Лекція 9. Списки

Поняття зв'язної структури даних.
Тип для моделювання ланок лінійного однозв'язного списку.
Побудова списку з послідовності значень.
Перебір елементів списку, виведення на консоль.
Вставляння значення у впорядкований список.
Вилучення списку з динамічної пам'яті.
Сортування списком. Алгоритм злиття впорядкованих послідовностей.

Ми вже вміємо використовувати масиви для зберігання послідовності значень. Наприклад, для зберігання {2, 4, 6, 8} можемо використати статичний масив:

```
const int n = 4;
int A[n] = {2, 4, 6, 8};
```

Якщо кількість членів послідовності не відома на етапі трансляції, використаємо гнучкіший варіант і створимо динамічний масив:

```
int n; cin >> n;     // припустимо, n == 4
int * B = new int[n];
for (int i = 0; i < n; ++i) B[i] = (i + 1) * 2;
```

У обох випадках створені нами структури даних використовують *векторну пам'ять*, тобто пам'ять, виділену неперервною ділянкою, поділену на індексовані комірки однакового розміру. Це дуже зручний тип пам'яті, оскільки надає швидкий доступ до довільної комірки за її номером (індексом). Але чи у всіх випадках він підійде для ефективного задоволення потреб програми?

Уявіть, що в деякий момент програмі стало мало чотирьох парних чисел, і вона потребує ще одного – 10 наприкінці масиву. Як його туди дописати? Потрібен довший масив з додатковим елементом. Як його додати? Для статичного масиву *A* таке завдання непосильне. Неможливо змінити розмір статичного масиву без повторної компіляції. Для динамічного масиву *B* можемо виконати наступне:

```
int * C = new int[n + 1];        // нова ділянка векторної пам'яті
for (int i = 0; i < n; ++i) C[i] = B[i];  // копіюємо старий вміст
C[n++] = 10;                              // додаємо нове значення
delete[] B; B = C;                // міняємо старий масив на новий
// тепер масив B має на один елемент більше
```

Все вдалося, але ціною певних затрат часу і пам'яті. Подібні перебудови довелося б виконувати і в інших випадках, що потребують зміни структури масиву: якщо потрібно дописати значення на його початок (наприклад, нуль), вставити значення всередину (вставити у послідовність непарні числа) тощо.

Якщо зміна розташування даних у програмі відбувається часто, то, можливо, доцільно використовувати не векторну, а *зв'язну пам'ять*. Зв'язною називають пам'ять, яка займає декілька, можливо, несуміжних ділянок, пов'язаних між собою посиланнями. У програмах мовою C++ зв'язні структури будують за допомогою вказівників. Розглянемо способи побудови та використання однієї з найпростіших зв'язних структур даних – *лінійного однозв'язного списку*.

З математичного погляду *список* – це скінчена *послідовність пов'язаних* між собою *об'єктів* довільної природи і розміру. Ці об'єкти прийнято називати ланками. Впорядкованість ланок залежить від призначення списку, і нею може керувати програміст. Ланку списку умовно можна поділити на дві частини:

1) тіло (елемент даних);

2) довідкова інформація про розмір і тип конкретної ланки, взаємозв'язок з іншими ланками тощо.

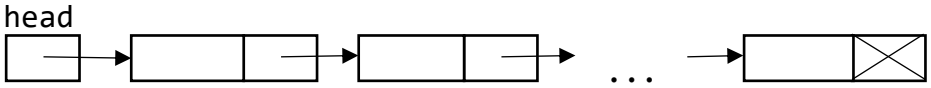
Зі списком можна виконувати такі *основні операції*:

- перехід до сусідньої ланки;
- включення у довільному місці нової ланки;
- вилучення зі списку довільної ланки;
- зміна порядку ланок у списку.

Зазначимо, що перелічені вище дії не вимагають фізичного переміщення ланок, а лише передбачають зміну довідкової частини окремих ланок.

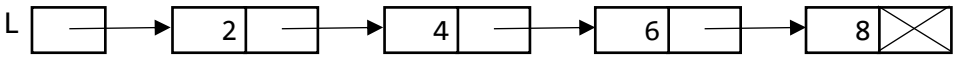
Є багато різних *типів списків*. Найрозповсюдженішими з них є такі:

- лінійний однозв'язний список;
- двозв'язний список;
- кільцевий, як частковий випадок двозв'язного;
- ієрархічний;
- асоціативний.



Ми детальніше розглянемо реалізацію засобами мови C++ алгоритмів опрацювання однозв'язних списків, тобто таких, у яких чергова ланка має вказівку на місцезнаходження наступної ланки (див. рис.). Остання ланка не має наступника і тому містить порожній вказівник *nullptr*. Вказівка на першу ланку є у змінній – заголовку списку. Якщо список порожній (немає жодного елемента), то в заголовку – *nullptr*.

Згадана на початку лекції послідовність чисел мала б такий вигляд:



Який тип використати для створення ланок зображеної структури? Очевидно, ланка містить різнотипні значення: ціле число і вказівник. Тому використаємо структуру з двох полів. Перше поле містить елемент даних – ціле число, тому матиме тип *int*. Друге поле містить вказівник, що вказує на таку саму ланку (структуру), членом якої він є. Таким чином, приходимо до оголошення:

```
struct Node
{
     int value;     // елемент даних
     Node* link;    // поле зв'язку
     Node(int x, List_t p = nullptr) :value(x), link(p) {}
};
```

Справді, поле *link* є частиною структури *Node* і містить вказівник на (наступний) *Node*. Заради певної зручності типові «вказівник на ланку» *Node** варто дати окреме ім'я. Ми вже вміємо це робити за допомогою *typedef* або *using*. Але виникає одна трудність: щоб оголосити вказівник на *Node*, треба мати оголошення структури *Node*, а, щоб оголосити структуру, треба мати ім'я вказівника – замкнуте коло! C++ розриває його за допомогою *попереднього оголошення* типу *Node*, як в тексті нижче.

```
struct Node; // попереднє оголошення типу
typedef Node* List_t;
struct Node  // власне оголошення
{
     int value;     // елемент даних
     List_t link;   // поле зв'язку
     Node(int x, List_t p = nullptr) :value(x), link(p) {}
};
```

Тепер бажану послідовність чисел можна утворити програмно за допомогою таких інструкцій:

```
List_t L = new Node(2, new Node(4));
L->link->link = new Node(6);
L->link->link->link = new Node(8);
```

Тут не надто привабливо виглядає велика кількість переходів за вказівниками в полі зв'язку, і одразу виникає питання: як «автоматизувати» видовження ланцюжка *link->* так, щоб можна було створювати списки будь-якої довжини.

Відповідь дамо у вигляді процедури, що завантажує в список послідовність чисел, уведену з клавіатури. Ознакою закінчення послідовності є будь-який нечисловий символ.

```
List_t readList()
{
	// кожне прочитане число дописується в кінець списку
	List_t L = nullptr;
	cout << "Input a succession of integers terminated by 'stop':\n";
	int x;
	cin >> x;
	// перша ланка особлива, бо на неї вказує голова списку
	// її створюємо окремо
	L = new Node(x);
	// допоміжний вказівник на останню ланку
	Node* p = L;
	// цикл для решти чисел і ланок
	while (cin >> x)
	{
		p->link = new Node(x);
		p = p->link;
	}
	// відновлення стану потоку введення
	cin.clear();
	while (cin.get() != '\n') continue;
	return L;
}
```

Пояснення є в тілі функції в коментарях. Один з найважливіших – перша ланка списку особлива! Вона єдина, на кого вказує заголовок списку. На всі інші вказує поле зв'язку попередньої ланки. Через цю особливість першу ланку було створено окремо ще перед початком циклу.

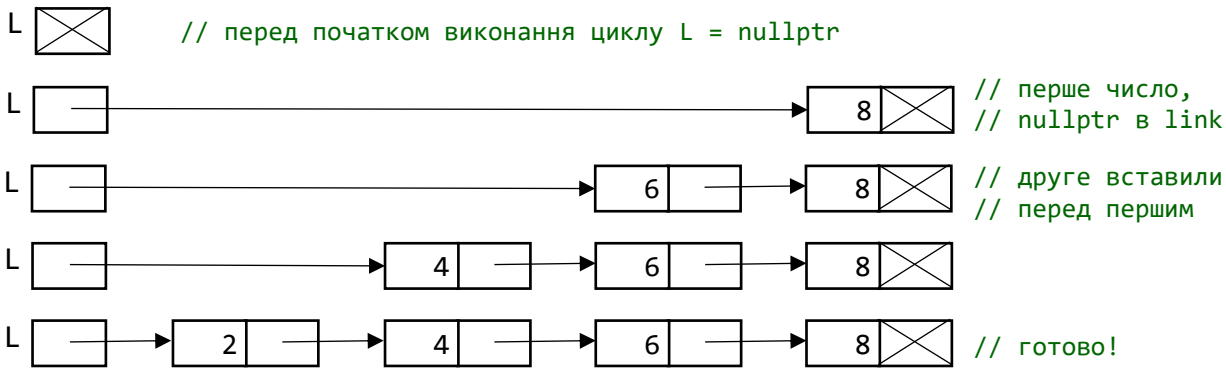
У більшості алгоритмів першу ланку доводиться опрацьовувати окремо. На щастя, є й такі, для яких байдуже, з якою ланкою працювати. Один з них – алгоритм виведення елементів списку на екран.

```
void writeList(List_t L)
{
	if (L == nullptr)
	{
		cout << "The list is empty\n";
		return;
	}
	// локальна змінна L перебере всі ланки списку
	while (L != nullptr)
	{
		cout << '\t' << L->value;
		L = L->link;
	}
	cout << '\n';
}
```

Можна спробувати обійти особливість першої ланки і працювати з усіма числами так, ніби кожне з них перше в послідовності:

```
List_t reverseReadList()
{
	// кожне число вставляється на початок списку
	cout << "Input a succession of integers terminated by 'stop':\n";
	int x;
	List_t L = nullptr;
	while (cin >> x)
	{
		L = new Node(x, L);
	}
	cin.clear();
	while (cin.get() != '\n') continue;
	return L;
}
```

Тут усі числа опрацьовані в одному циклі, але послідовність завантажено в список у оберненому порядку. Якщо користувач уведе послідовність «8 6 4 2 stop», то процес побудови списку матиме вигляд, як на рисунку нижче.



Переваги списку стають очевидними тоді, коли потрібно втрутитися всередину послідовності: вставити чи вилучити значення. Давайте розглянемо таке завдання: потрібно вставити задане число у щойно створений список так, щоб не порушити впорядкованості чисел. (Ви ж помітили, що ми отримали зростаючу послідовність?)

Можливі дві ситуації, що суттєво відрізняються одна від одної:

1) задане число не більше за перший елемент списку, тоді його потрібно вставити перед першою ланкою (зміниться вказівник *L*);
2) задане число більше за перший елемент списку, тоді потрібно знайти йому місце в списку і вставити між двома ланками, змінивши поле *link* першої з них.

Перша ситуація є доволі простою і не вимагає особливих пояснень: вставляння першої ланки ми виконували в циклі в попередній функції. А от пошуку місця при ситуації 2 варто приділити трохи уваги. Щоб знайти серед елементів списку значення, більше за задане число, потрібно в циклі перебрати ланки списку від другої дод останньої. На якій ланці потрібно зупинити пошук? Наприклад, якщо шукаємо місце для 5, то який вказівник нам потрібен: на ланку з числом 6, чи на ланку з 4? Не поспішайте з відповіддю. На перший погляд – на 6, бо 6 > 5, але як тепер виконати вставку? Адже змінити потрібно поле зв'язку *попередньої* ланки. Нам потрібен вказівник на ланку з четвіркою! Отже, дивитися треба на 6, а зупинитися на 4. Тобто, під час перебирання ланок потрібно заглядати на одну ланку попереду. Якщо *p* вказівник на поточну ланку, то *p->value* – значення в цій ланці, а *p->link->value* – в наступній. Такий запис і використаємо.

Випадок, коли задане число більше за останній елемент списку, є частковим випадком ситуації 2, просто «вказівником на другу ланку» в парі, між якими треба вставити число, буде *nullptr*. Розв'язком завдання є функція *insert*:

```
void insert(int x, List_t& L)
{
	if (x <= L->value)
	{	// вставляємо першу ланку списку
		L = new Node(x, L);
	}
	else	// шукатимемо місце
	{
		List_t place = L;
		// поки є ланка попереду і значення в ній замале
		while (place->link != nullptr && place->link->value < x)
			// йдемо до наступної ланки
			place = place->link;
		// вставка нового елемента
		place->link = new Node(x, place->link);
		place = nullptr;
	}
}
```

Зверніть увагу на умову ітераційного циклу. Вираз *place->link != nullptr* обов'язково має бути першим! У випадку, коли перебір досяг останньої ланки списку, вираз набуде значення *false*, і до обчислення другої половини умови справа не дійде. І це дуже добре, бо *place->link->value* просто не існує. Якщо спробувати поміняти місцями вирази в умові циклу, то в якийсь момент програма завершиться аварійно, як тільки спробує розіменувати порожній вказівник.

Параметр *List_t& L* не випадково є посиланням, адже функція змінює *L*, якщо вставляє ланку на початок списку.

У функції *insert* ми особливим способом опрацювали першу ланку, адже вона «не така, як інші». А чи не можна якось обійти цю особливість, і записати універсальний алгоритм, що не робить різниці між ланками? Адже алгоритми без особливостей зазвичай є надійнішими. Щоб досягти бажаного, доведеться позбавити першу ланку її особливого статусу. Чим вона не така? Вона не має попередньої ланки. То створимо її! Хоча б на час виконання вставки. Оновлена версія функції *insert* матиме вигляд:

```
void insert(int x, List_t& L)
{
	Node phantom(0, L);
	// перед першою ланкою списку з'явилася ще одна
	// тепер шукатимемо місце незалежно від того, чи ланка перша чи ні
	List_t place = &phantom;
	// поки є ланка попереду і значення в ній замале йдемо до наступної ланки
	while (place->link != nullptr && place->link->value < x)
		place = place->link;
	// вставка нового елемента
	place->link = new Node(x, place->link);
	place = nullptr;
	L = phantom.link;
	// локальна змінна phantom автоматично ліквідується при виході з функції
}
```

Ми досить уважно розбиралися з побудовою списків. Тепер варто звернути увагу на очищення динамічної пам'яті від списку. Зробити одне *delete L;* було б помилкою, бо така інструкція очистить пам'ять тільки від *однієї* ланки списку – від першої. Всі інші буде втрачено! Щоб цього не сталося, потрібно перед вилученням першої ланки списку запам'ятовувати адресу продовження:

```
int delFirst(List_t& L)
{	// вилучає зі списку першу ланку, повертає значення, яке в ній зберігалося
	int x = L->value;
	List_t victim = L;
	L = L->link;
	delete victim;
	return x;
}
```

Тепер легко записати процедуру вилучення цілого списку.

```
void removeList(List_t& List)
{
	// вилучає з динамічної пам'яті весь список
	while (List != nullptr)
		delFirst(List);
}
```

Сортування списком

У попередній лекції ми обговорювали можливості використання зв'язної структури список для гнучкого зберігання послідовності значень. Зокрема, ми оголосили процедуру вставляння значення до впорядкованого списку. Її можна використати для впорядкування масиву чисел: спочатку вставимо елементи масиву по одному до впорядкованого списку, а тоді скопіюємо їх назад. Такий алгоритм мав би працювати суттєво швидше від звичайного алгоритму вставками, оскільки він не потребує переміщень елементів усередині масиву.

```
void sortByList(int * A, size_t n)
{
	// щоб упорядкувати масив A, помістимо всі його елементи у впорядкований список
	List_t List = nullptr;
	for (size_t i = 0; i < n; ++i) insert(A[i], List);
	// а тоді повернемо зі списку назад в масив
	List_t curr = List;
	for (size_t i = 0; i < n; ++i, curr = curr->link) A[i] = curr->value;
	removeList(List);
}
```

Тут використано оголошені раніше функції *insert* і *removeList*.

Злиття впорядкованих послідовностей

У майбутньому вам стане в пригоді вміння об'єднати дві впорядковані послідовності в одну, також впорядковану. Алгоритм такого об'єднання називають алгоритмом злиття. Розглянемо його на прикладах злиття масивів і злиття списків.

Здавалося б, що тут вигадувати, якщо ми вміємо вставляти елемент у впорядковану послідовність? Вставимо по одному елементи однієї послідовності в іншу, та й по всьому. Але є набагато ефективніший спосіб: вважатимемо, що обидві послідовності рівноправні, їхні початкові елементи конкурують за потрапляння до об'єднаної послідовності, і проходить менший з них; конкуренція припиниться, коли закінчиться одна з послідовностей – тоді залишиться дописати в результат «хвіст» іншої, що залишився.

```
// об'єднання двох впорядкованих масивів у новий
int * merge(int*A, size_t n, int*B, size_t m)
{
	size_t i = 0, j = 0, k = 0;
	int* C = new int[n + m]; // новий масив міститиме обидва задані
	while (i < n && j < m)   // конкурують елементи заданих масивів
	{                        // «виграє» менший
		if (A[i] < B[j]) C[k++] = A[i++];
		else C[k++] = B[j++];
	}
	// дописування «хвоста», спрацює один з циклів
	while (i < n) C[k++] = A[i++];
	while (j < m) C[k++] = B[j++];
	return C;
}
```

Злиття списків дуже схоже до злиття масивів. Відмінність лише в способі доступу до елемента послідовності та переходу до наступного: *A*[*i*] – у масиві, *a->value* – у списку; *++i* – у масиві, *a* = *a->link* – у списку.

```
// об'єднує два впорядковані списки (копіює елементи в новий)
List_t merge(List_t a, List_t b)
{
	// допоміжна ланка, щоб перша ланка результату не була "особливою"
	Node phantom(0);
	List_t r = &phantom;
	while (a != nullptr && b != nullptr)
	{
		// до результуючого списку першим проходить менший
		if (a->value < b->value)
		{
			r->link = new Node(a->value);
			a = a->link;
		}
		else
		{
			r->link = new Node(b->value);
			b = b->link;
		}
		r = r->link;
	}
	// дописуємо "хвости"
	while (a != nullptr)
	{
		r->link = new Node(a->value);
		a = a->link;
		r = r->link;
	}
	while (b != nullptr)
	{
		r->link = new Node(b->value);
		b = b->link;
		r = r->link;
	}
	return phantom.link;
}
```

Попередні дві функції *копіюють* елементи заданих послідовностей в нову. Списки можна об’єднати інакше: *перемістити* ланки заданих послідовностей в одну нову. При цьому значення елементів не копіюють, а змінюють лише значення полів зв’язку. Після завершення злиття перший заданий список міститиме результат, а другий – спорожніє.

```
void MergeAndCut(List& A, List& B)
{
	List C;
	if (A->value <= B->value) C = A;
	else
	{
		C = B; B = A; A = C;
	}
	while (B != nullptr)
	{
		while (A->link != nullptr && A->link->value < B->value)
			A = A->link;
		List t = A->link;
		A->link = B;
		B = t;
		A = A->link;
	}
	A = C;
}
```

На жаль, я не знайшов у підручниках прикладів побудови списків. Є один у Прати, але там він «вкладений» в оголошення класу, який моделює чергу. Це трохи зарано для прочитання, хіба для сміливців.

Література

1. Стивен Прата Язык программирования C++.