



# KURS JĘZYKA C++

I. ŁAGODNE WPROWADZENIE DO JĘZYKA C++



# SPIS TREŚCI

- Pierwsze programy w C++
- Struktura programu w C++
- Zmienne ustalone `const` i `constexpr`
- Jednolita inicjalizacja
- Ograniczanie zasięgu zmiennej
- Referencje
- Napisy typu `string`
- Tablice typu `vector`
- Pętla `for` dla przeglądania tablic
- Wskaźnik pusty `nullptr`
- Standardowe wejście i wyjście

# PIERWSZE PROGRAMY

## ■ Najprostszy program w języku C++:

```
main() { }
```

## ■ Program powitalny w języku C++:

```
#include <iostream>
```

```
using namespace std;
```

```
int main (int argc, char *argv[]) {  
    cout << "witaj na kursie C++" << endl;  
    return 0;  
}
```

# PROGRAM, KTÓRY COŚ OBLICZA

■ Oto program, który zamieni milimetry na cale:

```
#include <iostream>

using namespace std;

int main () {
    cerr << "[mm]: ";
    double mm;
    cin >> mm;
    double inch = mm/25.3995;
    cout << inch << endl;
    cerr << mm << "[mm] = " << inch << "[in]" << endl;
    return 0;
}
```

# STRUKTURA PROGRAMU W C++

## ■ Podział na pliki:

- nagłówkowe (rozszerzenie .hpp) z deklaracjami,
- źródłowe (rozszerzenie .cpp) z definicjami.

## ■ W plikach nagłówkowych stosujemy włączanie warunkowe:

```
#ifndef moje_hpp  
#define moje_hpp  
/* właściwa zawartość pliku moje_hpp */  
#endif
```

## ■ Aby otrzymać uruchamialny plik wynikowy w jednym z plików źródłowych musi się znaleźć definicja funkcji `main()`.

# STANDARDOWE PLIKI NAGŁÓWKOWE

- Pliki nagłówkowe odnoszące się do biblioteki standardowej nie mają żadnego rozszerzenia, na przykład:

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

- Nazwy odnoszące się do starych plików nagłówkowych z języka C są poprzedzone literą "c", na przykład:

```
#include <cmath>
#include <cstdlib>
```

- Wszystkie definicje z biblioteki standardowej są umieszczone w przestrzeni nazw `std`, dlatego wygodnie jest na początku (małego) programu włączyć tę przestrzeń poleceniem:

```
using namespace std;
```

# DEDUKCJA TYPÓW DANYCH

- W definicji zmiennej z jawnym inicjowaniem można użyć słowa kluczowego `auto` zamiast typu – można w ten sposób utworzyć zmienną o typie takim, jak typ inicjującej wartości:

```
auto zmienna = wyrażenie;
```

- Przykład:

```
auto x = a * 2 - 1e-6;
```

- Słowo kluczowe `decltype` może być zastosowane w celu określenia typu w czasie kompilacji na podstawie typu wyrażenia:

```
decltype(wyrażenie) zmienna;
```

- Przykład:

```
decltype(b / 2 + 1e-6) y = 5;
```

# STAŁE, CZYLI ZMIENNE USTALONE

- Stałe są oznaczone deklaratorem **const** w deklaracji:  
`const TYP stała = wyrażenie;`
- Stałą należy zainicjalizować podczas deklaracji.
- Inicjalizacja stałego argumentu w funkcji następuje podczas wywołania funkcji.
- Do stałej nie wolno w programie nic przypisać – jej wartość określamy tylko podczas inicjalizacji.
- Przykład:  
`const double phi = 1.618'033'989;`



# STAŁE W PORÓWNANIU Z MAKRODEFINICJAMI

- Dlaczego stałe są bezpieczniejsze od makrodefinicji?
  - znany jest typ stałej
  - można określić zasięg nazwy stałej
  - nazwa stałej jest znana kompilatorowi
  - stała to komórka pamięci posiadająca swój adres
  - łatwiejsza praca z debugerem
- Używajmy stałych zamiast makrodefinicji !

# WYRAŻENIA STAŁE

- Stałe wyrażenia są oznaczone deklaratorem **constexpr** w deklaracji:

**constexpr** *TYP stała = wyrażenie;*

- Stałe wyrażenia **constexpr** są obliczane przez kompilator na etapie kompilacji a nie wykonania programu.
- Funkcje też mogą być **constexpr**.

- Przykład:

```
const double pi = 3.141'592'653'589'793;
```

# TYPY DANYCH

- Każdy nazwany obiekt, który deklarujemy w programie musi być jakiegoś typu.
- Deklaracja – informuje kompilator, że dana nazwa reprezentuje obiekt jakiegoś typu, ale nie rezerwuje dla niego miejsca w pamięci.
- Definicja zaś – dodatkowo rezerwuje miejsce. Definicja jest miejscem w programie, gdzie tworzony jest obiekt.
- Systematyka typów w C++:
  - typy wbudowane (podstawowe),
  - typy zdefiniowane przez użytkownika,
  - typy pochodne.

# TYPY O PRECYZYJNIE ZDEFINIOWANEJ SZEROKOŚCI

- Typy całkowite ze znakiem:  
`int8_t, int16_t, int32_t, int64_t.`
- Typy całkowite bez znaku:  
`uint8_t, uint16_t, uint32_t, uint64_t.`
- Największy typ całkowity dostępny na danym komputerze:  
`intmax_t, uintmax_t`
- Typy te zostały umieszczone w pliku nagłówkowym `<cstdint>` w przestrzeni nazw `std.`
  
- Kodowanie znaków Unicode na konkretnie 16 lub 32 bitach:  
`char16_t, char32_t`

# BINARNA POSTAĆ LICZBY CAŁKOWITEJ

■ Literał binarny: 0b...

■ Pisanie binarne:

```
#include <iostream>
```

```
#include <bitset>
```

```
int main() {
```

```
    int a = -58, b = a>>0b11, c = -315;
```

```
    std::cout << "a = " << std::bitset<8>(a) << std::endl;
```

```
    std::cout << "b = " << std::bitset<8>(b) << std::endl;
```

```
    std::cout << "c = " << std::bitset<16>(c) << std::endl;
```

```
}
```

# JEDNOLITA INICJALIZACJA

- Celem jednolitej inicjalizacji w C++ jest jeden uniwersalny sposób inicjalizacji danych, włącznie z kontenerami.
- Składnia inicjalizacji z wykorzystaniem nawiasów klamrowych jest teraz dozwolona we wszystkich przypadkach.

- Przykłady:

```
int var1 {5};  
const int con2 {10};  
int arr3[] {1, 2, var1, var1 + con1};  
const Point pt4 {10, 20};  
std::complex<double> c5 {4.0, 2.0};  
std::vector<std::string> cities6 = {  
    "Wroclaw",  
    "Cracow"  
};
```

# JEDNOLITA INICJALIZACJA

- Jednolita inicjalizacja zabrania inicjalizacji z wykorzystaniem zawężających konwersji, za wyjątkiem niejawnej konwersji, która zawęża typ.

- Przykład:

```
int arr1[] = { 1, 2, 4.5 };  
// OK in C++98; error in C++11
```

- Jednolita inicjalizacja wymusza inicjalizację wartością (ang. value initialization), co oznacza, że nawet lokalne zmienne typów podstawowych, które posiadają niezdefiniowaną wartość początkową, są inicjalizowane zerem (albo wartością `nullptr` w przypadku wskaźników).

- Przykłady:

```
double d{}; // d jest równe 0.0  
char *ptr{}; // ptr jest równe nullptr  
int tab[4]{}; // tab jest równe [0, 0, 0, 0]
```

# JEDNOLITA INICJALIZACJA

- Inicjalizacja agregatów za pomocą klamer przebiega dokładnie w taki sam sposób jak w C++98 i powoduje inicjalizację składowych według kolejności ich definicji w agregacie; liczba elementów na liście musi być równa lub mniejsza od ilości elementów w agregacie.
- Jednolita inicjalizacja klas czy struktur nie będących agregatami powoduje wywołanie konstruktora (z ilością i typem parametrów pasujących do konstruktora).



# OGRANICZANIE ZASIĘGU ZMIENNEJ W INSTRUKCJI WARUNKOWEJ IF-ELSE

- Dobrym zwyczajem w programowaniu jest maksymalne ograniczanie zasięgu zmiennej. Jednak czasami trzeba najpierw otrzymać wartość, która będzie mogła być dalej przetwarzana tylko po spełnieniu określonego warunku.

- Przykład:

```
if (auto itr(char_map.find(c)); itr != char_map.end())  
{  
    // itr jest poprawny i zostanie wykorzystany  
    // do pewnych operacji  
}  
else {  
    // itr jest iteratorem końcowym i nie wolno  
    // z niego korzystać  
}
```

- Zmiennej lokalnej zadeklarowanej w tuż przed warunkiem w instrukcji warunkowej można potem używać w dalszych zagnieżdżonych albo kaskadowo połączonych instrukcjach warunkowych.

# OGRANICZANIE ZASIĘGU ZMIENNEJ W INSTRUKCJI WYBORU SWITCH-CASE

- Instrukcje if-else i switch-case wraz z inicjalizatorami to lukier syntaktyczny (osiągnięcie takiego efektu jest możliwe po ujęciu kodu w dodatkowy nawias klamrowy).

- Przykład:

```
switch (char c (getchar())); c) {  
case 'a': move_left(); break;  
case 's': move_back(); break;  
case 'w': move_fwd(); break;  
case 'd': move_right(); break;  
case 'q': quit_game(); break;  
//...  
}
```

- Krótszy cykl życiowy zmniejsza liczbę zmiennych w zasięgu, co z kolei przekłada się na większą przejrzystość kodu i jego łatwiejszą refaktoryzację.

# REFERENCJE

- Operatory, które umożliwiają tworzenie typów pochodnych:
  - `()` funkcja
  - `[]` tablica
  - `*` wskaźnik
  - `&` referencja
  - `&&` r-wyrażenie (wartość tymczasowa)
- Referencja odnosi się do istniejącego w pamięci obiektu.
- Referencję trzeba zainicjalizować.
- Referencja nie może zmienić obiektu, z którym została związana w czasie inicjalizacji.
- Referencję implementuje się jako stały wskaźnik.

# REFERENCJE

- Definicja referencji:

```
typ &ref = obiekt;
```

- Przykład referencji:

```
int x = 4;  
int &r = x;
```

- Referencje mają zastosowanie głównie jako argumenty funkcji i jako wartości zwracane przez funkcje.

- Przykład funkcji, która zamienia miejscami wartości zewnętrznych zmiennych:

```
void zamiana (double &a, double &b) {  
    double c = a;  
    a = b;  
    b = c;  
}
```

# NAPISY I ŁAŃCUCHY ZNAKOWE

- C-string to napis umieszczony w tablicy typu `const char[]` zakończony znakiem o kodzie 0 `'\0'`.
- Łańcuch znakowy to napis typu `string` przechowywany w obiekcie.
- Stringi są zadeklarowane w pliku nagłówkowym `<string>`.
- Stringi można ze sobą konkatelować za pomocą operatorów `+` i `+=`.
- W przypadku stringów nie trzeba się martwić o miejsce na napis – zostanie ono automatycznie zaalokowane.

# WEKTOR

- Obiekt klasy `vector<T>` zastępuje tablicę obiektów typu `T`.
- Szablon klasy `vector<>` jest zdefiniowany w pliku nagłówkowym `<vector>`.
- Wektor jest zaimplementowany jako tablica dynamiczna.

- Deklaracja:

```
vector<T> u;  
vector<T> v = {t0, t1, ...};  
const vector<T> w = {t0, t1, ...};
```

- Do komórek wektora odwołujemy się za pomocą operatora indeksowania, albo funkcji składowej `at()`:

```
v[i]  
v.at(i)
```

# PĘTLA FOR OPARTA NA ZAKRESIE

- Zakresy reprezentują kontrolowaną listę pomiędzy dwoma jej punktami. Kontenery uporządkowane są nad zbiorem koncepcji zakresu i dwa iteratory w kontenerze uporządkowanym także definiują zakres.
- Nowa pętla `for` została stworzona do łatwej iteracji po zakresie; jej ogólna postać jest następująca:  
`for (TYP &x: kolekcja<TYP>) instrukcja;`
- Przykład:  

```
int moja_tablica[5] = {1, 2, 3, 4, 5};  
for(int &x: moja_tablica) { x *= 2; }
```
- Pierwsza sekcja nowego `for` (przed dwukropkiem) definiuje zmienną, która będzie użyta do iterowania po zakresie. Zmienna ta, tak jak zmienne w zwykłej pętli `for`, ma zasięg ograniczony do zasięgu pętli.
- Druga sekcja (po dwukropku), reprezentuje iterowany zakres. W tym przypadku, zwykła tablica jest konwertowana do zakresu. Mógłby to być na przykład `std::vector` albo inny obiekt spełniający koncepcję zakresu.

# PARY

- Klasa `pair` umożliwia potraktowanie dwóch wartości jako pojedynczego elementu.
- Struktura `pair` zdefiniowana jest w pliku nagłówkowym `<utility>`.
- Struktura `pair` zawiera zagnieżdżone definicje typów `first_type` i `second_type`, reprezentujące typy składowych odpowiednio dla pól `first` i `second`.
- Szablon funkcji `make_pair()` umożliwia tworzenie pary wartości bez jawnego określania typów.
- Przykłady:  

```
std::pair<int, float> p(51, 3e-4);  
auto q = std::make_pair(53, "witaj");
```



# TYP `VOID`

- Typ `void` informuje nas o braku typu.
- Typ `void` jest typem fundamentalnym, jednak nie wolno zadeklarować zmiennej typu `void`.
- Słowo `void` może wystąpić jako typ prosty w deklaracji typu złożonego:
  - `void *ptr;`  
oznacza wskaźnik do pamięci na obiekt nieznanego typu;
  - `void fun ();`  
oznacza, że funkcja nie będzie zwracała żadnego wyniku.

# WSKAŹNIK PUSTY `NULLPTR`

- W starszym C++, stała 0 spełnia dwie funkcje: stałej całkowitej i pustego wskaźnika; programiści obchodzili tę niejednoznaczność za pomocą identyfikatora `NULL` zamiast 0.
- W języku C identyfikator `NULL` jest makrem preprocesora zdefiniowanym jako `((void*) 0)`; w starym C++ niejawną konwersję z `void*` do wskaźnika innego typu jest niedozwolona, więc nawet takie proste przypisanie jak `char *c = NULL` mogłoby być w tym przypadku błędem kompilacji.
- Sytuacja komplikuje się w przypadku przeciążania:  
`void foo(char*);`  
`void foo(int);`  
Gdy programista wywoła `foo(NULL)`, to wywoła wersję `foo(int)`, która prawie na pewno nie była przez niego zamierzona.

## WSKAŹNIK PUSTY `NULLPTR`

- Wskaźnik pusty, który nie pokazuje na żaden obiekt w pamięci zapisujemy jako `nullptr` – zastępuje makro `NULL` albo `0` (jest to adres o wartości 0 – adres pierwszej komórki w pamięci operacyjnej) i jest typu `nullptr_t`.
- Wskaźnik `nullptr` nie może być przypisany do typów całkowitych, ani porównywany z nimi.
- Wskaźnik `nullptr` może być porównywany z dowolnymi typami wskaźnikowymi.

# STOS I STERTA

- Stos to pamięć zarządzana przez program.
- Zmienne lokalne tworzone w instrukcji blokowej są automatycznie usuwane przy wychodzeniu z bloku.

- Przykład:

```
{  
    int p = 123;  
    ...  
}
```

- Sterta to pamięć, którą zarządza programista.
- Programista przydziela obszar pamięci dla zmiennej operatorem **new**, ale musi pamiętać o zwolnieniu tej pamięci operatorem **delete**.

- Przykład:

```
{  
    int *p = new int(123);  
    ...  
    delete p;  
}
```

# STANDARDOWE WEJŚCIE I WYJŚCIE

- W bibliotece standardowej są zdefiniowane cztery obiekty związane ze standardowym wejściem i wyjściem:
  - `cin` standardowe wejście,
  - `cout` standardowe wyjście,
  - `clog` standardowe wyjście dla błędów,
  - `cerr` niebuforowane standardowe wyjście dla błędów.
- Do czytania ze strumienia wejściowego został zdefiniowany operator `>>`:  
`cin >> zmienna;`
- Do pisania do strumieni wyjściowych został zdefiniowany operator `<<`:  
`cout << wyrażenie;`  
`clog << wyrażenie;`  
`cerr << wyrażenie;`
- Operatory czytające `>>` ze strumienia i piszące `<<` do strumienia można łączyć kaskadowo w dłuższe wyrażenia (wielokrotne czytanie albo pisanie).

```
double delta = b * b - 4 * a * c;  
if (delta < 0)  
    throw out_of_range("delta < 0");
```

# WYJĄTKI

- W przypadku dostarczenia błędnych (niezgodnych ze specyfikacją) argumentów do funkcji należy zgłosić wyjątek.
- Wyjątki zgłaszamy instrukcją `throw`:  
`throw wyjątek;`
- Powinno używać się prostych wyjątków zdefiniowanych w pliku nagłówkowym `<stdexcept>`:
  - `domain_error` – wartość spoza dziedziny;
  - `invalid_argument` – błędny argument;
  - `length_error` – niedopuszczalna długość;
  - `out_of_range` – wartość spoza zakresu.
- Przykład:

```
double delta = b * b - 4 * a * c;  
if (delta < 0)  
    throw out_of_range("delta < 0");
```