Początki z C++

1.1 Witaj Świecie!

```
#include <iostream>

using namespace std;

int main()
{
	cout << "Hello World!" << endl;

	return 0;
}
```

Analiza:

- `#include <iostream>` jest dyrektywą preprocesora, która dołącza **plik nagłówkowy** `iostream`.

  `iostream` to **plik nagłówkowy** standardowej biblioteki która zawiera definicje standardowego wejścia (input) i wyjścia (output).

- `using namespace std;` jest to użycie przestrzeni nazw std, co oznacza, że nie musimy poprzedzać instrukcji wykorzystywanych odpowiednim przedrostkiem. Np. `std` dla `iostream`.

- `int main() { ... }` to definicja nowej funkcji nazwanej `main`. W programie C++ zawarta może być tylko jedna funkcja `main`, która musi zwracać wartość `int`.

  Ogólnie przyjęło się, że kodem wyjściowym, zwracającym sukces jest `0`. Każda inna cyfra oznacza błąd. Jeśli funkcja nie zawiera instrukcji `return 0;`, `main` zwraca domyślnie `0`.

  Wszystkie inne funkcje (z wyjątkiem funkcji typu `void`) muszą zwracać jakąś wartość.

- `cout << "Hello World!" << endl;` wypisuje “Hello World!” na standardowym strumieniu wyjścia (konsola):
    - `cout` jest obiektem znajdującym się w `iostream` reprezentujący standardowe wyjście programu.
    - `<< endl;` odpowiada za zakończenie linii i opróżnienie bufora strumienia.

      Standardowa biblioteka definiuje operator `<<` jako sposób konkatenacji strumieni tekstowych, zmiennych itp. i wyświetla to w strumieniu wyjścia.

- Średnik (`;`) informuje kompilator o zakończeniu instrukcji. Wszystkie instrukcji i definicje klas wymagają zakończenia linii średnikiem.

1.2 Komentarze

Komentarze to sposób opisywania kodu. Kompilator nie będzie brał pod uwagę tego kodu podczas kompilacji. Komentarze często dają wgląd w metodykę programu.
Są dwa rodzaje komentarzy:

```
int main()
{
// To jest komentarz mieszczący się w jednej linii
int a; // To też jest komentarz w jednej linii
int i; // I to.
}
```

Ciąg znaków `/*` jest deklaracją początku bloku komentarza, a `*/` jego końcem. Wszystko pomiędzy interpretowane jest jako komentarz. Nawet jeśli napiszesz instrukcję C++ nie zostanie ona wykonana.

```
int main()
{
   /*
    To jest blok komentarza
    */
    int a;
}
```

Blok komentarzy może zaczynać się i kończyć w jednej linii:

```
void Funkcja(/* argument 1 */ int a, /*argument 2 */ int b);
```

Jak w każdym języku programowania, komentarze zapewniają wiele korzyści:

- Kod staje się łatwiejszy w odczycie i zarządzaniu
- Wyjaśniają znaczenie i funkcjonalności kodu
- Często w komentarzach umieszcza się informacje o: licencjach, notatkach projektowych, specjalnych podziękowaniach itd.

Komentarze mają również swoje wady:

- Z każdą zmianą kodu, komentarz również musi zostać zmieniony
- Zbyt duża ilość komentarzy sprawia, że kod staje się mniej czytelny

Najczęstszym sposobem używania komentarzy jest dezaktywowanie (wyłączanie) poszczególnych funkcji kodu podczas testów i procesów debuggowania kodu.

1.3 Pętle

Pętle wywołują grupy procedur, aż nie spełnią warunku końca. Wyróżniamy 3 podstawowe typy pętli w C++: `for`, `while`, `do...while`.

1.3.1 Pętla for

Pętla `for` wykonuje polecenia w niej zawarte, dopóki warunek pętli jest spełniony. Przed pierwszym obiegiem pętli wyrażenie inicjalizujące wykonywane jest tylko raz. Po każdym obiegu wykonywana jest iteracja.

```
for(/*wyrażenie inicjalizujące*/;/*warunek*/;/*wykonanie inkrementacji*/)
{
      //polecenia w pętli
}
```

Wyjaśnienia:

- *wyrażenie inicjalizujące*: to polecenie wykonywane jest tylko raz, na początku pętli `for`. Możesz tutaj wpisać deklarację kilku zmiennych jednego typu, np. `int i = 0, a = 2, b = 3`. Te zmienne są widoczne jedynie w zakresie pętli. Zmienne zdefiniowane przed pętlą o tej samej nazwie są ukryte podczas wykonywania pętli.
- *warunek*: to polecenie jest sprawdzane przed każdym następnym obiegiem pętli, zwraca `true` lub `false`, przerywa pętlę jeśli zwróci wynik `false`
- *wykonanie iteracji*: to polecenie jest wykonywane po wszystkich poleceniach z pętli, przed sprawdzeniem *warunku,* chyba że pętla została wcześniej przerwana jednym z poleceń (`break`, `return` albo zwróci wyjątek)

Przybliżony odpowiednik dla pętli `for` napisany jako pętla `while`:

```
/*inicjalizacja*/
while (/*condition*/)
{
	//polecenia w pętli; użycie 'continue' pominie część inkrementacji poniżej
	/*wykonanie iteracji*/
}
```

Najpopularniejszym przypadkiem używania pętli `for` jest wykonywanie danego polecenia/danych poleceń konkretną ilość razy.
Na przykład:

```
for(int i = 0; i < 10; i++) {
	std::cout << i << std::endl;
}
```

Działającą pętlą jest również ten przykład:

```
for(int a = 0, b = 10, c = 20; a+b+c < 100; c--, b++, a+=c) {
	std:cout << a << " " << b << " " << c << std::endl;
}
```

Przykład ukrycia zadeklarowanych zmiennych przed pętlą:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //deklarujemy nową zmienną i
	//kilka poleceń, wartość i sięga od 0 do 9 podczas wykonywania pętli
}
//po wykonaniu pętli, mamy dostęp do zmiennej i z wartością 99
```

Ale jeśli chcesz użyć wcześniej zadeklarowanej zmiennej i jej nie ukrywać, pomiń część deklaracji:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //używamy zadeklarowanej wcześniej zmiennej i
	//kilka poleceń, wartość i sięga od 0 do 9 podczas wykonywania pętli
}
//po wykonaniu pętli, mamy dostęp do zmiennej i z wartością 10
```

Adnotacje:

- Polecenia inicjalizacji i inkrementacji mogą wykonywać operacje niezwiązane z warunkiem zakończenia pętli. Ale dla czytelności kodu, najlepszą praktyką jest tworzyć operacje bezpośrednio powiązane z pętlą.
- Zmienna zadeklarowana w *wyrażeniu inicjalizującym* jest widoczna tylko w zakresie pętli for i po pętli jest usuwana z pamięci komputera
- Nie zapomnij, że zmienna, która została zadeklarowana w *wyrażeniu inicjalizującym* może być modyfikowana podczas działania pętli, tak samo jak w trakcie sprawdzania *warunku*.

Przykład pętli która liczy od 0 do 10:

```
for(int licznik = 0; licznik <= 10; licznik++){
      std::cout << licznik << '\n';
}
//zmienna licznik nie jest tu dostępna
```

Wyjaśnienie kodu:

- `int licznik = 0` jest to inicjalizacja zmiennej `licznik` wartością 0. (Ta zmienna może być używana **tylko** wewnątrz pętli).
- `licznik <= 0` jest to **warunek**, który sprawdza czy `licznik` jest mniejszy lub równy od 10. Jeśli jest pętla się wykonuje, jeśli nie - następuje koniec.
- licznik++ jest to operacja inkrementacji, która po każdym obiegu pętli zwiększa wartość licznika o 1.

Jeśli zostawisz nawias pust (tylko z średnikami) stworzysz nieskończoną pętle.

```
for(;;)
   std::cout << "Bez końca! \n";
```

Jednak z takiej pętli można wyjść. Wystarczy użyć instrukcji `break`, `return` albo zwrócić wyjątek.

1.3.2 Pętla while

Pętla while wykonuje polecenia tak długo, dopóki *warunek* nie zwróci `false`.
Poniższy przykład wypisze wszystkie liczby od 0 do 9:

```
int i = 0;
while (i < 10)
{
      std::cout << i << " ";
      ++i; //licznik inkrementacji
}
std:: cout << std::endl;
// zakończenie linii; Zostanie wypisane "0 1 2 3 4 5 6 7 8 9".
```

Od wersji C++17, dwa pierwsze polecenia mogą być złączone:

```
while(int i = 0; i < 10)
//... reszta jest taka sama
```

Przykład tworzący nieskończoną nieskończoną pętlę:

```
while (true)
{
    /* Rób coś w nieskończoność (wciąż możesz przerwać pętlę za pomocą
komendy break;) */
}
```

1.3.3 Pętla Do-while

Pętla do-while jest bardzo podobna do pętli while, tylko że warunek jest sprawdzany pod koniec każdego obiegu, a nie na początku. Pętla do-while zawsze wykonuje się przynajmniej raz.

Poniższy przykład wypisze 0, po czym warunek zwróci `false` na zakończeniu pierwszego obiegu.

```
int i = 0;
do
{
    std::cout << i;
    ++i; //licznik inkrementacji
}
while (i < 0);
std::cout << std::endl;
//W konsoli zostanie wypisane 0
```

Ważne: Nie zapomnij o średniku na końcu linii while(warunek);, który jest wymagany w konstrukcji pętli do-while.

By porównać obie pętle, poniższy przykład nie wypisze nic, ponieważ warunek zwróci `false` na początku pierwszej iteracji:

```
int i = 0;
while (i < 0)
{
    std::cout << i;
    ++i; //licznik inkrementacji
}
std::cout << std::endl; /* zakończenie linii; W konsoli nic nie zostanie
wypisane */
```

Ważne: Pętla while może zostać przerwana nie tylko w momencie, gdy warunek zwróci `false`, ale też za pomocą komend break lub return.

```
int i = 0;
do
{
    std::cout << i;
    ++i; //licznik inkrementacji
    if (i > 5) break;
}
while (true);
std::cout << std::endl; /* zakończenie linii; W konsoli zostanie wypisane
"0 1 2 3 4 5" */
```

1.3.4 Wyrażenia kontrolne: ***Break*** i ***Continue***

Wyrażenia kontrole w pętlach używane są do kontroli ich przebiegu.

Instrukcja `break` przerywa pętle natychmiast. (Obieg nie zostanie dokończony)

```
for (int i = 0; i < 10; i++)
{
    if (i == 4) break; // natychmiast wyjdziemy z pętli.
    std::cout << i << '\n';
}
```

Wyjście powyższego kodu:

```
1
2
3
```

Polecenie `continue` zamiast opuszczać pętla pomija obecny obieg.

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // zwraca true gdy i jest podzielne przez 2
        continue; // natychmiast przechodzi do kolejnego obiegu
/* Ta linia zostanie wykonana tylko gdy nie wykona się instrukcja
continue */
    std::cout << i << " jest nieparzyste\n";
}
```

Wyjście:

```
1 jest nieparzyste
3 jest nieparzyste
5 jest nieparzyste
```

Ponieważ wyrażenie kontrolne często są trudne do zrozumienia dla Ludzi instrukcję `break` i `continue` powinny być używane oszczędnie. Pierwsza pętla `for` mogła zostać zapisana w taki sposób:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << std::endl;
}
```

Drugą pętle mogliśmy zapisać tak:

```
for (int i = 0; i < 6; i++)
{
    if(i % 2 != 0){
        std::cout << i << " jest nieparzyste\n";
    }
}
```

1.4 Operacje warunkowe

1.4.1 **Switch**

Instrukcja `switch` powoduje przejście kodu do jednej z wielu instrukcji w zależności od wartości warunku.

Po słowie `switch` występuje blok kodu (klamry { }), który zawierą etykiety `case` i opcjonalnie może zawierać etykietę `default`. Kiedy `switch` jest wykonywany kod "przeskakuje" do etykiety (`case`) pasującej do warunku instrukcji. Jeśli nie pasuje do żadnego, przejdzie do etykiety domyślnej (`default`).

```
char zmienna = 'k';
bool potwierdzenie;
switch (zmienna) {
    case 'y':
        potwierdzenie= true; //Nie wykona się
    break;
    case 'n':
        potwierdzenie = false; //Nie wykona się
    break;
    default:
        std::cout << "Zła litera!\n";
}
```

1.4.2 **If**

Instrukcja `if` sprawdza czy warunek jest prawdziwy i jeśli jest wykonuje instrukcję/blok instrukcji.

```
int x;
std::cout << "Proszę podać liczbę dodatnią." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "Podałeś liczbę mniejszą od zera!" << std::endl;
}
```

1.4.3 **else**

Po instrukcji `if` może występować słowo `else`. Oznacza to, że gdy instrukcja `if` **nie** zostanie spełniona wykona się instrukcja/blok kodu znajdujący się po tym słowie.

```
int x;
std::cin >> x; //3
if (x%2 == 0) { // 3%2 zwróci nam 1
    std::cout << "Liczba jest parzysta\n"; // nie wykona się
} else {
    std::cout << "Liczba nie jest parzysta\n"; // wykona się
}
```

po instrukcji `else` możemy również dać `if`.

```
if (zmienna=='a') {
    // kiedy zmienna będzie równa a
} else if (zmienna=='b') {
    // kiedy zmienna będzie równa b
} else if (zmienna=='c') {
    // kiedy zmienna będzie równa c
} else {
    //kiedy zmienna będzie inna od powyższych
}
```

1.5 Tablice

Tablice są zbiorami elementów tego samego typu danych,które są umieszczone obok siebie w pamięci komputera. Pozwalają one na zadeklarowanie wielu wartości tego samego typu i dostęp do każdej z nich osobna, pozwalają uniknąć definiowania wielu zmiennych.

1.5.1 Inicjalizacja tablicy

Tablice deklarujemy w taki sam sposób jak zmienną tylko dodajemy kwadratowy nawias [ ] po jej nazwie. W [ ] przeważnie podajemy ilość elementów tablicy.

Przykład zadeklarowania tablicy typu całkowitego o nazwie “LearnIT”, która zawiera 14 elementów:

```
int LearnIT[14];
```

Tablice można deklarować i inicjalizować jednocześnie w taki sposób:

```
int LearnIT[5] = { 1, 2, 3, 4, 5};
```

Kiedy podajemy blok inicjalizacyjny nie musimy podawać ilości elementów wewnątrz [ ]. Kompilator automatycznie to policzy. Przykład dla 3 elementów:

```
int LearnIT[] = { 1, 2, 3};
```

Istnieje także możliwość inicjalizacji tylko kilku początkowych elementów, jednocześnie rezerwując więcej pamięci. Poniżej przykład dla 5 elementowej tablicy, kompilator automatycznie zadeklaruje pozostałe wartości z podstawową wartością danego typu.

```
int LearnIT[5] = { 10, 20 }; // Tablica wygląda tak: 10, 20, 0, 0, 0
```

Warto również wspomnieć, że index(pozycja) pierwszego elementu tablicy liczona jest od 0.

```
int tablica[5] = { 10 /* Element 0 */, 20, 30, 40, 50 /* Element 4 */};
std::cout << array[4]; // wyjście 50
std::cout << array[0] // wyjście 10
```