

C++ Memory Management

Bogumił Chojnowski
Michał Orynicz

Wprowadzenie

O czym jest prezentacja?

1. Skąd się bierze i jak wygląda pamięć w programie?
2. Jak zarządzać pamięcią w obliczu sytuacji wyjątkowych?
3. Jakie są techniki automatycznego zapobiegania wyciekom pamięci w C++?
4. Jak określić właściciela - byt odpowiedzialny za zwolnienie zasobu?
5. Jakie są techniki bezpiecznego współdzielenia zasobu między wątkami?
6. Jak zlecić wygenerowanie kodu zarządzającego pamięcią kompilatorowi języka C++?
7. Jak biblioteka standardowa C++ pomaga zarządzać pamięcią w *sprytny* sposób?
8. Czy zarządzanie pamięcią jest kosztowne?

Zarządzanie pamięcią

Zestaw technik kontroli nad zasobami wykorzystywanymi przez program w trakcie jego działania.

Alokacja

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Co to jest alokacja zasobów?

Alokacja to przypisywanie zasobów do możliwości ich użycia.

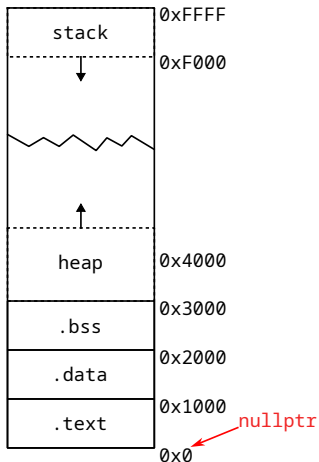
Alokacja zasobów

Skąd się bierze pamięć w programie?

Z punktu widzenia systemu

W trakcie swojego działania program może zażądać od systemu operacyjnego większej ilości pamięci (**alokacja**) lub zwolnić niepotrzebny już obszar (**dealokacja**). Zadaniem systemu jest pamięć przydzielić, o ile dysponuje jej *ciągłym* obszarem w wymaganym rozmiarze.

Przestrzeń adresowa



`.text` kod wykonywalny programu

`.data` stałe i napisy

`.bss` zmienne globalne

`heap` sarta

`stack` stos (rośnie w dół)

`nullptr` - on też tu jest!

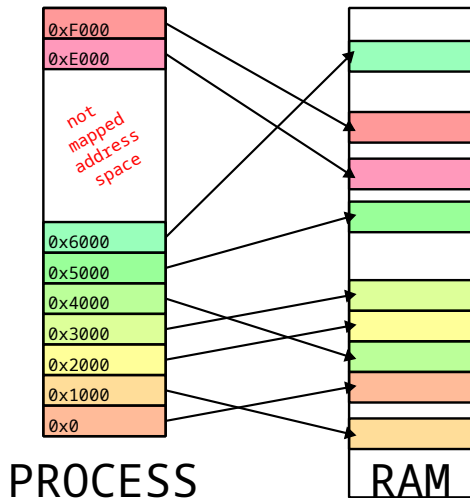
Przestrzeń adresowa

Przestrzeń adresowa \neq zaalokowana pamięć

Proces ma do dyspozycji *ciągłą* przestrzeń adresową.

Nie oznacza to, że system rezerwuje procesom po 4 gigabajty (lub więcej) RAM.

Przestrzeń adresowa

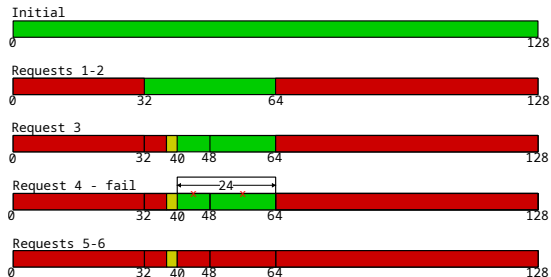


Wirtualizacja / mapowanie pamięci

Program otrzymuje dostęp do fizycznej pamięci w miarę zapotrzebowania i w kawałkach (stronach). Wypełniana jest nimi przestrzeń adresowa.

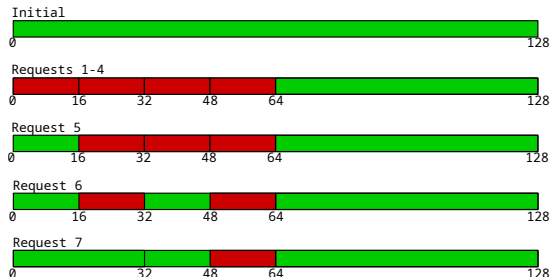
Algorytm bloków bliźniaczych

```
1  #include "Heap.hpp"
2
3  void demo_fragmentation()
4  {
5      Heap h(128);
6
7      h.allocate(32);
8      h.allocate(64);
9      h.allocate(5); // 3 bytes wasted
10
11     h.allocate(20); // fail!
12
13     h.allocate(16); // OK
14     h.allocate(8);  // OK
15 }
```



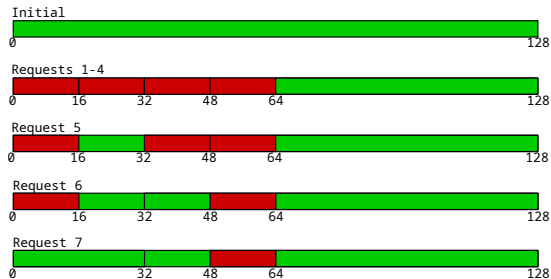
Algorytm bloków bliźniaczych

```
1  #include "Heap.hpp"
2
3  void demo_deallocation_1()
4  {
5      Heap h(128);
6
7      h.allocate(16);
8      h.allocate(16);
9      h.allocate(16);
10     h.allocate(16);
11
12     h.deallocate(0);
13     h.deallocate(32);
14     h.deallocate(16); // coalescing
15 }
```



Algorytm bloków bliźniaczych

```
1  #include "Heap.hpp"
2
3  void demo_deallocation_2()
4  {
5      Heap h(128);
6
7      h.allocate(16);
8      h.allocate(16);
9      h.allocate(16);
10     h.allocate(16);
11
12     h.deallocate(16);
13     h.deallocate(32);
14     h.deallocate(0); // coalescing
15 }
```



Jak dobrze nam poszło?

- Algorytm jest szybki $\mathcal{O}(\log M_{pages})$;
- umiarkowanie dokładny (fragmentacja *wewnętrzna* poniżej 50%);
- redukcja fragmentacji *zewnętrznej* przez scalanie bloków bliźniaczych.

Algorytm bloków bliźniaczych

Podsystem pamięci jądra systemu operacyjnego

Omówiony algorytm alokacji bloków bliźniaczych używany jest w Linux Kernel do zarządzania przydzielaną procesom pamięcią.

Szczegóły w załączniku **Interfejs programistyczny podsystemu pamięci**

Sterna, inaczej kopiec

Informacje o zajętości bloków można przedstawiać w formie kopca — struktury danych reprezentującej drzewo binarne.

Przykład kopca w załączniku **Struktura danych: kopiec**

Zadanie z gwiazdką

Ulepszyć implementację algorytmu, aby informacja o zajętości bloków trzymana była w tablicy o ustalonym rozmiarze.

`memory-management/research/buddy_allocation`

Fragmentacja pamięci

Fragmentacja wewnętrzna

Wynika z wyrównywania przydziału do rozmiaru bloku.
Przydzielony blok jest większy niż dane w nim umieszczone, naddatek się marnuje.

Fragmentacja zewnętrzna

Każda alokacja i dealokacja stwarza ryzyko pozostawienia przestrzeni niepasującej do zapotrzebowania.

Z punktu widzenia języka...

Język programowania C++ definiuje dwa sposoby do przeprowadzania alokacji danych, różniące się strukturą i zastosowaniami: **alokacja dynamiczna** i **alokacja automatyczna**.

Rodzaje alokacji

- alokacja automatyczna (na stosie)

```
1  int global_variable = 11;
2
3  int main(int ac, char* av[])
4  {
5      static int static_variable = 22;
6      int automatic_variable = 33;
7      int* heap_variable = new int(44);
8      delete heap_variable;
9
10     return 0;
11 }
```

Rodzaje alokacji

- alokacja automatyczna (na stosie)
- alokacja dynamiczna (na sterpie)

```
1  int global_variable = 11;
2
3  int main(int ac, char* av[])
4  {
5      static int static_variable = 22;
6      int automatic_variable = 33;
7      int* heap_variable = new int(44);
8      delete heap_variable;
9
10     return 0;
11 }
```

Sterna

W języku C++ pamięcią alokowaną dynamicznie zarządza *programista*.
Obiekty są tworzone i likwidowane na bieżąco, w dowolnym miejscu programu.
Obszarem przestrzeni adresowej, na którym przeprowadzamy alokację dynamiczną, jest **sterna** (ang. *heap*).

Operatory new i delete

Do tworzenia i usuwania obiektów na stercie służą operatory:

- `new` alokuje pamięć i uruchamia konstruktor;
- `delete` wykonuje destruktor i zwalnia pamięć.

```
1  #include <paro/Object.hpp>
2
3  int main(int ac, char* av[])
4  {
5      Object* p = new Object(3, 14, 92);
6      delete p;
7
8      return 0;
9  }
```

Operatory `new[]` i `delete[]`

Aby utworzyć lub usunąć tablicę można wykorzystać wersje operatorów dla tablic:

- `new[]` alokuje pamięć i wykonuje konstruktory,
- `delete[]` wykonuje destruktory i zwalnia pamięć.

```
1  int main(int ac, char* av[])
2  {
3      int* arr = new int[40];
4      arr[34] = 9;
5      arr[37] = 21;
6      delete[] arr;
7
8      return 0;
9  }
```


Alokacja dynamiczna

Konstrukcja tablicy typu z domyślnym konstruktorem

```
1  #include <string>
2
3  constexpr int arr_size = 40;
4
5  int main(int ac, char* av[])
6  {
7      std::string* arr = new std::string[arr_size];
8      delete[] arr;
9
10     return 0;
11 }
```

Alokacja dynamiczna

Konstrukcja tablicy typu bez domyślnego konstruktora

```
1  #include <paro/Object.hpp>
2
3  int main(int ac, char* av[])
4  {
5      constexpr auto arr_size = 10u;
6      Object** arr = new Object*[arr_size];
7      for (int i = 0; i < arr_size; ++i)
8          arr[i] = new Object(3*i, 2*i+14, i+92);
9      for (int i = 0; i < arr_size; ++i)
10         delete arr[i];
11     delete[] arr;
12     return 0;
13 }
```

Ale... całe strony dla pojedynczych obiektów?

Na szczęście nie.

Żądanie o nową stronę składane jest, gdy operator `new` nie znajdzie odpowiedniego miejsca w żadnej z dotychczas przydzielonych stron.

Operator `delete`, zwalniając ostatni blok pamięci na stronie, może zwrócić ją do systemu.

Alokacja dynamiczna - podsumowanie

- Pamięć zaalokowana dynamicznie musi zostać zwolniona,
- obowiązek zwalniania pamięci spoczywa na programiście,
- podczas alokowania na sterce może wystąpić fragmentacja,
- algorytmy alokujące powinny być szybkie i dokładne...
- ...albo dostosowane do specyfiki programu.

Można przeciążać operatory, aby zaimplementować alternatywne algorytmy alokacji.

Stos

W języku C++ czasem życia zmiennych zarządza *kompilator*.

Obiekty są tworzone i likwidowane w sposób automatyczny, związany ze strukturą kodu źródłowego.

Obszarem, w którym znajdują się takie obiekty, jest **stos** (ang. *Stack*).

```
1  int foo(int a, int b)
2  {
3      int x;
4      char y[8];
5
6      return 7;
7  }
8
9  int main(int ac, char* av[])
10 {
11     int f = foo(11, 13);
12
13     return 0;
14 }
```

Co znajdziemy na stosie?

- Argumenty wywołania funkcji,
- wszystkie lokalne zmienne,
- ...a co z typem zwracany?
 - Miejsce na wartość zwracaną jest zarezerwowane na stosie jeszcze przed wywołaniem.

Alokacja automatyczna

```
1  #include <paro/Object.hpp>
2
3  int proc(int a, int b, int c)
4  {
5      Object o(a, b, c);
6
7      return o.method();
8  }
9
10 int main(int ac, char* av[])
11 {
12     int p = proc(11, 13, 16);
13
14     return 0;
15 }
```

- obiekt jest tworzony konstruktorem o trzech argumentach;

Alokacja automatyczna

```
1  #include <paro/Object.hpp>
2
3  int proc(int a, int b, int c)
4  {
5      Object o(a, b, c);
6
7      return o.method();
8  }
9
10 int main(int ac, char* av[])
11 {
12     int p = proc(11, 13, 16);
13
14     return 0;
15 }
```

- obiekt jest tworzony konstruktorem o trzech argumentach;
- kompilator zadba o to, by destruktor obiektu został wykonany przy opuszczaniu funkcji;
- obiekt jest automatyczny.

A co ze zmiennymi globalnymi?

Zmienne globalne i statyczne funkcje nie są przechowywane na stosie ani na sterku. Kompilator umieści je w sekcjach `.data` i `.bss`.

Stos (struktura danych)

Stos programowy omawiany tutaj wziął swoją nazwę od liniowej struktury danych. Schemat działania stosu przedstawiono w załączniku **Struktura danych: stos**

Zakres zmiennych

Zakres zmiennych

W języku C++ zmienne mają swój zakres: obszar w kodzie programu, w którym można z nich korzystać.

Odwołanie się do zmiennej spoza zakresu powoduje wykonanie *niezdefiniowanego zachowania*.

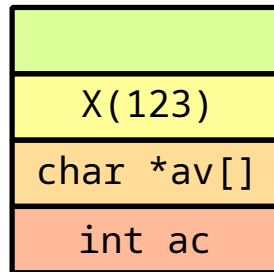
Zakres zmiennych

```
1  #include <print>
2
3  struct X
4  {
5      X(int x) : x(x) { std::println("X({})_created", x); }
6      ~X() { std::println("X({})_is_dead", x); }
7      int x;
8  };
9
10 int main(int ac, char* av[])
11 {
12     X a(123);
13     {
14         X b(42);
15     }
16     X c(89);
17
18     return 0;
19 }
```

```
X(123) created
X(42) created
X(42) is dead
X(89) created
X(89) is dead
X(123) is dead
```

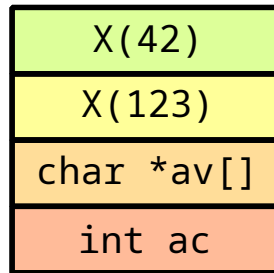
Zakres zmiennych

```
1  int main(int ac, char* av[])
2  {
3      X a(123);
4      {
5          X b(42);
6      }
7      X c(89);
8
9      return 0;
10 }
```



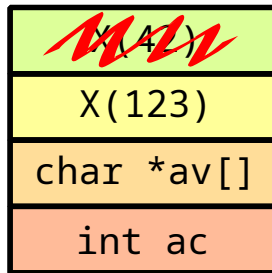
Zakres zmiennych

```
1  int main(int ac, char* av[])
2  {
3      X a(123);
4      {
5          X b(42);
6      }
7      X c(89);
8
9      return 0;
10 }
```



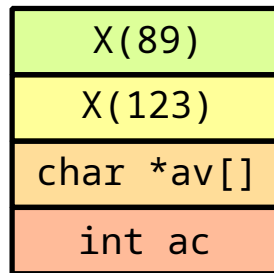
Zakres zmiennych

```
1  int main(int ac, char* av[])
2  {
3      X a(123);
4      {
5          X b(42);
6      }
7      X c(89);
8
9      return 0;
10 }
```



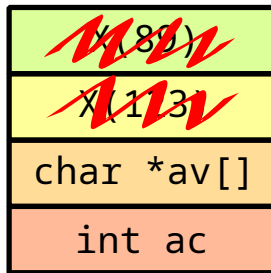
Zakres zmiennych

```
1  int main(int ac, char* av[])
2  {
3      X a(123);
4      {
5          X b(42);
6      }
7      X c(89);
8
9      return 0;
10 }
```



Zakres zmiennych

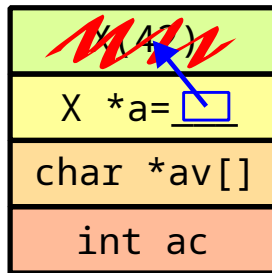
```
1  int main(int ac, char* av[])
2  {
3      X a(123);
4      {
5          X b(42);
6      }
7      X c(89);
8
9      return 0;
10 }
```



Zakres zmiennych

Co tu może pójść źle?

```
1  int main(int ac, char* av[])
2  {
3      X* a = nullptr;
4      {
5          X b(42);
6          a = &b; // save for later
7      }
8      a->x = 8; // whoops!
9
10     return 0;
11 }
```



Błędy odwołania do pamięci alokowanej na stosie

Odwołanie się do usuniętego obiektu jest proste do wykonania i łatwe do przeoczenia, a prowadzi do *niezdefiniowanego zachowania*.

Zakres

Zakres (ang. *Scope*) tworzy się przez parę klamer.
Przed wyjściem z zakresu uruchamiane są destruktory *obiektów automatycznych*.

Obiekty automatyczne

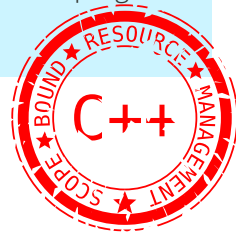
Do alokacji obiektu automatycznego na stosie wystarczy zadeklarować zmienną wewnątrz bloku z odpowiednim zestawem kwalifikatorów:

1. nie `const/constexpr`, bo kompilator ma prawo umieścić je poza stosem;
2. nie `static`, bo takie są alokowane razem z obiektami globalnymi.

Gwarancja na obiekty automatyczne

Obiekty automatyczne otrzymują silne gwarancje od kompilatora.

1. **Konstruktor** obiektu wykona się najpóźniej w chwili napotkania ich przez kod programu.
2. **Destruktor** obiektu wykona się przed opuszczeniem zakresu.



Funkcje i procedury

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Funkcja definiuje lokalny zakres

Definicja funkcji tworzy zakres dla swoich lokalnych zmiennych. Wewnątrz zakresu znajdują się wszystkie zadeklarowane zmienne *oraz* przekazane argumenty.

Argumenty funkcji

Argumenty funkcji lokalne? Ale one są z zewnątrz...

W C++ występuje *tylko* przekazywanie przez wartość.
Wskaźniki i referencje to też wartości typu adres: (64-bitowe) liczby całkowite.

Argumenty funkcji

```
1  #include <print>
2
3  void ptr(int* ptr)
4  {
5      if (ptr) {
6          std::println("{} ", *ptr);
7          delete ptr;
8      }
9      ptr = nullptr;
10 }
11
12 int main(int ac, char* av[])
13 {
14     int* p = new int(45);
15     ptr(p);
16     std::println("Is nullptr? {}.", p == nullptr);
17
18     return 0;
19 }
```

Modyfikowanie kopii nie wpływa na oryginał.

Argumenty funkcji

```
1  #include <print>
2
3  void ptr(int* ptr)
4  {
5      if (ptr) {
6          std::println("{} ", *ptr);
7          delete ptr;
8      }
9      ptr = nullptr;
10 }
11
12 int main(int ac, char* av[])
13 {
14     int* p = new int(45);
15     ptr(p);
16     std::println("Is nullptr? {}.", p == nullptr);
17
18     return 0;
19 }
```

Modyfikowanie kopii nie wpływa na oryginał.

45

Is nullptr? false.

Process finished with **exit** code 0

Argumenty funkcji

```
1 char& get_initial(S s) { return s.n; }
2
3 int main(int ac, char* av[])
4 {
5     auto& s = get_initial(S{'B'});
6     std::println("In_main_again: {}", s);
7     return 0;
8 }
```

S('A') created.
S('A') destroyed
Segmentation fault

warning: reference to local variable 's' returned [-Wreturn-local-addr]

Zwracanie referencji

Nie zwracaj referencji na obiekt lub jego część, bo może on przestać istnieć po wyjściu z zakresu.

Problem z napisami `std::<any>::c_str()`

```

1 struct logger
2 {
3     void set_file(std::filesystem::path p)
4     {
5         filename = p.filename().c_str();
6     }
7     const char * filename = nullptr;
8 };
9
10 int main(int ac, char* av[])
11 {
12     std::filesystem::path p = "/path/to/file.txt";
13     logger log;
14     log.set_file(p);
15     std::println("Filename:_{}", log.filename);
16     return 0;
17 }

```

Problem z napisami `std::any::c_str()`

```
1  struct logger
2  {
3      void set_file(std::filesystem::path p)
4      {
5          filename = p.filename().c_str();
6      }
7      const char * filename = nullptr;
8  };
9
10 int main(int ac, char* av[])
11 {
12     std::filesystem::path p = "/path/to/file.txt";
13     logger log;
14     log.set_file(p);
15     std::println("Filename: {}", log.filename);
16     return 0;
17 }
```

Filename: AaaaAAaA!!

Problem z napisami `std::<any>::c_str()`

Napisy w stylu C

`std::filesystem::path::c_str()` zwraca wskaźnik na tablicę znaków, która przestaje istnieć po wyjściu z zakresu.

Robi tak wiele obiektów w bibliotece standardowej C++.

Problem z napisami `std::any::c_str()`

```
1  struct logger
2  {
3      void set_file(std::filesystem::path p)
4      {
5          filename = p.filename().c_str();
6      }
7      std::string filename;
8  };
9
10 int main(int ac, char* av[])
11 {
12     std::filesystem::path p = "/path/to/file.txt";
13     logger log;
14     log.set_file(p);
15     std::println("Filename: {}", log.filename);
16     return 0;
17 }
```

Filename: file.txt

Napisy w stylu C

Pracując z napisami w języku C, powinniśmy jak najszybciej opakowywać je w `std::string`.

Wymusza to kopiowanie napisu, ale dane nie znikną po wyjściu z zakresu.

Dlaczego od razu nie zwracać `std::string`

Zwracanie wskaźnika na zasób jest uzasadnione wydajnością. Nie zawsze trzeba kopiować te dane.

Napisy pobrane metodami `*::c_str()` powinny być używane natychmiast (przykładowo: do zapisania pliku), a nie przechowywane. Są one w najlepszym przypadku referencją na pole (często tymczasowego) obiektu.

Tymczasowe obiekty znikają zaś przy wyjściu z funkcji...

Wyjście z funkcji

Po wykonaniu ostatniej instrukcji

```
1 void leak_exit_scope(int param)
2 {
3     Object* obj = new Object(3, 14, param);
4     store_result(obj->method());
5 } // whoops!
```

Wyjście z funkcji

Przez użycie wyrażenia `return`

```
1  int leak_exit_return(int param)
2  {
3      Object* obj = new Object(3, 14, param);
4      if (obj->is_odd())
5          return obj->method(); // whoops!
6      delete obj;
7      return 7;
8  }
```

Wyjście z funkcji

Przez rzucenie wyjątku

```

1  int leak_exit_exception(int param)
2  {
3      Object* obj = new Object(3, 14, param);
4      if (obj->is_odd())
5          throw std::runtime_error("odd_␣param"); // whoops!
6      int value = obj->method();
7      delete obj;
8      return value;
9  }

```


Blok try-catch

```
1  #include <paro/Integer.hpp>
2
3  void throwing(Integer e)
4  {
5      Integer f{'f'};
6      throw std::runtime_error("whoops!");
7      Integer g{'g'};
8  }
9
10 int main(int argc, char* argv[])
11 {
12     Integer a{'a'}, b{'b'};
13     try {
14         Integer c{'c'}, d{'d'};
15         throwing(Integer{'e'});
16     } catch (std::runtime_error const& ex) {}
17 }
```

main:

argc 0x50

argv 0x4C

a 0x48

b 0x44

c 0x40

d 0x3C

foo: e 0x38

f 0x34

g 0x30

Blok try-catch

```
1  #include <paro/Integer.hpp>
2
3  void throwing(Integer e)
4  {
5      Integer f{'f'};
6      throw std::runtime_error("whoops!");
7      Integer g{'g'};
8  }
9
10 int main(int argc, char* argv[])
11 {
12     Integer a{'a'}, b{'b'};
13     try {
14         Integer c{'c'}, d{'d'};
15         throwing(Integer{'e'});
16     } catch (std::runtime_error const& ex) {}
17 }
```

main:

argc 0x50

argv 0x4C

a 0x48

b 0x44

c 0x40

d 0x3C

foo:

e 0x38

f 0x34

g 0x30

THROW

Blok try-catch

```
1  #include <paro/Integer.hpp>
2
3  void throwing(Integer e)
4  {
5      Integer f{'f'};
6      throw std::runtime_error("whoops!");
7      Integer g{'g'};
8  }
9
10 int main(int argc, char* argv[])
11 {
12     Integer a{'a'}, b{'b'};
13     try {
14         Integer c{'c'}, d{'d'};
15         throwing(Integer{'e'});
16     } catch (std::runtime_error const& ex) {}
17 }
```

main:

argc 0x50

argv 0x4C

a 0x48

b 0x44

c 0x40

d 0x3C

foo:

e 0x38

f 0x34

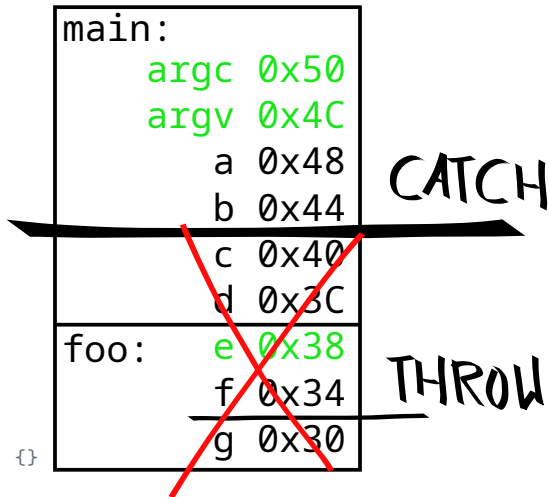
g 0x30

CATCH

THROW

Blok try-catch

```
1  #include <paro/Integer.hpp>
2
3  void throwing(Integer e)
4  {
5      Integer f{'f'};
6      throw std::runtime_error("whoops!");
7      Integer g{'g'};
8  }
9
10 int main(int argc, char* argv[])
11 {
12     Integer a{'a'}, b{'b'};
13     try {
14         Integer c{'c'}, d{'d'};
15         throwing(Integer{'e'});
16     } catch (std::runtime_error const& ex) {}
17 }
```



Strefa A: Destruktory (wszystkie!)

Podczas odwijania stosu wołane są destruktory obiektów automatycznych. Jeśli którykolwiek z nich rzuci wyjątkiem, *zachowanie jest niezdefiniowane*.

Strefa B: Konstruktor kopiujący obiektu wyjątku

Konstruktor kopiujący obiektu rzucanego może być wykorzystany do umieszczenia obiektu wyjątku w obszarze pamięci wydzielonym na to zadanie.¹

Dzieje się to już po wystąpieniu `throw`, a przed rozpoczęciem obsługi wyjątku w bloku `catch`. Rzucenie kolejnym wyjątkiem prowadzi do *niezdefiniowanego zachowania*.

Trzeba złapać je wszystkie!

Można wołać funkcje rzucające wyjątki, należy jednak zadbać, aby żaden z nich nie wyleciał na zewnątrz, bo powoduje to *niezdefiniowane zachowanie*.

Strefy bezwyjątkowe

Niezdefiniowane, czyli jakie?

Ale co to właściwie znaczy *prowadzi do niezdefiniowanego zachowania*?

Strefy bezwyjątkowe

Niezdefiniowane, czyli jakie?

Ale co to właściwie znaczy *prowadzi do niezdefiniowanego zachowania*?

Intuicyjnie: może się zdarzyć *cokolwiek*

Strefy bezwyjątkowe

Niezdefiniowane, czyli jakie?

Ale co to właściwie znaczy *prowadzi do niezdefiniowanego zachowania*?

Intuicyjnie: może się zdarzyć *cokolwiek*

z kolapsem dziennej gwiazdy włącznie, choć nie zaobserwowano tego (jeszcze) w naturze.

Właściciel zasobu

Właścicielem nazywamy obiekt odpowiedzialny za zwolnienie zasobu. Otwarte pliki trzeba zamknąć, pamięć oddać do systemu, bufor zapisu opróżnić. Stos jest gwarantem usunięcia obiektów automatycznych przed wyjściem z zakresu.

A dlaczego nie ma w C++ konstrukcji `try-catch-finally`?

A dlaczego nie ma w C++ konstrukcji `try-catch-finally`?

Nie potrzebujemy `finally`, gdy możemy wykorzystać stos oraz destruktory!

To odśmiecać potrzebuje *finally*

W niektórych językach właścicielem wszystkich obiektów jest *odśmiecać* (ang. *garbage collector*).

Jest on odpowiedzialny za zwalnianie pamięci, ale nie robi nic ponadto.

Wszelkie zasoby dodatkowe (otwarte pliki) pozostaną niesfinalizowane.

Przewidujący programista jest zobowiązany napisać kod finalizujący w **każdym bloku obsługi wyjątków**.

Posiadanie zasobu (ang. *Resource Ownership*)

Język C++ automatyzuje to zadanie przez destruktory.

Destruktory pozwalają na przypisywanie do obiektu jego kodu finalizującego. Technika ta nosi nazwę ukrytą pod akronimem **RAII**.

Resource Acquisition Is Initialization

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

Stos ma ograniczony rozmiar, łatwo go przepełnić!

Gdybyśmy tylko mogli używać każdego zasobu tak, jakby był na stosie...

Gdybyśmy tylko mogli używać każdego zasobu tak, jakby był na stosie...

Ależ możemy!

Resource Acquisition Is Initialization

Ta zagmatwana nazwa oznacza idiom zarządzania zasobami w sposób automatyczny.

Znana także pod akronimem **SBRM** - Scope-Bound Resource Management.

```
1  #include <paro/File.h>
2
3  int main(int ac, char *av[])
4  {
5      auto file = std::fopen(av[1], "r");
6      print_file(file);
7
8      return 0;
9  }
```

A czy funkcja `print_file`:

- Sprawdza, czy plik istnieje?


```
1  #include <paro/File.h>
2
3  int main(int ac, char *av[])
4  {
5      auto file = std::fopen(av[1], "r");
6      print_file(file);
7
8      return 0;
9  }
```

A czy funkcja `print_file`:

- Sprawdza, czy plik istnieje?
- Zamyka plik po odczytaniu?

```
1  #include <paro/File.h>
2
3  int main(int ac, char *av[])
4  {
5      auto file = std::fopen(av[1], "r");
6      if (not file)
7          return 1;
8      try {
9          print_file(file);
10     } catch (...) {
11         std::fclose(file); // again?
12         return 2;
13     }
14     return 0;
15 }
```

Założmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy, czy plik się otworzył;


```

1  #include <pario/File.h>
2
3  int main(int ac, char *av[])
4  {
5      auto file = std::fopen(av[1], "r");
6      if (not file)
7          return 1;
8      try {
9          print_file(file);
10     } catch (...) {
11         std::fclose(file); // again?
12         return 2;
13     }
14     return 0;
15 }

```

Założmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy, czy plik się otworzył;
- zamykamy plik po odczytaniu;
- umieszczamy wywołanie w bloku `try`;
- w bloku `catch` znowu zamykamy plik;
- ...tu przydałoby się `finally`.

```
1  #include <fstream>
2  #include <iostream>
3
4  int main(int ac, char *av[])
5  {
6      std::fstream file(av[1]);
7      std::cout << file.rdbuf();
8
9      return 0;
10 }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia;

```
1  #include <fstream>
2  #include <iostream>
3
4  int main(int ac, char *av[])
5  {
6      std::fstream file(av[1]);
7      std::cout << file.rdbuf();
8
9      return 0;
10 }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia;
- obiekt otwiera plik, sprawdza błędy;


```
1  #include <fstream>
2  #include <iostream>
3
4  int main(int ac, char *av[])
5  {
6      std::fstream file(av[1]);
7      std::cout << file.rdbuf();
8
9      return 0;
10 }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAI:

- opakowujemy plik w obiekt strumienia;
- obiekt otwiera plik, sprawdza błędy;
- obiekt udostępnia zawartość pliku;

```
1  #include <fstream>
2  #include <iostream>
3
4  int main(int ac, char *av[])
5  {
6      std::fstream file(av[1]);
7      std::cout << file.rdbuf();
8
9      return 0;
10 }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia;
- obiekt otwiera plik, sprawdza błędy;
- obiekt udostępnia zawartość pliku;
- destruktork obiektu zamyka plik;

```
1  #include <fstream>
2  #include <iostream>
3
4  int main(int ac, char *av[])
5  {
6      std::fstream file(av[1]);
7      std::cout << file.rdbuf();
8
9      return 0;
10 }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia;
- obiekt otwiera plik, sprawdza błędy;
- obiekt udostępnia zawartość pliku;
- destruktor obiektu zamyka plik;
- bonus: kod jest krótszy i bez powtórzeń.

W każdym momencie możemy sobie napisać klasę do obsługi zasobu. Zasady są następujące:

1. przejmij zasób w konstruktorze;
2. zwolnij go w destruktorze;
3. nie pozwól się skopiować.

Resource Handler

```
1  #include <paro/Resource.h>
2
3  class ResourceHandler
4  {
5  public:
6      ResourceHandler() : resource(acquireResource()) {}
7      ~ResourceHandler() { releaseResource(resource); }
8
9      // copy forbidden!
10     ResourceHandler(ResourceHandler const&) = delete;
11     ResourceHandler& operator=(ResourceHandler const&) = delete;
12
13     ResourceHandler(ResourceHandler&&) noexcept;
14     ResourceHandler& operator=(ResourceHandler&&) noexcept;
15
16 private:
17     Resource resource;
18 };
```

Bez kopiowania?

Kopia obiektu dysponowałaby wskaźnikiem na ten sam zasób i mógłby on zostać zwolniony powtórnie wraz ze skopiowanym obiektem.

A co z przenoszeniem?

Przenoszenie jest wspierane, należy jednak pamiętać o tym, by oryginalny obiekt pozbawić wskaźnika na zasób.

Resource Handler

```
1  #include <utility>
2
3  ResourceHandler::ResourceHandler(ResourceHandler&& other) noexcept
4      : resource(std::exchange(other.resource, 0x0))
5  {}
6
7  ResourceHandler& ResourceHandler::operator=(ResourceHandler&& other) noexcept
8  {
9      resource = other.resource;
10     other.resource = 0x0; // clear!
11     return *this;
12 }
```


Pisanie tego typu klas jest figurą podstawową w C++.

Do tego stopnia, że lista funkcji do napisania została skodyfikowana - więcej o tym w sekcji **Rule of Zero**.

Skoro to taka podstawowa technika, to pewnie już to ktoś zaimplementował w bibliotece standardowej...

Skoro to taka podstawowa technika, to pewnie już to ktoś zaimplementował w bibliotece standardowej...

Oto jest `std::unique_ptr`

`std::unique_ptr`

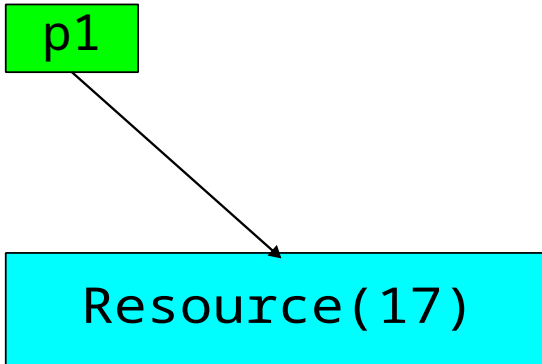
`std::unique_ptr` [C++11]

W bibliotece standardowej jest unikalny wskaźnik: `std::unique_ptr`. Nie da się go kopiować — jest on jedynym *właścicielem* obiektu, na który wskazuje. Przenoszenie pozbawia źródłowy obiekt wskaźnika na zasób i przypisuje go docelowemu.

Sprytny?

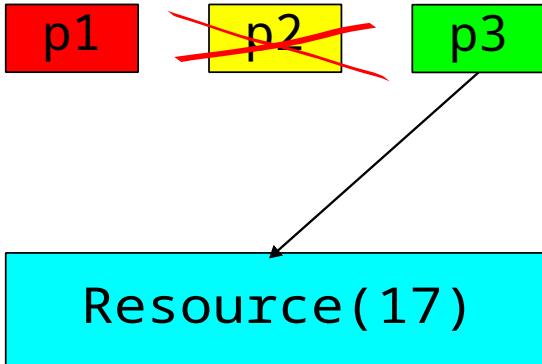
Spryt `std::unique_ptr` polega na wykorzystaniu RAI do *automatycznego* zwolnienia pamięci zaalokowanej *dynamicznie* — zapobiega to wyciekowi pamięci.

std::unique_ptr



```
1  #include <memory>
2
3  #include <paro/Resource.h>
4
5  int main(int ac, char *av[])
6  {
7      std::unique_ptr<Resource> p1(new Resource(17));
8      // std::unique_ptr<Resource> p2 = p1; // non-copyable!
9      std::unique_ptr<Resource> p3 = std::move(p1);
10
11     use_resource_the_intended_way(*p3);
12
13     p3.reset(); // delete resource
14     p1.reset(); // does nothing
15     return 0;
16 }
```

std::unique_ptr



```
1  #include <memory>
2
3  #include <paro/Resource.h>
4
5  int main(int ac, char *av[])
6  {
7      std::unique_ptr<Resource> p1(new Resource(17));
8      // std::unique_ptr<Resource> p2 = p1; // non-copyable!
9      std::unique_ptr<Resource> p3 = std::move(p1);
10
11     use_resource_the_intended_way(*p3);
12
13     p3.reset(); // delete resource
14     p1.reset(); // does nothing
15     return 0;
16 }
```

std::unique_ptr

Alternatywna funkcja usuwająca - malloc i free

```
1  #include <cstdlib> // malloc/free
2  #include <memory>
3
4  struct Deleter
5  {
6      void operator()(char* p) const noexcept { std::free(p); }
7  };
8  void perform_task(char* buffer, std::size_t buf_size);
9
10 int main(int ac, char* av[])
11 {
12     constexpr std::size_t buf_size = 1024;
13     std::unique_ptr<char, Deleter> buffer((char*)std::malloc(buf_size));
14     if (buffer) {
15         perform_task(buffer.get(), buf_size); // may throw!
16     }
17     return 0;
18 }
```


std::unique_ptr

Alternatywna funkcja usuwająca - fopen i fclose

```
1  #include <cstdio> // fopen/fclose
2  #include <memory>
3
4  struct Deleter
5  {
6      void operator()(std::FILE* file) const noexcept { std::fclose(file); }
7  };
8  void process_file(std::FILE* file);
9
10 int main(int ac, char* av[])
11 {
12     constexpr auto path = "/path/to/file.txt";
13     std::unique_ptr<std::FILE, Deleter> file(std::fopen(path, "r"));
14     if (file) {
15         process_file(file.get()); // may throw!
16     }
17     return 0;
18 }
```

std::unique_ptr

Standardowa funkcja usuwająca - std::default_delete

```
1  #include <memory>
2  #include <paro/Resource.h>
3
4  struct Deleter // std::default_delete<Resource>
5  {
6      void operator()(Resource* p) const noexcept
7      {
8          delete p;
9      }
10 };
11
12 int main(int ac, char* av[])
13 {
14     std::unique_ptr<Resource, Deleter> res(new Resource(17));
15     if (res) {
16         use_resource_the_intended_way(*res);
17     }
18     return 0;
19 }
```

`std::make_unique`

`std::make_unique` [C++14]

W standardowej bibliotece istnieje funkcja `std::make_unique`.
Służy do alokowania obiektów na sterce z jednoczesnym przypisaniem obiektu do `std::unique_ptr`.
Jest dostępna od C++14.

std::make_unique

```
1  #include <memory>
2  #include <paro/Resource.h>
3
4  struct Deleter // std::default_delete<Resource>
5  {
6      void operator()(Resource* p) const noexcept
7      {
8          delete p;
9      }
10 };
11
12 int main(int ac, char* av[])
13 {
14     std::unique_ptr<Resource, Deleter> res(new Resource(17));
15     if (res) {
16         use_resource_the_intended_way(*res);
17     }
18     return 0;
19 }
```

```
1  #include <memory>
2  #include <paro/Resource.h>
3
4  int main(int ac, char* av[])
5  {
6      auto res = std::make_unique<Resource>(17);
7      if (res) {
8          use_resource_the_intended_way(*res);
9      }
10     return 0;
11 }
```

std::make_unique

A jeśli utknęliśmy w 2011...

```
1  #include <memory>
2
3  template <typename T, typename... Args>
4  std::unique_ptr<T> make_unique(Args&&... args)
5  {
6      return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
7  }
```

A co z deleterem?

Funkcja `std::make_unique` pełni rolę skrótu do najczęściej pozyskiwanego w programie zasobu: kawałka pamięci na stercie alokowanego za pomocą operatora `new` i zwalnianego operatorem `delete`.

Bardziej zaawansowane przypadki obsługuje bezpośrednio konstruktor `std::unique_ptr`.

std::move

```
1 Data* create() { return new Data(0x19); }
2
3 void use(Data* p)
4 {
5     if (p) {
6         use_data_the_intended_way(p);
7         delete p;
8     }
9 }
10
11 int main(int argc, char const* argv[])
12 {
13     Data* data = create();
14     use(data);
15     // ...
16     use(data); // data is not NULL!
17     return 0;
18 }
```

- funkcja create alokuje pamięć i zwraca wskaźnik;

std::move

```
1 Data* create() { return new Data(0x19); }
2
3 void use(Data* p)
4 {
5     if (p) {
6         use_data_the_intended_way(p);
7         delete p;
8     }
9 }
10
11 int main(int argc, char const* argv[])
12 {
13     Data* data = create();
14     use(data);
15     // ...
16     use(data); // data is not NULL!
17     return 0;
18 }
```

- funkcja `create` alokuje pamięć i zwraca wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć;

std::move

```
1 Data* create() { return new Data(0x19); }
2
3 void use(Data* p)
4 {
5     if (p) {
6         use_data_the_intended_way(p);
7         delete p;
8     }
9 }
10
11 int main(int argc, char const* argv[])
12 {
13     Data* data = create();
14     use(data);
15     // ...
16     use(data); // data is not NULL!
17     return 0;
18 }
```

- funkcja `create` alokuje pamięć i zwraca wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć;
- odpowiedzialność za zasób jest nieustalona;

std::move

```
1 Data* create() { return new Data(0x19); }
2
3 void use(Data* p)
4 {
5     if (p) {
6         use_data_the_intended_way(p);
7         delete p;
8     }
9 }
10
11 int main(int argc, char const* argv[])
12 {
13     Data* data = create();
14     use(data);
15     // ...
16     use(data); // data is not NULL!
17     return 0;
18 }
```

- funkcja `create` alokuje pamięć i zwraca wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć;
- odpowiedzialność za zasób jest nieustalona;
- ten kod nie jest odporny na nieuwagę programisty.

std::move

```
1 Data* create() { return new Data(0x19); }
2
3 void use(Data* p)
4 {
5     if (p) {
6         use_data_the_intended_way(p);
7         delete p;
8     }
9 }
10
11 int main(int argc, char const* argv[])
12 {
13     Data* data = create();
14     use(data);
15     // ...
16     use(data); // data is not NULL!
17     return 0;
18 }
```

- funkcja `create` alokuje pamięć i zwraca wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć;
- odpowiedzialność za zasób jest nieustalona;
- ten kod nie jest odporny na nieuwagę programisty.

double free or corruption (top)
Aborted

Semantyka przenoszenia `std::move`.

W C++ istnieje tylko przekazywanie przez wartość, a czasem potrzebujemy zaznaczyć, że nie o zwykłą kopię nam chodzi.

std::move

```
1  std::unique_ptr<Data> create()
2  { // RVO
3      return std::make_unique<Data>(0x19);
4  }
5
6  void use(std::unique_ptr<Data> p)
7  {
8      if (p) {
9          use_data_the_intended_way(p.get());
10     }
11 }
12
13 int main(int argc, char const* argv[])
14 {
15     auto data = create();
16     use(std::move(data));
17     // ...
18     use(std::move(data)); // data is NULL!
19     return 0;
20 }
```

- funkcja `create` alokuje pamięć i zwraca *sprytny* wskaźnik;

std::move

```
1  std::unique_ptr<Data> create()
2  { // RVO
3      return std::make_unique<Data>(0x19);
4  }
5
6  void use(std::unique_ptr<Data> p)
7  {
8      if (p) {
9          use_data_the_intended_way(p.get());
10     }
11 }
12
13 int main(int argc, char const* argv[])
14 {
15     auto data = create();
16     use(std::move(data));
17     // ...
18     use(std::move(data)); // data is NULL!
19     return 0;
20 }
```

- funkcja `create` alokuje pamięć i zwraca *sprytny* wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć *automatycznie*;

std::move

```
1  std::unique_ptr<Data> create()
2  { // RVO
3      return std::make_unique<Data>(0x19);
4  }
5
6  void use(std::unique_ptr<Data> p)
7  {
8      if (p) {
9          use_data_the_intended_way(p.get());
10     }
11 }
12
13 int main(int argc, char const* argv[])
14 {
15     auto data = create();
16     use(std::move(data));
17     // ...
18     use(std::move(data)); // data is NULL!
19     return 0;
20 }
```

- funkcja `create` alokuje pamięć i zwraca *sprytny* wskaźnik;
- funkcja `use` używa zasobu i zwalnia pamięć *automatycznie*;
- odpowiedzialność za zasób jest *przenoszona*;

Ale to mi się nie kompiluje!

Składnia dla semantyki przenoszenia została wprowadzona w C++11. Podczas unowocześniania kodu może się zdarzyć, że dopisanie referencji do parametru typu `std::unique_ptr` rozwiązuje problem, a program zaczyna się kompilować...

```
1  #include <paro/Data.h>
2
3  std::unique_ptr<Data> load_initial_config();
4  void startup(std::unique_ptr<Data> const& init_cfg);
5
6  int main(int argc, const char* argv[])
7  {
8      auto init_cfg = load_initial_config();
9      startup(init_cfg); // init_cfg is NOT released
10
11     for (EVER) {
12         auto event = event_queue.front();
13         process(event);
14     }
15
16     // many events later...
17     return 0; // init_cfg is released here
18 }
```



std::move

```
1  #include <paro/Data.h>
2
3  std::unique_ptr<Data> load_initial_config();
4  void startup(std::unique_ptr<Data> const& init_cfg);
5
6  int main(int argc, const char* argv[])
7  {
8      auto init_cfg = load_initial_config();
9      startup(init_cfg); // init_cfg is NOT released
10
11     for (EVER) {
12         auto event = event_queue.front();
13         process(event);
14     }
15
16     // many events later...
17     return 0; // init_cfg is released here
18 }
```

std::move

```
1  #include <paro/Data.h>
2
3  std::unique_ptr<Data> load_initial_config();
4  void startup(std::unique_ptr<Data> init_cfg);
5
6  int main(int argc, const char* argv[])
7  {
8      auto init_cfg = load_initial_config();
9      startup(std::move(init_cfg)); // init_cfg is released after use
10
11     for (EVER) {
12         auto event = event_queue.front();
13         process(event);
14     }
15
16     // many events later...
17     return 0;
18 }
```


std::move

```
1  #include <paro/Data.h>
2
3  std::unique_ptr<Data> load_config() noexcept;
4  void startup(Data const& cfg) noexcept;
5
6  int main(int ac, const char* av[])
7  {
8      auto cfg = load_config();
9      startup(*cfg);
10
11     for (EVER) {
12         auto event = event_queue.front();
13         process(event, *cfg);
14     }
15
16     return 0;
17 }
```

Pobranie referencji do obiektu?

Jeśli chcemy wielokrotnie używać obiektu zarządzanego przez sprytny wskaźnik, możemy wyłuskać referencję na obiekt. Referencja (i zwykły wskaźnik) nie zmienia czasu życia obiektu, na który wskazuje.

Możemy zarządzać pamięcią na sterckie tak, jakby była na stosie

Używając `std::unique_ptr` otrzymujemy gwarancję, że dealokacja nastąpi niezależnie od sposobu, w jaki opuściliśmy zakres.

Do zmiany obowiązującego zakresu musimy wykorzystać semantykę przenoszenia `std::move`.

std::unique_ptr - podsumowanie

- Do `std::unique_ptr` możemy podać funkcję usuwającą (deleter).
- `std::make_unique` to alternatywa dla operatorów `new` i `delete`
- semantyka przenoszenia `std::move` służy w istocie do zmiany zakresu zasobu.

Gdzie jest wykorzystany RAI?

- sprytne wskaźniki (`std::unique_ptr`, `std::shared_ptr`, ...) - zapobieganie wyciekom pamięci;

Nie tylko `std::unique_ptr`

Gdzie jest wykorzystany RAI?

- sprytne wskaźniki (`std::unique_ptr`, `std::shared_ptr`, ...) - zapobieganie wyciekom pamięci;
- strumienie (`std::fstream`, `std::ostream`, ...) - dostęp do plików;

Gdzie jest wykorzystany RAII?

- sprytne wskaźniki (`std::unique_ptr`, `std::shared_ptr`, ...) - zapobieganie wyciekom pamięci;
- strumienie (`std::fstream`, `std::ostream`, ...) - dostęp do plików;
- STL (`std::vector`, `std::map`, ...) - elementy kolekcji są alokowane na stercie;

Gdzie jest wykorzystany RAII?

- sprytne wskaźniki (`std::unique_ptr`, `std::shared_ptr`, ...) - zapobieganie wyciekom pamięci;
- strumienie (`std::fstream`, `std::ostream`, ...) - dostęp do plików;
- STL (`std::vector`, `std::map`, ...) - elementy kolekcji są alokowane na stercie;
- `std::scoped_lock` - zwalnianie mutex'a przy wyjściu z zakresu - zapobiega blokadom (ang. *deadlock*);

Gdzie jest wykorzystany RAII?

- sprytne wskaźniki (`std::unique_ptr`, `std::shared_ptr`, ...) - zapobieganie wyciekom pamięci;
- strumienie (`std::fstream`, `std::ostream`, ...) - dostęp do plików;
- STL (`std::vector`, `std::map`, ...) - elementy kolekcji są alokowane na stercie;
- `std::scoped_lock` - zwalnianie mutexa przy wyjściu z zakresu - zapobiega blokadom (ang. *deadlock*);
- i wiele, wiele innych...

RAII - co to w ogóle za nazwa?

Dla współczesnego programisty wydaje się zagmatwana, bo została wymyślona, kiedy język C++ dopiero powstawał — nie było wtedy obsługi wyjątków. W tych mrocznych czasach konstruktor zawierał tylko bezpieczne operacje — cokolwiek bardziej ryzykownego delegowane było do osobnych funkcji, które zwracały kod błędu.

RAII - dygresja o nazwie

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  struct GameData
5  {
6      GameData() : player_level(1) {}
7
8      FILE* save;
9      pthread_mutex_t the_mutex;
10     int player_level;
11 };
12
13 int main(int argc, char* argv[])
14 {
15     GameData data; // initialization, then
16     // stepwise resource acquisition
17     if (0 != pthread_mutex_init(&data.the_mutex, NULL))
18         return 2;
19     data.save = fopen("save.dat", "r");
20     if (NULL == data.save)
21         return 1;
22     return 0;
23 }
```

RAII - dygresja o nazwie

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  struct GameData
5  {
6      GameData() : player_level(1) {}
7
8      FILE* save;
9      pthread_mutex_t the_mutex;
10     int player_level;
11 };
12
13 int main(int argc, char* argv[])
14 {
15     GameData data; // initialization, then
16     // stepwise resource acquisition
17     if (0 != pthread_mutex_init(&data.the_mutex, NULL))
18         return 2;
19     data.save = fopen("save.dat", "r");
20     if (NULL == data.save)
21         return 1;
22     return 0;
23 }
```

```
1  #include <fstream>
2  #include <mutex>
3
4  struct GameData
5  {
6      GameData() : save("save.dat"), player_level(1) {}
7
8      std::fstream save;
9      std::mutex the_mutex;
10     int player_level;
11 };
12
13 int main(int argc, char* argv[])
14 {
15     // resources are acquired during initialization
16     GameData data;
17
18     return 0;
19 }
```


Shared Resource

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Nie zawsze jednak możemy przydzielić odpowiedzialność za zasób na wyłączność.
Na szczęście tu również mamy pełne wsparcie ze strony biblioteki standardowej.

std::shared_ptr

```
1  #include <memory>
2
3  #include <paro/Resource.h>
4
5  int main(int ac, char *av[])
6  {
7      std::shared_ptr<Resource> p1(new Resource(17));
8      std::shared_ptr<Resource> p2 = p1; // copying is available now!
9      std::shared_ptr<Resource> p3 = std::move(p1); // still possible
10
11      use_resource_the_intended_way(*p3);
12      return 0; // p3, p2, p1 went out-of-scope, object is destroyed
13 }
```

`std::shared_ptr`

Skąd `std::shared_ptr` wie, że jest tym ostatnim?

Służy mu do tego mechanizm zliczania referencji.

1. Każdy `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.

`std::shared_ptr`

Skąd `std::shared_ptr` wie, że jest tym ostatnim?

Służy mu do tego mechanizm zliczania referencji.

1. Każdy `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.
2. Zniszczenie `std::shared_ptr` powoduje dekrementację licznika.

`std::shared_ptr`

Skąd `std::shared_ptr` wie, że jest tym ostatnim?

Służy mu do tego mechanizm zliczania referencji.

1. Każdy `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.
2. Zniszczenie `std::shared_ptr` powoduje dekrementację licznika.
3. Gdy licznik spadnie do zera, wołany jest destruktork obiektu.

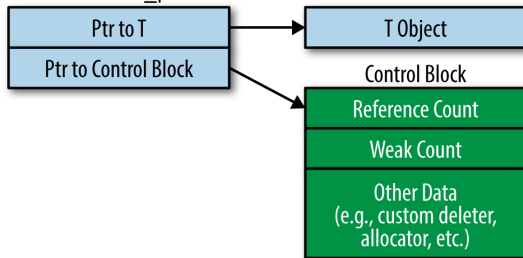
`std::shared_ptr`

Każdy `std::shared_ptr` to
w rzeczywistości dwa wskaźniki:

na zarządzany obiekt

na blok kontrolny

`std::shared_ptr<T>`



std::shared_ptr

Niestandardowe deletey

```
1  #include <memory>
2  #include <vector>
3
4  #include <paro/Resource.h>
5
6  int main(int ac, char* av[])
7  {
8      auto deleterOdd = [](Resource* r) { delete r; }; // custom deleter
9      auto deleterEven = [](Resource* r) { delete r; }; // different types
10
11     std::shared_ptr<Resource> p1(new Resource(13), deleterOdd);
12     std::shared_ptr<Resource> p2(new Resource(26), deleterEven);
13     // impossible with std::unique_ptr!
14     std::vector<std::shared_ptr<Resource>> vec{p1, p2};
15     return 0;
16 }
```

`std::shared_ptr`

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.
- Blok kontrolny nie może leżeć na stosie, bo każdy wątek ma swój stos.

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.
- Blok kontrolny nie może leżeć na stosie, bo każdy wątek ma swój stos.
- Wszystkie operacje na bloku kontrolnym muszą być synchronizowane.

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.
- Blok kontrolny nie może leżeć na stosie, bo każdy wątek ma swój stos.
- Wszystkie operacje na bloku kontrolnym muszą być synchronizowane.
- Cierpi wydajność, ale daje gwarancję poprawności.

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.
- Blok kontrolny nie może leżeć na stosie, bo każdy wątek ma swój stos.
- Wszystkie operacje na bloku kontrolnym muszą być synchronizowane.
- Cierpi wydajność, ale daje gwarancję poprawności.
- Tylko blok kontrolny jest synchronizowany!

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.
- Blok kontrolny nie może leżeć na stosie, bo każdy wątek ma swój stos.
- Wszystkie operacje na bloku kontrolnym muszą być synchronizowane.
- Cierpi wydajność, ale daje gwarancję poprawności.
- **Tylko blok kontrolny jest synchronizowany!**
- Dostęp do zasobu musimy zabezpieczyć samodzielnie.

`std::make_shared`

`std::make_shared` [C++11]

Analogicznie do `std::make_unique`, istnieje też funkcja `std::make_shared`. Dostępna od samego początku istnienia `std::shared_ptr`, czyli od C++11.

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji.

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji. Kompilator ma dowolność w przeplataniu ich wykonania, o ile uzna to za optymalizację:

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji. Kompilator ma dowolność w przeplataniu ich wykonania, o ile uzna to za optymalizację:

1. wykonanie operator `new`;

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji. Kompilator ma dowolność w przeplataniu ich wykonania, o ile uzna to za optymalizację:

1. wykonanie operator `new`;
2. wykonanie `nastyFunction`;
3. `nastyFunction` rzuca wyjątkiem;

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji. Kompilator ma dowolność w przeplataniu ich wykonania, o ile uzna to za optymalizację:

1. wykonanie operator `new`;
2. wykonanie `nastyFunction`;
3. `nastyFunction` rzuca wyjątkiem;
4. blok pamięci zarezerwowany przez operator `new` nie jest zwalniany;

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
16     return 0;
17 }
```

C++ nie gwarantuje kolejności wywołania argumentów w wywołaniu funkcji. Kompilator ma dowolność w przeplataniu ich wykonania, o ile uzna to za optymalizację:

1. wykonanie operator `new`;
2. wykonanie `nastyFunction`;
3. `nastyFunction` rzuca wyjątkiem;
4. blok pamięci zarezerwowany przez operator `new` nie jest zwalniany;
5. doszło do wycieku pamięci.

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::make_shared<Resource>(17), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared na ratunek!*

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::make_shared<Resource>(17), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared na ratunek!*

1. wykonanie std::make_shared - alokacja i konstruktory;

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::make_shared<Resource>(17), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared na ratunek!*

1. wykonanie std::make_shared - alokacja i konstruktory;
2. wykonanie nastyFunction;

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::make_shared<Resource>(17), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared na ratunek!*

1. wykonanie std::make_shared - alokacja i konstruktory;
2. wykonanie nastyFunction;
3. nastyFunction rzuca wyjątkiem;

std::make_shared

```
1  #include <paro/Resource.h>
2  #include <memory>
3
4  int nastyFunction(int param)
5  {
6      if (param % 2)
7          throw std::runtime_error("Oops!");
8      return 42;
9  }
10
11 void foo(std::shared_ptr<Resource> res, int i) {}
12
13 int main(int argc, char const* argv[])
14 {
15     foo(std::make_shared<Resource>(17), nastyFunction(17));
16     return 0;
17 }
```

std::make_shared na ratunek!*

1. wykonanie std::make_shared - alokacja i konstruktory;
2. wykonanie nastyFunction;
3. nastyFunction rzuca wyjątkiem;
4. wykonanie destruktor
std::shared_ptr, brak wycieku.

`std::make_shared`

`std::make_shared` na ratunek!*

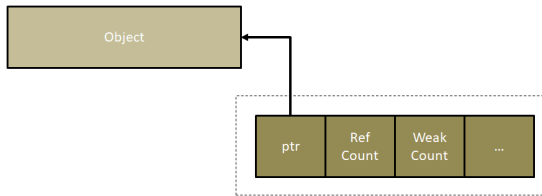
Od standardu C++17 zostało to poprawione.

Kompilator nadal może dowolnie zmieniać kolejność obliczania argumentów, ale każdy musi zostać w pełni ewaluowany bez przeplatania instrukcji.

`std::make_shared`

`std::shared_ptr`

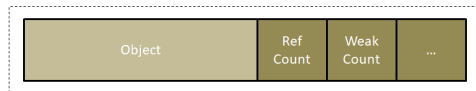
Dwie osobne alokacje: dla obiektu oraz dla bloku kontrolnego.
Możliwe zwolnienie pamięci po obiekcie do systemu operacyjnego po zawołaniu destruktora.



`std::make_shared`

`std::make_shared`

Jedna wspólna alokacja.
Nawet gdy obiektu już nie ma, obszar w pamięci wciąż jest zablokowany.
Nie da się oddać fragmentu pamięci. Albo wszystko, albo nic.



`std::weak_ptr`

`std::weak_ptr` to wskaźnik-obszernik.

- Nie zwiększa licznika odwołań.
- Nie uczestniczy we współdzielonym zarządzaniu obiektem.
- Trzeba go awansować na `std::shared_ptr` przed użyciem.
- Potrafi przerwać cykl odwołań między `std::shared_ptr`.
- Dopóki jest choć jeden `std::weak_ptr`, blok kontrolny musi żyć!

std::weak_ptr

```
1 void check(std::weak_ptr<int> w)
2 {
3     std::cout << "use_count_=" << w.use_count() << '\n';
4     // try to upgrade to std::shared_ptr, better than just checking .expired()
5     if (auto stillAlive = w.lock()) {
6         std::cout << "I_am_here!\n";
7     } else {
8         std::cout << "too_late...\n";
9     }
10 }
11
12 int main(int ac, char *av[])
13 {
14     std::weak_ptr<int> weak;
15     {
16         auto shared = std::make_shared<int>(17);
17         weak = shared;
18         check(weak);
19     }
20     check(weak);
21     return 0;
22 }
```

```
use_count = 1
I am here!
use_count = 0
too late...
```

std::weak_ptr

```
1  struct B;
2  struct A
3  {
4      std::shared_ptr<B> b;
5      ~A() { std::cout << "destructing A\n"; }
6  };
7
8  struct B
9  {
10     std::shared_ptr<A> a;
11     ~B() { std::cout << "destructing B\n"; }
12 };
13
14 void useBoth()
15 {
16     auto a = std::make_shared<A>();
17     auto b = std::make_shared<B>();
18     a->b = b;
19     b->a = a;
20 }
```

```
1  int main()
2  {
3      useBoth();
4      std::cout << "Finished using A and B\n";
5  }
```

Finished using A and B

std::weak_ptr

```
1 struct B;
2 struct A
3 {
4     std::shared_ptr<B> b;
5     ~A() { std::cout << "destructing A\n"; }
6 };
7
8 struct B
9 {
10     std::weak_ptr<A> a;
11     ~B() { std::cout << "destructing B\n"; }
12 };
13
14 void useBoth()
15 {
16     auto a = std::make_shared<A>();
17     auto b = std::make_shared<B>();
18     a->b = b;
19     b->a = a;
20 }
```

```
1 int main()
2 {
3     useBoth();
4     std::cout << "Finished using A and B\n";
5 }
```

```
destructing A
destructing B
Finished using A and B
```

Rule of Zero

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Rule of Zero

It's broken!

```
1  #include <cstddef>
2  #include <cstring>
3
4  struct Nickname
5  {
6      Nickname(const char* input)
7      {
8          if (input) {
9              std::size_t n = std::strlen(input) + 1;
10             nick_ = new char[n];
11             std::memcpy(nick_, input, n);
12         }
13     }
14
15     ~Nickname() { delete[] nick_; }
16
17 private:
18     char* nick_ = nullptr;
19 };
```

```
1  int main()
2  {
3      Nickname n1("Whatever");
4      // copy constructor implicitly generated by compiler
5      Nickname n2(n1);
6      // same for copy assignment operator
7      Nickname n3 = n2;
8  } // n3, n2, n1 go out-of-scope; destructors called;
9      // double free and crash
```

Rule of Zero

It's still broken!

```
1  #include <cstddef>
2  #include <cstring>
3
4  struct Nickname
5  {
6      Nickname(const char* input)
7      {
8          if (input) {
9              std::size_t n = std::strlen(input) + 1;
10             nick_ = new char[n];
11             std::memcpy(nick_, input, n);
12         }
13     }
14
15     ~Nickname() { if (nick_) delete[] nick_; }
16
17 private:
18     char* nick_ = nullptr;
19 };
```

```
1  int main()
2  {
3      Nickname n1("Whatever");
4      // copy constructor implicitly generated by compiler
5      Nickname n2(n1);
6      // same for copy assignment operator
7      Nickname n3 = n2;
8  } // n3, n2, n1 go out-of-scope; destructors called;
9      // double free and crash
```

Rule of Zero

Unbelievable - still broken!

```
1  struct Nickname
2  {
3      Nickname(const char* input)
4      {
5          if (input) {
6              std::size_t n = std::strlen(input) + 1;
7              nick_ = new char[n];
8              std::memcpy(nick_, input, n);
9          }
10     }
11     Nickname(const Nickname& other) {
12         if (other.nick_) {
13             std::size_t n = std::strlen(other.nick_) + 1;
14             nick_ = new char[n];
15             std::memcpy(nick_, other.nick_, n);
16         }
17     }
18     ~Nickname() { if (nick_) delete[] nick_; }
19
20 private:
21     char* nick_ = nullptr;
22 };

1  int main()
2  {
3      Nickname n1("Whatever");
4      // copy constructor written by hand
5      Nickname n2(n1);
6      // copy assignment generated by compiler
7      Nickname n3 = n2;
8  } // n3, n2, n1 go out-of-scope; destructors called;
9      // double free and crash
```

Rule of Zero

It... works? (still ugly)

```
1 Nickname(const Nickname& other) {
2     if (other.nick_) {
3         std::size_t n = std::strlen(other.nick_) + 1;
4         nick_ = new char[n];
5         std::memcpy(nick_, other.nick_, n);
6     }
7 }
8 Nickname& operator=(const Nickname& other) {
9     if (other.nick_) {
10         std::size_t n = std::strlen(other.nick_) + 1;
11         nick_ = new char[n];
12         std::memcpy(nick_, other.nick_, n);
13     }
14     return *this;
15 }
16 ~Nickname() { if (nick_) delete[] nick_; }
```

```
1 int main()
2 {
3     Nickname n1("Whatever");
4     // copy constructor written by hand
5     Nickname n2(n1);
6     // copy assignment written by hand
7     Nickname n3 = n2;
8 } // n3, n2, n1 go out-of-scope; destructors called;
9 // everything works fine... for now...
```

Rule of Zero

Rule of three

Jeżeli typ (klasa) potrzebuje do działania któregośkolwiek z poniższych:

- destruktor,
- konstruktora kopiującego,
- kopiującego operatora przypisania,

to musimy zdefiniować wszystkie trzy celem zagwarantowania poprawnego działania.

C++ precyzyjnie określa kiedy i jakie metody specjalne zostaną wygenerowane przez kompilator w ich domyślnych wersjach.

Special Members

compiler implicitly declares

user declares		default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
	Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
	Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
	default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
	destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
	copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
	copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
	move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
	move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Rule of Zero

Rule of three

```
1  Nickname(const Nickname& other)
2      : Nickname(other.nick_) {}
3
4  Nickname& operator=(const Nickname& other)
5  {
6      if (this == &other)
7          return *this; // check self-assignment!
8
9      std::size_t n = std::strlen(other.nick_) + 1;
10     char* new_nick = new char[n];
11     std::memcpy(new_nick, other.nick_, n);
12     delete[] nick_;
13
14     nick_ = new_nick;
15     return *this;
16 }
```

```
1  int main()
2  {
3      Nickname n1("Whatever");
4      // copy constructor is defined by us now
5      Nickname n2(n1);
6      // same for copy assignment operator
7      Nickname n3 = n2;
8  } // n3, n2, n1 go out-of-scope; destructors called;
9      // everything works correctly
```


Od C++11 i wprowadzenia `std::move`, pojawiły się dwie dodatkowe metody do uwzględnienia:

- konstruktor przenoszący
- przenoszący operator przypisania

Rule of Zero

Rule of five

```
1  Nickname(Nickname&& other) noexcept
2      : nick_(std::exchange(other.nick_, nullptr))
3  {}
4
5  Nickname& operator=(Nickname&& other) noexcept
6  {
7      std::swap(nick_, other.nick_);
8      return *this;
9  }
```

```
1  int main()
2  {
3      Nickname n1("Whatever");
4      // copy constructor is defined by us now
5      Nickname n2(n1);
6      // same for copy assignment operator
7      Nickname n3 = n2;
8      // ditto for move
9      Nickname n4 = std::move(n3);
10 } // n4, n3, n2, n1 go out-of-scope; destructors called;
11 // everything works correctly
```

Operacje przenoszące

Brak zdefiniowania tych dwóch metod najczęściej nie jest błędem, a straconą szansą na optymalizację.

Rule of Zero

Ale czy musimy się tak męczyć?

Ale czy musimy się tak męczyć?

Oczywiście, że nie.

Ale czy musimy się tak męczyć?

Oczywiście, że nie.

Korzystajmy z dobrodziejstw RAI1 oraz klas implementujących ten wzorzec.

Rule of Zero

Rule of zero

```
1  #include <string>
2
3  struct Nickname
4  {
5      explicit Nickname(std::string input) : nick_(std::move(input)) {}
6  private:
7      std::string nick_; // std::string is a RAII-wrapper around char*
8  };
9
10 int main(int ac, char* av[])
11 {
12     Nickname n1("Whatever");
13     // copy constructor is implicitly generated by compiler
14     Nickname n2(n1);
15     // same for copy assignment operator
16     Nickname n3 = n2;
17     // same for move constructor
18     Nickname n4 = std::move(n2);
19 } // n4, n3, n2, n1 go out-of-scope - destructors called
20 // everything works correctly automagically thanks to RAII
```

Wydajność

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Jedną z obietnic, jakie język C++ nam składa jest "zero overhead abstraction". Sprawdźmy, czy istotnie smart pointery dają nam zaletę bezpieczeństwa i wygody użycia, bez zabierania wydajności.

T*

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6      char tab_[42];
7  };
8
9  int main(int ac, char *av[])
10 {
11     constexpr unsigned size = 10'000'000u;
12     std::vector<Data*> vec;
13     vec.reserve(size);
14     for (auto i = 0u; i < size; i++) {
15         auto p = new Data();
16         vec.push_back(std::move(p));
17     }
18     for (auto p : vec) delete p;
19 }
```

std::unique_ptr<T>

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6      char tab_[42];
7  };
8
9  int main(int ac, char *av[])
10 {
11     constexpr unsigned size = 10'000'000u;
12     std::vector<std::unique_ptr<Data>> vec;
13     vec.reserve(size);
14     for (auto i = 0u; i < size; i++) {
15         std::unique_ptr<Data> p{new Data()};
16         vec.push_back(std::move(p));
17     }
18 }
```

std::shared_ptr<T>

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6      char tab_[42];
7  };
8
9  int main(int ac, char *av[])
10 {
11     constexpr unsigned size = 10'000'000u;
12     std::vector<std::shared_ptr<Data>> vec;
13     vec.reserve(size);
14     for (auto i = 0u; i < size; i++) {
15         std::shared_ptr<Data> p{new Data()};
16         vec.push_back(std::move(p));
17     }
18 }
```

std::weak_ptr<T>

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6      char tab_[42];
7  };
8
9  int main(int ac, char *av[])
10 {
11     constexpr unsigned size = 10'000'000u;
12     std::vector<std::shared_ptr<Data>> vec;
13     std::vector<std::weak_ptr<Data>> vec_observers;
14     vec.reserve(size);
15     vec_observers.reserve(size);
16     for (auto i = 0u; i < size; i++) {
17         std::shared_ptr<Data> p{new Data()};
18         std::weak_ptr<Data> weak{p};
19         vec.push_back(std::move(p));
20         vec_observers.push_back(std::move(weak));
21     }
22 }
```

std::make_shared<T>

```
1  #include <memory>
2  #include <vector>
3
4  struct Data
5  {
6      char tab_[42];
7  };
8
9  int main(int ac, char *av[])
10 {
11     constexpr unsigned size = 10'000'000u;
12     std::vector<std::shared_ptr<Data>> vec;
13     vec.reserve(size);
14     for (auto i = 0u; i < size; i++) {
15         auto p = std::make_shared<Data>();
16         vec.push_back(std::move(p));
17     }
18 }
```

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
-------------	----------	----------	-------------

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
T*	0.59	10'000'001	610

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
T*	0.59	10'000'001	610
std::unique_ptr<T>	0.58	10'000'001	610

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
T*	0.59	10'000'001	610
std::unique_ptr<T>	0.58	10'000'001	610
std::shared_ptr<T>	1.00	20'000'001	1043

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
T*	0.59	10'000'001	610
std::unique_ptr<T>	0.58	10'000'001	610
std::shared_ptr<T>	1.00	20'000'001	1043
std::weak_ptr<T>	1.21	20'000'002	1192

- GCC 11.3
- Pomiary wykonane przy pomocy:
 - *time* (real) -- czas
 - *valgrind* (memcheck) -- alokacje
 - *valgrind* (massif) -- zużycie pamięci

nazwa testu	czas [s]	alokacje	pamięć [MB]
T*	0.59	10'000'001	610
std::unique_ptr<T>	0.58	10'000'001	610
std::shared_ptr<T>	1.00	20'000'001	1043
std::weak_ptr<T>	1.21	20'000'002	1192
std::make_shared<T>	0.70	10'000'001	839

And that's all folks!

Pytania?

And that's all folks!

Warunki zaliczenia

materiały do pobrania	memory-management-2024-exercises.zip
polecenia	exercises/README_PL.md
e-mail do prowadzących	bogumil.chojnowski@nokia.com michal.orynicz@nokia.com
tytułem	[PARO2024] Memory Management C++
z załącznikiem	archiwum ZIP
	tylko zawartość katalogu z ćwiczeniami bez skompilowanych binarek. slajdów też nie potrzebujemy
w terminie	2024-11-29T23:59+01:00

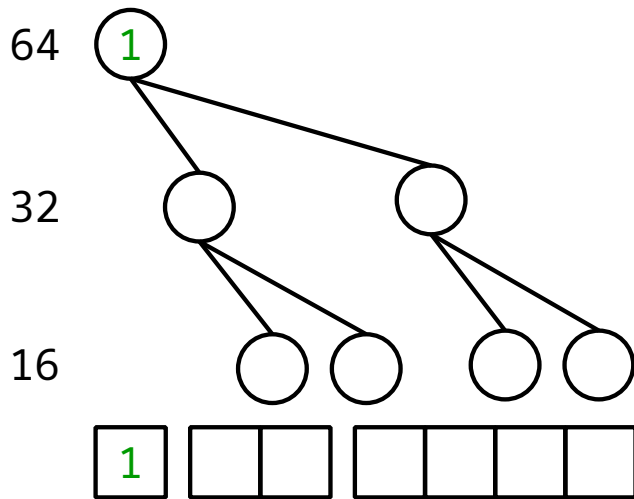


Algorytmy i Struktury Danych

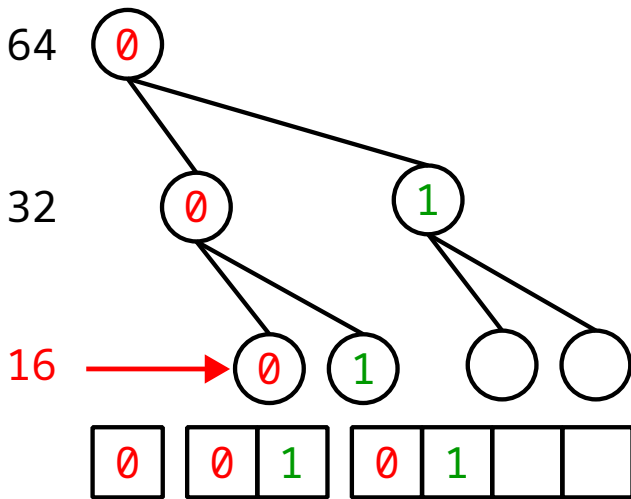
Algorytmy i Struktury Danych

Memory Management API

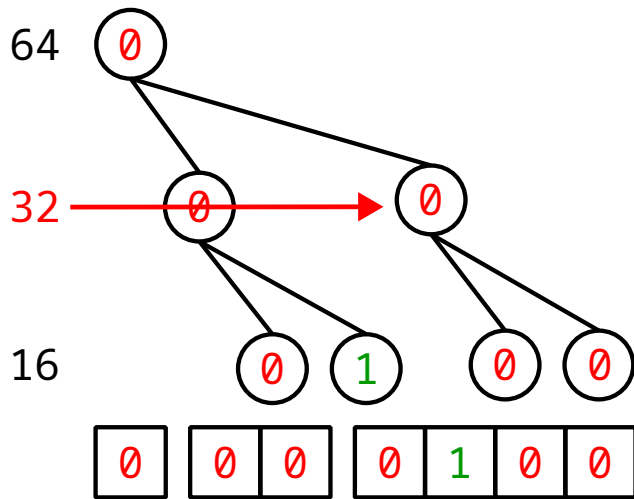
Struktura danych: kopiec



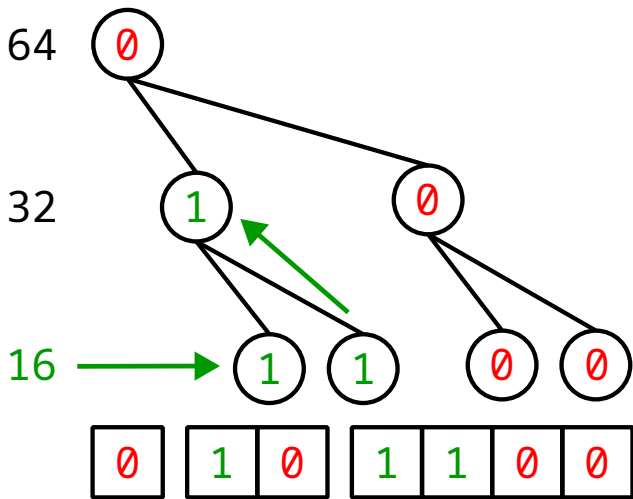
Struktura danych: kopiec



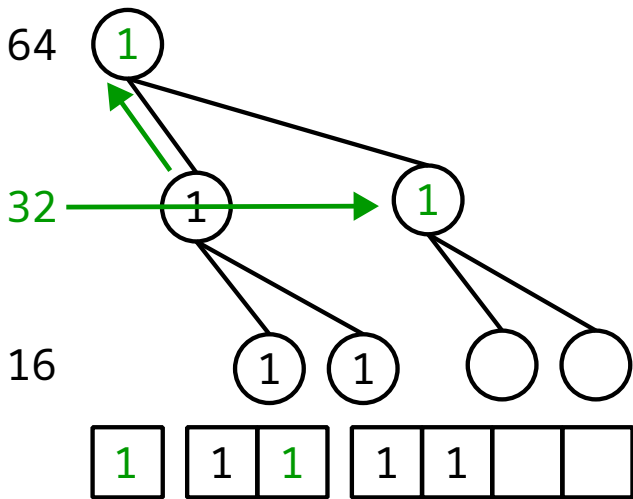
Struktura danych: kopiec



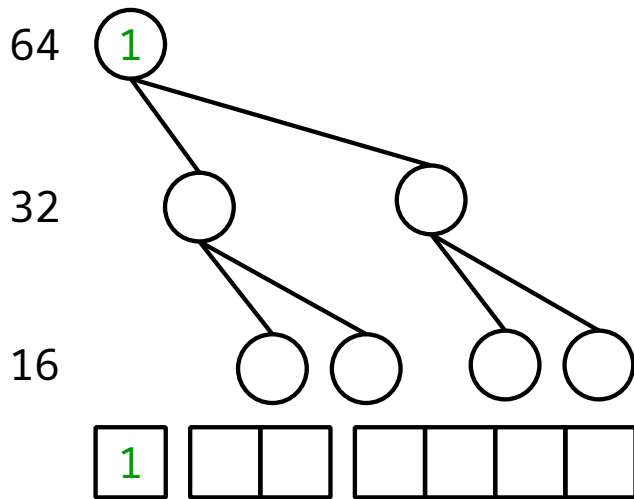
Struktura danych: kopiec



Struktura danych: kopiec



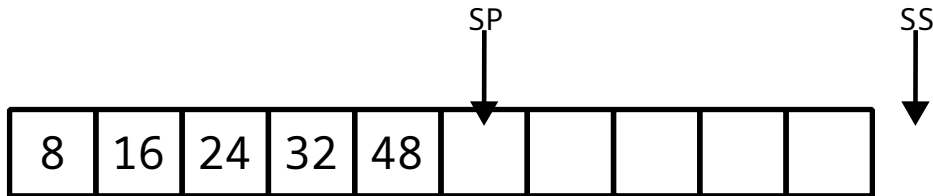
Struktura danych: kopiec



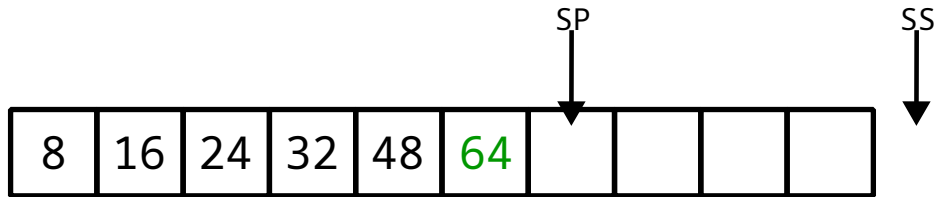
Struktura danych: stos

Implementacja stosu zawiera tablicę o ustalonym rozmiarze. Nowe elementy (operacja `push`) są zapisywane w komórce pamięci wskazywanej przez wskaźnik stosu, po czym wskaźnik jest przesuwany na następny element. Zdejmowanie wartości ze stosu (operacja `pop`) polega na cofnięciu wskaźnika i odczytania wartości przezeń wskazywanego.

push(64)



Struktura danych: stos

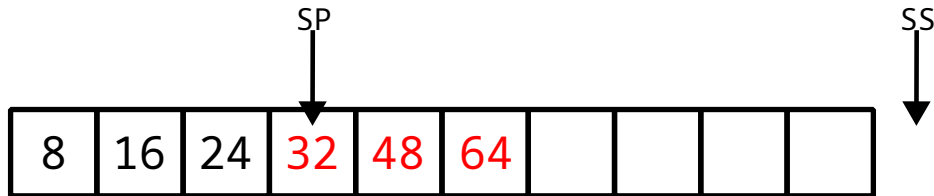


Struktura danych: stos

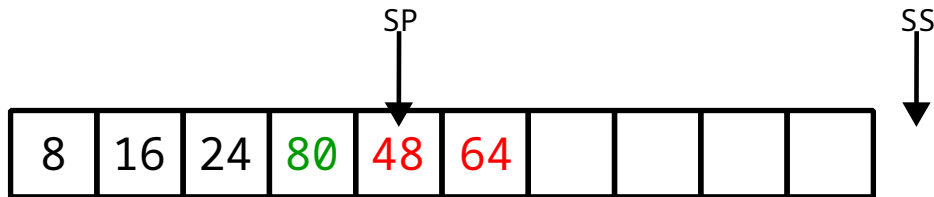
pop() 64

pop() 48

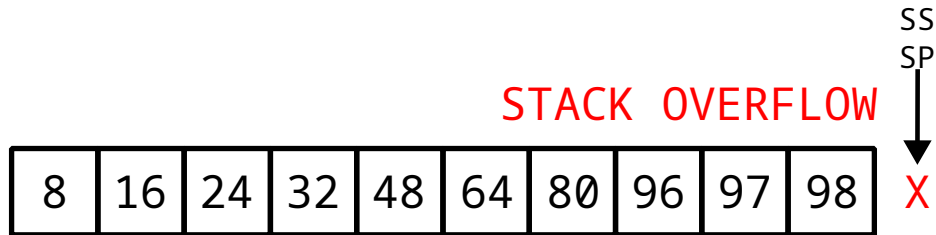
pop() 32



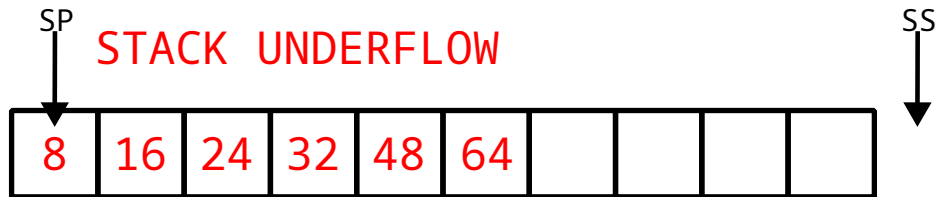
push(**80**)



push(99)



pop ()



Memory Management API

Algorytmy i Struktury Danych

Memory Management API

Interfejs programistyczny podsystemu pamięci

Poziom 1: linux kernel

Linux kernel: mmap i munmap

Mapowanie stron pamięci operacyjnej na przestrzeń adresową programu zostało zaimplementowane w jądrze systemu operacyjnego i udostępniane w nagłówku `<sys/mman.h>` jako funkcje `mmap` i `munmap`. To z nich korzystają `malloc` i `free`.

Interfejs programistyczny podsystemu pamięci

Poziom 2: biblioteka standardowa C

`cstdlib`: `malloc` i `free`

Biblioteka standardowa C dostarcza funkcje `malloc` i `free`, które zarządzają pamięcią pobraną z systemu i udostępniają ją do programu w formie wskaźników do zarezerwowanych bloków. Z nich korzystają operatory `new` i `delete`

Operatory `new` i `delete`

Operator `new` wykorzystuje funkcję `malloc` do pobrania wskaźnika na blok pamięci rozmiaru równego rozmiarowi tworzonego obiektu, następnie wykonywany jest konstruktor klasy obiektu. Operator `delete` woła destruktory obiektu, a następnie funkcję `free` co zwalnia blok pamięci.

Nie mieszaj metod alokacji!

Usunięcie obiektu, utworzonego na stercie przez operator `new`, za pomocą funkcji `free` nie wykona destruktora.

Usunięcie bloku pamięci zaalokowanego przez `malloc` przez operator `delete` wykona destruktor na obiekcie, którego konstruktor nie został wykonany.

Niestosowanie się do powyższych wykona *niezdefiniowane zachowanie*.