

# PODSTAWY C++



CODERS  
SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBRÓŃ

# AGENDA

1. Typy danych
2. Funkcje
3. Instrukcje warunkowe
4. Pętle
5. Tablice

# ZADANIA

Repo GH `coders-school/kurs_cpp_podstawowy`

[https://github.com/coders-school/kurs\\_cpp\\_podstawowy/tree/master/module1](https://github.com/coders-school/kurs_cpp_podstawowy/tree/master/module1)

# PODSTAWY C++

## TYPY DANYCH



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: 1 bajt
  - `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 jakiejś dodatniej wartości.

Rozmiar typów logicznych i 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](#)

# 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;
```

# 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
```

# ROZMIARY TYPÓW

Standard C++ definiuje taką zależność pomiędzy rozmiarami typów całkowitoliczbowych

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

# 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`

# 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++

## FUNKCJE



CODERS  
SCHOOL

# FUNKCJE

Funkcja jest to fragment programu, któremu nadano nazwę i który możemy wykonać poprzez podanie jego nazwy oraz ewentualnych argumentów.

Funkcja == podprogram == procedura

Przykładowo, w trakcie jazdy na rowerze naszą główną funkcją jest przemieszczanie się z punktu a do b. Jednak wykonujemy także kilka podprogramów, jak zmiana biegów, hamowanie, rozpędzanie, skręcanie. Podobnie w programie możemy wydzielić konkretne zachowania i przenieść je do funkcji, które nazwiemy tak, by sugerowały co robią. Ważne, aby funkcja robiła tylko jedną rzecz. Jedna funkcja zmienia biegi, druga hamuje, trzecia skręca.

# SYGNATURY FUNKCJI (DEKLARACJE)

`void fun(int)` - funkcja ma nazwę `fun`, nic nie zwraca a przyjmuje jeden argument typu `int`.

## ODGADNIJCIE SYGNATURY PO OPISIE

Funkcja o nazwie `foo`, która nic nie zwraca a przyjmuje jeden argument typu `double`.

```
void foo(double)
```

Funkcja o nazwie `bar`, która zwraca typ `double` a przyjmuje 2 argumenty. Pierwszy to `float`, a drugi to `const int` (`const` oznacza, że wartość ta nie może zostać zmodyfikowana).

```
double bar(float, const int)
```

# WYWOŁANIA FUNKCJI

`foo(5.0)` -> wywołujemy funkcję `foo` z argumentem `double`, który jest równy `5.0`

`double result = bar(5.4f, 10)` -> wywołujemy funkcję `bar` z argumentem `float (5.4f)` oraz `int (10)` a jej wynik przypisujemy do zmiennej typu `double` o nazwie `result`.

# ZADANIE

Dopisz brakującą funkcję `multiply`. Ma ona pomnożyć dwie liczby podane jako jej parametry. [Pobierz zadanie](#)

```
#include <iostream>

// Write missing function here

int main() {
    std::cout << "4 * 5: " << multiply(4, 5) << "\n";
    std::cout << "10 * 5: " << multiply(10, 5) << "\n";
    std::cout << "-5 * 5: " << multiply(-5, 5) << "\n";

    return 0;
}
```

# PODSTAWY C++

## INSTRUKCJE WARUNKOWE



CODERS  
SCHOOL

# INSTRUKCJA `if`

Instrukcja warunkowa to nic innego jak zadanie programowi pytania np.:

- Czy otrzymałeś już wszystkie dane?
- Czy życie bossa spadło do 0?
- Czy osiągnięcie zostało zdobyte przez gracza?
- Czy liczba jest większa od maksymalnie dopuszczanej?



# KONSTRUKCJA `if`

```
if (condition) {  
    // do sth  
}
```

# ŁĄCZENIE WARUNKÓW

A co w przypadku, gdy wiele informacji musi być spełnionych? Możemy połączyć warunki operatorem **lub** (`||`, `or`) bądź **i** (`&&`, `and`)

```
if (are_potatoes_eaten && is_meat_eaten && is_salad_eaten)
```

Wszystkie 3 warunki muszą zostać spełnione

```
if (player_has_20_dex || player_has_18_int || player_has_22_str)
```

W tym przypadku wystarczy spełnić jeden z 3 warunków. Mogą zostać spełnione wszystkie, ale wystarczy by został spełniony jeden dowolny.

# INSTRUKCJA `else`

Jeżeli program może różnie zareagować na spełnienie jakiś warunków możemy zastosować konstrukcje `if else`

```
if (number < 2) {  
    critical_miss();  
} else if (number < 18) {  
    hit();  
} else {  
    critical_hit();  
}
```

# INSTRUKCJA `switch/case`

```
char option = getInput();
switch (option) {
case 'l':
    goLeft();
    break;
case 'r':
    goRight();
    break;
default:
    exit();
}
```

- `case` oznacza konkretny przypadek
- `break` informuje, że wychodzimy z instrukcji warunkowej `switch` i kontynuujemy dalej program. Jego brak spowoduje, że wykonają się instrukcje z kolejnego `case`.
- `default` jest to miejsce gdzie program dotrze, gdy żaden inny warunek nie zostanie spełniony

# ZADANIE

Dopisz funkcję `max`. Ma ona zwracać maksymalną z trzech podanych wartości. [Pobierz zadanie](#)

```
#include <iostream>

// Write your function here

int main() {
    std::cout << "max (1, 2, 3): " << max(1, 2, 3) << "\n";
    std::cout << "max (2, 3, 1): " << max(2, 3, 1) << "\n";
    std::cout << "max (3, 2, 1): " << max(3, 2, 1) << "\n";

    return 0;
}
```

# PODSTAWY C++

## PĘTLE



CODERS  
SCHOOL

# PĘTLE

Pętla służy do powtarzania instrukcji, które chcemy by się wykonały więcej niż raz bez konieczności ich wielokrotnego pisania w kodzie.

Podstawowe pętle: `while`, `for`

# PĘTLA `while`

`while` używamy, gdy chcemy coś wykonać dopóki nie zostanie spełniony jakiś warunek. Przeważnie nie mamy pojęcia, kiedy to następy (nie znamy liczby kroków) np:

- Przeglądamy koszule w Internecie dopóki nie znajdziemy pasującej do nas
- Powtarzamy walkę z tym samym bossem aż go nie pokonamy
- Jemy zupę, aż talerz nie będzie pusty
- Przeszukujemy kontakty w telefonie aż nie znajdziemy interesującej nas osoby



# KONSTRUKCJA PĘTLI `while`

```
while (condition) {  
    // Do sth  
}
```

## PRZYKŁAD

```
while (a == b) {  
    std::cin >> a;  
    std::cin >> b;  
}
```

# PĘTLA `for`

`for` używamy, gdy chcemy coś wykonać określoną liczbę razy. Przeważnie znamy liczbę kroków np:

- Wypełniamy ankietę składającą się z 10 pytań -> liczba kroków 10
- Przemieszczamy się z punktu A do B -> liczba kroków = dystans / długość kroku
- Piszemy egzamin składający się z 4 zadań -> liczba kroków (jak umiemy to 4, jak nie to jeszcze wykonujemy podprogram `ściągaj`)
- Zapinamy koszule (o ile nie wyrwiemy żadnego guzika)

# KONSTRUKCJA PĘTLI `for`

```
for (variable = initial_value; condition; variable_change) {  
    // Do sth  
}
```

## PRZYKŁAD

```
for (size_t i = 0 ; i < 10 ; i+=2) {  
    std::cout << "i: " << i << '\n';  
}
```

Każdą pętlę `for` można zamienić na `while` i odwrotnie. Wybieramy wygodniejszy dla nas zapis, zazwyczaj w zależności od znajomości liczby kroków.

Istnieje jeszcze jeden rodzaj pętli. Jaki?

# PĘTLA `do/while`

```
do {  
    // Do sth  
} while(condition)
```

Kod w pętlach `while` lub `for` może się nie wykonać ani razu, gdy warunek nie będzie nigdy spełniony.

Kod w pętli `do/while` wykona się co najmniej raz.

# ZADANIE

Dopisz funkcję `printString`. Ma ona wypisywać tekst podany jako pierwszy argument tyle razy, jaka jest wartość liczby podanej jako drugi argument. [Pobierz zadanie](#)

```
#include <iostream>

// Write your function here

int main() {
    printString("Hello", 5);
    std::cout << "\n";

    printString("AbC", 2);
    std::cout << "\n";

    printString("HiHi ", 6);
    std::cout << "\n";

    return 0;
}
```

# PODSTAWY C++

## TABLICE



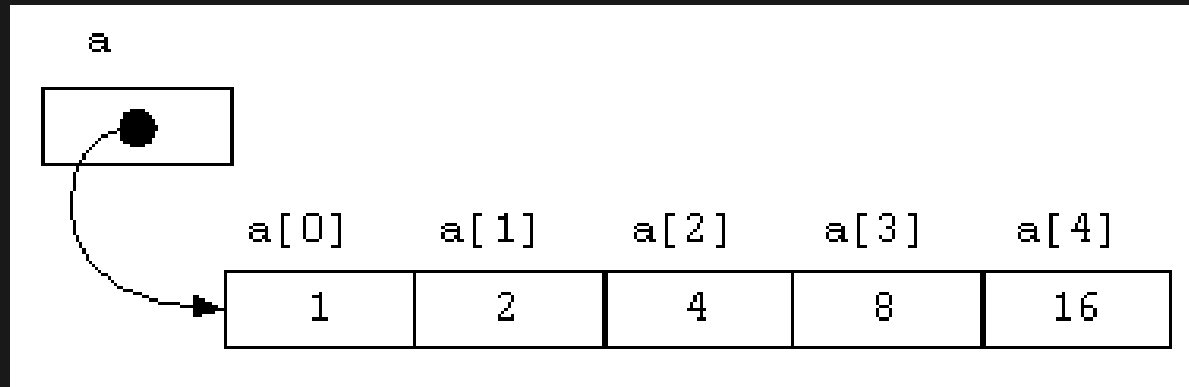
CODERS  
SCHOOL

# WPROWADZENIE TO 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-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++

## PODSUMOWANIE



CODERS  
SCHOOL

# CO PAMIĘTASZ Z DZISIAJ?

## NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

### 1. Typy danych

- `void, bool, char, int, double` + ich odmiany

### 2. Funkcje

- sygnatura (deklaracja) = typ zwracany, nazwa, argumenty

### 3. Instrukcje warunkowe

- `if, switch/case`

### 4. Pętle

- `for, while, do/while`

### 5. Tablice

- `Type t[N], operator[ ]`

# PRACA DOMOWA

## POST-WORK

- Poczytaj dokumentację `std::string`. Znajdziesz tam m.in. opis funkcji `std::to_string`. Przyda się :)
- Zadanie 1 - Calculate (5 punktów)
- Zadanie 2 - Fibonacci - rekurencja i iteracja (6 punktów)

## BONUS ZA PUNKTUALNOŚĆ

Za dostarczenie każdego zadania przed 24.05.2020 (niedziela) do 23:59 dostaniesz 2 bonusowe punkty (razem 4 punkty za 2 zadania).

## PRE-WORK

- Poczytaj dokumentację typu `std::vector`. Poklikaj na różne funkcje i patrz głównie na przykłady użycia na samym dole ston.
- Możesz przyjrzeć się plikom z testami w zadaniach i spróbować dopisać własne przypadki testowe

## ZADANIA W REPO

# ZADANIE 1 - CALCULATE

Zaimplementuj funkcję, której zadaniem ma być wykonywanie działań arytmetycznych na dwóch liczbach.

Sygnatura - `std::string calculate(const std::string& command, int first, int second)`.

## PARAMETRY

- `const std::string& command` - rodzaj działania. Jedno z `add`, `subtract`, `multiply`, `divide`
- `int first` - pierwsza liczba
- `int second` - druga liczba

## WARTOŚĆ ZWRACANA

- `std::string` - wynik działania jako tekst

W przypadku podania błędnego parametru `command` funkcja powinna zwrócić napis "Invalid data".

## PRZYKŁADY UŻYCIA

```
auto result = calculate("add", 2, 3); // result = "5"
result = calculate("multiply", 2, 3); // result = "6"
result = calculate("hello", 2, 3);    // result = "Invalid data"
```



# ZADANIE 2 - FIBONACCI

Zaimplementuj dwie funkcje. Obie mają liczyć n-tą liczbę **ciągu Fibonacciego**, ale na odmienne sposoby.

- iteracyjnie (z użyciem pętli)
- rekurencyjnie (funkcja ma wołać samą siebie)



Funkcje muszą mieć określone sygnatury:

```
int fibonacci_iterative(int sequence);  
int fibonacci_recursive(int sequence);
```

# DOSTARCZENIE ZADAŃ

1. Zrób fork repo [kurs\\_cpp\\_podstawowy](#)
2. Ściągnij swój fork - `git clone https://github.com/YOURNICK/kurs_cpp_podstawowy.git`
3. Przejdź do katalogu kurs\_cpp\_podstawowy - `cd kurs_cpp_podstawowy`
4. Utwórz gałąź o nazwie `calculate` na rozwiązanie zadania `calculate` - `git checkout -b calculate`
5. Przejdź do katalogu `homework/calculate` - `cd homework/calculate`
6. Tutaj znajduje się szkielet programu, który musisz wypełnić. Szkielet programu zawiera już testy, które sprawdzają, czy Twoja implementacja jest poprawna. Zanim rozpoczniesz implementację wpisz następujące zaklęcia:

```
mkdir build    # tworzy katalog build
cd build      # przechodzi do katalogu build
cmake ..      # generuje system budowania wg przepisu z pliku ../CMakeLists.txt
make          # kompiluje
ctest -V      # odpala testy
```

7. Zaimplementuj funkcjonalność (najlepiej po kawałku, np. zacznij od samego dodawania)
8. Sprawdź, czy implementacja przechodzi testy - `make` (kompilacja) oraz `ctest -V` (uruchomienie testów)
9. Zrób commit z opisem działającej funkcjonalności - `git commit -am"adding works"`
10. Wróć do punktu 7 i zaimplementuj kolejny kawałek. Jeśli rozwiązanie przechodzi wszystkie testy przejdź do kolejnego punktu
11. Wypchnij zmiany do swojego forka - `git push origin calculate`
12. Wyklikaj Pull Request na GitHubie.
13. Poczekaj chwilę na raport Continuous Integration (CI), aby sprawdzić, czy rozwiązanie kompiluje się i przechodzi testy także na GitHubie.
14. Jeśli jest  - brawo, rozwiązanie jest poprawne. Jeśli jest  kliknij na niego i sprawdź opis błędu. Popraw go (punkty 7-11) i poczekaj na kolejny raport CI.

# DOSTARCZENIE KOLEJNYCH ZADAŃ

Najpierw wróć na gałąź główną - `git checkout master` i postępuj od kroku 4 dla kolejnego zadania (stworzenie nowej gałęzi o innej nazwie)

Możesz zaobserwować, że przełączenie się na inną gałąź spowodowało, że nie masz rozwiązania pierwszego zadania. Spokojnie, jest ono po prostu na innej gałęzi. Możesz do niego wrócić przechodząc na gałąź tego zadania - `git checkout nazwa`.

# CODERS SCHOOL

