

C++ i inne zbrodnie Hajdukiewicza

CZYLI OSTATNI RATUNEK PO ZDALNYCH...

Oliwier Popielarczyk, kl. 2A SCI

Spis treści

1. Typy zmiennych i ich zasięgi, budowa kodu w c++
2. If, for, while, dowhile, switch
3. Tablice, tablice wielowymiarowe
4. Vectory
5. Funkcje w c++
6. Przydatne biblioteki w c++
7. Biblioteka math.h – zaawansowane obliczenia matematyczne
8. Biblioteki stdlib.h i time.h – liczby pseudolosowe
9. Biblioteka string – operacje na zmiennych tekstowych
10. Operacje na plikach – strumienie
11. Klasy w c++
12. Klasy wirtualne
13. Przydatne polecenia i metody bibliotek

Dział 1 – Typy zmiennych i ich zasięgi, budowa kodu w c++

W c++ mamy kilka rodzajów zmiennych

Bool – zmienna *true/false* (lub 1/0 – przyjmuje oba sposoby)

Int – zmienna z liczbą całkowitą

Float i *Double* – zmienne zmiennoprzecinkowe (czyli z ułamkami)

Char – jakiś znak np. '+', określa go liczba w kodzie ASCII

String – jakiś wyraz, zdanie, lub dłuższy tekst. Program rozumie go jako ciąg pojedynczych znaków, czyli char'ów.

Na screenie poniżej obczajcie sobie te zmienne w visualu:

```
#include <iostream>

int main()
{
    bool prawda = 1;
    bool fałsz = false; //przyjmuje oba sposoby zapisu
    int zmienna1 = 1;
    double zmienna2 = 1.234;
    float zmienna3 = 5.678;
    char znak = '+'; // apostrof, nie cudzysłów jak w stringu
    std::string zmienna4 = "ciąg jakiś znaków";
    // zauważ, że w stringu potrzeba std, i ma zielony kolor czcionki
}
```

PAMIĘTAJ: chary nie tylko są rozumiane przez program jako kod ASCII. Możesz samemu użyć tego kodu, aby utworzyć zmienną.

```
char znak2 = 63;
```

Program zrozumie liczbę 63, jako znak zapytania, według kodowania ASCII.

Zasięgi zmiennych

Pamiętaj, że każda zmienna ma swój zasięg „istnienia”. W tym zasięgu możesz jej używać. Przykładowe zasięgi zmiennych na screenie poniżej:

```
#include <iostream>

int zmienna_globalna = 11; // ta zmienna może być używana wszędzie, ma GLOBALNY ZASIĘG

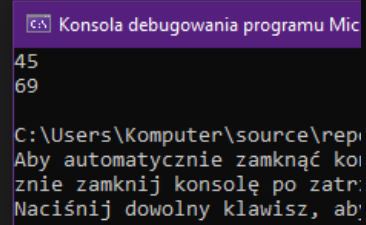
int funkcja(int a)
{
    int zmienna_funkcja = 15; // ta zmienna może być używana tylko w tej funkcji
    return a;
}

int main()
{
    int zmienna_main = 32; // ta zmienna może być używana tylko w funkcji main, I GŁÓWNEJ
    while (true)
    {
        int zmienna_petla = 45; //ta zmienna użyjesz tylko w tej petli,
    }
}
```

Skoro zmienna np. wewnątrz pętli działa tylko w tym jednym miejscu, to nic nie stoi na przeszkodzie, aby w dwóch różnych pętlach utworzyć zmienną o tej samej nazwie ;DD

```
int main()
{
    while (true)
    {
        int zmienna_petla = 45; //ta zmienna użyjesz tylko w tej petli,
        std::cout << zmienna_petla << std::endl;
        break; //zatrzymanie petli
    }

    while (true)
    {
        int zmienna_petla = 69; // nie ma błędów :DD
        std::cout << zmienna_petla << std::endl;
        break;
    }
}
```

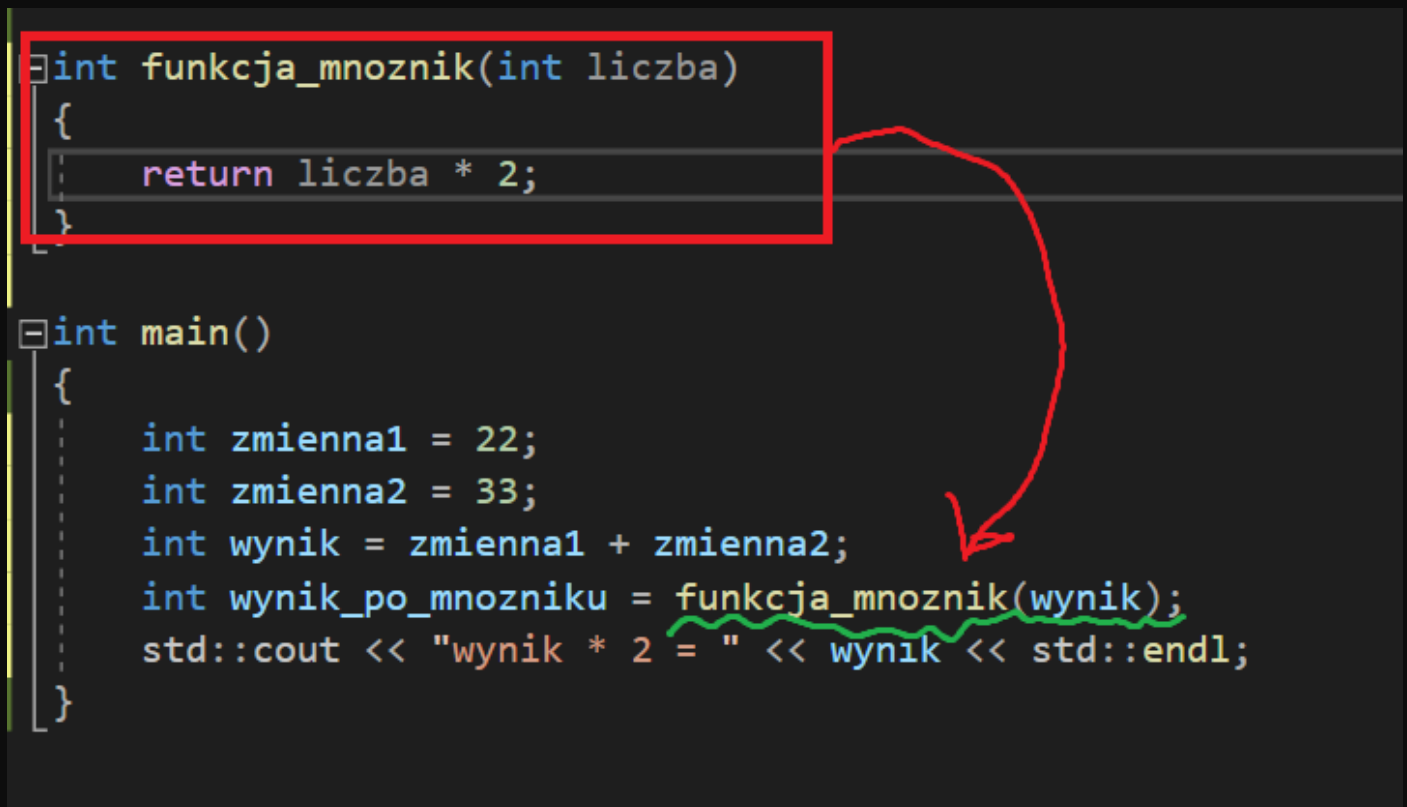


Jednak nie zaleca się takiego działania – kod staje się mniej czytelny.

Pamiętajcie po prostu że jest taka możliwość.

Budowa kodu w C++

Budowa kodu jest oparta na funkcjach, których zawartości są w klamrach {}. Główną i najważniejszą funkcją jest *Main()*. W dalszej części dowiesz się, że każda funkcja musi zwracać jakąś wartość (tak jak w matematyce). Funkcja *Main()* to wyjątek, nie zwraca ona żadnej wartości, nie posiada więc ona na końcu polecenia „*return*”, ale jeżeli chcesz, to możesz użyć „*return 0;*”. Program czyta kod domyślnie tylko z funkcji *Main()*, z góry do dołu. Inne funkcje program czyta dopiero po ich wywołaniu:



```
int funkcja_mnoznik(int liczba)
{
    return liczba * 2;
}

int main()
{
    int zmienna1 = 22;
    int zmienna2 = 33;
    int wynik = zmienna1 + zmienna2;
    int wynik_po_mnozniku = funkcja_mnoznik(wynik);
    std::cout << "wynik * 2 = " << wynik << std::endl;
}
```

Początkowo program pominie *funkcje_mnoznik*, a zacznie od *Main()*. Dopiero gdy zauważy wywołanie (zaznaczone na zielono), wykona polecenia z funkcji. Innymi słowy – polecenia z *funkcji_mnoznik* wpierdolą się w środek.

Dodatkowo: informacje po podwójnym slashu // są ignorowane przez program, dlatego są zaznaczone na zielono.

Dział 2 – If, for, while, dowhile, switch

Są to tak zwane instrukcje warunkowe. Do ich użycia potrzeba dodatkowych informacji. Zaczniemy od najprostszej instrukcji, czyli *if*

```
int main()
{
    int zmienna1 = 12;
    int zmienna2 = 12;
    if (zmienna1 == zmienna2) //uwazaj na podwojny znak =, sluzy do "porownywania" dwóch zmiennych
    {
        //jezeli rownanie w nawiasie jest prawda, to wykona sie czesc kodu w tym nawiasie
        //jezeli jest falszem, to ta czesc zostanie pominieta.
    }
}
```

Instrukcję warunkową *if* możemy rozszerzyć o polecenia „*else*”, i „*elseif*”. Pierwszego używamy, gdy chcemy aby dana część kodu została wykonana WTEDY I TYLKO WTEDY, gdy nasz warunek okaże się fałszywy. Przykład poniżej:

```
int main()
{
    int zmienna1 = 12;
    int zmienna2 = 12;
    if (zmienna1 == zmienna2) //uwazaj na podwojny znak =, sluzy do "porownywania" dwóch zmiennych
    {
        //jezeli rownanie w nawiasie jest prawda, to wykona sie czesc kodu w tym nawiasie
        //jezeli jest falszem, to ta czesc zostanie pominieta.
    }
    else
    {
        //jezeli warunek jest falszywy, wykona sie ta czesc kodu
        //jezeli warunek jest prawdziwy, to program to pominie
    }
}
```

Elseif służy do sprawdzania kilku możliwości pod rząd

Skoro pierwszy argument jest fałszywy, to może drugi będzie prawdziwy, a jak nie to wykonamy polecenia z else

```
int main()
{
    int zmienna1 = 12;
    int zmienna2 = 12;
    if (zmienna1 == zmienna2) //uważaj na podwojny znak =, służy do "porównywania" dwóch zmiennych
    {
        //jeżeli równanie w nawiasie jest prawdą, to wykona się część kodu w tym nawiasie
        //jeżeli jest fałszem, to ta część zostanie pominięta.
    }
    else if (zmienna1 > zmienna2)
    {
        //jeżeli warunek z pierwszego if to fałsz, program sprawdzi też ten warunek
        //jeżeli pierwsze if to prawda, to program to pominie
    }
    else
    {
        //jeżeli warunek jest fałszywy, wykona się ta część kodu
        //jeżeli warunek jest prawdziwy, to program to pominie
    }
}
```

Teraz pora na dwie pierwsze pętle – *while*, oraz bardzo podobną *dowhile*. Polecenia w pętli będą wykonywane w kółko dopóki warunek pętli jest prawdą. Warunek będzie sprawdzany:

- przed każdym powtórzeniem poleceń pętli – w przypadku pętli *while*
- po każdym wykonaniu poleceń pętli – w przypadku pętli *dowhile*

Dzięki temu polecenia znajdujące się w pętli *dowhile* zostaną wykonane PRZYNAJMNIEJ raz, przed pierwszym sprawdzeniem.

Poniżej przykłady tych pętli:

```
int main()
{
    bool pytanie = false;
    while (pytanie == false) //petla while
    {
        std::cout << "zakonczyć? 1 - tak, 0 - nie" << std::endl;
        std::cin >> pytanie;
    }
}
```

```

int main()
{
    bool pytanie = false;
    do //petla dowhile, poczatek bez warunku
    {
        std::cout << "zakonczyć? 1 - tak, 0 - nie" << std::endl;
        std::cin >> pytanie;
    } while (pytanie == false); //tutaj warunek, PAMIĘTAJ O ŚREDNIKU NA KONCU
}

```

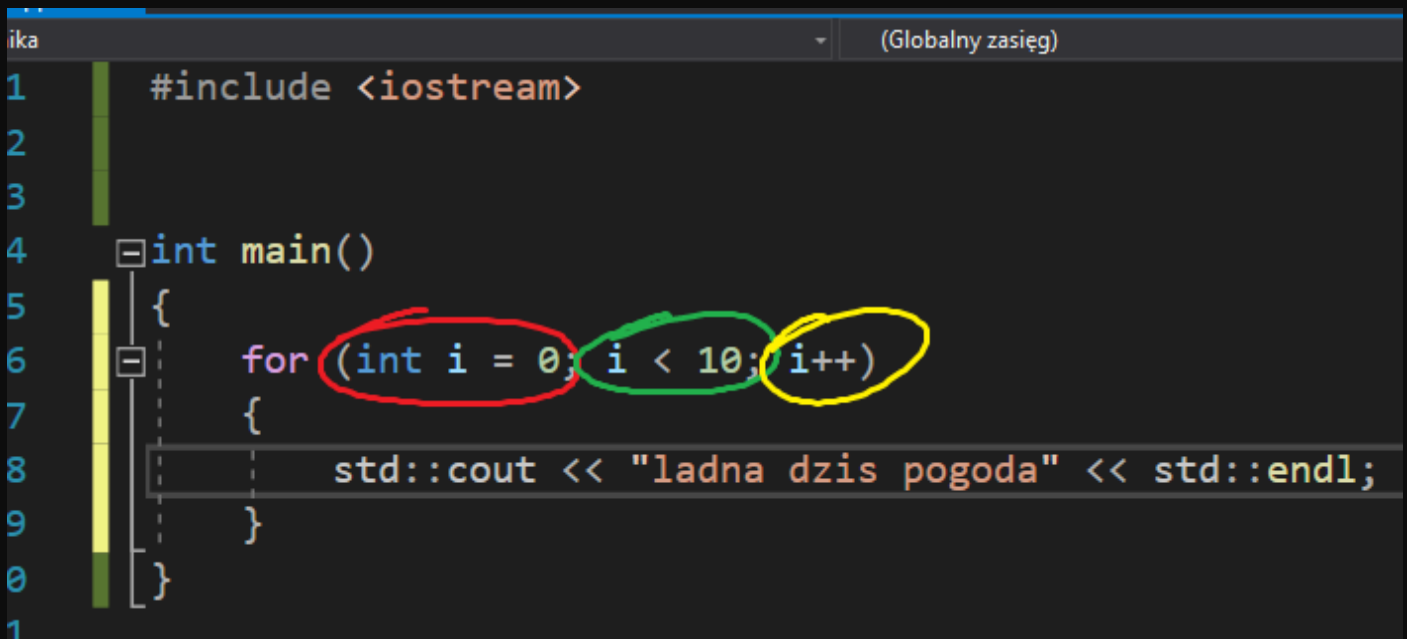
Instrukcja *switch* jest bardzo podobna do instrukcji *if*. Porównuje ona dany element do kilku innych. Jeżeli w jakimś porównaniu okaże się prawdą, to zostaną wykonane polecenia z danego *case'u*. Składnia na screenie poniżej

```

int main()
{
    std::cout << "ile to pierwiastek ze 144" << std::endl;
    int zmienna1;
    std::cin >> zmienna1;
    switch (zmienna1)
    {
        case 12:
            std::cout << "dobrze!";
            break;
        case -12:
            std::cout << "dobrze!";
            break;
        default: //kod wykona się gdy żadne poprzednie warunki nie zostaną spełnione
            std::cout << "zle!";
            break;
    }
}

```


Pętla *for* to najbardziej zaawansowana pętla, daje nam najwięcej możliwości.



```
1 #include <iostream>
2
3
4 int main()
5 {
6     for (int i = 0; i < 10; i++)
7     {
8         std::cout << "ładna dziś pogoda" << std::endl;
9     }
10 }
```

The screenshot shows a C++ program in Visual Studio. The code defines a `main` function containing a `for` loop. The loop's initialization `int i = 0` is circled in red, the condition `i < 10` is circled in green, and the increment `i++` is circled in yellow. The loop body prints the string "ładna dziś pogoda" followed by a newline character.

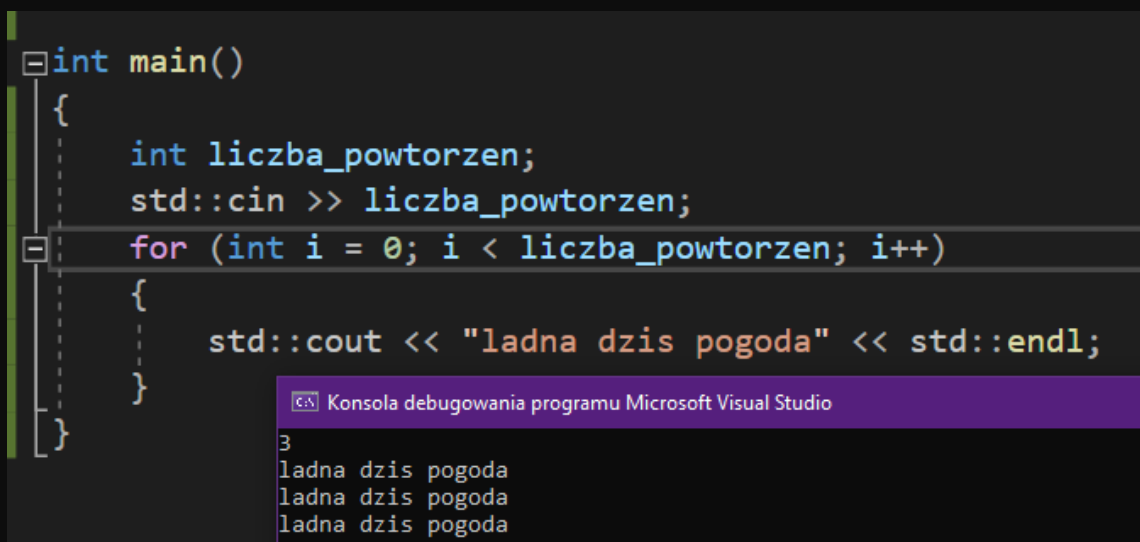
- Czerwonym zaznaczono tworzenie zmiennej które następuje na początku działania pętli. Początkowo i przyjmuje wartość 0

- Informacja w żółtym kółku mówi że co każde „powtórzenie” wartość zmiennej i wzrośnie o 1 ($i++$ to to samo co $i = i + 1$)

- Informacja w zielonym kółku mówi o tym kiedy pętla zakończy swoje działanie – będzie działać dopóki spełniany jest ten warunek, czyli dopóki i nie będzie równe 10.

Koniec końców pętla powtórzy kod ze swojego wnętrza 10 razy.

Poniżej przypadek w którym to użytkownik podaje ilość powtórzeń:



```
int main()
{
    int liczba_powtorzen;
    std::cin >> liczba_powtorzen;
    for (int i = 0; i < liczba_powtorzen; i++)
    {
        std::cout << "ładna dziś pogoda" << std::endl;
    }
}
```

The screenshot shows a C++ program that takes an integer input from the user and uses it as the upper bound for a `for` loop. The loop prints the string "ładna dziś pogoda" repeatedly. Below the code, the debug console shows the output of the program, which is "ładna dziś pogoda" printed three times, indicating that the user input was 3.

Dział 3 – tablice, tablice wielowymiarowe.

Założmy taki scenariusz – masz zadanie bojowe utworzyć 15 zmiennych i nadać im jakieś wartości całkowite. Pisanie po kolei...

```
int zmienna1 = 45;
```

```
int zmienna2 = 69;
```

```
int zmienna3 = 21;
```

...jest dość czasochłonne i męczące. I wtedy wchodzi tablica, czyli grupa jakiś zmiennych. Aby utworzyć tablicę jakiś zmiennych należy napisać:

```
Typ_danych nazwa_tablicy[wielkośc_tablicy]
```

Czyli wracając do naszego zadania bojowego, aby utworzyć tablice 15 wartości typu *int* należy napisać:

```
Int fajna_tablica[15];
```

Teraz aby móc nadać każdemu elementowi wartość, używamy takiej składni:

```
Fajna_tablica[2] = 69;
```

UWAGA: tablica ma 15 wartości, ale ich „numery” zaczynają się od zera

Czyli elementy nie mają numerów od 1 do 15, tylko od 0 do 14

```
Fajna_tablica[15] = 645
```

 wyrzuci błąd, bo nie ma takiego elementu.

Teraz założmy inny scenariusz: musisz wprowadzić dane 20 uczniów - ich imiona i nazwiska – w osobnych zmiennych.

Można wykonać dwie tablice *string*ów, jedną na imiona, drugą na nazwiska, ale prościej jest wykonać tablice wielowymiarową. W naszym wypadku będzie to tablica dwuwymiarowa. Jak wykonać takie zadanie?

```

int main()
{
    //skladnia - typ_danych nazwa_tablicy[ilosc_elementow] [ilosc_elementow];
    //czyli u nas to wyglada tak:
    std::string tablica[20][2];
}

```

Teraz wystarczy wykonać pętle i po kolei wprowadzać dane do tablicy. Tablice możemy porównać do takich tabel. Pierwsza wartość to ilość wierszy, druga to ilość kolumn. Możemy wykonywać też tablice o większej ilości wymiarów, np. 3 lub 4, tablice te jednak są o wiele bardziej złożone i zaawansowane. 4-wymiarowa tablica, w której każdy wymiar ma po 3 wartości – ma w sumie 81 elementów. Poniżej wywołanie takiej tablicy, i pętle potrzebne do jej wypełnienia

```

int main()
{
    int tablica_4_wymiary[3][3][3][3];
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            for (int o = 0; o < 3; o++)
            {
                for (int p = 0; p < 3; p++)
                {
                    std::cin >> tablica_4_wymiary[i][j][o][p];
                }
            }
        }
    }
}

```

Dział 4 – vectory

Vector są bardzo podobne do tablic – grupują nam zmienne danego typu. Różnica jest taka, że rozmiaru tablicy NIE ZMIENIMY po jej utworzeniu. Przy tworzeniu *vectora* nie piszemy jego rozmiaru. Po prostu gdy chcemy dodać do niego jakiś element używamy metody *.push_back()*, a gdy chcemy znać rozmiar (ilość elementów) *vectora*, używamy metody *.size()*. Aby usunąć jakiś element używamy metody *.pop_back()*. Przykład wykorzystania poniżej:

```
(Globalny zasięg) main()
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> liczby; //utworzenie vectora
    bool dodajemy = true; //tworzymy warunek petli while
    int temp; //zmienna tymczasowa
    while (dodajemy == true)
    {
        std::cout << "dodaj cos do vectora" << std::endl;
        std::cin >> temp;
        liczby.push_back(temp); //dodawanie do vectora wartosci ze zmiennej tymczasowej
        std::cout << "chcesz dodac wiecej? 1 - tak, 0 - nie" << std::endl;
        std::cin >> dodajemy;
    } //jezeli ktos wybral 0 to petla sie konczy
    temp = liczby.size();
    liczby.pop_back(); //usuniecie ostatniego elementu
}
```

UWAGA: PAMIĘTAJCIE ABY NA GÓRZE ZAŁĄCZYĆ BIBLIOTEKĘ VECTOR:
#include <vector>

VECTORY I TABLICE ZWANE SĄ TEŻ KONTENERAMI NA DANE

Dział 5 – funkcje w C++

Funkcje tworzymy głównie aby skrócić kod i nie powtarzać tego samego fragmentu programu kilka razy. W zależności od typu funkcji (takie jak typy danych) zwraca ona jakąś wartość, np.

funkcja `int funkcja()` powinna zwrócić liczbę całkowitą

funkcja `char funkcja()` zwróci jakiś znak z kodowania ASCII

Funkcja może nie tylko zwracać wartość, może też ją przyjmować, informacje o tym jakie dane ta funkcja przyjmuje podajemy w nawiasie przy deklaracji.

```
#include <iostream>

int funkcja_pole_kwadratu(int a_funkcja) //w nawiasie typ wartosci jaka przyjmuje
{
    int wynik;
    wynik = a_funkcja * a_funkcja;
    return wynik; //zwracamy wartosc
}

int main()
{
    int a;
    std::cin >> a;
    std::cout << "pole kwadratu tej liczby wynosi " << funkcja_pole_kwadratu(a);
    //powyzej wykonanie funkcji z argumentem podanym w nawiasie
}
```

Z matematycznego punktu widzenia w nawiasie podajemy ARGUMENTY funkcji. Niekoniecznie musi być to jeden, może być tego więcej

PAMIETAJ: jeżeli jako argument funkcji użyjesz jakąś zmienną, np. `x`, to funkcja nie będzie operować na tej zmiennej, tylko na takiej jakby „kopii”, zwanej wskaźnikiem.

Funkcji możemy używać do tzw. rekurencji, czyli funkcja wywołuje sama siebie

Dział 6 – przydatne biblioteki

W c++ tylko małą część funkcjonalności dostajemy na początku. Wiele z nich dostaniemy po załączeniu odpowiednich bibliotek. Każda biblioteka dodaje coś nowego, nowe metody i polecenia.

Biblioteki załączamy za pomocą polecenia `#include <nazwa biblioteki>` na samej górze naszego programu. Przydatną biblioteką jest biblioteka `stdlib.h`, dająca dostęp do polecenia `system`. Za jego pomocą możemy wykorzystywać w aplikacjach konsolowych wiele poleceń znanych z cmd, takich jak *pause*, *cls* i *time*. Poniżej program który po wypisaniu zmiennych zaczeka aż coś klikniemy, i potem wyczyści ekran.

```
#include <iostream>
#include <stdlib.h>

int main()
{
    std::string zmienna = "iaojdaijaw";
    for (int i = 0; i < 20; i++)
    {
        std::cout << zmienna;
    }
    std::cout << std::endl << "kliknij cos aby wyczyszcic ekran" << std::endl;
    system("pause");
    system("cls");
}
```

Dział 7 - biblioteka *math.h* – zaawansowane obliczenia matematyczne

Biblioteka *math.h* to jedna z ważniejszych bibliotek w c++. Znacznie ułatwia nam ona obliczenia matematyczne, ponieważ odblokowuje ona nam takie działania jak potęgowanie, pierwiastkowanie, logarytmy, zaokrąglanie, a nawet podstawowe funkcje trygonometryczne (sinus, cosinus, tangens) Poniżej przykład wykorzystania niektórych z tych poleceń w programie do liczenia właściwości kwadratu o boku *a*

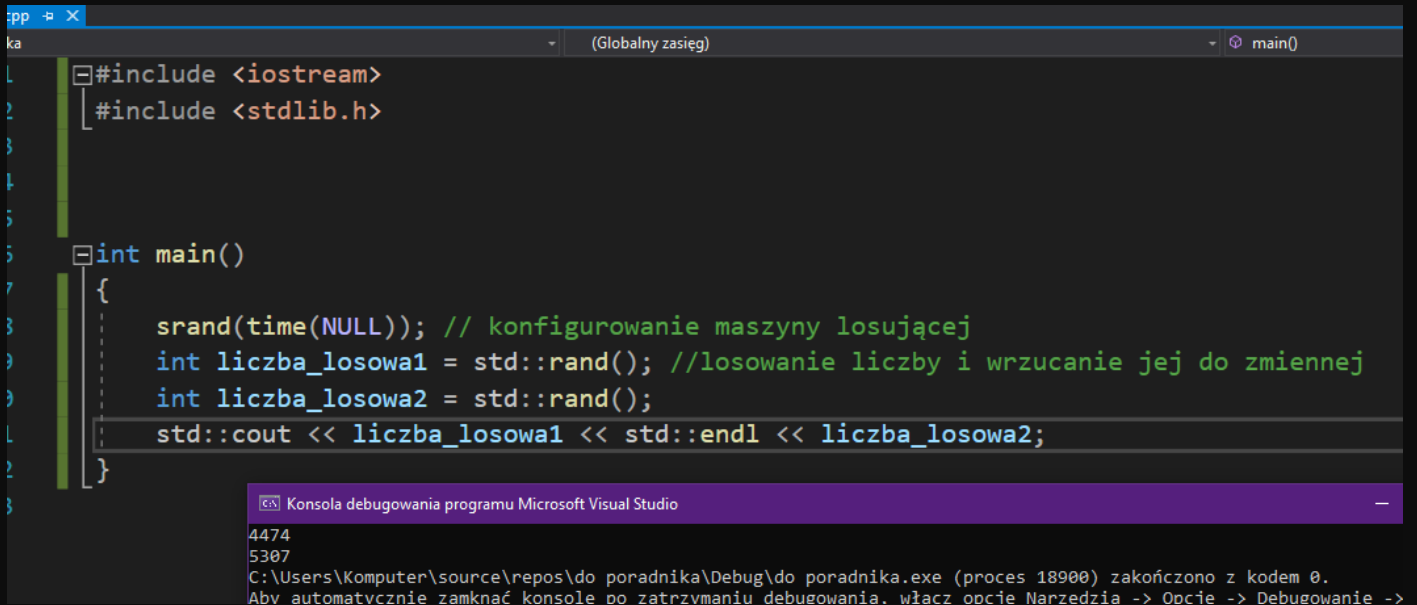
```
(Globalny zasięg) main()
#include <iostream>
#include <math.h>

int main()
{
    std::cout << "wprowadz bok a kwadratu" << std::endl;
    double a; //operujemy na double, aby mieć wartości po przecinku
    std::cin >> a;
    double pole = pow(a, 2);
    double przekatna = a * sqrt(2); //sqrt(2) to pierwiastek z dwóch
    std::cout << "pole kwadratu o tym boku wynosi " << pole << " a przekatna " << przekatna;
}
```

Dział 8 – biblioteki *stdlib.h* i *time.h* – liczby pseudolosowe

Dlaczego te liczby są PSEUDOlosowe? Ponieważ ich „losowość” polega jest oparta o czas systemowy.

Najpierw najprostsze – losowanie liczb bez dokładnego przedziału.



```
#include <iostream>
#include <stdlib.h>

int main()
{
    srand(time(NULL)); // konfigurowanie maszyny losującej
    int liczba_losowa1 = std::rand(); //losowanie liczby i wrzucanie jej do zmiennej
    int liczba_losowa2 = std::rand();
    std::cout << liczba_losowa1 << std::endl << liczba_losowa2;
}
```

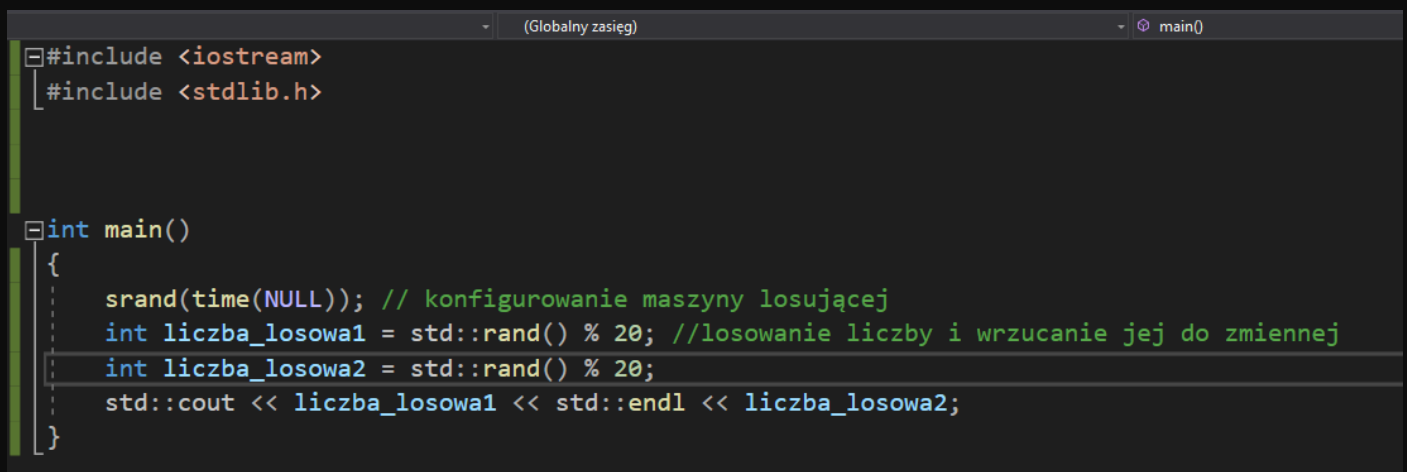
Konsola debugowania programu Microsoft Visual Studio

```
4474
5307
C:\Users\Komputer\source\repos\do poradnika\Debug\do poradnika.exe (proces 18900) zakończono z kodem 0.
Abv automatycznie zamknąć konsolę po zatrzymaniu debugowania. Włącz opcję Narzędzia -> Opcje -> Debugowanie ->
```

PAMIĘTAJ: maszynę konfigurujemy tylko raz, potem możemy brać nieskończenie wiele liczb.

Teraz trudniejsze – losowanie liczb z przedziału OD ZERA do X

Różnicy zbyt dużych nie ma, po prostu do losowania liczby dorzucamy na końcu działanie MODULO, które da nam resztę z dzielenia z liczby X . A jak wiemy, reszta z dzielenia z liczby X nie może być większa niż sama liczba X , więc tworzy nam to przedział od zera do $X - 1$;



```
#include <iostream>
#include <stdlib.h>

int main()
{
    srand(time(NULL)); // konfigurowanie maszyny losującej
    int liczba_losowa1 = std::rand() % 20; //losowanie liczby i wrzucanie jej do zmiennej
    int liczba_losowa2 = std::rand() % 20;
    std::cout << liczba_losowa1 << std::endl << liczba_losowa2;
}
```


Teraz najtrudniejsze – losowanie liczb z przedziału od X do Y

W tym wypadku również rozszerzamy polecenie losowania liczby.

Dotychczasowe „segmenty” po znaku = bierzemy w nawias, i po nawiasie dopisujemy liczbę od której mamy zacząć losować.

Wygląda to tak:

```
#include <iostream>
#include <stdlib.h>

int main()
{
    srand(time(NULL)); // konfigurowanie maszyny losującej
    int liczba_losowa1 = (std::rand() % 20) + 1; //losowanie liczby i wrzucanie jej do zmiennej
    int liczba_losowa2 = (std::rand() % 20) + 1;
    std::cout << liczba_losowa1 << std::endl << liczba_losowa2;
}
```

Teraz nasz program wylosuje coś z przedziału od 1 do 20, a nie od 0 do 19 jak poprzednio.

Dział 9 – biblioteka *string* – operacja na zmiennych tekstowych

Jak zdążyłaś/eś zauważyć - *string* sam w sobie – działa bez potrzeby dodatkowych bibliotek do kodu. Jednak można jego funkcjonalność rozszerzyć, załączając bibliotekę *string*. Daje nam ona wiele dodatkowych metod, przydatnych podczas operacji na takich plikach, takich jak *size()*, *clear()*, *compare()* i inne (pełna tabelka na końcu).

Przykład programu wykorzystującego metodę *compare()*, do sprawdzenia czy wpisujemy dwa razy to samo.

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

int main()
{
    std::string zmienna1, zmienna2; //ciekawostka: mozesz deklarowac kilka zmiennych na raz
    std::cout << "napisz cos" << std::endl;
    std::cin >> zmienna1;
    std::cout << "napisz to samo jeszcze raz :DD " << std::endl;
    std::cin >> zmienna2;
    if (zmienna1.compare(zmienna2) == true)
    {
        std::cout << "okej, wszystko dobrze" << std::endl;
    }
    else
    {
        std::cout << "OSZUKUJESZ! wrrr" << std::endl;
    }
}
```

Metodę tę możemy też wykorzystać na przykład gdy tworzymy część programu z rejestracją użytkownika, i prosimy o podanie hasła dwa razy. Możemy łatwo to sprawdzić :DD

EDIT 1: W instrukcji *if* w nawiasie powinno widnieć *false*.

Dział 10 – operacje na plikach, strumienie.

Program w c++ operują na tzw. Strumieniach. Sama nazwa standardowej biblioteki *IOSTREAM* wskazuje na to że dostajemy dwa strumienie – wejścia i wyjścia. Za kierowanie tymi strumieniami odpowiadają znane ci już polecenie

Std::cin – strumień wejścia – wprowadzamy coś do programu

Std::cout – wyprowadzamy informacje poza program na ekran monitora.

W pewnym momencie jednak przychodzi potrzeba zapisania danych na naszym dysku. Z pomocą przychodzi biblioteka *fstream* (file stream) która operuje na strumieniach z/do pliku.

Najpierw należy utworzyć strumień za pomocą:

std::fstream plik;

Następnie otwieramy plik za pomocą metody *open()*

plik.open(„plik.txt”, tryb otwarcia pliku)

Tryb otwarcia pliku będzie zależał od tego co chcemy z tym plikiem zrobić. Jeżeli chcemy odczytać dane, to po przecinku użyjemy *ios::in*

Jeżeli chcemy nadpisać zawartość pliku, używamy *ios::out*

Jeżeli chcemy tylko coś do pliku dopisać, bez traceniu zawartości, używamy *ios::app*.

Na ten moment program może odczytywać dane z pliku

```

(Globalny zasięg)
main()
#include <iostream>
#include <string>
#include <fstream>

int main()
{
    std::fstream plik; //tworzymy strumien
    plik.open("plik.txt", std::ios::in); //otwieramy strumien
    std::string z_pliku; //tworzymy zmienna ktora przechowa nam pobrane dane

    getline(plik, z_pliku); //wydobywamy linijke tekstu z pliku do zmiennej
    std::cout << z_pliku; //wypisywanie danych
}

```

Na jednym strumieniu możemy jednak używać kilku trybów, po prostu należy je wypisać oddzielając je znakiem |.

Teraz nasz program może i odczytywać, i zapisywać dane do pliku

```

(Globalny zasięg)
main()
int main()
{
    std::fstream plik; //tworzymy strumien
    plik.open("plik.txt", std::ios::in | std::ios::out); //otwieramy strumien
    std::string zdo_pliku; //tworzymy zmienna ktora przechowa nam pobrane dane

    getline(plik, zdo_pliku); //wydobywamy linijke tekstu z pliku do zmiennej
    std::cout << zdo_pliku; //wypisywanie danych
    std::cout << " chcesz cos dopisac? t/n" << std::endl;
    char znak;
    std::cin >> znak;
    if (znak == 'n') { plik.close(); } //zamkniecie pliku gdy wybierzemy n
    else
    {
        std::cout << "wprowadz co chcesz dopisac (bez spacji)" << std::endl;
        std::cin >> zdo_pliku;
        std::cout << zdo_pliku;
        plik.clear(); //czyszczenie strumienia, aby mozna bylo zapisac
        plik << zdo_pliku;
        plik.close();
    }
}

```

Zauważ jednak, że podczas gdy 1 strumień dzieli dwa tryby, za każdym razem przy zmianie trybu należy wyczyścić strumień. Wyjściem z tego problemu może być utworzenie dwóch strumieni – jeden na zapis, drugi na odczyt. Należy pamiętać jednak, że gdy zależy nam na wydajności, większa ilość strumieni wymaga więcej zasobów.

Dział 11 – klasy w c++

Klasy w c++ można łatwo określić jako nasze własne biblioteki, w których możemy tworzyć nasze własne obiekty o określonych parametrach i metody, które nimi w jakiś sposób zarządzają.

Do utworzenia klasy potrzebujemy (poza głównym plikiem .cpp naszego programu) plik .cpp i .h TAK SAMO NAZWANE. Plik .h to plik nagłówkowy – w nim umieszczamy deklaracje obiektów i metod z naszej klasy. Innymi słowy informujemy program co ma w sobie nasza klasa. W pliku .cpp jest cała reszta, czyli działanie poszczególnych metod, konstruktorów i destruktorów.

Konstruktor – metoda która tworzy obiekt w naszej klasie. Jest wywoływana za każdym razem podczas tworzenia obiektu

Destruktor – metoda która uruchamia się za każdym razem gdy obiekt naszej klasy kończy swój żywot. Bardzo często ta metoda pozostaje pusta.

Stwórzmy teraz klasę obiektów „uczeń”. Założmy że będzie ona przechowywać informacje o:

- imieniu i nazwisku
- ocenie z polskiego i matematyki
- ocenie z zachowania

W pliku .h deklarujemy konstruktor, destruktor oraz potrzebne zmienne które będą przechowywać informacje

```
#pragma once
#include <iostream>

class uczen
{
public: //okreslenie dostepnosci do metod/obiektow
    uczen(); //deklaracja konstruktora
    ~uczen(); //deklaracja destruktora UWAGA TYLDA
    int ocena_MAT, ocena_PL; //deklaracja potrzebnych zmiennych
    std::string imie, nazwisko, zachowanie;
};
```

Teraz plik .cpp. W nim zajmiemy się konstruktorem i destrukorem. Zaprogramujemy konstruktor tak, aby opbieral komenda *std::cin* od uzytkownika informacje o uczniu.

```
#include "uczen.h"
#include <iostream> //potrzebne do std::cin

uczen::uczen()
{
    std::cout << "wprowadz po kolei dane: imie, nazwisko, ocena z polskiego, ocena z matematyki, zachowanie" << std::endl;
    std::cin >> imie; //uczytkownik wpraowdza infoamcje o uczniu do naszego obiektu
    std::cin >> nazwisko;
    std::cin >> ocena_PL;
    std::cin >> ocena_MAT;
    std::cin >> zachowanie;
}

uczen::~uczen()
{
    //zostawiamy puste
}
```

Destruktor zostawiamy pusty, bo nie chcemy aby cos się działo w momencie usuwania naszego obiektu.

Teraz wracamy do naszego głównego pliku. Trzeba zadeklarować naszą klasę za pomocą *#include „uczen.h”* UWAGA CUDZYSŁÓW

Teraz zostało tylko dopisać polecenie które utworzy nasz obiekt.

```
#include <iostream>
#include "uczen.h"

int main()
{
    uczen nowy1; //BEZ NAWIASOW UWAGA
}
```

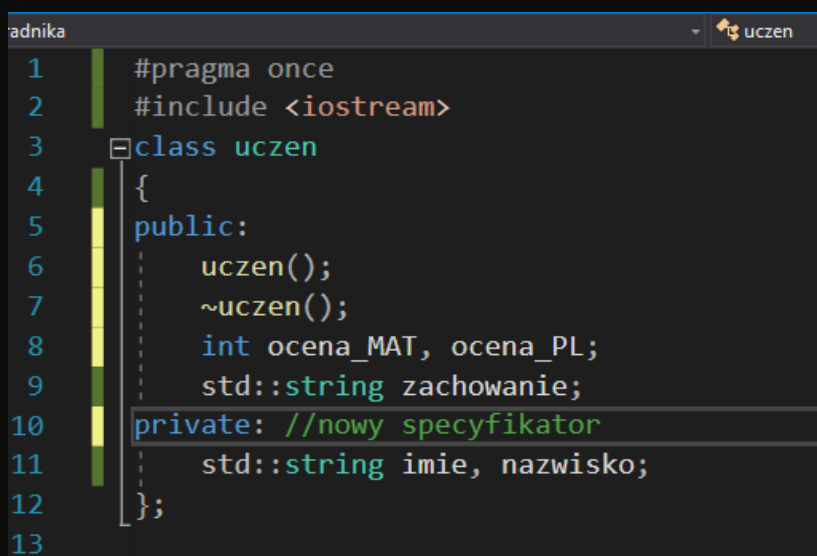
Nie omówiliśmy jeszcze jednej rzeczy. W pliku .h przed wypisaniem deklaracji napisaliśmy „*public:*”. Jest to tzw. *Specyfikator dostępu*. Mamy 3 takie specyfikatory:

Public – dane, obiekty i zmienne z klasy są dostępne wszędzie, nawet poza klasą

Protected – dane, obiekty i zmienne z klasy są dostępne tylko we wnętrzu tej klasy, i klas pochodnych, które coś dziedziczą.

Private – dane, obiekty i zmienne są dostępne tylko i wyłącznie we wnętrzu tej klasy.

Ze względu na RODO i ochronę danych osobowych zmienne o imieniu i nazwisku przenieśmy do nowo utworzonego specyfikatora *private*



```
adnika uczen
1  #pragma once
2  #include <iostream>
3  class uczen
4  {
5  public:
6      uczen();
7      ~uczen();
8      int ocena_MAT, ocena_PL;
9      std::string zachowanie;
10 private: //nowy specyfikator
11     std::string imie, nazwisko;
12 };
13
```

Dziedziczenie klas

Jak już wspomniałem, jedna klasa może coś dziedziczyć od innej. Przykładowo możemy utworzyć klasę samochód, która będzie przechowywała takie informacje o obiekcie:

- marka
- model
- prędkość maksymalna
- max ilość osób
- rok produkcji

Od tego możemy utworzyć klasę samochód sportowy, która będzie dziedziczyć od klasy pojazd tamte zmienne, i doda kilka nowych, kierowanych pod pojazdy sportowe

- pojemność silnika
- ilość koni mechanicznych
- czas przyspieszenia 0-100km/h

Od Klasy samochód możemy też utworzyć inną klasę pochodną – samochód transportowy, i nadać tej klasie takie dodatkowe zmienne

- pojemność bagażnika
- długość
- wysokość

Spróbujmy teraz wykonać takie klasy...

W tym wypadku informacje będziemy pobierać z pliku głównego, a w obiekcie będziemy umieszczać je za pomocą argumentów.

Najpierw plik.h i plik .cpp klasy bazowej:

```
#pragma once
#include <iostream>

class samochod
{
public:
    samochod();
    samochod(std::string marka, std::string model, int rocznik, int max_osob, int max_predkosc);
    ~samochod();
    std::string marka, model;
    int rocznik, max_osob, max_predkosc;
};
```

```
#include "samochod.h"

samochod::samochod()
{
    marka = "";
    model = "";
    rocznik = 0;
    max_osob = 0;
    max_predkosc = 0;
}

samochod::samochod(std::string marka, std::string model, int rocznik, int max_osob, int max_predkosc)
{
    this->marka = marka;
    this->model = model;
    this->rocznik = rocznik;
    this->max_osob = max_osob;
    this->max_predkosc = max_predkosc;
}

samochod::~samochod()
{
}
```

Mamy dwa konstruktory, jeden bez argumentów, drugi z nimi.

Teraz pora na plik .h i .cpp klasy pochodnej – samochód sportowy

```
#pragma once
#include "samochod.h"
#include <iostream>
class sportowy :
    public samochod
{
public:
    sportowy();
    sportowy(std::string marka, std::string model, int rocznik, int max_osob, int max_predkosc, int pojemnosc_silnika, int czas, int ilosc_koni);
    ~sportowy();
    double pojemnosc_silnika, czas;
    int ilosc_koni;
};
```

```
#include "sportowy.h"
#include <iostream>

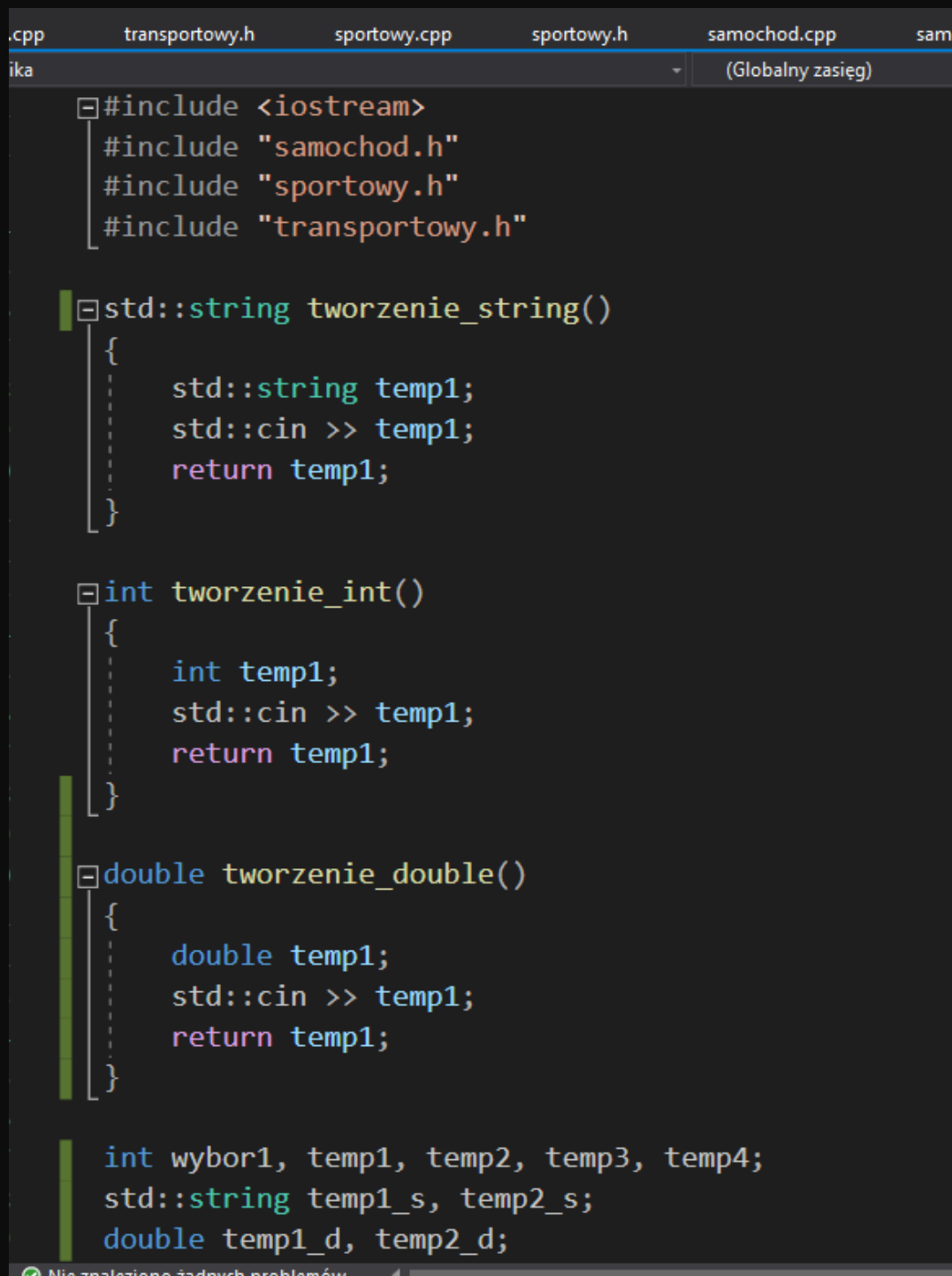
sportowy::sportowy()
{
    marka = "";
    model = "";
    rocznik = 0;
    max_osob = 0;
    max_predkosc = 0;
    czas = 0;
    pojemnosc_silnika = 0;
    ilosc_koni = 0;
}

sportowy::sportowy(std::string marka, std::string model, int rocznik, int max_osob, int max_predkosc, int pojemnosc_silnika, int czas, int ilosc_koni)
{
    this->marka = marka;
    this->model = model;
    this->rocznik = rocznik;
    this->max_osob = max_osob;
    this->max_predkosc = max_predkosc;
    this->ilosc_koni = ilosc_koni;
    this->pojemnosc_silnika = pojemnosc_silnika;
    this->czas = czas;
}

sportowy::~sportowy()
{
}
```

Zauważ że w deklaracji nie powtarzamy zmiennych z klasy pojazd, a możemy je wykorzystać w klasie pochodnej

Następnie, w identyczny sposób, klasę samochód transportowy. Wygląda ona bardzo podobnie, więc nie wrzucam jej screenów, poradzisz sobie. W głównym pliku załączamy wszystkie klasy, i tworzymy zmienne i funkcje które pozwolą na tworzenie obiektów w danej klasie. Tak wyglądają zmienne i funkcje dodatkowe.



```
.cpp      transportowy.h      sportowy.cpp      sportowy.h      samochod.cpp      sam
ika      (Globalny zasięg)

#include <iostream>
#include "samochod.h"
#include "sportowy.h"
#include "transportowy.h"

std::string tworzenie_string()
{
    std::string temp1;
    std::cin >> temp1;
    return temp1;
}

int tworzenie_int()
{
    int temp1;
    std::cin >> temp1;
    return temp1;
}

double tworzenie_double()
{
    double temp1;
    std::cin >> temp1;
    return temp1;
}

int wybor1, temp1, temp2, temp3, temp4;
std::string temp1_s, temp2_s;
double temp1_d, temp2_d;
```

A tak wygląda funkcja *main()*

```
transportowy.h  sportowy.cpp  sportowy.h  samochod.cpp  samochod.h  do_poradnika.cpp  X
(Globalny zasięg)  main()

int main()
{
    std::cout << "jaki pojazd chcesz stworzyc? 1 - zwykly 2 - sportowy 3 - transportowy" << std::endl;
    std::cin >> wybor1;
    if (wybor1 == 1)
    {
        std::cout << "podaj po kolei dane: marka, model, rocznik, max ilosc osob, max predkosc" << std::endl;
        temp1_s = tworzenie_string();
        temp2_s = tworzenie_string();
        temp1 = tworzenie_int();
        temp2 = tworzenie_int();
        temp3 = tworzenie_int();
        samochod nowy(temp1_s, temp2_s, temp1, temp2, temp3);
    }
    else if (wybor1 == 2)
    {
        std::cout << "podaj po kolei dane: marka, model, rocznik, max ilosc osob, max predkosc" << std::endl;
        std::cout << "ilosc koni, pojemnosc silnika, czas 0-100kmph" << std::endl;
        temp1_s = tworzenie_string();
        temp2_s = tworzenie_string();
        temp1 = tworzenie_int();
        temp2 = tworzenie_int();
        temp3 = tworzenie_int();
        temp4 = tworzenie_int();
        temp1_d = tworzenie_double();
        temp2_d = tworzenie_double();
        sportowy nowy(temp1_s, temp2_s, temp1, temp2, temp3, temp4, temp1_d, temp2_d);
    }
    else
    {
        std::cout << "podaj po kolei dane: marka, model, rocznik, max ilosc osob, max predkosc" << std::endl;
        std::cout << "pojemnosc bagaznika, wysokosc, dlugosc" << std::endl;
        temp1_s = tworzenie_string();
        temp2_s = tworzenie_string();
        temp1 = tworzenie_int();
        temp2 = tworzenie_int();
        temp3 = tworzenie_int();
        temp4 = tworzenie_int();
        temp1_d = tworzenie_double();
        temp2_d = tworzenie_double();
        transportowy nowy(temp1_s, temp2_s, temp1, temp2, temp3, temp4, temp1_d, temp2_d);
    }
}
```

Aby nie musieć wielokrotnie powtarzać `std::cin` wykorzystaliśmy funkcję która trochę zmniejszyła nasz kod

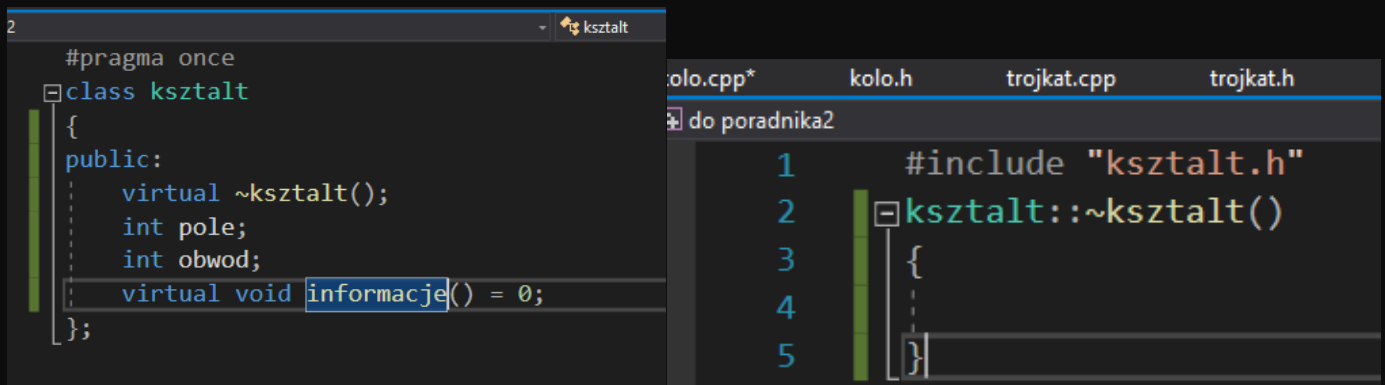
Dział 12 – klasy wirtualne

Klasy wirtualne to szczególny przypadek dziedziczenia klas, w którym przyjmujemy, że powstawać będą tylko obiekty z klasy pochodnej.

Utwórzmy klasę kształt – będzie to klasa wirtualna ze zmiennymi obwód i pole

Klasy pochodne które utworzymy to trójkąt, kwadrat i koło. Trójkąt będzie wyróżniała wysokość, kwadrat przekątna, a koło będzie posiadało promień i średnicę. Utworzymy też metodę która będzie wypisywała informacje o danej figurze.

Najpierw zajmijmy się klasą wirtualną:



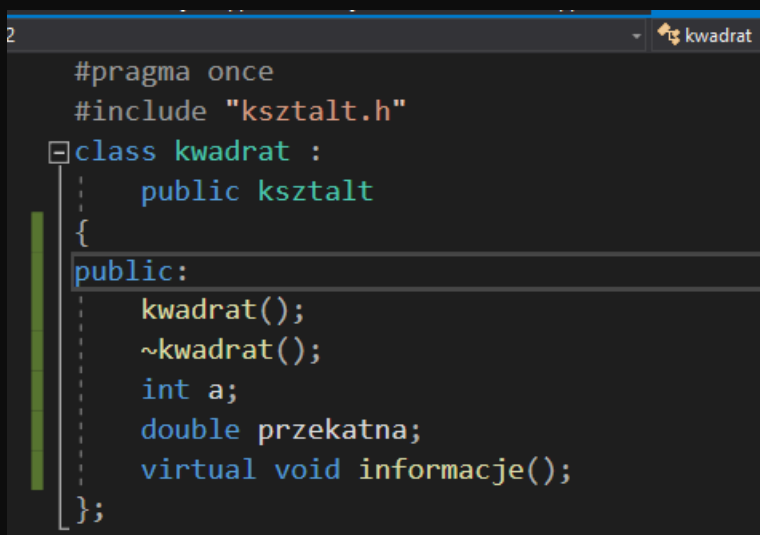
The image shows two code editors. The left editor, titled 'ksztalt', contains the following code:

```
#pragma once
class ksztalt
{
public:
    virtual ~ksztalt();
    int pole;
    int obwod;
    virtual void informacje() = 0;
};
```

The right editor shows a project view with files: 'kolo.cpp*', 'kolo.h', 'trojkat.cpp', and 'trojkat.h'. Below it, a file named 'do poradnika2' is open, showing the implementation of the destructor for the 'ksztalt' class:

```
1 #include "ksztalt.h"
2 ksztalt::~ksztalt()
3 {
4 }
5 }
```

Jak możesz zauważyć – jest ona praktycznie pusta, i nie zawiera konstruktora – nie będzie potrzebny. Zawiera tylko deklaracje zmiennych i metod które będą potrzebne do zmiennych pochodnych. Teraz pora na klasy pochodne, na początek kwadrat.



The image shows a code editor titled 'kwadrat' with the following code:

```
#pragma once
#include "ksztalt.h"
class kwadrat :
    public ksztalt
{
public:
    kwadrat();
    ~kwadrat();
    int a;
    double przekatna;
    virtual void informacje();
};
```

```

ka2      -> kwadrat      ~kwadrat()
#include "kwadrat.h"
#include <iostream>
#include <math.h>

kwadrat::kwadrat()
{
    std::cout << "wprowadz dane: bok a" << std::endl;
    std::cin >> a;
    przekatna = a * sqrt(2);
    pole = pow(a, 2);
    obwod = 4 * a;
}

kwadrat::~kwadrat()
{
}

void kwadrat::informacje()
{
    std::cout << "to kwadrat o boku a = " << a << " i polu powierzchni rownym a^2 = " << pole << std::endl;
    std::cout << "jego przekatna wynosi " << przekatna << " a obwod O = " << obwod << std::endl;
}

```

Zauważ, że nawet w klasie pochodnej dopisujemy przy deklaracji metody dopisek „*virtual*”. W pliku .cpp jednak nie jest on potrzebny, potrzebne za to jest dopisanie `kwadrat::`:

Pozostałe klasy pochodne mają tworzymy w ten sam sposób. Teraz główny plik:

```

kolo.h    trojkat.cpp    trojkat.h    kwadrat.cpp    kwadrat.h    ksztalt.cpp    ksztalt.h
(Globalny zasięg)

#include <iostream>
#include "ksztalt.h"
#include "kolo.h"
#include "kwadrat.h"
#include "trojkat.h"

int main()
{
    std::cout << "najpierw stworzmy kwadrat" << std::endl;
    kwadrat nowy1;
    nowy1.informacje();
    std::cout << "teraz trojkat" << std::endl;
    trojkat nowy2;
    nowy2.informacje();
    std::cout << "a na koniec kolo" << std::endl;
    kolo nowy3;
    nowy3.informacje();
}

```

Dział 13 – Przydatne polecenia i metody bibliotek.

I – przestrzeń nazw std.

Jak możesz zauważyć – w naszych kodach bardzo często powtarza się dopisek std:::

Otóż nie trzeba go pisać! Wystarczy że na początku kodu, pod załącznikami bibliotek `#include` napiszesz *using namespace std;*

Pozwoli to nam trochę odchudzić nasz kod:

```
(Globalny zasięg)

#include <iostream>

using namespace std;

int main()
{
    cout << "to jest komenda cout bez wypisywania std na początku" << endl;
    string tekst1;
    cin >> tekst1;
}
```

II – flaga \n na końcu tekstu.

Działa ona dokładnie tak samo jak `<< std::endl;` ale pisze się ją szybciej, i umieszcza się ją jeszcze w cudzysłowie tekstu.

```
(Globalny zasięg)

#include <iostream>

using namespace std;

int main()
{
    string tekst1;
    cin >> tekst1;
    cout << "pod spodem bedzie wypisana zmienna\n" << tekst1;
}
```

Poniżej wrzucam tabelki z metodami do danych bibliotek

Tymczasowo są to linki, kiedyś zrobię te tabelki :DD

Biblioteka vector:

<https://cpp0x.pl/dokumentacja/standard-C++/vector/819>

Biblioteka math.h:

<https://cpp0x.pl/kursy/Kurs-C++/Dodatkowe-materialy/Biblioteka-math-h/322>

Biblioteka string:

<http://www.algorytm.edu.pl/biblioteki/string.html>