AiSD Laboratorium 1 COMMENTS

Ten plik krótko omawia wybrane klasy języka C++, które będa przydatne na laboratorium. Jest to omówienie skrótowe, w razie wątpliwości należy skonsultować sie z podręcznikiem albo dokumentacją.

Spis treści

Zastosowana metoda to ,,przez przykład''. Ten plik można traktować jako źródło schematów do wykorzystania w innych programach.

Uwaga: Język C++ ma już ze 40 lat. Od standardu C++11 zaszły duże zmiany, które potem przyspieszyły. Zaszły również zmiany w sposobie pisania. Minimalny standard którego obecnie sie używa to C++17. Najnowszy standard to C++23, na widoku jest C++26. W ,,internetach'' można napotkać wiele materiałów, które prezentuja archaiczne podejście albo własności. Często prezentują one kalki z języka C, pomijają bibliotekę standardową albo propagują niezalecane praktyki. Prosze zwracać uwagę na aktualność wykorzystywanych źródeł.

Łańcuchy znakowe

W nagłówku `<string>` zdefiniowana jest klasa `std::string` służąca do przechowywania łańcuchów znakowych. Klasa ta zdecydowanie ułatwia obsługę łańcuchów znakowych w języku `C++` i powinna być używana zamiast łańcuchów znakowych w stylu `C`.

Klasa posiada dużo metod operujących na łańcuchach znakowych w tym:

- operator + pozwalający łączyć łańcuchy znakowe
- operator `[]` pozwalajacy na modyfikację poszczególnych znaków w takim sam sposób jak w zwykłej tablicy
- operator = pozwalający przypisać jeden łańcuch znakowy do drugiego
- operatory << i >> pozwalające zapisywać łańcuch znakowy do strumienia (`std::cout`, `std::fstream`) i odczytywać
- funkcję `size` zwracającą długośc łańcucha znakowego
- funkcje `clear`, `insert` oraz `erase`
- funkcję `substr` zwracającą podłańcuch znakowy
- funkcje `find`, `find_first_of`, `find_last_of` pozwalające wyszukiwać w łańcuchu znakowym
- i inne

Dodatkowo dostępne są funkcje:

- `std::stoi`, `std::stol`, `std::stof` etc. konwertujace łańcuch znakowy do typów liczbowych
- `std::to_string` konwertujące typy liczbowe do łańcuchów znakowych

Przykłady użycia

Poniżej zostanie podane kilka przykładów użycia `std::string`, wiele innych można znaleźć w dokumentacji.

Inicjalizacja

```
#include <string>

// konstruktor domyślny
std::string napis_1;

std::string napis_2 = "Hello" ;

// string(str) inicjalizuje łańcuchem str
std::string napis_3 {"World"};

// konstruktor kopiujący string(const string& str) inicjalizuje kopią str
std::string napis_4 (napis_3);

// string(count,ch) inicjalizuje kopią count znaków ch
std::string napis_5 (10,'-');       // napis_5 = ----------
```

Wyświetlanie

```
// na ekran (podobnie zapis do dowolnego strumienia)
        std::cout << napis_1 << '\n'
                  << napis_2 << '\n'
                  << napis_3 << '\n'
                  << napis_4 << '\n'
                  << napis_5 << '\n';
```

Przypisywanie i łączenie

```
// łączenie łańcuchów
napis_1 = napis_2 + " " + napis_3;  // napis_1 = "Hello World"

// dołączanie do łańcucha
```

```
napis_1 += "!";                     // napis_1 = "Hello World!"

// zmiana zawartości łańcucha
napis_4 = "AiSD";

napis_1 += '\n' + napis_4;          // napis_1 = ???

// assign(count,ch)
  napis_5.assign(4, '+');           // napis_5 = "++++"
```

Konwersja

```
// konwersja liczby na łańcuch bez formatowania
int i {42042};

napis_5 += " " + std::to_string(i); // to samo co sprintf(buf,"%d",i);
                                    // napis_1 = "++++ 42"

// Uwaga: dla liczb zmiennoprzecinkowych wynik może być zaskakujący
// wynika to z wewnętrznych zawiłości biblioteki
// np.: 23.3 może dac w wyniki 23.29999
```

Niestety `string` nie zezwala na formatowanie w taki sposób jak np.: `printf`, co trochę utrudnia życie. W standarcie `C++20` przewidziano klasę `std::format` która zwraca łańcuch znakowy sformatowany (naśladujac styl Pythona). Na dzień dzisiejszy żaden komilator nie ma zaimplementowanej obsługi, ale to się powinno zmienić niebawem.

Remedium jest używanie `sstringstream` albo funkcji z rodziny `printf` plus konwersja z `char*` na `std::string` (patrz niżej).

Dostęp do podłańcuchów i wyszukiwanie

```
// dostęp do i-tego znaku za pomoca operatora [], liczenie od 0!

    napis_5[4] = 'X';
    napis_5[7] = napis_5[4]; // napis_1 = "++++X42X42"


/* substr(begin,count), zwraca obiekt std::string zawierający podłańcuch
 * odpowiadający [begin,begin+count)
 *
 * w przypadku braku drugiego argumentu zwraca podłańcuch [begin,size())
 */
std::string napis_5a = napis_5.substr(4,4);    //napis_5a = X42X
std::string napis_5b = napis_5.substr(7);    //napis_5b = 42X

// typ danych w którym przechowywana jest pozycja pierwszego znaku znalezionego łańcucha
std::string::size_type n;

// find(string) zwraca pozycje pierwszego znaku wyszukiwanego łańcucha
    n = napis_5.find("42");             // n = 5

// wartość n można użyć np. tak
    std::cout << napis_5.substr(n);

// jeżeli łańcuch nie został znaleziony zwraca specjalny znacznik std::string:npos
    n = napis_5.find("Y");              // n = std::string::npos
```

std::string_view

Standard `C++17` wprowadza nowy typ łańcucha znakowego [`std::string_view`](). W uproszczeniu, jest to stały (read-only) łańcuch znakowy. Udostępnia on te funkcje z `std::string`, które nie modyfikują zawartości. Jest to obecnie preferowany sposób przechowywania stałych łańcuchów znakowych, gdyż w wielu sytuacjach jest wydajniejszy. Szczegóły są nieco techniczne.

std::stringstream

- W nagłówku <sstream> zdefiniowana jekst klasa `std::stringstream` obsługują sformatowane łańcuchy znakowe. Jej opis jest raczej skomplikowany ale, używa jej się tak samo jak standardowego strumienia wyjścia `std::cout` lub `std::fstream` z tą różnicą, że wszystko dzieje się w pamięci. Przykłady użycia:

```
#include <iostream>
#include <sstream>
#include <string>
...
std::stringstream bufor {}; //pusty strumień znakowy
std::string hello {"Hello"};

int i = 1;
float f = 2.3;

bufor << "Zaczynamy\n"  // zapis prostego tekstu
      << hello          // zapis std::string
      << 2              // zapis liczb całkowitych
      << i
      << 3.1            //zapis liczb zmiennoprzecinkowych
      << f
      << '\n';          //koniec wiersza

// zapisanie zawartości std::stringstream do std::string
std::string tekst = bufor.str();

// wyświetlanie zawartości stringstream na konsoli
// (podobnie dla zapisu do pliku przez std::fstream)
std::cout << "Z std::ostringstream:\t" << bufor.str()
          << '\n'
          << "Z std::string:\t" << tekst;

//czyszczenie
    bufor.str("");  // bufor jest pusty
    bufor << "nowy tekst";

std::cout << "Z std::ostringstream po wyczyszczeniu:\t" << bufor.str();
```

Formatowanie wyjścia

- wyjście sformatowane do strumieni takich jak `std::cout`, `std::stringstream` lub `std::fstream` można uzyskać korzystając z manipulatorów zdefiniowanych w nagłówku `<iomanip>`, np.:
  - `std::setw(n)` ustawia szerokość pola dla **następnego** elementu
  - `std::left`, `std::right` -- wyrównanie w ramach szerokości pola

```
#include <iomanip>
std::string_view AE {"ABCDE"};

std::cout << "*123456789 123456789 123456789 123456789 *\n"
          << '*'                   << AE                   << AE << "*\n"
          << '*' << std::setw(20) << AE                   << AE << "*\n"
          << '*' << std::setw(20) << AE << std::setw(20) << AE << "*\n"
          << '*'                   << AE << std::setw(20) << AE << "*\n"
          << std::left
          << '*'                   << AE                   << AE << "*\n"
          << '*' << std::setw(20) <<                         AE << "*\n"
          << '*' << std::setw(20) << AE << std::setw(20) << AE << "*\n"
          << '*'                   << AE << std::setw(20) << AE << "*\n";
```

W wyniku wykonania powyższego kodu otrzymamy:

```
*123456789 123456789 123456789 123456789 *
*ABCDEABCDE*
*               ABCDEABCDE*
*               ABCDE               ABCDE*
*ABCDE               ABCDE*
*ABCDEABCDE*
*ABCDE               *
*ABCDE               ABCDE               *
*ABCDEABCDE               *
```

- Do określenia formatu w jakim wyświetlane są liczby można wykorzystać manipulatory:
  - `std::showpoint` / `std::noshowpoint` -- określa czy ma być wyświetlany przecinek
  - `std::showpos` / `std::noshowpos` -- określa czy ma być wyświetlany znak + dla liczb nieujemnych
  - `std::fixed` / `std::scientific` / `std::defaultfloat` -- sprawdzić samemu
  - `std::dec` / `std::hex` -- przełącza na format dziesięty/szestnastkowy
  - `std::setprecision(n)` -- ustala ilość wyświetlanych cyfr liczby zmiennoprzecinkowej

```
    std::cout << "showpoint: " << std::showpoint << 1.0 << '\n'
              << "noshowpoint: " << std::noshowpoint << 1.0 << '\n';

    std::cout
        << "showpos: " << std::showpos << 42 << ' ' << 3.14 << ' ' << 0 << '\n'
        << "noshowpos: " << std::noshowpos << 42 << ' ' << 3.14 << ' ' << 0 << '\n';

const long double d = 1.2345678901234567890L;
    std::cout << "Domyślna precyzja: " << std::cout.precision() << "\n\n"
              << "precision = 6:         " << d << '\n'
              << std::setprecision(12)
              << "precision = 12         " << d << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
              << d << '\n';
```

Wykonanie powyższego kodu da w wyniku:

```
showpoint: 1.00000
noshowpoint: 1
showpos: +42 +3.14 +0
noshowpos: 42 3.14 0
```

```
default precision (6): 3.14159
std::setprecision(10): 3.141592654
max precision:         3.141592653589793239
```

Inny przykład

```
f = 1234.9876;
  std::cout << std::setprecision(6)
            <<  f <<" in fixed:      " << std::fixed << f << std::defaultfloat<< '\n'
            <<  f << " in scientific: " << std::scientific << f<< std::defaultfloat << '\n'
            <<  f << " in default:    " << std::defaultfloat << f<< std::defaultfloat <<'\n';
```

da w wyniku

```
1234.99 in fixed:      1234.987549
1234.99 in scientific: 1.234988e+03
1234.99 in default:    1234.99
```

UWAGA: Od C++17 w nagłówku <format> dostępny jest nowoczesny formatter do łańcuchów znakowych, kompatybilny ze strumieniami, o składni podobnej do printf i Pythona. Dobrze by było się z nim zapoznać.

Alternatywą dla powyższego sposobu formatowania jest używanie tablic znakowych w stylu C , nagłówków <cstring>, <cstdlib> oraz funkcji sprintf. Do sformatowania wyjścia można też skorzystać z funkcji sprintf lub printf.

```
const char* fmt_string = "Liczby %d %f %10.4f %+5.2f";
float f = 123.456;

// jezeli nie wiemy jak duży będzie wynik
// wywołanie ,,na sucho'' zwraca rozmiar
int buf_size = snprintf(0,0,fmt_string,12,f,f,f);

//rezerwacja + miejsce na \0


std::vector<char> buf (buf_size+1);  // równie dobrze char buf[buf_size+1];

    snprintf(&buf[0],buf_size,fmt_string,12,f,f,f);
/*
 *  jeżeli znany jest rozmiar bufora to mozna go odrazu zarewować bez
 *  wywolania na sucho i skorzystac z sprintf
 *
 * sprintf(&buf[0],fmt_string,12,d,d,d);
 */


// tak można skopiować do std::string
std::string out {buf.data() };

std::cout << "std::string out: " << out << '\n'
          << "std::vector buf: " << buf.data() << '\n';


// a tak można użyć std::string bezpośrednio
std::string tmp (buf_size,0);

    snprintf(tmp.data(),buf_size,fmt_string,12,f,f,f);

    std::cout << "std::string tmp: " <<  tmp << '\n';
```

Odczyt i zapis do pliku

- W nagłówku `<fstream>` zdefiniowane są strumienie we/wy realizowane przez klasy
- `std::ifstream` strumień wejściowy (odpowiada `std::cin`)
- `std::ofstream` strumień wyjściowy (odpowiada `std::cout`)
- `std::fstream` strumień we/wy

Sposób obsługi jest identyczny jak posługiwanie sie `std::cin` oraz `std::cout`. Z tą różnicą, że konieczne jest otwarcie i zamknięcie pliku (a potem odnalezienie go w katalogu roboczym)

Do podstawowej obsługi konieczne są funckje:

- `open(filename)` otwiera plik `filename` do odczytu/zapisu
- funkcja `is_open()` zwraca `true` jeżeli strumień jest skojarzony z plikem
- `close()` zamyka plik
- operatory >> i <<
- funckcja `getline`
- można uzywać manipulatorów opisanych przy podrozdziale o formatowaniu

Zapis

```
#include <fstream>
...

/*
 * utworzenie strumień wyjściowy
 * i otwarcie pliku
 *
 * jeżeli plik nie istnieje, to zostanie utworzony
 * UWAGA: domyślnie istniejący plik jest nadpisywany!
 *
 *
 */
std::ofstream out;
    out.open( out_filename );

// alternatywnym sposobem inicjalizacji jest: std::ofstream out( out_filename );


// jeżeli plik jest poprawnie otwarty to zapisujemy

    if( out.is_open() ){
        out << "Tekst zapisany do pliku oraz liczba " << 12;
    }
    else {
        // obsługa błędu
    }

    // zamykamy plik, niezamknięty plik zajmuje zasoby lub można o nim zapomnieć
    // niezamknięty plik może spowodować błedy przy kończeniu programu
    out.close();

/*
 *  TERAZ PROGRAM ROBI INNE RZECZY
 */

// Otwieramy plik ponownie
// tym razem w trybie dopisywania na koniec pliku: parametr std::ios::app
// bo nie chcemy zniszczyć wcześniejszyj pracy

    out.open(out_filename,std::ios::app);
```

```
    if(out.is_open()){
        out << std::setw(5) << "++\n";
    }
    else{
        //obsługa błędu
    }
    out.close();
```

Odczyt

Przykład:

```
// otwieramy plik do odczytu
std::ifstream in(out_filename);

std::string data;

    // sprawdzenie czy plik jest otwarty jest ważne
    if( in.is_open()){

        in >> data;         //odczytuje łańcuch znakowy az do napotkania
                            // spacji, tab, lub końca lini
                            // przesuwa pozycję w pliku na pierwszy nieodczytany znak

        // kolejne użycie operatora >> nadpisze zawartość data
        // in >> data;

        std::cout << "pierwszy odczyt --> " << data << '\n';

        std::getline(in,data);  //odczytuje plik od bierzącej pozycji do końca linii

        std::cout << "drugi odczyt --> " <<  data << '\n';

        std::getline(in,data);  //odczytuje plik od bierzącej pozycji do końca linii
                                // nie wczytuje znaku końca linii

        std::cout << "\ntrzeci odczyt --> " << data << '\n';

        data.clear();

        std::cout << "\nReszta pliku\n";

        // odczyt reszty pliku do łańcucha data
        // nie wczytuje znaków końca linii
        while( getline(in,data) ){

            std::cout << "\n Kolejna odczytana linia --> " << data << '\n';
        }


    }
    else {
        //obsługa błędu, w tym wypadku bardzo ważna
    }

    in.close();
```

Dane z pliku można odczytywać także do zmiennych innego typu. Sprawdzenie jak to zrobić jest ćwiczniem.

Uwaga: Standard C++17 definiuje bibliotekę `<filesystem>`, która pozwala na operacje na systemie plików taki jak sprawdzanie/ustawianie bierzącego katalogu lub kopiowanie plików.

Tablice w C++

W C++ można korzystać z trzech rodzajów tablic:

- wbudowany typ tablicowy (C-style)
- std::array -- tablica o stałym rozmiarze
- std::vector -- tablica dynamiczna

Tablice w stylu C

Dużą zaletą jest prostota, ale wadą brak kontroli indeksów, problemy z ustaleniem rozmiaru. Rozmiar tablicy musi być znany w czasie kompilacji albo trzeba posługiwać się przydzielaniem dynamicznym ze wszystkimi konsekwencjami.

Dużą zaletą jest gwarancja (jak w kazdym typie tablicowym), że elementy tablicy są rozmieszczone w pamięci po kolei, zatem można do nich odwoływac sie bezpośrednio przez podanie indeksu (albo za pomcą arytmetyki wskaźników).

```
const size_t N = 10
// inicjalizacja statyczna
// zainicjalizowany wartościami 2, 3, 4, 5 a potem 0
long A[N] {2, 3, 4, 5};

// inicjalizacja dynamiczna
int size_B = 10;
int B = new int[size_B];

    //wypełnieni tablicy liczbami
    for(int i =0 ; i< size; B[i]=-i,i++) {}

    //wypisanie zawartości tablicy
    for(int i =0 ; i< size; i++)
        std::cout << pA[i] << " ";

// B[i] znaczy efektywnie adres *( *B+ i*sizeof(int) )
// zatem kompilator nie widzi nic złego w poniższym
    pB[size] = 13; //bardzo częsty błąd

    for(int i =-1 ; i<= size; i++)
        std::cout << pB[i] << " ";

...
    // nie wolno zapominać o zwolnieniu pamięci przydzielonej dynamicznie
    delete[] B;
```

std::array

Kontener `std::array<T,N>` zdefiniowany w nagłówku `<array>` jest wrapperem na tablice w stylu C. Dostepne są dodatkowe funkcje: * `size()`zwracajaca rozmiar tablicy * `operator[]` działający identycznie jak ten dla tablic * `at()` realizuje dostęp za pomocą indeksów, ale w przypadku przekroczenia zakresu wyrzuca wyjątek

```
#include <array>
...
// zainicjalizowany wartościami 2, 3, 4, 5 a potem 0
std::array<int,10> waga = {2, 3, 4, 5}

/* dostęp po za zakres tablicy,
 *
 * waga[-1]; //undefined-behaviour
 * waga[10]; //undefined-behaviour
 *
 * funkcja at()
 * waga.at(-1); // zgłosi wyjątek
 * waga.at(10); // zgłosi wyjątek
 */

std::cout << "Rozmiar: " << waga.size();
```

```
std::cout << "trzeci element: " << waga[2] << '\n';
 waga[2] = -1;
std::cout << "trzeci element: " << waga[2] << '\n';
std::cout << "trzeci element: " << waga.at(2) << '\n'; //identyczne jak waga[2]

    //wypełnienie tablicy liczbami
    for(size_t i; i< waga.size(); waga[i]=-i,i++) {}

    //wypisanie zawartości tablicy
    for(size_t i = 0; i< waga.size()+1; i++)
        std::cout << waga[i] << " ";

/* skompiluje się ale zgłosi wyjątek out_of_range
    for(int i =-1; i< waga.size()+1; i++)
        std::cout << waga.at(i) << " ";
*/
```

std::vector

Kontener `std::vector<T>` jest zdefiniowany w nagłówku `<vector>`. Jest implemetacją tablicy dynamicznej.

```
#include <vector>
...
// zainicjalizowany wartościami 2, 3, 4, 5
std::vector<int> wzrost = {2, 3, 4, 5}

    for(int i =0 ; i< C.size(); C[i]=-i,i++) {}

    for(int i =0 ; i< C.size(); i++)
        std::cout << C[i] << " ";
```

Zapoznanie z metodami udostępnianymi przez kontenery `std::array` oraz `std::vector` i sposobem ich używania jest treścią jednego z zadań na Laboratorium 1.

Pomiar czasu działania

W nagłówku `<chrono>` zdefiniowano wiele funkcji i typów związanych z czasem. **Poniższe wyjaśnienia są troche techniczne, można przeskoczyć do przykładu a potem tu wrócić**

- `std::chrono::steady_clock` obiekt reprezentujący czasomierz
  - `std::chrono::steady_clock::now()` funkcja zwracająca typ `std::chrono::time_point`,
- `std::chrono::time_point` obiekt reprezentujący moment w czasie
  - różnica obiektów tego typu daje obiekt typu `std::chrono::duration`
- `std::chrono::duration<rep,period>` reprezentuje przedział czasowy
  - `rep` oznacza typ arytmetyczny np.: double, w którym wyrażona jest liczba ticków
  - `period` czas w sekundach pomiędzy tickami
  - funkcja `std::chrono::duration<rep,period>::count()` zwraca liczbę ticków
  - obiekty tego typu można ze sobą dodawać i odejmować
- predefiniowane period: `std::nano`, `std::milli`, `std::micro`, ogólnie `std::ratio<num,den>` odpowiada ułamkowi `num/den`
  - aby przekonwertować na minuty można użyć `std::chrono::duration<double,std::ratio<60> >`
  - aby przekonwertować na godziny można użyć `std::chrono::duration<double,std::ratio<3600> >`
- predefiniowane typy całkowitoliczbowe:
  - `std::chrono::nanoseconds duration<long, std::nano>`
  - `std::chrono::microseconds duration<long, std::micro>`
  - `std::chrono::milliseconds duration<long, std::milli>`
  - `std::chrono::seconds duration<long>`
  - `minutes` oraz `hours`
  - od C++ dostepne są takze `days`, `weeks`, `months` oraz `years`
- konwersja do tych typów musi być jawna za pomocą `std::chrono::duration_cast< T >`, nie sa wykonywane zaokrąglenia (59s = 0m, 1,9ms zstanie zamienione na 1ms)

Ponieważ nazwy tych typów sa długie i kłopotliwe do wpisywania poręcznie jest korzystac ze słowa kluczowego `auto`. Najprostsze zastosowanie, które na nasze potrzeby będzie wystarczające:

```
auto start = std::chrono::steady_clock::now();

    //nasz program
    ...

auto end = std::chrono::steady_clock::now();

// obliczenie czasu działania w millisekundach
std::chrono::duration<double,std::milli> delta_ms = end - start;

// obliczenie czasu działania w sekundach
std::chrono::duration<double> delta_s = end - start;

// obliczenie czasu działania w minutach
std::chrono::duration<double,std::ratio<60> > delta_m = end - start;

// rzutowanie na typy całkowitoliczbowe
auto delta_ms_int = std::chrono::duration_cast< std::chrono::milliseconds >(end-start);

std::cout << "Ten program wykonał zadanie w " << delta_ms.count() << " ms\n";
std::cout << "Ten program wykonał zadanie w " << delta_s.count() << " s\n";
std::cout << "Ten program wykonał zadanie w " << delta_m.count() << " minut\n";
std::cout << "Ten program wykonał zadanie w " << delta_ms_int.count() << "ms po zrzutowaniu\n";
```

Inny przykład:

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <numeric>
#include <chrono>

volatile int sink;
int main()
{
    std::cout << std::fixed << std::setprecision(9) << std::left;
    for (auto size = 1ull; size < 1000'000'000ull; size *= 100) {
        // record start time
        auto start = std::chrono::steady_clock::now();
        // do some work
        std::vector<int> v(size, 42);
        sink = std::accumulate(v.begin(), v.end(), 0u); // make sure it's a side effect
        // record end time
        auto end = std::chrono::steady_clock::now();
        std::chrono::duration<double> diff = end-start;
        std::cout << "Time to fill and iterate a vector of " << std::setw(9)
                  << size << " ints : " << diff.count() << " s\n";
    }
}
```

Proszę poeksperymentować...

Uwaga: Do pomiaru czasu można korzystać także z mniej rozbudowanej biblioteki [<ctime>] pochodzącej z C.

Generowanie wartości losowych

W C++ mamy dwa sposoby generowania liczb pseudolosowych:

1. opierajac się na wywodzącej się z jezyka C funkcji [int rand()] z biblioteki <cstdlib>
2. korzystajac ze standardowej biblioteki [<random>]

Korzystanie z `rand()`

```cpp
#include <cstdlib>
#include <iostream>
#include <ctime>

int main()
{
    std::srand(std::time(nullptr)); // use current time as seed for random generator
                                    // without this rand is called with srand(1)
    int random_variable = std::rand();
    std::cout << "Random value on [0 " << RAND_MAX << "]: "
              << random_variable << '\n';

    // roll a 6-sided die 20 times
    for (int n=0; n != 20; ++n) {
        int x = 7;
        while(x > 6)
            x = 1 + std::rand()/((RAND_MAX + 1u)/6);  // Note: 1+rand()%6 is biased
        std::cout << x << ' ';
    }
}
```

Korzystanie z `<random>`

Biblioteka oferuje rozmaite generatory liczb pseudolosowych w tym generujące liczby o rozkładzie jednostajnym, normalnym, Bernouliego, `std::discrete_distribution` itd.

- Najprostszy, bardzo podobny do `rand()`, wersja 1

```cpp
#include <random>
#include <iostream>

int main(){

    std::random_device rd1;     // inicjalizacja obiektu reprezentujacego
                                // generator liczb pseudolosowych

    std::cout << "rd.min(): " << rd1.min()
              << "\nrd.max(): " << rd1.max()
              << '\n';

    for(int i = 0; i<20 ; i++){
        std::cout << rd1() << " ";  // kolejna liczba pseudolosowa otrzyywana
                                    // jest przez wywołanie rd1()
                                    // zwracany typ: unsigned int
    }

    std::cout << '\n';
}
```

- Najprostszy, bardzo podobny do `rand()`, wersja 1

```cpp
#include <random>
#include <iostream>

int main()
{
    std::random_device rd;  //jako seed dla generatora

    // generator zainicjalizowany losową wartością
    // przy każdym uruchomieniu programu zwróci inne wartości
    std::default_random_engine generator1(rd());

    // generator zainicjalizowany ustaloną wartością
    // przy kazdym uruchomieniu programu zwróci takie same wartości
    std::default_random_engine generator2(2);

/* domyślnie zwracanym typem jest unsigned int
 *
 * niektóre inne dostępne generatory
 * std::minstd_rand -- linear congruential generator
 * std::mt19937 -- Mersene twister, wysoka jakość
 */

    for (int n=0; n<30; ++n)
        std::cout << generator1() << ' ';
    std::cout << '\n';

    for (int n=0; n<30; ++n)
        std::cout << generator2() << ' ';
    std::cout << '\n';

}
```

- liczby o rozkładzie jednostajnym

```cpp
#include <random>
#include <iostream>

int main()
{
    std::random_device rd;  //jako seed dla generatora

    // generator zainicjalizowany losową wartością
    // przy każdym uruchomieniu programu zwróci inne wartości
    std::default_random_engine generator1(rd());

    // generator zainicjalizowany ustaloną wartością
```

```
    // przy kazdym uruchomieniu programu zwróci takie same wartości
    std::default_random_engine generator2(2);

    std::cout << "\n--------------------------------\n";
    std::cout << "Z rozkładem jednostajnym na [a,b]:\n";

    // przekształca wartość zwracana przez generator
    // na liczbę zgodną z rozkładem  P(x|a,b) = 1/(b-a+1)
    // domyślnie zwracanym typem jest int
    std::uniform_int_distribution<> distribution1(-10, 10);

    std::cout << "Parametry rozkładu:"
              << "\ndistribution1.a: " << distribution1.a()
              << "\ndistribution1.b: " << distribution1.b()
              << '\n';

    std::cout << "\nZ użyciem generator1 - losowe wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution1(generator1) << ' ';
    std::cout << '\n';

    std::cout << "\nZ użyciem generator2 - zawsze te same wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution1(generator2) << ' ';
    std::cout << '\n';

}
```

- liczby o rozkładzie normalnym

```
#include <random>
#include <iostream>

int main()
{
    std::random_device rd;  //jako seed dla generatora

    // generator zainicjalizowany losową wartością
    // przy każdym uruchomieniu programu zwróci inne wartości
    std::default_random_engine generator1(rd());

    // generator zainicjalizowany ustaloną wartością
    // przy kazdym uruchomieniu programu zwróci takie same wartości
    std::default_random_engine generator2(2);

    std::cout << "\n--------------------------------\n";
    std::cout << "Z rozkładem normalnym o średniej mean i odchyleniu stddev:\n";

    // przekształca wartość zwracana przez generator na liczbę zgodną z rozkładem
    // domyślnie zwracanym typem jest double
    std::normal_distribution<> distribution2 (4, 20);

    std::cout << "Parametry rozkładu:"
              << "\ndistribution2.mean: " << distribution2.mean()
              << "\ndistribution2.stddev: " << distribution2.stddev()
              << '\n';

    std::cout << "\nZ użyciem generator1 - losowe wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution2(generator1) << ' ';
    std::cout << '\n';

    std::cout << "\nZ użyciem generator2 - zawsze te same wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution2(generator2) << ' ';
    std::cout << '\n';


}
```

- liczby o rozkładzie dyskretnym

```cpp
#include <random>
#include <iostream>

int main()
{
    std::random_device rd;  //jako seed dla generatora

    // generator zainicjalizowany losową wartością
    // przy każdym uruchomieniu programu zwróci inne wartości
    std::default_random_engine generator1(rd());

    // generator zainicjalizowany ustaloną wartością
    // przy kazdym uruchomieniu programu zwróci takie same wartości
    std::default_random_engine generator2(2);

    std::cout << "\n--------------------------------\n";
    std::cout << "Z rozkładem dyskretnym na [0,n):\n";

    std::discrete_distribution<> distribution3 ({5,1,4,4,1,5});
    // jako argument podaje się wagi w_i, wtedy P(i) = w_i/Suma wag

    std::vector<double> p = distribution3.probabilities();
    std::cout << "Parametry rozkładu: ";
    for(auto n : p)
        std::cout << n << ' ';
    std::cout << '\n';

    std::cout << "\nZ użyciem generator1 - losowe wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution3(generator1) << ' ';
    std::cout << '\n';

    std::cout << "\nZ użyciem generator2 - zawsze te same wartości\n";
    for (int n=0; n<30; ++n)
        std::cout << distribution3(generator2) << ' ';
    std::cout << '\n';
}
```