PODSTAWY C++

TYPY DANYCH

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. Typy podstawowe
2. Aliasy typów
3. Arytmetyka
4. Tablice
5. Referencje
6. Typy wyliczeniowe `enum`
7. Wskaźniki
8. Zagrożenia związane ze wskaźnikami

REPOZYTORIUM

coders-school/cpp-fundamentals

PODSTAWY C++

TYPY PODSTAWOWE

CODERS SCHOOL

PROSTA MATEMATYKA

- 1 bajt == 8 bitów
- W binarnym totolotku wylosowane liczby mogą mieć 0 lub 1
- Zatem podczas losowania 8 numerków możemy otrzymać przykładowo: `1 0 1 0 1 0 1 0`
- Takich kombinacji jest dokładnie `256 -> (2^8)`
- Zatem na 1 bajcie (8 bitach) możemy zapisać 256 liczb, np. od 0 do 255
- Jeżeli w totolotku losujemy 32 numerki, (32/8 = 4) czyli 4 bajty to takich kombinacji jest `2^32` (czyli ponad 4 miliardy)

TYP PUSTY - `void`

- Nie można tworzyć obiektów typu `void`
- Służy do zaznaczenia, że funkcja nic nie zwraca
- Można tworzyć wskaźniki `void*` (zła praktyka w C++)
- NIE służy do oznaczania, że funkcja nie przyjmuje argumentów

```
int fun(void) { /* ... */ }  // bad practice, C style
int fun() { /* ... */ }      // good practice, C++ style
```

TYP LOGICZNY - `bool`

- Rozmiar: co najmniej 1 bajt (zazwyczaj równy właśnie 1)
  - `sizeof(bool) >= 1`
- 2 możliwe wartości
  - `false`
  - `true`

TYPY ZNAKOWE

- Rozmiar: 1 bajt
- 256 możliwych wartości
- `char` -> od `-128` do `127`
- `unsigned char` -> od `0` do `255`

Przedrostek `unsigned` oznacza, że typ jest bez znaku (bez liczb ujemnych), czyli od 0 do jakieś dodatniej wartości.

Rozmiar typów znakowych to zawsze 1 bajt.

Rozmiary dalszych typów zależą od platformy np. 32 bity, 64 bity.

TYPY CAŁKOWITOLICZBOWE

- `short (unsigned short)` - co najmniej 2 bajty
- `int (unsigned int)` - co najmniej 2 bajty
- `long (unsigned long)` - co najmniej 4 bajty
- `long long (unsigned long long)` - co najmniej 8 bajtów

TYPY ZMIENNOPRZECINKOWE

- `float` - zwykle 4 bajty
- `double` - zwykle 8 bajtów
- `long double` - zwykle 10 bajtów (rzadko stosowany)
- Typy zmiennoprzecinkowe zawsze mogą mieć ujemne wartości (nie istnieją wersje unsigned)
- Posiadają specjalne wartości:
  - `0`, `-0` (ujemne zero)
  - `-Inf`, `+Inf` (Infinity, nieskończoność)
  - `NaN` (Not a Number)

Uwaga! Porównanie `NaN == NaN` daje `false`

Zaawansowana lektura: Standard IEEE754 definiujący typy zmiennoprzecinkowe

ROZMIARY TYPÓW

Standard C++ definiuje taką zależność pomiędzy rozmiarami typów całkowitoliczbowych

```
1 == sizeof(char) \
  <= sizeof(short) \
  <= sizeof(int) \
  <= sizeof(long) \
  <= sizeof(long long);
```

TYP `auto`

W pewnych miejscach możemy użyć typu `auto`. Kompilator sam wydedukuje typ, np. na podstawie przypisanej wartości.

```
auto num = 5;          // int
auto num = 5u;         // unsigned int
auto num = 5.5;        // double
auto num = 5.f;        // float
auto letter = 'a';     // char
auto num = false;      // bool
auto sth;              // compilation error, unable to deduce type
```

MAŁY SUCHAR

Kim jest Hobbit?

Jest to 1/8 Hobbajta :)

LINKI DLA POSZERZENIA WIEDZY

- Fundamental types on cppreference.com
- Standard IEEE754 definiujący typy zmiennoprzecinkowe

PODSTAWY C++

ALIASY TYPÓW

CODERS SCHOOL

ALIASY TYPÓW

Istnieją też typy, która są aliasami (inne nazewnictwo w celu lepszego zrozumienia typu).

`std::size_t` w zależności od kompilatora może być typu (`unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`). Przeważnie jest on typu `unsigned int`. Warto wykorzystywać go, gdy nasza zmienna będzie odnosić się do jakiegoś rozmiaru np. wielkość tablicy.

Własne aliasy typów możemy tworzyć używając `typedef` lub `using`

```
typedef int Number;
Number a = 5;   // int a = 5;

using Fraction = double;
Fraction b = 10.2;  // double b = 10.2;
```

PODSTAWY C++

OPERACJE ARYTMETYCZNE

CODERS SCHOOL

OPERACJE ARYTMETYCZNE

- Podstawowe: + - * /
- Modyfikujące zmienną: += -= *= /=
- Inkrementujące (+1) zmienną: ++
- Dekrementujące (-1) zmienną: --

PRZYKŁADY

```
int a = 5 + 7; // a = 12
```

```
int a = 5;
a += 7; // a = 12
```

```
int a = 5;
++a; // a = 6
a--; // a = 5
```

PYTANIA

```
int i = 5;
auto j = i++ - 1;
```

Ile wynoszą wartości `i` oraz `j`?

`i = 6`

`j = 4`

Jakiego typu jest `j`?

`int`

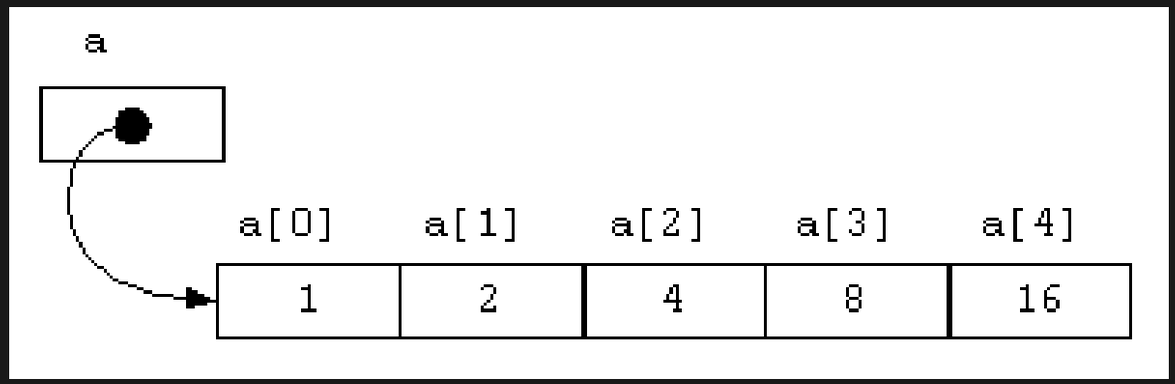
PODSTAWY C++

TABLICE

CODERS SCHOOL

WPROWADZENIE DO TABLIC

- Tablice można traktować jak wagony w pociągu
- Ustawione kolejno jeden po drugim i połączone ze sobą
- Mogą pomieścić różne typy, jak człowiek, węgiel, itp.
- 10 wagonów z węglem możemy zapisać jako `Coal tab[10]` - oznacza to, że tworzymy tablicę, która przechowuje 10 elementów typu Coal (węgiel).



- W C++ tablica znajduje się w jednym, ciągłym obszarze w pamięci i jest nierozłączna (nie można usuwać jej elementów)
- Wszystkie elementy są tego samego typu
- Tablica jest zawsze indeksowana od 0
- `tab[0]` - pierwszy element tablicy `tab`
- `tab[9]` - ostatni element 10-cio elementowej tablicy `tab`

PRZYKŁAD MODYFIKACJI TABLICY

```
int tab[10];
tab[0] = 1;
tab[1] = 2;
// ...
tab[9] = 10;
```

Można to zrobić lepiej z użyciem pętli.

operator[]

Do elementu tablicy odwołujemy się przez `operator[]`. Musimy pamiętać, żeby zawsze odwoływać się do istniejącego elementu tablicy. Inaczej program będzie miał niezdefiniowane zachowanie, gdyż spróbujemy uzyskać dostęp do pamięci, która nie należy do tablicy. Mówimy, że znajdują się tam śmieci. W najlepszym przypadku system operacyjny to wykryje i dostaniemy **crash** (segmentation fault). W najgorszym będziemy działać na niepoprawnych losowych danych. Skutki mogą być bardzo poważne (katastrofy promów kosmicznych, napromieniowanie od aparatury medycznej).

```
int tab[10];
tab[10] = 42;  // !!! undefined behavior (UB)
```

ZADANIE

Zmodyfikuj program, tak aby wypełniał tablicę kolejnymi nieparzystymi liczbami: 1, 3, 5, 7, ...

Pobierz zadanie

```
#include <iostream>

constexpr size_t tab_size = 100;

int main() {
    int tab[tab_size];

    for (size_t i = 0; i < tab_size; ++i) {
        tab[i] = i;
    }

    for (size_t i = 0; i < tab_size; ++i) {
        std::cout << tab[i] << "\n";
    }

    return 0;
}
```

PODSTAWY C++

REFERENCJE

CODERS SCHOOL

&

Magiczny znaczek `&` (ampersand) oznacza referencję.

```
int value = 5;
int & number = value;
```

Powyższy zapis oznacza zmienną `number` typu `int&`, czyli referencję na typ `int`.

Nie ma znaczenia, czy referencję dokleimy do typu, czy nazwy zmiennej, ale referencja jest oddzielnym typem, więc sugerujemy nie doklejać jej do nazwy zmiennej.

```
int& number = value;  // lewak
int &number = value;  // prawak (odradzane)
int & number = value; // neutralny
```

CZYM JEST REFERENCJA?

Spójrzmy na fragment kodu.

```
int number = 5;
int& reference = number;

std::cout << reference << '\n';   // 5
std::cout << ++reference << "\n"; // 6
std::cout << number << "\n";  // 6
```

- Referencja odwołuje się do istniejącego obiektu
- Jeżeli utworzymy obiekt `int value` to poprzez referencje `int& reference = value` będziemy mogli się do niego bezpośrednio odwoływać.
- Referencja to inna, dodatkowa nazwa dla tej samej zmiennej (alias)
- Modyfikacja referencji = modyfikacja oryginalnego obiektu

CO ZYSKUJEMY W TEN SPOSÓB?

- Nie musimy kopiować elementów. Wystarczy, że przekażemy referencje.
  - W ten sposób możemy swobodnie w wielu miejscach programu odczytywać wartość tej zmiennej, bez zbędnego jej kopiowania.
- Referencja zajmuje w pamięci tyle, ile zajmuje adres (4 lub 8 bajtów).
- Tworzenie referencji do typu `int` (zazwyczaj 4 bajty) nie zawsze ma sens optymalizacyjny, chyba, że chcemy zmodyfikować ten element wewnątrz funkcji.
- Przekazywanie argumentów przez referencje nabierze więcej sensu, kiedy poznamy już klasy i obiekty :)

Ile miejsca zajmuje referencja? - stackoverflow.com

JAK PRZEKAZAĆ ELEMENT PRZEZ REFERENCJĘ?

```
void foo(int& num) {
    std::cout << num; // good
    num += 2;         // good
}
```

Jeśli chcemy mieć pewność, że funkcja nie zmodyfikuje nam wartości (chcemy ją przekazać tylko do odczytu) dodajemy `const`.

```
void bar(const int& num) {
    std::cout << num; // good
    num += 2;         // compilation error, num is const reference
}
```

Wywołanie funkcji to po prostu:

```
int num = 5;
foo(num);
bar(num);
```

ZADANIE

Zaimplementuj funkcję `modifyString()`. Ma ona przyjąć i zmodyfikować przekazany tekst. Na ekranie chcemy zobaczyć `"Other string"`. Pobierz zadanie

```
#include <iostream>
#include <string>

// TODO: Implement modifyString()
// It should modify passed string to text "Other string"

int main() {
    std::string str{"Some string"};
    modifyString(str);
    std::cout << str << '\n';
    return 0;
}
```

PODSUMOWANIE

- referencja jest aliasem (inną nazwą dla zmiennej)
- modyfikacja referencji to modyfikacja oryginalnego obiektu
- przy przekazywaniu parametru przez referencję:
  - unikamy zbędnych kopii
  - modyfikacja obiektu będzie skutkowała zmodyfikowaniem oryginału przekazanego do funkcji

PODSTAWY C++

enum | enum class

CODERS SCHOOL

TYP WYLICZENIOWY

`enum` to po polsku typ wyliczeniowy. W C++11 wprowadzono także `enum class` zwany silnym typem wyliczeniowym.

PRZYKŁAD

Załóżmy, że piszemy oprogramowanie do pralki. Chcielibyśmy utworzyć także interfejs zwracający numer błędu np:

- brak wody
- zbyt duże obciążenie
- problem z łożyskami
- blokada pompy

W celu warto użyć typu `enum` lub lepiej - `enum class`.

IMPLEMENTACJA PRZYKŁADU

```
enum ErrorCode {
    lack_of_water;
    too_much_load;
    bearing_problem;
    block_of_pump;
};

// or better ↓

enum class ErrorCode {
    lack_of_water;
    too_much_load;
    bearing_problem;
    block_of_pump;
};
```

NUMERACJA

Typ `enum` pod spodem numerowany jest od `0` do `n - 1`, gdzie `n` to liczba elementów.

Jeżeli chcemy nadać inne wartości musimy to zrobić ręcznie:

```
enum class ErrorCode {
  lack_of_water = 333;
  to_much_load; // will be 334
  bearing_problem = 600;
  block_of_pump; // will be 601
}
```

`enum` VS `enum class`

`enum` od `enum class` różni się głównie tym, że możemy niejawnie skonwertować typ `enum` na `int` (w końcu to typ wyliczeniowy).

Natomiast typ `enum class` możemy skonwertować na `int`, tylko poprzez jawne rzutowanie. Nie będziemy na razie omawiać rzutowania. Warto tylko pamiętać, że robimy to wywołując:

```
int num = static_cast<int>(ErrorCode::lack_of_water)
```

W jakich innych przypadkach zastosujesz typ wyliczeniowy?

`enum` VS `enum class`

Druga różnica - dla `enum` możemy mieć konflikt nazw, dla `enum class` nie.

```
enum Color {
    RED,    // 0
    GREEN,  // 1
    BLUE    // 2
};
```

```
enum TrafficLight {
    GREEN,  // 0
    YELLOW, // 1
    RED     // 2
};
```

```
auto lightColor = getColor();
if (lightColor == RED) {  // 0 or 2?
    stop();
} else {
    go();
}
```

UŻYCIE WARTOŚCI Z `enum class`

Aby uniknąć konfliktu nazw stosujemy `enum class`.

```
enum class Color {
    RED,
    GREEN,
    BLUE,
}
```

```
enum class TrafficLight {
    GREEN,
    YELLOW,
    RED
}
```

```
auto lightColor = getColor();
if (lightColor == TrafficLight::RED) {
    stop();
} else {
    go();
}
```

PODSTAWY C++

WSKAŹNIKI

CODERS SCHOOL

WSKAŹNIKI - ANALOGIA

Poza referencjami istnieją także wskaźniki. Wskaźniki działają podobnie jak referencje.

Wyobraźmy sobie, że planujemy wycieczkę na Majorkę. Wsiadamy do samolotu i lecimy. Na miejscu okazuje się, ze zapomnieliśmy jaki jest adres hotelu :(W celu znalezienia go musimy zadzwonić do biura podróży, poczekać na obsługę, wytłumaczyć całą zawiłą historię, aż w końcu po długim czasie otrzymujemy adres naszego hotelu. Proces zdobycia tych informacji był dla nas czasochłonny.

Wyobraźmy sobie jednak, że uprzednio zapisaliśmy sobie w telefonie adres naszego hotelu. Aby przypomnieć sobie, gdzie on się znajdował wystarczy, że sprawdzimy telefon i już wiemy. Proces ten zajął nam dużo mniej czasu.

WSKAŹNIKI W C++

Podobnie jest w C++. Wskaźniki służą do wskazywania miejsca w pamięci, gdzie znajduje się pożądany przez nas obiekt.

Procesor nie musi odpytywać każdorazowo magistrale pamięci, gdzie znajduje się podana zmienna, tylko od razu wie, jaki jest jej adres (unikamy pośredników jak telefon do biura obsługi).

Ponadto jeżeli funkcja przyjmuje wskaźnik, nie musi ona kopiować całej zawartości obiektu, co jest czasochłonne. Można dużo szybciej wskazać gdzie ten obiekt już istnieje.

JAK PRZEKAZAĆ ELEMENT PRZEZ WSKAŹNIK?

```
void foo (int* num) {
    std::cout << *num;  // good
    *num += 2;          // good
}
```

Gdy chcemy mieć pewność, że nikt nie zmodyfikuje nam wartości (chcemy ją przekazać tylko do odczytu) dodajemy `const`.

```
void bar (int const* num) {
    std::cout << *num;  // good
    *num += 2;          // compilation error, num is a pointer to const
}
```

Wywołanie funkcji to:

```
int num = 5;
foo(&num);
bar(&num);
```

GDZIE DAĆ CONST?

CO TO JEST?

```
const int * ptr;
```

Wskaźnik na stałą (`const int`).

```
int const * ptr;
```

Również wskaźnik na stałą (`const int = int const`).

```
int * const ptr;
```

Stały wskaźnik na zmienną (`int`).

STAŁE WSKAŹNIKI A WSKAŹNIKI NA STAŁE

```
int const * const ptr;
const int * const ptr;
```

Stały wskaźnik na stałą (`int const = const int`).

Jest to częste pytanie z rozmów kwalifikacyjnych. Aby stały był wskaźnik, `const` musi być za gwiazdką.

RÓŻNICE

WSKAŹNIK NA STAŁĄ

```
const int * ptr = new int{42};
*ptr = 43;     // compilation error: assignment of read-only location ‘* ptr’
ptr = nullptr; // ok
```

- Nie możemy zmodyfikować obiektu wskazywanego przez wskaźnik
  - Odwołania z * nie mogą modyfikować obiektu
- Możemy zmodyfikować sam wskaźnik, np. aby wskazywał na inny obiekt
  - Odwołania bez * mogą modyfikować wskaźnik

RÓŻNICE

STAŁY WSKAŹNIK

```
int * const ptr = new int{42};
*ptr = 43;      // ok
ptr = nullptr; // compilation error: assignment of read-only variable ‘ptr’
```

- Możemy zmodyfikować obiekt wskazywany przez wskaźnik
  - Odwołania z * mogą modyfikować obiekt
- Nie możemy zmodyfikować samego wskaźnika, np. aby wskazywał na inny obiekt
  - Odwołania bez * nie mogą modyfikować wskaźnika

STAŁY WSKAŹNIK NA STAŁĄ

```
const int * const ptr = new int{42};
*ptr = 43;     // compilation error: assignment of read-only location ‘* ptr’
ptr = nullptr; // compilation error: assignment of read-only variable ‘ptr’
```

- Nie możemy zmodyfikować obiektu wskazywanego przez wskaźnik
  - Odwołania z * nie mogą modyfikować obiektu
- Nie możemy zmodyfikować samego wskaźnika, np. aby wskazywał na inny obiekt
  - Odwołania bez * nie mogą modyfikować wskaźnika

ZADANIE

Zaimplementuj funkcje `foo()` i `bar()`.

`foo()` powinno zmodyfikować wartość przekazaną przez wskaźnik na 10, a `bar()` na 20.

Czy `foo()` lub `bar()` mogą przyjąć wskaźnik na stałą lub stały wskaźnik? Pobierz zadanie

```
#include <iostream>

// TODO: Implement foo() and bar()
// foo() should modify value under passed pointer to 10
// bar() should modify value under passed pointer to 20
// Can we have a pointer to const or a const pointer?
int main() {
    int number = 5;
    int* pointer = &number;
    std::cout << number << '\n';
    foo(&number);
    std::cout << number << '\n';
    bar(pointer);
    std::cout << number << '\n';

    return 0;
}
```

RÓŻNICE MIĘDZY WSKAŹNIKIEM I REFERENCJĄ

ODWOŁANIA

- Do referencji odwołujemy się tak samo jak do zwykłego obiektu - za pomocą nazwy
- Aby uzyskać element wskazywany przez wskaźnik musimy dodać * przed nazwą wskaźnika

PRZEKAZYWANIE JAKO ARGUMENT

- Argument jest referencją lub zwykłą zmienną (kopią) - przekazujemy nazwę
- Argument jest wskaźnikiem a przekazujemy zmienną - musimy dodać & przed nazwą zmiennej.

OZNACZENIA

- Symbol * (operator dereferencji) oznacza dostęp do obiektu wskazywanego
- Jeżeli nie damy * przy wskaźniku dostaniemy adres obiektu wskazywanego
- Symbol & oznacza pobranie adresu naszej zmiennej
- Powyższe ma sens, ponieważ wskaźnik wskazuje miejsce w pamięci (adres wskazywanego obiektu)

RÓŻNICE W KODZIE

```
void copy(int a) { a += 2; }
void ref(int& a) { a += 2; }
void ptr(int* a) ( *a += 2; )

void example() {
    int c = 10;
    int& r = c;
    int* p = &c;  // typically int* p = new int{10};
    copy(c);
    copy(r);
    copy(*p);
    ref(c);
    ref(r);
    ref(*p);
    ptr(&c);
    ptr(&r);
    ptr(p);
}
```

CO OZNACZA * W KODZIE?

```
int a = 5 * 4;       // jako operacja arytmetyczna - mnożenie
int* b = &a;         // przy typie - wskaźnik na ten typ
int *c = &a;         // przy typie - wskaźnik na ten typ
std::cout << *b;     // przy zmiennej wskaźnikowej - dostęp do obiektu
int fun(int* wsk);   // w argumencie funkcji - przekazanie wskaźnika (adresu)
```

CO OZNACZA & W KODZIE?

```
int a = 5 & 4;       // jako operacja arytmetyczna - suma bitowa
int& b = a;          // przy typie - referencja na ten typ
int &c = a;          // przy typie - referencja na ten typ
std::cout << &a;     // przy zmiennej - adres tej zmiennej w pamięci
int fun(int& ref);   // w argumencie funkcji - przekazanie adresu
```

WAŻNA ZASADA

Jeśli nie ma absolutnej potrzeby, to nie używamy wskaźników w ogóle.

PODSTAWY C++

ZAGROŻENIA

W STOSOWANIU REFERENCJI I WSKAŹNIKÓW

CODERS SCHOOL

PUSTE WSKAŹNIKI

```
int* a = nullptr;
std::cout << *a;
```

Dostęp do zmiennej wskazywanej przez pusty wskaźnik to niezdefiniowane zachowanie.

Pusty wskaźnik oznaczamy zawsze używając `nullptr`.

Nie używamy `NULL` znanego z języka C lub wcześniejszych standardów, bo jest on mniej bezpieczny.

```
void foo(int);
foo(NULL);     // bad - no error
foo(nullptr);  // good - compilation error
```

NIEZAINICJALIZOWANE WSKAŹNIKI

```
int* a;
std::cout << *a;
```

Wskaźnik `a` zawiera tzw. śmieci. Dostęp do obiektu wskazywanego przez taki wskaźnik to niezdefiniowane zachowanie.

ODWOŁANIA DO USUNIĘTYCH ZMIENNYCH

Jak już dobrze wiemy, zmienne lokalne są usuwane po wyjściu z zakresu, w którym je utworzyliśmy. Można już domyślać się, jakie problemy sprawią nam wskaźniki i referencje, gdy będą dalej istniały, a obiekt, do którego się odwołują już zostanie zniszczony. Będzie to w najlepszym przypadku „crash", w najgorszym „undefined behaviour".

JAK ZAPOBIEGAĆ TAKIM PRZYPADKOM?

Zawsze musimy zapewnić, aby czas życia zmiennej, był dłuższy niż czas życia jej wskaźnika, czy referencji.

USUNIĘTE ZMIENNE - PRZYKŁAD

```
std::vector<int*> vec;

void createAndAddToVec(int amount) {
    for (int i = 0 ; i < amount ; ++i) {
        vec.push_back(&i);
    }
    // local variable i does not exist here anymore
    // vec contains addresses to not existing local variables
}

int main() {
    createAndAddToVec(5);
    for (const auto& el : vec) {
        std::cout << *el << '\n';   // UB
    }
}
```

JAK SOBIE PORADZIĆ Z TAKIM PROBLEMEM?

Odpowiedzią może być dynamicznie alokowana pamięć.

Najprościej jest to osiągnąć używając biblioteki `#include <memory>`, która posiada `std::shared_ptr<T>`.

Wskaźnik ten nie bez powodu nazywany jest *inteligentnym*. Odpowiada on za zarządzanie dynamiczną pamięcią i sam zwalnia zasób, gdy już go nie potrzebujemy.

JAK TAKI WSKAŹNIK UTWORZYĆ?

```
auto ptr = std::make_shared<int>(5);  // preferred
auto ptr = std::shared_ptr<int>(new int{5});
```

POPRAWIONY LISTING

```
std::vector<std::shared_ptr<int>> vec; // previously: std::vector<int*> vec;

void createAndAddToVec(int amount) {
    for (int i = 0 ; i < amount ; ++i) {
        vec.push_back(std::make_shared<int>(i));
        // previously: vec.push_back(&i);

        // the same in 2 lines:
        // auto num = std::make_shared<int>(i);
        // vec.push_back(num);
    }
}

int main() {
    createAndAddToVec(5);
    for (const auto& el : vec) {
        std::cout << *el << '\n';
    }
}
```

ZADANIE

Napisz funkcję `foo()`. Ma ona przyjmować shared_ptr na int i ma przypisać wartość 20 do wskazywanego przez niego obiektu.

Ponadto `foo()` ma wyświetlić wartość inta wskazywanego przez wskaźnik oraz liczbę shared_ptrów, które wskazują na ten obiekt.

Wyświetl także to samo w `main()` przed i po zawołaniu `foo()`. Pobierz zadanie

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

// TODO: Implement foo()
// It should take shared_ptr to int and assign value 20 to the pointed int.
// It should also display the value of this int and the number of how many pointers are poin
// Display the same information in main() before and after calling foo()

int main() {
    std::shared_ptr<int> number = std::make_shared<int>(10);
    // display the value under number pointer and use_count() of it
    foo(number);
    // display the value under number pointer and use_count() of it

    return 0;
}
```

INTELIGENTNE WSKAŹNIKI ROZWIĄZANIEM WSZYSTKICH PROBLEMÓW?

Teraz po utworzeniu inteligentnego wskaźnika, nie musimy się martwić o czas życia zmiennej. Możemy spokojnie po wyjściu z funkcji wypisać te wartości.

Jeżeli funkcja potrzebuje przyjąć zwykły wskaźnik (ang. raw pointer), czyli np. `int* i` możemy to zrobić używając funkcji `std::shared_ptr::get()` jak na przykładzie:

```
void foo(int* num) {
    do_sth(num);
}

int main() {
    auto ptr = std::make_shared<int>(5);
    foo(ptr.get())
}
```

PUŁAPKA POWRACA

```
void foo(int* num) {
    if (num) {
        do_sth(num);
    }
}

int main() {
    auto ptr = std::make_shared<int>(5);
    int* raw = ptr.get();
    ptr.reset();    // delete object, deallocate memory
    foo(raw);       // problem, dangling pointer is passed
    foo(ptr.get()); // not a problem, nullptr is passed
}
```

Jeżeli wszystkie obiekty `shared_ptr<T>` odwołujące się do tej zmiennej zostaną usunięte, to zasób zostanie zwolniony.

Nasz zwykły wskaźnik, który pobraliśmy wcześniej za pomocą `get()`, będzie posiadał adres do nieistniejącego już zasobu.

Próba jego użycia spowoduje UB lub crash. Należy bardzo uważać na zwykłe wskaźniki.

WNIOSKI

- wskaźniki mogą nie wskazywać na nic (`nullptr`), referencje muszą wskazywać na jakiś wcześniej stworzony obiekt
- wskaźniki i referencje mogą być niebezpieczne (częściej wskaźniki), jeśli są powiązane z nieistniejącymi już obiektami
  - są to tzw. dangling pointers/references, wiszące wskaźniki/referencje
- referencji nie można przypisać innego obiektu niż ten podany podczas jej inicjalizacji
- wskaźnikom można przypisać nowe adresy, aby wskazywały inne obiekty (za wyjątkiem stałych wskaźników)
- lepiej domyślnie nie używać zwykłych wskaźników (raw pointers)
- lepiej stosować inteligentne wskaźniki

ZADANIE

Napisz funkcję `calculateProduct()`. Ma ona przyjąć 2 wartości typu `int` oraz zwrócić ich iloczyn jako `shared_ptr`. Sprawdź ilu właścicieli posiada `shared_ptr`. Pobierz zadanie

```
#include <iostream>

// TODO: Implement calculateProduct()
// It should take 2 int values and return their product as a shared_ptr.
// Additionally, check how many owners are there.

int main() {
    auto number = calculateProduct(10, 20);
    std::cout << "num: " << *number << " | owners: " << number.use_count() <<

    return 0;
}
```

CODERS SCHOOL