Klasa std::string

Utworzenie obiektu typu **std::string** odbywa się podobnie jak dowolnej zmiennej typu wbudowanego. Jednak w tym przypadku można też stworzyć obiekt zainicjalizowany danymi – obiekt budowany jest przez specjalną metodę składową, **konstruktor**. Konstruktorów może być dowolnie wiele, muszą różnić się argumentami.

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

using namespace std;

auto main() -> int {
  string s1;  // pusty string

}
```

Zbadajmy jaki jest rozmiar
i bufor obiektu s1:

```
for (auto i(0); i<1025; ++i) {
  s1 += "a";
  cout << s1.size() << " – "
        << s1.capacity() << endl;
}
```

Dodatkowo co będzie gdy:

```
s1.clear();
s1.empty(); // zwraca true lub false
s1.shrink_to_fit();
s1.reserve(57);  // jakie capacity() ?
```

Tworzymy kolejne obiekty std::string

Oto kilka sposobów na utworzenie / przypisanie obiektu typu std::string

```
const char *t = "tekst do inicjalizacji";
s1 = t;
string s2( s1); // obiekt „na wzór” istniejącego wcześniej
string s3( t, 8 ); // pierwsze 8 znaków
string s4( s2, 6, 8 ); // od 6-tego do 6+8 -mego, czyli…
string s5( 100, ’*’ ); // chcę mieć sto gwiazdek
string s6 = "konstrukcja";
string s7 = { "uniwersalna inicjalizacja" }; // = opcjonalnie
```

Działania na stringach bez problemu:

```
s1 =  s1 + " drugi " + s2;
s1 += s6;
```

Rozmiary, usuwanie…

Maksymalny rozmiar i pewna stała:

```
max_size()  // zwykła metoda składowa
string().max_size(); // string „w locie”
```

Sprawdźcie jaka jest wartość tej stałej:

```
std::string::npos
```

Wielkie usuwanie (erase – metoda składowa):

```
erase( nr_od, nr_ile ); // zwraca „referencję do”
erase( adres_od, adres_do ); // zwraca „adres” nast. znaku
```

Specjalne funkcje adresowe (zwracające tzw. iteratory czyli obiekty „udające” wskaźniki – przechowalniki adresu i wiedzy o typie):

```
begin(); // adres początku „zerowej pozycji”
end(); // adres za ostatnim elementem, „za-ostatni”
```

Usuwanie…

Przykład, dodatkowo z algorytmem **find**:

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main () {
    string s = "To jest dobry przyklad";
    cout << s << '\n';

    s.erase(0, 3); // usuń "To "
    cout << s << '\n';

    s.erase( std::find(s.begin(), s.end(), ' ') ); // usuń pierwszą spację ' '
    cout << s << '\n';

    s.erase( s.find(' ') ); // Znajdź kolejną i usuń wszystko od niej do końca
    for ( auto n : s ) cout << n << " - "; // literka po literce
}
```

Małe ćwiczenie

Narysujmy za pomocą „erase” taką sekwencję…

****

****

****

****

I tak dalej…

```
#include <iostream>
#include <string>
using namespace std;

int main () {
    string str (20, '*'); // tworzymy łańcuch znakowy z 20-tu gwiazdek
    while ( ! str.empty() )    {
                cout << str << endl;
                str.erase( str.end()-1 ); // samo end() to pozycja „za-ostatnia”
    }
}
```

Wyścig wątków

```
#include <iostream>
#include <thread>
#include <string>
using namespace std;

void addstring( unsigned n, string& s ) {
  while (n--) {
    s += "*"; cout << "A";
  }
}
void removestring( string& s ) {
  while ( !s.empty() ) {
     s.erase( s.end()-1 ); cout << "B";
  }
}
int main() {
  string m;
  thread t1( addstring, 100, ref(m) );
  thread t2( removestring, ref(m) );
  t1.join();
  t2.join();
  cout << endl << m << endl;
}
```

Argument funkcji: string& s
oznacza „przez referencję”, czyli przez „przezwisko”, z intencją pracy na oryginale.

Argumenty funkcji wywoływanej w wątku (obiekt thread) przekazuje się po prostu jako kolejne wielkości, po przecinku, gdzie pierwszym argumentem obiektu thread jest nazwa funkcji, która ma być wykonywana.

join() informuje główny wątek (program), że ma poczekać z zakończeniem aż do skończenia działania danego wątku podrzędnego

Opakowanie **std::ref** jest konieczne tutaj, jeśli chcemy do wątku przekazać obiekt-oryginał przez referencję. Ewentualnie można też użyć wskaźnik.

Losowanie liczb

```
#include <random>
#include <iostream>
#include <string>
#include <ctime>
using namespace std;
int main() {
  /* Inicjalizacja. Tylko raz, na początku. */
  random_device rd;
  mt19937_64 gen(rd()); // seed z rd
// można też tak:
// mt19937::result_type seed = time(0);
// mt19937_64 gen( seed );
  /* Generator płaski w oparciu o typ short */
  uniform_int_distribution<short> dis;

  /* Kilka liczb, konwertujemy na string */
 for (auto n=0; n<10; ++n)
  cout << dis(gen) << ' ' << to_string( dis(gen) ) << ' ';
   endl ( cout );
}
```

std::random_device to generator liczb całkowitych o jednorodnym rozkładzie, produkujący liczby w sposób niedeterministyczny (zależnie od dostępu do sprzętowego niedeterministycznego źródła)

Mersenne Twister to algorytm generatora liczb pseudolosowych. Silnik tego generatora inicjalizowany jest często poprzez generator std::random_device, domyślna wersja 64-bitowa to **std::mt19937_64**

Rozkłady losowe, takie jak **std::uniform_int_distribution**, używają silników (j.w.), generując liczby. Są też rozkłady Bernoulliego, normalny, Poissona.

rand() z języka C jest oznaczony w C++14 jako *deprecated* (przestarzały), w C++17 zniknie

Wczytywanie z pliku

Utwórzmy obiekt do obsługi strumienia plikowego i wczytajmy… a potem wypiszmy!

```
#include <fstream>
   string s10;
   string str;
   cout << "Wprowadz tekst: ";
   cin >> str;
   cout << "Wczytano to: " << str << endl;
   getline (cin, str, '@'); // koniec = znaczek @
   cout << "Wczytano tamto: " << str << endl;
```

Bufor cin nadal trzyma starą zawartość, tu poczytajcie jak to wyczyścić
http://cpp0x.pl/kursy/Kurs-C++/Poziom-1/Obsluga-strumienia-wejsciowego/12

```
   ifstream plik("tekst.txt"); // np. wziąć z: pl.lipsum.com
   while ( ! plik.eof() )    {
      getline (plik, str);
      s10 += str; // czego tu brakuje? Znak końca linii… + '\n'
   }
   // wypiszcie na ekran…  cout << s10;
```

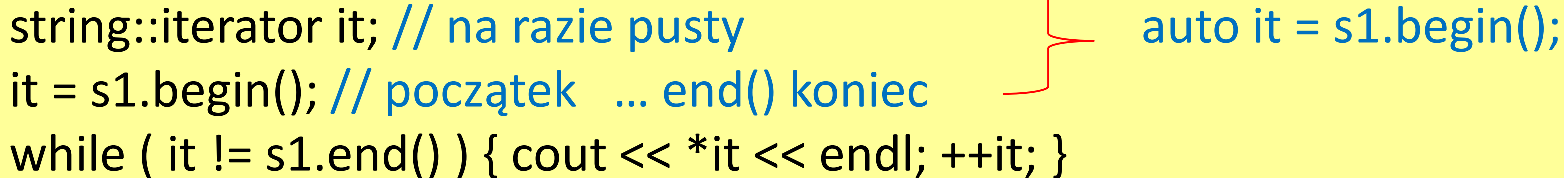
Przebiegamy po stringu…

String to forma kontenera sekwencyjnego… jakby tablicy znaków…

```
string s1 = "wlazl kotek na plotek i mruga";
for ( auto c : s1 ) cout << c << " ";  // range-based loop
for ( auto& c : s1 ) c = ( c==’w’ ) ? ’W’ : c; // zamieniamy na wielkie W, co z nawiasami?

for ( int i=0; i < s1.length(); ++i ) cout << s1[i] << " ";
```

ITERATOR – inteligentny „pośrednik” pomiędzy kontenerami (zasobnikami), „wskaźnik” z adresem do operacji na konkretnych typach, strumieniach…



ITERATOR STRUMIENIA

```
copy (s1.begin(), s1.end(), ostream_iterator<char>(cout,"\n"));
// używamy algorytmu copy (ten z nagłówka <algorithm>)
// tworzymy w locie iterator strumienia wyjściowego, ostream_iterator
// konieczny nagłówek #include <iterator>
```