PARO2024 - Wrocław University

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?

Wprowadzenie

Zarządzanie pamięcią

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

Alokacja

Alokacja zasobów

Co to jest alokacja zasobów?

Alokacja to przypisywanie zasobów do możliwości ich użycia.

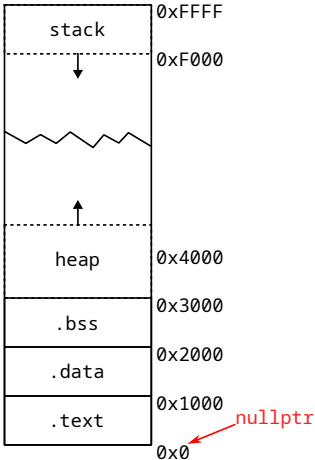
Alokacja zasobów

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` sterta

`stack` stos (rośnie w dół)

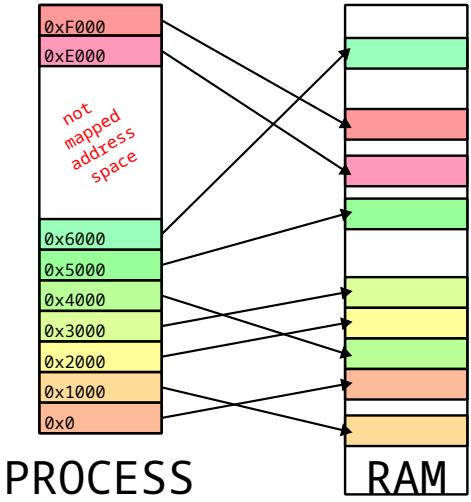
`nullptr` - on też tu jest!

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.

Przestrzeń adresowa

Algorytm przydzielania pamięci procesom

Chcemy, żeby był szybki

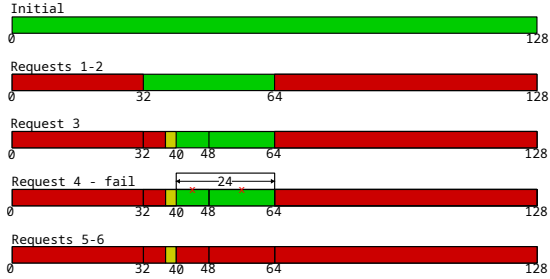
Częstotliwość używania sprawia, że sumaryczny koszt wykonania ma wpływ na szybkość działania programu.

Chcemy, żeby był dokładny

Algorytm nie powinien obchodzić się z pamięcią w sposób rozrzutny.
Programy potrzebują coraz większej ilości pamięci do działania.
Nie zawsze możemy sobie pozwolić na marnotrawstwo.

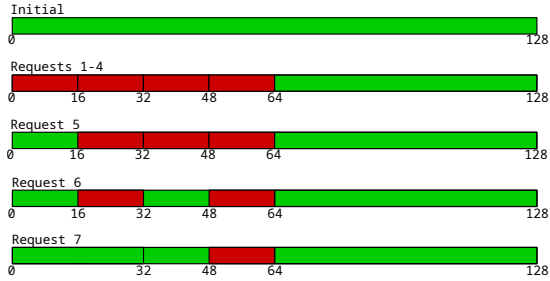
Algorytm bloków bliźniaczych

```cpp
#include "Heap.hpp"

void demo_fragmentation()
{
    Heap h(128);

    h.allocate(32);
    h.allocate(64);
    h.allocate(5); // 3 bytes wasted

    h.allocate(20); // fail!

    h.allocate(16); // OK
    h.allocate(8); // OK
}
```



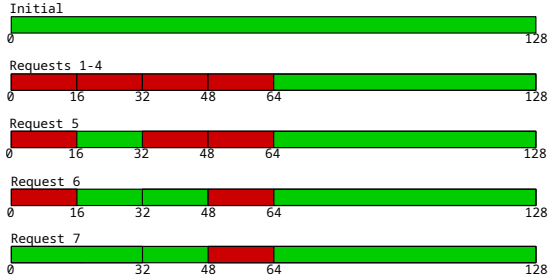
Algorytm bloków bliźniaczych

```cpp
#include "Heap.hpp"

void demo_deallocation_1()
{
    Heap h(128);

    h.allocate(16);
    h.allocate(16);
    h.allocate(16);
    h.allocate(16);

    h.deallocate(0);
    h.deallocate(32);
    h.deallocate(16); // coalescing
}
```



Algorytm bloków bliźniaczych

```
#include "Heap.hpp"

void demo_deallocation_2()
{
    Heap h(128);

    h.allocate(16);
    h.allocate(16);
    h.allocate(16);
    h.allocate(16);

    h.deallocate(16);
    h.deallocate(32);
    h.deallocate(0); // coalescing
}
```



Algorytm bloków bliźniaczych

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**

Algorytm bloków bliźniaczych

Sterta, inaczej kopiec

Informacje o zajętości bloków można przedstawiać w formie kopca — struktury danych reprezentującej drzewo binarne.
Przykład kopca w załączniku **Struktura danych: kopiec**

Algorytm bloków bliźniaczych

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.

Rodzaje alokacji

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 stercie)

```
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 }
```

Alokacja dynamiczna

Sterta

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 **sterta** (ang. *heap*).

Alokacja dynamiczna

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  }
```

Alokacja dynamiczna

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

```cpp
#include <paro/Object.hpp>

int main(int ac, char* av[])
{
    constexpr auto arr_size = 10u;
    Object** arr = new Object*[arr_size];
    for (int i = 0; i < arr_size; ++i)
        arr[i] = new Object(3*i, 2*i+14, i+92);
    for (int i = 0; i < arr_size; ++i)
        delete arr[i];
    delete[] arr;
    return 0;
}
```

Alokacja dynamiczna

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

Alokacja dynamiczna - podsumowanie

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

Można przeciążać operatory, aby zaimplementować alternatywne algorytmy alokacji.

Alokacja automatyczna

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*).

Alokacja automatyczna

```
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 zwracanym?
  - 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;

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.

Alokacja automatyczna

A co ze zmiennymi globalnymi?

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

Alokacja automatyczna

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*.

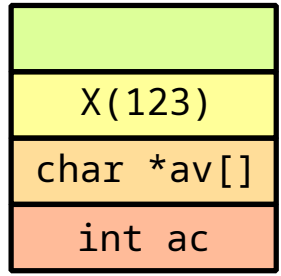
Zakres zmiennych

```cpp
#include <print>

struct X
{
    X(int x) : x(x) { std::println("X({})␣created", x); }
    ~X() { std::println("X({})␣is␣dead", x); }
    int x;
};

int main(int ac, char* av[])
{
    X a(123);
    {
        X b(42);
    }
    X c(89);

    return 0;
}
```

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

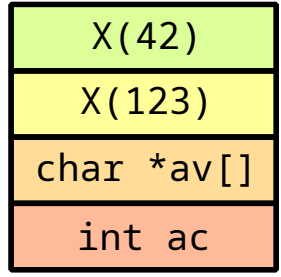
Zakres zmiennych

```
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 }
```



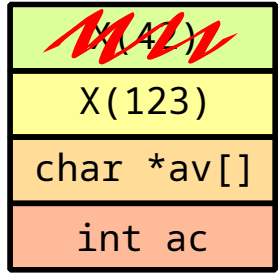
Zakres zmiennych

```
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 }
```



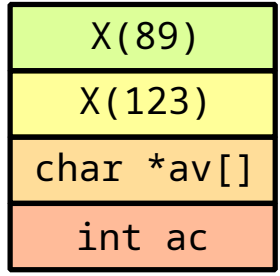
Zakres zmiennych

```
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 }
```



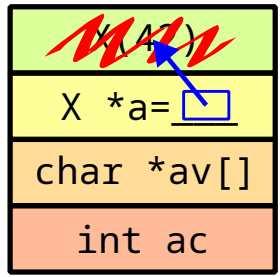
Zakres zmiennych

```
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 }
```



Zakres zmiennych

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 zmiennych

Zakres

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

Zakres zmiennych

Obiekty automatyczne

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

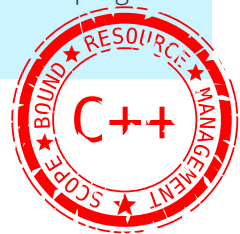
1. nie `const/constexpr`, bo kompilator ma prawo umieścić je poza stosem;
2. nie `static`, bo takie są alokowane razem z obiektami globalnymi.

Zakres zmiennych

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

Argumenty funkcji

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

```cpp
#include <print>

void ptr(int* ptr)
{
    if (ptr) {
        std::println("{}", *ptr);
        delete ptr;
    }
    ptr = nullptr;
}

int main(int ac, char* av[])
{
    int* p = new int(45);
    ptr(p);
    std::println("Is␣nullptr?␣{}.", p == nullptr);

    return 0;
}
```

Modyfikowanie kopii nie wpływa na oryginał.

Argumenty funkcji

```cpp
#include <print>

void ptr(int* ptr)
{
    if (ptr) {
        std::println("{}", *ptr);
        delete ptr;
    }
    ptr = nullptr;
}

int main(int ac, char* av[])
{
    int* p = new int(45);
    ptr(p);
    std::println("Is␣nullptr?␣{}.", p == nullptr);

    return 0;
}
```

Modyfikowanie kopii nie wpływa na oryginał.

```
45
Is nullptr? false.

Process finished with exit code 0
```

Argumenty funkcji

```cpp
char& get_initial(S s) { return s.n; }

int main(int ac, char* av[])
{
    auto& s = get_initial(S{'B'});
    std::println("In main again: {}", s);
    return 0;
}
```

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

```
warning: reference to local variable ‘s’ returned [-Wreturn-local-addr]
```

Argumenty funkcji

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()`

```cpp
struct logger
{
    void set_file(std::filesystem::path p)
    {
        filename = p.filename().c_str();
    }
    const char * filename = nullptr;
};

int main(int ac, char* av[])
{
    std::filesystem::path p = "/path/to/file.txt";
    logger log;
    log.set_file(p);
    std::println("Filename: {}", log.filename);
    return 0;
}
```

```
Filename: AaaaAAAaA!!
```

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()`

```cpp
struct logger
{
    void set_file(std::filesystem::path p)
    {
        filename = p.filename().c_str();
    }
    std::string filename;
};

int main(int ac, char* av[])
{
    std::filesystem::path p = "/path/to/file.txt";
    logger log;
    log.set_file(p);
    std::println("Filename: {}", log.filename);
    return 0;
}
```

```
Filename: file.txt
```

Argumenty funkcji

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.

Argumenty funkcji

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

Odwijanie stosu

Wszystkie *zmienne automatyczne*, które znajdują się w zakresie między instrukcjami `throw` i `catch`, są niszczone w procesie znanym jako *odwijanie stosu* (ang. *stack unwinding*).

Niszczone?

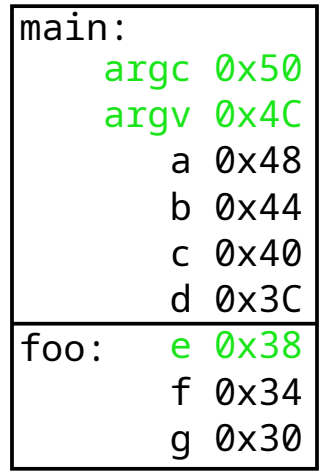
Czyli wykonanie **destruktora** dla każdej zmiennej automatycznej (o ile go posiada).

W zakresie?

Czyli umieszczonej na stosie pomiędzy miejscem rzucenia wyjątku, a strefą jego obsługi.

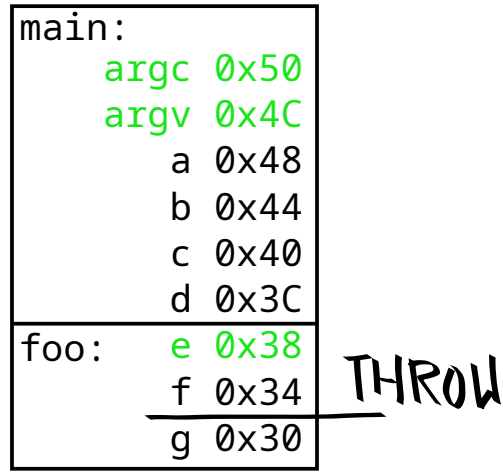
Blok try-catch

```
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 }
```



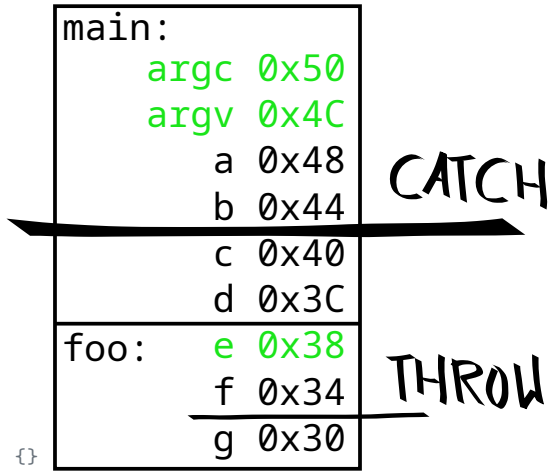
Blok try-catch

```
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 }
```



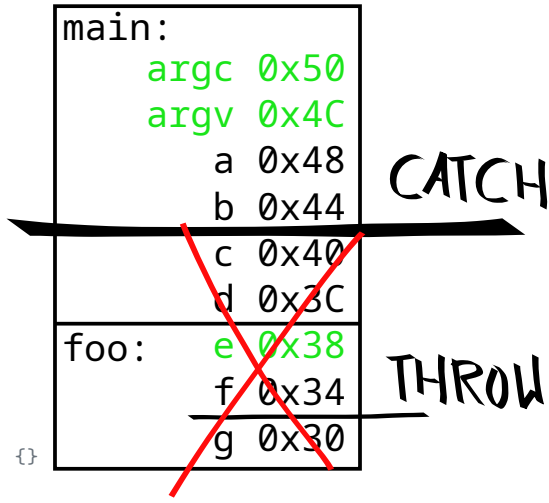
Blok try-catch

```
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 }
```



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 }
```



Strefy bezwyjątkowe

Strefa A: Destruktory (wszystkie!)

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

Strefy bezwyjątkowe

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*.

[1] zależy od kompilatora, GCC używa sterty.

Strefy bezwyjątkowe

Słowo kluczowe `noexcept`

Każdą funkcję można zadeklarować ze specyfikacją **`noexcept`**. Zastosowanie go informuje kompilator, że funkcja nie rzuci wyjątkiem na zewnątrz. Posłuży mu to do wygenerowania nieco szybszego/bardziej zwięzłego kodu, ale nieodpornego na wyjątki.

Strefa C: funkcje `noexcept`

Jeśli programista złamie dane kompilatorowi słowo i wyjątek zostanie rzucony z funkcji oznaczonej jako **`noexcept`**, wykonane zostanie *niezdefiniowane zachowanie*.

Strefy bezwyjątkowe

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.

Strefy bezwyjątkowe

Niezdefiniowane, czyli jakie?

Niezdefiniowane zachowanie (ang. *undefined behaviour*)

W praktyce jest to wykonanie funkcji `std::terminate`, która przerywa program w sposób daleki od zgrabnego (ang. *ungraceful termination*).
Zasoby są zwalniane do systemu, ale nie ma żadnych gwarancji co do opróżnienia buforów zapisu i zamknięcia otwartych deskryptorów plików.
Możliwa jest utrata danych albo ciężkie ich uszkodzenie.

Posiadanie zasobu (ang. *Resource Ownership*)

Właściciel zasobu

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

Posiadanie zasobu (ang. *Resource Ownership*)

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

Posiadanie zasobu (ang. *Resource Ownership*)

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

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

Posiadanie zasobu (ang. *Resource Ownership*)

To odśmiecacz potrzebuje `finally`

W niektórych językach właścicielem wszystkich obiektów jest *odśmiecacz* (ang. *garbage collector*).
Jest on odpowiedzialny za zwalnianie pamięci, ale nie zrobi 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 destruktor.

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

Resource Acquisition Is Initialization

RAII

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

RAII

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

Stos ma ograniczony rozmiar, łatwo go przepełnić!

RAII

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

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

Ależ możemy!

RAII

Resource Acquisition Is Initialization

Ta zagmatwana nazwa oznacza idiom zarządzania zasobami w sposób automatyczny.
Znana także pod akronimem **SBRM** - Scope-Bound Resource Management.

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    print_file(file);

    return 0;
}
```

A czy funkcja `print_file`:

- Sprawdza, czy plik istnieje?

Obsługa plików

```
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    print_file(file);

    return 0;
}
```

A czy funkcja `print_file`:

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

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    if (not file)
        return 1;
    try {
        print_file(file);
    } catch (...) {
        std::fclose(file);  // again?
        return 2;
    }
    return 0;
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy, czy plik się otworzył;

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    if (not file)
        return 1;
    try {
        print_file(file);
    } catch (...) {
        std::fclose(file);  // again?
        return 2;
    }
    return 0;
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy, czy plik się otworzył;
- zamykamy plik po odczytaniu;

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    if (not file)
        return 1;
    try {
        print_file(file);
    } catch (...) {
        std::fclose(file);  // again?
        return 2;
    }
    return 0;
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy, czy plik się otworzył;
- zamykamy plik po odczytaniu;
- umieszczamy wywołanie w bloku `try`;

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    if (not file)
        return 1;
    try {
        print_file(file);
    } catch (...) {
        std::fclose(file);  // again?
        return 2;
    }
    return 0;
}
```

Załóżmy, 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;

Obsługa plików

```cpp
#include <paro/File.h>

int main(int ac, char *av[])
{
    auto file = std::fopen(av[1], "r");
    if (not file)
        return 1;
    try {
        print_file(file);
    } catch (...) {
        std::fclose(file);  // again?
        return 2;
    }
    return 0;
}
```

Załóżmy, 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`.

Obsługa plików

```cpp
#include <fstream>
#include <iostream>

int main(int ac, char *av[])
{
    std::fstream file(av[1]);
    std::cout << file.rdbuf();

    return 0;
}
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia;

Obsługa plików

```
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;

Obsługa plików

```
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;

Obsługa plików

```
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;

Obsługa plików

```
#include <fstream>
#include <iostream>

int main(int ac, char *av[])
{
    std::fstream file(av[1]);
    std::cout << file.rdbuf();

    return 0;
}
```

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.

Resource Handler

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

```cpp
#include <paro/Resource.h>

class ResourceHandler
{
public:
    ResourceHandler() : resource(acquireResource()) {}
    ~ResourceHandler() { releaseResource(resource); }

    // copy forbidden!
    ResourceHandler(ResourceHandler const&) = delete;
    ResourceHandler& operator=(ResourceHandler const&) = delete;

    ResourceHandler(ResourceHandler&&) noexcept;
    ResourceHandler& operator=(ResourceHandler&&) noexcept;

private:
    Resource resource;
};
```

Resource Handler

Bez kopiowania?

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

Resource Handler

A co z przenoszeniem?

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

Resource Handler

```cpp
#include <utility>

ResourceHandler::ResourceHandler(ResourceHandler&& other) noexcept
    : resource(std::exchange(other.resource, 0x0))
{}

ResourceHandler& ResourceHandler::operator=(ResourceHandler&& other) noexcept
{
    resource = other.resource;
    other.resource = 0x0; // clear!
    return *this;
}
```

Resource Handler

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**.

Resource Handler

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]

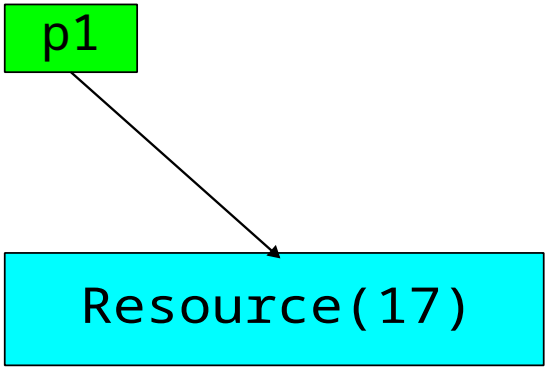
W bibliotece standardowej jest unikalny wskaźnik: `std::unique_ptr`.
Nie da się go kopiować — jest on jedynym *właścicielem* obiektu, na który wskazuje.
Przenoszenie pozbawia źródłowy obiekt wskaźnika na zasób i przypisuje go docelowemu.

std::unique_ptr

Sprytny?

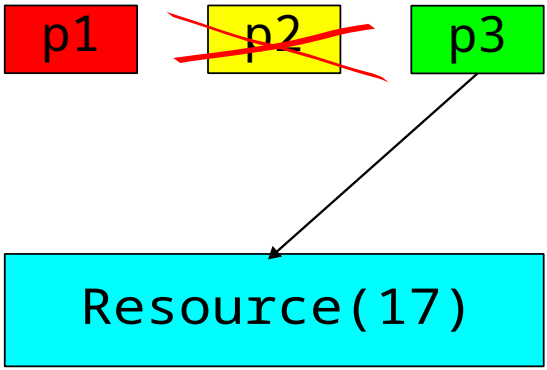
Spryt **std::unique_ptr** polega na wykorzystaniu RAII do *automatycznego* zwolnienia pamięci zaalokowanej *dynamicznie* — zapobiega to wyciekom pamięci.

std::unique_ptr




```cpp
#include <memory>

#include <paro/Resource.h>

int main(int ac, char *av[])
{
    std::unique_ptr<Resource> p1(new Resource(17));
    // std::unique_ptr<Resource> p2 = p1; // non-copyable!
    std::unique_ptr<Resource> p3 = std::move(p1);

    use_resource_the_intended_way(*p3);

    p3.reset(); // delete resource
    p1.reset(); // does nothing
    return 0;
}
```

std::unique_ptr




```cpp
#include <memory>

#include <paro/Resource.h>

int main(int ac, char *av[])
{
    std::unique_ptr<Resource> p1(new Resource(17));
    // std::unique_ptr<Resource> p2 = p1; // non-copyable!
    std::unique_ptr<Resource> p3 = std::move(p1);

    use_resource_the_intended_way(*p3);

    p3.reset(); // delete resource
    p1.reset(); // does nothing
    return 0;
}
```

std::unique_ptr

Alternatywna funkcja usuwająca - malloc i free

```
#include <cstdlib> // malloc/free
#include <memory>

struct Deleter
{
    void operator()(char* p) const noexcept { std::free(p); }
};
void perform_task(char* buffer, std::size_t buf_size);

int main(int ac, char* av[])
{
    constexpr std::size_t buf_size = 1024;
    std::unique_ptr<char, Deleter> buffer((char*)std::malloc(buf_size));
    if (buffer) {
        perform_task(buffer.get(), buf_size); // may throw!
    }
    return 0;
}
```

std::unique_ptr

Alternatywna funkcja usuwająca - fopen i fclose

```cpp
#include <cstdio> // fopen/fclose
#include <memory>

struct Deleter
{
    void operator()(std::FILE* file) const noexcept { std::fclose(file); }
};
void process_file(std::FILE* file);

int main(int ac, char* av[])
{
    constexpr auto path = "/path/to/file.txt";
    std::unique_ptr<std::FILE, Deleter> file(std::fopen(path, "r"));
    if (file) {
        process_file(file.get()); // may throw!
    }
    return 0;
}
```

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 stercie z jednoczesnym przypisaniem obiektu do `std::unique_ptr`.
Jest dostępna od C++14.

std::make_unique

```cpp
#include <memory>
#include <paro/Resource.h>

struct Deleter // std::default_delete<Resource>
{
    void operator()(Resource* p) const noexcept
    {
        delete p;
    }
};

int main(int ac, char* av[])
{
    std::unique_ptr<Resource, Deleter> res(new Resource(17));
    if (res) {
        use_resource_the_intended_way(*res);
    }
    return 0;
}
```

```cpp
#include <memory>
#include <paro/Resource.h>

int main(int ac, char* av[])
{
    auto res = std::make_unique<Resource>(17);
    if (res) {
        use_resource_the_intended_way(*res);
    }
    return 0;
}
```

std::make_unique

A jeśli utknęliśmy w 2011...

```cpp
#include <memory>

template <typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

std::make_unique

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
```

`std::move`

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*;

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*;
- kod jest odporny na niewłaściwe użycie.

std::move

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...

```cpp
#include <paro/Data.h>

std::unique_ptr<Data> load_initial_config();
void startup(std::unique_ptr<Data> const& init_cfg);

int main(int argc, const char* argv[])
{
    auto init_cfg = load_initial_config();
    startup(init_cfg); // init_cfg is NOT released

    for (EVER) {
        auto event = event_queue.front();
        process(event);
    }

    // many events later...
    return 0; // init_cfg is released here
}
```

std::move

std::move

```cpp
#include <paro/Data.h>

std::unique_ptr<Data> load_initial_config();
void startup(std::unique_ptr<Data> const& init_cfg);

int main(int argc, const char* argv[])
{
    auto init_cfg = load_initial_config();
    startup(init_cfg); // init_cfg is NOT released

    for (EVER) {
        auto event = event_queue.front();
        process(event);
    }

    // many events later...
    return 0; // init_cfg is released here
}
```

std::move

```cpp
#include <paro/Data.h>

std::unique_ptr<Data> load_initial_config();
void startup(std::unique_ptr<Data> init_cfg);

int main(int argc, const char* argv[])
{
    auto init_cfg = load_initial_config();
    startup(std::move(init_cfg)); // init_cfg is released after use

    for (EVER) {
        auto event = event_queue.front();
        process(event);
    }

    // many events later...
    return 0;
}
```

std::move

Nigdy nie powinniśmy używać referencji do sprytnych wskaźników.

Pobieranie argumentów funkcji przez referencje sprawia, że obiekty
nie są przenoszone do nowego zakresu.
Sprytne wskaźniki należy przekazywać przez wartość i przenosić między zakresami.
Zupełnie tak, jak zwykłe, surowe wskaźniki.

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 zwyky wskaźnik) nie zmienia czasu życia obiektu, na który wskazuje.

std::move

Możemy zarządzać pamięcią na stercie tak, jakby była na stosie

Używając `std::unique_ptr` otrzymujemy gwarancję, że dealokacja nastąpi niezależnie od sposobu, w jaki opusciliśmy 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 RAII?

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

Nie tylko `std::unique_ptr`

Gdzie jest wykorzystany RAII?

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

Nie tylko `std::unique_ptr`

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;

Nie tylko `std::unique_ptr`

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*);

Nie tylko `std::unique_ptr`

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 - dygresja o nazwie

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

```cpp
#include <pthread.h>
#include <stdio.h>

struct GameData
{
    GameData() : player_level(1) {}

    FILE* save;
    pthread_mutex_t the_mutex;
    int player_level;
};

int main(int argc, char* argv[])
{
    GameData data; // initialization, then
    // stepwise resource acquisition
    if (0 != pthread_mutex_init(&data.the_mutex, NULL))
        return 2;
    data.save = fopen("save.dat", "r");
    if (NULL == data.save)
        return 1;
    return 0;
}
```

RAII - dygresja o nazwie

```c
#include <pthread.h>
#include <stdio.h>

struct GameData
{
    GameData() : player_level(1) {}

    FILE* save;
    pthread_mutex_t the_mutex;
    int player_level;
};

int main(int argc, char* argv[])
{
    GameData data; // initialization, then
    // stepwise resource acquisition
    if (0 != pthread_mutex_init(&data.the_mutex, NULL))
        return 2;
    data.save = fopen("save.dat", "r");
    if (NULL == data.save)
        return 1;
    return 0;
}
```

```c
#include <fstream>
#include <mutex>

struct GameData
{
    GameData() : save("save.dat"), player_level(1) {}

    std::fstream save;
    std::mutex the_mutex;
    int player_level;
};

int main(int argc, char* argv[])
{
    // resources are acquired during initialization
    GameData data;

    return 0;
}
```

Shared Resource

Shared Resource

Nie zawsze jednak możemy przydzielić odpowiedzialność za zasób na wyłączność.

Na szczęście tu również mamy pełne wsparcie ze strony biblioteki standardowej.

std::shared_ptr

std::shared_ptr

W sytuacji, gdy nie jesteśmy w stanie określić pojedynczego właściciela obiektu, a nadal chcemy korzystać ze zautomatyzowanej smart pointerów, na odsiecz przychodzi nam `std::shared_ptr`.
Można go kopiować, tworząc w ten sposób kolejne wskaźniki na ten sam obiekt w pamięci.
Destruktor zostanie zawołany tylko raz, gdy ostatni skojarzony `std::shared_ptr` przestanie istnieć.

std::shared_ptr

```cpp
#include <memory>

#include <paro/Resource.h>

int main(int ac, char *av[])
{
    std::shared_ptr<Resource> p1(new Resource(17));
    std::shared_ptr<Resource> p2 = p1; // copying is available now!
    std::shared_ptr<Resource> p3 = std::move(p1); // still possible

    use_resource_the_intended_way(*p3);
    return 0;  // p3, p2, p1 went out-of-scope, object is destroyed
}
```

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?

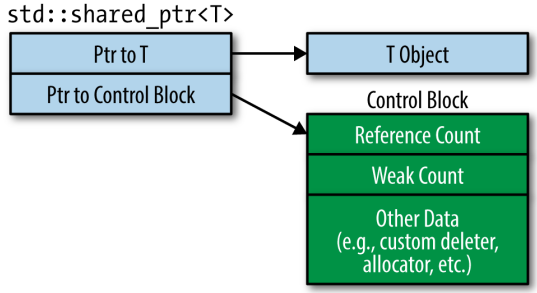
Służy mu do tego mechanizm zliczania referencji.

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 destruktor obiektu.

std::shared_ptr

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

- na zarządzany obiekt
- na blok kontrolny



std::shared_ptr

Niestandardowe deletery

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

#include <paro/Resource.h>

int main(int ac, char* av[])
{
    auto deleterOdd = [](Resource* r) { delete r; }; // custom deleter
    auto deleterEven = [](Resource* r) { delete r; }; // different types

    std::shared_ptr<Resource> p1(new Resource(13), deleterOdd);
    std::shared_ptr<Resource> p2(new Resource(26), deleterEven);
    // impossible with std::unique_ptr!
    std::vector<std::shared_ptr<Resource>> vec{p1, p2};
    return 0;
}
```

std::shared_ptr

Wielowątkowość

- Dostęp do zasobu możemy współdzielić między wątkami.

std::shared_ptr

Wielowątkowość

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

std::shared_ptr

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.

std::shared_ptr

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.

std::shared_ptr

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!

std::shared_ptr

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.

std::make_shared

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

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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`;

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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;

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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;

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::shared_ptr<Resource>(new Resource(17)), nastyFunction(17));
    return 0;
}
```

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

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::make_shared<Resource>(17), nastyFunction(17));
    return 0;
}
```

`std::make_shared` na ratunek!*

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::make_shared<Resource>(17), nastyFunction(17));
    return 0;
}
```

`std::make_shared` na ratunek!*

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

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::make_shared<Resource>(17), nastyFunction(17));
    return 0;
}
```

`std::make_shared` na ratunek!*

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

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::make_shared<Resource>(17), nastyFunction(17));
    return 0;
}
```

`std::make_shared` na ratunek!*

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

std::make_shared

```cpp
#include <paro/Resource.h>
#include <memory>

int nastyFunction(int param)
{
    if (param % 2)
        throw std::runtime_error("Oops!");
    return 42;
}

void foo(std::shared_ptr<Resource> res, int i) {}

int main(int argc, char const* argv[])
{
    foo(std::make_shared<Resource>(17), nastyFunction(17));
    return 0;
}
```

`std::make_shared` na ratunek!*

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

std::make_shared

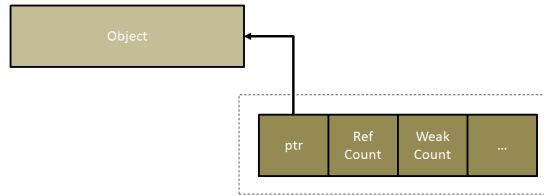
std::make_shared na ratunek!*

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

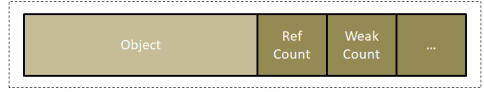
Dwie osobne alokacje: dla obiektu oraz dla bloku kontrolnego.
Możliwe zwolnienie pamięci po obiekcie do systemu operacyjnego po zawołaniu destruktora.



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::make_shared`

Wady i zalety `std::make_shared`

Zalety	Wady
Wspólna alokacja dla obiektu oraz bloku kontrolnego	

std::make_shared

Wady i zalety std::make_shared

Zalety	Wady
Wspólna alokacja dla obiektu oraz bloku kontrolnego	
Niewrażliwy na kolejność wykonania argumentów funkcji	

std::make_shared

Wady i zalety std::make_shared

Zalety	Wady
Wspólna alokacja dla obiektu oraz bloku kontrolnego	
Niewrażliwy na kolejność wykonania argumentów funkcji	
Pomaga z fragmentacją pamięci	

std::make_shared

Wady i zalety std::make_shared

Zalety	Wady
Wspólna alokacja dla obiektu oraz bloku kontrolnego	
Niewrażliwy na kolejność wykonania argumentów funkcji	Nie da się przekazać niestandardowego deletera
Pomaga z fragmentacją pamięci	

std::make_shared

Wady i zalety std::make_shared

Zalety	Wady
Wspólna alokacja dla obiektu oraz bloku kontrolnego	Wspólna alokacja dla obiektu oraz bloku kontrolnego
Niewrażliwy na kolejność wykonania argumentów funkcji	Nie da się przekazać niestandardowego deletera
Pomaga z fragmentacją pamięci	

std::weak_ptr

std::weak_ptr to wskaźnik-obserwator.

- 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

```cpp
void check(std::weak_ptr<int> w)
{
    std::cout << "use_count␣=␣" << w.use_count() << '\n';
    // try to upgrade to std::shared_ptr, better than just checking .expired()
    if (auto stillAlive = w.lock()) {
        std::cout << "I␣am␣here!\n";
    } else {
        std::cout << "too␣late...\n";
    }
}

int main(int ac, char *av[])
{
    std::weak_ptr<int> weak;
    {
        auto shared = std::make_shared<int>(17);
        weak = shared;
        check(weak);
    }
    check(weak);
    return 0;
}
```

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

std::weak_ptr

```cpp
struct B;
struct A
{
    std::shared_ptr<B> b;
    ~A() { std::cout << "destructing␣A\n"; }
};

struct B
{
    std::shared_ptr<A> a;
    ~B() { std::cout << "destructing␣B\n"; }
};

void useBoth()
{
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->b = b;
    b->a = a;
}
```

```cpp
int main()
{
    useBoth();
    std::cout << "Finished␣using␣A␣and␣B\n";
}
```

```
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

Rule of Zero

It's broken!

```cpp
#include <cstddef>
#include <cstring>

struct Nickname
{
    Nickname(const char* input)
    {
        if (input) {
            std::size_t n = std::strlen(input) + 1;
            nick_ = new char[n];
            std::memcpy(nick_, input, n);
        }
    }

    ~Nickname() { delete[] nick_; }

private:
    char* nick_ = nullptr;
};
```

```cpp
int main()
{
    Nickname n1("Whatever");
    // copy constructor implicitly generated by compiler
    Nickname n2(n1);
    // same for copy assignment operator
    Nickname n3 = n2;
} // n3, n2, n1 go out-of-scope; destructors called;
  // double free and crash
```

Rule of Zero

It's still broken!

```cpp
#include <cstddef>
#include <cstring>

struct Nickname
{
    Nickname(const char* input)
    {
        if (input) {
            std::size_t n = std::strlen(input) + 1;
            nick_ = new char[n];
            std::memcpy(nick_, input, n);
        }
    }

    ~Nickname() { if (nick_) delete[] nick_; }

private:
    char* nick_ = nullptr;
};
```

```cpp
int main()
{
    Nickname n1("Whatever");
    // copy constructor implicitly generated by compiler
    Nickname n2(n1);
    // same for copy assignment operator
    Nickname n3 = n2;
} // n3, n2, n1 go out-of-scope; destructors called;
  // double free and crash
```

Rule of Zero

Unbelieveable - still broken!

```cpp
struct Nickname
{
    Nickname(const char* input)
    {
        if (input) {
            std::size_t n = std::strlen(input) + 1;
            nick_ = new char[n];
            std::memcpy(nick_, input, n);
        }
    }
    Nickname(const Nickname& other) {
        if (other.nick_) {
            std::size_t n = std::strlen(other.nick_) + 1;
            nick_ = new char[n];
            std::memcpy(nick_, other.nick_, n);
        }
    }
    ~Nickname() { if (nick_) delete[] nick_; }

private:
    char* nick_ = nullptr;
};
```

```cpp
int main()
{
    Nickname n1("Whatever");
    // copy constructor written by hand
    Nickname n2(n1);
    // copy assignment generated by compiler
    Nickname n3 = n2;
} // n3, n2, n1 go out-of-scope; destructors called;
  // double free and crash
```

Rule of Zero

It... works? (still ugly)

```
Nickname(const Nickname& other) {
    if (other.nick_) {
        std::size_t n = std::strlen(other.nick_) + 1;
        nick_ = new char[n];
        std::memcpy(nick_, other.nick_, n);
    }
}
Nickname& operator=(const Nickname& other) {
    if (other.nick_) {
        std::size_t n = std::strlen(other.nick_) + 1;
        nick_ = new char[n];
        std::memcpy(nick_, other.nick_, n);
    }
    return *this;
}
~Nickname() { if (nick_) delete[] nick_; }
```

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

Rule of Zero

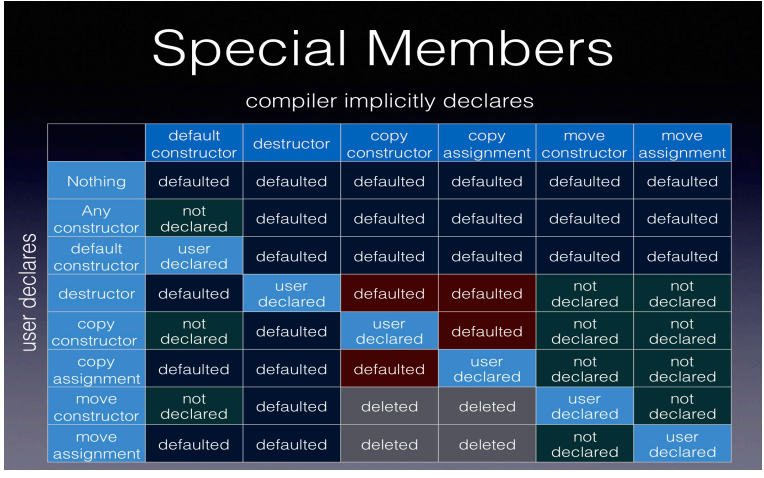
Rule of three

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

- destruktora,
- 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.

Rule of Zero


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

```
Nickname(const Nickname& other)
    : Nickname(other.nick_) {}

Nickname& operator=(const Nickname& other)
{
    if (this == &other)
        return *this; // check self-assignment!

    std::size_t n = std::strlen(other.nick_) + 1;
    char* new_nick = new char[n];
    std::memcpy(new_nick, other.nick_, n);
    delete[] nick_;

    nick_ = new_nick;
    return *this;
}
```

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

Rule of Zero

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

```
    Nickname(Nickname&& other) noexcept
        : nick_(std::exchange(other.nick_, nullptr))
    {}

    Nickname& operator=(Nickname&& other) noexcept
    {
        std::swap(nick_, other.nick_);
        return *this;
    }
```

```
int main()
{
    Nickname n1("Whatever");
    // copy constructor is defined by us now
    Nickname n2(n1);
    // same for copy assignment operator
    Nickname n3 = n2;
    // ditto for move
    Nickname n4 = std::move(n3);
} // n4, n3, n2, n1 go out-of-scope; destructors called;
  // everything works correctly
```

Rule of Zero

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 RAII oraz klas implementujących ten wzorzec.

Rule of zero

```cpp
#include <string>

struct Nickname
{
    explicit Nickname(std::string input) : nick_(std::move(input)) {}
private:
    std::string nick_; // std::string is a RAII-wrapper around char*
};

int main(int ac, char* av[])
{
    Nickname n1("Whatever");
    // copy constructor is implicitly generated by compiler
    Nickname n2(n1);
    // same for copy assignment operator
    Nickname n3 = n2;
    // same for move constructor
    Nickname n4 = std::move(n2);
} // n4, n3, n2, n1 go out-of-scope - destructors called
  // everything works correctly automagically thanks to RAII
```

Wydajność

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.

Wydajność

T*

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

struct Data
{
    char tab_[42];
};

int main(int ac, char *av[])
{
    constexpr unsigned size = 10'000'000u;
    std::vector<Data*> vec;
    vec.reserve(size);
    for (auto i = 0u; i < size; i++) {
        auto p = new Data();
        vec.push_back(std::move(p));
    }
    for (auto p : vec) delete p;
}
```

Wydajność

std::unique_ptr<T>

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

struct Data
{
    char tab_[42];
};

int main(int ac, char *av[])
{
    constexpr unsigned size = 10'000'000u;
    std::vector<std::unique_ptr<Data>> vec;
    vec.reserve(size);
    for (auto i = 0u; i < size; i++) {
        std::unique_ptr<Data> p{new Data()};
        vec.push_back(std::move(p));
    }
}
```

Wydajność

std::shared_ptr<T>

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

struct Data
{
    char tab_[42];
};

int main(int ac, char *av[])
{
    constexpr unsigned size = 10'000'000u;
    std::vector<std::shared_ptr<Data>> vec;
    vec.reserve(size);
    for (auto i = 0u; i < size; i++) {
        std::shared_ptr<Data> p{new Data()};
        vec.push_back(std::move(p));
    }
}
```

Wydajność

std::weak_ptr<T>

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

struct Data
{
    char tab_[42];
};

int main(int ac, char *av[])
{
    constexpr unsigned size = 10'000'000u;
    std::vector<std::shared_ptr<Data>> vec;
    std::vector<std::weak_ptr<Data>> vec_observers;
    vec.reserve(size);
    vec_observers.reserve(size);
    for (auto i = 0u; i < size; i++) {
        std::shared_ptr<Data> p{new Data()};
        std::weak_ptr<Data> weak{p};
        vec.push_back(std::move(p));
        vec_observers.push_back(std::move(weak));
    }
}
```

Wydajność

std::make_shared<T>

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

struct Data
{
    char tab_[42];
};

int main(int ac, char *av[])
{
    constexpr unsigned size = 10'000'000u;
    std::vector<std::shared_ptr<Data>> vec;
    vec.reserve(size);
    for (auto i = 0u; i < size; i++) {
        auto p = std::make_shared<Data>();
        vec.push_back(std::move(p));
    }
}
```

Wydajność

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

nazwa testu	czas [s]	alokacje	pamięć [MB]

Wydajność

- 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

Wydajność

- 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

Wydajność

- 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

Wydajność

- 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

Wydajność

- 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	`2024-11-24T23:59+01:00`


NOKIA

Algorytmy i Struktury Danych

Memory Management API

Struktura danych: kopiec

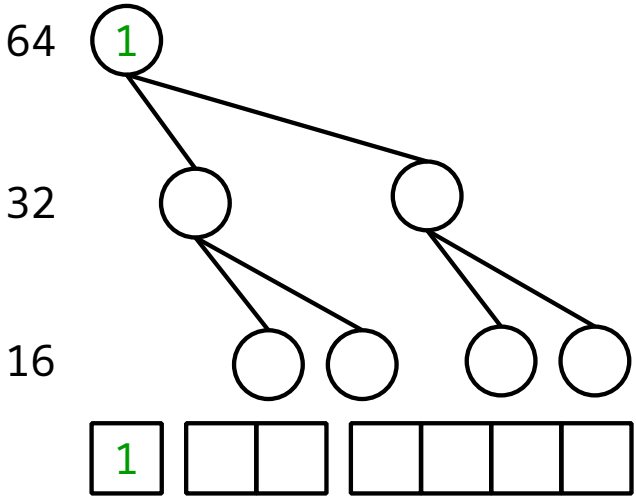
Kopiec - drzewo binarne na tablicy

Własność kopca: węzły potomne pozycji N znajdują się na pozycjach $2N$ i $2N+1$. Taka tablica reprezentuje drzewo binarne *prawie pełne*.

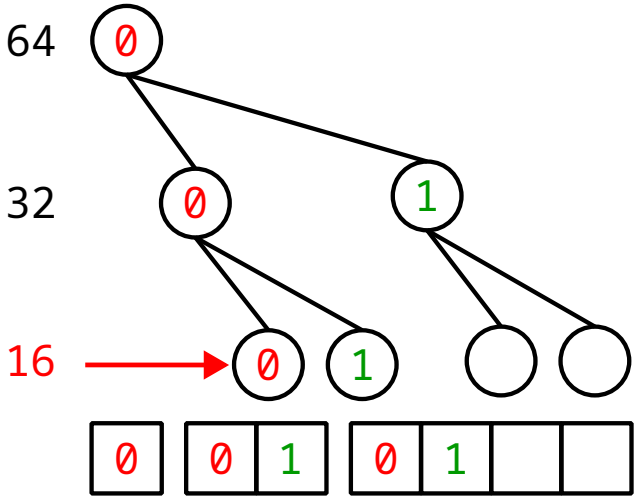
Rodzeństwo (siblings)

Para sąsiednich pozycji parzysty-nieparzysty (np 2-3, 4-5; ale nie 1-2, 3-4) to bliźniaki — reprezentują bloki pamięci, które mogą zostać połączone, jeśli tylko oba są wolne. Mają wspolnego rodzica — są podblokami bloku bedącego ich sumą.

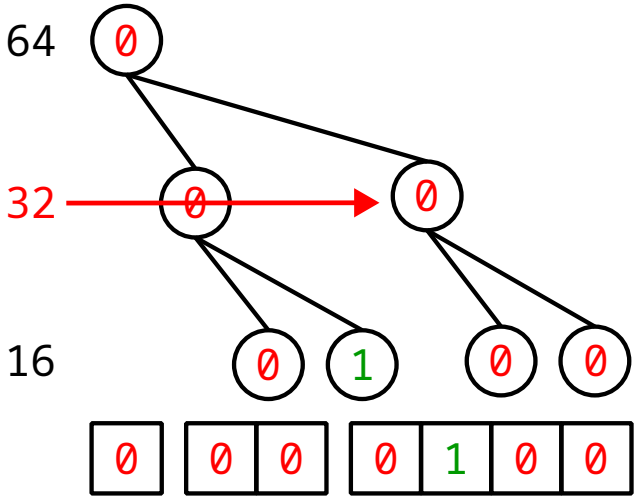
Struktura danych: kopiec



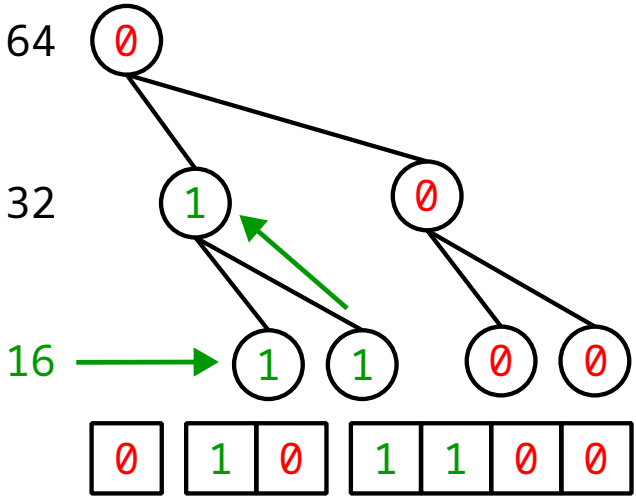
Struktura danych: kopiec



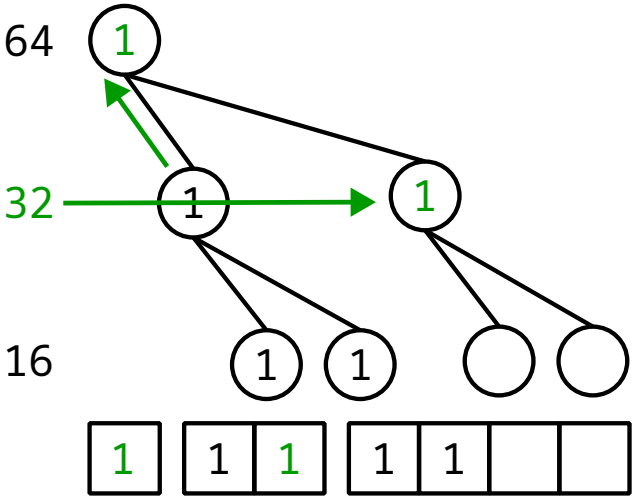
Struktura danych: kopiec



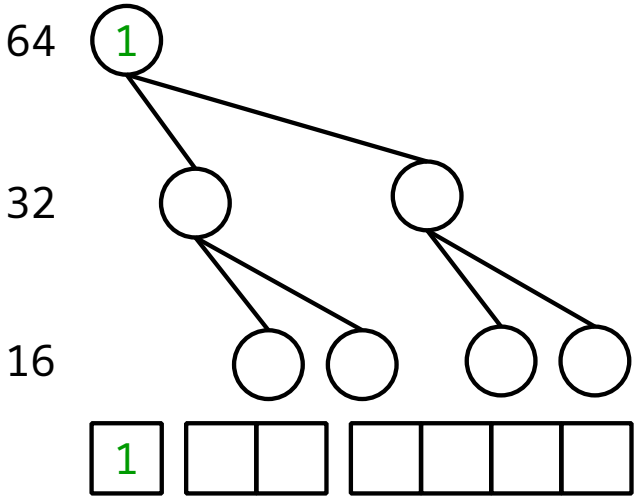
Struktura danych: kopiec



Struktura danych: kopiec



Struktura danych: kopiec



Struktura danych: stos

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.

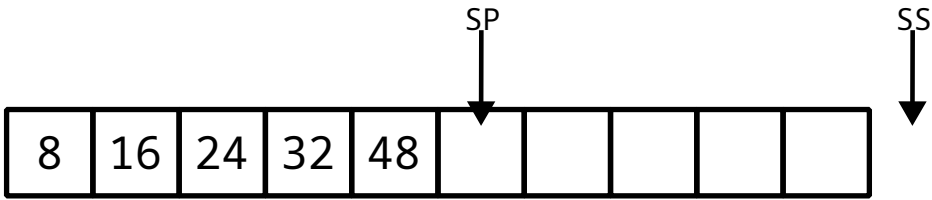
Struktura danych: stos

Rozmiar stosu jest stały

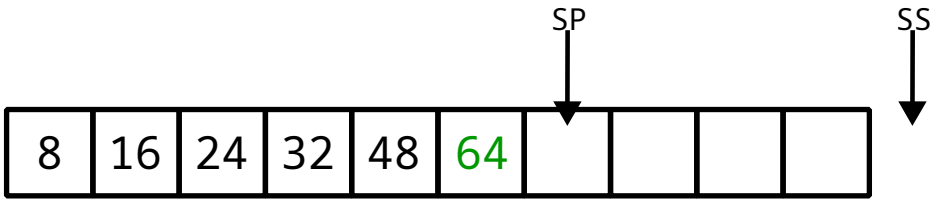
Stos wywołań funkcji jest stałego rozmiaru - alokowany raz przy uruchomieniu programu. Z racji wykorzystania prostego wskaźnika stosu, musi to być obszar ciągły. Realokowanie stosu zdezaktualizowałoby wszystkie wskaźniki na lokalne zmienne. Z ustalonego rozmiaru wynika niebezpieczeństwo przepełnienia stosu (ang. *stack overflow*).

Struktura danych: stos

```
push(64)
```

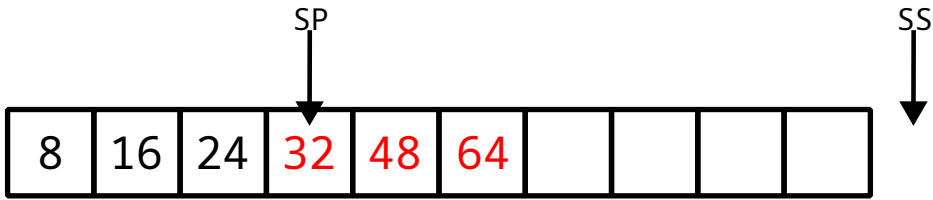


Struktura danych: stos



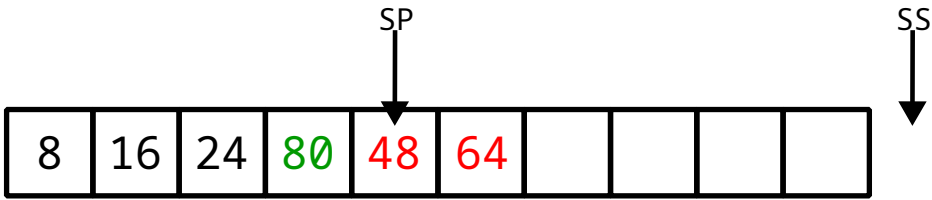
Struktura danych: stos

pop() 64
pop() 48
pop() 32



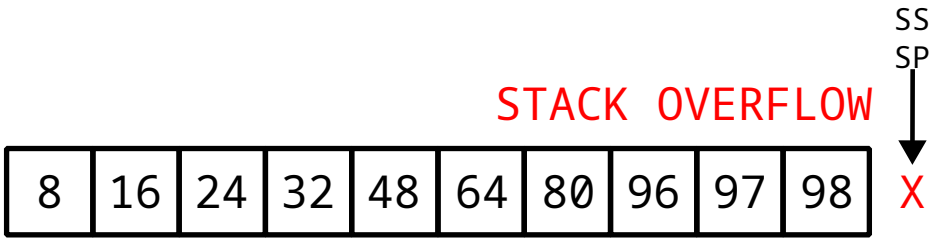
Struktura danych: stos

```
push(80)
```



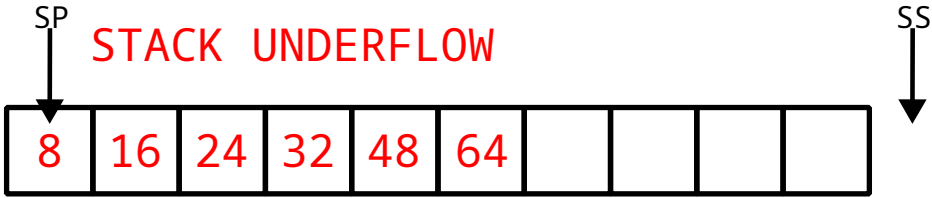
Struktura danych: stos

```
push(99)
```



Struktura danych: stos

pop()



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`**

Interfejs programistyczny podsystemu pamięci

Poziom 3: język C++

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 destruktor obiektu, a następnie funkcję **`free`** co zwalnia blok pamięci.

Interfejs programistyczny podsystemu 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*.