Лекція 4. Оголошення та використання масивів мовою С++

Модифікатор const, іменовані константи, тип перелік.
Загальна структура оголошення імені, typedef, using.
Структуровані типи даних: масиви, рядки.
Використання масивів для зберігання/побудови послідовності значень.
Введення-виведення рядків. Функції для роботи з рядками (cstring).

Зберігання послідовності значень.

Обговорення матеріалу нинішньої лекції розпочнемо з задачі.

Приклад 1. Задано дійсні числа a_1, a_2, ..., a_{100}. Обчислити їхнє середнє арифметичне $\bar{a} = \frac{1}{100}\sum_{i=1}^{100} a_i$ і дисперсію $D = \frac{1}{100}\sum_{i=1}^{100}\left(a_i - \bar{a}\right)^2$.

Щось схоже ми вже робили, тому, здається, нічого складного. Особливо для обчислення середнього: щоб обчислити суму, використаємо цикл з параметром *i*, що набуває значень від 1 до 100 (або від 0 до 99); суму, як це вже було раніше, накопичуватимемо у змінній *S*, а змінну *a* використаємо для поступового введення заданих чисел. Сказане легко зобразити програмою:

```
// Приклад 1. Обчислення середнього арифметичного

	cout << "Input 100 numbers one by one: ";
	double S = 0.0;		// сума, згодом - середнє
	for (int i = 0; i < 100; ++i)
	{
		double a;	// чергове задане число
		cin >> a;	// прочитати з клавіатури
		S += a;
	}
	S /= 100;		// обчислюємо середнє арифметичне
	cout << "average = " << S << endl;
```

Зверніть увагу на використаний спосіб покрокового введення заданої послідовності чисел: їх по черзі, по одному зчитує інструкція «*cin >> a;*» у змінну *a*. У наступному рядку програми прочитане число потрапляє до суми, а на наступному кроці циклу на його місце інструкція введення зчитує наступне число.

Тепер треба скласти закінчення алгоритму. Усі задані числа введено, значення $\bar{a}$ обчислено, отже, використаємо ще один цикл і змінну *D* для накопичення суми квадратів. Закінчення алгоритму зображено нижче:

```
// Приклад 1 (продовження). Спроба обчислення дисперсії

	double D = 0.0;		// сума квадратів, згодом - дисперсія
	for (int i = 0; i < 100; ++i)
	{
		D += pow(a - S, 2);
	}
	D /= 100;		// обчислюємо дисперсію
	cout << "variance = " << D << endl;
```

Закінчення виглядає досить правдоподібно, але проаналізуймо його уважніше. Не важко переконатися, що маємо проблему в тілі циклу: в обчисленнях бере участь те саме значення змінної *a*, адже ніде в циклі воно не змінюється і не оновлюється, а нам треба врахувати всі задані числа. Звідки ця проблема? Річ у тім, що числа *були* введені, використані і витерті, а в змінній *a* залишилось значення тільки останнього.

Як же виправити ситуацію? Можна спробувати перенести обчислення *D* в попередній цикл, адже там є введення *a*, та – знову халепа: значення середнього арифметичного стає відомо тільки після закінчення цього циклу. Може вставити перед інструкцією «*D += pow(a – S, 2);*» блок введення (як у попередньому циклі)? Та чи захоче користувач алгоритму знову вводити ті самі 100 чисел, які він уже ввів? Сумнівно. Краще б наш алгоритм не забував введені значення, а якось їх запам'ятовував. Для зберігання значень всякий алгоритм використовує змінні. Наш також, адже зберігає він значення a_{100} у змінній *a*. Щоб запам'ятати решту чисел, нам треба було б ще 99 змінних. Проблему вибору такої великої кількості імен легко вирішити за допомогою цифр: *a*1, *a*2, ..., *a*100. Як автоматизувати перебір таких імен, наприклад, в циклі? На жаль, для компілятора імена *a*1, *a*55, *ai* не мають нічого спільного!

Отож, доходимо висновку, що для ефективного розв'язання задачі нам бракує або знань, або «інструментів». До тепер ми використовували в алгоритмах *прості змінні*. Вони дають змогу зберігати окремі значення. Для зберігання сукупності значень (наприклад, числової послідовності, як у цій задачі) використовують спеціальні *структуровані змінні*, або структури даних.

Структура даних складається з декількох елементів (неподільних значень або простіших структур) і має власне ім'я, єдине для цілої структури. Для неї визначено набір операцій і спосіб доступу до елементів даних.

Найпоширенішою, найбільш уживаною структурою даних є масив – сукупність фіксованої кількості однотипних елементів, кожен з яких має власний номер. Щоб звернутися до елемента, зазначають ім'я масиву і його номер, наприклад, *a*[1], *B*[5], *a*[*i*] тощо. Масиви зручно опрацьовувати за допомогою арифметичних циклів: кількість елементів масиву відома і незмінна, тому достатньо розташувати в тілі циклу з параметром *i* опрацювання елемента *a*[*i*].

Для розв'язання нашої задачі використаємо масив ста елементів: він зберігатиме всі числа заданої послідовності, щоб можна було ввести їх один раз і звертатися до них стільки разів, скільки буде треба.

```
// Приклад 1. Обчислення середнього арифметичного та дисперсії

	double a[100];		// масив для зберігання заданих чисел

	// уведення заданих величин
	cout << "Input 100 numbers one by one: ";
	for (int i = 0; i < 100; ++i)
	{
		cin >> a[i];	// прочитати з клавіатури
	}

	// обчислення середнього арифметичного
	double S = 0.0;
	for (int i = 0; i < 100; ++i)
	{
		S += a[i];
	}
	S /= 100;

	// обчислення дисперсії
	double D = 0.0;
	for (int i = 0; i < 100; ++i)
	{
		D += pow(a[i] – S, 2);
	}
	D /= 100;

	// виведення отриманих результатів
	cout << "average = " << S << "\nvariance = " << D << endl;
```

Записана вище програма починається з оголошення масиву, або резервування пам'яті для масиву. Перший цикл зчитує вхідні дані в різні елементи масиву. Дані в масиві зберігатимуться аж до закінчення роботи програми: тепер до них можна звертатися і в циклі для обчислення середнього 6, і в циклі дисперсії (у разі потреби, і в інших місцях програми).

Оголошення іменованих констант

Досить часто в умовах задач трапляються особливі значення. Наприклад, «задано десять дійсних чисел, обчисліть їхнє середнє арифметичне, визначте, скільки з них менші від середнього». Тут особливим буде число десять, оскільки траплятиметься в багатьох місцях програми: в декількох циклах, в оголошенні пам'яті для зберігання десяти чисел, в обчисленнях. У попередньому прикладі «особливим» є число 100. Такі особливі значення доречно позначати символьними іменами.

Використання іменованих констант підвищує читабельність коду, полегшує його супровід і модифікацію.

Для оголошення іменованих значень (або символьних констант) використовують модифікатор *const*. Схематично оголошення іменованої константи виглядає так:

```
const <тип> <ім'я> = <ініціалізатор>; // ініціалізатор – обов'язковий
```

Приклади оголошення констант:

```
const int n = 10;
const int size = 25;
const char delimiter = '\t';
const double eps = 1.e-12;
```

Програма, що розв'язує щойно сформульовану задачу:

```
const int n = 10;
double number[n]; // виділення пам'яті
for (int i = 0; i < n; ++i)
   cin >> number[i]; // введення заданих величин
// обчислення середнього
double average = 0.0;
for (int i = 0; i < n; ++i) average += number[i];
average /= n;
// обчислення кількості
int count = 0;
for (int i = 0; i < n; ++i)
   if (number[i] < average) ++count;
cout << "average = " << average << "\ncount = " << count << endl;
```

Якщо обчислення потрібно буде виконати для іншої кількості заданих чисел, то достатньо буде змінити *одне* значення – значення константи *n*. Увесь інший текст програми залишиться незмінним, програмістові не потрібно буде вишукувати кожна місце, де було вказано число 10.

Поміркуйте, як зміниться програма на початку лекції, якщо на її початку використати оголошення «*const int n = 100;*»?

Для оголошення багатьох констант у одному місці можна використати *перелік*. Перелік – це тип, оголошений програмістом, що містить скінченну послідовність іменованих значень. Значення кодуються цілими числами, починаючи від 0. Схема оголошення переліку:

```
enum <nameOfType> {constant1, constant2, ...};
// constant1 == 0, constant2 == 1, ...
```

У оголошенні можна явно вказувати числовий код константи. Ім'я типу – не обов'язкове. Приклади оголошень:

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo};
enum ColorConstant {BlackOnWhite=0xF0, BlueOnWhite, GreenOnWhite, CyanOnWhite,
      RedOnWhite, MagentaOnWhite, BrownOnWhite, GreyOnWhite};
enum bits {one=1, two=2, four=4, eight=8};
enum bigstep {first/*0*/, second=100, third/*101*/};
enum {zero/*0*/, null=0, one/*1*/, numero_uno=1};
```

Якщо перелік має ім'я, то в програмі можна оголосити також змінну цього типу:

```
spectrum band = red; bits current = bits(4);
```

але такі змінні не мають широкого використання у програмах.

Загальна структура оголошення

Усі імена в програмах C++ оголошують за єдиною схемою (квадратні дужки позначають необов'язкові частини оголошення):

```
[специфікатор|модифікатор] <базовий тип> <частина оголошення> [ініціалізатор];
```

- специфікатори: *virtual*, *extern*; модифікатор: *const*;
- базовий тип – вбудований, або оголошений програмістом;
- частина оголошення складається з *імені* і, можливо, операторів оголошення;
- ініціалізатор починається знаком =, а його вигляд залежить від базового типу.

Оператори оголошення:

- префіксні
  - * – вказівник (pointer);
  - **const* – незмінний вказівник;
  - & – посилання (reference);
- суфіксні
  - [ ] – масив
  - ( ) – функція

Наприклад:

```
const char * kings[] = {"Антигон", "Селевк", "Птолемей"};
```

Тут:

- *const* – модифікатор;
- *char* – базовий тип;
- **kings*[] – частина оголошення;
- = {"Антигон", "Селевк", "Птолемей"} – ініціалізатор масиву.

Оголошення читають справа наліво: масив *kings* вказівників на літери (на незмінні ланцюжки літер).

Суфіксні оператори оголошення мають вищий пріоритет, ніж префіксні. Вони «сильніше прив'язані до імені», .

Створення синонімів типу

Якщо перед оголошенням імені вказано ключове слово *typedef*, то оголошення визначає не нову змінну чи константу, а синонім типу. Наприклад:

```
const int n = 100;
typedef int NumType;
typedef double Array[n];
```

Тут *NumType* – синонім стандартного типу *int*. Всюди в програмі, де ви напишете *NumType*, стоятиме (за лаштунками) *int*. Використання такого синоніма подібне до використання іменованої константи: варто в оголошенні *typedef* замінити *int* на *double*, і заміна одразу вплине на всю програму.

Синонім *Array* має додаткове значення: він спрощує написання складного типу – одним словом замість базового типу і операторів оголошення. Порівняйте:

```
double a[100], b[100];   /* і */  Array a, b; /* як краще? */
```

Сучасний стандарт мови заохочує використовувати директиву *using* для створення синонімів імен типів:

```
using NumType = int;
using Array = double [100];
```

Результат – такий самий, як і в попередньому випадку, але *using* краще працює з узагальненими типами (про узагальнені – нам ще трохи зарано).

Масиви

У двох попередніх програмах ми вже використали числові масиви. Тепер викладемо усі правила їхнього використання. З масивів ми розпочинаємо вивчення структурованих типів, тому пригадаємо спочатку деякі важливі поняття, вивчені раніше.

Що таке тип? З точки зору мови програмування тип – це деяка множина значень, які можна присвоювати змінним і над якими можна виконувати певний набір операцій.

Що таке простий тип? Це множина атомарних (неподільних) значень. Так простими є всі числові типи, логічний, літерний, перелічувані типи. Пригадуєте, тип *bool* – це множина всього двох значень: *true* і *false*, їх можна порівнювати, застосовувати до них логічні операції (!, &&, ||); тип *short int* – множина цілих від –32 768 до 32 767, операції – порівняння, арифметичні, побітові. Перелік прикладів можна продовжити.

Що таке структурований тип? У програмуванні це *правило* побудови нового, складнішого значення з декількох простіших (їх називають базовими для нового типу): у структуроване можна об'єднати декілька атомарних значень, або структуроване об'єднати з іншим структурованим чи з декількома простими чи структурованими – синтаксис мови не накладає майже ніяких обмежень щодо використання базових типів як «будівельного матеріалу» щодо створення нових типів. Програміст завжди має змогу побудувати таку структуру даних, яка найкраще підходить для потреб задачі. Точніше кажучи, програміст повідомляє компіляторові, значення яких типів і в якій кількості він хоче об'єднати і як це все буде називатися.

Щоб порозумітися з компілятором, необхідно вивчити всі способи об'єднання простих значень у складні: таких способів є декілька (масиви, рядки, комбіновані типи або структури, класи). Найчастіше у структуру даних об'єднують фіксовану кількість однотипних елементів. У C++ таке об'єднання називають *типом масив*. Змінні цього типу реалізують структуру даних *масив*.

Масив – структура даних, послідовність фіксованого розміру однотипних елементів, що зберігаються в суміжних нумерованих комірках пам'яті, причому доступитися можна до будь-якого елемента за його номером (індексом).

Оголошення одновимірного масиву

Схема оголошення:

```
ім'я_типу_елемента ім'я_масиву [розмір];
```

Тут *розмір* – константа, або константний вираз, значення якого може визначити компілятор. Оголошений масив можна відразу ініціалізувати:

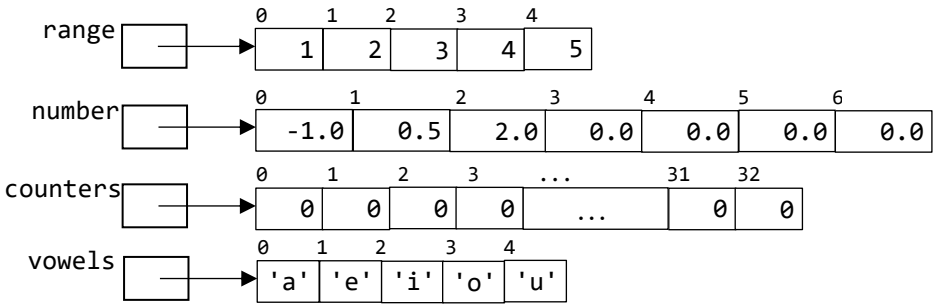
```
int range[5] = {1, 2, 3, 4, 5};
const int n = 7;
double number[n] = {-1.0, 0.5, 2.0}; // -1., .5, 2., 0., 0., 0., 0.
long counters[33] = { 0L }; // всі елементи == 0
char vowels[] = {'a', 'e', 'i', 'o', 'u'}; // розмір визначить компілятор == 5
```

Індексація елементів масиву починається з нуля: *range*[0]==1, *range*[1]==2, …, *vowels*[4]=='u'. Індекс елемента можна задавати виразом, наприклад, *counters*[5 * *i* – 1].

Якщо розмір масиву задає ініціалізатор, то програмно розмір можна визначити за допомогою оператора *sizeof*:

```
int vowels_size = sizeof vowels / sizeof vowels[0];
```

Структури в пам'яті комп'ютера після виконання записаних вище оголошень матимуть вигляд:



Оголошення багатовимірного масиву

```
base_type name_of_array[dim1][dim2]…[dimN];
```

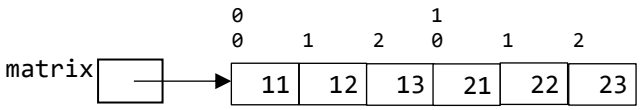
Наприклад, оголошення (та ініціалізація) двохвимірного масиву:

```
int matrix[2][3] = {{11, 12, 13},
                    {21, 22, 23}};
```

Оголошення тривимірного масиву:

```
double cube[5][5][5];
```

Скільки б не було вимірів у масиву, він завжди займає неперервну лінійну ділянку пам'яті. Кількість елементів легко обчислити як добуток розмірів у кожному з вимірів. Масив *matrix* має шість елементів, розташованих за рядками матриці: 11, 12, 13, 21, 22, 23.



Масив *cube* має 125 елементів, їхні значення невідомі.

Доступ до елементів багатовимірного масиву: *matrix*[0][1] == 12, *cube*[*i*][*j*][*k*], …

Належність індекса елемента до правильного діапазону повинен контролювати програміст. Використання неправильних індексів компілятором не контролюється і є невичерпним джерелом помилок у програмах.

Введення масиву

Масиви вводять поелементно:

```
const int n = 10;
double number[n]; // виділення пам'яті
cout << "Input " << n << " real numbers: ";
for (int i = 0; i < n; ++i)
   cin >> number[i]; // введення заданих величин
```

або так:

```
cout << "Input up to " << n << " real numbers (or 'stop'): ";
int i = 0;
while (i < n && cin >> number[i]) ++i;
cout << i << " numbers entered\n";
// перевірити стан cin і очистити за потреби
```

Виведення масиву

Масиви виводять поелементно. Одновимірний:

```
const int n = 10;
double number[n];
 // обчислення, перетворення number
for (int i = 0; i < n; ++i)
   cout << '\t' << number[i]; // виведення елементів у рядок
cout << '\n';                 // завершення рядка виведення
```

Двохвимірний:

```
const int n = 2;
const int m = 3;
int matrix[n][m];
// обчислення
for (int i = 0; i < n; ++i)
{
   for (int j = 0; j < m; ++j)
      cout << '\t' << matrix[i][j]; // виведення рядка матриці у рядок на екрані
   cout << '\n'; //закінчився рядок матриці --> перехід на новий рядок на екрані
}
```

Обчислення з масивами.

Приклад 2. Задано 10-ти елементні масиви a, b, c дійсних чисел. Обчисліть величину $u = \begin{cases} (a,c), \text{ якщо } (a,a) > 5, \\ (b,c) \text{ у іншому випадку.} \end{cases}$ Нагадаємо, що $(a,b) = \sum_{i=0}^{9} a_i b_i$.

```
const int n = 10;
// оголошення масивів
using Array = double [n];
Array a = {0}, b = {0}, c = {0};
// введення заданих векторів
cout << "Input array a: ";
for (int i = 0; i < n; ++i) cin >> a[i];
cout << "Input array b: ";
for (int i = 0; i < n; ++i) cin >> b[i];
cout << "Input array c: ";
for (int i = 0; i < n; ++i) cin >> c[i];
// обчислення
double s = 0;
for (int i = 0; i < n; ++i) s += a[i] * a[i];
double u = 0;
if (s > 5)
   for (int i = 0; i < n; ++i) u += a[i] * c[i];
else
   for (int i = 0; i < n; ++i) u += b[i] * c[i];
// виведення знайденої величини
cout << "U = " << u << endl;
```

Приклад 3. Відомо, що матриця $A = \begin{pmatrix} 5 & 3 & 1 & 0 \\ -3 & 4 & 2 & -1 \\ 1 & 3 & -5 & 2 \\ 0 & 0 & -2 & 6 \end{pmatrix}$.

Обчисліть $y = Ax$, де x – заданий вектор з чотирьох елементів.

```
const int n = 4;
// оголошення масивів
using Vector = int [n];
using Matrix = int [n][n];
Vector x, y;
```

```
Matrix A = {{5,3,1,0},{-3,4,2,-1},{1,3,-5,2},{0,0,-2,6}};
// введення заданого вектора
cout << "Input array x of " << n << " elements: ";
for (int i = 0; i < n; ++i) cin >> x[i];
// обчислення
for (int i = 0; i < n; ++i)
{
   double s; s = 0;
   for (int j = 0; j < n; ++j) s += A[i][j] * x[j];
   y[i] = s;
}
// виведення обчисленого вектора
cout << "The calculated array is:\n";
for (int i = 0; i < n; ++i)
   cout << '\t' << y[i]; // виведення елементів у рядок
cout << '\n';            // завершення рядка виведення
```

Приклад 4. Дано послідовність малих латинських літер, що закінчується крапкою. Літери можуть повторюватись. Підрахувати, скільки раз в послідовності зустрічається кожна літера.
Пояснення. Було б зовсім легко підрахувати, наприклад, кількість літер *'a'*: переглянули б послідовно кожну літеру, порівняли її з *'a'* і збільшили б на 1 змінну-лічильник при співпадінні. Але тут потрібно 26 лічильників! А як знати, котрий з них збільшувати? Використати цілу купу умовних операторів чи один великий оператор варіанту? Але все можна зробити набагато простіше: використаймо масив лічильників, індекси якого пов'язані з літерами: нехай самі літери вибирають відповідну змінну-лічильник!

```
const char first_letter = 'a';
const int ABC_size = 26;
// оголошення масиву лічильників
unsigned counters[ABC_size] = {0};
char current_letter;
cout << "Input a dot terminated line of 'a..z': ";
cin >> current_letter; // прочитали першу літеру
while (current_letter != '.')
{
   if (current_letter < 'a' || current_letter > 'z')
   {
      cout << "ERROR: A wrong char was entered!\n";
      break;
   }
   // знайшли відповідний лічильник
   ++counters[current_letter - first_letter];
   cin >> current_letter; // прочитали наступну літеру
}
// опрацювання завершене – друкуємо результати
for (int i = 0; i < ABC_size; ++i)
   cout << char(first_letter + i) << " : " << counters [i] << endl;
```

Рядки

Рядок – послідовність літер (ланцюжок), записаних у послідовні байти пам'яті. Використовується повсякчас для виведення програмою текстових повідомлень і для програмного опрацювання текстів.

Рядок – невичерпна тема для різних способів реалізації, та суперечок, який спосіб ефективніший. У програмах мовою C++ можна знайти з десяток варіантів. Найдавніший спосіб – це подання рядка за допомогою масиву літер, що закінчується термінальним (кінцевим) елементом – літерою '\0'. Такі рядки називають «рядок в стилі C». Більш сучасний варіант – використання контейнера *string*, про який поговоримо згодом, а поки що трохи уваги до класики.

Оголошення

```
char dog[5] = { 'b','e','a','u','x' }; // масив, не рядок
char cat[5] = { 'f','a','t','s','\0' };// рядок
// Рядки можна задавати компактніше
char bird[12] = "Mr. Cheep"; // bird[9]=bird[10]=bird[11]='\0'
char fish[] = "Bubbles"; // компілятор сам порахує розмір рядка
```

Уведення

```
char str[20] = { '\0' };
cin >> str; // читання до першого "порожнього" символа
// таке введення - ризикована дія, бо розмір прочитаного невідомий
```

Оператор >> автоматично дописує в кінець рядка термінальний символ, але ніяк не контролює розмір масиву, який хочуть заповнити значеннями з потоку введення. Тому завжди є ризик виходу за межі зарезервованої пам'яті. Безпечніші методи введення рядків:

```
cin.get(str, 20); // прочитає не більше 19 символів до першого [Enter]
cin.getline(str, 20); // прочитає рядок і забере з потоку [Enter]
```

Виведення

Можна виконати звичайним оператором <<. Головне, щоб рядок *str* закінчувався літерою '\0', інакше після очікуваного тексту на екрані можна буде побачити ще багато чого.

```
cout << str;
```

Перетворення рядків

Рядок в стилі С – масив літер, тому його можна опрацьовувати поелементно. Кожен рядок містить власну ознаку закінчення – термінальний символ, тому зазвичай їх опрацьовують за допомогою ітераційних циклів з передумовою.

Приклад 5. Замініть у заданому рядку всі малі латинські літери «а» на великі.

```
const int n = 20;
char str[n] = { '\0' };
cout << "Input a string: ";
cin.getline(str, n);
int i = 0;
while (str[i] != '\0') // можна і так: while (str[i])
{
	if (str[i] == 'a') str[i] = 'A';
	++i;
}
cout << str << endl;
```

Для опрацювання С-рядків використовують ресурси модуля *cstring*. Тут оголошено низку стандартних функцій для роботи з рядками. Найуживаніші з них:

- *size_t strlen(const char *str)*; повертає довжину рядка
- *char *strcpy(char *strDestination, const char *strSource)*; копіює з strSource в strDestination
- *int strcmp(const char *string1, const char *string2)*; порівнює два рядки
- *char *strcat(char *strDestination, const char *strSource)*; дописує strSource до strDestination
- *char *strchr(char * str, int c)*; знаходить літеру c в рядку str

та ще багато інших. Докладніше про них можна довідатися тут: https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/string-manipulation-crt?view=vs-2019

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Дудзяний І.М. Програмування мовою С++.
4. Брюс Эккель, Чак Эллисон Философия С++.
5. Бьерн Страуструп Язык программирования С++.
6. Герберт Шилдт С++ Базовый курс.