PARO - Wrocław University

C++ Memory Management

Bogumił Chojnowski
Paweł Krysiak

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

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Alokacja zasobów

Co to jest alokacja zasobów?

Alokacja - przypisywanie zasobów do możliwości ich użycia.

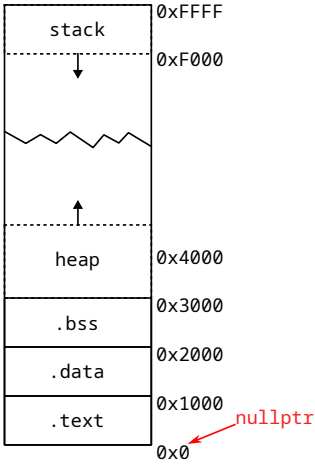
Alokacja zasobów

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` niezainicjalizowane globalne

`heap` sterta

`stack` stos - rośnie w dół

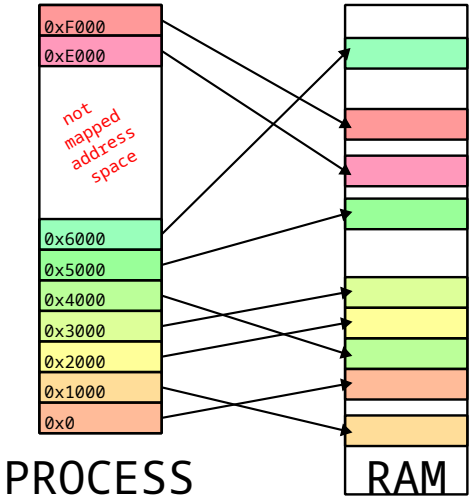
`nullptr` - on też tu jest!

Przestrzeń adresowa

Przestrzeń adresowa $\neq$ zaalokowana pamięć

Proces ma do dyspozycji *ciągłą* przestrzeń adresową. Ale nie oznacza to, że system rezerwuje dla każdej uruchomionej aplikacji po 4 GiB (32-bit), lub więcej (64-bit), RAMu.

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

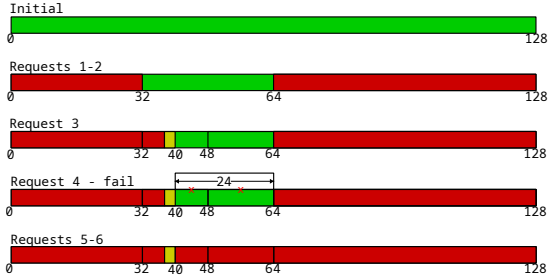
Z racji częstotliwości używania, sumaryczny koszt wykonania wpływa znacząco na szybkość działania programu.

Chcemy, żeby był dokładny

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

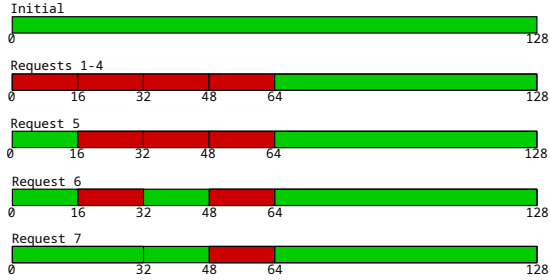
Algorytm bloków bliźniaczych

```
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 }
```



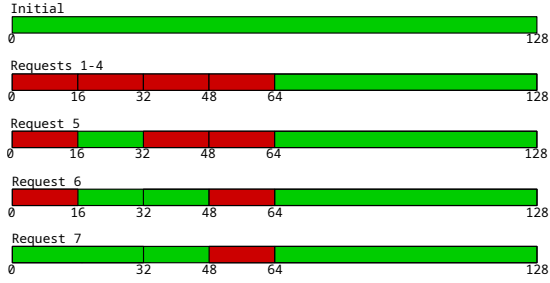
Algorytm bloków bliźniaczych

```
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 }
```



Algorytm bloków bliźniaczych

`memory-management/research/buddy_allocation`

Zadanie z gwiazdką

Tak ulepszyć implementację, aby informacja o zajętości bloków trzymane były w tablicy o ustalonym rozmiarze.

Algorytm bloków bliźniaczych

Jak dobrze nam poszło?

- dość 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 pamięcią przydzielaną procesom.
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**

Fragmentacja pamięci

Fragmentacja wewnętrzna

Wynika z wyrównywania rozmiaru przydzielanego bloku do odgórnie ustalonego rozmiaru - przydzielony blok jest większy niż dane w nim umieszczone, naddatek się marnuje.

Fragmentacja zewnętrzna

Z każdą alokacją i dealokacją związane jest ryzyko powstania "dziury" w zaalokowanej 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  #include <iostream>
2
3  int global_variable = 11;
4
5  int main(int ac, char* av[])
6  {
7      static int static_variable = 22;
8      int automatic_variable = 33;
9      int* heap_variable = new int(44);
10     delete heap_variable;
11
12     return 0;
13 }
```

Rodzaje alokacji

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

```
1  #include <iostream>
2
3  int global_variable = 11;
4
5  int main(int ac, char* av[])
6  {
7      static int static_variable = 22;
8      int automatic_variable = 33;
9      int* heap_variable = new int(44);
10     delete heap_variable;
11
12     return 0;
13 }
```

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 wykonuje konstruktor
- `delete` - wykonuje destruktor i zwalnia pamięć.

```
1  #include "Object.hpp"
2
3  int main(int ac, char* av[])
4  {
5      Object* o = new Object(3, 14, 92);
6      delete o;
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

```cpp
#include <string>

constexpr int arr_size = 40;

int main(int ac, char* av[])
{
    std::string* arr = new std::string[arr_size];
    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.

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?

- parametry wywołania funkcji
- wszystkie lokalne zmienne
- ...a co z typem zwracanym?
  - miejsce na wartość zwracaną jest zarezerwowane na stosie jeszcze przed wykonaniem.

Alokacja automatyczna

```
1  #include "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 z użyciem konstruktora o 3 parametrach

Alokacja automatyczna

```
1  #include "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 z użyciem konstruktora o 3 parametrach
- kompilator zadba o to, by destruktor obiektu został wykonany przy opuszczaniu funkcji

Alokacja automatyczna

```
1  #include "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 z użyciem konstruktora o 3 parametrach
- kompilator zadba o to, by destruktor obiektu został wykonany przy opuszczaniu funkcji
- obiekt jest automatyczny

Alokacja automatyczna

A co z globalami?

Zmienne globalne i statyczne funkcji nie są przechowywane na stosie, a 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 — czyli obszar w programie, w którym można z nich korzystać. Odwołanie się do pamięci spoza zakresu powoduje wykonanie *niezdefiniowanego zachowania*.

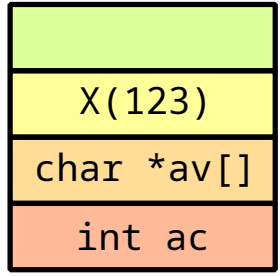
Zakres zmiennych

```cpp
#include <fmt/format.h>

struct X
{
    X(int x) : x(x) { fmt::print("X({}) created\n", x); }
    ~X() { fmt::print("X({}) is dead\n", 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
```

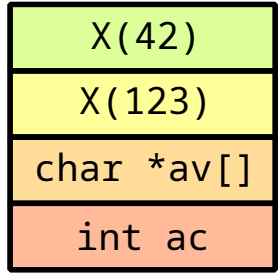
Zakres zmiennych

```
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 }
```



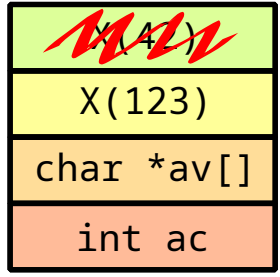
Zakres zmiennych

```
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 }
```



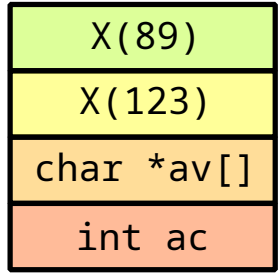
Zakres zmiennych

```
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 }
```



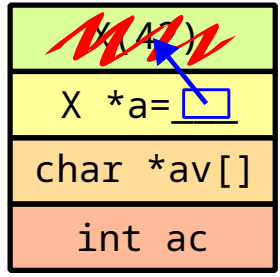
Zakres zmiennych

```
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;
7      }
8      a->x = 8; // whoops!
9
10     return 0;
```



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 w odpowiednim miejscu: wewnątrz bloku, oraz z odpowiednim zestawem kwalifikatorów

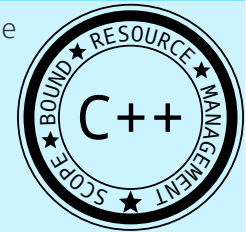
1. nie `const/constexpr`, bo kompilator ma prawo je *wyoptymalizować*
2. nie `static`, bo takie są alokowane razem z obiektami globalnymi

Zakres zmiennych

Gwarancja na obiekty automatyczne

Dla obiektów automatycznych otrzymujemy dwie 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

Parametry funkcji

Funkcja definiuje lokalny zakres

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

Parametry funkcji

```
1  #include <string>
2
3  std::string& foo(std::string s) { return s; }
4
5  int main(int ac, char* av[])
6  {
7      std::string& x = foo("text");
8      x.front();
9      return 0;
10 }
```

Parametry funkcji

```cpp
#include <string>

std::string& foo(std::string s) { return s; }

int main(int ac, char* av[])
{
    std::string& x = foo("text");
    x.front();
    return 0;
}
```

```
function_scope.cpp:3:42: warning: reference to local variable ‘s’
returned [-Wreturn-local-addr]
```

Parametry funkcji

```cpp
#include <string>

std::string& foo(std::string s) { return s; }

int main(int ac, char* av[])
{
    std::string& x = foo("text");
    x.front();
    return 0;
}
```

```
Segmentation fault.
```

Parametry funkcji

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

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

Parametry funkcji

```cpp
#include <fmt/format.h>

struct X
{
    X(int x) : x(x) { fmt::print("X({}) created\n", x); }
    ~X() { fmt::print("X({}) is dead\n", x); }

    int x;
};

void foo(X x) { x.x = 8; }

int main(int ac, char* av[])
{
    foo(X{3});
    fmt::print("In main again\n");
    return 0;
}
```

```
X(3) created
X(8) is dead
In main again
```

Wyjście z funkcji

Po wykonaniu ostatniej instrukcji

```
1  void foo(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 bar(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 baz(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 baz_value = obj->method();
7      delete obj;
8      return baz_value;
9  }
```

Blok try-catch

Odwijanie stosu

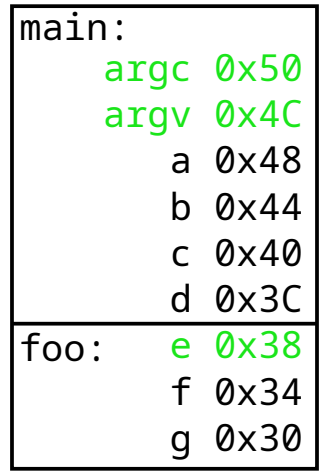
Wszystkie *zmienne automatyczne*, które znajdują się w zakresie między instrukcjami `throw` i `catch`, są niszczone w procesie znanym jako *odwijanie stosu* (ang. *stack unwinding*).

Niszczone?

Czyli wykonanie destruktora dla każdej zmiennej automatycznej (o ile jej typ takowy posiada) umieszczonej na stosie pomiędzy rzuceniem wyjątku, a strefą jego obsługi.

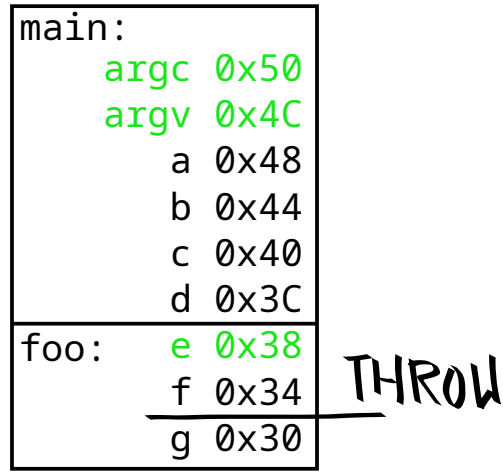
Blok try-catch

```
1  void foo(int e)
2  {
3      int f;
4      throw std::runtime_error("whoops!");
5      int g;
6  }
7
8  int main(int argc, char* argv[])
9  {
10     int a, b;
11     try {
12         int c, d;
13         foo(10);
14     } catch (std::runtime_error const& ex) {}
15 }
```



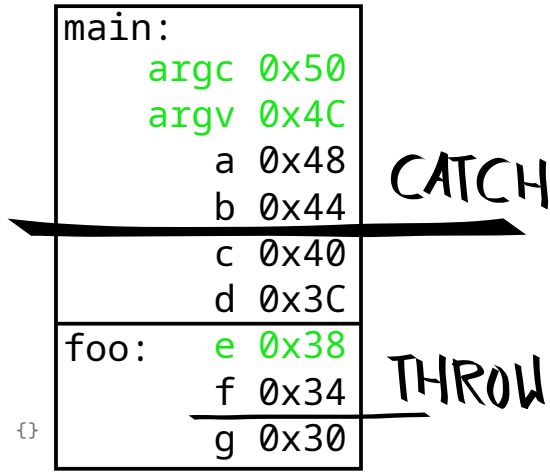
Blok try-catch

```
1  void foo(int e)
2  {
3      int f;
4      throw std::runtime_error("whoops!");
5      int g;
6  }
7
8  int main(int argc, char* argv[])
9  {
10     int a, b;
11     try {
12         int c, d;
13         foo(10);
14     } catch (std::runtime_error const& ex) {}
15 }
```



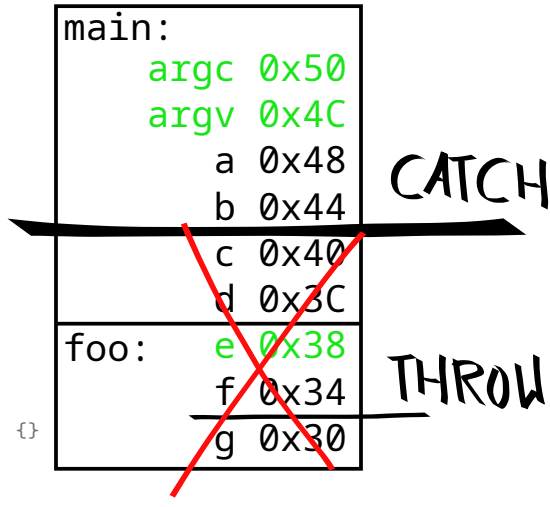
Blok try-catch

```
1  void foo(int e)
2  {
3      int f;
4      throw std::runtime_error("whoops!");
5      int g;
6  }
7
8  int main(int argc, char* argv[])
9  {
10     int a, b;
11     try {
12         int c, d;
13         foo(10);
14     } catch (std::runtime_error const& ex) {}
15 }
```



Blok try-catch

```
1  void foo(int e)
2  {
3      int f;
4      throw std::runtime_error("whoops!");
5      int g;
6  }
7
8  int main(int argc, char* argv[])
9  {
10     int a, b;
11     try {
12         int c, d;
13         foo(10);
14     } catch (std::runtime_error const& ex) {}
15 }
```



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.
Dzieje się to już po wystąpieniu `throw`, a przed rozpoczęciem obsługi wyjątku w bloku `catch` - rzucenie kolejnego prowadzi do *niezdefiniowanego zachowania*.

Strefy bezwyjątkowe

Strefa C: funkcje `noexcept`

Każdą funkcję możemy też ręcznie oznaczyć słowem kluczowym **`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 sytuacje wyjątkowe.
Jeśli programista złamie dane kompilatorowi słowo i wyjątek zostanie wyrzucony z funkcji oznaczonej jako **`noexcept`**, wykonane zostanie *niezdefiniowane zachowanie*.

Strefy bezwyjątkowe

Trzeba złapać je wszystkie!

We wszystkich strefach bezwyjątkowych obowiązuje zasada *Pokémon - gotta catch'em all*.
Można wołać funkcje rzucające wyjątki - należy tylko 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, z kolapsem naszej 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.
Jak najbardziej możliwa jest utrata danych lub ich ciężkie uszkodzenie.

Posiadanie zasobu (ang. *Resource Ownership*)

Właściciel zasobu

Właścicielem nazywamy obiekt odpowiedzialny za jego zwolnienie. 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 *garbage collector* - to on jest odpowiedzialny za zwalnianie pamięci, ale nie zrobi nic ponadto - usunie tylko obiekt z RAMu. Wszelkie zasoby dodatkowe (np. otwarte pliki) pozostaną niesfinalizowane.

Przewidujący wystąpienie wyjątkowyh sytuacji programista jest zobowiązany do tego, aby napisać kod finalizera **w każdym bloku obsługi wyjątków.**

Język C++ pozwala na automatyzację tego zadania przez destruktor - można przypisać do obiektu jego kod finalizujący. 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

RAII - wprowadzenie

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

RAII - wprowadzenie

Stos zachowuje się porządnie, to po co dynamiczna alokacja?

Stos ma ograniczony rozmiar, łatwo go przepełnić!

RAII - wprowadzenie

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

RAII - wprowadzenie

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

Ależ możemy!

RAII - wprowadzenie

Resource Acquisition Is Initialization

Tą zagmatwaną poniekąd nazwą określamy idiom zarządzania zasobami w sposób automatyczny.
Znana także pod akronimem **SBRM** - Scope-Bound Resource Management.

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    print(file);
}
```

A czy funkcja `print`:

- sprawdza czy plik istnieje?

Obsługa plików

```
1  #include <cstdio>
2
3  void print_file(const char* path)
4  {
5      FILE* file = fopen(path, "r");
6      print(file);
7  }
```

A czy funkcja `print`:

- sprawdza czy plik istnieje?
- zamyka plik po odczytaniu?

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    if (not file)
        return;
    try {
        print(file);
        fclose(file);
    } catch (...) {
        fclose(file); // again?
        throw;
    }
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy czy plik się otworzył

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    if (not file)
        return;
    try {
        print(file);
        fclose(file);
    } catch (...) {
        fclose(file); // again?
        throw;
    }
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy czy plik się otworzył
- zamykamy plik po odczytaniu

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    if (not file)
        return;
    try {
        print(file);
        fclose(file);
    } catch (...) {
        fclose(file); // again?
        throw;
    }
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy czy plik się otworzył
- zamykamy plik po odczytaniu
- umieszczamy wszystko w bloku `try`

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    if (not file)
        return;
    try {
        print(file);
        fclose(file);
    } catch (...) {
        fclose(file); // again?
        throw;
    }
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy czy plik się otworzył
- zamykamy plik po odczytaniu
- umieszczamy wszystko w bloku `try`
- a w `catch` znowu zamykamy plik...

Obsługa plików

```cpp
#include <cstdio>

void print_file(const char* path)
{
    FILE* file = fopen(path, "r");
    if (not file)
        return;
    try {
        print(file);
        fclose(file);
    } catch (...) {
        fclose(file); // again?
        throw;
    }
}
```

Załóżmy, że nie robi żadnego z powyższych, wtedy:

- sprawdzamy czy plik się otworzył
- zamykamy plik po odczytaniu
- umieszczamy wszystko w bloku `try`
- a w `catch` znowu zamykamy plik...
- ...tu by się przydało `finally`

Obsługa plików

```
1  #include <fstream>
2  #include <iostream>
3
4  void print_file(const char* path)
5  {
6      std::fstream file(path);
7      std::cout << file.rdbuf();
8  }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

- opakowujemy plik w obiekt strumienia

Obsługa plików

```
1  #include <fstream>
2  #include <iostream>
3
4  void print_file(const char* path)
5  {
6      std::fstream file(path);
7      std::cout << file.rdbuf();
8  }
```

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  void print_file(const char* path)
5  {
6      std::fstream file(path);
7      std::cout << file.rdbuf();
8  }
```

Biblioteka standardowa dostarcza rozwiązanie oparte na RAII:

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

Obsługa plików

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

void print_file(const char* path)
{
    std::fstream file(path);
    std::cout << file.rdbuf();
}
```

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

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

void print_file(const char* path)
{
    std::fstream file(path);
    std::cout << file.rdbuf();
}
```

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 "Resource.h"

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

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

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

private:
    Resource resource;
};
```

Resource Handler

Bez kopiowania?

Kopia obiektu handler dysponowałaby wskaźnikiem na ten sam zasób i mogłby 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)
    : resource(std::exchange(other.resource, 0x0))
{}

ResourceHandler& ResourceHandler::operator=(ResourceHandler&& other)
{
    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 specjalnych 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...

Tak jest w istocie

std::unique_ptr

std::unique_ptr

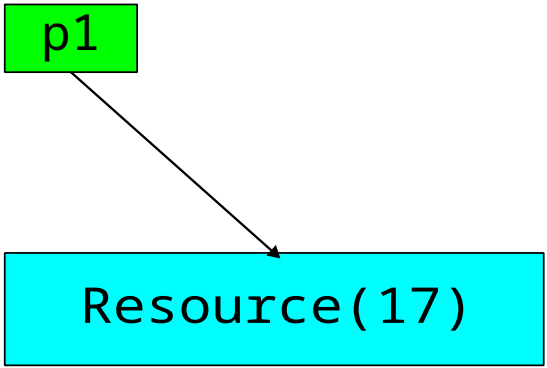
Od C++11 dostępny jest w bibliotece standardowej sprytny wskaźnik (ang. *smart pointer*): `std::unique_ptr`.
Nie da się go skopiować - jest on jedynym *właścicielem* obiektu, na który wskazuje - stąd nazwa.
Można go przenieść - transakcja pozbawia źródłowy obiekt wskaźnika na zasób i przypisuje go docelowemu.

std::unique_ptr

Sprytny?

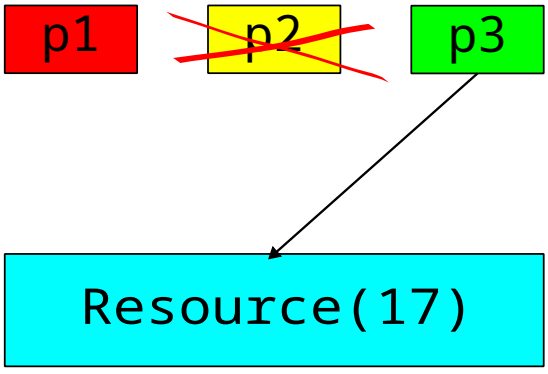
Spryt `std::unique_ptr` polega na wykorzystaniu RAII do *automatycznego* zwolnienia pamięci zaalokowanej *dynamicznie* - zapobiega to wyciekom pamięci.

std::unique_ptr




```cpp
#include <memory> // std::unique_ptr

#include "Resource.h"

void unique_resource()
{
    std::unique_ptr<Resource> p1(new Resource(17));
    // std::unique_ptr<Resource> p2 = p1; // error!
    std::unique_ptr<Resource> p3 = std::move(p1);

    use_resource_the_intended_way(*p3);

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

std::unique_ptr




```cpp
#include <memory> // std::unique_ptr

#include "Resource.h"

void unique_resource()
{
    std::unique_ptr<Resource> p1(new Resource(17));
    // std::unique_ptr<Resource> p2 = p1; // error!
    std::unique_ptr<Resource> p3 = std::move(p1);

    use_resource_the_intended_way(*p3);

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

std::unique_ptr

Alternatywna funkcja usuwająca - malloc i free

```
1  #include <cstdlib> // malloc/free
2  #include <memory> // std::unique_ptr
3
4  void perform_task(char* buffer, std::size_t buf_size);
5
6  void malloc_deleter(std::size_t buf_size)
7  {
8      std::unique_ptr<char, decltype(&free)> buffer((char*)malloc(buf_size),
9                                                    &free);
10
11     if (buffer) {
12         perform_task(buffer.get(), buf_size); // may throw!
13     }
14 }
```

std::unique_ptr

Alternatywna funkcja usuwająca - fopen i fclose

```
#include <cstdio> // fopen/fclose
#include <memory> // std::unique_ptr

void process_file(FILE* file);

void deleter_file(const char* path)
{
    std::unique_ptr<FILE, decltype(&fclose)> file(fopen(path, "r"), &fclose);

    if (file) {
        process_file(file.get()); // may throw!
    }
}
```

std::make_unique

```cpp
#include <memory> // std::make_unique

#include "Resource.h"

void make_unique_resource()
{
    auto p1 = std::make_unique<Resource>(17);

    use_resource_the_intended_way(*p1);
}
```

std::make_unique

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

A jeśli utknęliśmy w 2011...

```cpp
#include <memory>

template <typename T, typename... Args>
std::unique_ptr<T> makeUniqueCpp11(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
- kod nie jest odporny na nieuwagę użytkownika

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
- kod nie jest odporny na nieuwagę użytkownika

```
double free or corruption (top)
Aborted
```

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 nieuwagę użytkownika

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 sprawę - program zaczyna się kompilować...

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 sprawę - program zaczyna się kompilować...

std::move

```cpp
#include "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);

    for (EVER) {
        // event = event_queue.pop();
        // process(event);
    }

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

std::move

```cpp
#include "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)); // config is released

    for (EVER) {
        // event = event_queue.pop();
        // 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

```cpp
#include "Data.h"

std::unique_ptr<Data> load_config();
void startup(Data const& cfg);

int main(int argc, const char* argv[])
{
    auto cfg = load_config();
    startup(*cfg);

    for (EVER) {
        // event = event_queue.pop();
        // process(event, *cfg);
    }

    return 0;
}
```

Pobranie referencji do obiektu?

Jeśli chcemy wielokrotnie uzywać obiektu zarządzanego przez sprytny wskaźnik, to możemy wyłuskać referencję na obiekt. Referencja (i zwyky wskaźnik) nie zmienia czasu życia (zakresu) 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.

Nie tylko `std::unique_ptr`

Gdzie jest wykorzystany RAII?

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 była tworzona gdy język C++ dopiero powstawał - nie było nawet 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

```
#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 with initialization
    GameData data;

    return 0;
}
```

Shared Resource

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

Wydajność

Shared Resource

Nie zawsze jednak możemy (lub chcemy) dawać odpowiedzialności za zasób jednej encji na wyłączność.
Na szczęście tu również mamy pełne wsparcie ze strony biblioteki standardowej.

std::shared_ptr

std::shared_ptr

Gdy nie jesteśmy w stanie określić pojedynczego właściciela obiektu,
a nadal chcemy korzystać z dobrodziejstw 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> // std::shared_ptr

#include "Resource.h"

void shared_resource()
{
    std::shared_ptr<Resource> p1(new Resource(17));
    std::shared_ptr<Resource> p2 = p1; // now can easily be done!
    std::shared_ptr<Resource> p3 = std::move(p1); // still possible

    use_resource_the_intended_way(*p3);
} // p3, p2, p1 go out-of-scope, Resource is destroyed
```

std::shared_ptr

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

`std::shared_ptr`

Skąd jednak **`std::shared_ptr`** wie, że jest tym ostatnim?
Służy mu do tego mechanizm zliczania referencji.

std::shared_ptr

Skąd jednak `std::shared_ptr` wie, że jest tym ostatnim?
Służy mu do tego mechanizm zliczania referencji.
Każdy kolejny `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.

std::shared_ptr

Skąd jednak `std::shared_ptr` wie, że jest tym ostatnim?
Służy mu do tego mechanizm zliczania referencji.
Każdy kolejny `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.
Zniszczenie `std::shared_ptr` powoduje dekrementację licznika.

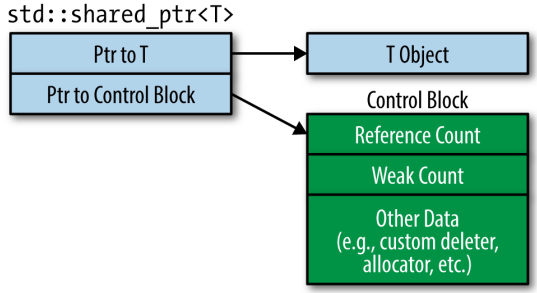
std::shared_ptr

Skąd jednak `std::shared_ptr` wie, że jest tym ostatnim?
Służy mu do tego mechanizm zliczania referencji.
Każdy kolejny `std::shared_ptr` wskazujący na obiekt podbija licznik odwołań.
Zniszczenie `std::shared_ptr` powoduje dekrementację licznika.
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 "Resource.h"

auto deleterOdd = [](Resource* res) { /*...*/ }; // custom deleters
auto deleterEven = [](Resource* res) { /*...*/ }; // different types

std::shared_ptr<Resource> p1(new Resource(13), deleterOdd);
std::shared_ptr<Resource> p2(new Resource(26), deleterEven);

std::vector<std::shared_ptr<Resource>> vec{p1, p2};
// would be impossible with std::unique_ptr!
```

std::shared_ptr

Wielowątkowość

`std::shared_ptr`

Wielowątkowość

Dostęp do zasobu możemy współdzielić nie tylko pomiędzy kolejnymi `std::shared_ptr`, wskaźniki te mogą znajdować się w różnych wątkach naszej aplikacji.

std::shared_ptr

Wielowątkowość

Dostęp do zasobu możemy współdzielić nie tylko pomiędzy kolejnymi `std::shared_ptr`, wskaźniki te mogą znajdować się w różnych wątkach naszej aplikacji.
Dlatego właśnie blok kontrolny nie może leżeć na stosie, ale również musi być alokowany na stercie.

std::shared_ptr

Wielowątkowość

Dostęp do zasobu możemy współdzielić nie tylko pomiędzy kolejnymi `std::shared_ptr`, wskaźniki te mogą znajdować się w różnych wątkach naszej aplikacji.
Dlatego właśnie blok kontrolny nie może leżeć na stosie, ale również musi być alokowany na stercie.
Co więcej, wszystkie operacje przeprowadzane na bloku kontrolnym muszą być synchronizowane.

std::shared_ptr

Wielowątkowość

Dostęp do zasobu możemy współdzielić nie tylko pomiędzy kolejnymi `std::shared_ptr`, wskaźniki te mogą znajdować się w różnych wątkach naszej aplikacji.
Dlatego właśnie blok kontrolny nie może leżeć na stosie, ale również musi być alokowany na stercie.
Co więcej, wszystkie operacje przeprowadzane na bloku kontrolnym muszą być synchronizowane.
Jest to obarczone kosztem wydajnościowym, ale daje gwarancję braku wyścigu.

std::shared_ptr

Wielowątkowość

Dostęp do zasobu możemy współdzielić nie tylko pomiędzy kolejnymi `std::shared_ptr`, wskaźniki te mogą znajdować się w różnych wątkach naszej aplikacji.
Dlatego właśnie blok kontrolny nie może leżeć na stosie, ale również musi być alokowany na stercie.
Co więcej, wszystkie operacje przeprowadzane na bloku kontrolnym muszą być synchronizowane.
Jest to obarczone kosztem wydajnościowym, ale daje gwarancję braku wyścigu.
Ważne! Tylko blok kontrolny jest thread-safe by design.

std::make_shared

std::make_shared

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

`std::make_shared`

Wady i zalety

- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.

`std::make_shared`

Wady i zalety

- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.
- Unikamy problemu kolejności ewaluacji argumentów funkcji.

`std::make_shared`

Wady i zalety

- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.
- Unikamy problemu kolejności ewaluacji argumentów funkcji.
- Pomaga z fragmentacją pamięci.

`std::make_shared`

Wady i zalety

- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.
- Unikamy problemu kolejności ewaluacji argumentów funkcji.
- Pomaga z fragmentacją pamięci.
- Nie da się przekazać własnego deletera.

`std::make_shared`

Wady i zalety

- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.
- Unikamy problemu kolejności ewaluacji argumentów funkcji.
- Pomaga z fragmentacją pamięci.
- Nie da się przekazać własnego deletera.
- `std::make_shared` wykonuje tylko jedną alokację, we wspólnym bloku pamięci dla obiektu i bloku kontrolnego.

std::make_shared

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

```
1  #include "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

```cpp
#include "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

```
#include "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.
Możemy zatem otrzymać taką kolejność:

std::make_shared

```cpp
#include "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.
Możemy zatem otrzymać taką kolejność:

- wykonanie `operator new`

std::make_shared

```cpp
#include "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.
Możemy zatem otrzymać taką kolejność:

- wykonanie `operator new`
- wykonanie `nastyFunction`

std::make_shared

```
#include "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.
Możemy zatem otrzymać taką kolejność:

- wykonanie `operator new`
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem

std::make_shared

```cpp
#include "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.
Możemy zatem otrzymać taką kolejność:

- wykonanie `operator new`
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem
- blok pamięci zarezerwowany przez `operator new` nie jest zwalniany - mamy wyciek.

std::make_shared

`std::make_shared` na ratunek!*

```
#include "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

```
#include "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!*

- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu

std::make_shared

```cpp
#include "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!*

- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu
- wykonanie `nastyFunction`

std::make_shared

```cpp
#include "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!*

- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem

std::make_shared

```cpp
#include "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!*

- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem
- wykonanie destruktora `std::shared_ptr`, brak wycieku.

std::make_shared

```cpp
#include "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!*

- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem
- wykonanie destruktora `std::shared_ptr`, brak wycieku.

std::make_shared

```cpp
#include "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!*

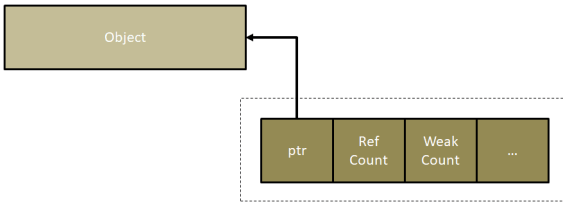
- wykonanie `std::make_shared` - alokacja bloku pamięci i zawołanie konstruktora obiektu
- wykonanie `nastyFunction`
- `nastyFunction` rzuca wyjątkiem
- wykonanie destruktora `std::shared_ptr`, brak wycieku.

*) Od standardu C++17 zostało to poprawione. Kompilator nadal może ewaluować argumenty w dowolnej kolejności, ale każdy z argumentów musi zostać w pełni ewaluowany bez przeplatania instrukcji.

std::make_shared

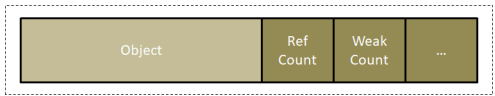
std::shared_ptr

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



std::make_shared

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



std::weak_ptr

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

std::weak_ptr

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

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'm here!\n";
    } else {
        std::cout << "too late...\n";
    }
}

int main()
{
    std::weak_ptr<int> weak;
    {
        auto shared = std::make_shared<int>(17);
        weak = shared;

        check(weak);
    }
    check(weak);
}
```

Output:

std::weak_ptr

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

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'm here!\n";
    } else {
        std::cout << "too late...\n";
    }
}

int main()
{
    std::weak_ptr<int> weak;
    {
        auto shared = std::make_shared<int>(17);
        weak = shared;

        check(weak);
    }
    check(weak);
}
```

Output:

```
use_count = 1
I'm 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";
}
```

Output:

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";
}
```

Output:

Finished using A and B

std::weak_ptr

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

struct B
{
    std::weak_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";
}
```

Output:

std::weak_ptr

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

struct B
{
    std::weak_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";
}
```

Output:

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

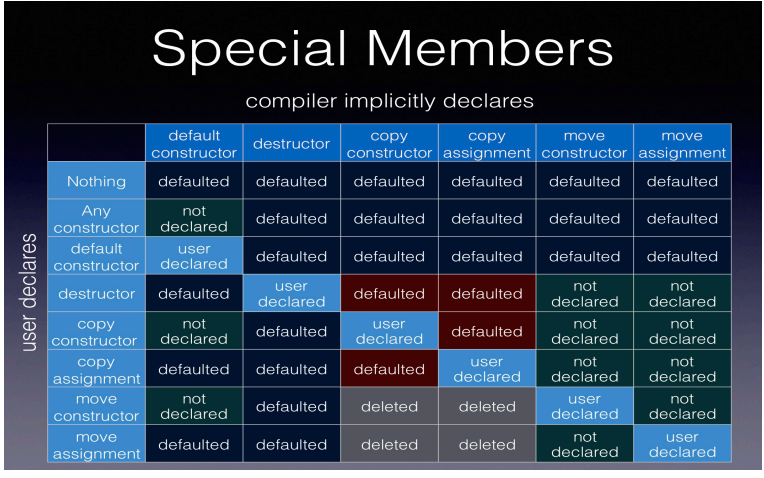
Rule of Zero

Jeżeli stworzona przez nas klasa potrzebuje zdefiniowania w niej któregokolwiek z poniższych:

- destruktora
- konstruktora kopiującego
- kopiującego operatora przypisania

powinna mieć zdefiniowane wszystkie trzy w celu zapewnienia jej 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

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_;
};
```

```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

Rule of three

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

Nickname& operator=(const Nickname& other)
{
    if (this == &other)
        return *this;

    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 jedynie 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

Rule of zero

```cpp
#include <string>

struct Nickname
{
    Nickname(const std::string& input) : nick_(input) {}

private:
    std::string nick_;
};

int main()
{
    Nickname n1("Whatever");
    // copy constructor is implicitly generated by compiler again
    Nickname n2(n1);
    // same for copy assignment operator
    Nickname n3 = n2;
} // n3, n2, n1 go out-of-scope; destructors called;
  // everything works correctly automagically thanks to RAII
```

Wydajność

Alokacja

Funkcje i procedury

Resource Acquisition Is Initialization

Shared Resource

Rule of Zero

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*

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

struct Data
{
    char tab_[42];
};

int main()
{
    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()
{
    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>

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

struct Data
{
    char tab_[42];
};

int main()
{
    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>

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

struct Data
{
    char tab_[42];
};

int main()
{
    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()
{
    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-2023.zip`
polecenia	`exercises/README_PL.md`
e-mail do prowadzących	`bogumil.chojnowski@nokia.com`
	`pawel.krysiak@nokia.com`
tytułem	`[PARO2023] Memory Management C++`
z załącznikiem	archiwum ZIP
	tylko zawartość katalogu z ćwiczeniami
	bez skompilowanych binarek
	slajdów też nie potrzebujemy


NOKIA

Algorytmy i Struktury Danych

Memory Management API

Struktura danych: kopiec

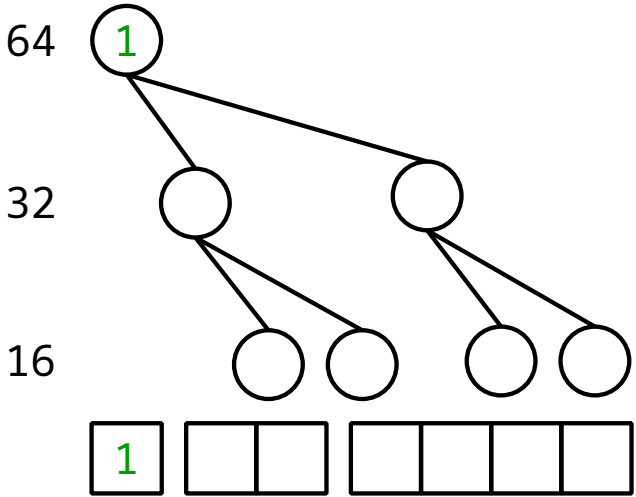
Kopiec - drzewo binarne na tablicy

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

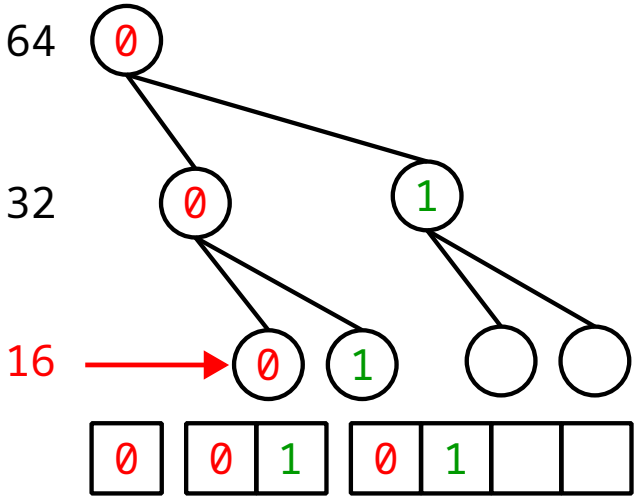
Rodzeństwo (siblings)

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.

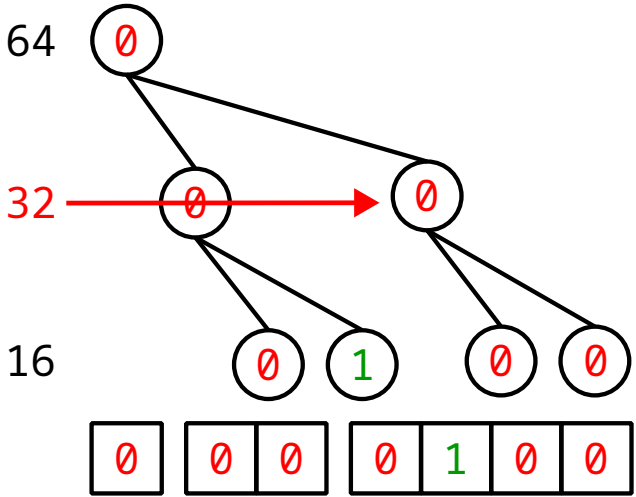
Struktura danych: kopiec



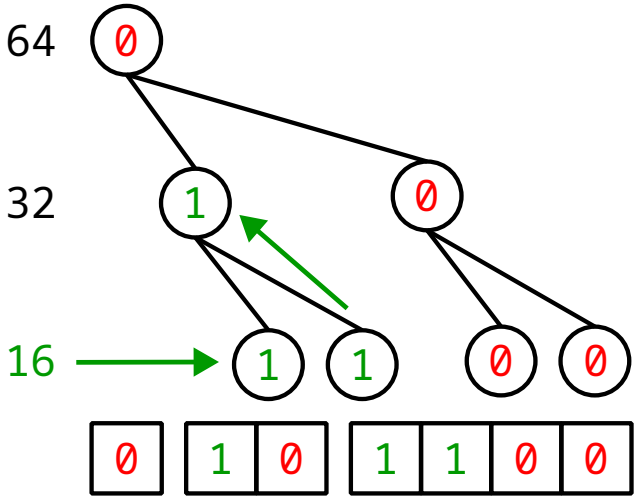
Struktura danych: kopiec



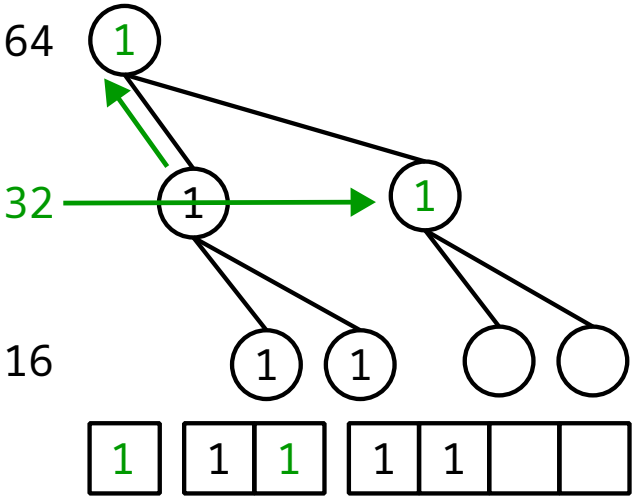
Struktura danych: kopiec



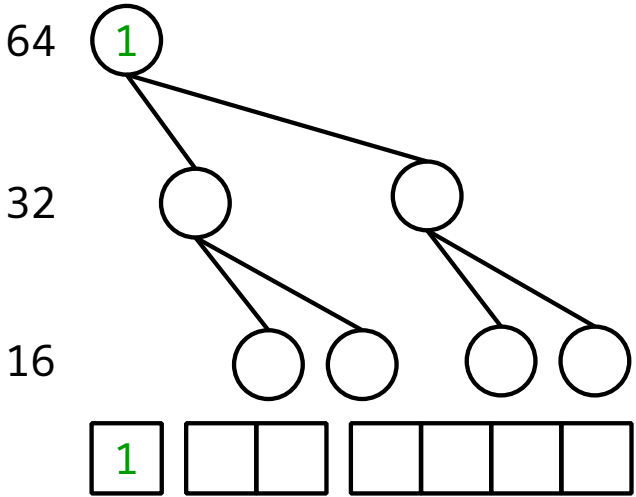
Struktura danych: kopiec



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.

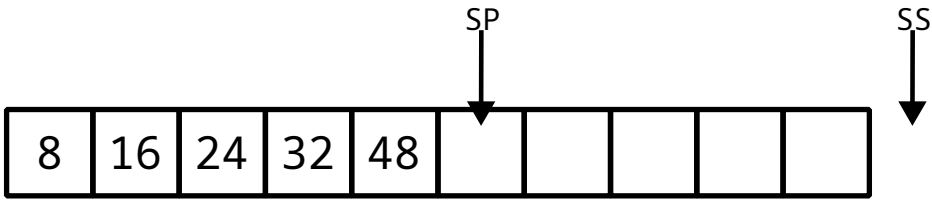
Struktura danych: stos

Rozmiar stosu jest stały

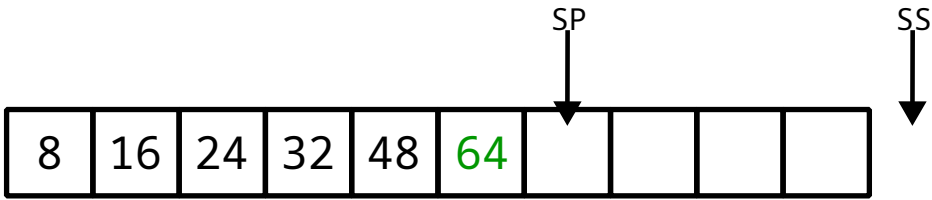
Stos wywołań funkcji jest stałego rozmiaru - alokowany raz przy uruchomieniu programu. Z racji wykorzystania prostego wskaźnika stosu, musi to być obszar ciągły. Realokowanie stosu zdezaktualizowałoby wszystkie wskaźniki na lokalne zmienne. Z ustalonego rozmiaru wynika niebezpieczeństwo przepełnienia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

Struktura danych: stos

```
push(64)
```

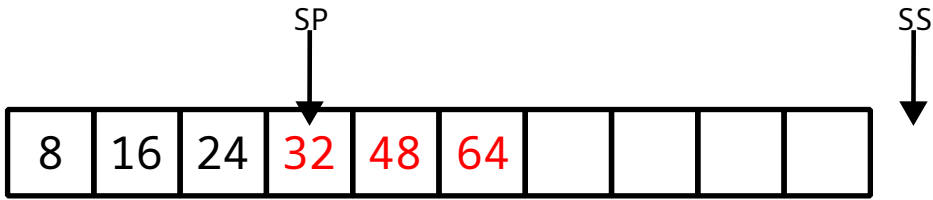


Struktura danych: stos



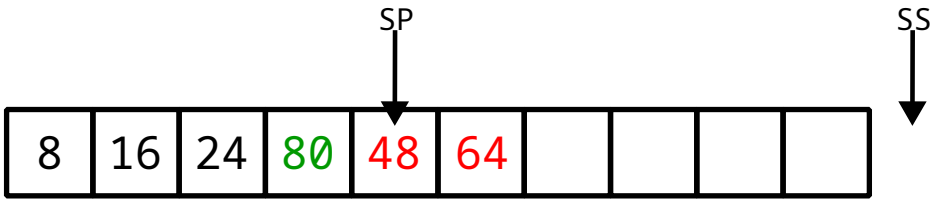
Struktura danych: stos

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



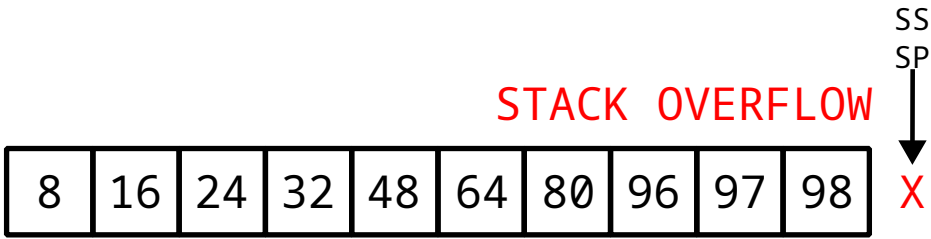
Struktura danych: stos

```
push(80)
```



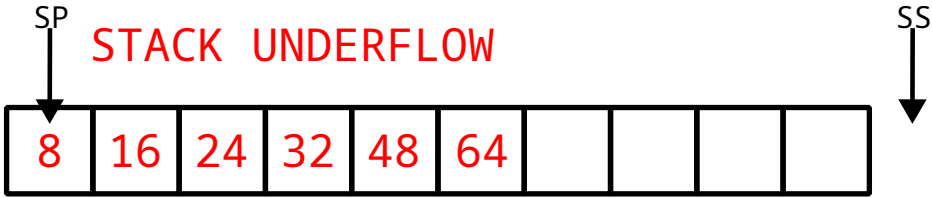
Struktura danych: stos

```
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`**

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*.