Gliwice 16.12.2019

Programowanie Komputerów

Temat: Kolejka 4B

Autor: Adrian Michałek
Semestr: Trzeci
Grupa: III
Sekcja: -

Prowadzący: dr. inż. Jerzy Rospondek

Analiza Rozwiązania

Głównym zadaniem kontenera (klasy generycznej) **Queue** jest przechowywanie danych dowolnego typu oraz dostęp do nich za pomocą odpowiednich metod. Klasa **Queue** zaprojektowania zostanie w myśl zasady FIFO (First In, First Out) *pierwszy na wejściu pierwszy na wyjściu.* Kolejka zaimplementowana zostanie za pomocą listy dwukierunkowej składającej się z węzłów wskazujących bezpośrednio na poprzedni i następny węzeł. Do implementacji kontenera użyte zostaną trzy klasy pomocnicze. Pierwszą klasą generyczną będzie **Node** opisujący węzeł listy dwukierunkowej. Drugą klasą generyczną będzie **Guard**. Klasa ta pomoże w szybki i łatwy sposób znaleźć ewentualne wycieki pamięci w trakcie projektowania pozostałych klas, jej zadaniem będzie między innymi porównanie ilości zaalokowanych bajtów przed użyciem obiektów z ilością zaalokowanych bajtów po zniszczeniu obiektów. Trzecią klasą generyczną będzie lista dwukierunkowa **DoublyLinkedList** składająca się z elementów typu **Node** umożliwiająca operacje na węzłach. Klasa **Queue** składać się będzie z obiektu klasy **DoublyLinkedList** dodając jedynie warstwę abstrakcji, umożliwiając tym samym operacje dodawania/usuwania elementu z kolejki, wypisywania jej zawartości, możliwość dostania się do pierwszego i ostatniego elementu jak i informacje o ilości elementów w kolejce. Węzły będą tworzone jak i niszczone na stercie ze względu na większe możliwości jakie oferuje sterta. Klasa **Node** przetrzymywać będzie informacje o ilości zaalokowanych bajtów za pomocą statycznej zmiennej. Klasa **DoublyLinkedList** posiadać będzie limit ilości węzłów, który w łatwy sposób może zostać zwiększony przez programistę. Klasa **Queue** oprócz dodatkowej abstrakcji zawierać będzie stan typu wyliczeniowego oraz operacje typu wyliczeniowego. Dla kolejki zaimplementowana zostanie metoda zwracająca aktualny stan tj. czy kolejka jest pusta, pełna lub wypełniona oraz metoda zwracająca aktualny błąd operacji na niej np. próba usunięcia elementu w przypadku gdy kolejka jest pusta lub próba dodania elementu w przypadku gdy kolejka jest pełna.

Specyfikacja zewnętrzna

Kolejkę należy wykorzystać wszędzie tam gdzie zależy nam na odpowiedniej kolejności elementów. Kolejkę używamy definiując obiekt typu **Queue<T>** gdzie **T** oznacza typ danych jaki chcielibyśmy przechować. Przykładowe zadeklarowanie kolejki na stosie:

```
Queue<int> queue;
```

Kolejka posiada również dwa dodatkowe konstruktory kopiujące (poprzez referencję i przesunięcie semantyczne):

```
Queue<int> queue(q);
Queue<int> queue(std::move(q));
```

W celu otrzymania informacji o ilości utworzonych obiektów klasy **Queue** należy posłużyć się metodą count():

```
unsigned int count = queue.count();
```

W celu otrzymania informacji o ilości elementów znajdujących się w kolejce należy użyć metody size():

```
uint64_t size = queue.size();
```

W celu dostania się danych typu **T** utworzonej kolejki należy użyć kolejno front() zwracający pierwszy element lub back() zwracający ostatni element:

```
int back = queue.back();
int front = queue.front();
```

W celu otrzymania aktualnego stanu kolejki należy użyć metody state():
QueueState state = queue.state();
Możliwe do otrzymania stany to kolejno: EMPTY, FULL, FILLED

W celu otrzymania aktualnego błędu operacji kolejki należy użyć metody error():
QueueOperation error = queue.error();
Możliwe do otrzymania stany to kolejno: NO_ERROR, REMOVE_EMPTY, INSERT_OVERFLOW, CLEAR_EMPTY

W celu dodania elementu do kolejki należy posłużyć się metodą insert() (za pomocą literału, zmiennej, lub poprzez przesunięcie):
bool status = queue.insert(1);
bool status = queue.insert(tmp);
bool status = queue.insert(std::move(tmp));
Możliwy do otrzymania błąd to tzw. INSERT_OVERFLOW:
"Error: Unsigned 64-bit integer overflow, extend m_max range..."
Należy wówczas zwiększyć zakres zmiennej m_max lub zmniejszyć ilość alokowanych elementów.

W celu usunięcia elementu z kolejki należy użyć metody remove() (spowoduje usunięcie pierwszego elementu z kolejki):
bool status = queue.remove();
Możliwy do otrzymania błąd to REMOVE_EMPTY objawiający się przy próbie usunięcie elementu w przypadku gdy kolejka jest pusta.

W celu usunięcia elementów z kolejki należy użyć metody clear() (spowoduje usunięcie wszystkich elementów z kolejki):
bool status = queue.clear();
Możliwy do otrzymania błąd to CLEAR_EMPTY objawiający się przy próbie wyczyszczenia kolejki gdy jest ona pusta.

Specyfikacja wewnętrzna

```
template<class T>
class Node
{
	static __int64 m_allocated; // Zmienna przechowująca ilość zaalokowanych bajtów wszystkich węzłów typu T
public:
	T m_data;		// Zmienna przechowująca dane typu T
	Node<T>* m_prev;	// Zmienna wskazująca na poprzedni węzeł
	Node<T>* m_next;	// Zmienna wskazująca na następny węzeł
	explicit Node(T, Node<T>* = nullptr, Node<T>* = nullptr); // Konstruktor przyjmujący daną typu T, wskaźnik na poprzedni i następny węzeł
	~Node(); // Destruktor przypisujący zmiennym m_prev i m_next nullptr
	Node<T> operator=(const Node<T>&); // Operator nadpisujący zmienną m_data, przyjmujący węzeł
	void* operator new(std::size_t); // Operator new zwiększający licznik bajtów m_allocated
	void operator delete(void*); // Operator delete zmniejszający licznik bajtów m_allocated
	static const __int64& allocated(); // Funkcja zwracająca m_allocated
};
```

Jako, że klasa **Guard** nie jest wymieniona w zadaniu nie zostanie ona opisana w dokumentacji.

```
template<class T>
class DoublyLinkedList
{
	// ...
	static const uint64_t m_max = 10000; // Zmienna ograniczająca ilość możliwych elementów
	static T m_error; // Zmienna zwracana w przypadku błędów operacji na liście
	// ...
public:
	explicit DoublyLinkedList(); // Konstruktor podstawowy
	DoublyLinkedList(const DoublyLinkedList<T>&); // Konstruktor kopiujący przyjmujący obiekt typu DoublyLinkedList poprzez referencję
	DoublyLinkedList(const DoublyLinkedList<T>&&); // Konstruktor kopiujący przyjmujący obiekt typu DoublyLinkedList poprzez przesunięcie
	~DoublyLinkedList(); // Destruktor
	DoublyLinkedList<T>& operator=(const DoublyLinkedList<T>&); // Operator przypisania przyjmujący obiekt DoubleLinkedList poprzez referencję
	DoublyLinkedList<T>& operator=(const DoublyLinkedList<T>&&); // Operator przypisania przyjmujący obiekt DoubleLinkedList poprzez przesunięcie
	DoublyLinkedList<T>& operator+=(const DoublyLinkedList<T>&); // Operator dodający elementy z drugiej listy za pomocą referencji na obiekt
	DoublyLinkedList<T>& operator+=(const DoublyLinkedList<T>&&); // Operator dodający elementy z drugiej listy za pomocą przesunięcia obiektu
	bool insert(const T&); // Metoda umożliwia dodanie elementu do listy za pomocą referencji, zwraca true gdy nie nastąpiło przepełnienie listy
	bool insert(const T&&); // Metoda umożliwia dodanie elementu do listy za pomocą przesunięcia, zwraca true gdy nie nastąpiło przepełnienie listy
	bool remove(uint64_t); // Metoda usuwająca element z listy za pomocą indeksu elementu, zwraca true gdy lista nie była pusta
	bool remove(const Node<T>*);  // Metoda usuwająca element z listy za pomocą adresu elementu, zwraca true gdy lista nie była pusta
	const T& front() const; // Metoda zwracająca pierwszy element listy
	const T& back() const; // Metoda zwracająca ostatni element listy
	bool pop_front(); // Metoda usuwająca pierwszy element z listy, zwraca true gdy lista nie była pusta
	const Node<T>* head() const; // Metoda zwracająca pierwszy węzeł listy
	const Node<T>* tail() const; // Metoda zwracająca ostatni węzeł listy
	bool clear(); // Metoda usuwająca wszystkie elementy z listy
	uint64_t size() const; // Metoda zwracająca aktualną ilość elementów w liście
	uint64_t max() const; // Metoda zwracająca maksymalną możliwą ilość elementów w liście
	template<class U> // Operator <<
	friend std::ostream& operator<<(std::ostream&, const DoublyLinkedList<U>&);
};
```

```
template<class T>
class Queue
{
	// ...
public:
	Queue(); // Konstruktor podstawowy
	Queue(const Queue<T>&); // Konstruktor kopiujący (poprzez referencję do drugiego obiektu)
	Queue(const Queue<T>&&);  // Konstruktor kopiujący (poprzez przesunięcie drugiego obiektu)
	~Queue(); // Destruktor niszczący wszystkie elementy typu T
	bool insert(const T&); // Funkcja dodaje nowy element typu T poprzez referencje, zwraca true gdy nie nastąpiło przepełnienie kolejki
	bool insert(const T&&); // Funkcja dodaje nowy element typu T poprzez przesunięcie, zwraca true gdy nie nastąpiło przepełnienie kolejki
	bool remove(); // Funkcja usuwa pierwszy element z kolejki, zwraca true gdy kolejka nie była pusta
	uint64_t size() const; // Funkcja zwraca aktualną ilość elementów w kolejce
	const T& front(); // Funkcja zwraca pierwszy element z kolejki
	const T& back(); // Funkcja zwraca ostatni element z kolejki
	QueueState state() const; // Funkcja zwraca aktualny stan kolejki
	const QueueOperation& error() const; // Funkcja zwraca aktualny błąd lub jego brak
	static const unsigned int& count(); // Funkcja zwraca aktualną liczbę utworzonych instancji klasy Queue<T>
	bool clear(); // Funkcja usuwa wszystkie elementy z kolejki, zwraca true w przypadku gdy kolejka nie była pusta

	// Zdefiniowane przeładowane operatory kolejki to:
	Queue<T>& operator=(const Queue&);
	Queue<T>& operator=(const Queue&&);
	Queue<T>& operator+=(const Queue&);
	Queue<T>& operator+=(const Queue&&);
	template<class U>
	friend std::ostream& operator<<(std::ostream& os, const Queue<U>&);
};
```

Testowanie

Skuteczność klasy **Queue** była testowana w następujący sposób:

Test I:

```
Queue<int> q1;
std::cout << q1 << std::endl;
q1.insert(1);
const int tmp = 2;
q1.insert(tmp);
q1.insert(std::move(tmp));
std::cout << q1 << " = 1 2" << std::endl;
q1.insert(3);
std::cout << q1 << " = 1 2 3" << std::endl;
std::cout << Queue<int>::count() << " = 1" << std::endl;
Queue<char> dll2;
std::cout << Queue<char>::count() << " = 1" << std::endl;
Queue<int> dll3;
std::cout << Queue<int>::count() << " = 2" << std::endl;
```

Oczekiwany wynik z konsoli to (program działa poprawnie):

```
Queue (empty) <=
Queue (filled) <= 1 2 2 = 1 2
Queue (filled) <= 1 2 2 3 = 1 2 3
1 = 1
1 = 1
2 = 2
```

Test II:

```
Queue<int> q1;
std::cout << q1 << std::endl;
q1.insert(1);
q1.insert(2);
q1.insert(3);
q1.insert(4);
std::cout << q1 << " = 1 2 3 4" << std::endl;
std::cout << q1.size() << " = 4" << std::endl;
q1.remove();
std::cout << q1 << " = 2 3 4" << std::endl;
q1.remove();
std::cout << q1 << " = 3 4" << std::endl;
std::cout << q1.size() << " = 2" << std::endl;
q1.remove();
std::cout << q1 << " = 4" << std::endl;
q1.remove();
std::cout << q1 << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (empty) <=
Queue (filled) <= 1 2 3 4 = 1 2 3 4
4 = 4
Queue (filled) <= 2 3 4 = 2 3 4
Queue (filled) <= 3 4 = 3 4
2 = 2
Queue (filled) <= 4 = 4
Queue (empty) <=

Test III:

```
Queue<int> q1;
std::cout << q1 << std::endl;
q1.insert(1);
std::cout << q1.front() << " = 1" << std::endl;
std::cout << q1.back() << " = 1" << std::endl;
q1.insert(2);
std::cout << q1.front() << " = 1" << std::endl;
std::cout << q1.back() << " = 2" << std::endl;
q1.insert(3);
std::cout << q1.front() << " = 1" << std::endl;
std::cout << q1.back() << " = 3" << std::endl;
q1.insert(4);
std::cout << q1.front() << " = 1" << std::endl;
std::cout << q1.back() << " = 4" << std::endl;
q1.remove();
std::cout << q1.front() << " = 2" << std::endl;
std::cout << q1.back() << " = 4" << std::endl;
q1.remove();
std::cout << q1.front() << " = 3" << std::endl;
std::cout << q1.back() << " = 4" << std::endl;
q1.remove();
std::cout << q1.front() << " = 4" << std::endl;
std::cout << q1.back() << " = 4" << std::endl;
q1.remove();
std::cout << q1.front() << std::endl;
std::cout << q1.back() << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (empty) <=
1 = 1
1 = 1
1 = 1
2 = 2
1 = 1
3 = 3
1 = 1
4 = 4
2 = 2
4 = 4
3 = 3
4 = 4
4 = 4
4 = 4
Fatal error: Trying to read uninitialized doubly linked list...
0
Fatal error: Trying to read uninitialized doubly linked list...
0

Test IV:

```
Queue<int> q1;
std::cout << q1 << std::endl;
q1.insert(1);
q1.insert(2);
q1.insert(3);
q1.insert(4);
std::cout << q1.size() << " = 4" << std::endl;
std::cout << q1 << " = 1 2 3 4" << std::endl;
q1.clear();
std::cout << q1.size() << " = 0" << std::endl;
std::cout << q1 << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (empty) <=
4 = 4
Queue (filled) <= 1 2 3 4 = 1 2 3 4
0 = 0
Queue (empty) <=

Test V:

```
Queue<__int64> q1;
q1.insert(1);
q1.insert(2);
q1.insert(3);
Queue<__int64> q2(q1);
Queue<__int64> q3(std::move(q1));
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
q2.remove();
q3.remove();
q3.remove();
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (filled) <= 1 2 3 Queue (filled) <= 1 2 3 Queue (filled) <= 1 2 3
Queue (filled) <= 1 2 3 Queue (filled) <= 2 3 Queue (filled) <= 3

Test VI:

```
Queue<__int64> q1;
q1.insert(1);
q1.insert(2);
q1.insert(3);
Queue<__int64> q2 = q1;
Queue<__int64> q3 = std::move(q1);
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
q2.remove();
q3.remove();
q3.remove();
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (filled) <= 1 2 3 Queue (filled) <= 1 2 3 Queue (filled) <= 1 2 3
Queue (filled) <= 1 2 3 Queue (filled) <= 2 3 Queue (filled) <= 3

Test VII:

```
Queue<__int64> q1;
q1.insert(1);
q1.insert(2);
q1.insert(3);
Queue<__int64> q2;
q2.insert(4);
q2.insert(5);
q2 += q1;
Queue<__int64> q3;
q3.insert(6);
q3.insert(7);
q3 += std::move(q1);
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
q2.remove();
q3.remove();
q3.remove();
std::cout << q1 << " " << q2 << " " << q3 << std::endl;
```

Oczekiwany wynik z konsoli to (program przechodzi test):
Queue (filled) <= 1 2 3 Queue (filled) <= 4 5 1 2 3 Queue (filled) <= 6 7 1 2 3
Queue (filled) <= 1 2 3 Queue (filled) <= 5 1 2 3 Queue (filled) <= 1 2 3