Język C++ (Extra)

Kolory w konsoli

Kolorowanie czcionki w oparciu o kody ANSI (*C++ / Python*) możliwe jest w *Linux* i *Windows 10/11* (nie działa w Windows 8.1). Niestety, w niektórych wersjach *Windows 10* (po niektórych aktualizacjach) możliwość ta jest wyłączona. Aby włączyć ją na stałe, należy utworzyć następujący klucz w rejestrze Windows:

[HKEY_CURRENT_USER\Console] → "**VirtualTerminalLevel**"=dword:0000000**1**

Alternatywnie, można też wydać polecenie w PowerShell (w trybie Administratora):

PS C:\Windows\system32> **Set-ItemProperty** *HKCU:\Console* VirtualTerminalLevel **-Type** *DWORD 1*

Można też aktywować taką opcję wewnątrz programu (kolorowanie czcionki będzie możliwe tylko w tym programie). Przykład poniżej:

```
#include <iostream>
#include <windows.h> // Potrzebne do aktywacji kolorów w konsoli

using namespace std;

int main(void) {
    // Tworzymy tzw. uchwyt do tego, co będzie pojawiać się na konsoli (do bufora konsoli):
    HANDLE konsola = GetStdHandle(STD_OUTPUT_HANDLE);
    // Aktywujemy virtualny terminal i to, co będzie się na nim pojawiać:
    #ifndef ENABLE_VIRTUAL_TERMINAL_PROCESSING // Jeśli nie jest zdefiniowany, to:
    #define ENABLE_VIRTUAL_TERMINAL_PROCESSING 0x0004
    #endif
    // Aby powyższe działało, musimy aktywować i to, co poniżej:
    #ifndef ENABLE_PROCESSED_OUTPUT // Jeśli nie jest zdefiniowany, to:
    #define ENABLE_PROCESSED_OUTPUT 0x0001
    #endif

    // Wartość trybu (intup lub output). Słowo "dw" to skrót od "Display Window",
    // jest to jednak nazwa zmiennej, i może być inna:
    DWORD dwMode = 0;
    dwMode |= ENABLE_PROCESSED_OUTPUT | ENABLE_VIRTUAL_TERMINAL_PROCESSING;
    SetConsoleMode(konsola, dwMode);

    cout << "To jest test: \033[1;31;40mKolor czerwony\033[0m.\n\n";
    cout << "Naciśnij ENTER, aby zakończyć...";
    system("pause > nul");
}
```

Kolorowe znaki można także uzyskać korzystając z funkcji **SetConsoleTextAttribute()**:

```
#include <iostream>
#include <windows.h> // Potrzebne do kolorowania składni, tylko Windows 10/11
```

```
using namespace std;

int main(void) {
    // Tworzymy tzw. uchwyt do konsoli:
    HANDLE konsola = GetStdHandle(STD_OUTPUT_HANDLE);
    // Zapisujemy domyślną kolorystykę konsoli do zmiennej „domyslnaKonsola”:
    CONSOLE_SCREEN_BUFFER_INFO domyslnaKonsola;
    GetConsoleScreenBufferInfo(konsola, &domyslnaKonsola);

    // Test kolorów:
    cout << "TEST KOLORÓW z użyciem SetConsoleTextAttribute():\n\n";
    SetConsoleTextAttribute(konsola, 1);
    // Można zlikwidować „\n”, aby mieć różne kolory w jednej linii:
    cout << "Tekst jest niebieski\n";
    SetConsoleTextAttribute(konsola, 2);
    cout << "Tekst jest zielony\n";
    SetConsoleTextAttribute(konsola, 4);
    cout << "Tekst jest czerwony\n";

    // Przywracamy domyślne ustawienia konsoli:
    SetConsoleTextAttribute(konsola, domyslnaKonsola.wAttributes);

    cout << "Naciśnij ENTER, aby zakończyć...\n";
    system("pause > nul");
}
```

Tabela kolorów (końcowy efekt = foreground + background * 16, np. 4+(15*16) = **244**, co daje czerwony na białym):

Kod	***Kolor***	***ANSI Codes***
0	**black**	
1	**blue**	
2	**green**	
3	**cyan**	\033[1;**36**;40m *jakiśTekst* \033[0m
4	**red**	\033[1;**31**;40m *jakiśTekst* \033[0m
5	**magenta**	\033[1;**35**;40m *jakiśTekst* \033[0m
6	**brown**	
7	**white**	\033[1;**37**;40m *jakiśTekst* \033[0m
8	**gray**	\033[1;**30**;40m *jakiśTekst* \033[0m
9	**light blue**	\033[1;**34**;40m *jakiśTekst* \033[0m
10	**light green**	\033[1;**32**;40m *jakiśTekst* \033[0m
11	**light cyan**	

Kod	Kolor	ANSI Codes
12	**light red**	
13	**light magenta**	
14	**yellow**	\033[1;**33**;40m *jakiśTekst* \033[0m
15	**light white**	
n+128	**blinking**	\033[**5**;**37**;40m *jakiśTekst* \033[0;**37**;40m
	pochyłe	\033[**3**;**37**;40m *jakiśTekst* \033[0;**37**;40m
	pogrubione	\033[**1**;**37**;40m *jakiśTekst* \033[0;**37**;40m
	podkreślenie	\033[**2**;**37**;40m *jakiśTekst* \033[0;**37**;40m

Predefiniowane funkcje, instrukcje i znaki

\” - znak “;

\\ - znak \;

\13 - numer znaku w kodzie ASCII, w tym przypadku ENTER;

\a - beep (domyślny dźwięk systemu);

\n - nowa linia;

%8s - osiem spacji;

Beep(1000, 5000); - generuje dźwięk;

char napis[] = "To jest jakis napis"; - zmienna łańcuchowa, kompilator sam policzy ilość znaków, więc puste [];

clrscr(); - czyści ekran;

cin - zapisuje do zmiennej, ale nie „białe znaki” (np. spacje); należy unikać mieszania w jednym kodzie „cin” i „getline” (albo jedno, albo drugie);

cprintf(“Tekst”); - wprowadzanie tekstu w kolorach (*color printing*);

DeleteFile(plik) - usuwa plik;

dlugosc = strlen(tekst); - długość napisu;

fprintf(stdout, “Tekst”); - drukuje na ekran;

fprintf(stdprn, “Tekst”); - drukuje na drukarkę;

fscanf(“%f”, &x); - pobiera liczbę z klawiatury;

getch(); - czyli *„get char”*; czeka na naciśnięcie klawisza (odpowiednik “*pause*”);

gets(string1); - pobierz ciąg;

if(getch() == '.') - jeśli nie zostanie naciśnięta kropka, wykonaj...;

return 0 - umieszczone na końcu funkcji informuje, czy wykonanie funkcji powiodło się; nazwa funkcji musi być wtedy poprzedzona „int” (gdyż zwraca liczbę całkowitą);

sound(5000); delay(100); nosound(); - generuje dźwięk;

strcat(napis1, napis2); - łączenie napisów;

strcpy(string1, string2); - skopiowanie całego łańcucha z ciągu1 do ciągu2;

strncpy(nowyNapis, staryNapis, 3); - wytnij 3 znaki ze starego napisu i wklej do nowego;

strstr(gdzieSzukać, "coSzukać"); - wyszukuje w ciągu jakieś znaki;

system(“mkdir nazwaFolderu”); - odwołanie do polecenia DOS;

using namespace std; - umieszczenie tej linii przed nazwą funkcji main(), zwalnia przed podawaniem pełnych nazw, np. zamiast *std::cout << „Tekst”*, można użyć *cout << „Tekst”*;

vector - dynamiczna tablica, której wielkość możemy zwiększać w każdym momencie

void nazwaFunkcji - funkcja nie będzie zwracała systemowi żadnych wyników powodzenia czy niepowodzenia działania;

Uwaga:
Exit value: 0 - brak błędów;

Kompilacja w Linux

$ **g++ plik.cpp -o plik.exe** (tworzy plik wykonywalny)

Uwaga: GCC służy domyślnie do kompilowania plików *.c. Oba jednak kompilatory wchodzą w skład GNU Compiler.

Obiektowość

klasa - zbiór wszystkich cech obiektu (które chcemy uwzględnić, ale bez konkretnych wartości!), np. waga, wiek, nazwa; klasa definiuje, jaki zestaw cech może mieć obiekt; obiektów w świecie jest zbyt dużo, aby dla każdego z nich definiować osobne zmienne (każdy obiekt byłby wtedy klasą zawierającą jeden obiekt: samego siebie); zauważamy, że wiele z nich ma cechy wspólne; możemy utworzyć klasę tego wszystkiego, co ma jakiekolwiek wspólne cechy; różne klasy mogą posiadać zmienne o tych samych nazwach; mogą także posiadać metody o tych samych nazwach; przykładem klasy jest wyrażenie „ssak”, zaś konkretne zwierzę, to obiekt tej klasy;

metoda - to funkcja zdefiniowana wewnątrz klasy, np. *nazwaObiektu.nazwaMetodyFunkcji(arg1, arg2)*; metody powinny zawsze zawierać mechanizmy kontroli błędów (danych wejściowych);

konstruktor - wewnątrz klasy tworzy metodę o takiej samej nazwie jak klasa w celu inicjalizacji zmiennych / parametrów i nadania im wartości startowych / domyślnych, np.: KlasaXYZ::KlasaXYZ (int x) {} lub niejawnie po prostu KlasaXYZ(int x) {} lub z wartościami domyślnymi KlasaXYZ(int x=56) {}; ten typ metody jest wywoływany automatycznie (niejawnie) podczas tworzenia obiektu; konstruktor nie ma typu danych ani nawet *void*; w tym momencie zmienne, których będzie używał obiekt są już zadeklarowane; wywołanie takiej metody w *main()* (utworzonej przez konstruktor) to: NazwaKonstruktora obiekt(56) zamiast NazwaKlasy obiekt; obiekt.nazwaMetody(56); czyli: przy tworzeniu obiektu wymuszamy podanie wartości domyślnych (co jest dobrym rozwiązaniem), powinniśmy zawsze nadawać atrybutom wartości domyślne; konstruktor może wywołać inne metody ze swojej klasy;

destruktor - specjalny typ metody, która jest wywoływana przed usunięciem obiektu: public: ~nazwaObiektu(){}; sprząta po obiekcie (mogą istnieć wskaźniki wskazujące na obszar pamięci, w której nie ma już obiektu); może wywołać inne metody ze swojej klasy;

this - wskaźnik, który wskazuje na konkretny obiekt (każdy obiekt ma swój własny wskaźnik), dla którego została wywołana metoda (funkcja wewnątrz klasy); określa zasięg zmiennej, do której chcesz się odwołać; nie można go modyfikować; zastępuje nazwę metody w obiekcie odwołując się do atrybutów, np. this.**name**, this.**age**; czyli możemy przypisać zmienne publiczne do zmiennych prywatnych; powinno używać się „this” tylko wtedy, gdy nie ma innego wyjścia (np. nazwa parametru w metodzie jest taka sama, jak nazwa zmiennej w klasie), np.:

```
class nazwaKlasy {
        private:
        int wiek;
        public:
        void pies(int wiek) {
                this→wiek = wiek; // Będę mógł w funkcji main() odwołać się do prywatnej
                                  // zmiennej„wiek”.
```

```
    }
    // lub też jeszcze inaczej:
    void kot(int iloscLat) {
        this→wiek = iloscLat; // Będę mógł w funkcji main() odwołać się do „wiek”.
        // Argument przekazany do parametru „iloscLat” zostanie przypisany do „wiek”.
        // Wskaźnik „this” umożliwia więc przypisanie do zmiennych prywatnych jakiś
        // wartości.
    }
};
```

new nazwaObiektu - operator „new” tworzy nowy obiekt na „stercie”;

private - zmienne (czyli atrybuty) w tej sekcji, a nawet metody, można użyć **bezpośrednio** tylko wewnątrz klasy; tylko metody klasy mają dostęp do tych zmiennych / metod (lub klasy zaprzyjaźnione); zabezpiecza to zmienne przed przypadkową zmianą wartości przez funkcje spoza klasy; określa się to mianem „hermetyzacji”; klauzula „private” jest domyślna (jeśli użytkownik nie zadeklarował żadnej klauzuli); atrybuty powinny zawsze być prywatne, chyba że jest ważny powód, aby były publiczne;

protected - dostęp do tej zawartości mają metody klasy, ale także klasy dziedziczące;

public - do zmiennych w tej sekcji można odwołać się poza klasą;

friend typ **nazwaZewnętrznejFunkcji()** - deklaracja umieszczona w klasie (plik nagłówkowy) w sekcji *public*; sprawia, że ta zewnętrzna funkcja jest zaufana / zaprzyjaźniona i ma dostęp do prywatnych atrybutów klasy; do tej funkcji, jako argumenty wysyłamy całe obiekty; może ona być przyjacielem kilku klas;

friend class **drugaKlasa** - pierwsza klasa deklaruje, że jej przyjacielem jest druga klasa;

getter / **setter** - funkcje pobierające lub ustawiające wartości atrybutów;

Uwaga 1: Klasy można zapisywać w osobnych plikach (np. nazwaKlasy.cpp), a spis treści jej metod umieszczać (dla przyspieszenia działania) w pliku z rozszerzeniem **.h* i odwoływać się do nich za pomocą #include <nazwaKlasy.h> (rozszerzenie **.h* oznacza właśnie „nagłówki” metod). Pliki **.h* zawierają także komentarze, co dana metoda robi (stanowią więc „spis treści” przy większych projektach).

Uwaga 2: Jeśli podpowiedzi w edytorze kodu (np. *Code::Blocks*) mają czerwony kwadracik ■, to znaczy, że są to zmienne prywatne i nie możemy się do nich odwołać. Możemy odwoływać się tylko do tych zmiennych, które mają zielony kwadrat ■ (zmienne publiczne).

Wskaźniki (*pointers*) i referencje

Wskaźnik (*pointer*) to 4-bajtowa liczba, która reprezentuje miejsce (adres) w pamięci i wskazuje na np. jednobajtowy *char*. Pointer nie zwraca wartości zmiennej, ale jej adres w pamięci. Typ danych pointera musi być taki sam, jak typ danych zmiennej, do której odwołuje się. Wskaźnik można przypisać do innej zmiennej niż na początku. Za pomocą wskaźnika można także zmienić wartość zmiennej, na którą wskazuje. Wskaźniki tworzymy za pomocą gwiazki: ***wskaźnik**.

Referencja to rodzaj aliasu do obiektu, dzięki któremu możemy odwoływać się do danego obiektu bez tworzenia jego kopii; wszelkie operacje i zmiany dokonane za pomocą referencji mają wpływ na oryginalny obiekt; jest na stałe przypisana do zmiennej (nie można referencji przypisać do innej zmiennej). Referencje tworzymy za pomocą znaku „*and*”: **&referencja**.

char *wskaźnikA - wskaźnik (*pointer*) do *char*;

char **wskaźnikB - wskaźnik do wskaźnika do *char*;

char ***wskaźnikC - wskaźnik do wskaźnika do wskaźnika do *char*;

Info: Możliwy jest także zapis „*pointer * zmienna*” lub „*pointer* zmienna*”.

Przykład 1:
char **tablica[]** = "*Jakiś tekst*";
tablica[2**]** = '*Z*'; // Trzeci element zamieniony zostanie na „Z”

ale:

char ***tablica** = "*Jakiś tekst*"; // To jest wskaźnik do „tablica”, a nie sama „tablica”
tablica[2**]** = '*Z*'; // To się nie uda, wystąpi błąd; wskaźniki nie pozwalają na indeksowanie;
// można jedynie odczytać;

Przykład 2:
string **nazwaZmiennej** = "*Jakiś tekst*";
string ***nazwaPointera** = &**nazwaZmiennej**; // Ustawiamy wskaźnik.
// Znak „&” jest tutaj konieczny.
int **wartośćZmiennej** = ***nazwaPointera**; // Dereferencja wskaźnika
cout << **nazwaPointera** << "\n"; // Zwróci np. 0x7ffcc7fc6ea0
cout << ***nazwaPointera** << "\n"; // Zwróci wartość pointera np. „Jakiś tekst”
cout << &**nazwaZmiennej** << "\n"; // Również zwróci 0x7ffcc7fc6ea0

***nazwaPointera** = „*Nowy tekst*”; // Zmodyfikuje wartość zmiennej

Tablice

Rodzaje tablic:

- **statyczne** - rozmiar tablicy musi być określony w momencie kompilacji programu, a potem jest już niezmienny;
- **dynamiczne** - z dynamiczną alokacją pamięci w sytuacji, gdy to użytkownik podaje rozmiar tablicy:

```
int rozmiarTablicy;
cout << "Podaj rozmiar tablicy: ";
cin >> rozmiarTablicy;
// Alokacja dynamiczna pamięci dla tablicy int-ów:
int *tablica = new int[rozmiarTablicy];
// Przypisanie wartości do tablicy:
for (int i=0; i<rozmiarTablicy; ++i) {
   tablica[i] = i * 2; // Jakaś wartość
}
// Zwolnienie dynamicznie zaalokowanej pamięci. Konieczne!
delete[] tablica;
```

- **vector** - kontener sam zarządza pamięcią, więc nie trzeba alokować i zwalniać pamięci:

```
int rozmiarTablicy;
cout << "Podaj rozmiar tablicy: ";
```

```
cin >> rozmiarTablicy;
vector<int> tablica(rozmiarTablicy);
// Przypisanie wartości do tablicy:
for (int i=0; i<rozmiarTablicy; ++i) {
   tablica[i] = i * 2; // Jakaś wartość
}
```

- **vector z obiektami** - należy pamiętać o tym, że klasa to także typ danych:

```
int rozmiarTablicy;
cout << "Podaj rozmiar tablicy: ";
cin >> rozmiarTablicy;
vector<nazwaKlasy> vectorObiektow;
// Dodanie obiektów do vectora:
for (int i=0; i<rozmiarTablicy; ++i) {
   vectorObiektow.push_back(nazwaKonstruktora(i*2, i*3)); // Przykładowe wartości
}

// Wyświetlenie zawartości vectora obiektów:
cout << "Zawartość vectora obiektów:" << endl;
// Wyrażenie „auto” rozpoznaje typ danych poszczególnych obiektów:
for (const auto &nazwaObiektu : vectorObiektow) {
   cout << "Zawartość wektora obiektów: (" << nazwaObiektu.x << ", " <<
   nazwaObiektu.y << ")" << endl;
}
```

Pętla zakresowa FOR

Wprowadzona w wersji C++11, pozwala iterować po elementach (np. tablicy) bez używania licznika (sama wykrywa ilość elementów):

```
int tablica[] = {1, 2, 3, 4, 5};
// Pętla przy każdym kolejnym przejściu przypisuje do zmiennej „liczba” każdy kolejny element
// tablicy. Zmienna „liczba” jest zmienną pomocniczą, służącą tylko do wypisania liczb
// wewnątrz pętli FOR.
for (int liczba : tablica) {
   cout << liczba << " ";
}
```

Struktury typów danych

```
struct nazwaTypu
{
        int x;
        float y;
        char c;
};
```

Odwołanie:

```
nazwaTypu.x = 10;
nazwaTypu.y = 1.3;
```

nazwaTypu.**c** = 'a';

struct *nazwaTypu2*
{
	char **miasto**[32]; (--> 32 znaki możliwe)
	char **nazwisko**[24]; (--> 24 znaki)
	int **wiek**[2];
	bool **plec**;
}

Odwołanie:
nazwaTypu.**miasto**

Uwaga: Struktury danych mogą zawierać elementy innych struktur (mogą się do nich odwoływać).

Okienka

WinMain() - odpowiednik funkcji main() dla programów DOS-owych;
WM_CREATE - tworzy komunikat (*Windows Message: Create!*);
WS_VISIBLE - okienko będzie widoczne (*Window Style: Visible*);
MB_... - elementy okienka komunikatów (*Message Box*);
TextOut(0, 0, "Napis: np. Hello world.", dlugosc_tekstu); - napis w okienku;
DestroyWindow(hWnd); - zamyka okno;
hControlWnd = CreateWindow ("BUTTON", " Napis_na_Klawiszu ", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE, 10, 20, 30, 40, hWnd, ID_Elem, hInstance, 0); - okienko z przyciskiem;

Biblioteki

Aby program wykrywał potrzebne składniki podczas kompilacji, należy ustawić odpowiednie ścieżki w sekcji *„Settings / Compiler”*:
Zakładka „Compiler” (tu kopiujemy pliki *.h):
Visual Studio: *C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include*
Microsoft SDK: *C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\include*
Code::Blocks: *C:\Program Files (x86)\CodeBlocks\MingGW\include*

Zakładka „Linker” (tu kopiujemy pliki *.a):
Visual Studio: *C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\lib*
Microsoft SDK: *C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\lib*
Code::Blocks: *C:\Program Files (x86)\CodeBlocks\MingGW\lib*

Zakładka „Resource Compiler”:
Visual Studio: *C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include*
Microsoft SDK: *C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\include*
Code::Blocks: *C:\Program Files (x86)\CodeBlocks\MingGW\include*

W Linuksie zazwyczaj wystarczy podać ścieżkę do pliku wykonywalnego kompilatora:

Settings / Compilator / zakładka „Toolchain executables” / Compiler’s installation directory → /usr/bin

Jeśli nadal ma problemy z odnalezieniem plików / bibliotek, to musimy najpierw sprawdzić jakiej wersji kompilatora używamy:

\# **gcc -v**

Następnie ustawiamy dodatkowe ścieżki:

Settings / Compilator / zakładka „Toolchain executables” / Additional Paths → /usr/lib64/gcc/x86_64-pc-linux-gnu/4.9.4

/usr/lib64/gcc/

/usr/include/codeblock/.h*

/usr/lib32/.a*

Popularniejsze biblioteki:

conio.h - CONsole I/O, m. in. *clrscr()*, t*extcolor()*;

stdio.h - StanDard I/O, m. in. *printf()*;

graphics.h – tryb graficzny 2D (składnik biblioteki *winbgim*, przestarzały, należy skopiować go do folderu „*include*” kompilatora, oraz dodatkowo bibliotekę „*libbgi.a*” do folderu „*lib*” kompilatora); jako że w bibliotece są odwołania do *windows.h* – będzie działać tylko w systemie *Windows*. W parametrach linkera należy dodać opcje kompilacji: *-lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32*.

windows.h, windowsx.h, owl.h - budowanie okienek (*WinAPI* jest specyficzna dla *Windows*, nie działa w *Linux*);

winbgim.h - m.in. *delay()*;

iostream - m. in. za pisanie w konsoli;

string - m. in. za możliwość używania zmiennych typu string;

~~**string.h**~~ - to jest stara biblioteka języka C (nie używać!);

Bloki try{} / throw / catch{}

Blokiem **try{}** obejmujemy **każdą** instrukcję, która może spowodować błąd. Blok **catch{}** umieszczamy bezpośrednio po bloku try{}. W kodzie może występować wiele bloków try{} i catch{}, a każdy try{} może mieć wiele bloków catch{}.

Funkcja **throw** zgłasza / rzuca wyjątek bezpośrednio w bloku try{} (np. jeżeli wpisana liczba jest mniejsza od 0, to rzuć wyjątkiem). Czyli wyrzuca treść (jakiś kod) o możliwej przyczynie błędu, która potem zostaje przechwycona przez catch{}.

```
try {
        int x = 5;
        if (x == 5) {
                throw ”kodBłędu”;
                // Uwaga: Nie zawsze musimy jawnie rzucać wyjątkiem.
                // Może to za nas zrobić niejawnie sam język programowania.
        }
}
// Sami ustalamy, do jakiej zmiennej przypiszemy przechwycony wyjątek. Może to być np. „ex”.
// Jeśli wystąpi kilka wyjątków, do tej zmiennej zostanie przypisany pierwszy z nich.
// Wyjątki przechwytujemy kierując się typem danych:
catch (typDanych zmiennaDlaBłędu) {
        cout << ”Tu wypisać rzucony przez throw kod błędu: ” << zmiennaDlaBłędu << ”oraz
        komentarz dla użytkownika” << endl;
```

```
}

// Wyłapanie wyjątków rzuconych przez sam program (niezadeklarowanych przez nas):
catch (const std::invalid_argument& zmiennaDlaBłędu) {
        cout << "Jakieś wyjaśnienie" << zmiennaDlaBłędu.what() << endl;
}
```

Jeśli blok przechwytywania ma postać „**catch{…}**” - znaczy to, że przechwytuje każdy inny wyjątek, i umieszczamy go jako ostatni, np.:

```
catch(...) {
        cout << "Inny wyjątek" << endl;
}

// Uwaga: W C++ nie istnieje blok „finally”.
```

Czy string składa się tylko z liczb?

```
// Funkcja domyślnie zwraca prawdę, chyba że wykryje fałsz.
// Sprawdzany jest każdy znak z osobna (w pętli).
bool czyToJestLiczba(string liczbaJakoString) {
   for (int licznik = 0; licznik < liczbaJakoString.length(); licznik++) {
      if (isdigit(liczbaJakoString[licznik]) == false) {
      return false;
      }
   }
   return true;
}

int main() {
        string liczbaJakoString;

        cout << "Podaj liczbę: ";
        cin >> liczbaJakoString;
        if (czyToJestLiczba(liczbaJakoString)) {
                cout << "Jest to liczba." << endl;
        }
        else {
                cout << "To NIE jest liczba." << endl;
        }
}
```

Problemy i rozwiązania

Nie znaleziono obiektu libstdc++6.dll lub libgcc_s_sjlj-1.dll

Program został zbudowany z użyciem dynamicznych bibliotek. Należy zbudować go z użyciem statycznych bibliotek. Menu: *Settings / Compiler / Compiler Settings / Compiler Flags / Static linking [-static]*

Polskie znaki

W systemie *Linux*: wystarczy, że terminal będzie w trybie UTF-8, a także edytor *Code::Blocks* będzie w UTF-8 (*Settings / Editor / Encoding settings*). To wszystko.

W systemie *Windows*: kodowanie edytora powinno być w trybie UTF-8 (*Settings / Editor / Encoding Settings*), a opcja kompilatora powinna być ustawiona na *cp852* (bo *Wiersz Poleceń* wyświetla znaki w *cp852*). Aby kompilator ustawiony był na kodowanie *cp852*, należy: *Settings / Compiler / Compiler Settings / Other Compiler Options: -fexec-charset=cp852*.

Info:

- Aktualne kodowanie znaków można sprawdzić tutaj: *Edit / File encoding...*
- Procedura dotyczy tylko aplikacji konsolowych (nie „okienkowych”).

Drugi alternatywny sposób na polskie znaki, to umieszczenie w samym kodzie deklaracji:

```
#include <locale.h>
int main() {
setlocale(LC_CTYPE, ”Polish”);
```

Brak iostream

Kompilacja pliku kończy się błędem z powodu braku „iostream”:
C:\Users\krzysiu\Pulpit\cwiczenie.c|1|**fatal error: iostream**: No such file or directory|
Rozwiązanie: Zamień rozszerzenie pliku na *.cpp.

collect2: error: ld returned 1 exit status

Często powstaje przy przenoszeniu projektu.
Rozwiązanie: Usunąć pliki z *folderProjektu / obj / Debug / *.**

Sprawdzanie znaków wejściowych

```
if (zn >='A' && zn <='Z') cout << zn << " jest wielką literą" <<endl;
if (zn >='a' && zn <='z') cout << zn << " jest małą literą" <<endl;
if (zn >='0' && zn <='0') cout << zn << " jest cyfrą" <<endl;
```

Poza tym, do każdego klawisza przypisana jest liczba:
13 - ENTER
27 - ESC
49-57 - cyfra
65-90 - duża litera
97-122 - mała litera
na przykład:

```
if ((zn >=49) && (zn <=57)) cout << zn << „jest cyfrą” << endl;
```

Określić zakres liczb od 50 do 100

100>liczba>50

Dzielenie całkowite dużych liczb

Maksymalny zakres wyniku w C++ można wyrazić za pomocą typu *BigInt* (2^{64}). W przypadku większych liczb należy korzystać z trików.

- Czy 232^{123} jest podzielne całkowicie przez 53? Nie, gdyż każda liczba parzysta podniesiona do potęgi (obojętnie jakiej) – jest zawsze parzysta, więc nie będzie całkowicie podzielna przez liczbę nieparzystą.
- Czy 233^{123} jest całkowicie podzielne przez 52? Nie, gdyż każda liczba nieparzysta podniesiona do dowolnej potęgi – jest zawsze nieparzysta, więc nie będzie całkowicie podzielna przez liczbę parzystą.
- Dowolna liczba całkowita podniesiona do dowolnej potęgi będzie podzielna całkowicie przez dowolną liczbę, jeśli będzie przez nią podzielna bez podnoszenia do potęgi. Innymi słowy, podnoszenie do potęgi jest w tym przypadku nieistotne, pomijalne.

Podnoszenie do dużych potęg

Korzystając ze wzoru: $2^n = 2^{n-1} + 2^{n-1}$ możemy rozłożyć dużą potęgę na wartości cząstkowe, np.

$2^8 =$

$2^7 + 2^7 =$

$2^6 + 2^6 + 2^6 + 2^6 =$

$2^5 + 2^5 + 2^5 + 2^5 + 2^5 +2^5 + 2^5 + 2^5 =$

$2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 + 2^4 =$

$2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3$
$+ 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 + 2^3 =$

$2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2$
$+ 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 +$
$2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2 + 2^2$
$+ 2^2 + 2^2 + 2^2 + 2^2 + 2^2 =$

Zamiana stringa na małe litery

#include <algorithm>

string tekst = ”*Jakiś tam tekst*”;

transform(tekst.**begin()**, tekst.**end()**, tekst.**begin()**, ::**tolower)**;

// Parametry: początek danych wejściowych, koniec danych wejściowych, początek danych wyjściowych lub nowy początek lub wynik (jeśli chcemy zamienić dane wejściowe, to będzie taki sam), rodzaj transformacji.

Uwaga: Można także użyć funkcji **tolower()**, ale służy ona do zamiany liter (*char*, a nie *string*) i trzeba w tym przypadku użyć pętli (zamieniamy każdą z liter osobno).

Przeszukiwanie stringów w tablicy

Z użyciem pętli WHILE:

```
#include <iostream>
#include <algorithm> // Potrzebne do find()
using namespace std;

int main() {
    const string  tablicaOdpowiedziTwierdzacych[]  = {"Tak", "tak", "T", "t", "Yes", "yes", "y", "Y",
"taa", "ta", "ye"};
    string szukane;

    while (true) {
        cout << "Czy chcesz kontynuować (Tak/Nie): " << endl;
        // Pierwszy argument przekazuje wartość drugiemu (także naciśnięcie ENTER):
        getline(cin, szukane);
        // Find() porównuje każdy z elementów tablicy do szukanej wartości i zwraca "1" lub "0".
        // Begin = pierwszy element pierwszej przeszukiwanej tablicy;
        // End = ostatni element ostatniej przeszukiwanej tablicy (tu jest akurat tą samą).
        // Jeśli znajdzie, zatrzymuje się (znaleziony element nie jest tożsamy z ostatnim, więc
        // całość jest prawdą). Jeśli nie znajdzie i dojdzie do końca, wskaże na ostatni element
        // (zwróci go), a ostatni sprawdzany element będzie tożsamy z ostatnim elementem
        // tablicy, co znaczy, że nie znalazł (całość więc zwróci 0).
        // Find() - jeśli jest w stanie bezproblemowo przeszukać wszystkie elementy tablicy -
        // zawsze zwraca 1  bez względu na to, czy znajdzie czy nie (to jest tylko informacja,
        // że udało się przeszukanie). Dopiero porównanie z ostatnim elementem w tablicy daje
        // odpowiedź, czy znaleziono.
        bool jestObecne = find(begin(tablicaOdpowiedziTwierdzacych),
        end(tablicaOdpowiedziTwierdzacych), szukane) != end(tablicaOdpowiedziTwierdzacych);
        if (jestObecne == 0) {
            break;
        }
    }
    return 0;
}
```

Z użyciem instrukcji warunkowej IF:

```
#include<iostream>
#include<algorithm> // Potrzebne do find()
using namespace std;

int main() {
    string  tablicaOdpowiedziTak[]  = {"Tak", "tak", "T", "t", "Yes", "yes", "y", "Y", "taa", "yee",
    "ta", "Ta"};
    string szukane;

    cout << "Wpisz szukane słowo: " << endl;
    // Find() porównuje każdy z elementów tablicy do szukanej wartości i zwraca "1" lub "0".
    // Begin = pierwszy element pierwszej przeszukiwanej tablicy;
    // End = ostatni element ostatniej przeszukiwanej tablicy (tu jest akurat tą samą).
```

```
    // Jeśli znajdzie, zatrzymuje się (znaleziony element nie jest tożsamy z ostatnim, więc
    // całość jest prawdą). Jeśli nie znajdzie i dojdzie do końca, wskaże na ostatni element
    // (zwróci go), a ostatni sprawdzany element będzie tożsamy z ostatnim elementem
    // tablicy, co znaczy, że nie znalazł (całość więc zwróci 0).
    // Find() - jeśli jest w stanie bezproblemowo przeszukać wszystkie elementy tablicy -
    // zawsze zwraca 1  bez względu na to, czy znajdzie czy nie (to jest tylko informacja,
    // że udało się przeszukanie). Dopiero porównanie z ostatnim elementem w tablicy daje
    // odpowiedź, czy znaleziono.
    bool jestObecne = find(begin(tablicaOdpowiedziTak), end(tablicaOdpowiedziTak), szukane)
    != end(tablicaOdpowiedziTak);

    if (jestObecne) {
        cout << "Wyrażenie JEST w tablicy!" << endl ;
    }
    else {
        cout << "Wyrażenie NIE jest obecne w tablicy!" << endl;
    }
    return 0;
}
```

Wielkość czcionki w linuksowym Xterm

Settings / Enviroment / General settings / Terminal to launch console programs: xterm -fs 30 -fa 'Hack' -T $TITLE -e

Dynamiczna zmiana czcionki w Code::Blocks

LewyCRTL + ruchy myszki *UP/DOWN*

Deaktywacja sprawdzania pisowni

Windows → Menu: *Plugins / Manage Plugins / SpellChecker* → Disable.

Dokowanie okienka Logs & others

Sposób 1: Należy zmniejszyć główne okno, a następnie najechać okienkiem *Logs* na dolną krawędź głównego okna.

Sposób 2: *View / Perspectives / Delete current*.

Tablica jako argument

Bez użycia wskaźników i referencji:

```
void funkcjaPrzyjmujacaTablice(int tablica[], int rozmiar) {...}

int main() {
    int mojaTablica[] = {1, 2, 3, 4, 5};
    int rozmiarTablicy = sizeof(mojaTablica) / sizeof(mojaTablica[0]); // Zwracane w bytach,
    // czyli: każda z liczb (typ int) ma 4 byte'y, więc 4x5 = 20.
    // Dlatego dzielimy ją znowy przez element w tablicy.
    // Alternatywna metoda (ale działa tylko w przypadku tablic dynamicznych, w kontenerze
```

```
    // wektora i tylko jako referencja do tablicy):
    // void funkcjaPrzyjmujacaTablice(vector<int> &mojaTablica) {
    // rozmiarTablicy = mojaTablica.size(); }
    // funkcjaPrzyjmujacaTablice(mojaTablica);
    funkcjaPrzyjmujacaTablice(mojaTablica, rozmiarTablicy);
    return 0;
}
```

Inne

double wynik = pow(liczbaUzytkownika, 2); - podniesienie liczby do potęgi drugiej (zmienna *liczbaUzytkownika* musi także być typu *double*)

double wynik = sqrt(9); - pierwiastek z 9 (pierwiastkowana zmienna musi być typu *double*)