarnych zasad w programowaniu obiektowym w C++ są zasady zera i zasada pięciu. Zasada 0 mówi, że jeżeli nie chcemy wymusić żadnego szczególnego zachowania przy kopiowaniu, przenoszeniu lub niszczeniu obiektów, to nie należy definiować żadnej ze szczególnych metod i korzystać z tych, które zostaną domyślnie wygenerowane przez kompilator. Zasada 5 mówi z kolei, że jeżeli definiujemy choć jedną ze szczególnych metod, to powinniśmy także zdefiniować (lub zdefaultować,

jeżeli to możliwe) wszystkie pozostałe. Zasady te pomagają unikać bugów wynikających np. z nieświadomego zawołania kopiowania tam, gdzie możnaby jakiś obiekt przenieść.

Przydatna sztuczka: przyjmij kopię i przenieś

Na koniec pokażemy jeszcze często spotykany schemat, pozwalający zaoszczędzić kilka linijek kodu. Powiedzmy, że chcemy napisać klasę `ParaWektorow`, która wiąże ze sobą 2 obiekty typu `Wektor`. W najprostszym wydaniu, może wyglądać ona następująco:

```
struct ParaWektorow
{
    Wektor pierwszy;
    Wektor drugi;
};
```

Musimy teraz dopisać konstruktory. Aby w pełni wykorzystać optymalizacje opisane powyżej, możemy zdefiniować 4 konstruktory:

```
struct ParaWektorow
{
    ParaWektorow(const Wektor& w1, const Wektor& w2) : pierwszy{w1}, dru
    ParaWektorow(const Wektor& w1, Wektor&& w2) : pierwszy{w1}, drugi{st
    ParaWektorow(Wektor&& w1, const Wektor& w2) : pierwszy{std::move(w1)
    ParaWektorow(Wektor&& w1, Wektor&& w2) : pierwszy{std::move(w1)}, dr

    Wektor pierwszy;
    Wektor drugi;
};
```

Jest to nieco uciążliwe, a złożoność tego rozwiązania rośnie kombinatorycznie wraz z liczbą pól klasy. Zamiast tego, możemy zdefiniować tylko jeden konstruktor:

```
struct ParaWektorow
{
    ParaWektorow(Wektor w1, Wektor w2) : pierwszy{std::move(w1)}, drugi{

    Wektor pierwszy;
    Wektor drugi;
};

int main()
{
    Wektor w1 = getW1();
    Wektor w2 = getW2();
    ParaWektorow pw1{w1, w2};                          // *
    ParaWektorow pw2{std::move(w1), std::move(w2)}; // **
    // ...
}
```

Ceną, którą płacimy za takie rozwiązanie, jest dodatkowe wywołanie konstruktora *przenoszącego*. W linijce oznaczonej jedną gwiazdką, `w1` i `w2` są najpierw kopiowane, a następnie ich kopie są przenoszone do pól `pw1`. W linijce oznaczonej 2 gwiazdkami, `w1` i `w2` są najpierw przenoszone do konstruktora, a następnie dalej przenoszone do pól `pw2`. Koszt przenoszenia jest jednak bardzo niewielki, gdyż wiąże się jedynie z przestawieniem paru wskaźników, nie ma konieczności realokacji, ani kopiowania zawartości wektora. Rozwiązanie to jest zatem lepsze, gdyż zwiększa czytelność kodu i zmniejsza pole do popełnienia błędu.

Przykłady te ilustrują także bardzo dobrze działanie zasady 0. Dzięki odpowiedniemu zdefiniowaniu metod specjalnych klasy `Wektor`, możemy pozostawić stworzenie tych metod dla klasy `ParaWektorow` kompilatorowi. Ponownie, oszczędzamy pracy i nie dajemy sobie możliwości popełnienia błędu.

Zadanie na koniec Jeżeli nie jest dla Ciebie do końca jasne, kiedy wołany jest który konstruktor lub operator przenoszenia, nie przejmuj się. Pobaw się tym kawałkiem kodu - np. zakomentuj dla zawartej klasy semantykę przenoszenia, stwórz nowe obiekty i zobacz, jakie będą efekty (oraz w jakiej kolejności drukowane będą informacje). Być może pomocne okaże się także to nagranie (nie przejmuj się kodem obecnym na ekranie przez pierwszą minutę).