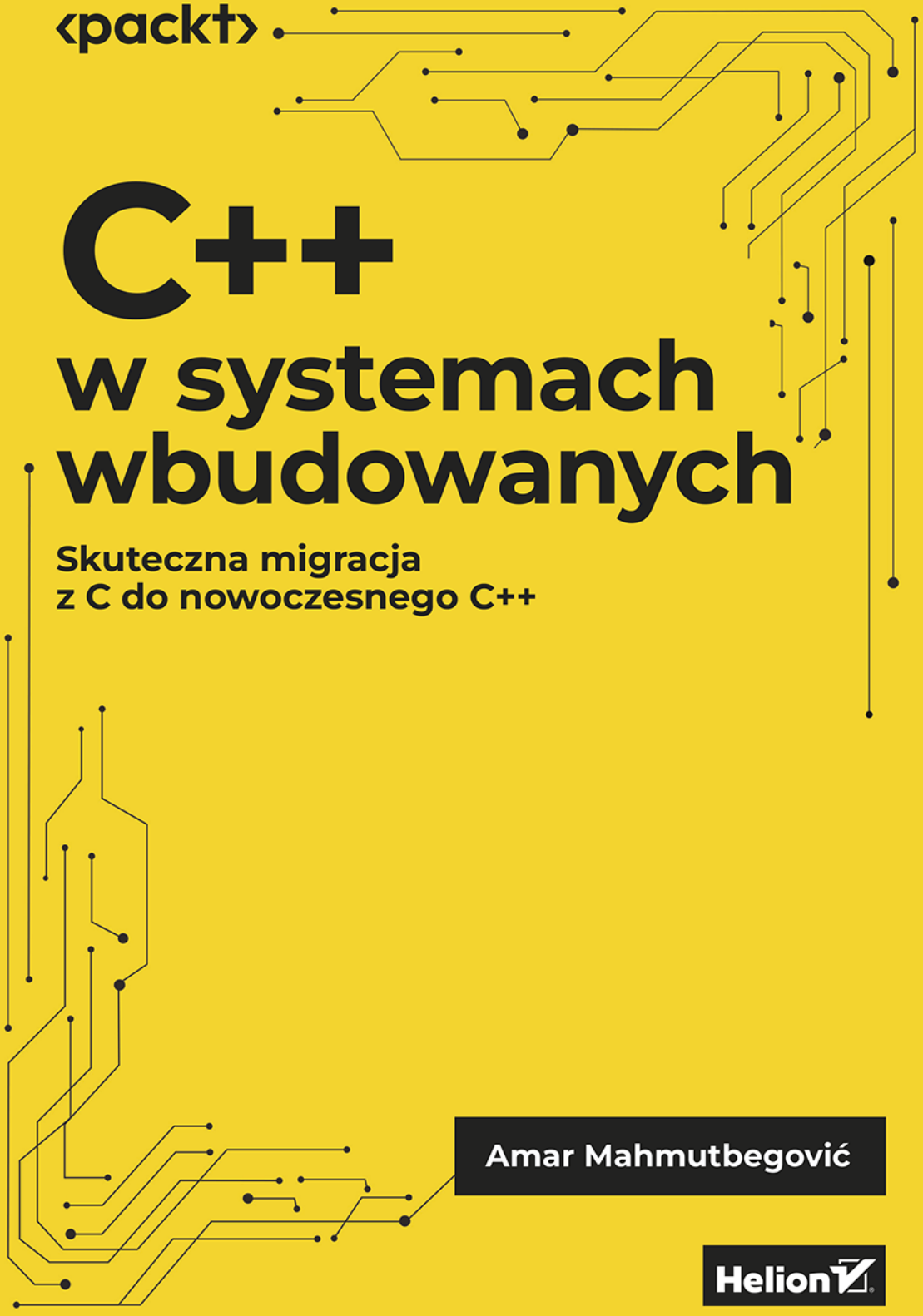
packt
C++
w systemach wbudowanych
Skuteczna migracja z C do nowoczesnego C++
Amar Mahmutbegović
Helion

Tytuł oryginału: C++ in Embedded Systems: A practical transition from C to modern C++

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3560-0

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

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

Printed in Poland.

Spis treści

CZĘŚĆ 2. Podstawy języka C++

CZĘŚĆ 4. Zastosowanie języka C++ do rozwiązywania problemów w systemach wbudowanych

Obalanie popularnych mitów na temat C++

Rozdział 1

Tworzenie oprogramowania dla mikrokontrolerów i systemów wbudowanych (ang. *embedded systems*) jest wymagające. Aby w pełni wykorzystać możliwości systemów o ograniczonych zasobach, programiści muszą dobrze poznać architekturę danej platformy. Powinni być świadomi dostępnych zasobów, w tym możliwości procesora, dostępnej pamięci i urządzeń peryferyjnych. Potrzeba uzyskania bezpośredniego dostępu do sprzętu poprzez urządzenia peryferyjne mapowane w pamięci spowodowała, że w ciągu ostatniego półwiecza **C** stał się językiem wybieranym do opracowywania projektów dla systemów wbudowanych.

Celem każdego języka programowania jest przeprowadzenie procesu przekształcania abstrakcji specyficznych dla danej aplikacji na kod, który następnie można przetworzyć na kod maszynowy. Na przykład język **COBOL** (ang. *common business-oriented language*) jest używany w aplikacjach bankowych, **Fortran** zaś w badaniach naukowych i złożonych obliczeniach matematycznych. Z kolei C to język ogólnego przeznaczenia, który jest powszechnie stosowany do tworzenia **systemów operacyjnych** (ang. *operating systems*) i aplikacji systemów wbudowanych.

C to język o prostej i łatwej do przyswojenia składni. Prosta składnia oznacza jednak, że nie jest w stanie wyrażać skomplikowanych koncepcji. Wprawdzie C umożliwia wykonywanie złożonych operacji, ale wymaga bardziej szczegółowego i rozbudowanego kodu do zarządzania złożonością w porównaniu z językami wyższego poziomu, które abstrahują te szczegóły.

Pod koniec lat 70. języki wysokiego poziomu nie mogły dorównać wydajności C. To zmotywowało duńskiego informatyka Bjarne Stroustrupa do rozpoczęcia prac nad **C z klasami** (ang. *C with Classes*), czyli poprzednikiem C++. Obecnie C++ jest językiem wieloparadygmatowym, który został zaprojektowany z myślą o wydajności działania. Pochodzenie C++ wciąż jest źródłem pewnych mitów, co często powstrzymuje przed jego stosowaniem w programowaniu systemów wbudowanych. W tym rozdziale przedstawiam te mity i je obalam.

Ten rozdział zawiera omówienie następujących zagadnień:

- krótka historia C++,
- C z klasami,
- wzrost ilości kodu i koszt czasowy.

Wymogi techniczne

Aby w pełni skorzystać z tego rozdziału, gorąco zachęcam do sięgnięcia po narzędzie Compiler Explorer (*https://godbolt.org/*) podczas analizowania przykładów. Wybierz GCC jako kompilator i architekturę x86 jako cel (ang. *target*). To pozwoli obserwować za pomocą standardowego wyjścia (stdio) dane wyjściowe kodu i jeszcze lepiej zrozumieć sposób jego działania. Przykłady z tego rozdziału są dostępne w repozytorium GitHuba (*https://github.com/PacktPublishing/Cpp-in-Embedded-Systems/tree/main/Chapter01*).

Krótka historia języka C++

W połowie lat 60. język programowania symulacyjnego **SIMULA** wprowadził do świata tworzenia oprogramowania pojęcia klas i obiektów. **Klasy** są abstrakcjami, które pozwalają w zwięzły sposób przedstawiać w programowaniu koncepcje ze świata rzeczywistego, a tym samym powodują, że kod staje znacznie czytelniejszy dla człowieka. W programowaniu systemów wbudowanych **UART**, **SPI**, **czujnik temperatury**, **regulator PID** czy **sterownik temperatury** to niektóre z koncepcji, które można zaimplementować za pomocą klas. SIMULA wprowadził również hierarchiczne relacje między klasami. Na przykład klasa `PT100` jest także klasą czujnika temperatury (`TemperatureSensor`), a klasa sterownika temperatury (`TemperatureController`) zawiera egzemplarze (obiekt) czujnika temperatury (`TemperatureSensor`) i regulatora PID (`PidController`). To podejście stało się znane jako **programowanie zorientowane obiektowo** (ang. *object-oriented programming*, **OOP**).

W rozważaniach dotyczących ewolucji języków programowania Bjarne Stroustrup, twórca C++, podzielił się podejściem, które przyjął podczas projektowania tego języka. Stroustrup dążył do zniwelowania różnic między abstrakcjami wysokiego poziomu i wydajnością niskopoziomową. Oto co powiedział:

> *Mój pomysł był bardzo prosty. Chodziło o to, aby koncepcje z języka SIMULA wykorzystać do utworzenia ogólnych abstrakcji, które ułatwiłyby użytkownikom reprezentowanie różnych rzeczy, i połączyć je z niskopoziomowymi mechanizmami. W tamtym czasie najlepszym językiem do tego celu był C, opracowany w Bell Labs przez Dennisa Ritchiego. Chciałem połączyć te dwie idee, aby umożliwić tworzenie abstrakcji wysokiego poziomu. Jednocześnie miały być one na tyle wydajne i bliskie sprzętowi, aby można było je wykorzystywać w naprawdę wymagających zadaniach obliczeniowych.*

Opracowany przez Bjarne Stroustrupa język C++ początkowo powstał jako „C z klasami". Z czasem przekształcił się w nowoczesny język programowania, który nadal zapewnia bezpośredni dostęp do sprzętu i urządzeń peryferyjnych mapowanych w pamięci. Wykorzystanie zaawansowanych abstrakcji umożliwia tworzenie w C++ ekspresyjnego i wysoce modularnego kodu. C++ jest językiem ogólnego przeznaczenia, który obsługuje wiele paradygmatów, w tym programowanie proceduralne, obiektowe oraz, w pewnym stopniu, także funkcyjne.

Choć C jest głównym językiem w programowaniu systemów wbudowanych (wykorzystuje go aż 60% projektów), popularność C++ stale rośnie. Szacuje się, że C++ jest używany w 20 – 30% projektów systemów wbudowanych, oferując m.in. obsługę klas, lepszą kontrolę typów i możliwość przeprowadzania obliczeń w trakcie kompilacji.

Mimo zalet C++ język C wciąż dominuje w programowaniu systemów wbudowanych. Istnieje wiele powodów takiego stanu rzeczy, które omawiam w tym rozdziale. C++ jest bardziej złożonym językiem niż C, co utrudnia jego naukę początkującym programistom. Natomiast C jest łatwiejszy do opanowania i pozwala szybciej wdrożyć nowych programistów do projektu.

Prostota C jest zaletą, ponieważ umożliwia początkującym programistom szybsze rozpoczęcie pracy nad projektami. Jednak w przypadku bardziej skomplikowanej logiki kod utworzony w C staje się zbyt rozwlekły. Zazwyczaj prowadzi to do powstania większej bazy kodu ze względu na brak ekspresyjności. W takich sytuacjach C++ oferuje wyższy poziom abstrakcji, który — jeśli zostanie właściwie wykorzystany — powoduje, że kod jest łatwiejszy do czytania i zrozumienia.

Inne powody, dla których C++ nie jest szerzej stosowany, wiążą się z mitami na jego temat. Wciąż pokutuje przekonanie, że C++ to tylko „C z klasami”, że użycie C++ jest absolutnie niedopuszczalne w systemach o znaczeniu krytycznym dla bezpieczeństwa ze względu na dynamiczną alokację pamięci w bibliotece standardowej, że C++ prowadzi do wygenerowania nadmiernie rozbudowanego kodu, zwiększa zużycie pamięci oraz wydłuża czas wykonania. W tym rozdziale odniosę się do niektórych z najpopularniejszych mitów dotyczących języka C++ w kontekście programowania systemów wbudowanych. Rozprawimy się z tymi mitami i rzucimy nowe światło na zastosowanie C++ w takich systemach!

C z klasami

Z perspektywy historycznej C++ wywodzi się z projektu „C z klasami”. Pierwszy kompilator C++, **Cfront**, przekształcał kod C++ na C, ale to było dawno temu. Z biegiem czasu C i C++ ewoluowały oddzielnie i obecnie są definiowane przez odrębne standardy językowe. C zachował swoją prostotę, podczas gdy C++ stał się nowoczesnym językiem, który umożliwia tworzenie abstrakcyjnych rozwiązań problemów bez poświęcania wydajności. Mimo to C++ wciąż bywa nazywany „C z klasami”, co sugeruje, że jedyną wartością dodaną w C++ są klasy.

Standard C++11 został wydany w 2011 roku i jest drugą główną wersją C++. Wprowadził on wiele nowych funkcji, które unowocześniły język, takich jak pętle zakresowe, wyrażenia lambda i `constexpr`. Kolejne wydania — C++14, C++17, C++20 i C++23 — kontynuowały modernizację języka i wprowadzały funkcje, dzięki którym określenie „C z klasami” stało się jedynie wspomnieniem odległego przodka nowoczesnego C++.

Nowoczesny C++

Aby się przekonać, że C++ to nie tylko C z klasami, przyjrzyjmy się kilku krótkim przykładowym fragmentom kodu w C i ich odpowiednikom w nowoczesnym języku C++. Rozpoczniemy od prostego przykładu, którego zadaniem jest wyświetlanie elementów pochodzących z bufora liczb całkowitych:

```
#define N 20
int buffer[N];

for(int i = 0; i < N; i ++) {
    printf("%d ", buffer[i]);
}
```

Tamten kod w języku C można w następujący sposób przedstawić w C++:

```
std::array<int, 20> buffer;

for(const auto& element : buffer) {
    printf("%d ", element);
}
```

Przede wszystkim widzimy, że wersja w C++ jest krótsza. Zawiera mniej słów i jest bliższa językowi angielskiemu niż kod utworzony w C. Ponadto jest łatwiejsza do czytania. Jeśli masz doświadczenie w C i nie miałeś styczności z językami wyższego poziomu, pierwsza wersja może wydawać się bardziej zrozumiała, ale porównajmy je. Zauważ, że kod w C definiuje stałą `N`, która określa wielkość bufora (`buffer`). Ta stała jest używana do zdefiniowania bufora i jako granica pętli `for`.

Pętle zakresowe, wprowadzone w C++11, eliminują konieczność używania wielkości kontenera w warunku zakończenia pętli. Informacja o wielkości jest już zawarta w kontenerze `std::array`, więc będzie wykorzystana przez pętlę zakresową do przeprowadzenia łatwej iteracji tablicy. Ponadto nie ma indeksowania bufora, ponieważ elementy są dostępne przez stałą referencję, co gwarantuje, że elementy nie zostaną zmodyfikowane wewnątrz pętli.

Spójrzmy na prosty kod w C, który kopiuje wszystkie elementy z tablicy `array_a` do `array_b`, o ile są mniejsze niż `10`:

```
int w_idx = 0;
for(int i = 0; i < sizeof(array_a)/sizeof(int); i++) {
    if(array_a[i] < 10) {
        array_b[w_idx++] = array_a[i];
    }
}
```

Oto kod w C++ o tej samej funkcjonalności:

```
auto less_than_10 =  [](auto x) -> bool {
    return x < 10;
};

std::copy_if(std::begin(array_a), std::end(array_a), std::begin(array_b),
less_than_10);
```

Zamiast ręcznej iteracji tablicy `array_a` i kopiowania elementów do `array_b` tylko wtedy, gdy ich wartość jest większa niż `10`, możemy użyć funkcji `copy_if` z biblioteki standardowej C++. Pierwsze dwa argumenty `std::copy_if` to iteratory, które określają zakres elementów do przeanalizowania w `array_a`: pierwszy iterator wskazuje początek tablicy, drugi zaś pozycję tuż za ostatnim elementem. Trzeci argument to iterator, który wskazuje początek `array_b`, natomiast czwarty to wyrażenie lambda `less_than_10`.

Wyrażenie lambda to anonimowy obiekt funkcji, który można zadeklarować w miejscu jego wywołania lub przekazać jako argument funkcji. Lambdy omawiam bardziej szczegółowo w rozdziale 10. W przypadku omawianego tutaj wywołania `std::copy_if` lambda `less_than_10` służy do określenia, które elementy tablicy `array_a` mają zostać skopiowane do `array_b`. Zamiast lambdy moglibyśmy również zdefiniować osobną funkcję `less_than_10`, która przyjmuje liczbę całkowitą i zwraca wartość logiczną, jeśli liczba jest większa niż `10`. Jednak dzięki użyciu lambdy tę funkcjonalność możemy zapisać blisko miejsca, w którym przekazujemy ją do algorytmu, co czyni kod bardziej zwięzłym i czytelnym.

Typy generyczne

W poprzednich przykładach wykorzystywaliśmy kontener `std::array` z biblioteki standardowej. Jest to szablon klasy, który opakowuje tablicę w stylu języka C wraz z informacją o jej wielkości. Szablony omawiam bardziej szczegółowo w rozdziale 8. Gdy używasz kontenera `std::array` z określonym typem bazowym i wielkością, wówczas w trakcie procesu **tworzenia egzemplarza** kompilator definiuje nowy typ.

Wywołanie `std::array<int, 10>` tworzy typ kontenera, który zawiera tablicę w stylu C wraz z dziesięcioma liczbami całkowitymi. Z kolei wywołanie `std::array<int, 20>` prowadzi do utworzenia typu kontenera z tablicą dwudziestu liczb całkowitych. Typy `std::array<int, 10>` i `std::array<int, 20>` są różne — wprawdzie oba mają ten sam typ bazowy, ale inną wielkość.

Wywołanie `std::array<float, 10>` doprowadziłoby do utworzenia trzeciego typu, ponieważ od `std::array<int, 10>` różni się typem bazowym. Użycie odmiennych parametrów prowadzi do powstania różnych typów. Typy zdefiniowane jako szablony są typami generycznymi (ogólnymi), które stają się konkretnymi dopiero po utworzeniu egzemplarza.

Aby lepiej zrozumieć i docenić typy generyczne, przyjrzyjmy się implementacji bufora cyklicznego w języku C i porównajmy ją z rozwiązaniem opartym na szablonach w C++.

Bufor cykliczny w języku C

Bufor cykliczny (ang. *ring buffer*) to struktura danych powszechnie stosowana w programowaniu systemów wbudowanych. Zazwyczaj jest implementowany jako zestaw funkcji operujących na tablicy, z wykorzystaniem indeksów zapisu i odczytu, które są używane w trakcie operacji dostępu do elementów tablicy. Zmienna licznika jest wykorzystywana do zarządzania przestrzenią tablicy. Interfejs składa się z funkcji wstawiania (`push`) i pobierania (`pop`) elementów, które działają w następujący sposób:

- Funkcja `push` służy do zapisywania elementów w buforze cyklicznym. W trakcie każdego jej wywołania element danych jest zapisywany w tablicy, a indeks zapisu

jest zwiększany. Jeśli indeks zapisu osiągnie wartość równą liczbie elementów w tablicy danych, to zostanie wyzerowany.

- Funkcja `pop` służy do pobierania elementu z bufora cyklicznego. W trakcie każdego jej wywołania, jeśli tablica bazowa nie jest pusta, wówczas jest zwracany element tablicy, na który wskazuje indeks odczytu. Następnie ten indeks jest zwiększany.

Podczas każdego wywołania funkcji `push` zwiększamy wartość licznika, natomiast w trakcie wywołania funkcji `pop` ją zmniejszamy. Jeśli wielkość licznika będzie równa wielkości tablicy danych, to indeks odczytu musimy przesunąć do przodu.

Oto wymogi techniczne dla bufora cyklicznego, który chcemy zaimplementować w module C:

- nie powinien korzystać z dynamicznej alokacji pamięci;
- gdy bufor jest pełny, najstarszy element zostanie nadpisany;
- funkcje `push` i `pop` zostają udostępnione w celu zapisywania i pobierania danych z bufora;
- w buforze cyklicznym będą przechowywane liczby całkowite.

Oto proste rozwiązanie utworzone w języku C, które spełnia powyższe wymogi:

```
#include <stdio.h>

#define BUFFER_SIZE 5

typedef struct {
  int arr[BUFFER_SIZE]; // Tablica przeznaczona do bezpośredniego przechowywania
                        // wartości całkowitych
  size_t write_idx;     // Indeks następnego elementu do zapisu (dodania)
  size_t read_idx;      // Indeks następnego elementu do odczytu (pobrania)
  size_t count;         // Liczba elementów w buforze
} int_ring_buffer;

void int_ring_buffer_init(int_ring_buffer *rb) {
  rb->write_idx = 0;
  rb->read_idx = 0;
  rb->count = 0;
}

void int_ring_buffer_push(int_ring_buffer *rb, int value) {
  rb->arr[rb->write_idx] = value;
  rb->write_idx = (rb->write_idx + 1) % BUFFER_SIZE;
  if (rb->count < BUFFER_SIZE) {
    rb->count++;
  } else {
    // Bufor jest pełny, przesuń read_idx do przodu
    rb->read_idx = (rb->read_idx + 1) % BUFFER_SIZE;
  }
}

int int_ring_buffer_pop(int_ring_buffer *rb) {
  if (rb->count == 0) {
```

```
    return 0;
  }
  int value = rb->arr[rb->read_idx];
  rb->read_idx = (rb->read_idx + 1) % BUFFER_SIZE;
  rb->count--;
  return value;
}

int main() {
  int_ring_buffer rb;
  int_ring_buffer_init(&rb);

  for (int i = 0; i < 10; i++) {
    int_ring_buffer_push(&rb, i);
  }

  while (rb.count > 0) {
    int value = int_ring_buffer_pop(&rb);
    printf("%d\n", value);
  }
  return 0;
}
```

Do inicjalizacji bufora używamy pętli `for`. Ponieważ wielkość bufora wynosi `5`, wartości od `5` do `9` zostaną w nim zapisane, gdy bufor cykliczny nadpisze istniejące dane. A co w sytuacji, jeśli w buforze cyklicznym chcemy przechowywać liczby zmiennoprzecinkowe, znaki lub struktury zdefiniowane przez użytkownika? Moglibyśmy zaimplementować tę samą logikę dla różnych typów oraz opracować nowy zestaw struktur danych i funkcji o nazwach `float_ring_buffer` lub `char_ring_buffer`. Czy da się utworzyć rozwiązanie, które mogłoby przechowywać różne typy danych i używać tych samych funkcji?

Moglibyśmy użyć tablicy typu `unsigned char` jako miejsca przechowywania różnych typów danych i wykorzystać wskaźnik `void` w celu przekazywania różnych typów danych do funkcji `push` i `pop`. Jedynym brakującym elementem jest tutaj informacja o wielkości typu danych, ale możemy sobie z tym poradzić poprzez dodanie elementu składowego `size_t elem_size` do struktury `ring_buffer`:

```
#include <stdio.h>
#include <string.h>

#define BUFFER_SIZE 20 // Całkowita liczba bajtów dostępnych w buforze

typedef struct {
  unsigned char data[BUFFER_SIZE]; // Tablica do przechowywania wartości bajtów
  size_t write_idx;                // Indeks następnego bajta do zapisu
  size_t read_idx;                 // Indeks następnego bajta do odczytu
  size_t count;     // Liczba bajtów aktualnie używanych w buforze
  size_t elem_size; // Wielkość elementu wyrażona w bajtach
} ring_buffer;

void ring_buffer_init(ring_buffer *rb, size_t elem_size) {
  rb->write_idx = 0;
  rb->read_idx = 0;
  rb->count = 0;
```

```
  rb->elem_size = elem_size;
}

void ring_buffer_push(ring_buffer *rb, void *value) {
  if (rb->count + rb->elem_size <= BUFFER_SIZE) {
    rb->count += rb->elem_size;
  } else {
    rb->read_idx = (rb->read_idx + rb->elem_size) % BUFFER_SIZE;
  }

  memcpy(&rb->data[rb->write_idx], value, rb->elem_size);
  rb->write_idx = (rb->write_idx + rb->elem_size) % BUFFER_SIZE;
}

int ring_buffer_pop(ring_buffer *rb, void *value) {
  if (rb->count < rb->elem_size) {
    // Za mało danych do pobrania
    return 0;
  }

  memcpy(value, &rb->data[rb->read_idx], rb->elem_size);
  rb->read_idx = (rb->read_idx + rb->elem_size) % BUFFER_SIZE;
  rb->count -= rb->elem_size;
  return 1; // Sukces
}

int main() {
  ring_buffer rb;
  ring_buffer_init(&rb, sizeof(int)); // Inicjalizacja bufora dla wartości typu int

  for (int i = 0; i < 10; i++) {
    int val = i;
    ring_buffer_push(&rb, &val);
  }

  int pop_value;
  while (ring_buffer_pop(&rb, &pop_value)) {
    printf("%d\n", pop_value);
  }
  return 0;
}
```

To rozwiązanie z buforem cyklicznym może być wykorzystane do przechowywania różnych typów danych. Ponieważ unikamy dynamicznej alokacji pamięci, a wielkość bufora danych jest określana w trakcie kompilacji, tracimy elastyczność w definiowaniu ilości pamięci potrzebnej dla różnych egzemplarzy bufora cyklicznego. Kolejnym problemem jest bezpieczeństwo typów. Możemy łatwo wywołać funkcję `ring_buffer_push` ze wskaźnikiem do liczby zmiennoprzecinkowej oraz `ring_buffer_pop` ze wskaźnikiem do liczby całkowitej. Kompilator nie jest w stanie wykryć tego problemu, co stwarza realne ryzyko wystąpienia błędów. Ponadto użycie wskaźnika `void` oznacza dodanie kolejnej warstwy pośredniej, ponieważ musimy polegać na pamięci, aby pobierać dane z bufora.

Czy możemy rozwiązać problemy z bezpieczeństwem typów i umożliwić definiowanie wielkości bufora cyklicznego w języku C? Rozwiązaniem jest użycie operatora wklejania tokenów (`##`) w celu utworzenia za pomocą makr zestawu funkcji dla różnych typów i wielkości. Przyjrzyjmy się prostemu przykładowi użycia operatora `##`, zanim przejdziemy do implementacji bufora cyklicznego z wykorzystaniem tej techniki:

```
#include <stdio.h>

// Makro definiujące funkcję, która oblicza sumę dwóch liczb
#define DEFINE_SUM_FUNCTION(TYPE) \
TYPE sum_##TYPE(TYPE a, TYPE b) { \
    return a + b; \
}

// Dla typów int i float definiujemy funkcje, które obliczają sumę dwóch liczb
DEFINE_SUM_FUNCTION(int)
DEFINE_SUM_FUNCTION(float)

int main() {
    int result_int = sum_int(5, 3);
    printf("Sum of integers: %d\n", result_int);

    float result_float = sum_float(3.5f, 2.5f);
    printf("Sum of floats: %.2f\n", result_float);

    return 0;
}
```

Wywołanie `DEFINE_SUM_FUNCTION(int)` utworzy funkcję `sum_int`, która przyjmuje i zwraca liczby całkowite. Jeśli wywołamy makro `DEFINE_SUM_FUNCTION` z typem `float`, to zostanie utworzona funkcja `sum_float`. Teraz, gdy już dobrze rozumiesz działanie operatora wklejania tokenów, przejdźmy do implementacji bufora cyklicznego:

```
#include <stdio.h>
#include <string.h>

// Makro przeznaczone do deklaracji typu bufora cyklicznego i funkcji dla określonego typu i wielkości
#define DECLARE_RING_BUFFER(TYPE, SIZE) \
typedef struct { \
    TYPE data[SIZE]; \
    size_t write_idx; \
    size_t read_idx; \
    size_t count; \
} ring_buffer_##TYPE##_##SIZE; \
void ring_buffer_init_##TYPE##_##SIZE(ring_buffer_##TYPE##_##SIZE *rb) { \
    rb->write_idx = 0; \
    rb->read_idx = 0; \
    rb->count = 0; \
} \
void ring_buffer_push_##TYPE##_##SIZE(ring_buffer_##TYPE##_##SIZE *rb,
TYPE value) { \
    rb->data[rb->write_idx] = value; \
    rb->write_idx = (rb->write_idx + 1) % SIZE; \
    if (rb->count < SIZE) { \
        rb->count++; \
```

```
        } else { \
            rb->read_idx = (rb->read_idx + 1) % SIZE; \
        } \
    } \
    int ring_buffer_pop_##TYPE##_##SIZE(ring_buffer_##TYPE##_##SIZE *rb, TYPE
    *value) { \
        if (rb->count == 0) { \
            return 0; /* Bufor jest pusty */ \
        } \
        *value = rb->data[rb->read_idx]; \
        rb->read_idx = (rb->read_idx + 1) % SIZE; \
        rb->count--; \
        return 1; /* Sukces */ \
    }

    // Przykład użycia z typem int i wielkością 5
    DECLARE_RING_BUFFER(int, 5) // Deklaracja typu bufora cyklicznego i funkcji dla liczb całkowitych

    int main() {
        ring_buffer_int_5 rb;
        ring_buffer_init_int_5(&rb); // Inicjalizacja bufora cyklicznego

        // Dodawanie wartości do bufora cyklicznego
        for (int i = 0; i < 10; ++i) {
            ring_buffer_push_int_5(&rb, i);
        }

        // Pobieranie wartości z bufora cyklicznego i ich wyświetlanie
        int value;
        while (ring_buffer_pop_int_5(&rb, &value)) {
            printf("%d\n", value);
        }

        return 0;
    }
```

Wprawdzie przedstawione rozwiązanie rozwiązuje nasze problemy z bezpieczeństwem typów i definiowaniem wielkości bufora cyklicznego, ale cierpi na problemy z czytelnością, zarówno w implementacji, jak i podczas użytkowania. Musimy „wywołać" `DECLARE_RING_BUFFER` poza jakąkolwiek funkcją, ponieważ w zasadzie to jest makro, które definiuje zestaw funkcji. Musimy też znać sposób jego działania i sygnatury funkcji, które zostaną wygenerowane. Rozwiązanie możemy przygotować lepiej za pomocą szablonów. Przyjrzyjmy się, jak wygląda implementacja bufora cyklicznego w C++.

Bufor cykliczny w C++

Utworzymy ogólną implementację bufora cyklicznego przy użyciu szablonów. Jako typ podstawowy możemy wykorzystać szablon klasy `std::array` i obudować go naszą logiką dodawania i usuwania elementów. Poniższy kod ilustruje, jak w kodzie C++ mógłby wyglądać typ `ring_buffer`:

```
#include <array>
#include <cstdio>
```

```
template <class T, std::size_t N> struct ring_buffer {
  std::array<T, N> arr;
  std::size_t write_idx = 0; // Indeks następnego elementu do zapisu (push)
  std::size_t read_idx = 0;  // Indeks następnego elementu do odczytu (pop)
  std::size_t count = 0;     // Liczba elementów w buforze

  void push(T t) {
    arr.at(write_idx) = t;
    write_idx = (write_idx + 1) % N;
    if (count < N) {
      count++;
    } else {
      // Bufor jest pełny, przesuń indeks odczytu
      read_idx = (read_idx + 1) % N;
    }
  }

  T pop() {
    if (count == 0) {
      // Bufor jest pusty, zwróć domyślnie skonstruowany obiekt T
      return T{};
    }
    T value = arr.at(read_idx);
    read_idx = (read_idx + 1) % N;
    --count;
    return value;
  }

  bool is_empty() const { return count == 0; }
};

int main() {
  ring_buffer<int, 5> rb;

  for (int i = 0; i < 10; ++i) {
    rb.push(i);
  }

  while (!rb.is_empty()) {
    printf("%d\n", rb.pop());
  }

  return 0;
}
```

Implementacja bufora cyklicznego w C++ z wykorzystaniem szablonów jest bardziej czytelna i łatwiejsza w użyciu niż rozwiązanie oparte na wklejaniu tokenów w C. Szablon klasy `ring_buffer` może być używany do tworzenia buforów cyklicznych z różnymi typami bazowymi, takimi jak liczby całkowite, zmiennoprzecinkowe lub dowolne inne typy o różnych wielkościach. Ta sama logika dodawania i usuwania elementów może być stosowana do buforów cyklicznych z różnymi typami bazowymi. Dzięki szablonom możemy do różnych typów zastosować zasadę **nie powtarzaj się** (ang. *Don't repeat yourself*, **DRY**).

Szablony ułatwiają implementację typów generycznych, co jest dość trudne i rozwlekłe w języku C.

Szablony są również wykorzystywane w **metaprogramowaniu z użyciem szablonów** (ang. *template metaprogramming*, **TMP**), czyli technice programowania, w której kompilator używa szablonów do generowania tymczasowego kodu źródłowego, który następnie jest łączony z pozostałą częścią kodu i kompilowany. Jednym z najbardziej znanych przykładów TMP jest **obliczanie silni w trakcie kompilacji**. TMP to zaawansowana technika, którą omawiam w rozdziale 8. Nowoczesny język C++ oferuje również specyfikator `constexpr`, czyli znacznie bardziej przyjazną dla początkujących technikę obliczeń w trakcie kompilacji.

constexpr

Standard C++11 wprowadził specyfikator `constexpr`, który umożliwia obliczenie wartości funkcji lub zmiennej na etapie kompilacji. Funkcjonalność tego specyfikatora ewoluowała z czasem, rozszerzając jego możliwości. Zmienna oznaczona jako `constexpr` musi zostać natychmiast zainicjalizowana, a jej typ musi być typem literalnym (`int`, `float` itp.). Oto jak deklarujemy zmienną `constexpr`:

```
constexpr double pi = 3.14159265359;
```

W C++ użycie specyfikatora `constexpr` jest preferowanym sposobem deklarowania stałych znanych w trakcie kompilacji, w przeciwieństwie do stosowania makr w stylu języka C. Przeanalizujmy prosty przykład, który wykorzystuje makra w stylu C:

```
#include <cstdio>

#define VOLTAGE 3300
#define CURRENT 1000

int main () {
    const float resistance = VOLTAGE / CURRENT;
    printf("resistance = %.2f\r\n", resistance);

    return 0;
}
```

Dane wyjściowe tego prostego programu mogą okazać się zaskakujące:

```
resistance = 3.00
```

Zarówno `VOLTAGE`, jak i `CURRENT` są interpretowane jako literały całkowite, podobnie jak wynik ich dzielenia. Literały zmiennoprzecinkowe deklaruje się za pomocą przyrostka `f`, który w tym przypadku został pominięty. Bezpieczniejszym sposobem definiowania stałych czasu kompilacji jest użycie specyfikatora `constexpr`, który pozwala określić typ stałej. Oto jak ten sam przykład można zapisać z użyciem `constexpr`:

```
#include <cstdio>
constexpr float voltage = 3300;
constexpr float current = 1000;

int main () {
    const float resistance = voltage / current;
```

```
    printf("resistance = %.2f\r\n", resistance);
    return 0;
}
```

Kod spowoduje wyświetlenie następujących danych wyjściowych:

```
resistance = 3.30
```

Ten prosty przykład pokazuje, że stałe czasu kompilacji, które zostały zdefiniowane za pomocą specyfikatora `constexpr`, są zarówno bezpieczniejsze, jak i łatwiejsze do odczytania niż tradycyjne stałe makr w stylu języka C. Innym ważnym zastosowaniem tego specyfikatora jest wskazanie kompilatorowi, że funkcja może być obliczona w trakcie kompilacji. Niektóre z wymogów, które musi spełniać funkcja oznaczona specyfikatorem `constexpr`, są następujące:

- zwracany typ musi być typem literalnym;
- każdy z parametrów funkcji musi być typem literalnym;
- jeśli funkcja `constexpr` nie jest konstruktorem, musi zawierać dokładnie jedno polecenie `return`.

Przyjrzyjmy się prostemu przykładowi wykorzystania funkcji, która zostanie użyta wraz ze specyfikatorem `constexpr`:

```
int square(int a) {
    return a*a;
}

int main () {
    int ret = square(2);
    return ret;
}
```

Aby lepiej zrozumieć, co dzieje się pod maską, przyjrzymy się kodowi asemblera wygenerowanemu na podstawie powyższego programu. Asembler jest bardzo bliski kodowi maszynowemu, czyli instrukcjom, które będą wykonywane w urządzeniu docelowym. Analiza kodu asemblera daje nam więc przybliżony obraz pracy (liczby instrukcji), jaka zostanie wykonana przez procesor. W kolejnym fragmencie przedstawiłem kod asemblera wygenerowany w wyniku kompilacji omawianego programu dla architektury ARM przy użyciu kompilatora ARM GCC bez optymalizacji:

```
square(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
main:
        push    {r7, lr}
        sub     sp, sp, #8
```

```
        add     r7, sp, #0
        movs    r0, #2
        bl      square(int)
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mov     r0, r3
        adds    r7, r7, #8
        mov     sp, r7
        pop     {r7, pc}
```

Wynikowy kod asemblera wykonuje następujące operacje:

- przeprowadza operacje na wskaźniku stosu;
- wywołuje funkcję `square`;
- zapisuje wartość zwróconą przez `r0`, pod adresem zawartym w `r7` z przesunięciem 4;
- do rejestru `r3` wczytuje wartość spod adresu przechowywanego w rejestrze `r7`, z przesunięciem 4;
- przenosi wartość z rejestru `r3` do `r0`, który zgodnie z konwencją wywoływania ARM jest przeznaczony do przechowywania wartości zwrotnych.

Możemy zauważyć, że w wygenerowanym kodzie binarnym występują zbędne operacje, które zwiększają wielkość pliku wykonywalnego i wpływają niekorzystnie na wydajność. Ten przykład jest poprawnym kodem zarówno w C, jak i w C++, a jego kompilacja w obu przypadkach prowadzi do wygenerowania takiego samego kodu asemblera.

Jeśli użyjemy specyfikatora `constexpr` dla funkcji `square`, wówczas poinformujemy kompilator, że w trakcie kompilacji możliwe jest obliczenie jej wartości zwrotnej:

```
constexpr int square(int a) {
    return a*a;
}

int main() {
    constexpr int val = square(2);
    return ret;
}
```

Ten kod oblicza wartość wyrażenia `square(2)` w trakcie kompilacji, co powoduje, że zmienna `val` staje się zmienną oznaczoną specyfikatorem `constexpr`, czyli stałą, której wartość jest znana podczas kompilacji. Oto wynikowy kod asemblera:

```
main:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        movs    r3, #4
        str     r3, [r7, #4]
        movs    r3, #4
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Jak widać, program zwraca wartość `4`, która jest wynikiem obliczenia, w trakcie kompilacji, wyrażenia `square(2)`. W wygenerowanym kodzie asemblera nie ma funkcji `square`, lecz jedynie wynik obliczenia, które kompilator wykonał za nas. Ten prosty przykład pokazuje potężne możliwości, jakie kryją się w obliczeniach wykonywanych w trakcie kompilacji. Zatem złożone obliczenia, gdy znamy wszystkie ich parametry, co zdarza się często, możemy przenieść z etapu wykonywania programu do etapu kompilacji. Takie podejście można wykorzystać do generowania tablic przeglądowych lub złożonych sygnałów matematycznych, co zostanie zaprezentowane w kolejnych rozdziałach tej książki.

Język C++ przeszedł długą drogę od czasów „C z klasami". Przykłady zamieszczone w tym rozdziale pokazują, co C++ może zaoferować w porównaniu do C: bardziej ekspresyjny, czytelny i zwięzły kod, kontenery z biblioteki standardowej, algorytmy, zdefiniowane przez użytkownika typy generyczne, obliczenia w trakcie kompilacji — a to dopiero początek. Mam nadzieję, że udało się obalić mit, zgodnie z którym C++ to tylko C z klasami. Kolejnym powszechnym mitem na temat C++ jest to, że prowadzi on do utworzenia rozbudowanego kodu i dodatkowych kosztów w czasie wykonywania programu. Kontynuujmy więc obalanie mitów o C++!

Nadmiar kodu i obciążenie wydajności

Termin **bloatware** odnosi się do niepożądanego oprogramowania, które jest preinstalowane wraz z systemem operacyjnym w urządzeniu. W świecie programowania niepożądane oprogramowanie to kod wstawiony do pliku wykonywalnego przez framework, bibliotekę lub samą konstrukcję języka. W przypadku C++ do konstrukcji językowych, które są obwiniane o to, że prowadzą do nadmiernego rozrostu kodu, zaliczamy konstruktory, destruktory i szablony. Przeanalizujemy te błędne przekonania na przykładzie kodu asemblera, który został wygenerowany na podstawie kodu C++.

Konstruktory i destruktory

Oto pierwsza kwestia, jaka przychodzi na myśl programistom, którzy nie korzystają z C++, gdy wspomina się o tym języku: jest on obiektowy i trzeba w nim tworzyć egzemplarze obiektów. Obiekty są egzemplarzami klas. To zmienne, które zajmują pamięć. Do tworzenia egzemplarzy obiektów używa się funkcji specjalnych zwanych **konstruktorami**.

Konstruktory służą do inicjalizowania obiektów, w tym elementów składowych klasy, destruktory zaś są używane do zwalniania zasobów. Wymienione funkcje są ściśle związane z cyklem życiowym obiektu. Zatem obiekt jest tworzony przy użyciu konstruktora, natomiast gdy zmienna reprezentująca obiekt wychodzi poza zakres, wówczas jest wywoływany **destruktor**.

Konstruktory i destruktory zwiększają plik wykonywalny, a także dodają koszt w postaci dłuższego czasu wykonania programu, ponieważ wywołanie tych funkcji wymaga nieco czasu. Przyjrzyjmy się wpływowi konstruktorów i destruktorów na prostym przykładzie klasy z jednym prywatnym elementem składowym, konstruktorem, destruktorem i metodą typu getter:

```
class MyClass
{
    private:
         int num;
    public:
        MyClass(int t_num):num(t_num){}
        ~MyClass(){}

        int getNum() const {
            return num;
        }
};

int main () {
   MyClass obj(1);
   return obj.getNum();
}
```

Klasa `MyClass` jest bardzo prosta, ma tylko jedną prywatną zmienną składową, która została zdefiniowana przez konstruktor. Dostęp do tej zmiennej jest możliwy poprzez metodę typu getter. Ponadto dla porządku zadeklarowano też pusty destruktor. Oto wygenerowany w asemblerze odpowiednik omawianego kodu, który został skompilowany bez włączonej optymalizacji:

```
MyClass::MyClass(int) [base object constructor]:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        str     r1, [r7]
        ldr     r3, [r7, #4]
        ldr     r2, [r7]
        str     r2, [r3]
        ldr     r3, [r7, #4]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
MyClass::~MyClass() [base object destructor]:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
MyClass::getNum() const:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
```

```
        ldr     r3, [r7, #4]
        ldr     r3, [r3]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
main:
        push    {r4, r7, lr}
        sub     sp, sp, #12
        add     r7, sp, #0
        adds    r3, r7, #4
        movs    r1, #1
        mov     r0, r3
        bl      MyClass::MyClass(int) [complete object constructor]
        adds    r3, r7, #4
        mov     r0, r3
        bl      MyClass::getNum() const
        mov     r4, r0
        nop
        adds    r3, r7, #4
        mov     r0, r3
        bl      MyClass::~MyClass() [complete object destructor]
        mov     r3, r4
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        pop     {r4, r7, pc}
```

Nie przejmuj się kodem asemblera, jeśli go nie rozumiesz. Widać, że zawiera etykiety dla funkcji i mnóstwo instrukcji. To jest duża ilość kodu jak na prostą abstrakcję klasy — to właśnie jest ten nadmiarowy kod, którego nie chcemy w pliku wykonywalnym. Dokładniej mówiąc, mamy tutaj 59 wierszy kodu asemblera. Gdybyśmy włączyli optymalizację, otrzymany kod asemblera miałby zaledwie kilka wierszy. Jednak w niniejszym rozdziale analizujmy ten problem bez użycia optymalizacji. Pierwszą widoczną kwestią jest to, że destruktor nie wykonuje żadnych użytecznych zadań. Jeśli usuniemy go z kodu w C++, otrzymany kod asemblera, który będzie miał 44 wiersze:

```
MyClass::MyClass(int) [base object constructor]:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        str     r1, [r7]
        ldr     r3, [r7, #4]
        ldr     r2, [r7]
        str     r2, [r3]
        ldr     r3, [r7, #4]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

```
MyClass::getNum() const:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        ldr     r3, [r3]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
main:
        push    {r7, lr}
        sub     sp, sp, #8
        add     r7, sp, #0
        adds    r3, r7, #4
        movs    r1, #1
        mov     r0, r3
        bl      MyClass::MyClass(int) [complete object constructor]
        adds    r3, r7, #4
        mov     r0, r3
        bl      MyClass::getNum() const
        mov     r3, r0
        nop
        mov     r0, r3
        adds    r7, r7, #8
        mov     sp, r7
        pop     {r7, pc}
```

Jak widać, w pliku binarnym nie ma wywołania destruktora ani kodu destruktora. Wniosek z tego płynie taki, że *nie płacimy za to, czego nie używamy*. To jest jedna z zasad projektowych C++. Dzięki usunięciu destruktora kompilator nie musi generować dla niego kodu ani wywoływać go, gdy zmienna obiektowa wychodzi poza zakres.

Przejdźmy do kolejnej kwestii, którą musimy sobie uświadomić: C++ nie jest językiem czysto obiektowym. To język, w którym wykorzystano wiele paradygmatów — jest jednocześnie proceduralny, obiektowy, generyczny, a nawet trochę funkcyjny. Jeśli chcemy mieć prywatne elementy składowe, które można definiować tylko za pomocą konstruktorów, musimy za to zapłacić pewną cenę. Struktury w C++ domyślnie mają publiczne elementy składowe, więc zmieńmy klasę `MyClass` na strukturę `MyClass` bez konstruktora:

```
struct MyClass
{
    int num;
};

int main () {

   MyClass obj(1);

   return obj.num;
}
```

Wprawdzie funkcje typu setter i getter są powszechne w paradygmacie programowania obiektowego, ale C++ nie jest (tylko) językiem obiektowym, więc nie jesteśmy zmuszeni do ich używania. Zatem po usunięciu funkcji `getNum` otrzymujemy bardzo prosty przykład struktury z tylko jednym elementem. Wynikowy kod asemblerowy ma zaledwie 14 wierszy:

```
main:
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        movs    r3, #1
        str     r3, [r7, #4]
        ldr     r3, [r7, #4]
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Choć ten przykład jest wręcz trywialny, jego celem jest ustalenie dwóch podstawowych prawd:

- nie płacisz za to, czego nie używasz;
- korzystanie z C++ nie oznacza, że jesteś ograniczony do paradygmatu programowania obiektowego.

Jeśli chcemy korzystać z abstrakcji takich jak konstruktory i destruktory, musimy ponieść koszt związany z wielkością pliku binarnego. Użycie typów (klas i struktur) bez tworzenia obiektów w C++ daje znaczące korzyści podczas projektowania aplikacji dla systemów wbudowanych. Wykraczają one poza te, które są dostępne w przypadku tradycyjnego podejścia obiektowego. Przyjrzymy się temu bliżej na konkretnych przykładach w kolejnych rozdziałach.

W tym i w poprzednich przykładach kod w C++ kompilowaliśmy z wyłączonymi optymalizacjami, co pozwoliło zobaczyć w wygenerowanym kodzie asemblera zbędne operacje, które można usunąć. Sprawdźmy teraz kod asemblera dla ostatniego przykładu z włączonym poziomem optymalizacji O3:

```
main:
        movs    r0, #1
        bx      lr
```

Powyższy kod asemblera został wygenerowany dla oryginalnego przykładu z klasą, konstruktorem, destruktorem i funkcją dostępową. Wynikowy program składa się z zaledwie dwóch instrukcji. Wartość składowej `num` zmiennej `obj` jest przechowywana w rejestrze `r0` jako wartość zwrotna. Kod asemblera został pozbawiony wszystkich zbędnych instrukcji związanych z operacjami na stosie i użyciem rejestru `r3` do przechowywania wartości za pomocą wskaźnika stosu z przesunięciem `4` i z ponownym wczytywaniem wartości do `r3` i przenoszenia jej do `r0`. W efekcie otrzymujemy zaledwie kilka wierszy kodu asemblera.

Usuwanie niepotrzebnych instrukcji to zadanie dla procesu optymalizacji. Jednak w projektach przeznaczonych dla systemów wbudowanych często unika się optymalizacji, ponieważ niektórzy twierdzą, że psuje ona kod. Ale czy to prawda?

Optymalizacja

Nieoptymalny kod skutkuje niepotrzebnymi instrukcjami, które wpływają niekorzystnie na wielkość pliku wykonywalnego i wydajność programu. Mimo to wiele projektów dla systemów wbudowanych nadal kompiluje się z wyłączoną optymalizacją, ponieważ programiści *nie ufają kompilatorowi* i obawiają się, że *zepsuje on program*. Wprawdzie jest w tym trochę prawdy, ale jak się okazuje, dzieje się tak tylko wtedy, gdy program został utworzony niepoprawnie. Program nie jest poprawny, jeśli zawiera niezdefiniowane zachowania.

Jednym z najbardziej znanych przykładów niezdefiniowanego zachowania jest **przepełnienie liczby całkowitej** ze znakiem. Standard nie określa, co się stanie, gdy dodamy `1` do maksymalnej wartości liczby całkowitej ze znakiem na danej platformie. W takiej sytuacji skompilowany program nie musi robić nic sensownego. Program, który zawiera takie operacje, nie jest poprawnie utworzony. Przyjrzyjmy się następującemu fragmentowi kodu:

```
#include <cstdio>
#include <limits>

int foo(int x) {
    int y = x + 1;
    return y > x;
}

int main() {

    if(foo(std::numeric_limits<int>::max())) {
        printf("X is larger than X + 1\r\n");
    }
    else {
        printf("X is NOT larger than X + 1. Oh nooo !\r\n");
    }
    return 0;
}
```

Kompilacja kodu za pomocą GCC dla architektury zarówno x86, jak i Arm Cortex-M4 prowadzi do tego samego wyniku. Jeśli program zostanie skompilowany bez optymalizacji, wówczas wywołanie funkcji `foo` zwróci `0`, w wyniku zaś pojawi się komunikat **X is NOT larger than X + 1. Oh nooo !**. Kompilator nie reaguje na przepełnienie liczby całkowitej, natomiast jeśli funkcji `foo` przekażemy maksymalną wartość typu `integer`, to jej wartością zwrotną będzie `0`. Należy pamiętać, że standard nie określa tego zachowania i zależy ono od konkretnego kompilatora.

Jeśli skompilujemy program z włączoną optymalizacją, wynikiem będzie komunikat **X is NOT larger than X + 1. Oh nooo !**, co oznacza, że wartością zwrotną funkcji `foo` jest `1`.

Spójrz teraz na kod w języku asemblera wygenerowany dla programu skompilowanego z optymalizacją:

```
foo(int):
        movs    r0, #1
        bx      lr
.LC0:
        .ascii  "X is larger then X + 1\015\000"
main:
        push    {r3, lr}
        movw    r0, #:lower16:.LC0
        movt    r0, #:upper16:.LC0
        bl      puts
        movs    r0, #0
        pop     {r3, pc}
```

Jak widać, funkcja `foo` nie wykonuje żadnych obliczeń. Kompilator zakłada, że program jest poprawnie skonstruowany i nie występuje w nim niezdefiniowane zachowanie. Funkcja `foo` zawsze zwróci wartość `1`. To programista odpowiada za zagwarantowanie, że w programie nie będzie niezdefiniowanego zachowania. To właśnie dlatego wciąż jest żywy mit o tym, że optymalizacja psuje program. Łatwiej jest obwinić kompilator za brak obsługi niezdefiniowanego zachowania.

Oczywiście możliwe jest wystąpienie w kompilatorze błędu, który zakłóca działanie programu, kiedy stosowana jest optymalizacja, podczas gdy program działa poprawnie bez niej. Jednak taka sytuacja zdarza się naprawdę bardzo rzadko. Dlatego stosuje się techniki weryfikacji, takie jak testy jednostkowe i integracyjne, które zapewniają poprawne działanie kodu niezależnie od tego, czy jest on skompilowany z włączoną optymalizacją, czy bez niej.

Optymalizacja polega na zmniejszaniu wielkości pliku wykonywalnego i poprawie wydajności działania programu poprzez usunięcie zbędnych instrukcji z kodu maszynowego. Niezdefiniowane zachowanie zależy od kompilatora i musi zostać obsłużone przez programistę, aby została zapewniona poprawność działania programu. Należy stosować techniki takie jak testy jednostkowe i integracyjne, aby zweryfikować funkcjonalność programu i zminimalizować ryzyko jego zniekształcenia przez kompilator. Proces optymalizacji jest niezbędny do efektywnego wykorzystania abstrakcji w kodzie w C++, przy jednoczesnym zachowaniu minimalnej wielkości pliku wykonywalnego i maksymalnej wydajności działania programu. W dalszej części książki będziemy korzystać z najwyższego poziomu optymalizacji, `O3`.

Szablony to kolejny podejrzany o zwiększenie wielkości. Przyjrzymy im się w następnym punkcie. W jaki sposób zwiększają one kod źródłowy i jaką wartość wnoszą do projektów przeznaczonych dla systemów wbudowanych?

Szablony

Tworzenie egzemplarzy **szablonów** z różnymi parametrami powoduje, że kompilator generuje oddzielne typy, co ma zdecydowanie niekorzystny wpływ na wielkość pliku wykonywalnego. Jest to oczekiwane zachowanie. Podobną sytuację mamy w przypadku implementacji bufora cyklicznego w języku C przy użyciu operatora wklejania tokenów

i makr. Alternatywą jest wymazywanie typów, które zastosowaliśmy w implementacji C za pomocą wskaźnika `void`. Jednak takie rozwiązanie traci na elastyczności, jeśli narzucimy ograniczenie statycznej alokacji danych, a także na wydajności z powodu pośredniej adresacji przez wskaźnik.

Korzystanie z typów generycznych jest kwestią wyboru projektowego. Możemy je stosować, a ceną za to jest zwiększenie pliku wykonywalnego. Taka sama sytuacja miałaby miejsce, gdybyśmy implementowali oddzielne bufory cykliczne dla różnych typów danych (`ring_buffer_int`, `ring_buffer_float` itp.). Opracowanie jednego typu w postaci szablonu jest znacznie łatwiejsze niż poprawianie tego samego błędu w kilku miejscach w kodzie. Wykorzystanie typów generycznych nie powoduje, że plik wykonywalny jest większy niż w przypadku równoważnej implementacji z wykorzystaniem osobnych typów. Przyjrzyjmy się wpływowi szablonów na wielkość pliku wykonywalnego w porównaniu z oddzielnymi implementacjami na przykładzie bufora cyklicznego:

```
int main() {
#ifdef USE_TEMPLATES
  ring_buffer<int, 10> buffer1;
  ring_buffer<float, 10> buffer2;
#else
  ring_buffer_int buffer1;
  ring_buffer_float buffer2;
#endif
  for (int i = 0; i < 20; i++) {
    buffer1.push(i);
    buffer2.push(i + 0.2f);
  }

  for (int i = 0; i < 10; i++) {
    printf("%d, %.2f\r\n", buffer1.pop(), buffer2.pop());
  }

  return 0;
}
```

Jeśli zostanie skompilowany z zdefiniowaną opcją `USE_TEMPLATES`, wówczas przedstawiony program będzie używał ogólnego typu `ring_buffer`. W przeciwnym razie użyje typów `ring_buffer_int` i `ring_buffer_float`. Jeżeli skompilujemy ten przykład za pomocą kompilatora GCC bez włączonej optymalizacji, wersja z szablonami będzie miała nieco większy plik wynikowy (o 24 bajty). To wynika z prostego faktu: tablica symboli w programie w wersji z szablonami ma większe elementy. Jeśli usuniemy tablicę symboli z plików obiektowych, wielkość plików wykonywalnych będzie taka sama. Ponadto kompilacja obu wersji z optymalizacją `03` prowadzi do wygenerowania pliku wykonywalnego o takiej samej wielkości.

Typy generyczne nie prowadzą do powstawania większych plików wynikowych, przynajmniej nie bardziej niż w przypadku ręcznego zdefiniowania oddzielnych typów dla poszczególnych egzemplarzy. Szablony mają wpływ na czas kompilacji ze względu na tworzenie konkretnych typów w różnych jednostkach kompilacji. Jednak istnieją techniki, które pozwalają tego uniknąć, o ile to konieczne. Wszystkie funkcje związane z egzemplarzami typów o tych samych parametrach zostaną połączone w jedną funkcję w pliku wynikowym, ponieważ linker usunie powtarzające się symbole.

RTTI i wyjątki

Informacja o typie w czasie wykonywania (ang. *runtime type information*, **RTTI**) w C++ to mechanizm, który umożliwia określenie typu obiektu podczas działania programu. Większość kompilatorów implementuje RTTI za pomocą tablic wirtualnych. Każda klasa polimorficzna (czyli klasa z co najmniej jedną funkcją wirtualną) ma tablicę wirtualną, która oprócz innych informacji zawiera także dane o typie, potrzebne do jego identyfikacji w trakcie wykonywania programu. RTTI wiąże się z kosztami (czas i pamięć), ponieważ zwiększa plik wykonywalny i zmniejsza wydajność działania programu, jeśli używana jest identyfikacja typów. Z tego powodu kompilatory oferują możliwość wyłączenia RTTI. Przyjrzyjmy się prostemu przykładowi z klasą bazową i pochodną:

```
#include <cstdio>

struct Base {
    virtual void print () {
        printf("Base\r\n");
    }
};
struct Derived : public Base {
    void print () override {
        printf("Derived\r\n");
    }
};

void printer (Base &base) {
    base.print();
}

int main() {
    Base base;
    Derived derived;
    printer(base);
    printer(derived);

  return 0;
}
```

Wynik działania programu jest następujący:

```
Base
Derived
```

Klasy z funkcjami wirtualnymi mają tablice funkcji wirtualnych (`vtable`), które są wykorzystywane na potrzeby dynamicznego wiązania, czyli procesu wyboru odpowiedniej implementacji funkcji polimorficznej. Funkcja `printer` przyjmuje referencję do klasy bazowej. W zależności od typu referencji przekazanej do funkcji `printer` (klasa bazowa lub pochodna) mechanizm dynamicznego wiązania wybierze metodę `print` z klasy bazowej lub pochodnej. Tablice `vtable` służą również do przechowywania informacji o typie.

Jeśli użyjesz operatora `dynamic_cast`, który jest częścią składową mechanizmu RTTI, możesz otrzymać informacje o typie obiektu — wystarczy skorzystać z referencji lub wskaźnika do klasy bazowej. Zmodyfikujmy teraz metodę `printer` z poprzedniego przykładu:

```
void printer (Base &base) {
    base.print();
    if(Derived *derived = dynamic_cast<Derived*>(&base); derived!=nullptr) {
        printf("We found Base using RTTI!\r\n");
    }
}
```

Wynik działania kodu jest następujący:

```
Base
Derived
We found Base using RTTI!
```

Jak już wspomniałem, mechanizm RTTI może zostać wyłączony. W przypadku kompilatora GCC następuje to po użyciu flagi `-fno-rtti`. Jeśli zmodyfikowany przykład spróbujemy skompilować z użyciem tej flagi, zostanie wygenerowany błąd `error: dynamic_cast' not permitted with '-fno-rtti'`. Jeśli przywrócimy pierwotną implementację metody `printer`, usuniemy instrukcję warunkową `if` i skompilujemy program, z włączonym, a następnie z wyłączonym mechanizmem RTTI, to zauważymy, że plik wykonywalny jest większy po włączeniu RTTI. Wprawdzie ten mechanizm okazuje się przydatny w niektórych scenariuszach, ale jednocześnie oznacza znacznie większe obciążenie dla urządzeń o ograniczonych zasobach i dlatego nie będziemy z niego korzystać.

Wyjątki to inna funkcja C++, która często jest wyłączana w projektach dla systemów osadzonych. Jest to oparty na blokach `try-catch` mechanizm obsługi błędów. Przyjrzyjmy się prostemu przykładowi, który pomoże zrozumieć wyjątki:

```
#include <cstdio>

struct A {
  A() { printf("A is created!\r\n"); }
  ~A() { printf("A is destroyed!\r\n"); }
};

struct B {
  B() { printf("B is created!\r\n"); }
  ~B() { printf("B is destroyed!\r\n"); }
};

void bar() {
    B b;
    throw 0;
}
void foo() {
  A a;
  bar();
  A a1;
}

int main() {
  try {
    foo();
  } catch (int &p) {
    printf("Catching an exception!\r\n");
  }
```

```
    return 0;
}
```

Dane wyjściowe programu są następujące:

```
A is created!
B is created!
B is destroyed!
A is destroyed!
Catching an exception!
```

W tym prostym przykładzie funkcja `foo` jest wywoływana w bloku `try`. Tworzy ona lokalny obiekt, `a`, i wywołuje funkcję `bar`. Następnie funkcja `bar` tworzy lokalny obiekt, `b`, i zgłasza wyjątek. W wyniku tego widzimy, że tworzone są egzemplarze klas `A` i `B`, następnie jest niszczony `B`, a później `A`, na końcu zaś będzie wykonywany blok `catch`. Ten proces nazywa się **rozwijaniem stosu**. Aby go zrealizować, standardowe implementacje najczęściej wykorzystują tablice rozwijania, które przechowują informacje o procedurach obsługi wyjątków, destruktorach do wywołania itp. Tablice rozwijania mogą stać się duże i skomplikowane, co zwiększa zużycie pamięci aplikacji oraz wprowadza nieprzewidywalność ze względu na mechanizm obsługi wyjątków używany w czasie wykonywania programu. Z tego powodu wyjątki są często wyłączane w projektach dla systemów wbudowanych.

Podsumowanie

C++ kieruje się **zasadą zerowego kosztu**. Jedyne elementy języka, które nie są z nią zgodne, to mechanizm RTTI i wyjątki. Dlatego kompilatory umożliwiają ich wyłączenie.

Zasada zerowego kosztu opiera się na dwóch stwierdzeniach, które ustaliliśmy w tym rozdziale:

- nie płacisz za to, czego nie używasz;
- to, czego używasz, jest równie wydajne jak to, co mógłbyś rozsądnie zdefiniować samodzielnie.

RTTI i wyjątki są wyłączone w większości projektów dla systemów osadzonych, więc nie ponosisz związanych z nimi kosztów. Korzystanie z typów generycznych i szablonów to wybór projektowy. Takie rozwiązanie nie prowadzi do wyższych kosztów w porównaniu do samodzielnego zdefiniowania poszczególnych typów (`ring_buffer_int`, `ring_buffer_float` itp.). Pozwala jednak ponownie wykorzystać logikę kodu dla różnych typów, a tym samym powoduje, że kod jest czytelniejszy i łatwiejszy w późniejszej obsłudze.

Praca nad systemami wysokiego ryzyka nie jest powodem do wyłączania optymalizacji kompilatora. Funkcjonalność kodu musi być weryfikowana niezależnie od tego, czy budujemy program z wyłączoną, czy włączoną optymalizacją. Najczęstszym źródłem błędów przy włączonej optymalizacji jest niezdefiniowane zachowanie. Zrozumienie niezdefiniowanego zachowania i zapobieganie mu leży w gestii programisty.

Nowoczesny C++ to język, który ma wiele do zaoferowania światu systemów wbudowanych. Celem niniejszej książki jest pomóc Ci odkryć C++ i to, co może on zrobić dla

Twoich projektów przeznaczonych dla systemów wbudowanych. Zatem wyruszmy w podróż, w której trakcie będziesz odkrywać C++ i zobaczysz, jak można go wykorzystywać do rozwiązywania problemów związanych z systemami wbudowanymi.

W następnym rozdziale omówię wyzwania pojawiające się podczas pracy z systemami wbudowanymi o ograniczonych zasobach oraz związane z zarządzaniem pamięcią dynamiczną w C++.

Nowoczesny C++: obal mity i twórz doskonały kod!

Programiści systemów wbudowanych najczęściej korzystają z języka C ze względu na jego prostotę i przystępność. Jednak prostota C sprawia, że tworzenie złożonych systemów jest trudne, a kod staje się podatny na błędy. Tych problemów można uniknąć, wybierając C++, który świetnie się sprawdza w systemach wbudowanych za sprawą takich cech jak programowanie generyczne, polimorfizm czy zwiększone bezpieczeństwo typów i pamięci.

Dzięki tej praktycznej książce nauczysz się wykorzystywać zaawansowane możliwości nowoczesnego języka C++, aby zachować wydajność przy jednoczesnym tworzeniu bezpieczniejszego i bardziej przejrzystego kodu. Rozpoczniesz od konfiguracji środowiska programistycznego, a następnie dowiesz się, jak bezpiecznie używać niektórych elementów biblioteki standardowej C++ w środowiskach o ograniczonych zasobach. Zapoznasz się też z biblioteką Embedded Template Library (ETL). Zrozumiesz podstawowe i zaawansowane koncepcje języka C++, takie jak szablony, silne typowanie, obsługa błędów, obliczenia w trakcie kompilacji i technika RAII. Na podstawie praktycznych przykładów zaimplementujesz sekwencer, utworzysz bezpieczną warstwę abstrakcji sprzętowej i zastosujesz wzorce projektowe do realizacji typowych scenariuszy w programowaniu systemów wbudowanych.

W książce:

- zalety języka C++ w systemach wbudowanych
- kompilacja kodu C++ w środowiskach o ograniczonych zasobach
- silne typowanie i poprawa bezpieczeństwa typów
- inicjalizacja zasobów (RAII) i inne techniki nowoczesnego C++
- praktyczne użycie biblioteki Boost SML
- najlepsze praktyki tworzenia oprogramowania w systemach wbudowanych

Amar Mahmutbegović jest ekspertem w zakresie użycia nowoczesnego C++ do programowania systemów wbudowanych. Zajmuje się zaawansowanymi urządzeniami medycznymi oraz takimi, które wykorzystują technologię Bluetooth Low Energy. Jest mentorem dla młodych inżynierów, pomaga im rozwijać umiejętności w obszarze nowoczesnych praktyk programistycznych.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-289-3560-0
9 788328 935600

Cena: 89,00 zł

‹packt›