
Architektura oprogramowania bez tajemnic
Wykorzystaj język C++ do tworzenia wydajnych aplikacji i systemów
Adrian Ostrowski
Piotr Gaczkowski
Helion

Tytuł oryginału: Software Architecture with C++: Design modern systems using effective architecture concepts, design patterns, and techniques with C++20

Tłumaczenie: Krzysztof Bąbol

ISBN: 978-83-283-8666-2

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/aropbe
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Część IV. Zasady projektowania natywnego dla chmury

6

ROZDZIAŁ

Wzorce projektowe a język C++

Język C++ nie jest wyłącznie językiem obiektowym i oferuje coś więcej niż tylko dynamiczny polimorfizm, więc programowanie w nim nie polega jedynie na stosowaniu wzorców Gangu Czterech. W tym rozdziale poznasz często używane w języku C++ idiomy oraz wzorce projektowe i dowiesz się, gdzie się je stosuje.

W rozdziale zostaną poruszone następujące zagadnienia:

- pisanie idiomatycznego kodu C++,
- ciekawie rekurencyjny wzorzec szablonu (ang. *Curiously Recurring Template Pattern*),
- tworzenie obiektów,
- śledzenie stanu obiektów i przechodzenie do nich w języku C++,
- efektywna obsługa pamięci.

To spora lista! Nie marnujmy czasu i od razu zabierajmy się do pracy.

Wymagania techniczne

Zbudowanie i uruchomienie kodu z tego rozdziału wymaga następujących narzędzi:

- kompilatora z obsługą standardu C++20,
- programu CMake w wersji 3.15+.

Kody źródłowe z tego rozdziału znajdują się pod adresem *ftp://ftp.helion.pl/przyklady/aropbe.zip*, w katalogu *Rozdzial06*.

Pisanie idiomatycznego kodu C++

Jeśli znasz języki programowania obiektowego, z pewnością wiesz coś o wzorcach projektowych Gangu Czterech. Choć możliwe jest ich zaimplementowanie w C++ (i nierzadko jest to robione), w tym wieloparadygmatowym języku często stosuje się inne podejście do osiągania tych samych celów. Czasem koszty wirtualnej dyspozycji (ang. *virtual dispatch*) nie pozwalają pobić wydajności tzw. języków kawowych, takich jak Java czy C#. W wielu przypadkach programista wie jednak z góry, z jakimi typami będzie mieć do czynienia. W takich sytuacjach często możliwe jest pisanie bardziej wydajnego kodu dzięki narzędziom dostępnym w języku i bibliotece standardowej. Wyróżnia się wśród nich grupa, od której rozpoczniemy ten rozdział — idiomy języka. Rozpocznijmy naszą wyprawę od przyjrzenia się niektórym z nich.

Z definicji idiom to konstrukcja powtarzająca się w danym języku, swoiste dla niego wyrażenie. „Rodzimi użytkownicy" języka C++ powinni intuicyjnie znać jego idiomy. Wspomnieliśmy już o jednym z najczęściej używanych — inteligentnych wskaźnikach (ang. *smart pointers*). Omówmy teraz inny, podobny do niego.

Automatyzowanie operacji wyjścia z zakresu przy użyciu strażników RAII

Jednym z wyrażeń o największych możliwościach w języku C++ jest nawias klamrowy zamykający zakres. To właśnie tu zostają wywołane destruktory i mają miejsce tajemnicze działania związane z RAII. Aby je ujarzmić, nie trzeba korzystać z inteligentnych wskaźników. Wystarczy obiekt — strażnik RAII (ang. *RAII guard*). Podczas jego konstruowania zapamiętuje on, co ma robić, gdy będzie niszczony. Tym sposobem działanie zostanie wykonane automatycznie, niezależnie od tego, czy wyjście z zakresu nastąpi normalnie, czy na skutek wyjątku.

Najlepsze jest to, że strażnika RAII nie trzeba wcale pisać od zera, bo w rozmaitych bibliotekach istnieją już przetestowane implementacje. Jeśli korzystasz ze wspomnianej w poprzednim rozdziale biblioteki GSL, możesz użyć funkcji `gsl::finally()`. Weźmy pod uwagę następujący przykład:

```
using namespace std::chrono;

void self_measuring_function() {
  auto timestamp_begin = high_resolution_clock::now();

  auto cleanup = gsl::finally([timestamp_begin] {
    auto timestamp_end = high_resolution_clock::now();
    std::cout << "Wykonanie trwało: " <<
duration_cast<microseconds>(timestamp_end - timestamp_begin).count() << "
us";
  });
  // Wykonaj pracę.
  // Zgłoś wyjątek std::runtime_error{"Nieoczekiwany błąd"};
}
```

Pobieramy tutaj jeden znacznik czasu (ang. *timestamp*) na początku funkcji, a drugi na końcu. Spróbuj uruchomić ten przykład i zobacz, jak na jego wykonanie wpłynie odkomentowanie instrukcji `throw`. W obu przypadkach strażnik RAII wypisze poprawnie czas wykonania (przy założeniu, że wyjątek zostanie gdzieś przechwycony).

Omówmy teraz parę bardziej popularnych idiomów języka C++.

Zarządzanie możliwościami kopiowania i przenoszenia

Przy projektowaniu w języku C++ nowego typu należy zdecydować, czy powinien on być kopiowalny i przenaszalny (ang. *moveable*). Jeszcze ważniejsze jest poprawne zaimplementowanie tej semantyki w klasie. Omówmy teraz te zagadnienia.

Implementowanie typów niekopiowalnych

Istnieją przypadki, gdy klasa nie powinna być kopiowana, na przykład wtedy, gdy jest to bardzo kosztowne albo gdy może spowodować to błąd wywołany odcinaniem (ang. *slicing*). W przeszłości najczęstszym sposobem zapobiegania kopiowaniu takich obiektów było użycie idiomu niekopiowalności (ang. *non-copyable idiom*):

```
struct Noncopyable {
  Noncopyable() = default;
  Noncopyable(const Noncopyable&) = delete;
  Noncopyable& operator=(const Noncopyable&) = delete;
};

class MyType : NonCopyable {};
```

Warto jednak zauważyć, że taka klasa nie jest też przenaszalna, co łatwo przeoczyć podczas czytania jej definicji. Lepszym podejściem będzie po prostu jawne dodanie dwóch brakujących składowych (konstruktora przenoszącego, ang. *move constructor*, i przenoszącego operatora przypisania, ang. *move assignment operator*). Dobrą regułą przy deklarowaniu takich specjalnych funkcji składowych jest deklarowanie ich zawsze w komplecie. Oznacza to, że począwszy od standardu C++11, preferowany byłby następujący zapis:

```
struct MyTypeV2 {
  MyTypeV2() = default;
  MyTypeV2(const MyTypeV2 &) = delete;
  MyTypeV2 & operator=(const MyTypeV2 &) = delete;
  MyTypeV2(MyTypeV2 &&) = delete;
  MyTypeV2 & operator=(MyTypeV2 &&) = delete;
};
```

Tym razem składowe zostały zdefiniowane bezpośrednio w typie docelowym, a nie w pomocniczym typie `NonCopyable`.

Przestrzeganie zasad trzech i pięciu

Przy omawianiu specjalnych funkcji składowych warto wspomnieć o jeszcze jednej rzeczy: jeśli programista ich nie usuwa i dostarcza własne implementacje, najprawdopodobniej powinien zdefiniować ich komplet, łącznie z destruktorem. W standardzie C++98 reguła ta nosiła nazwę zasady trzech (ang. *rule of three*, bo trzeba było definiować trzy funkcje: konstruktor kopiujący, ang. *copy constructor*, kopiujący operator przypisania, ang. *copy assignment operator*, i destruktor), a od czasu wprowadzenia w C++11 operacji przenoszenia zastąpiono ją zasadą pięciu (ang. *rule of five*, dwiema dodatkowymi funkcjami są konstruktor przenoszący, ang. *move constructor*, i przenoszący operator przypisania, ang. *move assignment operator*). Stosowanie tych reguł pozwala uniknąć problemów z zarządzaniem zasobami.

Przestrzeganie zasady zera

Z drugiej strony, jeśli wystarczają Ci po prostu domyślne implementacje wszystkich specjalnych funkcji składowych, to nie deklaruj ich w ogóle. To jasny sygnał, że chcesz domyślnego działania. Jest to też najmniej mylące. Przyjrzyjmy się następującemu typowi:

```
class PotentiallyMisleading {
public:
  PotentiallyMisleading() = default;
  PotentiallyMisleading(const PotentiallyMisleading &) = default;
  PotentiallyMisleading &operator=(const PotentiallyMisleading &) =
default;
  PotentiallyMisleading(PotentiallyMisleading &&) = default;
  PotentiallyMisleading &operator=(PotentiallyMisleading &&) = default;
  ~PotentiallyMisleading() = default;

private:
  std::unique_ptr<int> int_;
};
```

Klasy tej nie można skopiować, mimo że wszystkim zmiennym nadaliśmy wartości domyślne, a to dlatego, że jej składowa typu `unique_ptr` nie jest kopiowalna. Na szczęście kompilator Clang ostrzega przed taką sytuacją, jednak GCC domyślnie tego nie robi. Lepszym podejściem byłoby zastosowanie zasady zera (ang. *rule of zero*) i napisanie następującego kodu:

```
class RuleOfZero {
  std::unique_ptr<int> int_;
};
```

Mamy teraz mniej standardowego kodu i po spojrzeniu na składową łatwiej dostrzec, że nie można jej kopiować.

Istnieje jeszcze jeden idiom związany z kopiowaniem, który warto poznać, i wkrótce go omówimy. Zanim to się jednak stanie, wspomnijmy o kolejnym idiomie, za pomocą którego można (i trzeba) implementować ten pierwszy.

Korzystanie z ukrytych funkcji zaprzyjaźnionych

Zasadniczo ukryte funkcje zaprzyjaźnione (ang. *hidden friends*) to funkcje nieskładowe zdefiniowane w obrębie typu, w którym zadeklarowano je jako zaprzyjaźnione. Dzięki temu nie można ich wywołać w inny sposób niż za pomocą wyszukiwania zależnego od argumentów (ang. *Argument-Dependent Lookup*, ADL), co w efekcie prowadzi do ich ukrycia. Ponieważ pozwalają ograniczyć liczbę przeciążeń branych pod uwagę przez kompilator, przyczyniają się do przyspieszenia kompilacji. Ich zaletą w porównaniu z alternatywnymi rozwiązaniami są też krótsze komunikaty o błędach. Ostatnią ciekawą własnością jest to, że nie da się ich wywołać, jeśli miałoby przy tym dojść do konwersji niejawnej. Pozwala to uniknąć takich przypadkowych konwersji.

Choć nie zaleca się na ogół stosowania w języku C++ funkcji zaprzyjaźnionych, inaczej sprawy się mają wtedy, gdy są one ukryte; jeśli wymienione w poprzednim akapicie zalety Cię nie przekonały, musisz też wiedzieć, że jest to zalecany sposób implementowania punktów dostosowywania (ang. *customization points*). Prawdopodobnie zastanawiasz się, co to takiego. Krótko mówiąc, są to elementy wywoływalne używane w kodzie biblioteki, które użytkownik może specjalizować w swoich typach. W bibliotece standardowej zarezerwowano dla nich całkiem sporo nazw, m.in. `begin`, `end`, wraz z ich odmianami odwrotnymi, i `const`, `swap`, `(s)size`, `(c)data` i wiele operatorów. Jeśli zdecydujesz się dostarczyć własne implementacje tych punktów dostosowywania, dobrze byłoby, gdyby działały one zgodnie z oczekiwaniami twórców biblioteki standardowej.

W porządku, na razie dosyć teorii. Zobaczmy, jak w praktyce dokonać specjalizacji punktu dostosowywania przy użyciu ukrytej funkcji zaprzyjaźnionej. Utwórzmy na przykład mocno uproszczoną klasę do zarządzania tablicami typów:

```
template <typename T> class Array {
public:
  Array(T *array, int size) : array_{array}, size_{size} {}

  ~Array() { delete[] array_; }

  T &operator[](int index) { return array_[index]; }
  int size() const { return size_; }

  friend void swap(Array &left, Array &right) noexcept {
    using std::swap;
    swap(left.array_, right.array_);
    swap(left.size_, right.size_);
  }

private:
  T *array_;
  int size_;
};
```

Jak widać, zdefiniowaliśmy destruktor, co oznacza, że powinniśmy dostarczyć również inne specjalne funkcje składowe. Zaimplementujemy je w następnym punkcie przy użyciu ukrytej zaprzyjaźnionej funkcji `swap`. Warto zauważyć, że choć została ona zadeklarowana wewnątrz klasy `Array`, wciąż jest to funkcja nieskładowa. Przyjmuje ona dwie instancje klasy `Array`, ale sama nie uzyskuje do nich dostępu.

Polecenie `using std::swap` sprawia, że kompilator najpierw szuka funkcji `swap` w przestrzeniach nazw zamienianych składowych. Jeśli jej nie znajdzie, wróci do korzystania z funkcji `std::swap`. Można to nazwać dwuetapowym idiomem wyszukiwania ADL i wyjścia zastępczego (ang. *two-step ADL and fallback idiom*) albo w skrócie idiomem dwuetapowym (ang. *two-step*), ponieważ najpierw uwidaczniamy `std::swap`, a potem wywołujemy funkcję `swap`. Słowo kluczowe `noexcept` informuje kompilator, że nasza funkcja `swap` nie zgłasza wyjątków, co pozwala w pewnych sytuacjach na generowanie szybszego kodu. Dlatego też należy zawsze oznaczać tym słowem kluczowym nie tylko funkcję `swap`, ale też konstruktory: domyślny i przenoszący.

Skoro już mamy funkcję `swap`, wykorzystajmy ją w celu użycia w klasie `Array` jeszcze jednego idiomu.

Zapewnianie bezpieczeństwa wyjątkowego przy użyciu idiomu „skopiuj i przestaw" (copy-and-swap)

Jak wspomnieliśmy w poprzednim punkcie, z uwagi na to, że w klasie `Array` zdefiniowaliśmy destruktor, zgodnie z regułą pięciu powinniśmy też zdefiniować inne specjalne funkcje składowe. W tym punkcie zostanie przedstawiony idiom pozwalający to robić bez powielania kodu, a dodatkowo zapewniający silne bezpieczeństwo pod względem wyjątków.

Dla osób nieobeznanych z bezpieczeństwem wyjątkowym przypomnijmy krótko, na jakich poziomach mogą je zapewniać funkcje i typy:

- *Brak gwarancji* (ang. *no guarantees*): to najbardziej podstawowy poziom. Nie ma żadnych gwarancji dotyczących stanu obiektu, jeśli w czasie korzystania z niego zostanie zgłoszony wyjątek.
- *Podstawowe bezpieczeństwo wyjątkowe* (ang. *basic exception safety*): możliwe są efekty uboczne, ale w obiekcie nie nastąpi wyciek zasobów, pozostanie on w poprawnym stanie i będzie zawierać poprawne dane (niekoniecznie te same co przed operacją). Twoje typy powinny zawsze mieć co najmniej ten poziom bezpieczeństwa.
- *Wysokie bezpieczeństwo wyjątkowe* (ang. *strong exception safety*): nie wystąpią efekty uboczne. Stan obiektu będzie taki sam jak przed operacją.
- *Gwarancja niezgłaszania wyjątku* (ang. *no-throw guarantee*): operacje zawsze kończą się powodzeniem. Jeśli podczas operacji wystąpi wyjątek, zostanie on przechwycony i obsłużony wewnętrznie, tak że nie przedostanie się na zewnątrz. Takie operacje można oznaczać słowem kluczowym `noexcept`.

Czy jest jakiś sposób, by mieć dwa wróble w garści, czyli pisać składowe specjalne bez powielania kodu i zapewnić im wysokie bezpieczeństwo wyjątkowe? Tak naprawdę to całkiem łatwe. Skoro mamy funkcję `swap`, użyjmy jej w implementacji operatorów przypisania:

```
Array &operator=(Array other) noexcept {
  swap(*this, other);
  return *this;
}
```

W tym przypadku jeden operator wystarczy zarówno do przypisania kopiującego, jak i przenoszącego. W przypadku kopiowania pobieramy parametr jako wartość, więc w tym miejscu wykonywana jest kopia tymczasowa. Jedyne, co potem musimy zrobić, to zamiana składowych. Mamy nie tylko wysokie bezpieczeństwo wyjątkowe, ale obyliśmy się też bez zgłaszania wyjątków w treści operatora. Wyjątek nadal może jednak być zwrócony tuż przed wywołaniem funkcji, gdy przeprowadzane będzie kopiowanie. W przypadku przypisania przenoszącego nie dokonuje się kopii, bo przy pobieraniu przez wartość przyjmowany jest po prostu przenoszony obiekt.

Teraz zdefiniujmy konstruktor kopiujący:

```
  Array(const Array &other) : array_{new T[other.size_]},
size_{other.size_} {
    std::copy_n(other.array_, size_, array_);
  }
```

Ten „koleś” może zgłosić wyjątek zarówno z powodu użycia typu `T`, jak i dlatego, że alokuje pamięć. Zdefiniujmy teraz także konstruktor przenoszący:

```
  Array(Array &&other) noexcept
      : array_{std::exchange(other.array_, nullptr)},
size_{std::exchange(other.size_, 0)} {}
```

Użyliśmy tutaj funkcji `std::exchange` po to, by zainicjować nasze składowe, a wyczyścić składowe obiektu `other`, a wszystko to na liście inicjacyjnej. Konstruktor został zadeklarowany ze słowem kluczowym `noexcept` ze względu na wydajność. Na przykład w klasie `std::vector` w razie zwiększenia się liczby elementów można je przenosić tylko wtedy, gdy w ich konstruktorze przenoszącym występuje specyfikator `noexcept`, w przeciwnym wypadku zostaną one skopiowane.

To by było na tyle. Niewielkim wysiłkiem i bez duplikacji kodu utworzyliśmy klasę `Array` zapewniającą wysokie bezpieczeństwo wyjątkowe.

Zajmijmy się teraz kolejnym idiomem języka C++, który można zauważyć w kilku miejscach biblioteki standardowej.

Pisanie niebloidów

Niebloidy (ang. *niebloids*), nazwane tak na cześć Erica Nieblera, to rodzaj obiektów funkcyjnych stosowanych jako punkty dostosowywania od czasu standardu C++17. Wzrost ich popularności nastąpił po wprowadzeniu zakresów standardowych, opisanych w rozdziale 5., „Wykorzystywanie cech języka C++", ale po raz pierwszy zostały zaproponowane przez Nieblera jeszcze w 2014 r. *Ich celem jest wyłączanie mechanizmu ADL wtedy, gdy nie jest pożądany, tak by kompilator nie brał pod uwagę przeciążeń z innych przestrzeni nazw.* Pamiętasz idiom dwuetapowy (ang. *two-step idiom*) z poprzednich punktów? Ponieważ jego stosowanie jest niewygodne i łatwo o nim zapomnieć, wprowadzono pojęcie *obiektów-punktów dostosowywania* (ang. *customization point objects*). Zasadniczo są to obiekty funkcyjne wykonujące automatycznie te dwa etapy.

Jeśli Twoje biblioteki mają mieć punkty dostosowywania, prawdopodobnie dobrze byłoby zaimplementować je przy użyciu niebloidów. Nie bez powodu, począwszy od standardu C++17, w ten sposób implementowane są wszystkie punkty dostosowywania. Zastanów się nad użyciem niebloidów nawet wtedy, gdy chcesz po prostu utworzyć obiekt funkcyjny. Zapewniają one wszystko to, co dobre w wyszukiwaniu ADL, a ograniczają jego wady. Możliwa jest ich specjalizacja, a w połączeniu z koncepcjami pozwolą Ci dopasować do swoich potrzeb zestaw przeciążeń Twoich elementów wykonywalnych. Umożliwiają też lepsze dostosowanie algorytmów, a wszystko to niewielkim kosztem obszerniejszego niż zwykle kodu.

W tym punkcie utworzymy prosty algorytm zakresowy, który zaimplementujemy w postaci niebloidu. Nazwijmy go `contains` (zawiera), bo będzie po prostu zwracał wartość logiczną wskazującą, czy dany element znajduje się w zakresie, czy nie. Najpierw utwórzmy sam obiekt funkcyjny, zaczynając od deklaracji jego opartego na iteratorach operatora wywołania:

```
namespace detail {
struct contains_fn final {
  template <std::input_iterator It, std::sentinel_for<It> Sent, typename T,
            typename Proj = std::identity>
  requires std::indirect_binary_predicate<
      std::ranges::equal_to, std::projected<It, Proj>, const T *> constexpr
bool
  operator()(It first, Sent last, const T &value, Proj projection = {})
const {
```

Kod ten wygląda na rozwlekły, ale wszystko ma swój cel. Strukturę oznaczamy słowem kluczowym `final` po to, by pomóc kompilatorowi generować bardziej efektywny kod. Jeśli spojrzysz na parametry szablonu, będą tam iterator i wartownik (ang. *sentinel*) — podstawowe elementy konstrukcyjne każdego zakresu standardowego. Wartownikiem jest często iterator, ale może nim być każdy typ półregularny (ang. *semiregular*), który da się porównać z iteratorem (typ półregularny to taki, który można kopiować i inicjować w sposób domyślny). Dalej `T` jest typem szukanego elementu, a `Proj` oznacza projekcję — operację wykonywaną na każdym elemencie zakresu przed jego porównaniem (domyślna wartość `std::identity` powoduje po prostu przekazanie wejścia na wyjście).

Po parametrach szablonu określone są ich wymagania; operator wymaga, byśmy mogli porównać pod kątem równości wartość będącą wynikiem projekcji z wartością wyszukiwaną. Po określeniu ograniczeń podajemy po prostu parametry funkcji.

Przyjrzyjmy się, jak to zostało zaimplementowane:

```
    while (first != last && std::invoke(projection, *first) != value)
      ++first;
    return first != last;
}
```

Przechodzimy tu po prostu po elementach, dokonując projekcji każdego z nich i porównując je z wyszukiwaną wartością. Jeśli wartość ta zostanie znaleziona, zwracamy `true`, a jeśli nie — `false` (gdy `first == last`).

Powyższa funkcja działałaby nawet wtedy, gdybyśmy nie używali standardowych zakresów; przydałoby się jeszcze przeciążenie specjalnie dla nich. Jego deklaracja może być następująca:

```
template <std::ranges::input_range Range, typename T,
          typename Proj = std::identity>
requires std::indirect_binary_predicate<
    std::ranges::equal_to,
    std::projected<std::ranges::iterator_t<Range>, Proj>,
    const T *> constexpr bool
operator()(Range &&range, const T &value, Proj projection = {}) const {
```

Tym razem przyjmujemy jako parametry szablonu: typ spełniający koncepcję `input_range`, wartość elementu i typ projekcji. Podobnie jak wcześniej wymagamy, by iterator zakresu po wywołaniu projekcji mógł być porównywany pod względem równości z obiektami typu `T`. Na koniec używamy zakresu, wartości i projekcji jako parametrów przeciążenia.

Implementacja tego operatora będzie także dość prosta:

```
    return (*this)(std::ranges::begin(range), std::ranges::end(range), value,
                   std::move(projection));
  }
};
}  // przestrzeń nazw detail
```

Wywołujemy po prostu poprzednie przeciążenie przy użyciu iteratora i wartownika z podanego zakresu, a wartość i projekcję przekazujemy niezmienione. Ostatnią rzeczą do zrobienia będzie zdefiniowanie niebloidu `contains`, a nie tylko wywoływalnego elementu `contains_fn`:

```
inline constexpr detail::contains_fn contains{};
```

Dzięki zadeklarowaniu ze specyfikatorem `inline` zmiennej typu `contains_fn` o nazwie `contains` niebloid będzie można wywołać przy użyciu nazwy zmiennej. Wywołajmy go teraz sami, by sprawdzić, czy to działa:

```
int main() {
  auto ints = std::ranges::views::iota(0) | std::ranges::views::take(5);

  return contains(ints, 42);
}
```

I to by było tyle. Funktor blokujący wyszukiwanie ADL działa zgodnie z założeniami.

Jeśli uważasz, że cały ten kod jest trochę zbyt rozwlekły, pewnie zainteresuje Cię funkcja `tag_invoke`, która być może w przyszłości stanie się częścią standardu. W podrozdziale „Materiały dodatkowe" znajdziesz poświęcony temu artykuł oraz klip wideo z serwisu YouTube, w którym doskonale wyjaśniono mechanizm ADL, niebloidy, ukryte funkcje zaprzyjaźnione i `tag_invoke`.

Przejdźmy teraz do kolejnego przydatnego idiomu języka C++.

Idiom klas parametryzowanych wytycznymi

Klasy parametryzowane wytycznymi (ang. *policy-based design*) przedstawił po raz pierwszy Andrei Alexandrescu w doskonałej książce *Nowoczesne projektowanie w C++*. Choć została ona wydana w 2001 r., wiele przedstawionych w niej idei jest w użyciu po dziś dzień. Zachęcamy do jej przeczytania; łącze znajduje się w końcowym podrozdziale „Materiały dodatkowe". Idiom klas parametryzowanych wytycznymi jest w zasadzie odpowiednikiem czasu kompilacji dla przedstawionego przez Gang Czterech wzorca Strategia. Jeśli potrzebna jest klasa o adaptowalnym działaniu, można napisać ją w postaci szablonu, którego parametrami będą odpowiednie wytyczne. Przykładem z życia mogą być standardowe alokatory, które są przekazywane w postaci wytycznych do wielu kontenerów C++ jako ostatnie parametry szablonów.

Powróćmy do klasy `Array` i dodajmy wytyczną do wypisywania informacji debugowania:

```
template <typename T, typename DebugPrintingPolicy = NullPrintingPolicy>
class Array {
```

Możemy zastosować domyślną wytyczną, która niczego nie wypisze. Typ `NullPrintingPolicy` można zaimplementować następująco:

```
struct NullPrintingPolicy {
  template <typename... Args> void operator()(Args...) {}
};
```

Jak widać, niezależnie od podanych argumentów struktura ta nic nie robi. Kompilator w trakcie optymalizacji całkowicie ją usunie, więc jeśli nie stosuje się wypisywania informacji debugowania, nie będzie żadnego narzutu.

Jeśli chcemy, by klasa była nieco bardziej „gadatliwa", możemy użyć innej wytycznej:

```
struct CoutPrintingPolicy {
  void operator()(std::string_view text) { std::cout << text << std::endl;
}
};
```

Tym razem tekst przekazany do zasady wypisujemy po prostu do strumienia `cout`. Musimy też zmodyfikować klasę, by rzeczywiście korzystała z tej wytycznej:

```
  Array(T *array, int size) : array_{array}, size_{size} {
    DebugPrintingPolicy{}("konstruktor");
  }

  Array(const Array &other) : array_{new T[other.size_]},
size_{other.size_} {
    DebugPrintingPolicy{}("konstruktor kopiujący");
    std::copy_n(other.array_, size_, array_);
  }

  // ...inne składowe...
```

Wywołujemy po prostu składową `operator()` wytycznej, przekazując do niej tekst do wypisania. Ponieważ wytyczne są bezstanowe, możemy w razie potrzeby tworzyć ich instancje bez dodatkowych kosztów. Alternatywą może być wywoływanie z poziomu wytycznej funkcji statycznej.

Teraz wystarczy tylko utworzyć instancję klasy `Array` z żądaną wytyczną i można ją stosować:

```
Array<T, CoutPrintingPolicy>(new T[size], size);
```

Jedną z wad używania wytycznych ustalanych w czasie kompilacji jest to, że konkretyzacje szablonów za pomocą różnych wytycznych mają różne typy. Oznacza to na przykład, że przypisania ze zwykłej klasy `Array` do takiej z wytyczną `CoutPrintingPolicy` wymagają więcej pracy. W tym celu operatory przypisania muszą być zaimplementowane jako funkcje szablonowe, w których wytyczne są parametrem szablonu.

Czasami alternatywą dla wytycznych może być korzystanie z cech (ang. *traits*). Weźmy na przykład klasę `std::iterator_traits`, pozwalającą uzyskać różne informacje na temat iteratorów przy pisaniu algorytmów z ich zastosowaniem. Przykładem może być funkcja `std::iterator_traits<T>::value_type`, działająca zarówno z niestandardowymi iteratorami o zdefiniowanej składowej `value_type`, jak i prostymi, na przykład wskaźnikami (w tym przypadku `value_type` odnosi się do typu wskazywanego).

To tyle na temat klas parametryzowanych wytycznymi. Następną pozycją na liście będzie idiom, który daje duże możliwości i jest stosowany w wielu sytuacjach.

Ciekawie rekurencyjny wzorzec szablonu

Mimo mylącej nazwy **ciekawie rekurencyjny wzorzec szablonu** (ang. *Curiously Recurring Template Pattern*, CRTP) to nie wzorzec, ale idiom języka C++. Używa się go między innymi do implementacji innych idiomów i wzorców projektowych oraz do wprowadzania statycznego polimorfizmu. Zacznijmy od tego ostatniego zastosowania, a inne opiszemy później.

Rozeznanie, kiedy używać polimorfizmu dynamicznego, a kiedy statycznego

Kiedy wspomina się o polimorfizmie, wielu programistów myśli o polimorfizmie dynamicznym, w którym informacja potrzebna do wywołania funkcji jest uzyskiwana w czasie wykonania programu. Polimorfizm statyczny z kolei polega na ustalaniu wywołań w czasie kompilacji. Zaletą tego pierwszego jest to, że listę typów można modyfikować w czasie wykonania, co pozwala na rozszerzanie hierarchii klas za pomocą wtyczek i bibliotek. Wielką zaletą drugiego jest to, że można uzyskać lepszą wydajność, jeśli typy są z góry znane. Oczywiście w pierwszym przypadku czasem można oczekiwać, że kompilator zdewirtualizuje wywołania, ale nie zawsze można na to liczyć. Z kolei w drugim przypadku wydłuża się czas kompilacji.

Wygląda na to, że nie da się wygrać na wszystkich frontach. Mimo to dobranie właściwego typu polimorfizmu dla typów może wymagać sporo czasu. Jeśli zagrożona jest wydajność, zdecydowanie rekomendujemy, by rozważyć użycie statycznego polimorfizmu. Stosuje się w tym celu idiom CRTP.

Wiele wzorców projektowych można zaimplementować w jeden albo w drugi sposób. Ponieważ nie zawsze warto ponosić koszt dynamicznego polimorfizmu, wzorce projektowe Gangu Czterech w języku C++ nie zawsze są najlepszymi rozwiązaniami. Jeśli chcesz mieć możliwość rozszerzania hierarchii typów w czasie wykonania, a czasy kompilacji są dla Ciebie dużo większym problemem niż wydajność (i nie planujesz w najbliższym czasie używać modułów), wtedy klasyczne implementacje wzorców Gangu Czterech mogą być odpowiednie. W przeciwnym razie możesz spróbować zaimplementować je przy użyciu statycznego polimorfizmu albo prostszych rozwiązań dostosowanych do języka C++. Niektóre z nich omówimy w tym rozdziale. Chodzi w tym wszystkim o to, by wybrać do pracy najlepsze narzędzie.

Implementowanie polimorfizmu statycznego

Zaimplementujmy teraz statycznie polimorficzną hierarchię klas. Potrzebujemy bazowej klasy szablonowej:

```
template <typename ConcreteItem> class GlamorousItem {
public:
  void appear_in_full_glory() {
```

```
    static_cast<ConcreteItem *>(this)->appear_in_full_glory();
  }
};
```

Parametrem szablonu klasy bazowej jest klasa pochodna. Z początku może się to wydać dziwne, ale pozwala to w funkcji interfejsu, która w tym przypadku nosi nazwę `appear_in_full_glory`, za pomocą operatora `static_cast` dokonać konwersji na właściwy typ. Następnie wywołujemy implementację tej funkcji w klasie pochodnej. Klasy pochodne można zaimplementować następująco:

```
class PinkHeels : public GlamorousItem<PinkHeels> {
public:
  void appear_in_full_glory() {
    std::cout << "Różowe szpilki pojawiły się raptem w całej krasie\n";
  }
};

class GoldenWatch : public GlamorousItem<GoldenWatch> {
public:
  void appear_in_full_glory() {
    std::cout << "Każdy chciał obejrzeć ten zegarek\n";
  }
};
```

Każda z tych klas dziedziczy po klasie bazowej `GlamorousItem` i jest w niej użyta jako argument szablonu. W każdej też zaimplementowano wymaganą funkcję.

Warto zauważyć, że — inaczej niż w przypadku dynamicznego polimorfizmu — klasa bazowa użyta w idiomie CRTP jest szablonem, więc każda klasa pochodna będzie miała inny typ bazowy. Oznacza to, że nie da się łatwo utworzyć kontenera dla klasy bazowej `GlamorousItem`. Można jednak zrobić kilka innych rzeczy:

- Zapisywać klasy pochodne w krotce (ang. *tuple*).
- Utworzyć z nich `std::variant`.
- Dodać jedną wspólną klasę do opakowywania wszystkich konkretyzacji klasy bazowej. Ta klasa też może być wariantem.

W pierwszym przypadku klasy możemy użyć w następujący sposób. Najpierw utworzymy krotkę konkretyzacji klasy bazowej:

```
template <typename... Args>
using PreciousItems = std::tuple<GlamorousItem<Args>...>;

auto glamorous_items = PreciousItems<PinkHeels, GoldenWatch>{};
```

W krotce, dla której utworzyliśmy alias typu, będziemy mogli przechowywać dowolne zbytkowne artykuły. Teraz wystarczy wywoływać ciekawą funkcję:

```
std::apply(
    []<typename... T>(GlamorousItem<T>... items) {
        (items.appear_in_full_glory(), ...); },
    glamorous_items);
```

Ponieważ próbujemy iterować po krotce, najprostszym na to sposobem będzie użycie funkcji `std::apply`, która wykona dany element wywoływalny na rzecz wszystkich elementów danej krotki. W tym przypadku elementem wywoływalnym będzie wyrażenie lambda, które przyjmuje tylko klasę bazową `GlamorousItem`. Aby funkcja została wywołana na rzecz wszystkich elementów, korzystamy z wprowadzonych w standardzie C++17 wyrażeń zwijania (ang. *fold expressions*).

Gdybyśmy mieli użyć wariantu zamiast krotki, musielibyśmy skorzystać z funkcji `std::visit`:

```
  using GlamorousVariant = std::variant<PinkHeels, GoldenWatch>;
  auto glamorous_items = std::array{GlamorousVariant{PinkHeels{}},
GlamorousVariant{GoldenWatch{}}};
  for (auto& elem : glamorous_items) {
    std::visit([]<typename T>(GlamorousItem<T> item){
      item.appear_in_full_glory(); }, elem);
  }
```

Funkcja `std::visit` przyjmuje po prostu wariant i wywołuje na rzecz zapisanego w nim obiektu przekazane do niej wyrażenie lambda. Tworzymy tutaj tablicę wariantów zbytkownych artykułów po to, by móc po niej przechodzić jak po każdym innym kontenerze i przy odwiedzaniu każdego wariantu wywoływać odpowiednie wyrażenie lambda.

Jeśli wydaje Ci się, że z punktu widzenia użytkownika interfejsu jest to nieintuicyjne, możesz rozważyć inne podejście, polegające na opakowaniu wariantu w kolejną klasę, w tym przypadku o nazwie `CommonGlamorousItem`:

```
class CommonGlamorousItem {
public:
  template <typename T> requires std::is_base_of_v<GlamorousItem<T>, T>
  explicit CommonGlamorousItem(T &&item)
      : item_{std::forward<T>(item)} {}
private:
  GlamorousVariant item_;
};
```

Aby skonstruować opakowanie, używamy konstruktora przekazującego (ang. *forwarding constructor*, jego parametrem jest pochodzące z szablonu wyrażenie `T&&`). Zamiast przenosić to wyrażenie, przekazujemy je dalej w celu utworzenia opakowującego wariantu `item_`. Przenosimy w ten sposób tylko wejściowe wartości prawostronne (ang. *r-value inputs*). Ograniczamy również parametry szablonu, więc z jednej strony opakowywać będziemy jedynie klasę bazową `GlamorousItem`, a z drugiej nasz szablon nie zostanie użyty jako konstruktor kopiujący ani przenoszący.

Musimy też opakować funkcję składową:

```
void appear_in_full_glory() {
  std::visit(
      []<typename T>(GlamorousItem<T> item) {
          item.appear_in_full_glory(); },
      item_);
}
```

Tym razem wywołanie funkcji `std::visit` jest szczegółem implementacji. Programista może używać tej klasy opakowującej następująco:

```
auto glamorous_items = std::array{CommonGlamorousItem{PinkHeels{}},
                                  CommonGlamorousItem{GoldenWatch{}}};
    for (auto& elem : glamorous_items) {
      elem.appear_in_full_glory();
    }
```

Podejście to pozwala użytkownikowi klasy pisać zrozumiały kod przy zachowaniu wydajności statycznego polimorfizmu.

Aby zapewnić podobny komfort programowania, jednak przy gorszej wydajności, można też użyć techniki zwanej wymazywaniem typów, którą zaraz omówimy.

Przerywnik — wymazywanie typów

Choć wymazywanie typów (ang. *type erasure*) nie ma związku z idiomem CRTP, świetnie pasuje do wcześniejszego przykładu — i dlatego właśnie tu je przedstawiamy.

Idiom wymazywania typów polega na ukrywaniu konkretnego typu pod polimorficznym interfejsem. Doskonały przykład tego podejścia przedstawił Sean Parent w wykładzie pt. *Inheritance Is The Base Class of Evil* na konferencji GoingNative 2013. Gorąco zachęcamy do obejrzenia go w wolnym czasie; łącze znajduje się w podrozdziale „Materiały dodatkowe". W bibliotece standardowej idiom ten można spotkać m.in. w klasie `std::function`, dealokatorze typu `std::shared_ptr` i klasie `std::any`.

Wygoda użycia i wszechstronność mają swoją cenę — w idiomie tym używane są wskaźniki i wirtualna dyspozycja (ang. *virtual dispatch*), co powoduje, że użycie wspomnianych narzędzi z biblioteki standardowej jest niekorzystne wtedy, gdy wymagana jest wydajność. Trzeba na to uważać.

Aby wprowadzić w naszym przykładzie wymazywanie typów, nie potrzebujemy już idiomu CRTP. Tym razem w klasie `GlamorousItem` dynamicznie polimorficzne obiekty zostaną opakowane we wskaźnik inteligentny:

```
class GlamorousItem {
public:
  template <typename T>
  explicit GlamorousItem(T t)
      : item_{std::make_unique<TypeErasedItem<T>>(std::move(t))} {}

  void appear_in_full_glory() { item_->appear_in_full_glory_impl(); }

private:
  std::unique_ptr<TypeErasedItemBase> item_;
};
```

Tym razem przechowujemy wskaźnik do klasy bazowej (`TypeErasedItemBase`), który będzie wskazywał pochodne tej klasy opakowujące artykuły (`TypeErasedItem<T>`). Klasę bazową zdefiniujemy następująco:

```
struct TypeErasedItemBase {
  virtual ~TypeErasedItemBase() = default;
  virtual void appear_in_full_glory_impl() = 0;
};
```

Interfejs ten należy też zaimplementować w każdej pochodnej klasie opakowującej:

```
  template <typename T> class TypeErasedItem final : public
TypeErasedItemBase {
  public:
    explicit TypeErasedItem(T t) : t_{std::move(t)} {}
    void appear_in_full_glory_impl() override { t_.appear_in_full_glory();
}

  private:
    T t_;
  };
```

Interfejs klasy bazowej jest implementowany poprzez wywoływanie funkcji z opakowywanego obiektu. Warto zauważyć, że idiom ten nosi nazwę „wymazywania typów" dlatego, że klasa `GlamorousItem` nie ma informacji o tym, jaki typ `T` faktycznie opakowuje. Informacja o typie zostaje usunięta podczas konstruowania klasy `TypeErasedItem`, ale wszystko działa, a to dlatego, że w typie `T` zaimplementowano wymagane metody.

Konkretne artykuły można zaimplementować prościej:

```
class PinkHeels {
public:
  void appear_in_full_glory() {
    std::cout << "Różowe szpilki pojawiły się raptem w całej krasie\n";
  }
};

class GoldenWatch {
public:
  void appear_in_full_glory() {
    std::cout << "Każdy chciał obejrzeć ten zegarek\n";
  }
};
```

Tym razem nie muszą one być dziedziczone po żadnej klasie bazowej. Wystarczy kacze typowanie (ang. *duck typing*) — jeśli coś kwacze jak kaczka, prawdopodobnie nią jest, a jeśli coś można pokazać w całej krasie, zapewne jest zbytkowne.

Interfejsu API z wymazywaniem typów używa się następująco:

```
auto glamorous_items =
    std::array{GlamorousItem{PinkHeels{}}, GlamorousItem{GoldenWatch{}}};
for (auto &item : glamorous_items) {
  item.appear_in_full_glory();
}
```

Utworzyliśmy właśnie tablicę obiektów opakowujących, po której przechodzimy, stosując prostą semantykę opartą na wartościach. Wydaje się to najbardziej przyjemne w użyciu, bo polimorfizm — jako szczegół implementacji — jest ukryty przed wywołującym.

Dużą wadą tego podejścia jest jednak, jak wcześniej wspomnieliśmy, słaba wydajność. Wymazywanie typów ma swoją cenę, więc należy je stosować wstrzemięźliwie i z pewnością nie na ścieżce aktywnej (ang. *hot path*).

Skoro opisaliśmy, jak opakowywać i wymazywać typy, przejdźmy do omówienia ich tworzenia.

Tworzenie obiektów

W tym podrozdziale omówimy typowe rozwiązania problemów związanych z tworzeniem obiektów. Omówimy różne typy fabryk obiektów, przestudiujemy budowniczych oraz poruszymy temat kompozytów i prototypów. Zastosujemy jednak trochę odmienne podejście, niż przy opisywaniu swoich rozwiązań przyjął Gang Czterech. Za właściwe implementacje swoich wzorców uznał on złożone, dynamicznie polimorficzne hierarchie klas. W świecie C++ wiele praktycznych problemów można rozwiązać za pomocą wzorców bez wprowadzania aż tylu klas i bez narzutu dynamicznego dysponowania (ang. *dynamic dispatch*). Dlatego też przedstawione przez nas implementacje będą inne, często prostsze lub wydajniejsze (choć bardziej wyspecjalizowane, a w odczuciu Gangu Czterech zapewne mniej „ogólne"). Przejdźmy od razu do analizy.

Korzystanie z fabryk

Pierwszym typem wzorców kreacyjnych, które tu omówimy, będą fabryki. Są one użyteczne wtedy, gdy konstrukcję obiektu można przeprowadzić w jednym kroku (wzorzec przydatny w sytuacji, gdy jest to niemożliwe, zostanie omówiony zaraz po fabrykach), ale sam konstruktor po prostu do tego nie wystarczy. Istnieją trzy typy fabryk: metody fabrykujące (ang. *factory methods*), funkcje fabrykujące (ang. *factory functions*) i klasy-fabryki (ang. *factory classes*). Przedstawmy je po kolei.

Korzystanie z metod fabrykujących

Metody fabrykujące, zwane też *nazwanymi konstruktorami* (ang. *named constructor idiom*), to zasadniczo funkcje składowe, które wywołują prywatny konstruktor. Kiedy się ich używa? Oto kilka scenariuszy:

- *Gdy istnieje wiele różnych sposobów konstruowania obiektu, które mogłyby z dużym prawdopodobieństwem doprowadzić do błędu.* Wyobraź sobie na przykład konstruowanie klasy do przechowywania różnych kanałów koloru danego piksela; każdy kanał jest reprezentowany przez wartość jednobajtową. W razie użycia samego konstruktora bardzo łatwo o przekazanie do niego niewłaściwej kolejności kanałów albo wartości przewidzianych dla zupełnie

innej przestrzeni barwnej. Również zmiana wewnętrznej reprezentacji kolorów piksela dość szybko stałaby się skomplikowana. Można argumentować, że do reprezentowania kolorów w tych różnych formatach powinny służyć osobne typy, ale często poprawnym podejściem jest też użycie metody fabrykującej.

- *Gdy chcesz wymusić tworzenie obiektu na stercie lub w innym specyficznym obszarze pamięci.* Skorzystanie z metody fabrykującej może być dobre wtedy, gdy obiekt zajmuje dużo miejsca na stosie i programista obawia się, że zabraknie tam pamięci. Sprawy mają się podobnie również wtedy, gdy wszystkie instancje muszą być tworzone w pewnym obszarze pamięci urządzenia.
- *Jeśli konstruowanie obiektu może się nie udać, a nie wolno zgłaszać wyjątków.* Na ogół należy korzystać z wyjątków, a nie z innych metod obsługi błędów. Dzięki poprawnemu użyciu wyjątków można uzyskać czystszy i bardziej wydajny kod. Niektóre projekty albo środowiska wymagają jednak ich wyłączenia. W takich przypadkach użycie metody fabrykującej pozwala na raportowanie błędów, które wystąpią podczas konstruowania.

Metoda fabrykująca w pierwszym opisanym przez nas przypadku mogłaby wyglądać następująco:

```
class Pixel {
public:
  static Pixel fromRgba(char r, char b, char g, char a) {
    return Pixel{r, g, b, a};
  }
  static Pixel fromBgra(char b, char g, char r, char a) {
    return Pixel{r, g, b, a};
  }

  // inne składowe

private:
  Pixel(char r, char g, char b, char a) : r_(r), g_(g), b_(b), a_(a) {}
  char r_, g_, b_, a_;
}
```

Klasa ta ma dwie metody fabrykujące (tak naprawdę w standardzie C++ nie uznaje się terminu *metoda*, a używa się pojęcia *funkcja składowa*, ang. *member function*): `fromRgba` i `fromBgra`. Teraz trudniej popełnić błąd i zainicjować kanały w niewłaściwej kolejności.

Zauważ, że istnienie konstruktora prywatnego skutecznie uniemożliwia dziedziczenie typu przez inne klasy, bo bez dostępu do konstruktora nie można utworzyć żadnych instancji. Jeśli jest to Twoim celem, a nie efektem ubocznym, preferuj jednak oznaczanie swoich klas słowem kluczowym `final`.

Korzystanie z funkcji fabrykujących

Zamiast korzystać ze składowych funkcji wytwórczych, fabryki możemy też implementować przy użyciu funkcji niesładowych. Ten sposób zapewnia lepszą hermetyzację. Opisał to Scott Meyers w artykule, do którego łącze znajduje się w podrozdziale „Materiały dodatkowe”.

W przypadku typu `Pixel` możemy również utworzyć funkcję swobodną fabrykującą jego instancje. Uprości to kod tego typu:

```
struct Pixel {
  char r, g, b, a;
};

Pixel makePixelFromRgba(char r, char g, char b, char a) {
  return Pixel{r, g, b, a};
}

Pixel makePixelFromBgra(char b, char g, char r, char a) {
  return Pixel{r, g, b, a};
}
```

To podejście sprawia, że konstrukcja jest zgodna z zasadą otwarty-zamknięty (ang. *open-closed principle*), opisaną w rozdziale 1., „Znaczenie architektury oprogramowania i zasady dobrego projektowania". Dodanie kolejnych funkcji fabrykujących dla innych przestrzeni barwnych jest łatwe i nie wymaga modyfikowania samej struktury `Pixel`.

Ta implementacja typu `Pixel` pozwala inicjować go ręcznie, zamiast korzystać z jednej z dostarczonych funkcji. Jeśli chcemy, możemy to uniemożliwić przez zmianę deklaracji klasy. Oto jak może ona wyglądać po poprawce:

```
struct Pixel {
  char r, g, b, a;

private:
  Pixel(char r, char g, char b, char a) : r(r), g(g), b(b), a(a) {}
  friend Pixel makePixelFromRgba(char r, char g, char b, char a);
  friend Pixel makePixelFromBgra(char b, char g, char r, char a);
};
```

Tym razem funkcje fabrykujące są zaprzyjaźnione z klasą. Typ nie jest już jednak agregatem, więc nie możemy już korzystać z inicjacji agregatowej (`Pixel{}`) ani też z inicjatorów wyznaczonych (ang. *designated initializers*). Nie możemy również przestrzegać zasady otwarty-zamknięty. Te dwa podejścia mają różne plusy i minusy, więc należy dokonać między nimi rozsądnego wyboru.

Wybór typu zwracanego z fabryki

Kolejną rzeczą, którą należy wybrać przy implementowaniu fabryki obiektów, jest faktyczny typ, jaki ma ona zwracać. Omówmy teraz te zagadnienia.

W przypadku typu `Pixel`, który jest wartościowy i niepolimorficzny, najlepsze będzie najprostsze podejście — zwracamy go po prostu przez wartość. Jeśli tworzysz typ polimorficzny, zwracaj go poprzez wskaźnik inteligentny (*nigdy* nie używaj do tego gołego wskaźnika, bo spowoduje to w pewnym momencie wycieki pamięci). Jeśli wywołujący powinien być właścicielem utworzonego obiektu, zwykle najlepszym podejściem będzie zwrócenie go we wskaźniku `unique_ptr` do klasy bazowej. W niezbyt często spotykanych wypadkach,

gdy obiekt musi mieć na własność zarówno fabryka, jak i kod wywołujący, użyj wskaźnika `shared_ptr` albo innej alternatywy opartej na zliczaniu referencji. Czasami wystarczy, by fabryka śledziła obiekt, ale go nie przechowywała. W takich przypadkach wewnątrz fabryki przechowuj wskaźnik `weak_ptr`, a na zewnątrz zwróć `shared_ptr`.

Niektórzy programiści C++ mogą argumentować, że pewne typy należy zwracać poprzez parametr wyjściowy, ale w większości przypadków nie jest to najlepsze podejście. Jeśli wziąć pod uwagę wydajność, zwykle najlepiej jest zwracać obiekt przez wartość, bo kompilator nie tworzy jego dodatkowych kopii. Jeśli sedno sprawy leży w tym, że typ, począwszy od C++17, nie jest kopiowalny, to w standardzie określono, gdzie pomijanie kopiowania (ang. *copy elision*) jest obowiązkowe, więc zwracanie takich typów przez wartość zwykle nie jest problemem. Jeśli funkcja zwraca wiele obiektów, skorzystaj z pary (ang. *pair*), krotki (ang. *tuple*), struktury lub kontenera.

Jeśli podczas konstruowania coś pójdzie nie tak, masz kilka możliwości:

- Zwróć szablon `std::optional` dla Twojego typu, jeśli nie ma potrzeby dostarczać wywołującemu komunikatu o błędzie.
- Zgłoś wyjątek, jeśli błędy podczas konstruowania są rzadkie i powinny być przekazane dalej.
- Zwróć szablon `absl::StatusOr` dla Twojego typu, jeśli błędy podczas konstruowania są częste (dokumentacja biblioteki Abseil na temat tego szablonu znajduje się w podrozdziale „Materiały dodatkowe”).

Skoro już wiesz, co trzeba zwracać, omówmy ostatni typ fabryk.

Korzystanie z klas-fabryk

Klasy-fabryki (ang. *factory classes*) to typy, które wytwarzają obiekty. Pozwalają one odłączyć typy obiektów polimorficznych od ich wywołujących. Umożliwiają też korzystanie z pul obiektów (ang. *object pools*, w których utrzymywane są często używane obiekty, aby nie trzeba było wciąż przydzielać im pamięci i ją zwalniać) oraz innych schematów alokacji. To tylko kilka przykładów na to, do czego przydają się klasy-fabryki. Przyjrzyjmy się jeszcze innemu przykładowi. Załóżmy, że musimy tworzyć różne typy polimorficzne w zależności od parametrów wejściowych. W niektórych przypadkach polimorficzna funkcja fabrykująca z poniższego listingu okazuje się niewystarczająca:

```
std::unique_ptr<IDocument> open(std::string_view path) {
    if (path.ends_with(".pdf")) return std::make_unique<PdfDocument>();
    if (name == ".html") return std::make_unique<HtmlDocument>();

    return nullptr;
}
```

Co by było, gdybyśmy chcieli otwierać także innego rodzaju dokumenty, na przykład pliki tekstowe OpenDocument? Może się okazać, o ironio, że powyższa otwarta fabryka nie jest otwarta na rozszerzanie. Jeśli sami odpowiadamy za kod, nie będzie to wielkim problemem, ale może się nim stać, gdyby użytkownicy biblioteki chcieli rejestrować własne typy.

Aby rozwiązać tę kwestię, użyjmy klasy-fabryki, która pozwoli na rejestrowanie funkcji do otwierania różnorodnych dokumentów:

```
class DocumentOpener {
public:
  using DocumentType = std::unique_ptr<IDocument>;
  using ConcreteOpener = DocumentType (*)(std::string_view);

private:
  std::unordered_map<std::string_view, ConcreteOpener> openerByExtension;
};
```

Klasa ta jeszcze wiele nie robi, ale zawiera mapę przyporządkowującą rozszerzenia funkcjom, która ma być wywoływana do otwierania plików danego typu. Dodamy teraz dwie publiczne funkcje składowe. Pierwsza będzie rejestrowała nowe typy plików:

```
  void Register(std::string_view extension, ConcreteOpener opener) {
    openerByExtension.emplace(extension, opener);
  }
```

Mamy teraz sposób na zapełnienie mapy. Druga z nowych funkcji publicznych będzie otwierała dokumenty przy użyciu odpowiedniego elementu otwierającego (ang. *opener*):

```
  DocumentType open(std::string_view path) {
    if (auto last_dot = path.find_last_of('.');
        last_dot != std::string_view::npos) {
      auto extension = path.substr(last_dot + 1);
      return openerByExtension.at(extension)(path);
    } else {
      throw std::invalid_argument{"Próba otwarcia pliku bez
rozszerzenia"};
    }
  }
```

Zasadniczo wyodrębniamy ze ścieżki pliku rozszerzenie, zgłaszamy wyjątek, jeśli jest ono puste, a jeśli nie, szukamy w mapie elementu otwierającego. Jeżeli zostanie znaleziony, otwieramy za jego pomocą podany plik, a jeśli nie, mapa zgłosi nam kolejny wyjątek.

Teraz możemy utworzyć instancję fabryki i zarejestrować własne typy plików, na przykład format tekstowy OpenDocument:

```
auto document_opener = DocumentOpener{};

document_opener.Register(
    "odt", [](auto path) -> DocumentOpener::DocumentType {
      return std::make_unique<OdtDocument>(path);
    });
```

Zwróć uwagę, że rejestrujemy wyrażenie lambda, bo może ono być skonwertowane na typ `ConcreteOpener`, który jest wskaźnikiem do funkcji. Nie dałoby się jednak tego zrobić, gdyby wyrażenie lambda miało stan. W takiej sytuacji musielibyśmy użyć czegoś do opakowania. Jedną z takich rzeczy może być typ `std::function`, ale jego wadą jest konieczność ponoszenia

kosztów wymazywania typu (ang. *type erasure*) za każdy razem, gdy zechcielibyśmy uruchomić funkcję. W przypadku otwierania plików prawdopodobnie jest to dopuszczalne. Jeśli jednak potrzebujesz lepszej wydajności, zastanów się nad użyciem typu takiego jak `function_ref`.

Przykładowa implementacja tego narzędzia zaproponowanego do włączenia do standardu C++ (jeszcze nie zaakceptowanego) znajduje się w repozytorium Sy Brand w serwisie GitHub, do którego łącze znajduje się w podrozdziale „Materiały dodatkowe".

W porządku, skoro element otwierający został już zarejestrowany w fabryce, otwórzmy za jego pomocą plik i wyodrębnijmy z niego trochę tekstu:

```
auto document = document_opener.open("file.odt");
std::cout << document->extract_text().front();
```

I to wszystko! Jeśli chcesz umożliwić użytkownikom swojej biblioteki rejestrowanie własnych typów, muszą oni mieć dostęp do mapy w czasie wykonania. Możesz dostarczyć im API pozwalające na dostęp do niej albo uczynić fabrykę statyczną i pozwolić rejestrować typy w dowolnym miejscu kodu.

W ten sposób w jednym kroku załatwiliśmy sprawę fabryk i budowania obiektów. Omówmy kolejny popularny wzorzec, używany wtedy, gdy fabryki nie są odpowiednie.

Korzystanie z budowniczych

Budowniczy, podobnie jak fabryka, to wzorzec kreacyjny pochodzący od Gangu Czterech. W przeciwieństwie do fabryk budowniczy pozwalają budować bardziej złożone obiekty, takie których nie da się zbudować w jednym kroku, na przykład typy złożone z wielu oddzielnych części: Pozwalają też na dostosowywanie operacji konstruowania obiektów do własnych potrzeb. W naszym przypadku pominiemy projektowanie złożonych hierarchii budowniczych, a zamiast tego pokażemy, w jaki sposób budowniczy może pomóc. Implementowanie hierarchii zostawimy Ci jako ćwiczenie.

Budowniczy są konieczni wtedy, gdy obiekt nie może być wytworzony w jednym kroku, ale z powodu płynnego interfejsu przyjemnie się ich używa również wtedy, gdy ten jeden krok nie jest trywialny. Zademonstrujmy tworzenie hierarchii płynnych budowniczych przy użyciu idiomu CRTP.

W tym przypadku utworzymy za pomocą CRTP bazowego budowniczego `GenericItemBuilder` i bardziej wyspecjalizowanego `FetchingItemBuilder`, zdolnego do pobierania danych ze zdalnego adresu, jeśli taka funkcja jest obsługiwana. Takie specjalizacje mogą się znajdować nawet w różnych bibliotekach i na przykład korzystać z różnych interfejsów API, które mogą, ale nie muszą być dostępne w czasie budowania.

Dla zademonstrowania zagadnienia będziemy budować instancje struktury `Item` z rozdziału 5., „Wykorzystywanie cech języka C++":

```
struct Item {
  std::string name;
  std::optional<std::string> photo_url;
  std::string description;
  std::optional<float> price;
  time_point<system_clock> date_added{};
  bool featured{};
};
```

Jeśli chcesz, możesz wymusić, by instancje struktury `Item` były budowane przy użyciu budowniczego, przez uczynienie konstruktora domyślnego prywatnym i oznaczenie budowniczych jako zaprzyjaźnionych:

```
template <typename ConcreteBuilder> friend class GenericItemBuilder;
```

Implementację budowniczego można zacząć w następujący sposób:

```
template <typename ConcreteBuilder> class GenericItemBuilder {
public:
  explicit GenericItemBuilder(std::string name)
      : item_{.name = std::move(name)} {}
protected:
  Item item_;
```

Chociaż tworzenie chronionych składowych nie jest z reguły zalecane, chcemy, by pochodni budowniczy mieli dostęp do artykułów. Alternatywą może być używanie w pochodnych budowniczych wyłącznie publicznych metod budowniczego bazowego.

W konstruktorze budowniczego przyjmujemy nazwę artykułu, bo jest to jedyna jego składowa pochodząca od użytkownika, której należy nadać wartość przy tworzeniu artykułu. Dzięki temu mamy pewność, że zostanie ona ustawiona. Alternatywnie można by sprawdzać, czy jest w porządku, na ostatnim etapie, gdy obiekt jest zwalniany do użytkownika. W naszym przypadku etap budowania można zaimplementować następująco:

```
Item build() && {
  item_.date_added = system_clock::now();
  return std::move(item_);
}
```

Gdy metoda ta jest wywoływana, wymuszamy „skonsumowanie" budowniczego; musi on być wartością prawostronną (ang. *r-value*). Oznacza to, że możemy jej użyć jednowierszowo albo przesunąć ją na ostatni krok, by zaznaczyć, że budowniczy kończy działanie. Następnie ustawiamy czas utworzenia artykułu i przenosimy go poza budowniczego.

W interfejsie API budowniczego można udostępnić na przykład następujące funkcje:

```
ConcreteBuilder &&with_description(std::string description) {
  item_.description = std::move(description);
  return static_cast<ConcreteBuilder &&>(*this);
}
```

```
  ConcreteBuilder &&marked_as_featured() {
    item_.featured = true;
    return static_cast<ConcreteBuilder &&>(*this);
  }
```

Każda z nich zwraca konkretny (pochodny) obiekt budowniczego jako referencję do wartości prawostronnej (ang. *r-value reference*). Być może jest to nieintuicyjne, ale tym razem lepiej nie zwracać obiektu przez wartość, tylko użyć takiego typu zwracanego. Pozwala to uniknąć podczas budowania tworzenia niepotrzebnych kopii obiektu `item_`. Z kolei zwracanie referencji do wartości lewostronnych (ang. *l-value reference*) prowadziłoby do powstania wiszących referencji (ang. *dangling references*) i utrudniałoby wywoływanie funkcji `build()`, bo zwracana referencja do l-wartości nie odpowiadałaby oczekiwanej (takiej do r-wartości).

Finalny typ budowniczego może wyglądać następująco:

```
class ItemBuilder final : public GenericItemBuilder<ItemBuilder> {
  using GenericItemBuilder<ItemBuilder>::GenericItemBuilder;
};
```

Jest to po prostu klasa, w której wykorzystuje się konstruktory z uogólnionego budowniczego. Używa się jej następująco:

```
auto directly_loaded_item = ItemBuilder{"Garnek"}
                                .with_description("Porządny")
                                .with_price(100)
                                .build();
```

Jak widać, ostatecznie w interfejsie funkcje wywołuje się kaskadowo, a korzystanie z nazw metod sprawia, że całe wywołanie jest płynne. Stąd wzięła się nazwa *płynnych interfejsów* (ang. *fluent interfaces*).

A co by się stało, gdybyśmy nie ładowali każdego artykułu bezpośrednio, ale użyli bardziej wyspecjalizowanego budowniczego, który ładowałby partie danych ze zdalnego punktu końcowego? Moglibyśmy zdefiniować go następująco:

```
class FetchingItemBuilder final
    : public GenericItemBuilder<FetchingItemBuilder> {
public:
  explicit FetchingItemBuilder(std::string name)
      : GenericItemBuilder(std::move(name)) {}

  FetchingItemBuilder&& using_data_from(std::string_view url) && {
    item_ = fetch_item(url);
    return std::move(*this);
  }
};
```

Użycie idiomu CRTP pozwala dziedziczyć po uogólnionym budowniczym i wyegzekwować podanie nazwy artykułu. Tym razem jednak rozszerzamy bazowego budowniczego własną funkcją do pobierania treści i wstawiania jej do budowanego artykułu. Dzięki idiomowi

CRTP, gdy wywołamy funkcję z naszego bazowego budowniczego, zwracany jest budowniczy pochodny, co znacznie ułatwia korzystanie z interfejsu. Można go wywoływać w następujący sposób:

```
auto fetched_item =
    FetchingItemBuilder{"Lniana bluzka"}
        .using_data_from("https://example.com/items/linen_blouse")
        .marked_as_featured()
        .build();
```

To bardzo ładne i eleganckie!

Budowniczy przydają się również wtedy, gdy trzeba zawsze tworzyć niezmienne (ang. *immutable*) obiekty. Ponieważ budowniczy ma dostęp do prywatnych składowych klasy, może je modyfikować nawet wtedy, gdy w klasie nie ma dla nich żadnych funkcji ustawiających (ang. *setters*). To oczywiście niejedyny przypadek, gdy użycie budowniczych może być korzystne.

Budowanie w kompozytach i prototypach

Przypadkiem, w którym należałoby użyć budowniczego, jest tworzenie kompozytu. Kompozyt to wzorzec projektowy, w którym grupa obiektów jest traktowana jak jeden, a wszystkie mają ten sam interfejs (albo ten sam typ bazowy). Przykładem niech będzie graf, który może się składać z podgrafów, albo dokument, w którym mogą być zagnieżdżone inne dokumenty. Kiedy na rzecz takiego obiektu wywoływana jest funkcja `print()`, w celu wypisania całego kompozytu uruchomione zostają funkcje `print()` wszystkich podobiektów. Wzorzec Budowniczy przydaje się przy tworzeniu każdego z podobiektów i przy ich kompozycji.

Kolejnym wzorcem używanym do konstruowania obiektów jest Prototyp. Używa się go wtedy, gdy budowanie obiektu od podstaw jest bardzo kosztowne albo po prostu programista chce mieć bazowy obiekt, na podstawie którego buduje inne. Wzorzec ten można sprowadzić do zapewnienia sposobu klonowania obiektu, którego potem używa się takim, jaki jest, albo modyfikuje się go po to, by stał się taki, jaki być powinien. W przypadku hierarchii polimorficznej wystarczy dodać funkcję `clone()` w następujący sposób:

```
class Map {
public:
    virtual std::unique_ptr<Map> clone() const;
    // ...inne składowe...
};
class MapWithPointsOfInterests {
public:
    std::unique_ptr<Map> clone() override const;
    // ...inne składowe...
private:
    std::vector<PointOfInterest> pois_;
};
```

W obiekcie `MapWithPointsOfInterests` również można klonować punkty orientacyjne, więc nie musimy ręcznie dodawać każdego z nich. Pozwala to zapewnić wartości domyślne użytkownikowi końcowemu, gdy będzie tworzył własną mapę. Warto też zwrócić uwagę, że w niektórych przypadkach nie trzeba korzystać z prototypu, a wystarczy zwykły konstruktor kopiujący.

Omówiliśmy już tworzenie obiektów. Po drodze wspomnieliśmy o wariantach, więc czemu by do nich nie wrócić i zobaczyć, czy nie mogą nam pomóc w inny sposób?

Śledzenie stanu i odwiedzanie obiektów w języku C++

Stan (ang. *State*) to wzorzec projektowy mający pomagać w modyfikacji działania obiektu, gdy zmienia się jego stan wewnętrzny. Podejmowane w poszczególnych stanach działania powinny być od siebie niezależne, tak by dodanie nowego stanu nie miało wpływu na już istniejące. Najprostsze podejście, polegające na zaimplementowaniu wszystkich działań w obiekcie stanowym, nie jest skalowalne ani otwarte na rozszerzanie. Użyciu wzorca Stan pozwala dodawać nowe działania przez wprowadzanie nowych klas stanów i definiowanie tranzycji między nimi. W tym podrozdziale pokażemy, w jaki sposób implementować stany i maszynę stanów (ang. *state machine*) z wykorzystaniem klasy `std::variant` i podwójnego polimorfizmu statycznego (ang. *statically polymorphic double dispatch*). Innymi słowy, budujemy automat skończony przez połączenie wzorców Stan i Wizytator (ang. *Visitor*) w sposób właściwy dla języka C++.

Najpierw zdefiniujmy stany. W naszym przykładzie będziemy modelować stan produktu w magazynie. Może on być następujący:

```
namespace state {

struct Depleted {};

struct Available {
  int count;
};

struct Discontinued {};
} // przestrzeń nazw state
```

Poszczególne stany mogą mieć swoje właściwości, na przykład liczbę dostępnych artykułów. Odmiennie niż w przypadku polimorfizmu dynamicznego, nie muszą też dziedziczyć po jednym typie bazowym. Zamiast tego, jak widać poniżej, wszystkie przechowuje się w jednym wariancie:

```
using State = std::variant<state::Depleted, state::Available,
state::Discontinued>;
```

Oprócz stanów potrzebujemy też zdarzeń do przeprowadzania tranzycji pomiędzy stanami. Spójrz na poniższy kod:

```
namespace event {

struct DeliveryArrived {
  int count;
};

struct Purchased {
  int count;
};

struct Discontinued {};

} // przestrzeń nazw event
```

Jak widać, zdarzenia też mogą mieć właściwości i nie dziedziczą po wspólnym typie. Teraz musimy zaimplementować tranzycje pomiędzy stanami. Można to zrobić następująco:

```
State on_event(state::Available available, event::DeliveryArrived
delivered) {
  available.count += delivered.count;
  return available;
}

State on_event(state::Available available, event::Purchased purchased) {
  available.count -= purchased.count;
  if (available.count > 0)
    return available;
  return state::Depleted{};
}
```

Jeśli dokonano zakupu, stan może się zmienić, ale niekoniecznie. Do obsługi kilku stanów naraz możemy użyć szablonów:

```
template <typename S> State on_event(S, event::Discontinued) {
  return state::Discontinued{};
}
```

Jeśli artykuł zostanie wycofany, nieważne, jaki wcześniej był jego stan. Dobra, zaimplementujmy teraz ostatnią obsługiwaną tranzycję:

```
State on_event(state::Depleted depleted, event::DeliveryArrived delivered)
{
  return state::Available{delivered.count};
}
```

Kolejnym niezbędnym elementem układanki jest znalezienie sposobu na uogólnione zdefiniowanie wielu operatorów wywołania w jednym obiekcie, tak by można było wywołać najbardziej pasujące przeciążenie. Będziemy tego potrzebowali później do wywoływania zdefiniowanych dopiero co tranzycji. Ten pomocnik może wyglądać następująco:

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

Przy użyciu szablonów o zmiennej liczbie argumentów (ang. *variadic templates*), wyrażenia zwijania (ang. *fold expression*) i podpowiedzi dedukcyjnej dla argumentu szablonu klasy (ang. *class template argument deduction guide*) tworzymy strukturę `overload`, która podczas konstruowania udostępni nam wszystkie przekazane do niej operatory wywołania. Z dokładniejszymi wyjaśnieniami i alternatywną implementacją odwiedzania można się zapoznać we wpisie na anglojęzycznym blogu Bartłomieja Filipka, pod adresem podanym w podrozdziale „Materiały dodatkowe".

Możemy już przystąpić do implementowania samej maszyny stanów:

```
class ItemStateMachine {
public:
  template <typename Event> void process_event(Event &&event) {
    state_ = std::visit(overload{
        [&](const auto &state) requires std::is_same_v<
            decltype(on_event(state, std::forward<Event>(event))), State> {
          return on_event(state, std::forward<Event>(event));
        },
        [](const auto &unsupported_state) -> State {
          throw std::logic_error{"Nieobsługiwana tranzycja między stanami"};
        }
      },
      state_);
  }

private:
  State state_;
};
```

Funkcja `process_event` przyjmie którekolwiek ze zdefiniowanych zdarzeń. W celu przejścia do nowego stanu będzie wywoływała odpowiednią funkcję `on_event` z użyciem bieżącego stanu i przekazanego zdarzenia. Jeśli dla danego stanu i zdarzenia zostanie znalezione przeciążenie `on_event`, dojdzie do wywołania pierwszego wyrażenia lambda. W przeciwnym wypadku ograniczenie nie zostanie spełnione i wywołane będzie drugie, bardziej ogólne przeciążenie. Tym samym, jeśli tranzycja między stanami nie jest obsługiwana, po prostu zgłosimy wyjątek.

Teraz zapewnijmy możliwość raportowania bieżącego stanu:

```
std::string report_current_state() {
  return std::visit(
      overload{[](const state::Available &state) -> std::string {
                 return std:: "Dostępne sztuki: " +
                 to_string(state.count);
               },
               [](const state::Depleted) -> std::string {
                 return "Artykuł jest tymczasowo niedostępny.";
               },
               [](const state::Discontinued) -> std::string {
                 return "Artykuł został wycofany.";
               }},
      state_);
}
```

Przy użyciu struktury `overload` przekazujemy tutaj trzy wyrażenia lambda, z których każde zwraca ciąg raportu generowanego przy odwiedzaniu obiektu stanu.

Możemy już uruchomić całe rozwiązanie:

```
auto fsm = ItemStateMachine{};
std::cout << fsm.report_current_state() << '\n';
fsm.process_event(event::DeliveryArrived{3});
std::cout << fsm.report_current_state() << '\n';
fsm.process_event(event::Purchased{2});
std::cout << fsm.report_current_state() << '\n';
fsm.process_event(event::DeliveryArrived{2});
std::cout << fsm.report_current_state() << '\n';
fsm.process_event(event::Purchased{3});
std::cout << fsm.report_current_state() << '\n';
fsm.process_event(event::Discontinued{});
std::cout << fsm.report_current_state() << '\n';
// fsm.process_event(event::DeliveryArrived{1});
```

Po jego uruchomieniu uzyskamy następujące dane wyjściowe:

```
Artykuł jest tymczasowo niedostępny.
Dostępne sztuki: 3
Dostępne sztuki: 1
Dostępne sztuki: 3
Artykuł jest tymczasowo niedostępny.
Artykuł został wycofany.
```

Chyba że zostanie odkomentowany ostatni wiersz z nieobsługiwaną tranzycją, bo wtedy na końcu zostanie zgłoszony wyjątek.

To rozwiązanie jest dużo bardziej wydajne niż takie oparte na polimorfizmie dynamicznym, choć lista obsługiwanych stanów i zdarzeń jest ograniczona do tych zdefiniowanych w czasie kompilacji. Aby uzyskać więcej informacji na temat stanów, wariantów i rozmaitych sposobów wizytacji, zapoznaj się z wykładem Mateusza Pusza z konferencji CppCon 2018, również wymienionym w podrozdziale „Materiały dodatkowe”.

Ostatnią rzeczą, o której chcielibyśmy opowiedzieć, zanim zamkniemy ten rozdział, jest obsługa pamięci. Rozpocznijmy więc ostatni podrozdział.

Efektywne postępowanie z pamięcią

Wykorzystaniu pamięci warto się przyglądać nawet wtedy, gdy jej ilość nie jest bardzo ograniczona. Przepływność (ang. *throughput*) pamięci stanowi zwykle ograniczenie wydajności współczesnych systemów, więc dobre wykorzystanie pamięci jest zawsze istotne. Dokonywanie zbyt wielu dynamicznych alokacji spowalnia program i prowadzi do fragmentacji pamięci. Poznajmy kilka sposobów na to, jak radzić sobie z tymi problemami.

Ograniczenie dynamicznych alokacji dzięki optymalizacji SSO/SOO

Dynamiczne alokacje mogą czasem powodować inne problemy niż tylko zgłaszanie wyjątków podczas konstruowania obiektów z powodu braku wystarczającej ilości pamięci. Często zabierają one cykle procesora i mogą powodować fragmentację pamięci. Na szczęście istnieje sposób, by się przed tym uchronić. Każdy, kto używał typu `std::string` (począwszy od GCC 5.0), najprawdopodobniej korzystał z techniki zwanej **optymalizacją krótkich łańcuchów** (ang. *Small String Optimization*, SSO). Stanowi ona przykład bardziej ogólnej optymalizacji zwanej **optymalizacją małych obiektów** (ang. *Small Object Optimization*, SOO), z którą można się spotkać m.in. w typie `InlinedVector` z biblioteki Abseil. Zasadniczo pomysł jest dość prosty: jeśli obiekt jest odpowiednio mały, powinien być przechowywany wewnątrz klasy, do której należy, a nie alokowany dynamicznie. W przypadku typu `std::string` przechowywane są zwykle pojemność (ang. *capacity*) i długość ciągu oraz sam łańcuch. Jeśli jest on wystarczająco krótki (w przypadku kompilatora GCC na platformach 64-bitowych do 15 bajtów), będzie on zapisany w jednej ze składowych.

Przechowywanie obiektów na miejscu zamiast alokowania ich gdzie indziej i zapisywania tylko wskaźnika daje jeszcze jedną korzyść: mniej pogoni za wskaźnikami. Każdy dostęp do danych ukrytych za wskaźnikiem zwiększa obciążenie pamięci podręcznych procesora i ryzyko tego, że trzeba będzie ściągnąć dane z pamięci głównej. Jeśli taka sytuacja powtarza się często, może mieć wpływ na ogólną wydajność aplikacji, zwłaszcza jeśli wskazywane adresy nie są zgadywane przez jednostkę wstępnego pobierania (ang. *prefetcher*) procesora. Korzystanie z technik takich jak SSO i SOO jest nieocenioną pomocą w ograniczaniu tych problemów.

Oszczędzanie pamięci dzięki technice COW

Każdy, kto korzystał w programie GCC przed wersją 5.0 z typu `std::string`, miał okazję zetknąć się z inną optymalizacją zwaną **kopiowaniem przy zapisie** (ang. *Copy-On-Write*, COW). W tej technice implementacji w przypadku tworzenia wielu instancji ciągu mających taką samą bazową tablicę znaków współdzieliły one jeden adres pamięci. Gdy następował zapis do ciągu, bazowa pamięć była kopiowana — stąd nazwa.

Technika ta pozwalała oszczędzać pamięć i utrzymać pamięć podręczną w stanie rozgrzania, a przy jednym wątku często zapewniała solidną wydajność. Należy się jej jednak wystrzegać w kontekstach wielowątkowych. Konieczność używania blokad może naprawdę zniszczyć wydajność. Tak jak w każdym zagadnieniu związanym z wydajnością, najlepiej jest po prostu zmierzyć, czy w danym przypadku będzie to najlepsze rozwiązanie.

Omówmy teraz cechę standardu C++17 pozwalającą osiągać dobrą wydajność alokacji dynamicznych.

Korzystanie z alokatorów polimorficznych

Cechą, o której mowa, są alokatory polimorficzne, a konkretnie `std::pmr::polymorphic_allocator` i klasa polimorficzna `std::pmr::memory_resource`, używana przez alokator do przydziału pamięci.

Zasadniczo pozwalają one łatwo tworzyć łańcuchy zasobów pamięci (ang. *memory resources*), mające na celu jak najlepsze jej wykorzystanie. Łańcuchy mogą być proste, przykładowo jeden zasób rezerwuje duży fragment pamięci i rozdziela go, a jeśli pamięć się wyczerpie, następuje przejście do innego, który po prostu wywołuje operatory `new` i `delete`. Mogą też być znacznie bardziej skomplikowane: można zbudować długi łańcuch zasobów pamięci obsługujących pule różnych rozmiarów, zapewniających bezpieczeństwo wątkowe tylko wtedy, gdy to konieczne, bezpośrednio odwołujących się do pamięci systemowej z pominięciem sterty, zwracających ostatnio zwolniony fragment pamięci, by zapewnić gorącą pamięć podręczną, i dokonujących innych wymyślnych rzeczy. Standardowe polimorficzne zasoby pamięci nie zapewniają wszystkich tych możliwości, ale zaprojektowano je tak, by łatwo je było rozszerzyć.

Podejmijmy najpierw temat aren pamięci.

Używanie aren pamięci

Arena pamięci, zwana też regionem, to po prostu duży fragment pamięci istniejący przez ograniczony czas. Korzysta się z niego w celu alokowania mniejszych obiektów używanych w czasie istnienia areny. Znajdujące się w niej obiekty mogą być dealokowane tak jak zwykle albo usuwane wszystkie naraz w procesie zwanym *wygaszaniem* (ang. *winking out*). Opiszemy go później.

Areny mają kilka ogromnych zalet w porównaniu ze zwykłymi alokacjami i dealokacjami — podnoszą wydajność, bo ograniczają przydziały pamięci wymagające pozyskiwania zasobów wyższego rzędu. Redukują też fragmentację pamięci, bo może ona mieć miejsce tylko w obrębie areny. Po zwolnieniu pamięci areny problem fragmentacji również przestaje istnieć. Doskonałym pomysłem jest tworzenie oddzielnych aren dla każdego wątku. Jeśli arena jest używana tylko w jednym z nich, nie trzeba stosować blokad ani innych mechanizmów bezpieczeństwa wątków, co ogranicza ich rywalizację i daje niezły wzrost wydajności.

Jeśli program jest jednowątkowy, można niskim kosztem podnieść jego wydajność w następujący sposób:

```
auto single_threaded_pool = std::pmr::unsynchronized_pool_resource();
std::pmr::set_default_resource(&single_threaded_pool);
```

Domyślnym zasobem, o ile nie podano jawnie innego, jest `new_delete_resource`. Zasób ten wywołuje za każdym razem operatory `new` i `delete` tak samo jak zwykły `std::allocator` oraz zapewnia to samo bezpieczeństwo wątkowe (i powoduje takie same koszty).

Dzięki powyższemu fragmentowi kodu wszystkie przydziały dokonywane przez alokatory z przestrzeni nazw `pmr` będą wykonywane bez blokad. Należy jednak faktycznie korzystać przy tym z typów `pmr`. W przypadku kontenerów standardowych wystarczy w tym celu przekazać w parametrze szablonu alokator `std::pmr::polymorphic_allocator<T>`. Przestrzeń nazw `pmr` zawiera aliasy typów dla wielu kontenerów standardowych. Obie utworzone poniżej zmienne są tego samego typu i obie korzystają z domyślnego zasobu pamięci:

```
  auto ints = std::vector<int,
std::pmr::polymorphic_allocator<int>>(std::pmr::get_default_resource());
  auto also_ints = std::pmr::vector<int>{};
```

W przypadku pierwszej z nich zasób przekazywany jest jednak w sposób jawny. Przejrzyjmy teraz zasoby dostępne w przestrzeni nazw `pmr`.

Korzystanie z monotonicznego zasobu pamięci

Pierwszym omawianym przez nas zasobem będzie `std::pmr::monotonic_buffer_resource`. To zasób, który wyłącznie alokuje pamięć i nie wykonuje niczego podczas dealokacji. Pamięć jest dealokowana wyłącznie podczas niszczenia zasobu albo po jawnym wywołaniu funkcji `release()`. W połączeniu z brakiem bezpieczeństwa wątkowego sprawia to, że typ ten jest niezwykle wydajny. Jeśli aplikacja od czasu do czasu wykonuje zadanie, w którym dokonuje mnóstwa alokacji w określonym wątku, a później zwalnia naraz wszystkie użyte obiekty, skorzystanie z zasobu monotonicznego może dać wielkie zyski. Stanowi on również doskonały element bazowy do budowania łańcuchów zasobów.

Korzystanie z zasobów pulowych

Często spotykaną kombinacją jest użycie zasobu pulowego opartego na zasobie monotonicznego bufora. Standardowe zasoby pulowe tworzą pule różnej wielkości fragmentów pamięci (ang. *chunks*). W przestrzeni nazw `std::pmr` są dwa typy: `unsynchronized_pool_resource`, przeznaczony do użycia wtedy, gdy tylko jeden wątek alokuje i dealokuje pamięć zasobu, oraz `synchronized_pool_resource` do użytku wielowątkowego. Obydwa typy powinny zapewnić dużą lepszą wydajność w porównaniu z alokatorem globalnym, zwłaszcza gdy jako zasób wyższego rzędu zostanie użyty bufor monotoniczny. Jeśli zastanawiasz się, jak utworzyć z nich łańcuch, oto przykład:

```
  auto buffer = std::array<std::byte, 1 * 1024 * 1024>{};
  auto monotonic_resource =
      std::pmr::monotonic_buffer_resource{buffer.data(), buffer.size()};
  auto pool_options = std::pmr::pool_options{.max_blocks_per_chunk = 0,
      .largest_required_pool_block = 512};
  auto arena =
      std::pmr::unsynchronized_pool_resource{pool_options,
&monotonic_resource};
```

Tworzymy dla areny bufor o rozmiarze 1 MB. Przekazujemy go do zasobu monotonicznego, który z kolei jest przekazywany do niezsynchronizowanego zasobu pulowego. Powstaje prosty, ale wydajny łańcuch alokatorów. Operator `new` nie zostanie wywołany, zanim nie zostanie wykorzystany cały początkowy bufor.

Do obu typów pulowych można przekazać obiekt `std::pmr::pool_options` w celu ograniczenia maksymalnej liczby bloków danej wielkości (`max_blocks_per_chunk`) albo rozmiaru największego bloku (`largest_required_pool_block`). Przekazanie liczby 0 powoduje użycie określonych w implementacji wartości domyślnych. W przypadku biblioteki GCC faktyczna liczba bloków przypadających na fragment pamięci różni się w zależności od rozmiaru bloku. Jeśli przekroczony zostanie maksymalny rozmiar, zasób pulowy przydzieli pamięć bezpośrednio z zasobu wyższego rzędu. Tak samo będzie w przypadku zasobu wyższego rzędu, jeśli wyczerpie się jego początkowa pamięć. W takiej sytuacji będzie on alokował fragmenty pamięci zwiększające się w postępie geometrycznym.

Pisanie własnego zasobu pamięci

Jeśli standardowe zasoby pamięci nie spełniają wszystkich Twoich oczekiwań, zawsze możesz w dość prosty sposób utworzyć własny. Przykładem dobrej optymalizacji, nie zawsze zapewnianej przez wszystkie implementacje biblioteki standardowej, jest śledzenie ostatnio zwolnionych fragmentów pamięci o danym rozmiarze i zwracanie ich przy następnych alokacjach o tej samej wielkości. Taki bufor ostatnio używanych (ang. *most recently used*) fragmentów pamięci pozwala na lepsze rozgrzanie pamięci podręcznej danych, co powinno wpływać korzystnie na wydajność aplikacji. Można to potraktować jak zestaw kolejek LIFO dla fragmentów pamięci.

Zdarza się, że programista chce debugować alokacje i dealokacje. Poniższy listing zawiera prosty zasób pomagający w tym zadaniu:

```
class verbose_resource : public std::pmr::memory_resource {
  std::pmr::memory_resource *upstream_resource_;
public:
  explicit verbose_resource(std::pmr::memory_resource *upstream_resource)
      : upstream_resource_(upstream_resource) {}
```

Klasa `verbose_resource` dziedziczy po polimorficznym zasobie bazowym. Przyjmuje też zasób wyższego rzędu, za pomocą którego faktycznie przydziela pamięć. Konieczna jest implementacja trzech funkcji prywatnych — jednej do alokowania, drugiej do dealokowania, a trzeciej do porównywania instancji samego zasobu. Oto pierwsza funkcja:

```
private:
  void *do_allocate(size_t bytes, size_t alignment) override {
    std::cout << "Alokowanie " << bytes << " bajtów\n";
    return upstream_resource_->allocate(bytes, alignment);
  }
```

Wypisuje ona jedynie rozmiar alokacji na standardowym wyjściu, a potem alokuje pamięć przy użyciu zasobu wyższego rzędu. Kolejna funkcja będzie podobna:

```
  void do_deallocate(void *p, size_t bytes, size_t alignment) override {
    std::cout << "Dealokowanie " << bytes << " bajtów\n";
    upstream_resource_->deallocate(p, bytes, alignment);
  }
```

Rejestrujemy, ile pamięci dealokujemy, a wykonanie tego zadania przekazujemy wyżej. A teraz ostatnia wymagana funkcja:

```
[[nodiscard]] bool
do_is_equal(const memory_resource &other) const noexcept override {
  return this == &other;
}
```

Porównujemy po prostu adresy instancji, by dowiedzieć się, czy są one jednakowe. Atrybut `[[nodiscard]]` zapewnia, że wartość zwrócona do miejsca wywołania zostanie rzeczywiście wykorzystana, co pozwala zapobiec niezamierzonemu niewłaściwemu wykorzystaniu funkcji.

To by było na tyle. Prawda, że jak na tak zaawansowaną funkcję, interfejs API alokatorów `pmr` nie jest zbyt skomplikowany?

Przestrzeni nazw `pmr` można użyć nie tylko do śledzenia alokacji, ale do zapobiegania przydzielaniu pamięci wtedy, gdy nie powinno to mieć miejsca.

Zapobieganie nieoczekiwanym alokacjom

Specjalna funkcja `std::pmr::null_memory_resource()` zgłasza wyjątek, gdy ktoś próbuje przydzielać pamięć za jej pomocą. Poprzez ustawienie jej jako zasobu domyślnego można zabezpieczyć się przed przeprowadzaniem alokacji przy użyciu przestrzeni nazw `pmr`:

```
std::pmr::set_default_resource(null_memory_resource());
```

Można też za jej pomocą ograniczyć niepożądane przydziały z zasobu wyższego rzędu. Spójrz na poniższy kod:

```
auto buffer = std::array<std::byte, 640 * 1024>{}; // 640 k powinno wystarczyć każdemu
auto resource = std::pmr::monotonic_buffer_resource{
    buffer.data(), buffer.size(), std::pmr::null_memory_resource()};
```

Jeśli ktoś spróbuje zaalokować więcej, niż wynosi ustalona wielkość bufora, zostanie zgłoszony wyjątek `std::bad_alloc`.

Przejdźmy do ostatniej pozycji w tym rozdziale.

Wygaszanie pamięci

Czasami brak konieczności cofania przydziałów pamięci, na co pozwala zasób bufora monotonicznego, nie zapewnia wystarczającej wydajności. Pomóc wtedy może specjalna technika zwana *wygaszaniem* (ang. *winking out*). Wygaszanie obiektów oznacza, że nie tylko nie są one osobno dealokowane, ale też nie są wywoływane ich destruktory. Obiekty po prostu wyparowują, co pozwala oszczędzić czas poświęcany normalnie na wywoływanie destruktorów wszystkich obiektów zawartych w arenie wraz z ich składowymi (i składowymi składowych…).

> To zaawansowany temat. Zachowaj ostrożność przy korzystaniu z tej techniki i używaj jej tylko wtedy, gdy gra jest warta świeczki.

Technika ta pozwala zaoszczędzić cenne cykle procesora, ale nie zawsze może być używana. Nie wolno jej stosować wtedy, gdy obiekty obsługują zasoby inne niż pamięć, bo grozi to wyciekiem zasobów. To samo dotyczy sytuacji, gdy kod jest uzależniony od efektów ubocznych destruktorów obiektów.

Zobaczmy teraz, jak działa wygaszanie:

```
auto verbose = verbose_resource(std::pmr::get_default_resource());
auto monotonic = std::pmr::monotonic_buffer_resource(&verbose);
std::pmr::set_default_resource(&monotonic);

auto alloc = std::pmr::polymorphic_allocator{};
auto *vector = alloc.new_object<std::pmr::vector<std::pmr::string>>();
vector->push_back("pierwszy ciąg");
vector->emplace_back("drugi, długi ciąg, który musi być alokowany");
```

Utworzyliśmy tutaj ręcznie polimorficzny alokator, który będzie korzystał z naszego domyślnego zasobu — monotonicznego, który rejestruje każde sięgnięcie do zasobu wyższego rzędu. Do tworzenia obiektów użyjemy wprowadzonej w standardzie C++20 w przestrzeni nazw `pmr` funkcji `new_object`. Tworzymy wektor ciągów. Pierwszy z nich możemy przekazać przy użyciu funkcji `push_back`, bo jest na tyle mały, że zmieści się w buforze małych ciągów, który mamy dzięki mechanizmowi SSO. Gdybyśmy w przypadku drugiego ciągu też użyli funkcji `push_back`, musiałby on być zaalokowany przy użyciu zasobu domyślnego i dopiero potem przekazany do wektora. Zastosowanie funkcji `emplace_back` powoduje, że ciąg jest konstruowany wewnątrz funkcji wektora (a nie przed wywołaniem), więc zostanie użyty alokator wektora. Nie wywołujemy też nigdzie destruktorów alokowanych obiektów i po prostu dealokujemy wszystko naraz przy wyjściu z zakresu. Powinno to zapewnić trudną do pobicia wydajność.

Była to ostatnia pozycja na liście tego rozdziału. Podsumujmy, czego się dowiedzieliśmy.

Podsumowanie

W tym rozdziale zapoznaliśmy się z rozmaitymi idiomami i wzorcami używanymi w świecie języka C++. Jesteś już w stanie pisać płynny, idiomatyczny kod w tym języku. Wyjaśniliśmy, jak przeprowadzać automatyczne czyszczenie. Potrafisz teraz tworzyć bezpieczniejsze typy, które można właściwie przenosić, kopiować i zamieniać. Wiesz, jak wykorzystać wyszukiwanie ADL zarówno w celu zmniejszenia czasu kompilacji, jak i tworzenia punktów dostosowywania. Omówiliśmy, jak wybierać pomiędzy polimorfizmem statycznym a dynamicznym. Dowiedzieliśmy się też, jak wprowadzać do typów wytyczne, kiedy korzystać z wymazywania typów, a kiedy nie.

Co więcej, pokazaliśmy sposób tworzenia obiektów przy użyciu fabryk i płynnych budowniczych. Używanie w tym celu aren pamięci również nie jest już tajemnicą, podobnie jak tworzenie maszyn stanów za pomocą narzędzi takich jak warianty.

Przy okazji tego wszystkiego poruszyliśmy też parę dodatkowych zagadnień. Uff! Następny przystanek w naszej podróży będzie dotyczył budowania i pakowania oprogramowania.

Pytania

1. Czego dotyczą zasady trzech, pięciu i zera?
2. Kiedy należy korzystać z niebloidów, a kiedy z ukrytych funkcji zaprzyjaźnionych?
3. Jak można ulepszyć interfejs klasy `Array`, by lepiej nadawał się do zastosowań produkcyjnych?
4. Czym są wyrażenia redukcji (ang. *fold expressions*)?
5. Kiedy nie należy używać polimorfizmu statycznego?
6. W jaki sposób można uchronić się przed jeszcze jedną alokacją w przykładzie z wygaszaniem (ang. *winking out*)?

Materiały dodatkowe

- Lewis Baker, Eric Niebler, Kirk Shoop, *tag_invoke: A general pattern for supporting customisable functions*, propozycja do standardu ISO C++, *https://wg21.link/p1895*.
- Gašper Ažman, *tag_invoke :: niebloids evolved*, wykład przygotowany na konferencję Core C++, klip wideo w serwisie YouTube, *https://www.youtube.com/watch?v=oQ26YL0J6DU*.
- Sean Parent, *Inheritance Is The Base Class of Evil*, wykład na konferencji GoingNative 2013, klip wideo w serwisie Channel9, *https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil*.
- Andrei Alexandrescu, *Nowoczesne projektowanie w C++*, Helion, Gliwice 2011.
- Scott Meyers, *How Non-Member Functions Improve Encapsulation*, artykuł w serwisie Dr. Dobbs, *https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197*.
- *Returning a Status or a Value*, Status User Guide („Przewodnik użytkownika bibliotek Status"), dokumentacja pakietu Abseil, *https://abseil.io/docs/cpp/guides/status#returning-a-status-or-a-value*.

- `function_ref`, repozytorium w serwisie GitHub, *https://github.com/TartanLlama/function_ref*.
- Bartłomiej Filipek, *How To Use std::visit With Multiple Variants*, wpis na programistycznym blogu Bartka, *https://www.bfilipek.com/2018/09/visit-variants.html*.
- Mateusz Pusz, CppCon 2018: *Effective replacement of dynamic polymorphism with std::variant*, wideo w serwisie YouTube, *https://www.youtube.com/watch?v=gKbORJtnVu8*.

Dzięki tej książce poznasz narzędzia i rozwiązania, które ułatwiają projektowanie w języku C++ nawet najbardziej skomplikowanych aplikacji. Autorzy przybliżają samo pojęcie architektury oprogramowania i na praktycznych przykładach wyjaśniają, na czym polega jej tworzenie. Pokazują również aktualne trendy projektowe i uczą, jak za pomocą C++ krok po kroku, element po elemencie budować aplikacje i systemy na dowolnym poziomie zaawansowania. W trakcie lektury dowiesz się, jakie warunki powinna spełniać efektywna architektura oprogramowania i jak sprawić, by gwarantowała wysoki poziom bezpieczeństwa, skalowalności i wydajności. Liczne przykłady, zrozumiałe objaśnienia i przyjazny język pozwalają na efektywne i szybkie przyswajanie wiedzy dotyczącej tworzenia rozproszonych, skomplikowanych aplikacji w C++.

W książce:

- projektowanie aplikacji bazujących na wydajnej, nowoczesnej i czytelnej architekturze oprogramowania
- używanie najważniejszych zasad i wzorców projektowych umożliwiających tworzenie efektywnego kodu za pomocą języka C++
- analizowanie różnych koncepcji architektury oprogramowania i stosowanie tych, które w największym stopniu odpowiadają danemu projektowi
- efektywne wykorzystywanie mechanizmów i rozwiązań dostępnych w najnowszej wersji języka C++

Adrian Ostrowski — od ponad 10 lat w branży informatycznej, obecnie specjalizuje się w integracji oprogramowania Intel i Habana z aplikacjami uczenia maszynowego. Pasjonat języka C++, który nie ma przed nim tajemnic. Chętnie dzieli się wiedzą i doświadczeniem — ta książka potwierdza, że potrafi robić to naprawdę dobrze.

Piotr Gaczkowski — entuzjasta automatyzacji i upraszczających codzienne życie nowoczesnych rozwiązań informatycznych, których jest twórcą. Od ponad 10 lat posługuje się językiem C++ i stosuje metodykę DevOps. Ma bogate doświadczenie w branży informatycznej — zdobywał je w pracy zarówno na etacie, jak i w ramach freelancingu.

ISBN 978-83-283-8666-2