C++17 i STL

Iteratory strumieniowe i plikowe

Iteratory strumieni

- Iterator strumieniowy (ang. stream iterator) to adaptator iteratora, który umożliwia wykorzystanie strumienia jako źródła danych dla algorytmów albo jako przeznaczania danych generowanych przez algorytmy.
- Iterator strumieniowy wejściowy służy do odczytywania wartości ze strumienia wejściowego.
- Iterator strumieniowy wyjściowy służy do wpisywania wyrażeń do strumienia wyjściowego.

Iteratory strumieniowe wyjściowe

- Iteratory strumieniowe wyjściowe zapisują przypisywane wartości do strumienia wyjściowego.
- Wykorzystując iteratory strumieniowe wyjściowe, algorytmy mogą zapisywać dane bezpośrednio do strumieni.
- Implementacja iteratora strumieniowego wyjściowego wykorzystuje tę samą koncepcję co implementacja iteratorów wstawiających – jedyna różnica polega na tym, że przekształcają one operację przypisania nowej wartości w operację wyjścia z wykorzystaniem operatora <<.

Iteratory strumieniowe wyjściowe

- Podczas tworzenia iteratora strumienia wyjściowego musimy podać strumień wyjściowy, do którego będą zapisywane wartości.
- Możemy również podać w konstruktorze dodatkowy opcjonalny łańcuch, który zapisywany będzie jako separator pojedynczych wartości.
- Przykład:

```
#include <iterator>
//…
main() {
    ostream_iterator<double>
        dblWriter(cout);
    ostream_iterator<int>
        intWriter(cout, "\n");
    // …
}
```

Iteratory strumieniowe wyjściowe

- W stosunku do iteratora strumieniowego wyjściowego `iter` można ale nie trzeba stosować operatora wyłuskania `*iter` w przypisaniu.
- Przykład:

```
#include <iterator>
// …
main() {
    ostream_iterator<int>
        intWriter(cout,"\n");
    intWriter = 123;
    *intWriter = 127;
    //…
}
```

Iteratory strumieniowe wyjściowe

- W stosunku do iteratora strumieniowego wyjściowego `iter` operator przesunięcia na następną pozycję `++iter` albo `iter++` nie daje żadnego efektu (dlatego nie używa się go w stosunku do iteratorów strumieniowych).
- Przykład:

```
#include <iterator>
#include <vector>
//…
main() {
    vector<int> coll = {2, 3, 5, 7};
    copy(coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout, ", "));
    cout << endl;
    //…
}
```

Iteratory strumieniowe wejściowe

- Iterator strumieniowy wejściowy odczytuje elementy ze strumienia wejściowego.
- Wykorzystując iteratory strumieniowe wejściowe, algorytmy mogą odczytywać dane bezpośrednio ze strumieni.
- Podczas tworzenia iteratora strumieniowego wejściowego inicjalizowany jest on strumieniem wejściowym, z którego realizowany będzie odczyt – odczyt danych element po elemencie jest realizowany za pomocą operatora >>.
- Przykład:

```
istream_iterator<int> intReader(cin);
```

Iteratory strumieniowe wejściowe

- Odczyt danych ze strumienia może zakończyć się niepowodzeniem – sygnalizowane jest to za pomocą iteratora końca strumienia (ang. end-of-stream iterator); iterator końca strumienia tworzony jest za pomocą konstruktora domyślnego iteratorów strumieni wejściowych.
- W przypadku niepowodzenia operacji odczytu każdy iterator strumieniowy wejściowy staje się iteratorem końca strumienia. Tak więc, po każdej próbie odczytu, należy porównać iterator strumieniowy wejściowy z iteratorem końca strumienia w celu sprawdzenia, czy iterator posiada poprawną wartość.
- Przykład:

```
istream_iterator<int> intReaderEOF;
```

Iteratory strumieniowe wejściowe

- W stosunku do iteratora strumieniowego wejściowego `iter` należy stosować operator wyłuskania `*iter` do odczytania wartości – wyłuskanie zwraca aktualną ostatnio odczytaną wartość.
- Konstruktor iteratora strumieniowego wejściowego otwiera strumień oraz odczytuje pierwszy element, aby w przypadku wywołania `*iter` bezpośrednio po inicjalizacji możliwe było uzyskanie prawidłowego rezultatu.

Iteratory strumieniowe wejściowe

- W stosunku do iteratora strumieniowego wejściowego `iter` operator przesunięcia na następną pozycję `++iter` odczytuje kolejną wartość i zwraca bieżącą pozycję a `iter++` odczytuje kolejną wartość lecz zwraca poprzednią pozycję.
- Iteratory strumieniowe wejściowe można ze sobą porównywać za pomocą == lub `!=`.

Iteratory strumieniowe wejściowe

- Przykład:

```
#include <iterator>
//…
main() {
    istream_iterator<int> intReader(cin);
    istream_iterator<int> intReaderEOF;
    while (intReader != intReaderEOF) {
        //…
        ++intReader;
    }
    // …
}
```

Operacje na lokalnym systemie plików

- W C++17 wprowadzono do biblioteki standardowej nową funkcjonalność do obsługi lokalnego systemu plików.
- Jej najlepszą cechą jest to, że działa tak samo na różnych systemach operacyjnych.
- Funkcje, obiekty i iteratory do obsługi systemu plików zdefiniowane są w pliku nagłówkowym `<filesystem>` i umieszczone w przestrzeni nazw `std::filesystem` (został zaadaptowany bezpośrednio z BOOST filesystem).

Biblioteka Filesystem

- Biblioteka Filesystem udostępnia narzędzia do wykonywania operacji w systemie plików i jego składnikach takich jak ścieżki, pliki i katalogi.
- Funkcje biblioteki Filesystem mogą być niedostępne, jeśli hierarchiczny system plików nie jest dostępny w tej implementacji biblioteki lub jeśli nie zapewnia niezbędnych możliwości. Niektóre funkcje mogą być niedostępne, jeśli nie są obsługiwane przez podstawowy system plików.

Podstawowe pojęcia używane w bibliotece Filesystem

- Plik: obiekt w systemie plików, który przechowuje dane, może być zapisywany albo odczytywany albo jedno i drugie; pliki mają nazwy i atrybuty. Jednym z atrybutów jest typ pliku:
  - katalog: plik, który działa jako kontener wpisów w katalogu – wpisy te identyfikują inne pliki (niektóre z nich mogą być innymi, zagnieżdżonymi katalogami); rozważając konkretny plik katalog, w którym pojawia się on jako wpis, jest jego katalogiem nadrzędnym; katalog nadrzędny może być reprezentowany przez względną nazwę ścieżki z użyciem "..";
  - zwykły plik: wpis w katalogu, który kojarzy nazwę z istniejącym plikiem (twarde łącze); jeśli obsługiwanych jest wiele łączy twardych, plik jest usuwany po usunięciu ostatniego łącza do niego;
  - dowiązanie symboliczne: wpis w katalogu, który kojarzy nazwę ze ścieżką, która może istnieć lub nie;
  - inne specjalne typy plików: blokowe, znakowe, fifo, gniazda.
- Nazwa pliku: ciąg znaków identyfikujący plik (po nazwie); dopuszczalne znaki, rozróżnianie wielkości liter, maksymalna długość i niedozwolone nazwy są określone w implementacji systemu plików; nazwy specjalne "." (kropka) i ".." (dwie kropki) mają specjalne znaczenie na poziomie biblioteki Filesystem.

Podstawowe pojęcia używane w bibliotece Filesystem

- Ścieżka: sekwencja elementów, która identyfikuje plik; ścieżka rozpoczyna się opcjonalną nazwą urządzenia (np. "C:" lub "//server" w systemie Windows), po której następuje opcjonalny katalog główny (np. "/" w systemie Unix), po którym następuje sekwencja zero lub więcej nazw plików (wszystkie oprócz ostatniego muszą być katalogami lub linkami do katalogów); format natywny (znaki używane jako separatory) i kodowanie znaków reprezentujących ścieżkę (nazwę ścieżki) jest zdefiniowane w implementacji systemu plików – biblioteka Filesystem zapewnia przenośną reprezentację ścieżek.
- Ścieżka bezwzględna: ścieżka, która jednoznacznie identyfikuje lokalizację pliku.
- Ścieżka kanoniczna: ścieżka bezwzględna, która nie zawiera dowiązań symbolicznych oraz elementów "." lub "..".
- Ścieżka względna: ścieżka, która identyfikuje położenie pliku względem jakiejś lokalizacji w systemie plików; specjalne nazwy elementów ścieżki "." (kropka - "bieżący katalog") i ".." (dwie kropki - "katalog nadrzędny") są wykorzystywane w nazwach ścieżek względnych.

Klasa `path`

- Obiekty typu `path` reprezentują ścieżki w systemie plików. Obsługiwane są tylko składniowe aspekty ścieżek: nazwa ścieżki może reprezentować nieistniejącą ścieżkę lub nawet taką, która nie może istnieć w bieżącym systemie plików lub systemie operacyjnym.
- Separator elementów ścieżki jest określony za pomocą `path::preferred_separator`.
- Ścieżki można przetwarzać za pomocą iteratorów zwracanych przez funkcje `begin()` i `end()`; iteracja rozpoczyna się od nazwy głównej, katalogu głównego, poprzez kolejne elementach nazw katalogów w ścieżce (separatory katalogów są pomijane z wyjątkiem tgo, który identyfikuje katalog główny).

Funkcje narzędziowe

- Używając funkcji `filesystem::exists()`, można sprawdzić, czy podana ścieżka dostępu naprawdę istnieje w systemie plików.
- Obiekt `path` można utworzyć na podstawie dowolnego ciągu tekstowego, nawet w żaden sposób nie powiązanego z systemem plików – funkcja `exists()` testuje egzemplarz `path` i zwraca informację o istnieniu pliku czy katalogu w lokalnym systemie plików.
- Funkcja `exists()` potrafi rozróżniać bezwzględne i względne ścieżki dostępu do pliku czy katalogu.
- Funkcja `filesystem::canonical()` zwraca obiekt `path` w znormalizowanej postaci – funkcja `canonical()` próbuje pozbyć się ze ścieżki dostępu wszelkich odwołań w postaci jednej lub dwóch kropek.
- Jeśli do funkcji `canonical()` zostanie dostarczony drugi opcjonalny argument, którym jest bazowa ścieżka dostępu, to druga ścieżka dostępu zostanie dołączona na początku pierwszej ścieżki.

Przeglądanie zawartości katalogu

- Aby przeglądnąć zawartość katalogu, należy utworzyć egzemplarz obiektu `directory_iterator` a następnie przeiterować po nim:

```
for (const directory_entry &entry :
directory_iterator{dir}) {
    // Dowolna operacja
}
```

- Obiekt `directory_entry` reprezentuje element katalogu.
- Typ `file_status` określa status elementu katalogu, który można uzyskać za pomocą funkcji `status()`:

```
const file_status fs(status(entry));
```

- Można zbadać jaki jest status elementu za pomocą jednej z funkcji: `is_directory(fs)`, `is_regular_file(fs)`, `is_symlink(fs)`, itp.

Przeglądanie zawartości katalogu i podkatalogów

- Aby przeglądnąć zawartość katalogu i rekurencyjnie zawartych w nim podkatalogów, należy utworzyć egzemplarz obiektu `recursive_directory_iterator` a następnie przeiterować po nim.
- Przykład:

```
for (const auto &entry :
recursive_directory_iterator{current_path()}) {
    // …
}
```

Manipulacja zawartością systemu plików

- Funkcje `copy()` i `copy_file()` umożliwiają kopiowanie plików i całych katalogów.
- Funkcje `remove()` i `remove_all()` umożliwiają usuwanie plików i całych katalogów.
- Funkcje `create_directory()` i `create_directories()` umożliwiają tworzenie katalogów.
- Funkcja `rename()` pozwala zmienić nazwę albo lokalizację pliku.

Uzupełnienie

- Bardziej szczegółowe informacje można znaleźć w dokumentacji albo na stronie: https://en.cppreference.com/w/cpp/filesystem