8. Опрацювання текстової інформації

Обробка текстової інформації складає основу більшості програм. Завдяки таким алгоритмам використання комп'ютерів стало можливим у різних сферах життя.

Довільний алгоритм роботи з символами по суті включає лише дві основні дії: пошук і заміну. Згідно з тезою Маркова, довільний алгоритм можна зобразити як послідовність таких дій.

Зрозуміло, що пошук можна використати для виявлення певних властивостей вхідного тексту. Заміна дає змогу цей текст модифікувати певним чином.

Розглянемо декілька алгоритмів роботи з літерною інформацією для ілюстрації основних прийомів її опрацювання. Зокрема поговоримо про перетворення числа на текст і навпаки, тобто, про перетворення між внутрішнім (для програми) та зовнішнім (для користувача) кодуванням числової інформації. Такі перетворення відбуваються кожного разу при введенні та виведенні числових даних, тому було б добре розібратися у роботі алгоритмів перетворень.

8.1. Розпізнавання чисел

Задача 35. *Задано послідовність символів, яку закінчено крапкою. Визначити кількість цифр у цій послідовності.*

Щоб розв'язати поставлену задачу, треба врахувати, що символи всіх цифр розташовано у кодовій таблиці підряд, починаючи від '0'. Тому цифрою є довільний символ c, для якого істинною є умова $'0' \le c \le '9'$.

Найпростіший алгоритм підрахунку цифр використовує посимвольне введення.

```
void CountCharByChar()
{
	cout << "\n *Кількість цифр у рядку літер*\n\n"
		<< "Введіть послідовність літер, що закінчується крапкою:\n";
	unsigned digits_quantity = 0;	// лічильник цифр
	char letter;
	do
	{
		letter = cin.get();	// отримання з потоку чергової літери
		if (letter >= '0' && letter <= '9') ++ digits_quantity;
	} while (letter != '.');	// опрацювання триває до першої крапки

	cout << "Послідовність містить " << digits_quantity << " цифр\n";
	return;
}
```

Посимвольне введення з потоку трапляється не часто, тому варто розглянути інші способи розв'язування задачі: з використанням структури даних *рядок*. У цьому випадку задана послідовність може не містити термінального елемента (крапка в умові задачі), оскільки закінчення рядка позначає структура даних. У мові C++ можна використовувати рядки «в стилі C» та контейнери-рядки. Рядок в стилі C – це масив літер, що закінчується термінальною літерою '\0', а контейнер-рядок – змінна типу *string*. Ми розглянемо обидва способи. Головна їхня відмінність у способах резервування пам'яті та введення рядків: для масиву потрібно вказати сталий розмір, а контейнер «вміє» підлаштовувати свій розмір відповідно до потреб.

```
#include <string>
void CountInString()
{
	cout << "\n *Кількість цифр у рядку - масиві літер та в контейнері*\n";
	cout << "\n-Рядок в стилі С-\n";
	const unsigned size = 256;    // потрібно задати розмір масиву літер
	char c_line[size] = { '\0' }; // масив ініціалізуємо літерами з кодом 0
	cout << "Введіть послідовність літер, що містить цифри:\n";
	cin.get(c_line, size);        // введений рядок обмежений розміром масиву
	while (cin.get() != '\n')     // очищаємо потік від можливого залишку рядка
		continue;
	unsigned digits_quantity = 0;             // лічильник цифр
	for (unsigned i = 0; c_line[i] != '\0'; ++i)
	{
		if (c_line[i] >= '0' && c_line[i] <= '9') ++digits_quantity;
	}
	cout << "Послідовність містить " << digits_quantity << " цифр\n";

	cout << "\n-Рядок бібліотеки std-\n";
	std::string line;         // контейнер змінного розміру
	cout << "Введіть послідовність літер, що містить цифри:\n";
	getline(cin, line);       // будуть прочитані всі літери до кінця рядка '\n'
	digits_quantity = 0;      // лічильник цифр використаємо той самий
	for (unsigned i = 0; i < line.length(); ++i)
	{
		if (line[i] >= '0' && line[i] <= '9') ++digits_quantity;
	}
	cout << "Послідовність містить " << digits_quantity << " цифр\n";
	return;
}
```

Розроблені алгоритми можна використати для розв'язування складнішої задачі, яку розв'язує кожен компілятор.

Задача 36. *Задано послідовність символів. Перевірити, чи задана послідовність є записом цілого числа у десятковій системі числення, і якщо так, то обчислити значення цього числа.*

Ціле число – це послідовність цифр, якій може передувати знак '+' чи '–'. Її можна зобразити рядком – масивом літер або структурою даних типу *string*, і скористатись стандартними засобами обробки рядків, або детально описати опрацювання кожного символу. Ми опишемо обидва підходи.

Під час посимвольного розпізнавання доведеться виконати декілька завдань: перевірити, чи не починається рядок з пропусків – їх треба пропустити і знайти початок запису числа; опрацювати знак, якщо він є (плюс можна пропустити, а мінус – запам'ятати); перетворити кожну цифру до числового значення. Код літери '0' відмінний від нуля, а літери '5' – від п'яти, то як же їх перетворити на числа? Допоможе проста властивість кодів цифр: у таблиці кодів вони розташовані підряд, тому код '5' більший від коду '0' на 5. Таким чином, щоб перетворити цифру до її числового значення, достатньо відняти від її коду код літери '0'. У мові програмування C++ літерний тип є одним з цілих типів, тому до його значень можна застосовувати оператор віднімання без жодних додаткових перетворень:

```
	// перетворення цифри c на число k
	char digit = '5'; // або '0', '3', '8', ...
	int number = c - '0';
```

Нагадаємо, що побудову значення числа за значеннями його цифр ми вже виконували в алгоритмі *IsPalindrome* (п. 1.2).

Правильний запис цілого десяткового числа може починатися одним знаком '+' чи '–' і містить лише десяткові цифри. Усередині запису не має бути ніяких інших символів: крапки, букви, пропуски, зірочки тощо – заборонені. Для перевірки правильності ми можемо послідовно аналізувати всі символи рядка. Достатньо знайти один неправильний символ, щоб зупинити цикл перевірки і повідомити користувача про помилку в записі.

На практиці може трапитися інша, не така очевидна помилка: запис числа може бути задовгим. Найбільше ціле без знаку, яке можна помістити в пам'ять комп'ютера в змінну стандартного типу має значення 18 446 744 073 709 551 615. Його запис містить 20 цифр, а довжина рядка може перевищувати 4000 знаків. У такому випадку для відображення числа у пам'ять використовують спеціальні типи. Про один з них ми поговоримо в наступному розділі.

Як повідомляти користувача про виявлені помилки в записі? Існує декілька способів. Функція розпізнавання числа могла б друкувати текстові повідомлення на екран, але такі повідомлення не можуть читати інші функції програми. За класичного підходу функція мала б встановити код завершення, що може містити код помилки. Але тоді в неї буде два результати: саме число та код завершення. Для такого коду використовують додатковий параметр функції або спеціальну глобальну змінну. Сучасний підхід до сповіщення про помилки використовує механізм винятків. Але він дещо затратний з огляду на ресурси комп'ютера.

Ми використаємо класичний підхід і додамо до списку параметрів функції перетворення рядка спеціальний параметр *pos* для повернення ознаки того, чи було перетворення успішним. Параметр міститиме номер першого неправильного символу в рядку, якщо такий є, або –1 в протилежному випадку. Ми не будемо окремо повідомляти про переповнення, а задовгий запис вважатимемо неправильним. Для простоти алгоритму перевірку та перетворення об'єднаємо в одному циклі.

```
int RecognizeInteger(const char* line, int& pos)
{
	const int max = INT_MAX / 10;
	const char* curr = line;
	int result = 0;                    // число, яке треба сформувати
	pos = -1;
	while (*curr == ' ' || *curr == '\t') ++curr;  // пропускаємо пропуски
	bool sign = false;                 // sign пам'ятає про наявність мінуса
	if (*curr == '+') ++curr;          // плюс можна пропустити,
	if (*curr == '-')                  // а мінус треба запам'ятати
	{
		sign = true;
		++curr;
	}
	while (*curr != '\0')              // переглядаємо весь рядок
	{
		if (*curr >= '0' && *curr <= '9')
		{                                // знайдену цифру перетворюємо на число
			int digit = *curr - '0';  // чи не загрожує переповнення?
			if (max < result || max == result && digit > 7)
			{                              // якщо так, то перериваємо обчислення
				pos = curr - line;
				break;
			}
			result = result * 10 + digit;// додаємо нову цифру до результату
```

```
                ++curr;
            }
            else
            {                                   // фіксуємо позицію нечислового символа
                pos = curr - line;
                break;
            }
        }
        return sign ? -result : result;
}
```

Наша функція в деталях описує процес перетворення рядка символів на число. Якщо ці деталі не дуже важливі для вашої програми, ви можете використати і стандартні засоби. Функція *int atoi(const char*)* дуже подібна за можливостями до нашої. Для коду завершення вона використовує глобальну змінну *errno*. Головна відмінність *atoi* у тому, що перший нечисловий символ у записі вона сприймає як ознаку закінчення, а не як помилку.

Опишемо ще один стандартний спосіб. Перетворення рядка на число відбувається кожного разу під час уведення з консолі чи файлу. Таке перетворення вміють робити потокові оператори введення. Якщо накласти потік введення на рядок, то число вдасться прочитати за допомогою *operator>>*. Можливості цього оператора такі ж, як у функції *atoi*. Використання усіх згаданих засобів продемонстровано в програмі нижче.

```
void StringToNumber()
{
        cout << "\n *Перетворення рядка (масиву літер та контейнера) на число*\n";

        cout << "\n-Рядок в стилі С-\n";
        const unsigned size = 256;     // потрібно задати розмір масиву літер
        char c_line[size] = { '\0' }; // масив ініціалізуємо літерами з кодом 0
        cout << "Введіть послідовність літер, що містить цифри:\n";
        cin.get(c_line, size);         // введений рядок обмежений розміром масиву

        cout << "\n   Перетворення власною функцією\n";
        int error_code;
        int number = RecognizeInteger(c_line, error_code);
        if (error_code == -1)
              cout << "Введене число: " << number << '\n';
        else
              cout << "Рядок містить помилку в позиції " << error_code
              << " Вдалося прочитати " << number << '\n';

        cout << "\n   Перетворення стандартною функцією\n";
        number = std::atoi(c_line);
        if (errno == ERANGE) cout << "Помилка: Сталося переповнення\n";
        cout << "Введене число: " << number << "  Код помилки " << errno << '\n';
        errno = 0;

        cout << "\n   Використання потоку, накладеного на рядок\n";
        std::string line(c_line);
        std::istringstream stream(line);
        stream >> number;                  // перетворення виконує оператор введення
        cout << "Введене число: " << number << '\n';
        return;
}
```

Функцію *RecognizeInteger* можна використати дещо несподіваним способом. Несподіванка в тому, що код помилки завершення не завжди свідчить про помилку. Продемонструємо сказане на прикладі.

Задача 37. *Задано рядок символів, що містить послідовність записів цілих чисел, відокремлених пропусками. Обчисліть суму членів послідовності.*

Схожі задачі ми розв'язувати в розділі 2. Ця відрізняється лише джерелом вхідних даних. Для отримання чергового числа з рядка використаємо *RecognizeInteger*. Її параметр *pos* повідомлятиме позицію першого нечислового символу, пропуска, що позначає закінчення запису попереднього числа та, водночас, початку наступного. Процес читання даних завершиться, коли *pos* поверне значення –1: останнє число має прочитатися без помилок.

```
void IntSuccession()
{
	cout << "\n *Обчислення суми цілих чисел, записаних у рядку*\n";
	char* succ = "25 -32 777 12 -98 -5 2018";
	cout << "Задана послідовність:\n" << succ << '\n';
	int sum = 0;
	int pos = 0;
	cout << "Члени послідовності:\n";
	do
	{
		int term = recognizeInteger(succ += pos, pos);
		cout << term << '\n';
		sum += term;
	} while (pos != -1);
	cout << " Сума = " << sum << '\n';
}
```

Тут зображення послідовності чисел задано статичним рядком, але ця обставина не є обов'язковою. Програма не зміниться, якщо рядок увести з консолі чи з файлу.

8.2. Пошук і заміна в рядку

У більшості реалізацій рядок літер займає неперервну ділянку пам'яті, тому зміну довжини рядка виконують як виділення нової ділянки потрібного розміру, копіювання та звільнення попередньої. Таким чином заміна фрагмента рядка новим зазвичай призводить до побудови нового рядка. Підстановку вдається виконати «на місці», якщо фрагмент замінюють новим такої ж довжини. Найпростіше проілюструвати сказане на прикладі заміни однієї літери іншою.

Припустимо, потрібно замінити в заданому рядку кожну малу літеру 'a' на велику. У цьому випадку рядок можна трактувати як масив елементів типу *char* і працювати з ним відповідно: пошук і заміну можна виконати за один перегляд масиву.

```
	// заміна літери
	char test_line[] = "abrakadabra";
	cout << "before: " << test_line << endl;
	unsigned i = 0;
	while (tes_line[i] != '\0')
	{
		if (test_line[i] == 'a') test_line[i] = 'A';
		++i;
	}
	cout << "after: " << test_line << endl;
```

Ситуація ускладнюється, якщо фраґменти мають різний розмір. Наприклад, якщо новий фраґмент довший, то новий рядок потребує більше місця. Аби уникнути перебудов пам'яті під час кожної підстановки і ефективно реалізувати всі заміни, доцільно спочатку визначити розмір нового рядка, зарезервувати для нього місце, і тільки тоді братися за його побудову.

Задача 38. *Запропонувати ефективний спосіб заміни вказаної літери в заданому рядку на задану групу літер.*

Заданий рядок переглядатимемо двічі: перший раз, щоб порахувати кількість замін і визначити розмір нового рядка, другий – щоб виконати всі підстановки. Додаткові затрати на перший перегляд рядка окупляться з надлишком завдяки швидкості підстановок.

```
char* Replace(char* source, char what, const char* by)
{
	// порахуємо, скільки разів зустрічається шукана літера
	// водночас дізнаємося довжину рядка
	unsigned counter = 0;
	unsigned length = 0;
	while (source[length] != '\0')
	{
		if (source[length] == what) ++counter;
		++length;
	}
	// виділимо достатньо місця для нового рядка
	unsigned by_len = strlen(by);
	char* new_line = new char[length + 1 + (by_len - 1) * counter];
	unsigned dest = 0;
	for (unsigned i = 0; i < length; ++i)  // і виконаємо заміни
	{
		if (source[i] != what) new_line[dest++] = source[i];
		else
		{
			for (unsigned j = 0; j < by_len; ++j) new_line[dest++] = by[j];
		}
	}
	new_line[dest] = '\0';   // термінальна літера
	return new_line;
}
```

Використати функцію *Replace* можна, наприклад, так:

```
	// заміна літери на групу літер
	char cpp[] = "I like C++";
	cout << "before: " << cpp << endl;
	char* res = Replace(cpp, '+', " plus");
	cout << "after: " << res << endl;
	delete[] res;
```

Алгоритм заміни групи літер меншою залишимо як вправу для читача.

8.3. Форматування виведення числової інформації

Для ілюстрації пошуку і заміни в тексті розглянемо ще декілька задач, які виникають при форматному виведенні. Оператор виведення в потік перетворює число на рядок

символів і форматує його відповідно до своїх налаштувань. Частковий випадок такого перетворення ми вже розглядали в параграфі 1.4, коли будували запис цілого числа у двійковій та шістнадцятковій системах числення.

Зазвичай оператор виведення перетворює ціле число на рядок, що містить його запис у десятковій системі без пропусків. Щоб змінити ширину поля виведення чи основу системи числення використовують відповідні методи потоку або маніпулятори. Наприклад, щоб надрукувати задане ціле у шістнадцятковій системі в полі виведення з десяти символів, можна використати таку послідовність інструкцій:

```
int num; cin >> number;
cout.setf(ios_base::hex, ios_base::basefield);
cout.width(10);
cout << "num(16) = " << number << '\n';
```

Використання маніпуляторів дещо скорочує запис, проте також є досить багатослівним:

```
int num; cin >> number;
cout << hex << setw(10) << "num(16) = " << number << '\n';
```

Зручною альтернативою могло б стати використання специфікацій форматування. Наприклад, бажаний результат ми могли б отримати простіше, якби мали процедуру, що «розуміє» специфікації форматів виведення:

```
int num; cin >> number;
FormattedPrint("num(16) = %h10\n", number);
```

Тут символ '*%*' позначає початок специфікації, '*h*' означає шістнадцяткову систему числення, а '10' – ширину поля виведення. Процедура *FormattedPrint* мала б знайти в рядку специфікацію, розпізнати її та замінити на відповідний запис числа *num*.

Задача 39. *Задано рядок і ціле число. У рядку розташовано спеціальні символи, які задають специфікації перетворення числа при виведенні. Перетворити текст відповідно до специфікацій і надрукувати його. Тобто, помістити число у вказане місце рядка у відповідному вигляді.*

Початок специфікації позначено літерою '%', наступна літера задає основу числення: 'b' або 'B' позначають двійкову систему, 'o' або 'O' – вісімкову, 'd' або 'D' – десяткову, 'h' та 'H' позначають шістнадцяткову систему з написанням малих та великих літер відповідно. Після основи може бути вказано ціле число, що задає ширину поля виведення: якщо воно більше за довжину запису числа, то запис доповнюється зліва пропусками, якщо менше або відсутнє, то шириною вважають довжину запису числа. Щоб надрукувати літеру '%', яка не є специфікацією формату, її потрібно вказати двічі підряд.

Для розв'язання цієї задачі нам доведеться виконати декілька простіших завдань: знайти в заданому рядку початок специфікації форматування – це легко зробити послідовним переглядом літер рядка; розпізнати основу системи числення – оскільки її задано одною літерою, то це легко зробити за допомогою інструкції вибору; розпізнати число, що задає ширину поля виведення – можемо використати функцію *RecognizeInteger*, яку ми визначили у попередньому параграфі; побудувати запис заданого числа в певній системі числення – модифікуємо функцію *HexaFormStr* (або *BinaryFormStr*), визначену в п. 1.4 так, щоб основу системи числення можна було передавати їй параметром. Про виявлені можливі помилки в структурі специфікації повідомлятимемо винятками.

Почнемо з функції, що будує запис цілого числа в заданій системі числення. Передаватимемо їй у параметрах це число, основу системи числення, ознаку того, які літери (великі

чи малі) використовувати в записі, та бажану довжину побудованого рядка. Перш за все така функція мала б аналізувати задане число: нуль не потрібно ніяк перетворювати, оскільки у всіх системах його записують однаково. Для зображення від'ємного числа використаємо прямий код (не доповняльний) – це звичний запис зі знаком мінус.

```
#include <exception>
// Функція будує рядок - запис числа number в системі з основою radix
//   довжина рядка не менша за width
char* RadixFormStr(int number, unsigned radix, bool upcase, unsigned width)
{
	if (radix < 2 || radix > 36)
		throw std::invalid_argument("Wrong radix value");
	// нуль - особливий випадок, у всіх системах записується однаково
	if (number == 0)
	{
		char * str;
		if (width > 1)
		{
			str = new char[width + 1];
			str[width] = '\0';
			str[--width] = '0';
			while (width > 0) str[--width] = ' ';
		}
		else str = new char[2]{'0', '\0'};
		return str;
	}
	// відмінне від 0 число - послідовність цифр
	bool sign;
	if (number > 0) sign = false;
	else
	{
		sign = true;
		number = -number;
	}
	// запис від'ємного числа містить '-', тому на 1 літеру довший
	int length = std::log(double(number)) / std::log(radix) + 1 + sign;
	// готуємо місце для запису
	if (width >= length)
	{
		width -= length;
		length += width;
	}
	char* str = new char[length + 1];
	str[length] = '\0';
	// запис починатиметься з пропусків
	while (width > 0) str[--width] = ' ';
	const char digit = upcase ? 'A' - 10 : 'a' - 10;
	while (number > 0)
	{                               // будуємо послідовність цифр
		--length;
		unsigned rightmost_digit = number % radix;  // остання цифра числа
		if (rightmost_digit < 10)
			str[length] = '0' + rightmost_digit;  // звичайні цифри
		else str[length] = digit + rightmost_digit; // старші цифри-букви
		number /= radix;
	}
```

```
      if (sign) str[--length] = '-';
      return str;
}
```

Тут задане ціле число передаємо параметром *number*, основу числення – параметром *radix*, а параметр *upcase* відповідає за величину літер, наприклад, у шістнадцятковому записі. Функція стала дещо складнішою, порівняно з *HexaFormStr*, оскільки тепер вона опрацьовує довільні цілі числа – не тільки натуральні. Алгоритм переведення числа до нової системи не залежить від величини її основи, тому за допомогою *RadixFormStr* можемо будувати запис числа і в таких екзотичних системах, як, наприклад, трійкова чи система з основою 32. Усі системи з основами більшими за 10 використовують як цифри літери латинського алфавіту. Зрозуміло, що за такого підходу величина основи обмежена числом 36 (10 цифр + 26 літер).

Процедуру форматованого виведення можна умовно поділити на три частини: виведення тексту, що передує специфікації форматування; розпізнавання специфікації та перетворення числа; виведення решти тексту. Під час виведення тексту потрібно пам'ятати про особливе призначення літери '%'.

```
void FormattedPrint(const char* format, int number)
{
      cout <<
   "\n *Виведення цілого числа на друк відповідно до формату, заданого рядком*\n";
      char specification;
      while (*format != '\0')
      {
            if (*format != '%') // звичайна літера
            {
                  cout << *format++;
            }
            else                // можливо, це початок специфікації
            {
                  ++format;           // перевіримо наступну літеру
                  if (*format == '%') // рядок містить літеру відсоток
                  {
                        cout << *format++;
                  }
                  else          // знайшли специфікацію
                  {
                        specification = *format++;
                        break;
                  }
            }
      }
      if (*format == '\0')
            throw std::invalid_argument("No format specification found");
      unsigned radix = 10;
      bool upcase = false;
      switch (specification)              // розпізнавання основи системи числення
      {
      case 'b': case 'B': radix = 2;  break;
      case 'd': case 'D': break;
      case 'o': case 'O': radix = 8; break;
      case 'h': radix = 16; break;
      case 'H': radix = 16; upcase = true; break;
      default: throw std::invalid_argument("Unknown format specification found");
      }
```

```
        int pos;              // розпізнавання ширини поля виведення
        int width = recognizeInteger(format, pos);
        if (width < 0)
              throw std::invalid_argument("Invalid width specification found");
        // побудова запису числа відповідно до заданого формату
        char* str = RadixFormStr(number, radix, upcase, width);
        cout << str;  delete[] str;         // друкуємо число і звільняємо пам'ять
        if (pos == -1) return;              // рядок формату закінчився числом
        // опрацювання залишків тексту
        if (pos > 0) format += pos;
        while (*format != '\0')
        {
              if (*format != '%') // звичайна літера
                    cout << *format++;
              else if (*++format != '%')
                    throw std::invalid_argument("Extra format specification found");
              else                  // відсоток в тексті
                    cout << *format++;
        }
}
```

Бачимо, що значну увагу присвячено виявленню можливих помилок: специфікація має бути одна і лише одна, літеру відсоток в рядку потрібно подвоювати, дозволені лише чотири системи числення, згадані в умові задачі.

Зауважимо також, що форматований текст функція виводить частинами до потоку *cout*. Ми не будували рядок, який би містив результати форматування. Але це легко зробити. Достатньо оголосити в блоці функції змінну *string result;* і замінити кожну інструкцію виведення *cout << вираз* на інструкцію дописування до рядка *result += вираз*. Після таких змін *result* міститиме новий рядок – результат форматування заданого числа.

Функцію *FormattedPrint* можна використовувати для виведення і окремих чисел, і послідовностей, наприклад так, як у фрагменті нижче.

```
cout << " --- Форматоване виведення числа\n";
int percentage = 75;
FormattedPrint("Якісна успішність склала %d%%\n", percentage);
cout << " --- Форматоване виведення послідовності\n";
int a[] = { 1, 2, 4, 8, 16, 32, 64, 128, 256 };
cout << "Степені 2 {";
for (int i = 0; i < 9; ++i) FormattedPrint("%H5", a[i]);
cout << " }(16)\n";
```

Наведений приклад демонструє, що навіть не дуже складна обробка тексту вимагає значних зусиль. Більшу частину з них затрачено на перевірку правильності вхідного рядка.

8.4. Відшукання найдовшого слова

Досить типовим є завдання розбиття довгого рядка літер на складові частини і опрацювання частин. Наприклад, текст розбивають на слова, запис формули – на операнди і знаки операцій.

Задача 40. *Словом називають групу відмінних від пропуску літер, записаних поруч. Тестом називають рядок, що містить відокремлені одним або декількома пропусками слова (довжина слів, загалом, довільна).*

Вивести слово найбільшої довжини. Якщо таких слів декілька, вивести перше з них.

Ми вже знаємо, що для опрацювання рядків мовою C++ можна використовувати різні структури даних: рядки-масиви і рядки-контейнери. Покажемо, як розв'язати задачу за допомогою кожної з них.

Для використання рядка в стилі C доведеться зробити припущення про довжину заданого тексту, щоб зарезервувати пам'ять для рядка. Зазначимо, що обмеження довжини не принципове для побудови алгоритму. Першу його версію можна сформулювати так: «розділити заданий рядок на слова, обчислити довжину кожного з них, знайти найбільшу». Шукати найбільше значення в послідовності чисел ми добре вміємо, тому версія здається правдоподібною. «Розділити на слова» означає знайти групи відмінних від пропуску символів, або знайти пропуски: все, що стоїть між ними, є словами. Але чи потрібно знаходити *всі* слова відразу? Їх довелося б складати в якийсь масив, розмір якого спочатку невідомий. Доведеться знову робити припущення про максимальну кількість слів, резервувати пам'ять. Це складно, а для відшукання найбільшого вистачило б отримувати слова по одному і запам'ятовувати розташування в тексті поточного кандидата на максимум.

За умовою задачі текст складається з груп відмінних від пропуску символів – слів – і груп пропусків (або одного) – розділювачів між словами. Тому, щоб виокремити перше слово тексту, достатньо переглянути заданий рядок від початку і знайти перший пропуск. Щоб знайти початок наступного слова – продовжити перегляд рядка і знайти перший символ не пропуск. Щоб виокремити друге слово – знову знайти пропуск. І так аж до закінчення тексту. Цікаво, що для розв'язання задачі вистачить одного циклу, а його параметр – індекс рядка – допоможе перебрати і слова, і розділювачі між ними.

```
void LongestByArray()
{
	const int text_size = 256;
	char text[text_size] = { 0 }; // місце для майбутнього тексту
	cout << "Введіть текст (до 255 літер): ";
	cin.getline(text, text_size);
	if (cin.failbit > 0)
	{
		// текст завеликий очищаємо потік від залишку рядка
		cin.clear();
		while (cin.get() != '\n') continue;
	}
	cout << "\nВи ввели: " << text << '\n';
	// політерне опрацювання
	int start = 0;			// початок найдовшого слова
	int max_length = 0;		// його довжина
	int curr = 0;			// початок тексту
	while (text[curr] != '\0') // опрацювання до термінального символа
	{
		int pos = curr;		// початок чергового слова
		// шукаємо його закінчення
		while (text[curr] != '\0' && text[curr] != ' ') ++curr;
		int length = curr - pos;  // довжина чергового слова
		if (length > max_length)
		{					// знайшли довше
			max_length = length;
			start = pos;
		}
		// шукаємо наступне слово
		while (text[curr] != '\0' && text[curr] == ' ') ++curr;
	}
	// розмір рядка підлаштуємо під найдовше слово
```

```
	char* longestWord = new char[max_length + 1];
	// скопіюємо найдовше слово з тексту
	for (int i = 0; i < max_length; ++i)
		longestWord[i] = text[start + i];
	// і додамо ознаку закінчення рядка
	longestWord[max_length] = '\0';
	cout << "Найдовше слово: " << longestWord << '\n';
	delete[] longestWord; // динамічну пам'ять треба звільнити
}
```

Це досить швидка процедура, оскільки всю роботу виконує за один перегляд тексту. У деталях видно весь процес посимвольного аналізу. Але, щоб її написати довелося витратити чимало часу. Процес розробки буває швидшим, якщо використовувати засоби вищого рівня абстракції, ніж звичайний масив. Завантажимо текст у рядок типу *string* і огорнемо його потоком введення. Головна перевага такого підходу – можливість використати оператор введення, який автоматично розпізнає пропуски.

```
void LongestByString()
{
	string text;
	cout << "Введіть текст: ";
	getline(cin, text);
	cout << "\nВи ввели: " << text << '\n';
	// опрацювання за допомогою потоку літер
	std::istringstream stream(text);
	string word, max_word;
	int max_length = 0;
	// введення буде успішним аж до кінця тексту
	while (stream >> word) // оператор введення розпізнає пропуски
	{
		int length = word.length();
		if (length > max_length)
		{
			max_length = length;
			max_word = word;
		}
	}
	cout << "Найдовше слово: " << max_word << '\n';
}
```

Вийшло набагато коротше і, загалом зрозуміліше. Чи так само швидко працює і ця процедура, пропонуємо перевірити читачеві.