Metody obsługi błędów

goto

```
#include <iostream>

bool isValid() { /* ... */ }

int main() {
    // ...
    if(!isValid()) {
        goto error;
    }

error:
    std::cerr << "Error occured" << '\n';
    return 1;
}
```

errno

W programowaniu w C/C++ jest też koncepcja `errno`, czyli statycznej współdzielonej zmiennej, którą ustawia się na odpowiednią wartość w przypadku wystąpienia błędu. Zobacz `errno` na cppreference.

```
int main() {
    double not_a_number = std::log(-1.0);
    std::cout << not_a_number << '\n';
    if (errno == EDOM) {
        std::cout << "log(-1) failed: " << std::strerror(errno) << '\n';
        std::setlocale(LC_MESSAGES, "de_DE.utf8");
        std::cout << "Or, in German, " << std::strerror(errno) << '\n';
    }
}
```

Kody powrotu / kody błędów

```
#include <iostream>

enum class ErrorCode { Ok, FailCause1, FailCause2 };
bool isValid() { /* ... */ }

ErrorCode foo() {
```

```
    if(!isValid()) {
        return ErrorCode::FailCause1;
    }
    // ...
    return ErrorCode::Ok;
}

int main() {
    if(foo() == ErrorCode::FailCause1) {
        std::cerr << "Error occured" << '\n';
        return 1;
    }
    return 0;
}
```

Obsługa błędów w konstruktorach i operatorach

Konstruktory i operatory mają ściśle zdefiniowane typy zwracane (lub ich brak). Niemożliwe jest zwrócenie w nich własnego kodu powrotu.

```
struct FileWrapper {
    FileWrapper(std::string const& filePath)
            : m_file(fopen(filePath.c_str(), "rw")) {
        /* What if the file did not open? */
    }

    ~FileWrapper() {
        fclose(m_file);
    }

    FileWrapper & operator<<(std::string const& text) {
        /* What if the file did not open? */
        fputs(text.c_str(), m_file);
        return *this;
    }

private:
    FILE* m_file;
};
```

---

throw

Zamiast zwracać specjalną wartość z funkcji lub ustawiać globalną zmienną po prostu rzucamy wyjątkiem. To wskazuje, że coś poszło nie tak, a błąd możemy obsłużyć w zupełnie innym miejscu.

```
struct FileWrapper {
    FileWrapper(std::string const& filePath)
```

```
            : m_file(fopen(filePath.c_str(), "rw")) {
        if(!m_file) {
            throw std::runtime_error("File not opened");
        }
    }

    ~FileWrapper() {
        fclose(m_file);
    }

    FileWrapper & operator<<(std::string const& text) {
        /* Not validation needed, invalid object cannot be created */
        fputs(text.c_str(), m_file);
        return *this;
    }

private:
    FILE* m_file;
};
```

---

try/catch

Za pomocą `try` oznaczamy blok kodu, w którym możliwe jest rzucenie wyjątku. Bloki `catch` służą do łapania wyjątków określonych typów.

```
#include <iostream>
#include <stdexcept>

void foo() { throw std::runtime_error("Error"); }

int main() {
    try {
        foo();
    } catch(std::runtime_error const&) {
        std::cout << "std::runtime_error" << std::endl;
    } catch(std::exception const& ex) {
        std::cout << "std::exception: " << ex.what() << std::endl;
    } catch(...) {
        std::cerr << "unknown exception" << std::endl;
    }
}
```

Result

```
std::runtime_error
```

Co to jest wyjątek?

W ogólności - dowolny obiekt. Każdy obiekt może być wyjątkiem.

```
throw 42;
```

Nie jest rekomendowane używanie wbudowanych typów lub tworzonych klas jako wyjątki.

```
throw std::runtime_error{"Huston, we have a problem"};
```

Poleca się, aby wyjątki były specjalnymi klasami, które dziedziczą po innych klasach wyjątków z biblioteki standardowej. Przykładem może być `std::runtime_error`, który dziedziczy po `std::exception`.

Jak to działa?

Dopasowanie typu wyjątku

```
struct TalkingObject {
    TalkingObject()  { cout << "Constructor" << '\n'; }
    ~TalkingObject() { cout << "Destructor" << '\n'; }
};

void foo() { throw std::runtime_error("Error"); }

int main() {
    TalkingObject outside;
    try {
        TalkingObject inside;
        foo();
    } catch(runtime_error const& ex) {
        cout << "runtime_error: " << ex.what() << '\n';
    } catch(exception const&) {
        cout << "exception" << '\n';
    }
}
```

Wynik

```
Constructor

Constructor

Destructor

runtime_error: Error
```

`Destructor`

Mechanizm odwijania stosu

- Rzucony wyjątek startuje mechanizm odwijania stosu (stack unwinding mechanism)
- Typ wyjątku jest dopasowywany do kolejnych klauzul `catch`
- Wyjątek jest polimorficzny, tzn. może zostać dopasowany do typu klasy bazowej
- Jeśli typ pasuje:
  - Wszystko zaalokowane na stosie jest niszczone w odwrotnej kolejności aż do napotkania bloku `try`
  - Kod z pasującej klauzuli `catch` jest wykonywany
  - Obiekt wyjątku jest niszczony
- Jeśli typ nie pasuje do żadnej klauzuli `catch`, odwijanie stosu jest kontynuowane do kolejnego bloku `try`

Nieobsłużony wyjątek

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

void foo() { throw std::runtime_error("Error"); }

void bar() {
    try {
        TalkingObject inside;
        foo();
    } catch(std::logic_error const&) {
        std::cout << "std::logic_error" << '\n';
    }
}

int main() {
    TalkingObject outside;
    bar();
}
```

Wynik

`Constructor`

`Constructor`

`>> abort() <<`

Czemu destruktory nie zostały wywołane?

- Mechanizm odwijania stosu najpierw sprawdza, czy w obecnym bloku `try` jest pasująca klauzula `catch` jeszcze przed zniszczeniem obiektów
- Wyjątek który nie został przechwycony i wypada po funkcję main powoduje zawołanie `std::terminate()`, które ubija program.

Ponowne rzucanie wyjątków

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

void foo() { throw std::runtime_error("Error"); }

void bar() try {
    TalkingObject inside;
    foo();
} catch(std::exception const&) {
    std::cout << "exception" << '\n';
    throw;
}

int main() {
    TalkingObject outside;
    try {
        bar();
    } catch(std::runtime_error const& ex) {
        std::cout << "runtime_error: " << ex.what() << '\n';
    }
}
```

Samo `throw` w bloku `catch` powoduje ponowne rzucenie aktualnego wyjątku.

Wynik

```
Constructor

Constructor

Destructor

exception

runtime_error: Error

Destructor
```

Ponowne rzucanie wyjątku

- Wyjątek rzucony ponownie raz jeszcze uruchamia mechanizm odwijania stosu
- Odwijanie stosu jest kontynuowane do kolejnego bloku `try`
- Klauzula `catch` dla typu bazowego przechwyci też wyjątki typów pochodnych

- Ponowne rzucenie wyjątki nie zmienia oryginalnego typu wyjątku

---

Rzucenie wyjątku podczas odwijania stosu

```
struct TalkingObject { /*...*/ };
struct ThrowingObject {
    ThrowingObject() { std::cout << "Throwing c-tor\n"; }
    ~ThrowingObject() {
        throw std::runtime_error("error in destructor");
    }
};

void foo() { throw std::runtime_error("Error"); }

int main() {
    TalkingObject outside;
    try {
        ThrowingObject inside;
        foo();
    } catch(std::exception const&) {
        std::cout << "std::exception" << '\n';
        throw;
    }
}
```

Wynik

```
Constructor

Throwing c-tor

>> abort() <<
```

Wnioski

- Można obsługiwać tylko jeden wyjątek na raz
- Wyjątek rzucony podczas odwijania stosu powoduje ubicie programu - woła się `std::terminate()`
- Nigdy nie rzucaj wyjątków w destruktorach

Czy wyjątki są kosztowne?

- Moje wideo z wyjaśnieniem
- Zwykły przebieg programu
- Przebieg z wyjątkiem

Wyjątki

Zalety

- Zgłaszanie błędów i ich obsługa są rozdzielone
- Czytelność kodu wzrasta - można wyrzucić z funkcji logikę odpowiedzialną za nietypowe przypadki
- Błędy można obsługiwać i zgłaszać w konstruktorach i operatorach
- Brak dodatkowych sprawdzeń przy standardowym przebiegu programu = brak dodatkowych `if` = brak kosztu

Wady

- Rozmiar binarki jest większy (kompilator dodaje dodatkowy kod na końcu każdej funkcji, która może uczestniczyć w obsłudze wyjątków)
- Czas obsługi wyjątków jest niezdefiniowany
- Zazwyczaj potrzebne są informacje z przebiegu programu aby śledzić jego przepływ (core dump, debugger)

Wnioski

- Czas obsługi wyjątków jest niezdefiniowany
  - Zależy od liczby, rozmiaru i typu danych na stosie pomiędzy miejscem rzucenia i obsługi wyjątku
- Nie używamy wyjątków w systemach czasu rzeczywistego (RTOS) ze ściśle zdefiniowanym czasem wykonania funkcji (m.in. urządzenia medyczne, automotive)
- Aby lepiej podjąć decyzję czy warto używać wyjątków należy zmierzyć sposób użycia programu. Jeśli ścieżki wyjątkowe występują bardzo rzadko to przejście na wyjątki może spowodować ogólny wzrost wydajności.

Rekomendacje

- Używaj wyjątków z STLa - zobacz na cppreference.com
- Pisząc własne klasy wyjątków dziedzicz je po wyjątkach z STLa
  - `catch(const std::exception & e)` złapie je wszystkie
- Unikaj `catch(...)` - to łapie absolutnie wszystko i nie jest to dobrą praktyką
- Łap wyjątki przez `const &` - dzięki temu zapobiegasz niepotrzebnym kopiom obiektów wyjątków
- Używaj wyjątków tylko w nietypowych sytuacjach i nie buduj standardowego przepływu programu w oparciu o wyjątki
- Używaj słówka `noexcept`, aby wskazywać funkcje, które nie będą rzucać wyjątków. To pomaga kompilatorowi zoptymalizować program i zredukować rozmiar binarki.

Problemy z pamięcią

Quiz

Jaki tu jest problem? #1

```cpp
#include <iostream>

int main() {
    const auto size = 10;
    int* dynamicArray = new int[size];

    for (int i = 0; i <= size; ++i) {
        *(dynamicArray + i) = i * 10;
    }

    for (int i = 0; i <= size; ++i) {
        std::cout << dynamicArray[i] << '\n';
    }

    delete[] dynamicArray;
}
```

Dostęp do pamięci poza zakresem tablicy

Jaki tu jest problem? #2

```cpp
#include <iostream>

struct Msg {
    int value{100};
};

void processMsg(Msg* msg) {
    std::cout << msg->value << '\n';
}

int main() {
    Msg* m = new Msg();
    delete m;
    processMsg(m);
    return 0;
}
```

Wiszący wskaźnik (ang. dangling pointer)

Wskaźnik, który pokazuje na niepoprawną (np. usuniętą) pamięć

Jaki tu jest problem? #3

```cpp
class Msg {};
```

```
void processMsg(Msg* msg) {
    // ...
    delete msg;
}

int main() {
    Msg* m = new Msg{};
    processMsg(m);
    delete m;
}
```

Podwójne usuwanie (ang. Double delete)

Występuje gdy usuwamy wiszący wskaźnik

Jaki tu jest problem? #4

```
#include <iostream>

int main() {
    int* p = new int{10};
    delete p;
    p = nullptr;

    std::cout << *p << '\n';

    return 0;
}
```

Null pointer dereference

Występuje, gdy próbujemy wyłuskać `nullptr`

Jaki tu jest problem? #5

```
class Msg {};

void processMsg(Msg* msg) {
    // ...
    delete msg;
}

int main() {
    Msg m;
    processMsg(&m);
```

```
    return 0;
}
```

Zwalnianie pamięci zaalokowanej na stosie

Jaki tu jest problem? #6

```
int main() {
    constexpr auto size = 4u;
    int* array = new int[size]{1, 2, 3, 4};
    delete array;

    return 0;
}
```

Zwalnianie niewłaściwym operatorem delete

Używanie `delete` zamiast `delete[]`

Jaki tu jest problem? #7

```
#include <iostream>

int main() {
    int* p = new int{10};
    p = new int{20};
    std::cout << *p << '\n';
    delete p;

    return 0;
}
```

Wyciek pamięci

Zaalokowana pamięć, która nie może zostać zwolniona, bo nie mamy do niej wskaźnika

---

Problemy z dynamiczną alokacją

- accessing out-of-bounds memory
- dangling pointer
- double deleting
- `null` pointer dereference
- freeing memory blocks that were not dynamically allocated

- freeing a portion of a dynamic block
- memory leak

Wszystkie powyższe problemy powoduję niezdefiniowane zachowanie.

Można je łatwo wykryć ASANem (Address Sanitizer) lub Valgrindem. Niestety, żadne z nich nie działa na Windowsie 😕

Memory corruption detection

- Address Sanitizer (ASAN)
  - dodaj flagi kompilacji:
    - `-fsanitize=address -g`
    - `-fsanitize=leak -g`
  - uruchom program
- Valgrind
  - skompiluj program
  - odpal go pod valgrindem:
    - `valgrind /path/to/binary`
  - użyj dodatkowych sprawdzeń:
    - `valgrind --leak-check=full /path/to/binary`

Żadne nie działa na Windowsie 😕

ResourceD (5 XP)

Repo

Uruchom program resourceD pod valgrindem i sprawdź wycieki pamięci.

W tym programie wycieka pamięć. Jeśli nie widzisz tego pod valgrindem to przeanalizuj kod programu i na jego podstawie zrób co trzeba, aby można było zobaczyć wycieki pod valgrindem.

Jako dowód odpowiedniego wykonania zadania zgłoś PR do repo memory-management:resource. Wyedytuj plik valgrind-output.txt i wklej do niego output z konsoli z uruchomienia valgrinda ilustrującym wyciek pamięci. Wklej też linię zawierającą uruchomienie valgrinda.

W komentarzu pod lekcją możesz napisać dlaczego wycieki pamięci mogą czasem wystąpić, a czasem nie 😃

Bonus

+3 XP za dostarczenie do 06.10.2021 włącznie

Proste pytanie...

Ile jest możliwych sposobów wykonania się tego kodu?

```
String EvaluateSalaryAndReturnName(Employee e)
{
    if( e.Title() == "CEO" || e.Salary() > 100000 )
```

```
    {
        cout << e.First() << " " << e.Last()
             << " is overpaid" << endl;
    }
    return e.First() + " " + e.Last();
}
```

- 23 (dwadzieścia trzy)
- Wyjątki!
- Przykład - Herb Sutter, GotW#20

RAII

- Resource Acquisition Is Initialization
  - idiom / wzorzec w C++
  - każdy zasób ma właściciela
  - acquired in constructor
  - released in destructor
- Zalety
  - krótszy kod
  - jasna odpowiedzialność
  - można stosować do dowolnych zasobów
  - nie potrzeba sekcji `finally`
  - przewidywalne czasy zwalniania
  - poprawność gwarantowana przez sam język

	Acquire	**Release**
memory	new, new[]	delete, delete[]
files	fopen	fclose
locks	lock, try_lock	unlock
sockets	socket	close

Zasada 0, Zasada 5

Zasada 5

- Jeśli musisz ręcznie zaimplementować jedną z poniższych funkcji:
  - destruktor
  - konstruktor kopiujący
  - kopiujący operator przypisania
  - konstruktor przenoszący
  - przenoszący operator przypisania

- To najprawdopodobniej oznacza, że musisz zaimplementować je wszystkie.

Zasada 0

- Jeśli używasz wrapperów RAII na zasoby, nie musisz implementować żadnej z powyższych 5 funkcji.

Zasada 3

Ta zasada istniała przed C++11, gdy jeszcze nie było operacji przenoszenia. Była stosowana zamiast Zasady 5.

Warto wiedzieć, że Zasada 5 jest tylko optymalizacją Zasady 3. Jeśli nie zaimplementujemy operacji przenoszenia to tylko tracimy możliwość wydajniejszego działania programu.

Praca domowa 💻

Post-work

- Zadanie `file-wrapper` (12 XP)

Bonusy

- 3 XP za dostarczenie do 03.10.2021 23:59
- 1 XP za pracę w grupie

Zadania w repo

FileWrapper

Pamiętasz klasę `FileWrapper` z lekcji o obsłudze błędów?

```
struct FileWrapper {
    FileWrapper(std::string const& filePath)
            : m_file(fopen(filePath.c_str(), "rw")) {
        /* What if the file did not open? */
    }

    FileWrapper & operator<<(std::string const& text) {
        /* What if the file did not open? */
        fputs(text.c_str(), m_file);
        return *this;
    }

    friend std::ostream& operator<<(std::ostream& os, const FileHandler&
fh);

private:
```

```
    FILE* m_file;
};
```

Dopisz implementację klasy `FileWrapper` zgodną z RAII. Pamiętaj o:

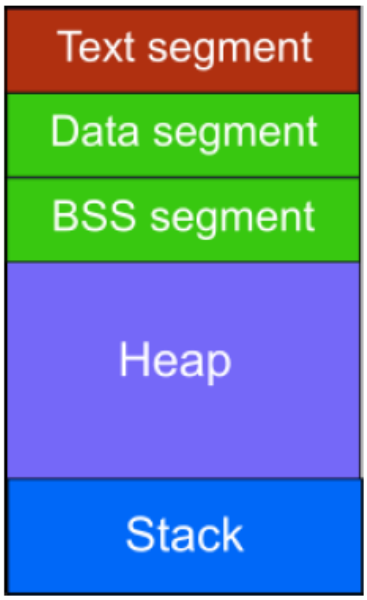
- rzuceniu wyjątku, jeśli nie udało się otworzyć pliku
- pozyskaniu zasobu w konstruktorze
- zwolnieniu zasobu w destruktorze
- zasadzie 5

Przetestuj:

- otwieranie istniejących plików `OpenExistingFile`
- otwieranie nieistniejących plików `OpenNotExistingFile`
- otwieranie katalogów `OpenDirectory`
- otwieranie plików do których nie masz uprawnień `OpenFileWithoutPermissions`
- odczyt z pliku `ReadFromFile`
- zapis do pliku `SaveToFile`
- wycieki pamięci - uruchom testy pod valgrindem

Mapa pamięci procesu

- .text - kod programu
- .rodata - dane tylko do odczytu
- .data - dane do odczytu i zapisu, zmienne globalne i statyczne
- .bss - block started by symbol = zero-initialized data
- sterta (heap) - dynamicznie zaalokowana pamięć
- stos (stack) - stos wywołań - adresy powrotu, parametry funkcji, zmienne lokalne, tymczasowe dane



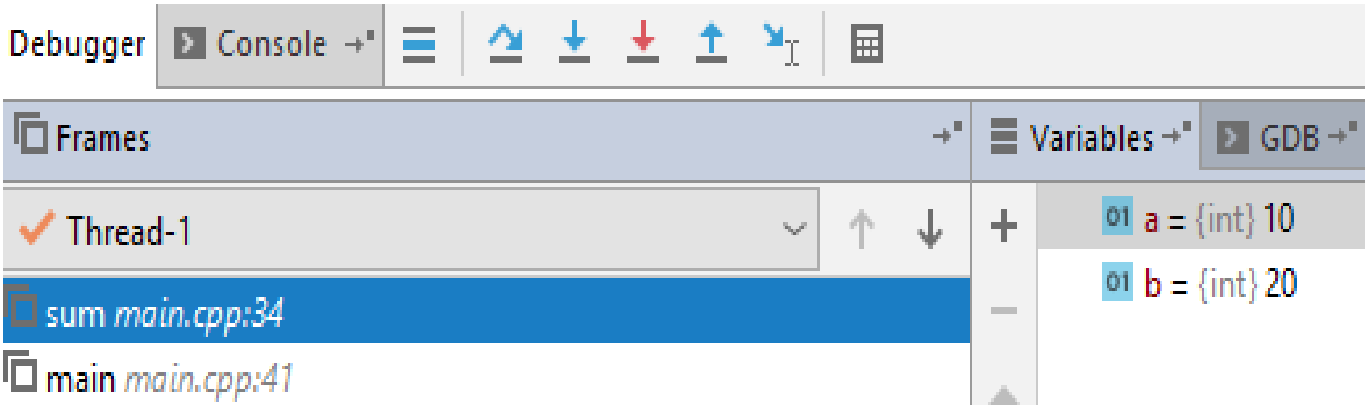
Stos vs Sterta

- Stos

    - bardzo szybki dostęp
    - limitowane miejsce (zależne od systemu operacyjnego)
    - ciągły obszar pamięci
    - automatyczne zarządzanie pamięcią przez CPU z wykorzystaniem wskaźnika stosu (Stack Pointer – SP)
- Sterta
    - wolniejszy dostęp
    - bez limitów pamięci (zarządzane przez system operacyjny)
    - pamięć może być pofragmentowana
    - ręczne zarządzanie pamięcią - alokowanie i zwalnianie

Alokacja na stosie

- Stos wywołań składa się z ramek stosu (1 funkcja = 1 ramka)
- Dokładną zawartość ramki stosu określa ABI, ale zazwyczaj składa się ona z:
    - argumentów przekazanych do funkcji
    - adresu powrotu do miejsca w funkcji wywołującej
    - miejsca na zmienne lokalne
- Zachodzi automatyczna dealokacja ramki, gdy wychodzimy poza zakres




```
#include <iostream>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 10;
    int b = 20;

    std::cout << sum(a, b);

    return 0;
}
```

Przepełnienie stosu (stack overflow)

- Stos ma ograniczony rozmiar (zależny od OS)

```
int foo()
{
    double x[1048576];
    x[0] = 10;
    return 0;
}

int main()
{
    foo();
    return 0;
}
```

Alokacja na stercie

Alokacja na stercie składa się z kilku kroków:

- alokacji wskaźnika na stosie
- alokacji `sizeof(T)` bajtów na stercie
- wywołania konstruktora `T` na zaalokowanej pamięci
- przypisania adresu do wskaźnika
- manualnego zwolnienia pamięci używając operatora `delete`

```
void heap()
{
    int *p = new int(100);
    delete p;
}

void heap()
{
    int *p;
    p = (int*)malloc(sizeof(int));
    *p = 100;
    free(p);
}
```

Wyrażenie `new` i operator `new`

wyrażenie `new` robi 3 rzeczy:

- alokuje `sizeof(T)` bajtów na stercie (za pomocą operatora `new`)
- wywołuje konstruktor `T` na zaalokowanej pamięci
- przypisuje adres do wskaźnika

```
// replaceable allocation functions
void* operator new ( std::size_t count );
void* operator new[]( std::size_t count );
// replaceable non-throwing allocation functions
void* operator new ( std::size_t count, const std::nothrow_t& tag);
void* operator new[]( std::size_t count, const std::nothrow_t& tag);
// user-defined placement allocation functions
void* operator new ( std::size_t count, user-defined-args... );
void* operator new[]( std::size_t count, user-defined-args... );
// additional param std::align_val_t since C++17, [[nodiscard]] since C++20
// some more versions on
https://en.cppreference.com/w/cpp/memory/new/operator_new
```

Dynamicznie zaalokowana tablica

- Pamiętaj o użyciu `delete[]` do zwolnienia pamięci

```
#include <iostream>

int main() {
    int staticArray[] = {1, 2, 3, 4, 5, 6};

    constexpr auto size = 10;
    int* dynamicArray = new int[size];
    for (int i = 0; i < size; ++i) {
        *(dynamicArray + i) = i * 10;
    }

    for (int i = 0; i < size; ++i) {
        std::cout << dynamicArray[i] << '\n';
    }

    delete[] dynamicArray;
}
```

Smart pointers

Inteligentne wskaźniki

- Inteligentny wskaźnik zarządza zwykłym wskaźnikiem do pamięci zaalokowanej na stercie
  - Usuwa wskazywany obiekt we właściwym czasie
  - `operator->()` woła metody wskazywanego obiektu

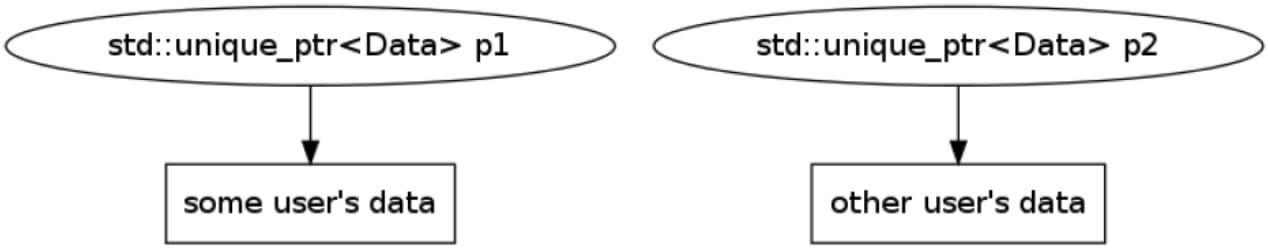
    - `operator.()` woła metody inteligentnego wskaźnika
    - inteligentny wskaźnik na klasę bazową może wskazywać na obiekt klasy pochodnej (działa polimorfizm)
- Inteligentne wskaźniki w STLu:
    - `std::unique_ptr<>`
    - `std::shared_ptr<>`
    - `std::weak_ptr<>`
    - `std::auto_ptr<>` - usunięty w C++17 (na szczęście)

`std::unique_ptr<>`

`std::unique_ptr<>`

Cechy

- jeden obiekt ma dokładnie jednego właściciela
- destruktor niszczy wskazywany obiekt i zwalnia pamięć
- kopiowanie jest niedozwolone
- możliwe jest tylko przenoszenie
- można przekazać własny `deleter` (funkcję do zawołania w destruktorze)



`std::unique_ptr<>` użycie

- stare podejście vs nowe podejście

```
#include <iostream> // old-style approach


struct Msg {
    int getValue() { return 42; }
};

Msg* createMsg() {
    return new Msg{};
}

int main() {
    auto msg = createMsg();
```

```
    std::cout << msg->getValue();
    delete msg;
}
```

```
#include <memory> // modern approach
#include <iostream>

struct Msg {
    int getValue() { return 42; }
};

std::unique_ptr<Msg> createMsg() {
    return std::make_unique<Msg>();
}

int main() {
    // unique ownership
    auto msg = createMsg();

    std::cout << msg->getValue();
}
```

`std::unique_ptr<>` użycie

- Kopiowanie niedozwolone
- Przenoszenie dozwolone

```
std::unique_ptr<MyData> source(void);
void sink(std::unique_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    // sink(ptr);       // compilation error
    sink(std::move(ptr));
    auto p1 = source();
    // auto p2 = p1;    // compilation error
    auto p2 = std::move(p1);
    // p1 = p2;         // compilation error
    p1 = std::move(p2);
}
```

```
std::unique_ptr<MyData> source(void);
void sink(std::unique_ptr<MyData> ptr);
```

```
void collections() {
    std::vector<std::unique_ptr<MyData>> v;
    v.push_back(source());

    auto tmp = source();
    // v.push_back(tmp); // compilation error
    v.push_back(std::move(tmp));

    // sink(v[0]);       // compilation error
    sink(std::move(v[0]));

}
```

`std::unique_ptr<>` kooperacja ze zwykłymi wskaźnikami

```
#include <memory>

void legacyInterface(int*) {}
void deleteResource(int* p) { delete p; }
void referenceInterface(int&) {}

int main() {
    auto ptr = std::make_unique<int>(5);
    legacyInterface(ptr.get());
    deleteResource(ptr.release());
    ptr.reset(new int{10});
    referenceInterface(*ptr);
    ptr.reset(); // ptr is a nullptr
    return 0;
}
```

- `T* get()` – zwraca zwykły wskaźnik bez zwalniania własności
- `T* release()` – zwraca zwykły wskaźnik i zwalnia własność
- `void reset(T*)` – podmienia wskazywany obiekt
- `T& operator*()` – wyłuskuje wskazywany obiekt

`std::make_unique()`

```
#include <memory>

struct Msg {
    Msg(int i) : value(i) {}
    int value;
};
```

```
int main() {
    auto ptr1 = std::unique_ptr<Msg>(new Msg{5});
    auto ptr2 = std::make_unique<Msg>(5);   // equivalent to above
    return 0;
}
```

`std::make_unique()` to funkcja, która tworzy `unique_ptr`

- dodana w C++14 dla symetrii (w C++11 było `make_shared`)
- dzięki niej nie trzeba stosować gołego `new`

`std::unique_ptr<T[]>`

```
struct MyData {};

void processPointer(MyData* md) {}
void processElement(MyData md) {}

using Array = std::unique_ptr<MyData[]>;

void use(void)
{
    Array tab{new MyData[42]};
    processPointer(tab.get());
    processElement(tab[13]);
}
```

- Podczas niszczenia:
  - `std::unique_ptr<T>` woła `delete`
  - `std::unique_ptr<T[]>` woła `delete[]`
- `std::unique_ptr<T[]>` ma dodatkowy `operator[]` do dostępu do konkretnego elementu tablicy
- Zazwyczaj `std::vector<T>` jest lepszym wyborem

Zadanie: `ResourceD`

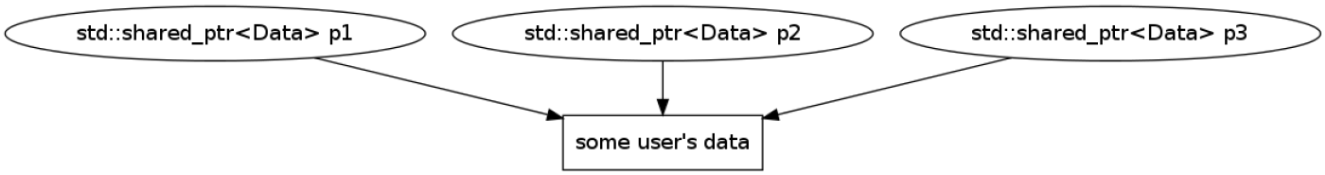
1. Skompiluj i odpal program
2. Sprawdź wycieki pamięci pod valgrindem
3. Napraw wycieki używając `delete`
4. Zamień zwykłe wskaźniki na `std::unique_ptr`
5. Użyj `std::make_unique()`

`std::shared_ptr<>`

`std::shared_ptr<>`

Cechy

- jeden obiekt ma wielu właścicieli
- ostatni sprząta
- możliwe kopiowanie
- możliwe przenoszenie
- można dostarczyć własny deleter
- można dostarczyć własny alokator



`std::shared_ptr<>` użycie

- Kopiowanie i przenoszenie jest dozwolone

```
std::shared_ptr<MyData> source();
void sink(std::shared_ptr<MyData> ptr);

void simpleUsage() {
    source();
    sink(source());
    auto ptr = source();
    sink(ptr);
    sink(std::move(ptr));
    auto p1 = source();
    auto p2 = p1;
    p2 = std::move(p1);
    p1 = p2;
    p1 = std::move(p2);
}
```

```
std::shared_ptr<MyData> source();
void sink(std::shared_ptr<MyData> ptr);

void collections() {
    std::vector<std::shared_ptr<MyData>> v;

    v.push_back(source());

    auto tmp = source();
    v.push_back(tmp);
```

```
    v.push_back(std::move(tmp));

    sink(v[0]);
    sink(std::move(v[0]));
}
```

`std::shared_ptr<>` użycie

```
#include <memory>
#include <map>
#include <string>

class Gadget {};
std::map<std::string, std::shared_ptr<Gadget>> gadgets;
// above wouldn't compile with C++03. Why?

void foo() {
    std::shared_ptr<Gadget> p1{new Gadget()};   // reference counter = 1
    {
        auto p2 = p1;                           // copy (reference counter
== 2)
        gadgets.insert(make_pair("mp3", p2));   // copy (reference counter
== 3)
        p2->use();
    }                                           // destruction of p2,
reference counter = 2
}                                               // destruction of p1,
reference counter = 1

int main() {
    foo();
    gadgets.clear();                            // reference counter = 0 -
gadget is removed
}
```

`std::shared_ptr<>` - cykliczne zależności

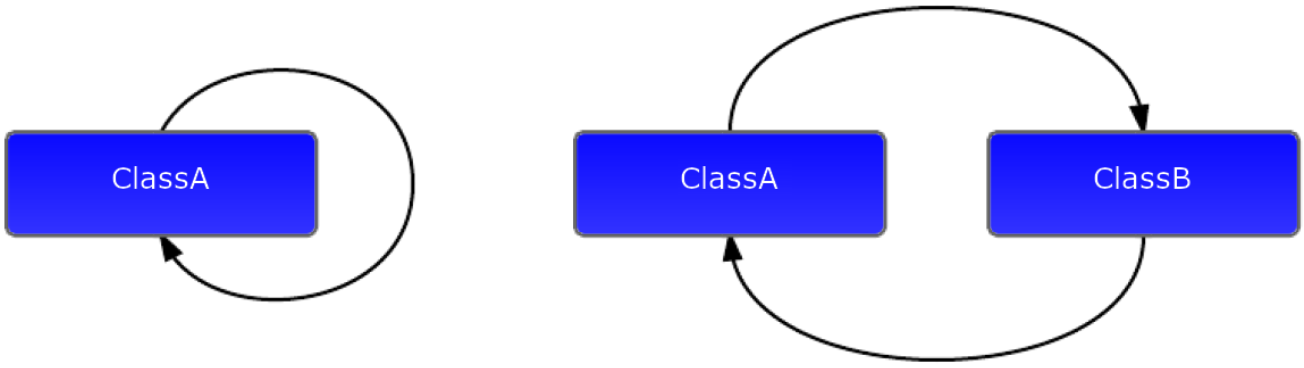
- Co my tu mamy?

```
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main () {
```

```
    auto root = std::shared_ptr<Node>(new Node);
    auto leaf = std::shared_ptr<Node>(new Node);

    root->child = leaf;
    leaf->parent = root;
}
```

Wyciek pamięci!

Cykliczne zależności



- Obiekt klasy A posiada wskaźnik/referencję ustawione na siebie
- Obiekt klasy A posiada wskaźnik/referencję na obiekt klasy B, który z kolei wskazuje na obiekt klasy A

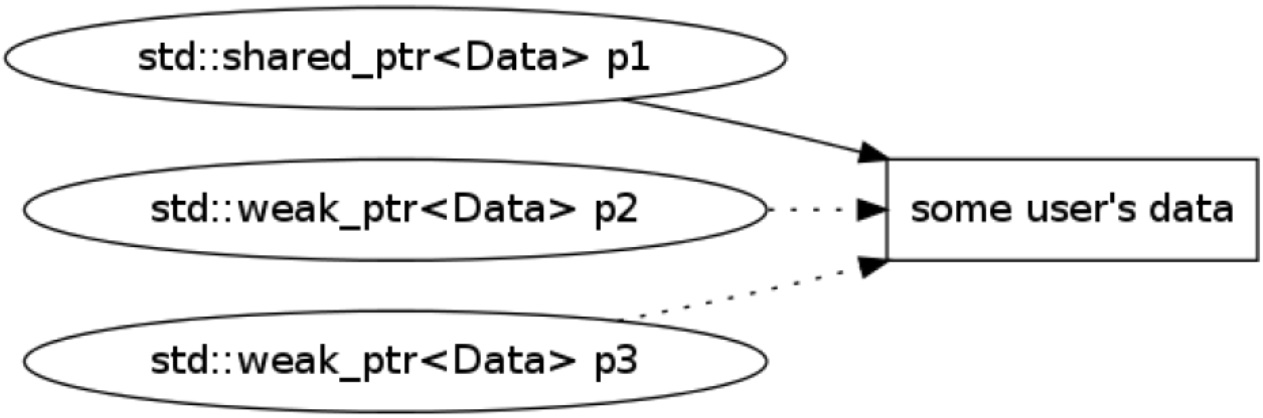
Cykliczne zależności

- Obiekt klasy A posiada wskaźnik/referencję ustawione na siebie
- Obiekt klasy A posiada wskaźnik/referencję na obiekt klasy B, który z kolei wskazuje na obiekt klasy A
- Jak to naprawić?

`std::weak_ptr<>` spieszy z pomocą!

Cechy

- nie jest właścicielem obiektu
- jedynie obserwuje obiekt
- przed dostępem do obiektu musi zostać skonwertowany na `std::shared_ptr<>`
- może zostać utworzony tylko z użyciem `std::shared_ptr<>`



`std::weak_ptr<>` użycie

```
#include <memory>
#include <iostream>

struct Msg { int value; };
```

```
void checkMe(const std::weak_ptr<Msg> & wp) {
    std::shared_ptr<Msg> p = wp.lock();
    if (p)
        std::cout << p->value << '\n';
    else
        std::cout << "Expired\n";
}

int main() {
    auto sp = std::shared_ptr<Msg>{new Msg{10}};
    auto wp = std::weak_ptr<Msg>{sp};
    checkMe(wp);
    sp.reset();
    checkMe(wp);
}
```

```
> ./a.out
10
Expired
```

`std::shared_ptr<>` - cykliczne zależności

- Jak rozwiązać ten problem?

```
#include <memory>

struct Node {
    std::shared_ptr<Node> child;
    std::shared_ptr<Node> parent;
};

int main () {
    auto root = std::shared_ptr<Node>(new Node);
    auto leaf = std::shared_ptr<Node>(new Node);

    root->child = leaf;
    leaf->parent = root;
}
```

Przerywanie cykli - rozwiązanie

- Użyć `std::weak_ptr<Node>` w jednym kierunku

```
#include <memory>
struct Node {
```

```
    std::shared_ptr<Node> child;
    std::weak_ptr<Node> parent;
};

int main () {
    auto root = std::shared_ptr<Node>(new Node);
    auto leaf = std::shared_ptr<Node>(new Node);

    root->child = leaf;
    leaf->parent = root;
}
```

```
==148== All heap blocks were freed -- no leaks are possible
```

`std::auto_ptr<>` - wyrzucony na śmietnik

- Jeśli znasz - zapomnij o nim
- Został wprowadzony w C++98
- Został poprawiony w C++03
- Ale ciągle był łatwy do niepoprawnego użycia
- Oznaczony jako przestarzały w C++11
- Usunięty w C++17
- Zamiast niego używamy `std::unique_ptr<>`

Inteligentne wskaźniki - podsumowanie

- `#include <memory>`
- `std::unique_ptr<>` dla wyłącznej własności
- `std::shared_ptr<>` dla współdzielonej własności
- `std::weak_ptr<>` do obserwowania i przerywania cykli

Zadanie: `ResourceFactory`

1. Skompiluj i uruchom program ResourceFactory
2. Czy zauważasz problematyczne miejsca? Oznacz je komentarzami w kodzie.
3. Jak usuwać elementy z kolekcji (`vector<Resource*>` resources)?
4. Uruchom program pod valgrindem
5. Napraw problemy

Najlepsze praktyki

Najlepsze praktyki

- Zasada 0, Zasada 5
- Unikaj jawnego `new`
- Używaj `std::make_shared()` / `std::make_unique()`
- Przekazuj `std::shared_ptr<>` przez `const&`
- Używaj referencji zamiast wskaźników

Rule of 0, Rule of 5

Rule of 5

- Jeśli musisz zaimplementować jedną z poniższych funkcji:
  - destructor
  - copy constructor
  - copy assignment operator
  - move constructor
  - move assignment operator
- To najprawdopodobniej potrzebujesz ich wszystkich, bo ręcznie zarządzasz zasobami.

Rule of 0

- Używaj wrapperów RAII, aby nie implementować żadnej z powyższych funkcji

---

Unikaj jawnego `new`

- Inteligentne wskaźniki eliminują potrzebę używania jawnie `delete`
- Aby zachować symetrię, nie używajmy też `new`
- Alokuj zasoby używając:
  - `std::make_unique()`
  - `std::make_shared()`
- Jedną z metryk jakości kodu może być liczba jawnych `new` i `delete`

Use `std::make_shared()` / `std::make_unique()`

- Jaki problem mógł tu wystąpić (przed C++17)?

```
struct MyData { int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    sink(Ptr{new MyData{41}}, Ptr{new MyData{42}});
}
```

- Wskazówka: ta wersja jest poprawna

```
struct MyData { int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);

void use(void) {
    Ptr oldData{new MyData{41}};
    Ptr newData{new MyData{42}};
    sink(std::move(oldData), std::move(newData));
}
```

Rozkładamy alokację

`auto p = new MyData(10);` oznacza:

- zaalokowanie `sizeof(MyData)` bajtów na stercie
- wywołanie konstruktora `MyData`
- przypisanie adresu zaalokowanej pamięci do wskaźnika `p`

The order of evaluation of operands of almost all C++ operators (including the order of evaluation of function arguments in a function-call expression and the order of evaluation of the subexpressions within any expression) is **unspecified**.

Nieokreślona kolejność ewaluacji

- Co gdy mamy dwie takie operacje?

first operation (A)	**second operation (B)**
(1) allocate `sizeof(MyData)` bytes	(1) allocate `sizeof(MyData)` bytes
(2) run `MyData` constructor	(2) run `MyData` constructor
(3) assign address of allocated memory to `p`	(3) assign address of allocated memory to `p`

- Nieokreślona kolejność ewaluacji oznaczała, że te operacje mogły być wykonane np. w takiej kolejności:
  - A1, A2, B1, B2, A3, B3
- A co, gdy B2 rzuci wyjątkiem?

`std::make_shared()` / `std::make_unique()`

- `std::make_shared()` / `std::make_unique()` rozwiązuje ten problem

```
struct MyData{ int value; };
using Ptr = std::shared_ptr<MyData>;
void sink(Ptr oldData, Ptr newData);
```

```
void use() {
    sink(std::make_shared<MyData>(41), std::make_shared<MyData>(42));
}
```

- Naprawia problem
- Nie powtarzamy konstruowanego typu w kodzie
- Nie używamy jawnie `new`
- Optymalizujemy zużycie pamięci i ułożenie danych (tylko w przypadku `std::make_shared()`)

Przekazywanie `std::shared_ptr<>`

```
void foo(std::shared_ptr<MyData> p);

void bar(std::shared_ptr<MyData> p) {
    foo(p);
}
```

- kopiowanie wymaga inkrementacji / dekrementacji liczników
- musi być bezpieczne wielowątkowo - `std::atomic` / `std::lock` nie są darmowe
- zawoła destruktory

Czy można lepiej?

Przekazywanie `std::shared_ptr<>`

```
void foo(const std::shared_ptr<MyData> & p);

void bar(const std::shared_ptr<MyData> & p) {
    foo(p);
}
```

- tak szybkie jak przekazywanie zwykłego wskaźnika
- bez dodatkowych operacji
- może być niebezpieczne w aplikacjach wielowątkowych (jak każde przekazywanie przez `&`)

Używaj referencji zamiast wskaźników

- Jaka jest różnica pomiędzy wskaźnikiem i referencją?
  - referencja nie może być pusta
  - nie można zmienić przypisania referencji, aby wskazywała na inny obiekt
- Priorytety użycia (jeśli możliwe):
  - `(const) T&`

- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `T*`

Zadanie: `List`

Zajrzyj do pliku `List.cpp`, w którym jest zaimplementowana prosta (i niepoprawna) lista (single-linked list).

- `void add(Node* node)` dodaje nowy element `Node` na końcu listy
- `Node* get(const int value)` iteruje po liście i zwraca pierwszy element, którego wartość wynosi `value` lub `nullptr`

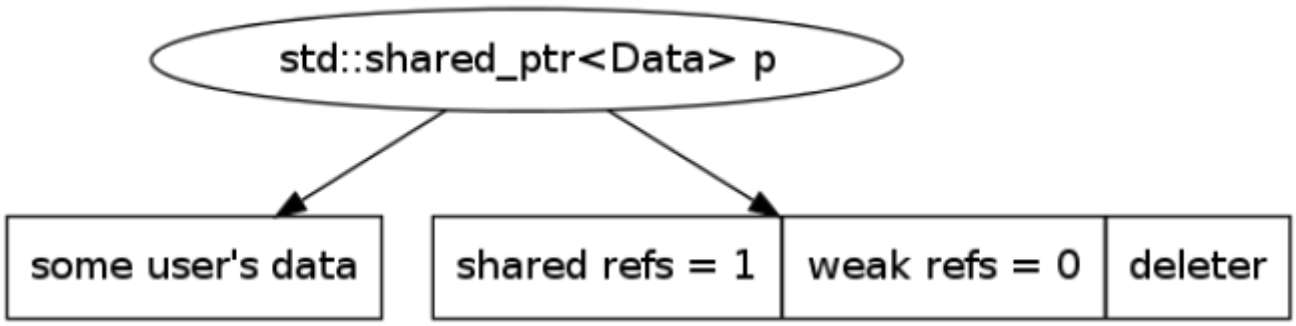
1. Skompiluj i uruchom aplikację List
2. Napraw wycieki pamięci bez stosowania smart pointerów
3. Napraw wycieki pamięci stosując smart pointery. Jaki ich rodzaj zastosujesz i dlaczego?
4. (Opcjonalnie) Co się stanie jeśli na listę dodamy ten sam element 2 razy? Napraw problem.
5. (Opcjonalnie) Utwórz wyjątek `EmptyListError` (dziedziczący po `std::runtime_error`). Dodaj jego rzucanie i łapanie we właściwych miejscach.

Szczegóły implementacyjne

`std::unique_ptr<>`

- To tylko "wrapper"
- Trzyma zwykły wskaźnik
- Konstruktor kopiuje wskaźnik (płytka kopia)
- Destruktor woła odpowiedni `operator delete`
- Brak operacji kopiowania
- Przeniesienie oznacza:
  - Skopiowanie wskaźników z obiektu źródłowego do docelowego
  - Ustawienie wskaźników w obiekcie źródłowym na `nullptr`
- Wszystkie metody są `inline`

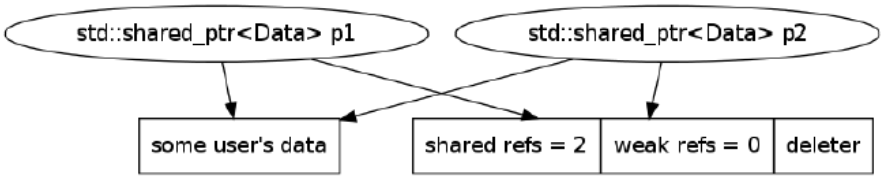
`std::shared_ptr<>`



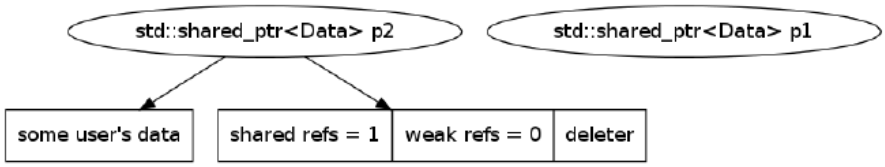
- Trzyma zwykły wskaźnik do obiektu
- Trzyma też wskaźnik na współdzielony blok kontrolny zawierający:
  - licznik shared_ptrów
  - licznik weak_ptrów
  - deleter
- Destruktor:
  - dekrementuje `shared-refs`
  - usuwa obiekt gdy `shared-refs == 0`
  - usuwa blok kontrolny, gdy `shared-refs == 0` i `weak-refs == 0`
- Wszystkie metody są `inline`

`std::shared_ptr<>`

- Kopiowanie oznacza:
  - Skopiowanie wskaźników
  - Inkrementację `shared-refs`

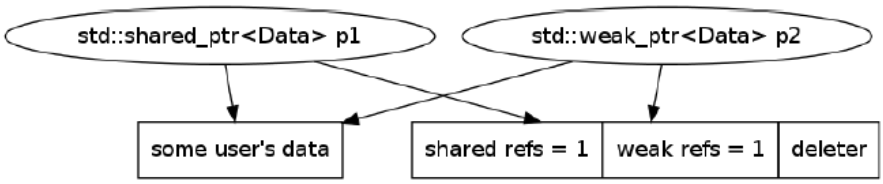


- Przeniesienie oznacza:
  - Skopiowanie wskaźników
  - Ustawienie wskaźników w obiekcie źródłowym na `nullptr`



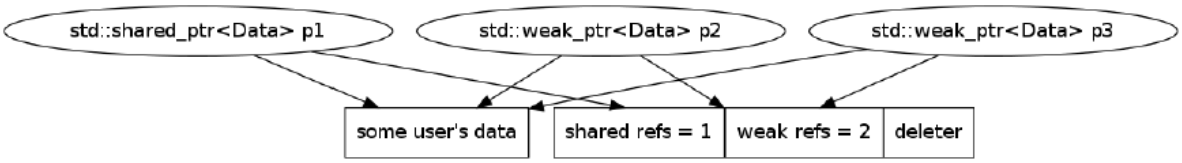
`std::weak_ptr<>`

- Trzyma zwykły wskaźnik do obiektu
- Trzyma też wskaźnik na współdzielony blok kontrolny zawierający:
  - licznik shared_ptrów
  - licznik weak_ptrów
  - deleter
- Destruktor:
  - dekrementuje `weak-refs`
  - usuwa blok kontrolny, gdy `shared-refs == 0` i `weak-refs == 0`

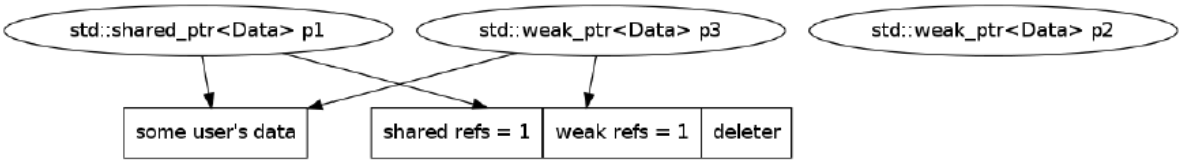


`std::weak_ptr<>`

- Kopiowanie oznacza:
  - Skopiowanie wskaźników
  - Inkrementację `weak-refs`

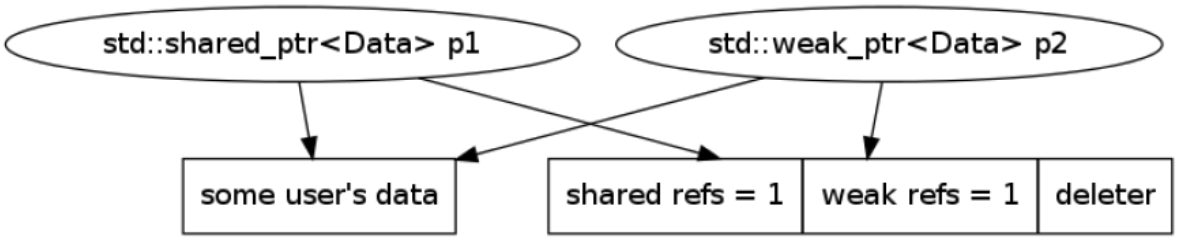


- Przeniesienie oznacza:
  - Skopiowanie wskaźników
  - Ustawienie wskaźników w obiekcie źródłowym na `nullptr`

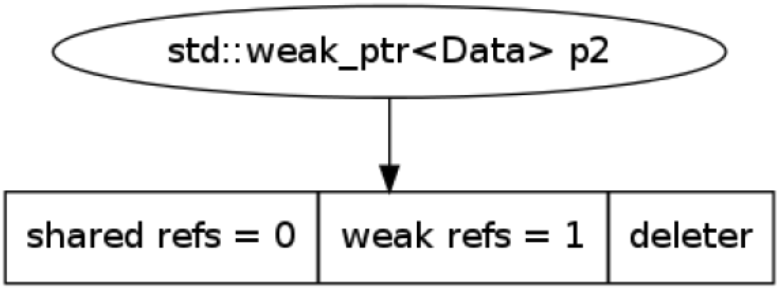


`std::weak_ptr<>` + `std::shared_ptr<>`

- Gdy mamy `shared_ptr` i `weak_ptr`

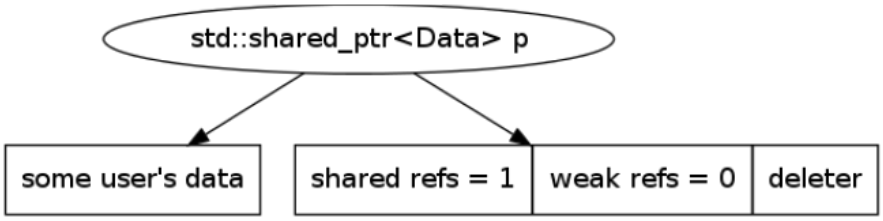


- Po usunięciu `shared_ptr`

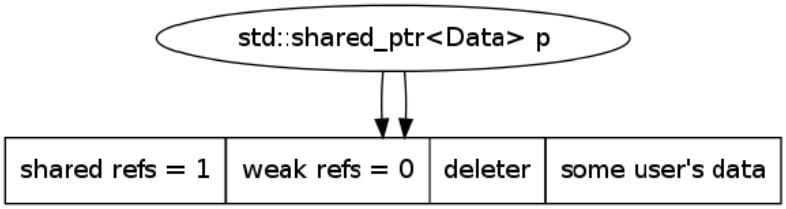


Tworzenie `std::shared_ptr<>`

- `std::shared_ptr<Data> p{new Data};`



- `auto p = std::make_shared<Data>();`
  - Mniejsze zużycie pamięci (w większości przypadków)
  - Tylko jedna alokacja
  - Cache-friendly



Wydajność

Raw pointer

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<Data *> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = new Data;
        v.push_back(std::move(p));
    }
    for (auto p: v)
        delete p;
}
```

Unique pointer

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::unique_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::unique_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

Shared pointer

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        v.push_back(std::move(p));
    }
}
```

Shared pointer – `make_shared`

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
```

```
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> v;
    v.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        auto p = std::make_shared<Data>();
        v.push_back(std::move(p));
    }
}
```

Weak pointer

```
#include <memory>
#include <vector>

struct Data {
    char tab_[42];
};

int main(void) {
    constexpr unsigned size = 10u * 1000u * 1000u;
    std::vector<std::shared_ptr<Data>> vs;
    std::vector<std::weak_ptr<Data>> vw;
    vs.reserve(size);
    vw.reserve(size);
    for (unsigned i = 0; i < size; ++i) {
        std::shared_ptr<Data> p{new Data};
        std::weak_ptr<Data> w{p};
        vs.push_back(std::move(p));
        vw.push_back(std::move(w));
    }
}
```

Pomiary

- gcc-4.8.2
- kompilacja z `-std=c++11 -O3 -DNDEBUG`
- pomiary z wykorzystaniem:
    - time (real)
    - htop (mem)
    - valgrind (liczba alokacji)

Wyniki

test name	time [s]	allocations	memory [MB]
raw pointer	0.54	10 000 001	686

test name	time [s]	allocations	memory [MB]
unique pointer	0.56	10 000 001	686
shared pointer	1.00	20 000 001	1072
make shared	0.76	10 000 001	914
weak pointer	1.28	20 000 002	1222

Podsumowanie

- RAII
  - pozyskujemy zasób w konstruktorze
  - zwalniamy zasów w destruktorze
- Rule of 5, Rule of 0
- Smart pointery:
  - `std::unique_ptr` – główny wybór, brak dodatkowych kosztów, można go skonwertować na `std::shared_ptr`
  - `std::shared_ptr` – wprowadza dodatkowy narzut pamięciowy i czasu wykonania
  - `std::weak_ptr` – do łamania cykli, konwertuje się z/na `std::shared_ptr`
- Twórz smart pointery używając `std::make_shared()` i `std::make_unique()`
- Zwykłe wskaźniki powinny zawsze oznaczać tylko dostęp do zasobu (bez własności)
- Kiedy to możliwe używaj referencji zamiast jakichkolwiek wskaźników

Homework

Pre-work

Przeczytaj artykuł Semantyka przenoszenia

Przyda się do post-worku 😃

Post-work

(30 XP) Zaimplementuj swój własny `unique_ptr` (trochę uproszczony).

`unique_ptr` to klasa RAII, która:

- Jest klasą szablonową
- Trzyma wskaźnik do zarządzanego obiektu
- Konstruktor kopiuje wskaźnik
- Destruktor zwalnia pamięć
- Kopiowanie jest niedozwolone
- Przenoszenie jest dozwolone i oznacza:
  - Skopiowanie wskaźnika z obiektu źródłowego
  - Ustawienie wskaźnika w obiekcie źródłowym na `nullptr`

- Wymagane metody: `operator*()`, `operator->()`, `get()`, `release()`, `reset()`
- Nie zapomnij o testach (pokrycie >90%)

+3 XP za dostarczenie do 13.10.2020 włącznie