Антонюк В.А., Задорожный С.С.

Язык программирования C/C++, часто встречающиеся ошибки при написании программ

Учебно-методическое пособие по дисциплине
«Программирование и информатика»

Москва
Физический факультет МГУ имени М.В. Ломоносова
2021

Антонюк Валерий Алексеевич, Задорожный Сергей Сергеевич

Язык программирования C/C++, часто встречающиеся ошибки при написании программ. –
М. : Физический факультет МГУ им. М. В. Ломоносова, 2021. – 60 с.

При написании программ на языке C++ студенты затрачивают много времени на исправление ошибок компиляции и отладку, поскольку не могут понять выдаваемые в процессе компиляции и отладки сообщения. Целью данной книги является разъяснение этих сообщений и способов устранения ошибок.
Приведены образцы программ для наиболее трудно воспринимаемых тем – использование массивов, матриц и создание собственных классов.
В пособии также приведен обширный список литературы с аннотациями, который поможет выбрать наиболее подходящее издание в соответствии с пожеланиями и уровнем подготовки.
Пособие поможет студентам более самостоятельно и оперативно решать задачи практикума по программированию.
Рассчитано на студентов первого и второго курсов физического факультета, но может быть полезно студентам старших курсов, аспирантам и сотрудникам, занимающимися разработкой программного обеспечения на языке C++.

Авторы – сотрудники кафедры математического моделирования и информатики физического факультета МГУ.

Рецензенты: доцент кафедры ОФиВП Шлёнов Святослав Александрович, ведущий электроник кафедры ОФиВП Лукашев А.А.

Подписано в печать 23.08.2021. Объем 4 п.л. Тираж 30 экз. Заказ No .

Физический факультет им. М. В. Ломоносова,
119991 Москва, ГСП-1, Ленинские горы, д. 1, стр. 2.
Отпечатано в отделе оперативной печати физического факультета МГУ.

Оглавление

1. Введение

При написании программ на языке C/C++ студенты затрачивают много времени на исправление ошибок компиляции и отладку, поскольку не могут понять выдаваемые в процессе компиляции и отладки сообщения. Целью данной книги является разъяснение этих сообщений и способов устранения ошибок.

Приведены образцы программ для наиболее трудно воспринимаемых тем – использование массивов, матриц и создание собственных классов.

В пособии также приведен обширный список литературы с аннотациями, который поможет выбрать наиболее подходящее издание в соответствии с пожеланиями и уровнем подготовки.

Надеемся, что пособие поможет студентам более самостоятельно и оперативно решать задачи практикума по программированию.

2. Замечания по синтаксису языка C/C++

Следует помнить, что программа на C/C++ состоит из модулей (функций) [1] и, кроме того, может содержать объявления глобальных переменных, которые доступны во всех функциях, расположенных после объявления этих переменных. Создавайте такие переменные только в том случае, когда это действительно полезно. Например, массив большого размера может не поместиться в стеке, если он объявлен локально. Константы, используемые в нескольких модулях, тоже можно объявить глобально. Давайте глобальным переменным осмысленные названия, соответствующие их назначению в программе (например, не *N*, а *MAXSIZE*).

Для передачи данных в функцию надо использовать параметры функции. Тип каждого фактического параметра (константы или переменной) в инструкции вызова функции должен совпадать с типом соответствующего формального параметра, указанного в объявлении функции. Если параметр функции используется для возврата результата, то в объявлении функции этот параметр должен быть указателем или ссылкой.

Объявления локальных переменных можно помещать в начале функции, сразу за заголовком, снабжая их кратким комментарием о назначении этих переменных. Однако язык C++ допускает объявление переменных практически в любом месте функции, поэтому имеет смысл объявлять их по мере необходимости. Это улучшает восприятие программы и уменьшает количество ошибок, связанных с неправильным написанием.

Прописные и строчные буквы различаются, поэтому, например, имена *n* и *N* обозначают разные переменные. Кроме того, компилятор Visual Studio допускает использование в именах кириллицы. Поэтому, следите за тем, в какой раскладке Вы вводите символы, т.к. одинаковые по начертанию буквы латиницы и кириллицы компилятор считает разными.

Инструкции присваивания предназначены для изменения значений переменных. Инструкция присваивания, выполняющая некоторое действие, может быть записана несколькими способами, например, вместо $x=x+dx$ можно записать $x+=dx$, а вместо $i=i+1$ можно воспользоваться оператором инкремента и записать $i++$.

Значение выражения в правой части инструкции присваивания зависит от типа операндов и операции, выполняемой над операндами. Целочисленные операции выполняются без учета переполнения. Т.е., например, при сложении двух достаточно больших целых чисел может получится отрицательный результат.

Результатом выполнения операции деления над целыми операндами является целое частное, а для получения остатка используется операция %.

[1] Примеры правильно построенных программ см. в разделе 3

Массив представляет собой набор, совокупность элементов одного типа. В инструкции объявления массива указывается количество элементов массива. Это значение является константой, которая должна быть известна на этапе компиляции. Элементы массива нумеруются с нуля, доступ к элементу массива осуществляется путем указания индекса (номера) элемента. В качестве индекса можно использовать выражение целого типа. Индекс может меняться от 0 до *n*-1, где *n* – количество элементов массива. К элементам массива можно также обращаться по их адресам. Компилятор не проверяет правильность адресации, поэтому контроль за их правильностью полностью возлагается на программиста.

Строка в языке С – это массив символов. Последним символом строки обязательно должен быть нуль-символ, код которого равен 0, и который изображается как '\0'. Константные строки можно использовать задавая их адрес: *const char* HelloTxt=”Hello”;* Строки нельзя сравнивать по их имени, т.к. это будет сравнение адресов. Необходимо использовать функцию сравнения строк, например, *strcmp*. При вводе русского текста с консоли необходимо учитывать, что по умолчанию используется кодовая страница DOS (866), для перевода в кодировку Windows (1251) можно использовать функцию *OemToChar*, определенную в заголовочном файле windows.h.

Следует внимательно относиться к приоритету операций. Особенно это важно при использовании перегруженных операций, т.к. их приоритет остается прежним. Ниже приведена таблица приоритетов. В первой колонке указаны приоритет и ассоциативность (порядок обработки) соответствующих операций (слева-направо L -> R или справа-налево R -> L). Серым цветом выделены операции языка C++.

Ассоциа-тивность	**Оператор**	**Описание**	**Пример**
1. Нет	::	Глобальная область видимости (унарный)	::name
	::	Область видимости класса (бинарный)	class_name::member_name
2. L -> R	( )	Круглые скобки	(expression)
	( )	Вызов функции	function_name(parameters)
	( )	Инициализация	type name(expression)
	{}	uniform инициализация (C++11)	type name{expression}
	type()	Конвертация типа	new_type(expression)
	type{}	Конвертация типа (C++11)	new_type{expression}
	[]	Индекс массива	pointer[expression]
	.	Доступ к члену объекта	object.member_name
	->	Доступ к члену объекта через указатель	object_pointer->member_name
	++	Пост-инкремент	lvalue++
	—	Пост-декремент	lvalue—
	typeid	Информация о типе во время выполнения	typeid(type) or typeid(expression)
	const_cast	Приведение константных типов	const_cast(expression)

	dynamic_cast	Приведение типа во время выполнения	dynamic_cast(expression)
	reinterpret_cast	Конвертация одного типа в другой	reinterpret_cast(expression)
	static_cast	Проверка типа во время компиляции	static_cast(expression)
3. R -> L	+	Унарный плюс	+expression
	—	Унарный минус	-expression
	++	Пре-инкремент	++lvalue
	—	Пре-декремент	—lvalue
	!	Логическое НЕ (NOT)	!expression
	~	Побитовое НЕ (NOT)	~expression
	(type)	Приведение типа в стиле языка C	(new_type)expression
	sizeof	Размер в байтах	sizeof(type) or sizeof(expression)
	&	Адрес	&lvalue
	*	Разыменование указателя	*expression
	new	Динамическое выделение памяти	new type
	new[]	Динамическое выделение массива	new type[expression]
	delete	Динамическое удаление памяти	delete pointer
	delete[]	Динамическое удаление массива	delete[] pointer
4. L -> R	->*	Обращение через указатель к члену объекта тоже заданного через указатель	object_pointer ->*pointer_to_member
	.*	Обращение к члену объекта заданного через указатель	Object .*pointer_to_member
5. L -> R	*	Умножение	expression * expression
	/	Деление	expression / expression
	%	Остаток	expression % expression
6. L -> R	+	Сложение	expression + expression
	—	Вычитание	expression — expression
7. L -> R	<<	Побитовый сдвиг влево	expression << expression
	>>	Побитовый сдвиг вправо	expression >> expression
8. L -> R	<	Сравнение: меньше чем	expression < expression
	<=	Сравнение: меньше чем или равно	expression <= expression
	>	Сравнение: больше чем	expression > expression
	>=	Сравнение: больше чем или равно	expression >= expression
9. L -> R	==	Равно	expression == expression
	!=	Не равно	expression != expression
10. L -> R	&	Побитовое И (AND)	expression & expression
11. L -> R	^	Побитовое исключающее ИЛИ (XOR)	expression ^ expression
12.	|	Побитовое ИЛИ (OR)	expression | expression

L -> R			
13. L -> R	&&	Логическое И (AND)	expression && expression
14. L -> R	||	Логическое ИЛИ (OR)	expression || expression
15. R -> L	?:	Тернарный условный оператор	expression ? expression : expression
	=	Присваивание	lvalue = expression
	*=	Умножение с присваиванием	lvalue *= expression
	/=	Деление с присваиванием	lvalue /= expression
	%=	Деление с остатком и с присваиванием	lvalue %= expression
	+=	Сложение с присваиванием	lvalue += expression
	-=	Вычитание с присваиванием	lvalue -= expression
	<<=	Присваивание с побитовым сдвигом влево	lvalue <<= expression
	>>=	Присваивание с побитовым сдвигом вправо	lvalue >>= expression
	&=	Присваивание с побитовой операцией И (AND)	lvalue &= expression
	|=	Присваивание с побитовой операцией ИЛИ (OR)	lvalue |= expression
	^=	Присваивание с побитовой операцией «исключающее ИЛИ» (XOR)	lvalue ^= expression
16. R -> L	throw	Генерация исключения	throw expression
17. L -> R	,	Оператор Запятая	expression, expression

3. Образцы (шаблоны) программ для типовых задач

Ниже приведено несколько образцов корректно построенных программ

- *Образец 1 (программа с функцией)*

Этот образец является примером программы, содержащей дополнительную функцию, которая вычисляет значение библиотечной функции $y=e^x$ разложением в ряд Тейлора. Обратите внимание, что очередной член ряда вычисляется рекуррентно через предыдущее значение.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
// функция вычисления exp(xp) разложением в ряд Тейлора
double MyExp(double xp, double e)
{
	double sum = 0.; // переменная для накопления суммы
	double an = 1.0; // переменная для хранения n-го члена ряда
	int n = 0;       // номер очередного члена ряда
	while (fabs(an) > e) // цикл закончится когда an станет достаточно малым
```

```
	{
		sum += an; // суммируем n-й член ряда
		n++;       // переходим к следующему члену ряда
		an *= xp / n; // вычисляем a(n) через a(n-1)
	}
	return sum;
}
//---------точка входа в консольную программу-----------------------------
int main()
{
	double x, xend, step,eps;
	printf("xbegin=");
	scanf("%lf", &x); // вводим начальное значение x
	printf("xend=");
	scanf("%lf", &xend); // вводим конечное значение x
	printf("step=");
	scanf("%lf", &step);  // вводим шаг
	printf("eps=");
	scanf("%lf", &eps);  // вводим точность
	getchar();   // убираем из потока ввода оставшийся там символ конца строки
	puts("   x     MyExp(x)  exp(x)"); // заголовок таблицы
	for (  ; x < xend + step / 10 ; x += step) // цикл по x с шагом step
	{
		double y1 = MyExp(x, eps); // вызов нашей функции
		double y2 = exp(x);        // вызов библиотечной функции
		printf("%6.2lf %8.4lf %8.4lf\n", x, y1, y2);
	}
	getchar();  // Ждем нажатия Enter
	return 0;
}
```

- *Образец 2 (одномерный массив)*

Образец иллюстрирует использование динамического массива на языке С, размер которого вводится с клавиатуры. Обработка массива выполняется в отдельных функциях в соответствии с требованиями к задаче.

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
//----- объявление прототипов используемых функций-------------------------
void Input(int* M, size_t n);  // ввод массива
void Print(int* M, size_t n);   // вывод массива
void RandomFill(int* M, size_t n);   // заполнение массива случайными числами
//--------------------------------------------------------------------------
int Work(int* M, size_t n);   // пример обработки массива
//   Сюда добавить протопипы тех функций которые дополнительно напишете
//--------------------------------------------------------------------------

//------- точка входа в программу консольного приложения --------
int main()
{
	setlocale(LC_ALL, "Rus");
	size_t n;
	puts("Размер вектора?");
	scanf_s("%d",&n);
	int* A = (int*)malloc(n*sizeof(int)); // создаем массив нужной длины
	//Input(A,n);      // если надо ввести с клавиатуры
	RandomFill(A, n);   // если надо заполнить случайными значениями
	Print(A, n);        // выводим массив
	int s = Work(A, n);   // выполняем обработку
	printf("Summa = %d\n" , s );
	free(A);        // освобождаем ресурсы
	system("pause");   // задержка закрытия окна
```

```
	return 0;
}
//----   описание функций ----------------------------------------
// ввод массива
void Input(int* M, size_t n)
{
	for (size_t i = 0; i<n; i++)
	{
		printf("M[%d]=?" , i );
		scanf_s("%d",&M[i]);
	}
}
// заполнение случайными числами
void RandomFill(int* M, size_t n)
{
	for (size_t i = 0; i<n; i++)
	{
		M[i] = rand() % 201 - 100; // получим диапазон от -100 до +100
	}
}
// вывод массива
void Print(int* M, size_t n)
{
	for (size_t i = 0; i<n; i++)
	{
		printf("%d ", M[i] );
	}
	puts("");
}
// пример обработки - подсчет суммы элементов
int Work(int* M, size_t n)
{
	int sum = 0;
	for (size_t i = 0; i<n; i++)
	{
		sum += M[i];
	}
	return sum;
}
```

- *Образец 3 (двумерный массив)*

Образец иллюстрирует использование динамического двумерного массива на языке С, в котором вводится количество строк и столбцов. Обработка массива выполняется в отдельных функциях в соответствии с требованиями к задаче.

```
#include <stdio.h>
#include <locale.h>
#include <stdlib.h>
#include <time.h>
//
// Создание матрицы
//
int ** Create(size_t n, size_t m) {
	int ** M = (int**)malloc(n * sizeof(int*));
	for (size_t i = 0; i < n; ++i) {
		M[i] = (int*)malloc(m * sizeof(int));
	}
	return M;
}
//
// Удаление матрицы
//
void Free(int ** M, size_t n) {
```

```
	for (size_t i = 0; i < n; ++i) {
		free(M[i]);
	}
	free(M);
}
//
//---- ввод матрицы---------------------------------------
//
void Input(int ** M, size_t n, size_t m) {
	for (size_t i = 0; i < n; ++i) {
		for (size_t j = 0; j < m; ++j) {
			printf("M[%d][%d]=", i, j);
			scanf_s("%d", &M[i][j]);
		}
	}
}
//
// заполнение матрицы случайными числами из диапазона [0, 99] -----------
//
void FillRandomNumbers(int **matrix, const size_t rows, const size_t columns)
{
	srand((unsigned int)time(0)); // инициализируем ПГСЧ

	for (size_t row = 0; row < rows; row++)
		for (size_t column = 0; column < columns; column++)
			matrix[row][column] = rand() % 100; // присваиваем СЧ
}
//
//-------- Печать матрицы -------------------------------------------------
//
void Print(int ** M, size_t n, size_t m) {
	for (size_t i = 0; i < n; ++i) {
		for (size_t j = 0; j < m; ++j) {
			printf("%d ", M[i][j]);
		}
		puts("");
	}
}
//
// пример обработки матрицы - подсчет сумм в каждой строке
//
void Process(int ** M, int *Sum, size_t n, size_t m) {
	for (size_t i = 0; i < n; ++i) {
		Sum[i] = 0;
		for (size_t j = 0; j < m; ++j) {
			Sum[i] += M[i][j];
		}
	}
}

// ...
// сюда вставить все дополнительные функции которые напишете
// ...

int main()
{
	setlocale(LC_ALL, "Rus"); // установление русской локали (windows)

	size_t n, m;

	// вводим размерность матрицы
	printf("Введите количество строк матрицы: ");
	scanf_s("%d", &n);
	printf("Введите количество столбцов матрицы: ");
	scanf_s("%d", &m);
```

```
	// выделяем память под матрицу
	int ** A = Create(n, m);

	// ввод матрицы
	//Input( A, n, m );
	// заполнение случайными числами (вместо ввода)
	FillRandomNumbers(A, n, m);

	// Вывод матрицы
	Print(A, n, m);

	// обработка матрицы
	int* S = (int*)malloc(n * sizeof(int)); // Вектор результата
	Process(A, S, n, m);

	// вывод результата
	puts("Result:");
	for (size_t i = 0; i < n; i++)
		printf("%d ", S[i]);
	puts("");

	// освобождаем память, выделенную под матрицу и вектор
	free(S);
	Free(A, n);

	// ждём нажатия клавиши перед выходом из приложения (windows)
	system("pause");

	return 0;
}
```

- *Образец 4 (базовый и производный классы C++)*

Образец иллюстрирует создание базового и производного класса, а также применение виртуальной функции и перегрузку потоковых операторов *operator<<* и *operator>>*

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class AA
{
private:
	// здесь данные и методы, которые используются только в базовом классе
protected:
	// здесь данные и методы, которые доступны в производном классе
	char* str; // для примера строка с динамически выделяемой памятью
public:
	virtual ~AA() { delete[] str; } // деструктор
	AA() { str = new char[1]; *str = 0; } // конструктор по умолчанию
	AA(char* x)	// конструктор с параметрами
	{
		str = new char[strlen(x) + 1];
		strcpy(str, x);
	}
	AA(const AA& a)  // копиконструктор
	{
		str = new char[strlen(a.str) + 1];
		strcpy(str, a.str);
	}
	AA& operator=(const AA& a)  // operator= (правило ТРЕХ)
```

```
	{
		if (this == &a)  // обязательно проверяем, что нет присвоения типа x=x;
			return *this;
		if (strlen(str) != strlen(a.str))
		{
			delete[] str;
			str = new char[strlen(a.str) + 1];
		}
		strcpy(str, a.str);
		return *this;
	}
	virtual void f() { cout << " class A\n"; } // виртуальная функция
	friend ostream& operator<<(ostream& os, const AA& a); // перегрузка потокового
вывода
	friend istream& operator>>(istream& os, AA& a); // перегрузка потокового ввода
};
//-------------------------------------------------------
ostream& operator<<(ostream& os, const AA& a)
{
	os << a.str << endl;
	return os;
}
istream& operator>>(istream& is, AA& a)
{
	char tmp[100];
	is.getline(tmp, 100);
	delete[] a.str;
	a.str = new char[strlen(tmp) + 1];
	strcpy(a.str, tmp);
	return is;
}
//-------------------------------------------------------

class BB : public AA
{
private:
	int SIZE; // дополнительные данные
public:
	~BB() {}
	BB() :AA() { SIZE = strlen(str); }
	BB(char* x) :AA(x) { SIZE = strlen(str); } // обязательно вызываем конструктор
базового класса
	BB(BB& a) :AA(a) { SIZE = strlen(str); } // обязательно вызываем конструктор
базового класса
	BB& operator=(const BB& a)
	{
		if (this == &a)
			return *this;
		if (SIZE != a.SIZE)
		{
			delete[] str;
			str = new char[SIZE + 1];
		}
		strcpy(str, a.str);
		return *this;
	}
	void f() { cout << " class B\n"; }
	friend ostream& operator<<(ostream& os, const BB& a);
	friend istream& operator>>(istream& os, BB& a);

};
//-------------------------------------------------------
ostream& operator<<(ostream& os, const BB& a)
{
	os << a.str << " Size=" << a.SIZE << endl;
```

```
	return os;
}
istream& operator>>(istream& is, BB& a)
{

	char tmp[100];
	is.getline(tmp, 100);
	delete[] a.str;
	a.SIZE = strlen(a.str);
	a.str = new char[a.SIZE + 1];
	strcpy(a.str, tmp);
	return is;
}
//-----------------------------------------------------
int main()
{
	char tst[] = "Hello";
	AA a(tst);  // базовый класс
	a.f();      // невиртуальный вызов f() из AA
	AA a2;
	a2 = a;  // operator=
	cout << a2;

	BB b(tst);  // производный класс
	b.f();      // невиртуальный вызов f() из BB
	BB b2 = b;    // копиконструктор эквивалентно BB b2(b);
	cout << b;

	AA* pa;
	// Указателю на базовый класс присваиваем адрес ПРОИЗВОДНОГО класса:
	pa = &b;
	pa->f();// виртуальный вызов f() из B
	pa->AA::f();// невиртуальный вызов f() из AA

			//system("pause");
	return 0;
}
```

4. Ошибки этапа компиляции (программа не компилируется или компилируется с предупреждениями)

- *Попытка модифицировать константу через указатель*

Объявим указатель, который адресует некоторый текст и попытаемся что-то в этом тексте изменить:

```
char* W = "Плоба";
W[1] = 'p'; // ошибка
```

Исправить можно, заменив указатель массивом :

```
char W[] = "Плоба";
W[1] = 'p';
```

В этом случае под *W* выделится память на длину строки+1 (в данном примере 6 байт) и в нее копируется указанный текст. Соответственно, его можно изменять.

Эта же ошибка присутствует и в таком примере:

```
char* S1 = "Hello";
gets(S1);   // ошибка
char S2[6] = "Hello";
fgets(S2, 6, stdin); // Правильно
char* S3 = (char*)malloc(255 * sizeof(char));
fgets(S3, 255, stdin); // Правильно
free(S3);
```

- ## *Неправильное понятие приведения (C++)*

Для приведения типов данных в C++ часто используются операции *static_cast* и *reinterpret_cast*

Операцию приведения *static_cast<новый_тип>(выражение)* можно использовать только в тех случаях, когда компилятор выдает предупреждающее сообщение, например:

```
double d = 1.5;
int n = static_cast<int>(d);
```

Обратите внимание, что круглые скобки вокруг выражения (d) ставятся всегда!

Если же выдается сообщение об ошибке, то нужно использовать операцию *reinterpret_cast<новый_тип>(выражение)*.

В этом случае только сам программист может вникнуть в суть преобразования и взять ответственность за преобразование на себя.

Вот пример, требующий приведение такого типа:

```
struct typ1 { int t; };
struct typ2 { int g; };

typ1 tp1 = { 1 };
typ2 * p = reinterpret_cast<typ2 *>(&tp1);
p->g = 2;
cout << typ1.t;
```

Здесь имеются два разных структурных типа *typ1*и *typ2*. К тому же и имена полей у них разные. Но внутренне устройство одинаковое. Т.е. мы понимаем, что типы данных абсолютно совместимы. Осталось доказать это компилятору, что мы и делаем с помощью явного преобразования

```
typ2 * p = reinterpret_cast<typ2 *>(&tp1);
```

- ## *Лишняя точка с запятой*

Часто при написании программы точка с запятой не задумываясь ставится в конце каждой строки:

```
int i;
for (i = 0; i < 10; ++i);  // лишняя ;
        printf("%d ", i);
```

Такой код приведет к выводу числа 10, а не последовательности чисел от 0 до 9. В данном случае просто 10 раз выполнится пустой цикл.

Такая же ошибка очень часто встречается в определении функций:

```
int cube_func(int x); // скопировали объявление, а ; убрать забыли
{
        return x * x * x;
}
// правильно:
int cube_func(int x)
{
        return x * x * x;
}
```

- *Отсутствие точки с запятой после определения классового типа*

После определения класса, структуры или объединения обязательно нужно ставить точку с запятой. Ее отсутствие приводит к выводу сообщений об ошибках, которые трудно понять:

```
struct A
{
        int n;
} // забыли ;

int main()
{
        A a;
        return 0;
}
```

Данная ошибка приводит к такому набору сообщений:

```
error C2628: недопустимый 'A' с последующим 'int' (возможно, отсутствует ';')
error C3874: возвращаемый тип 'main' должен быть 'int', а не 'A'
error C2143: синтаксическая ошибка: отсутствие ";" перед "."
error C2143: синтаксическая ошибка: отсутствие ";" перед "."
error C2664: A::A(const A &): невозможно преобразовать параметр 1 из 'int' в 'const A &'
```

- *Отсутствие возврата значения из функции*

Рассмотрим функцию:

```
int f(int x)
{
        if (x == 10)
          return 1;
}
```

В ней в случае невыполнения условия x==10 возврата из функции нет, что может привести к неопределенному поведению программы.

Правильный код:

```
int f(int x)
{
        if (x == 10)
          return 1;
        return 0;
}
```

Компилятор может выдать предупреждающее сообщение:

```
warning C4715: f: значение возвращается не при всех путях выполнения
```

Но для этого должен быть установлен соответствующий уровень предупреждений компилятора.

В Visual Studio это делается через команду меню:

```
Проект -> Свойства -> Свойства конфигурации -> C/C++ -> Общие -> Уровень
предупреждений -> Уровень 4 (/W4)
```

Реальный пример - функция поиска позиции первого вхождения символа в строку.

```
int indexOf(const char * str, const char c)
{
        for (int i = 0; str[i]; ++i)// пробегаемся по строке
            if (str[i] == c) // если текущий символ - искомый
```

```
                return i;   // то возвращаем индекс этого символа
        return -1;                  // если символ не найден возвращаем -1
}
```

- ### *Использование комментариев в #define*

Макрокоманда *#define* достаточно часто используется в языке С, однако она может быть источником ошибок, которые трудно понять. В частности, при наличии комментариев в той же строке:

```
#define VAR  5 // так делать не рекомендуется
#define VAR  5 /* так можно */
```

Препроцессор вместо текста *VAR* подставляет **весь** текст до конца строки.

Т.о. вот такое выражение:

```
int x = VAR;
```

превращается в первом случае в

```
 int x = 5 // так делать не рекомендуется ;
```

Т.е. потерялась точка с запятой(ушла в комментарии).

```
Во втором случае:
```

```
int x = 5  /* так можно */;
```

точка с запятой все-таки оказалась вне комментариев.

- ### *Компилятор не находит iostream.h (C++)*

При компиляции проекта содержащего

```
#include "iostream.h"
```

выдается сообщение

```
[Error] iostream.h: No such file or directory
```

Ошибка в том, что используется устаревший пример. Эту строку необходимо заменить на

```
#include <iostream>
using namespace std;
```

Заголовочные файлы C++ не содержат расширения *.h* и находятся в пространстве имен *std*. Старые (естественно стандартные) заголовочные файлы переписаны для *std* с добавлением буквы *c* впереди, например *#include <cmath>* вместо *#include <math.h>*.

Файл *conio.h* нестандартный, поэтому подключайте его как обычно. Но лучше его не использовать. Задержку закрытия окна можно сделать с помощью консольной команды:

```
system("pause");
```

- ### *Внутри switch ошибка «Case bypasses initialization of a local variable»*

Рассмотрим код:

```
int i = 2;
switch (i)
{
case 1:
        int j = 123;
        break;
case 2:
        cout << j << endl;
        break;
}
```

При *i* равном 2 переменной *j* не присваивается начальное значение. Соответственно, выводимое значение не определено.

Такая ошибка выдается компилятором даже в случае, когда объявленная переменная не используется в других ветвях оператора. Если действительно нужна локальная переменная, то обрамляйте соответствующий кусок кода фигурными скобками:

```
int i = 2;
switch (i)
{
 case 1:
 {
        int j = 123;
     cout << j * 2 << endl;
 }
 break;
 case 2:
 {
     int k = 456;
     cout << k * k << endl;
 }
 break;
}
```

- *Передача двумерных массивов и указателей в функцию*

Передача в функцию двумерных статических и динамически созданных массивов сильно различается.

Для статических массивов обязательно требуется указывать константу определяющую количество столбцов:

```
int f(int array[][], ...);//ошибка, не указано к-во столбцов
int f(int array[][3], int row); // правильно
int f(int array[3][3]);//Указание количества строк не является ошибкой
```

Двумерные динамические массивы создаются таким образом :
В начале создаем массив для хранения адресов строк

```
int** array = (int**)malloc(row*sizeof(int*));
```

Далее в цикле выделяем память для каждой строки:

```
for (int i = 0; i<row; i++)
        array[i] = (int*)malloc(col*sizeof(int));
```

В функцию необходимо передавать указатель на указатель, т.е.адрес массива, в котором хранятся адреса строк :

```
int f(int** array, int row, int col);
```

Также необходимо помнить об удалении выделенной памяти:

```
for (int i = 0; i<row; i++)
        free(array[i]); // удаление строк
free(array); // удаление массива указателей на строки
```

- *Не удается открыть файл stdafx.h или pch.h*

Заголовочный файл *stdafx.h (pch.h* в более поздних версиях) - это файл предварительно скомпилированных заголовков, используемый Visual Studio для ускорения компиляции. Эта опция автоматически устанавливается при создании непустого проекта, отключить ее можно командой меню

```
Проект -> Свойства -> Свойства конфигурации -> C/C++ ->
Предварительно скомпилированные заголовки
```

И выбрать «Не использовать предварительно скомпилированные заголовки».

- *Ошибка «unresolved external symbol _WinMain@16»*

При создании проекта был выбран вариант создания Windows приложения, которое требует наличие в проекте функции *WinMain*.

WinMain - это точка входа в Windows приложение, *main* – точка входа в консольное приложение.

Для устранения этой ошибки либо создавайте новый консольный проект, либо измените тип приложения командой меню

```
Проект->свойства->свойства конфигурации->компоновщик->система-> подсистема -> Консоль
```

- *Ошибка «This function or variable may be unsafe»*

При компиляции выдается ошибка C4996 вида:

```
error C4996: 'fopen': This function or variable may be unsafe.
```

Такого рода ошибка возникает при вызове функций, которые имеют более защищенные варианты в Visual Studio. Если не планируется использовать программу с другими компиляторами, то можно использовать предлагаемые варианты функций. Однако, имейте ввиду, что их список параметров может отличаться от списка параметров стандартных функций. Поэтому, прежде, чем использовать, ознакомьтесь с описанием соответствующей функции.

Для отключения контроля за этой ошибкой добавьте в начале программы строку

```
#define _CRT_SECURE_NO_WARNINGS
```

5. Ошибки этапа выполнения (программа прекращает работу с сообщением об ошибке)

- *Выделение памяти без дальнейшего освобождения или неверное освобождение (C++)*

При выделении памяти через оператор *new* не забывайте освобождать её с помощью оператора *delete* в конце программы

```
int * a = new int;
// ... использование переменной
delete a; // перед выходом из программы
```

В языке C++ не предусмотрено ее автоматическое освобождение, что может привести утечка памяти.

Используйте для освобождения памяти, выделенной под массив, оператор *delete[],* а под обычные переменные – *delete* (без [] скобок).

```
int* a = new int;     // переменная
int* b = new int[2];  // массив
// ...
// delete[] a;  // не ОК
delete a;       // ОК
// delete b;          // не ОК
delete[] b;           // ОК
```

Несмотря на то, что программа скомпилируется, попытка освобождения с помощью неверного оператора вызывает UB (неопределённое поведение), что приводит к трудноуловимым ошибкам.

Не стоит сочетать в программе, а тем более по отношению к одной и той же переменной/массиву, разные способы выделения/освобождения памяти. При выделении памяти с помощью *malloc/calloc* - освобождайте её с помощью функции *free*, при выделении через оператор *new/new[]*- освобождайте с помощью оператора *delete/delete[]*. Стоит отметить, что использование в C++ *malloc/calloc* способов работы с памятью в целом не одобряется.

- *Возврат указателя (C) или ссылки (C++) на локальную переменную*

Локальные переменные, созданные на стеке, уничтожаются при выходе из области видимости (в т.ч. и при выходе из функции), таким образом, память получает статус свободный и туда могут записаться какие-либо данные. Поэтому нельзя возвращать их по ссылке или указателю:

```
//------ C -------------------
int* foo2()
{
	int x = 2;
	return &x; //ошибка
}
int foo3()
{
	int x = 3;
	return x; //правильно, переменная копируется
}

//----------- C++ --------------
int& foo()
{
	int x = 1;
	return x; //ошибка
}
int& foo4()
{
	static int x = 0;
	return x; //правильно, переменная не уничтожается
}
int& foo5(int& x)
{
	x = 4;
	return x; //правильно, переменная не локальная
}
```

- *Использование неинициализированной переменной*

Это, похоже, одна из самых распространённых ошибок, о которой компилятор может выдать предупреждающее сообщение такого типа

```
warning C4700: использована неинициализированная локальная переменная "x"
```

Однако, этого не произойдет, если предупреждения отключены. В Visual Studio настройка уровня предупреждений выполняется командой:

```
Проект -> Свойства -> Свойства конфигурации -> C/C++ -> Общие -> Уровень предупреждений.
```

Простейший пример:

```
int x; // предполагалось, например, вводить x;
printf("%d", x);//выведет случайное значение, которое находилось в x
```

Чаще всего почему-то встречается вариант с использованием неинициализированного счётчика или переменной для подсчёта суммы, как показано ниже:

```
const int arrSize = 3, arr[arrSize] = { 1, 2, 3 };
int sum;//ложное предположение, что переменная неявно инициализируется нулём
//int sum=0; //  надо писать
for (int i = 0; i < arrSize; i++)
      sum += arr[i]; // мы к "мусору" добавляем значение элемента массива
std::cout << sum; // выведет не то, что вы ожидали
```

Такая же проблема кроется в частичной инициализации элементов структуры.

```
struct A
{
      int x, y, z;
};

A a;
a.x = 2;
a.y = 3;
a.y = 0; // хотели написать a.z=0;
printf("%d %d %d", a.x, a.y , a.z);// a.z неопределено
```

Рекомендуется всегда инициализировать переменные при объявлении какими-нибудь дефолтными значениями. Это особенно важно при использовании указателей, отладка которых занимает много драгоценных часов. Также имеет смысл объявлять переменные как можно ближе к месту их использования, это поможет отследить проблему.

- *Выход за пределы массива*

Как уже было сказано ранее, индекс массива может меняться от 0 до n-1, где n – количество элементов массива. Поэтому такой код

```
int arr[5] = { 1, 2, 3, 4, 5 };
for (int i = 1; i <= 5; ++i)
      printf("%d\n", arr[i]);
```

выведет 2 3 4 5 и ... всё что угодно, т.к. arr[5] находится за границей массива.

Чтение или запись информации вне выделенной памяти приводит к неопределенному поведению программы.

Также часто встречается выход за пределы массива в строках в стиле C из-за отсутствии терминального нуля:

```
char str[2] = { 'a', 'b' };
      cout << str;
```

Результат вывода непредсказуем, т.к. неизвестно сколько символов надо выводить.

```
for (int i = 0; i < strlen(str); i++)
      str[i] = 'x';
```

Неизвестно, что вернет *strlen()*, соответственно неизвестно к каким элементам массива обратимся.

Исправленный вариант

```
char str[3] = { 'a', 'b', 0 }; // или char str[3] = "ab";
printf("%s", str);       // вывод ab
for (int i = 0; i < strlen(str); i++) // strlen() возвратит 2
      str[i] = 'x';                   // в массиве окажется строка "xx"
```

- *Работа с локальной копией объекта, вместо работы с самим объектом*

Эта ошибка происходит при использовании параметра, передаваемого по значению. Рассмотрим пример:

```
void foo_create(int* Arr) {
      Arr = (int*)malloc(10*sizeof(int));
      for (int i = 0; i < 10; i++)
            Arr[i] = i;
}
void foo_delete(int * Arr) {
      free(Arr);
      Arr = NULL;
}
int main() {
      int* Ptr;
      foo_create(Ptr);
      for (int i = 0; i < 10; i++) cout << Ptr[i] << "  ";
      foo_delete(Ptr);
      return 0;
}
```

Здесь ошибка в том, что при вызове функции *foo_create* в нее передается значение (копия) указателя *Ptr*, который еще не инициализирован. При выделении памяти, ее адрес помещается в эту копию, т.е. значение самой переменной *Ptr* не изменится.

В результате, после возврата из функции, адрес потеряется и, соответственно, потеряется выделенная память.

Варианты решения проблемы:

а. Передавать в функцию адрес указателя

```
#include <stdio.h>
void foo_create(int** ptr_to_ptr)
{
      *ptr_to_ptr = (int*)malloc(10*sizeof(int));
      for (int i = 0; i < 10; i++)
```

```
            (*ptr_to_ptr)[i] = i;
}

void foo_delete(int** ptr_to_ptr)
{
      free( *ptr_to_ptr);
      *ptr_to_ptr = NULL;
}

int main()
{
      int* Ptr;
      foo_create(&Ptr);
      for (int i = 0; i < 10; i++)
            printf("%d ", Ptr[i]);
      foo_delete(&Ptr);
      printf("%x\n", Ptr);
}
```

б. Возвращать из функции адрес указателя

```
#include <stdio.h>
#include <stdlib.h>
int* foo_create()
{
      int* ptr = (int*)malloc(10 * sizeof(int));
      for (int i = 0; i < 10; i++)
            ptr[i] = i;
      return ptr;
}

void foo_delete(int* ptr)
{
      free(ptr);
        //ptr=NULL; // бесполезно
}

int main()
{
      int* Ptr=foo_create();
      for (int i = 0; i < 10; i++)
            printf("%d ", Ptr[i]);
      foo_delete(Ptr);
      // printf(“%x\n”, Ptr );//указатель не обнулили
}
```

в. Передавать в функцию ссылку на указатель (C++)

```
#include <iostream>
using std::cout;
void foo_create(int* &Arr)
{
      Arr = new int[10];
      for (int i = 0; i < 10; i++) Arr[i] = i;
}

void foo_delete(int* &Arr)
{
      delete[] Arr;
      Arr = nullptr;
}

int main()
```

```
{
	int* Ptr;
	foo_create(Ptr);
	for (int i = 0; i < 10; i++)
		cout << Ptr[i] << "  ";
	foo_delete(Ptr);
	cout << Ptr;
}
```

- *Интерпретация одиночного char символа как символьной строки*

Отдельный тестовый символ нельзя рассматривать как строку содержащую один символ.

```
char c; // отдельный символ, занимающий 1 байт в памяти.
char str[256]; // строка символов длиной не более 255 байт
```

Строка всегда должна заканчиваться признаком окончания строки – числом ноль.

Из-за этого различия вполне реально столкнутся с проблемой преобразования при работе со строками.

Простейший пример

```
char ch = 'z';
char S[255] = "TestString__";
strcat(S, (char *)ch); // попытка объединить строку с символом
```

Здесь используется явное приведение типа char к char *, т.к. написавший подразумевает символ строки как строку в 1 символ, но в этом-то и ошибка.

Для подобного преобразования строки нужно выполнить явное присвоение нужного символа и добавление терминального нуля:

```
char ch = 'z';
char S[255] = "TestString__";
int L = strlen(S);
S[L] = ch;
S[L + 1] = 0;
```

Другой вариант решения – создать массив в 2 символа. Принудительно забить сам символ и признак окончания в этот массив, после чего использовать этот массив вместо изначального символа.

```
char ch = 'z';
char S[255] = "TestString__";
char temp[2] = "";
temp[0] = ch;
temp[1] = 0; // Обязательно
strcat(S, temp);
```

Итак, всегда помните, что строка символов заканчивается нулем!

6. Неправильное поведение программы на этапе исполнения (программа исполняется, но не так, как хотелось бы).

- *Неожиданное закрытие окна*

Когда консольное приложение запускается непосредственно из среды программирования, то после выполнения последнего оператора программы (*return 0;*) окно закрывается.

Вставляйте оператор, ожидающий ввода символа с клавиатуры перед *return*:

```
include <stdlib.h>
int main()
{
      ...
      system("pause");
      return 0;
}
```

или

```
#include <stdio.h>
int main()
{
      int x;
      scanf("%d", x); // последняя операция ввода через scanf
      getchar(); // удаляем оставшийся символ \n из потока ввода

      ...
      puts( "Нажмите Enter для завершения" );
      getchar();
      return 0;
}
```

или с использованием библиотеки низкоуровневого ввода conio.h (на *NIX системах она не используется) :

```
 #include <conio.h>
int main()
{
      ...
      puts( "Нажмите любую клавишу для завершения" );
      _getch();
      return 0;
}
```

- *"Неожиданное" целочисленное деление в арифметических выражениях*

Следует помнить, что результат деления целого на целое - тоже целое:

```
int n; double y;
n = 2;
y = 1 / n;    // y будет равен нулю
y = pow(y, 1 / 3); // возведение в нулевую степень
z = y + n / (n + 1); // прибавляется ноль
```

Рекомендации:

Все константы, которые явно не должны быть целыми, делайте действительными (записывайте с точкой, первое время можете даже после точки ставить нуль 1.0) :

```
int n; double y;
 n = 2;
 y = 1. / n;                // y будет равен 0.5
 y = pow(y, 1. / 3.); //извлечение корня кубического
 z = y + n / (n + 1.); // прибавляется 0.666666666666
```

Там, где в выражении нет констант, используйте оператор приведения

```
 int n, m;
 n = 1;
 m = 2;
double y = (double)n / m;
```

Почему так происходит? Во всех арифметических операциях с двумя аргументами (кроме операций сдвига) они должны иметь один и тот же тип. Если аргументы имеют разный тип, то компилятор строит неявное приведение типа для одного из аргументов, чтобы в конечном итоге выполнить арифметическую операцию над значениями одного и того же типа. Чтобы выяснить, который из двух аргументов нужно приводить к типу другого, в стандарте языка существуют определенные правила.

В случае различных типов один из типов является более "широким", чем другой. Поэтому аргумент более "узкого" типа приводится к более "широкому" типу. Проверка идёт примерно в таком порядке:

- Вещественный тип всегда считается более широким, чем целочисленный
- Когда оба типа вещественные или оба целочисленные, то тип с большим размером всегда считается более широким, чем тип с меньшим размером
- В случае равенства размеров для целочисленных типов беззнаковый тип считается более широким, чем знаковый

В итоге код

```
double f;
int i;
unsigned u;
long long ll;
u + f;
i + u;
ll + u;
```

эквивалентен

```
int i;
unsigned u;
double f;
(double)u + f; // тип результата - double
(unsigned)i + u; // тип результата - unsigned
ll + (long long)u; // тип результата - long long
```

Если мы вернёмся к делению, то следующий код

```
int a, b;
double f;
f = a / b;
```

эквивалентен

```
int a, b, tmp;
double f;
tmp = a / b;
```

```
f = (double)tmp;
```

Сначала выполняем целочисленное деление (при котором отбрасывается остаток) и только потом полученный результат (с утерянной дробной частью) преобразуется в вещественный тип.

Поэтому, чтобы выполнить нормальное вещественное деление нужно хотя бы один из аргументов превратить в вещественный тип, после чего второй аргумент превратится в вещественный тип автоматически по выше описанным правилам. Т.е. явно написать

```
int a, b;
double f;
f = (double)a / b;
```

или

```
int a, b;
double f;
f = a / (double)b;
```

Если один из аргументов является константой, то её следует писать с десятичной точкой в виде "3.0".

- *Ошибки в логических выражениях*

Частой ошибкой является использование операции присваивания (=) вместо сравнения (= =):

```
if (a = 2)
```

всегда истина, т.к. переменной a присваивается двойка, что при приведению к bool дает true.

Рекомендации:

Читайте предупреждения компилятора, о таком присваивании он может сообщить (но не обязательно).

Можно в операции сравнения поменять местами левую и правую часть, тогда возникнет ошибка компиляции (невозможно присвоить значение константе):

```
if (2 = a)
```

Использование побитовых & и | вместо логических && и || тоже довольно частая ошибка

```
char n = 2, k = 1;
if (k & n)  // результат false
if (k && n) // результат true
```

Здесь n в двоичном представлении равно 00000010, а k равно 00000001.

При их побитовом (поразрядном) умножении получим везде нули 00000000, что при приведении к bool даст false.

Во втором случае каждая переменная сначала приводится к bool, в обоих случаях получается true, а потом выполняется логическое умножение.

- *Лишняя точка с запятой*

Довольно часто в конце условного оператора или оператора цикла машинально ставится точка с запятой, тем самым задавая тело оператора пустым.

```
int i;
for (i = 0; i < 10; ++i);   // лишняя ;
          printf("%d", i);
```

В данном случае выведется число 10, а не последовательность от 0 до 9, а оператор for просто 10 раз выполнит пустой цикл.

Такая же ошибка очень часто встречается в определении функций:

```
int cube_func(int x);// скопировали объявление функции, а ; убрать забыли
{
     return x * x * x;
}
```

- ### *switch без break*

Если в операторе switch забыть поставить break перед следующим case, то он тоже выполнится (происходит проваливание к следующей метке):

```
switch (i)
{
case 0: puts("zero");
case 1: puts( "one");
default: puts( "other");
}
```

При i равном 0 будет выведено *zero one other*. При i равном 1 будет выведено *one other*. В большинстве случаев, приведенный выше код является ошибочным, но иногда такое поведение может быть задуманным. В этих случаях следует для очевидности рекомендуется указать в комментарии, что отсутствие *break* это не ошибка, а явная задумка автора.

- ### *Сравнение вещественных чисел при вычислениях*

Поскольку десятичное представление чисел с плавающей запятой обладает некоторой погрешностью, то сравнение на равенство может быть некорректным. Не стоит писать:

```
if (a == b )
```

Сравнивать их надо с некоторой точностью (которая зависит от поставленной задачи), например

```
if (fabs(a - b) < 1e-3)
```

- ### *Сравнение символьных массивов*

При сравнение char массивов через операторы >, <, ==, != мы на самом деле сравниваем не содержимое, а указатели (адреса массивов).

Для правильного сравнения стоит использовать функции сравнения строк типа *strcmp*.

```
char c1[] = "b", c2[] = "a";
if (c1 < c2) // ошибка! ожидаем сравнения в лексикографическом порядке
     printf("I hope this line wouldn't printed on the screen\n");
if (strcmp(c1, c2) < 0) // правильное сравнение
     printf("And it was true\n");
else
     printf("But I was wrong :(");
```

- *Использование чисел, записанных в других системах счисления*

Может показаться, что значения следующих двух переменных одинаковые:

```
int x = 123;
int y = 0123;
```

Однако это не так. Первое число 123 записано в десятичной системе счисления. Поскольку по умолчанию мы работаем с десятичной системой счисления, то *x* так и будет равен 123.

Второе же число 0123 записано в восьмеричной системе счисления и при переводе в десятичную будет равно 83, а не 123. Если вы работаете с десятичной системой счисления, то не добавляйте цифру "0" перед числом.

- *Проверки на принадлежность значения определенному интервалу*

Иногда при проверке принадлежности значения переменной определенному интервалу можно увидеть нечто подобное

```
if (0 <= x <= 10)
```

Несмотря на кажущуюся логичность, этот код не проверяет истинность того, что *x* лежит между 0 и 10.

Если проверку разбить на отдельные шаги, то они будут следующими:

Т.к. оператор сравнения <= левоассоциативен (т.е., выполняется слева направо), то первым шагом будет сравнение нуля и *x* (0 <= x). Результат сравнения (true или false) будет преобразован 1 или 0 соответственно.

Далее сравниваем полученный результат с 10. Поскольку 10 больше и 1 и 0, то условие всегда истинно.

Проверку на принадлежность интервалу нужно записывать с помощью логического оператора И:

```
if (0 <= x && x <= 10)
```

Ошибкой также является использование запятой, вместо логического И:

```
if (0 <= x , x <= 10)
```

В этом случае, значения выражения 0 <= x никак не используется. Результатом будет значение выражения x <= 10.

Иногда вместо логического И ошибочно используют логическое ИЛИ.

Выражение

```
if (0 <= x || x <= 10)
```

не означает, что значение *x* лежит между 0 и 10. Это может быть вообще любое число. Почему так происходит:

Проверяем первое условие. Если *x* больше или равен нулю, то условие истинно. Значит, любое положительное значение уже истина.

Если первое условие не выполнилось, то проверяем второе условие. Любое отрицательное число (а предыдущее условие могло не пройти только отрицательное значение) меньше 10, значит, условие тоже истинное.

Следовательно, условие всегда истинное.

- *Неверный аргумент тригонометрических функций*

При использовании тригонометрических функций часто бывают ошибки такого типа:

```
x = sin(90); // x = 0.893997
```

Имелось ввиду 90 градусов, а функция sin принимает аргумент в радианах.
Решение: перевести в градусы:

```
x = sin(90 * M_PI / 180); // x = 1
```

- *Сравнение знаковой переменной с беззнаковой*

```
int x = -5;
unsigned int y = 100;
if (x < y) // результат равен false
```

Знаковое *x* приводится к беззнаковому, отчего возрастает до *UINT_MAX - 4* , т.к. *UINT_MAX* равно (*unsigned int*)(-1).
Решение – не сравнивать такие типы, заранее приводить к одному.

- *Использование запятой для отделения дробной части*

```
double x;
x = 3, 2; // ожидаем, что присвоится  3.2, а на самом деле 3
x = 3.2;  // Правильно
```

Здесь запятая отделяет две операции, причем первой выполняется левая операция. Поэтому результат присвоения равен 3. Число 2 является синтаксически правильной операцией, поэтому компилятор о такой ошибке не сообщает. Такая ошибка является весьма коварной и её сложно заметить при использовании длинных выражений.

- *Забытое выделение тела цикла for, while и операторов if else*

В циклах часто используют запись тела цикла без фигурных скобок, при условии, что в качестве тела цикла используется один оператор:

```
// допустимо и так:
for (int i = 0; i < n; ++i)
     arr[i] = i * i;

// и так:
for (int i = 0; i < n; ++i)
{
     arr[i] = i * i;
}
```

Это вводит в заблуждение новичков и они пытаются "запихнуть" в цикл несколько операторов, ограничившись отступами:

```
for (int i = 0; i < n; ++i)
    arr[i] = i * i;    // будет выполняться на каждом витке цикла
    printf(“%d”, arr[i]);    // выполнится после цикла
```

Если операторов несколько, то в цикле выполнится только первый, остальные же - только по окончании всех итераций (всех витков в цикле). Поэтому, чтобы добиться задуманного новичок должен был бы написать свой код так :

```
for (int i = 0; i < n; ++i)
{    // оба оператора будут выполняться на каждом витке цикла:
     arr[i] = i * i;
     printf(“%d”, arr[i]);
}
```

Во избежание подобных ошибок рекомендуется ставить скобки даже в случае одного оператора в теле цикла.

Аналогичная ошибка присутствует и в таком примере:

```
if (x == 0)
if (y == 2)
    printf("a\n");
else
    printf("b\n");
```

Из - за отсутствия скобок, оператор else относится не к первому if, а ко второму. Скобки устраняют эту ошибку :

```
if (x == 0)
{
     if (y == 2)
         printf("a\n");
}
else
{
     printf("b\n");
}
```

- ***Определение размера массива, переданного в качестве аргумента функции***

```
int f(int a[]) {
        return sizeof(a) / sizeof(*a);
}

int main() {
       int a[] = { 1, 2, 3 };
       printf("%d\n", sizeof(a) / sizeof(*a) );  // выводит 3
       printf("%d\n", f(a) );                    // выводит 1
}
```

Единственным правильным решением будет передавать размер массива как параметр функции :

```
int f(int* a, int size) {
       return size;
}
int main() {
       int a[] = { 1, 2, 3 };
       printf( "%d\n", f(a, sizeof(a) / sizeof(*a)) );
}
```

Размер массива можно задать при использовании шаблона (C++):

```
#include <iostream>
using std::cout;
template <typename T, int size>
int arr_size(T arr[size]) { return size; }
int  main()
{
        int arr[] = { 1, 2, 3, 4 };
        cout << arr_size(arr) << std::endl;
        return 0;
}
```

Если массив статический, то размер можно узнать, если передать ссылку на массив (C++):

```
#include <iostream>
template <typename T>
int f(T& a) {
         return sizeof(a) / sizeof(*a);
}

int main()
{
        const int N = 5;
        const int M = 15;
        char a[N];
        std::cout << f(a) << std::endl;
        char b[M];
        std::cout << f(b) << std::endl;
        return 0;
}
```

- *Порядок вычисления аргументов при вызове функции*

Нельзя рассчитывать на то, что вычисление значений аргументов в списке параметров функции выполняется слева-направо.

```
#include <stdio.h>
int f1(int* n1)
{
     return ++(*n1);
}
int f2(int n1, int n2)
{
     return n1 + n2;
}
int main() {
     int a = 1;
     printf( "%d", f2(f1(&a), a) );
     return 0;
}
```

Предполагаем, что в выражении *f2(f1(&a), a)* сначала вычисляется функция *f1(a)*, которая изменит *a*. Далее измененное *a* передается вторым параметром в функцию *f2*. Однако, это не так. В данном примере вторым параметром передается старое значение *a* равное 1. Т.о. *f2(f1(a), a)* возвратит значение 3, а не 4.

- *Некорректное использование логических переменных*

Использование логических переменных подразумевает понимание того, что такое логическое выражение. Поэтому такой код

```
bool tmp;
// здесь что то делается с tmp
if (tmp == true)
{
      ...
}
```

Вызывает недоумение, т.к.

```
 if (tmp == true)
```

является тавтологией (повтором)

Рассмотрим его подробнее. Условие *if* срабатывает если в скобках находится истинное выражение. Если *tmp* имеет значение *true*, то результат выражения tmp==true равен *true, if* исполняется. Если *tmp* имеет значение *false*, то результат равен *false*. Как видим результат выражения *tmp==true* равен *tmp*. Поэтому необходимо и достаточно написать ***if (tmp)*** вместо *if (tmp == true)*.

Эта ошибка не так безобидна, как кажется. В WinApi до сих пор используется тип BOOL. Это макрос, который в разных версиях компилятора имел значение и 1 и -1 (все биты единицы). В языке C/C++ принято соглашение все что не 0 – это ИСТИНА, а 0 это ЛОЖЬ.

Теперь представим себе что в переменную типа BOOL *tmp* записали 1, это по соглашениям языка все равно что tmp=ИСТИНА, но TRUE определен как -1. Тогда условие *if (tmp == TRUE)* не сработает, тогда *как if (tmp)* сработает правильно.

- ***Локальная переменная экранирует переменную с таким же именем из вышестоящей области видимости***

Рассмотрим ошибку очень похожую на ошибку описанную в параграфе "Работа с локальной копией объекта, вместо работы с самим объектом".

Ее можно назвать так: "Непреднамеренное затенение объекта в результате повторного локального объявления". Чаще всего она появляется в результате необдуманного копирования фрагмента кода.

Вот один из примеров:

```
int foo(int n) {
      int ret = 0;
      if (n) {
            int ret = 1; // бездумно скопировано с int ret_int=0;
      }
      return ret; // всегда возвращается ноль
}
```

Найти такую ошибку достаточно трудно, т.к. с переменной n все в порядке.

Второй случай. Пусть имеется наскоро написанная функция:

```
int foo(int a) {
      if (a == 0)return 0;
      if (a) {
            int ret = f(a + b);
            return ret;
      }
      return 1;
}
```

Количество return в ней очень раздражает. Попытаемся быстро ее переделать и получим, например:

```
int foo(int a) {
	int ret = 0;
	if (a) {
		int ret = f(a + b); // забыли убрать int
	}
	else
		ret = 1;
	return ret;
}
```

Эта функция при a равном 0 возвратит 1, в остальных случаях – ноль, а не *f(a+b)*. Стоит, однако, заметить, что если ret не инициализировать нулем при объявлении, то компилятор выдаст предупреждающее сообщение об использовании неинициализированной переменной.

7. Ошибки, допущенные при разработке алгоритма

• *Двойная перестановка строк или элементов массива*

Рассмотрим пример инвертирования строки:

```
char str[] = "123456";
int L = strlen(str);
for (int i = 0; i < L; i++)
{
	char tmp = str[i];
	str[i] = str[L - i - 1];
	str[L - i - 1] = tmp;
}
```

Когда в цикле мы дойдем до L / 2, то строка уже перевернута. Последующие итерации до L - 1 вернут буквы на прежние места. Исправленный вариант :

```
char str[] = "123456";
int L = strlen(str);
for (int i = 0; i < L / 2; i++)
{
	char tmp = str[i];
	str[i] = str[L - i - 1];
	str[L - i - 1] = tmp;
}
```

Аналогичные ошибки бывают и при перестановке строк матрицы, при транспонировании матрицы и т.п.

• *Использование символа цифры вместо числа*

Рассмотрим, например, такой код

```
char c;
int n;
c = '9';
n = c + 10;
printf("%d", n);
```

Переменной *n* присвоится не ожидаемое число 19, а 67 (или, может, другое число, если в какой-то кодировке другой код). Причина в том, что в таблице кодировки ASCII символу '9' соответствует код 57, который и используется в арифметическом выражении.

Чтобы избежать этого, т.е., именно получить число 9, а не код символа, необходимо вычесть код символа '0', который равен 48.

```
c = '9';
n = (c - '0') + 10;
printf("%d", n);
```

8. Ошибки ввода-вывода

- ### *Оставление символа '\n' в потоке ввода (C)*

При вводе данных часто забывают о том, что функция форматированного ввода вводит данные до первого пробельного символа и оставляет этот символ в потоке ввода.

```
puts("введите n:");
int n; scanf("%d",&n);      // вводится число после нажатия Enter
puts("введите символ:");
char c = getchar();
```

Предполагаем, что getchar() введет символ из новой строки, но вместо него вводится символ конца строки '\n'. Аналогичным образом:

```
puts("введите n:");
int n; scanf("%d", &n);   // вводится число после нажатия Enter
puts("введите строку:");
char c[100]; fgets(c, 100,stdin);
```

Предполагаем, что fgets введет новую строку, но вводится пустая строка.

Исправление:

```
puts("введите n:");
int n; scanf("%d", &n);   // вводится число после нажатия Enter
getchar(); // пропускаем Enter
puts("введите символ:");
char c = getchar();
```

Последний оператор введет символ с новой строки ('\n' второй строки еще не прочитали). Аналогично поступаем и для ввода строки:

```
puts("введите n:");
int n; scanf("%d", &n);   // вводится число после нажатия Enter
puts("введите строку:");
char c[100];
fgets(c, 100, stdin); // вводим остаток предыдущей строки
fgets(c, 100, stdin); // вводим новую строку
```

- ### *Оставление символа '\n' в потоке ввода (C++)*

При использовании потокового ввода C++ ситуация остается той же:

```
cout << "введите n:";
int n; cin >> n;// вводится число после нажатия Enter
cout << "введите символ:";
char c = cin.get();
```

Предполагаем, что cin.get() введет символ из новой строки, но вместо него вводится символ конца строки '\n'.

```
cout << "введите n:";
int n; cin >> n;     // вводится число после нажатия Enter
cout << "введите строку:";
char c[100]; cin.getline(c, 100);
```

Предполагаем, что cin.getline(c, 100) введет новую строку, но вводится пустая строка. Исправление:

```
cout << "введите n:";
int n;
(cin >> n).get();     // вводим число и пропускаем один символ
cout << "введите символ:";
char c;
cin.get(c);
```

Оператор cin.get(c) введет символ с новой строки('\n' второй строки еще не прочитали). Вместо cin.get() можно просто пропустить символы :

```
cin.ignore(k, '\n');
```

k - пропускаемое количество символов, параметр '\n' можно опускать.

```
cin.ignore(cin.rdbuf()->in_avail()); // пропустить все оставшиеся символы
```

Можно также в цикле прочитать оставшиеся символы

```
while (cin.get() != '\n')
     ; // !!!
```

Третий способ - синхронизация ввода с помощью cin.sync():

```
cout << "введите n:";
int n;
cin >> n;
cin.sync();    // вводим число и сбрасываем остаток строки
cout << "введите символ:";
char c;
cin >> c;
cin.sync();    // вводим символ на новой строке и тоже сбрасываем остаток строки.
```

В общем, возможности большие. Главное помнить об этой проблеме!

- *Ошибки при использовании функции scanf()*

Рассмотрим следующие примеры ошибок:

```
int k;
scanf("%d", &k);    //1
scanf("%d", k);     //2
scanf("%lf", &k);   //3
scanf("%d/n", &k);  //4
double a;
scanf("%lf", &a);   //5
char buf[100];
scanf("%s", buf);   //6
scanf_s("%s", buf); //7
scanf_s("%s", &buf); //8
```

1. Здесь все правильно с точки зрения Си. Однако компилятор может выдать предупреждение (VS 2008) или даже ошибку (VS 2012):

```
warning C4996: 'scanf': This function or variable may be unsafe. Consider using
scanf_s instead.
```

Это нюанс исключительно микрософтовский. Предлагается использовать более защищенную функцию *scanf_s*. Можете послушаться и заменить, а можете отключить это предупреждение добавив

```
#pragma warning(disable:4996)
```

или изменить уровень ошибок в свойствах проекта (VS 2008 2010):

```
Проект->Свойства->Свойства конфигурации->C/C++->Общие->Уровень предупреждений->
Уровень 2(W2).
```

2. Передано значение переменной, а не адрес. Если ранее переменной значение не присваивалось, то компилятор может выдать предупреждение о том, что используется неинициализированная переменная.

3. Вводится целое число, но в форматной строке указано иное (lf – используется для double).

4. Лишние знаки форматирования в командной строке (после ввода числа будет ожидаться еще один Enter).

5. Написано правильно. Однако, тут есть нюанс (который отсутствует в iostrem).

Если локаль языка установлена русская (*setlocale(LC_ALL,"Rus")* ;), то дробная часть отделяется от целой запятой, а не точкой.

6. По формату %s водится не строка целиком, а только очередное слово до пробельного символа. Для ввода всей строки используйте *gets(buf);*

7. *scanf_s* требует передачи размера массива в дополнительном параметре:

```
scanf_s("%s", buf,100); //7
```

8. Передача указателя на массив тоже выполняется правильно. Но в этом случае массив не должен быть динамическим.

- *При работе с fgetc чтение файла обрывается при достижении буквы 'я'*

```
FILE *fp;
char c;
fp = fopen("a.txt", "r");
while ((c = fgetc(fp)) != EOF)
{
      ...
}
```

Проблема кроется не в конкретной букве 'я', а в байте со значением 255. Потому что именно это значение, рассмотренное, как char (0xff) и приведённое к типу int (0xffffffff) совпадёт со значением EOF (которое равно -1). При работе под windows с файлами в кодировке win-1251 код номер 255 имеет буква 'я'. В других кодировках этому коду может соответствовать другая буква.

- *При считывании из файла последний элемент читается дважды*

Такая ошибка возникает, если конец файла проверяется до операции считывания:

```
char c;
```

```
FILE* infile;
while (!feof(infile)) { // Проверка
    c = fgetc(infile);  // считывание
        putchar(c);
}
```

Ситуация «конец файла» возникает только после того, как произойдет чтение за пределами файла. Поэтому создается ощущение, что последний элемент читается дважды.

Для проверки конца файла лучше использовать проверку на ошибочность операции чтения:

```
while ( (c = fgetc(infile))!=EOF ) {
        putchar(c);
}
```

9. Ошибки, связанные с отклонением от стандарта языка

- *Неверный тип функции main()*

Согласно стандарту функция main() должна возвращать целочисленное значение:

```
int main()
{
     // some code
     return 0;
}
```

Однако, для некоторых компиляторов (в том числе для Visual Studio 2008) допускаются и другие возвращаемые типы, в частности может использоваться и *void main()*. Следует помнить, что это может восприниматься как ошибка другими компиляторами. Например, Visual Studio 2017 выдает такое сообщение

```
warning C4326: возвращаемый тип "main" должен быть "int", а не "void".
```

10. Ошибки проектирования АТД (классов) (C++).

- *Отсутствие точки с запятой после определения класса/структуры*

```
struct A
{
       int n;
} // забыли ;
int main()
{
       A a;
       return 0;
}
```

Данная ошибка приводит компилятор в ступор:

```
error C2628: недопустимый 'A' с последующим 'int' (возможно, отсутствует ';')
error C3874: возвращаемый тип 'main' должен быть 'int', а не 'A'
error C2143: синтаксическая ошибка: отсутствие ";" перед "."
error C2664: A::A(const A &): невозможно преобразовать параметр 1 из 'int' в 'const
A &'
```

- *Вызов виртуальной функции из конструктора*

```
class B
```

```
{
	int value;
	virtual int f() const { return 1; }

public:
	B() { value = f(); } // переменной value присваиваем значение через функцию f
	int getValue() const { return value; }
};

class D : public B
{
	virtual int f() const { return 2; }
};

int main()
{
	B* d = new D;
	std::cout << d->getValue() << std::endl; // Выводит 1, а не 2
}
```

Суть в том, что при создании объекта D, сначала создается базовая часть (класс B), а в конструкторе базового класса ничего о классе D еще не известно, т.к. он еще не создан (и в том числе не заполнена таблица виртуальных функций).

- ***Неверный вызов конструктора базового класса из конструктора производного***

Конструктор класса явно вызывать нельзя:

```
class A {
public:
	int x;
	A() { x = 1; }
	A(int v) { x = v; }
};
class B : public A {
public:
	B() {}
	B(int v)
	{
		A(v); // попытка явно вызвать конструктор базового класса
	}
};
int main()
{
	B b(2);
	std::cout << b.x;	// b.x = 1, а не 2
}
```

В этом примере при создании объекта сначала неявно вызывается конструктор по умолчанию класса *A*, который присваивает переменной *x* значение 1. Далее в конструкторе класса *B* оператором *A(v)* создается временный объект класса *A*, в котором *x*=2. Однако, он удаляется при выходе из конструктора *B*.

Решение: для вызова конструктора базового класса необходимо использовать список инициализации:

```
class A {
public:
	int x;
	A() : x(1) {}
	A(int v) : x(v) {}
};
```

```
class B : public A {
public:
      B() {}
      B(int v) : A(v) {}
};

int main() {
      B b(2);
      std::cout << b.x;     // b.x = 2
}
```

- *Неверный порядок при инициализации*

Неверный порядок при инициализации может стать источником ошибок:

```
struct Test
{
      int x;
      int y;
      Test(int y_) : y(y_), x(y * 2) {}
};
int main() {
      Test t(10);
      std::cout << t.x << std::endl;
      std::cout << t.y << std::endl;
}
```

В списке инициализации конструктора, инициализация происходит в порядке объявления переменных, а не в порядке, указанном в списке инициализации.

В данном случае, сначала будет инициализирована переменная x некоторым случайным значением, а только потом переменная y.

- *Нарушение правила TPEX.*

Правило трёх (также известное как «Закон Большой Тройки» или «Большая Тройка») гласит, что если класс или структура определяет один из следующих методов, то они должны явным образом определить все три метода:

1. Деструктор
2. Конструктор копирования
3. Оператор присваивания

Эти три метода являются особыми членами-функциями.

Они автоматически создаются компилятором в случае отсутствия их явного объявления.

Если один из них должен быть определен программистом, то это означает, что версия, сгенерированная компилятором, не удовлетворит потребности класса и для двух других.

Пример:

```
class A
{
private:
      int * x; // указатель на массив
      int n;

public:
      A() : x(0), n(0) {}
      A(int N) : n(N)
      {
```

```
		x = new int[N];
		// под массив выделяется память - нужен деструктор для ее удаления
		for (int i = 0; i < N; i++)
			x[i] = i;
	}

	~A() // Создаем деструктор
	{
		delete[] x;
	}

	A(const A & a) : n(a.n)
	//  Обязательный копиконструктор (раз есть деструктор)
	{
		x = new int[a.n];
		for (int i = 0; i < a.n; i++)
			x[i] = a.x[i];
	}

	A & operator = (const A & a)
	// Обязательный оператор присвоения (раз есть деструктор)
	{
		if (this == &a)
			return *this;
		// присвоение самому себе, ничего делать не надо

		delete[] x;

		x = new int[a.n];
		for (int i = 0; i < a.n; i++)
			x[i] = a.x[i];

		return *this;
	}

	A operator + (A & a) // слияние массивов
	{
		A c;
		c.n = this->n + a.n;
		c.x = new int[c.n];

		int i = 0;
		for (; i < this->n; i++)
			c.x[i] = this->x[i];

		int j = 0;
		for (; i < c.n; i++, j++)
			c.x[i] = a.x[j];

		return c;
		// для передачи по значению используется копиконструктор
	}
};

int main()
{
	A a1(2), a2(3), a3;
	a3 = a1 + a2;
	// используется оператор присвоения
	return 0;
}
```

Примечание.

Не забывайте и о том, что и конструктор по умолчанию тоже в этом случае не создается. Правда, при необходимости его вызова компилятор выдаст предупреждающее сообщение:

```
error C2512: A: нет подходящего конструктора по умолчанию
```

- *Отсутствие виртуального деструктора в базовом классе*

Класс по умолчанию генерирует деструктор, но при этом он не является виртуальным. Отличие виртуального деструктора в том, что он предполагает освобождение ресурсов не только базового класса, но и всех производных. Ниже пример, явно демонстрирующий проблему:

```
class A
{
public:
      A() { cout << "A "; }
      ~A() { cout << "~A "; }
};
class B : public A
{
public:
      B() { cout << "B"; }
      ~B() { cout << "~B "; }
};
int main()
{
      A* p = new B;
      delete p;
}
```

Этот пример выводит "*A B ~A* ", т.е. деструктор класса B не вызывается.

При размещении объекта производного класса на стеке, вызов деструкторов будет правильным, а вот при динамическом (как выше), произойдёт неполное освобождение ресурсов т.к. деструктор не виртуальный. Таким образом, виртуальный деструктор нужен для правильного полиморфного удаления объектов.

Для исправления ошибки в примере выше достаточно сделать деструктор класса A виртуальным:

```
virtual ~A() { cout << "~A()"; }
```

Во избежание проблем деструктор следует объявлять виртуальным

– при наличии хотя бы одного виртуального метода,

– если класс предполагается в будущем сделать базовым.

- *Неправильное обращение к конструктору по умолчанию*

```
class A
{
public:
      A () {}
};
...
A a();
```

Здесь планировалось объявить переменную типа *A* с конструктором по умолчанию. Круглые скобки были поставлены для явного указания, что нужен конструктор без параметров.

Однако, такая конструкция интерпретируется компилятором как объявление (прототип) функции без параметров с именем *a*, которая возвращает значение типа *A*.

Правильным будет такое объявление:

```
A a;
```

- ### *Не очевидные моменты с вызовом конструктора базового класса*

При копировании объектов производного класса у базового класса вместо конструктора копии запускается конструктор по умолчанию.
Рассмотрим код:

```
struct base
{
	size_t	mSize;
	int *mData;
	~base() { delete[] mData; }
	base() :mSize(0), mData(nullptr) {}
	base(size_t size) : mSize(size), mData(new int[size]) {}
	base(const base & rhs)
		// раз мы его объявили, значит нам это важно
		// он обязательно должен запуститься при копировании объектов
		: mSize(rhs.mSize), mData(new int[mSize]) {}

};
struct der : base
{
	int mValue;
	der() : mValue(0)
		// какой конструктор base будет запущен?
	{}
	der(const size_t size, const int value = 0), mValue(value)
		// какой конструктор base будет запущен?
	{}

	der(const der & rhs) : mValue(rhs.mValue)
		// какой конструктор base будет запущен?
	{}
	void view() const
	{
		std::cout << "value = " << mValue
			<< " : size = " << mSize << '\n';
	}

};

int main()
{
	der d1(10, 10);
	d1.view(); // value = 10 : size = 10

	der d2 = d1; // вызов копиконструктора
	d2.view(); // что будет выведено?
}
```

Будет выведено:

```
value = 10 : size = 0
```

Отсюда делаем вывод : нужно всегда явно указывать, какой базовый конструктор использовать :

```
der() : mValue(0),
base() {}
der(const size_t size, const int value = 0) : mValue(value),
base(size) {}

der(const der & rhs) : mValue(rhs.mValue),
base(rhs) {}
```

- *Неявно объявленный конструктор по умолчанию*

Конструктор по умолчанию вставляется компилятором при отсутствии объявления других конструкторов. Но, как только мы описали хоть один конструктор, конструктор по умолчанию пропадает:

```
class A
{
};
A a;
```

Здесь сработает конструктор по умолчанию. А в таком коде

```
class A
{
    A(int b);
}
A a;
```

получим ошибку компиляции «нет подходящего конструктора», поскольку конструктор по умолчанию пропал, а конструктора без параметров нет. Необходимо описать конструктор без параметров.
Возможный выход – описание конструктора с параметрами по умолчанию

```
class A
{
    A(int b=0);
}
A a;
```

Примерно то же относится и к конструкторам копирования и деструкторам, если мы их не описали, то компилятор вставит свои, но они не всегда делают то что нужно.

- *Перегрузка операторов ввода/вывода (>> и <<) для шаблонных классов*

Объявим операторы operator>> и operator<< так, как ранее объявляли для нешаблонных классов:

```
template<class T>
Class A
{
     friend ostream& operator<<(ostream& out, const  A<T>&);
     friend istream& operator>>(istream& in, A<T>&);

};
```

Получим ошибки компиляции такого вида

```
Error 1 error LNK2019: unresolved external symbol "class
std::basic_istream<char,struct std::char_traits<char> > & __cdecl operator>>(class
std::basic_istream<char,struct std::char_traits<char> > &,class A<int> &)"
```

Причина в том, что для внешней функции нужно задать свой параметр шаблона:

```
template<class T>
Class A
{
    Template<class T2>
    friend ostream& operator<<(ostream& out, const  A<T2>& obj);
       template<class T2>
       friend istream& operator>>(istream& in, A<T2>& obj);
};
```

Или можно сделать предварительное объявление этих функций:

```
template< class T>
class A;
template< class T>
ostream & operator<< (ostream& out, const A<T>& obj);
template< class T>
istream & operator>> (istream& in, A<T>& obj);
template<class T>
Class A
{
    friend ostream& operator<<(ostream& out, const  A<T>& obj);
    friend istream& operator>>(istream& in, a<T>& obj);
};
```

11. Ошибки при использовании STL контейнеров

- *Невалидные ссылки/указатели, при перемещении объектов*

При использовании указателей на элементы STL контейнеров следует иметь ввиду, что при изменении контейнера эти ссылки могут стать невалидными.

Рассмотрим пример использования вектора:

1. В вектор записываем объекты по значению.
2. Из вектора запоминаем указатель на какое-либо из его значений.
3. В вектор добавляем ещё некоторое количество объектов.
4. В какой-то момент резерв памяти иссякает, и вектор «реалочится» – расширяет буфер перенося туда свои объекты. В результате объекты меняют адрес.
5. Выданный наружу указатель становятся недействительным.
6. Обращение по невалидному указателю приводит к ошибке времени выполнения.

Искать причину таких вылетов затруднительно, потому что:

1. Авария обычно происходит далеко от места причины аварии.
2. В целях оптимизации для вектора обычно резервируется память, например методом vector::reserve(), и вектор редко «реалочится».

В результате ошибки долгое время могут оставаться незамеченными.

```
#include <iostream>
#include <vector>
struct some
{
       int val;
       some() :val(10) {}
```

```
        void foo() {
            val += 10; std::cout << "val = " << val << '\n';
        }
};

int main()
{
        std::vector<some> vec;
        vec.reserve(2);       // зарезервировали 2 элемента
        vec.push_back(some());// записали 1 элемент
        some& s = vec.back();
        s.foo();
        vec.push_back(some());// записали 2 элемент, резерв кончился
        s.foo();              // OK
        vec.push_back(some());// записали 3 элемент
        s.foo();              // здесь уже невалидная ссылка
        return 0;
}
```

Подобную ошибку можно выделить в целый класс ошибок связанных с "неперемещаемостью объектов". Помимо вектора, она может встречаться в самых разных ситуациях.

- *Ошибки связанные с итераторами (удаление элементов по итератору в циклах)*

Рассмотрим пример:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
      list <int> mylist;

      for (size_t n = 0; n < 5; ++n)
            for (size_t i = 0; i < 3; ++i)
                  mylist.push_back(n);
      // --- мы хотим бежать по всему контейнеру
      // --- и удалить все элементы со значением 3
      list <int>::iterator it = mylist.begin();
      for (; it != mylist.end(); ++it)
            if (*it == 3)
                  mylist.erase(it); // <--- итератор стал невалидным
}
```

После удаления элемента, итератор ссылается на несуществующий элемент. Однако в цикле для него делается инкремент (++it).

Первая мысль, которая может прийти в голову, после удаления, присвоить итератору новое значение, чтобы итератор всегда оставался валидным:

```
it = mylist.erase(it);
```

Это может привести к другой весьма распространенной ошибке:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
```

```
        list <int> mylist;
        for (size_t n = 0; n < 5; ++n)
        {
                mylist.push_back(n),
                mylist.push_back(n),
                mylist.push_back(n);
        }


        // --- мы хотим бежать по всему контейнеру
        // --- и удалить все элементы с ключем 3
        list <int>::iterator it = mylist.begin();
        for (; it != mylist.end(); ++it)
                if (*it == 3)
                        it = mylist.erase(it);

        for (it = mylist.begin(); it != mylist.end(); ++it)
                cout << *it << ", ";
        cout << endl;
 }
```

Крушений больше не происходит, но логика работы нарушена - алгоритм "проскочил" мимо одного из удаляемых элементов.

Это связанно с тем, что инструкция: it = mylist.erase(it); присваивает итератору ссылку на элемент, идущий сразу же следом после удаляемого. После чего отрабатывает инструкция цикла ++it. В результате алгоритм перескакивает через элемент.
Правильная версия выглядит так:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
        list <int> mylist;
        for (size_t n = 0; n < 5; ++n)
        {
                mylist.push_back(n),
                        mylist.push_back(n),
                        mylist.push_back(n);
        }


        // --- мы хотим бежать по всему контейнеру
        // --- и удалить все элементы с ключем 3
        // --- цикл больше не инкрементирует счетчик
        list <int>::iterator it = mylist.begin();
        for (; it != mylist.end();)
                if (*it == 3)
                        it = mylist.erase(it);
                else
                        ++it; // <--- делаем инкремент вручную


        for (it = mylist.begin(); it != mylist.end(); ++it)
                cout << *it << ", ";
        cout << endl;
}
```

- *Ошибки связанные с итераторами (префикс-постфиксные инкременты при удалении элементов в цикле)*

В профессиональном коде часто можно встретить использование свойств префикс-постфикс операций:

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
	list <int> mylist;
	for (size_t n = 0; n < 5; ++n)
	{
		mylist.push_back(n),
			mylist.push_back(n),
			mylist.push_back(n);
	}


	// --- мы хотим бежать по всему контейнеру
	// --- и удалить все элементы с ключем 3
	list <int>::iterator it = mylist.begin();
	for (; it != mylist.end();)
		if (*it == 3)
			mylist.erase(it++); // постфиксная операция
		else
			++it; //


	for (it = mylist.begin(); it != mylist.end(); ++it)
		cout << *it << ", ";
	cout << endl;
}
```

Если нечаянно постфиксную операцию перепутать его с префиксной, то, в лучшем случае, получим удаление не того элемента, который планировался.

При постфиксной форме будет удален текущий элемент несмотря на то, что сам итератор инкрементируется ещё до удаления. А при префиксной - сначала инкрементируется, и только потом будет удален уже следующий элемент. При этом, попытка удалить таким образом последний элемент приведет к удалению элемента за пределами контейнера.

Т.о. не нужно обсуждать, что быстрее: ++i или i++, а нужно понимать, что делают обе формы инкремента, и использовать их по назначению.

Литература

Основы программирования и синтаксис языка

Поскольку уровень готовности у каждого разный, то выбор литературы зависит от уже имеющихся знаний. Постарайтесь выбрать ту книгу, которая бы наиболее подходила именно Вам. Бывает, что с первых слов не нравится стиль изложения материала - это повод обратиться к другим авторам. Если некоторые моменты остались непонятыми, прочитайте соответствующие главы в других книгах.

- *Брайан Керниган, Деннис Ритчи - Язык программирования Си*

Оригинальное название: The C Programming Language
Автор: Брайан Керниган (Brian Kernighan), Деннис Ритчи (Dennis Ritchie)
Издание: 2-е, 2016
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 288
ISBN: 978-5-8459-1874-1, 0-13-110362-8

Классическая книга по языку С, написанная самими разработчиками этого языка и выдержавшая в США уже 34 переиздания! Книга является как практически исчерпывающим справочником, так и учебным пособием по самому распространенному языку программирования. Предлагаемое второе издание книги было существенно переработано по сравнению с первым в связи с появлением стандарта ANSI C, для которого она частично послужила основой.

Книга не рекомендуется для чтения новичкам; для своего изучения она требует знания основ программирования и вычислительной техники. Книга предназначена для широкого круга программистов и компьютерных специалистов. Может использоваться как учебное пособие для вузов.

- *Брюс Эккель - Философия C++. Введение в стандартный C++*

Оригинальное название: Thinking in C++. Introduction to Standart C++
Автор: Брюс Эккель (Bruce Eckel)
Издание: 2-е, 2004
Издательство: Питер
Переплёт: мягкий
Количество страниц: 572
ISBN: 0-13-979809-9, 5-94723-763-6

В книге "Философия C++" последовательно и методично излагаются вопросы использования объектно-ориентированного подхода к созданию программ. Автор не просто описывает различные проблемы и способы их решения, он раскрывает перед читателем особый образ мышления, не владея которым невозможно комфортно чувствовать себя в объектно-ориентированной среде.

Это одна из тех книг, которые обязательно должен прочесть каждый, кто всерьез занимаете разработкой программного обеспечения в C++.

- *Стивен Прата - Язык программирования C++. Лекции и упражнения*

Оригинальное название: C++ Primer Plus
Автор: Стивен Прата (Stephen Prata)
Издание: 6-е, 2017
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 1248
ISBN: 978-5-8459-2048-5, 978-5-8459-1778-2

Книга представляет собой тщательно проверенный, качественно составленный полноценный учебник по одной из ключевых тем для программистов и разработчиков. Эта классическая работа по вычислительной технике обучает принципам программирования, среди которых структурированный код и нисходящее проектирование, а также использованию классов, наследования, шаблонов, исключений, лямбда-выражений, интеллектуальных указателей и семантики переноса.

Автор и преподаватель Стивен Прата создал поучительное, ясное и строгое введение в C++. Фундаментальные концепции программирования излагаются вместе с подробными сведениями о языке C++. Множество коротких практических примеров иллюстрируют одну или две концепции за раз, стимулируя читателей осваивать новые темы за счет непосредственной их проверки на практике. Вопросы для самоконтроля и упражнения по программированию, предлагаемые в конце каждой главы, помогут читателям сосредоточиться на самой критически важной информации и систематизировать наиболее сложные концепции.

Написанное в дружественном стиле, простое в освоении руководство для самостоятельного изучения подойдет как студентам, обучающимся программированию, так и разработчикам, имеющим дело с другими языками и стремящимся лучше понять фундаментальные основы этого ключевого языка программирования. Шестое издание этой книги обновлено и расширено с учетом последних тенденций в разработке на C++, а также с целью детального отражения нового стандарта C++11.

- *Стенли Липпман - Язык программирования C++. Базовый курс*

Оригинальное название: C++ Primer
Авторы: Стенли Липпман (Stanley Lippman), Жози Лажойе (Josée Lajoie), Барбара Му (Barbara Moo)
Издание: 5-е, 2017
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 1120
ISBN: 5-8459-1121-4, 0-201-72148-1

Книга "Язык программирования C++. Базовый курс" (5-е издание) - новое издание популярного и исчерпывающего бестселлера по языку программирования C++, которое было полностью пересмотрено и обновлено под стандарт C++11. Оно поможет вам быстро изучить язык и использовать его весьма эффективными и передовыми способами. В соответствии с самыми передовыми и современными методиками изложения материала

авторы демонстрируют использование базового языка и его стандартной библиотеки для разработки эффективного, читабельного и мощного кода.

С самого начала книги "Язык программирования C++. Базовый курс" читатель знакомится со стандартной библиотекой C++, ее самыми популярными функциями и средствами, что позволяет сразу же приступить к написанию полезных программ, еще не овладев всеми нюансами языка. Большинство примеров из книги было пересмотрено так, чтобы использовать новые средства языка и продемонстрировать их наилучшие способы применения. Книга "Язык программирования C++. Базовый курс" - не только проверенное руководство для новичков в C++, она содержит также авторитетное обсуждение базовых концепций и методик языка C++ и является ценным ресурсом для опытных программистов, особенно желающих побыстрей узнать об усовершенствованиях C++11.

- *Бьярне Страуструп - Программирование. Принципы и практика с использованием C++*

Оригинальное название: Programming: Principles and Practice Using C++
Авторы: Бьярне Страуструп (Bjarne Stroustrup)
Издание: 2-е, 2016
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 1328
ISBN: 978-5-8459-1949-6, 978-0-321-99278-9

Эта книга не является учебником по языку C++, это учебник по программированию. Несмотря на то что ее автор — автор языка C++, книга не посвящена этому языку программирования; он играет в книге сугубо иллюстративную роль. Автор задумал данную книгу как вводный курс по программированию. Поскольку теория без практики совершенно бессмысленна, такой учебник должен изобиловать примерами программных решений, и неудивительно, что автор языка C++ использовал в книге свое детище. В книге в первую очередь описан широкий круг понятий и приемов программирования, необходимых для того, чтобы стать профессиональным программистом, и в гораздо меньшей степени — возможности языка программирования C++.

В первую очередь, книга адресована начинающим программистам и студентам компьютерных специальностей, которые найдут в ней много новой информации, и смогут узнать точку зрения создателя языка C++ на современные методы программирования. Если вы решили стать программистом, и уже знакомы с азами C++ — эта книга для вас, в первую очередь потому, что программирование — это не только, и не столько знание инструмента (языка программирования C++), сколько понимание самого процесса. Автор недаром не ограничился своим первоклассным (но ни в коей мере не являющимся учебником для программистов без большого практического опыта) трудом Язык программирования C++.

Проводя грубую аналогию — виртуозное владение топором никого не делало настоящим плотником. Бьярне Страуструп в очередной раз приходит на помощь программистам — создав уникальный язык программирования, он не ограничивается им и рассказывает о том, как правильно им воспользоваться, даже не зная все его тонкости и возможности.

Основные темы книги:

Подготовка к созданию реальных программ. Автор книги предполагает, что читатели в конце концов начнут писать нетривиальные программы либо в качестве профессиональных разработчиков программного обеспечения, либо в качестве программистов, работающих в других областях науки и техники. Упор на основные концепции и методы. Основные концепции и методы программирования в книге излагаются глубже, чем это принято в традиционных вводных курсах. Этот подход дает основательный фундамент для разработки полезных, правильных, понятных и эффективных программ. Программирование на современном языке C++ (C++11 и C++14). Книга представляет собой введение в программирование, включая объектно-ориентированное и обобщенное программирование. Одновременно она представляет собой введение в язык C++, один из широко применяющихся языков программирования в современном мире. В книге описаны современные методы программирования на C++, включая стандартную библиотеку и возможности C++11 и C++14, позволяющие упростить программирование.

Для начинающих программистов и всех, кто хочет научиться программировать. Книга предназначена в основном для людей, никогда ранее не программировавших, и опробована на более чем тысяче студентов университета. Однако и опытные программисты, и студенты, уже изучившие основы программирования, найдут в книге много полезной информации, которая позволит им перейти на еще более высокий уровень мастерства.

Широкий охват тем. Первая половина книги охватывает широкий спектр основных понятий, методов проектирования и программирования, свойств языка C++ и его библиотек. Это позволит читателям писать программы, выполняющие ввод и вывод данных, вычисления и построение простых графических изображений. Во второй половине рассматриваются более специализированные темы (такие как обработка текста, тестирование и язык C). В книге содержится много справочного материала. Исходные тексты программ и иные материалы читатели могут найти на веб-сайте автора.

- *Харви Дейтел, Пол Дейтел - Как программировать на C++*

Оригинальное название: C++: How to Program
Авторы: Харви М. Дейтел (H. M. Deitel), Пол Дж. Дейтел (P. J. Deitel)
Издание: 5-е, 2008[1]
Издательство: Бином-Пресс
Переплёт: твёрдый
Количество страниц: 1456
ISBN: 978-5-9518-0224-8, 0-13-185757-6

Книга является одним из самых популярных в мире учебников по C++. Характерной ее особенностью является "раннее введение" в классы и объекты, т. е. начала объектно-ориентированного программирования вводятся уже в 3-й главе, без предварительного изложения унаследованных от языка C элементов процедурного и структурного программирования, как это делается в большинстве курсов по C++. Большое внимание уделяется объектно-ориентированному проектированию (OOD) программных систем с помощью графического языка UML 2, чему посвящен ряд факультативных разделов, описывающих последовательную разработку большого учебного проекта.

В текст книги включена масса примеров "живого кода" - подробно комментированных работающих программ с образцами их запуска, а также несколько подробно разбираемых интересных примеров. В конце каждой главы имеется обширный набор контрольных вопросов и упражнений.

Книга может служить учебным пособием для начальных курсов по C++, а также будет полезна широкому кругу как начинающих программистов, так и более опытных, не работавших прежде с C++.

Продвинутое изучение языка C++

Язык программирования C++ часто критикуют за сложность понимания, а также наличие потенциально опасных конструкций и возможностей. Вы уже владеете основами, все еще любите и желаете изучать C++? Нижеследующие книги позволят Вам разобраться в тонкостях и избежать многих ошибок.

- *Бьярне Страуструп - Язык программирования C++*

Оригинальное название: The C++ Programming Language
Автор: Бьярне Страуструп (Bjarne Stroustrup)
Издание: Специальное издание, 2011[1]
Издательство: Бином
Переплёт: твёрдый
Количество страниц: 1136
ISBN: 978-5-7989-0425-9, 0-201-70073-5

Книга написана Бьярне Страуструпом - автором языка программирования C++ - и является каноническим изложением возможностей этого языка. Помимо подробного описания собственно языка, на страницах книги вы найдете доказавшие свою эффективность подходы к решению разнообразных задач проектирования и программирования. Многочисленные примеры демонстрируют как хороший стиль программирования на C-совместимом ядре C++, так и современный объектно-ориентированный подход к созданию программных продуктов.
Книга адресована программистам, использующим в своей повседневной работе C++. Она также будет полезна преподавателям, студентам и всем, кто хочет ознакомиться с описанием языка "из первых рук".

- *Эндрю Кениг, Барбара Му - Эффективное программирование на C++*

Оригинальное название: C++ In-Depth Box Set First Edition, Vol. 2: Accelerated C++: Practical Programming by Example
Автор: Эндрю Кениг (Andrew Koenig), Барбара Му (Barbara E. Moo)
Издание: 1-е, 2015
Издательство: Вильямс
Переплёт: мягкий
Количество страниц: 368
ISBN: 5-8459-0350-5

Эта книга, в первую очередь, предназначена для тех, кому хотелось бы быстро научиться писать настоящие программы на языке C++. Зачастую новички в C++ пытаются

освоить язык чисто механически, даже не попытавшись узнать, как можно эффективно применить его к решению каждодневных проблем. Цель данной книги - научить программированию на C++, а не просто изложить средства языка, поэтому она полезна не только для новичков, но и для тех, кто уже знаком с C++ и хочет использовать этот язык в более натуральном, естественном стиле.

- *Скотт Мейерс - Эффективное использование C++: 55 верных советов улучшить структуру и код ваших программ*

Оригинальное название: Effective C++: 55 Specific Ways to Improve Your Programs and Designs
Автор: Скотт Мейерс (Scott Meyers)
Издание: 3-е, 2014
Издательство: ДМК-Пресс
Переплёт: мягкий
Количество страниц: 300
ISBN: 5-94074-304-8, 0-321-33487-6, 978-5-97060-088-7

Эта книга представляет собой перевод третьего издания американского бестселлера Effective C++ и является руководством по грамотному использованию языка C++. Она поможет сделать ваши программы более понятными, простыми в сопровождении и эффективными. Помимо материала, описывающего общую стратегию проектирования, книга включает в себя главы по программированию с применением шаблонов и по управлению ресурсами, а также множество советов, которые позволят усовершенствовать ваши программы и сделать работу более интересной и творческой. Книга также включает новый материал по принципам обработки исключений, паттернам проектирования и библиотечным средствам.

Издание ориентировано на программистов, знакомых с основами C++ и имеющих навыки его практического применения.

- *Скотт Мейерс - Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов*

Оригинальное название: More Effective C++: 35 New Ways to Improve Your Programs and Designs
Автор: Скотт Мейерс (Scott Meyers)
Издание: 1-е, 2016
Издательство: ДМК-Пресс
Переплёт: мягкий
Количество страниц: 298
ISBN: 5-469-01215-8, 0-201-63371-X

В книге С.Мейерса, которая является продолжением популярного издания Effective C++, приводятся рекомендации по наиболее эффективному использованию конструкций языка C++. Рассматриваются правила перегрузки операторов, способы приведения типов, реализация механизма RTTI и многое другое. Даны практические советы по применению буферизованного оператора new, виртуальных конструкторов, интеллектуальных

указателей, proxy-классов и двойной диспетчеризации. Особое внимание уделяется работе с исключениями и возможностям использования кода C в программах, написанных на C++. Подробно описаны новейшие средства языка и показано, как с их помощью повысить производительность программ. Приложения содержат код шаблона auto_ptr и аннотированный список литературы и Internet-ресурсов, посвященных C++.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

- *Скотт Мейерс - Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14*

Оригинальное название: Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14
Автор: Скотт Мейерс (Scott Meyers)
Издание: 1-е, 2016
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 304
ISBN: 978-5-8459-2000-3, 978-1-49-190399-5

В этой книге отражен бесценный опыт ее автора как программиста на C++. Новые возможности этого языка программирования, появившиеся в стандартах C++11 и C++14 - это не просто новые ключевые слова или функции, это появление совершенно новых концепций, так что для их эффективного использования недостаточно просто узнать об их существовании, и программировать на C++11, как на несколько улучшенном и расширенном функционально C++98.

Когда происходят такие глобальные изменения в языке программирования, их изучению следует посвятить определенное время, написать сотни, а еще лучше - тысячи строк кода, и столкнуться с массой проблем, кажущихся тем более странными и непонятными, чем большим опытом работы с C++98 вы обладаете. К программированию в полной мере относится фраза Евклида о том, что в геометрии нет царских путей. Но пройти путь изучения и освоения нового языка программирования вам может помочь проводник, показывающий наиболее интересные места и предупреждающий о ямах и ухабах. Таким проводником может послужить книга Скотта Мейерса. С ней вы не заблудитесь и не забредете в дебри, из которых будете долго и трудно выбираться с помощью отладчика.

При этом книга не просто научит вас использовать новые возможности языка - она научит использовать их эффективно. Но и это не все - книга не просто учит эффективному применению C++, но еще и рассказывает, почему ту или иную задачу следует решать именно так.

Эта книга заставляет вас не просто заучить правила - она заставляет думать.

И хотя эта книга в первую очередь предназначена для энтузиастов и профессионалов, она достойна места на полке любого программиста - как профессионала, так и зеленого новичка.
Освоение C++11 и C++14 - это больше, чем просто ознакомление с вводимыми этими стандартами возможностями (например, объявлениями типов auto, семантикой перемещения, лямбда-выражениями или поддержкой многопоточности). Вопрос в том, как использовать их эффективно - так, чтобы создаваемые программы были корректны, эффективны и переносимы, а также чтобы их легко можно было сопровождать. Именно этим вопросам и посвящена данная книга, описывающая создание по-настоящему хорошего программного обеспечения с использованием C++11 и C++14 - т.е. с использованием современного C++.

В книге рассматриваются следующие темы:

Преимущества и недостатки инициализации с помощью фигурных скобок, спецификации noexcept, прямой передачи и функций make интеллектуальных указателей;

Связь между std::move, std::forward, rvalue-ссылками и универсальными ссылками;

Методы написания понятных, корректных, эффективных лямбда-выражений;

Чем std::atomic отличается от volatile, как они используются и как соотносятся с API параллельных вычислений C++;

Какие из лучших методов "старого" программирования на C++ (т.е. C++98) должны быть пересмотрены при работе с современным C++.

Эффективный и современный C++, следуя принципам более ранних книг Скотта Мейерса, охватывает совершенно новый материал. Эта книга достойна занять свое место на полке каждого программиста на современном C++.

- *Герб Саттер - Решение сложных задач на C++*

Оригинальное название: Exceptional C++, More Exceptional C++[1]
Автор: Герб Саттер (Herb Sutter)
Издание: 1-е, 2015
Издательство: Вильямс
Переплёт: мягкий
Количество страниц: 400
ISBN: 978-5-8459-0352-5, 0-201-77581-6

В данном издании объединены две широко известные профессионалам в области программирования на C++ книги Герба Саттера Exceptional C++ и More Exceptional C++ , входящие в серию книг C++ In-Depth, редактором которой является Бьерн Страуструп, создатель языка C++.

Материал этой книги составляют переработанные задачи серии Guru of the Week, рассчитанные на читателя с достаточно глубоким знанием C++, однако книга будет полезна каждому, кто хочет углубить свои знания в этой области.

- *Герб Саттер - Новые сложные задачи на C++*

Оригинальное название: Exceptional C++ Style
Автор: Герб Саттер (Herb Sutter)
Издание: 1-е, 2015
Издательство: Вильямс

Переплёт: мягкий
Количество страниц: 272
ISBN: 978-5-8459-1972-4

Данная книга представляет собой продолжение вышедшей ранее книги Решение сложных задач на C++. В форме задач и их решений рассматриваются современные методы проектирования и программирования на C++. В книге сконцентрирован богатый многолетний опыт программирования на C++ не только самого автора, но и всего сообщества программистов на C++, так что некоторые рекомендации автора могут показаться неожиданными даже опытным программистам-профессионалам. Автор рассматривает и конкретные методики, приемы и идиомы программирования, однако основная тема книги - это стиль программирования, причем в самом широком понимании этого слова. Особое внимание во всех задачах книги уделено вопросу проектирования, которое должно обеспечить максимальную надежность, безопасность, производительность и сопровождаемость создаваемого программного обеспечения.

Книга рассчитана в первую очередь на профессиональных программистов с глубокими знаниями языка, однако она будет полезна любому, кто захочет углубить свои знания в данной области.

- *Стивен Дьюхерст - Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ*

Оригинальное название: C++ Gotchas: Avoiding Common Problems in Coding and Design
Автор: Стивен Дьюхерст (Stephen C. Dewhurst)
Издание: 1-е, 2017
Издательство: ДМК Пресс
Переплёт: мягкий
Количество страниц: 264
ISBN: 978-5-97060-475-5, 978-0-321-12518-7

Это руководство по тому, как не допускать и исправлять 99% типичных, разрушительных и просто любопытных ошибок при проектировании и реализации программ на языке C++. Эту книгу можно рассматривать также, как взгляд посвященного на нетривиальные особенности и приемы программирования на C++.

Обсуждаются как наиболее распространенные "ляпы", имеющиеся почти в любой программе на C++, так и сложные ошибки в использовании синтаксиса, препроцессора, преобразований типов, инициализации, управления памятью и ресурсами, полиморфизма, а также при проектировании классов и иерархий. Все ошибки и их последствия обсуждаются в контексте. Подробно описываются способы разрешения указанных проблем.

Автор знакомит читателей с идиомами и паттернами проектирования, с помощью которых можно решать типовые задачи. Читатель также узнает много нового о плохо понимаемых возможностях C++, которые применяются в продвинутых программах и проектах.

В книге рассказывается, как миновать наиболее серьезные опасности, подстерегающие программиста на C++.

Издание предназначено для всех программистов, желающих научиться писать правильные и корректно работающие программы на языке C++.

Стандартная Библиотека Шаблонов STL

Стандартная библиотека шаблонов (STL) - набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++. Стандартная библиотека шаблонов существенно облегчает и ускоряет разработку. Применение высокоуровневых конструкций позволяет почти полностью забыть о кропотливой работе с памятью. Любой современный компилятор должен поддерживать возможности STL. Не стоит пренебрегать всей мощью и возможностями, которые может предложить данная библиотека.

- *Николаи Йосуттис - C++. Стандартная библиотека*

Оригинальное название: The C++ Standard Library. A Tutorial and Reference
Автор: Николаи Йосуттис (Nicolai Josuttis)
Год издания: 2014
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 1136
ISBN: 978-5-8459-1837-6

Стандартная библиотека C++ содержит набор универсальных классов и интерфейсов, значительно расширяющих ядро языка C++. Однако эта библиотека не является самоочевидной. Для того чтобы полнее использовать возможности ее компонентов и извлечь из них максимальную пользу, необходим полноценный справочник, а не простое перечисление классов и их функций.

В данной книге описывается библиотека как часть нового стандарта ANSI/ISO C++ (C++11). Здесь содержится исчерпывающее описание каждого компонента библиотеки, включая его предназначение и структуру; очень подробно описываются сложные концепции и тонкости практического программирования, необходимые для их эффективного использования, а также ловушки и подводные камни; приводятся точные сигнатуры и определения наиболее важных классов и функций, а также многочисленные примеры работоспособных программ. Основным предметом изучения в книге является стандартная библиотека шаблонов (STL), в частности контейнеры, итераторы, функциональные объекты и алгоритмы.

В книге описаны все новые компоненты библиотеки, вошедшие в стандарт C++11, в частности:

Параллельная работа;
Арифметика рациональных чисел;
Часы и таймеры;
Кортежи;
Новые контейнеры STL;
Новые алгоритмы STL;
Новые интеллектуальные указатели;
Случайные числа и распределения;
Свойства типов и утилиты;
Регулярные выражения;

В книге также рассматривается новый стиль программирования на C++ и его влияние на стандартную библиотеку, включая лямбда-функции, диапазонные циклы for, семантику перемещения и вариативные шаблоны.

- *Яцек Галовитц - C++17 STL - Стандартная библиотека шаблонов*

Оригинальное название: C++17 STL Cookbook
Автор: Яцек Галовитц (Jacek Galowicz)
Год издания: 2018
Издательство: Питер
Переплёт: мягкий
Количество страниц: 432
ISBN: 978-5-4461-0680-6

C++ — объектно-ориентированный язык программирования, без которого сегодня немыслима промышленная разработка ПО. В этой замечательной книге описана работа с контейнерами, алгоритмами, вспомогательными классами, лямбда-выражениями и другими интересными инструментами, которыми богат современный C++. Освоив материал, вы сможете коренным образом пересмотреть привычный подход к программированию. Преимущество издания — в подробном описании стандартной библиотеки шаблонов C++, STL. Ее свежая версия была выпущена в 2017 году. В книге вы найдете более 90 максимально реалистичных примеров, которые демонстрируют всю мощь STL. Многие из них станут базовыми кирпичиками для решения более универсальных задач. Вооружившись этой книгой, вы сможете эффективно использовать C++17 для создания высококачественного и высокопроизводительного ПО, применимого в различных отраслях.

- *Дэвид Мюссер, Атул Сейни - C++ и STL. Справочное руководство*

Оригинальное название: STL Tutorial and Reference Guide: C++ Programming
Авторы: Дэвид Р. Мюссер (David R. Musser), Атул Сейни (Atul Saini)
Год издания: 2010
ISBN: 978-5-8459-1665-5, 978-0-321-70212-8

Написанная авторами, принимавшими участие в разработке и практическом применении STL, данная книга представляет собой полное справочное руководство по данной теме. Она включает небольшой учебный курс, подробное описание каждого элемента библиотеки и большое количество примеров.

В книге вы найдете подробное описание итераторов, обобщенных алгоритмов, контейнеров, функциональных объектов и т.д. Ряд нетривиальных приложений демонстрирует использование мощи и гибкости STL в повседневной работе программиста. Книга также разъясняет, как интегрировать STL с другими объектно-ориентированными методами программирования. Она будет вашим постоянным спутником и советчиком при работе над проектами любой степени сложности. Во втором издании отражены все самые последние изменения в STL на момент написания книги; в нем появились новые главы и приложения. Множество новых примеров иллюстрируют отдельные концепции и технологии; большие демонстрационные программы показывают, как использовать STL в реальной разработке приложений на языке программирования C++.

- *Скотт Мейерс - Эффективное использование STL*

Оригинальное название: Effective STL
Автор: Скотт Мейерс (Scott Meyers)
Год издания: 2002
ISBN: 5-94723-382-7

Библиотека STL (Standard Template Library) произвела настоящий переворот в программировании C++, но ее освоение традиционно считалось весьма сложной задачей. К счастью, ситуация изменилась. В этой книге известный автор и программист Скотт Мейерс раскрывает секреты мастерства, позволяющие добиться максимальной эффективности при работе с этой библиотекой. В книге приводится множество рекомендаций и приемов работы в STL. Эти рекомендации подкреплены подробным анализом и убедительными примерами, поэтому читатель легко узнает, как решить ту или иную задачу и принять верное решение.

Объектно-ориентированное программирование

Появление в ООП отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП - это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода - если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки полиморфизма - возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

В следующих книгах рассматривается объектно-ориентированное программирование с точки зрения C++.

- *Роберт Лафоре - Объектно-ориентированное программирование в C++*

Оригинальное название: Object-Oriented Programming in C++
Автор: Роберт Лафоре (Robert Lafore)
Год издания: 2018
Издательство: Питер
Переплёт: твёрдый
Количество страниц: 928
ISBN: 978-5-4461-0927-2, 978-5-496-00353-7, 0-672-32308-7

Благодаря этой книге тысячи пользователей овладели технологией объектно-ориентированного программирования в C++. В ней есть все: основные принципы языка, готовые полномасштабные приложения, небольшие примеры, поясняющие теорию, и множество полезных иллюстраций.

Книга пользуется стабильным успехом в учебных заведениях благодаря тому, что содержит более 100 упражнений, позволяющих проверить знания по всем темам.

Читатель может вообще не иметь подготовки в области языка C++. Необходимо лишь знание начальных основ программирования.

- *Гради Буч - Объектно-ориентированный анализ и проектирование с примерами приложений*

Оригинальное название: Object-Oriented Analysis and Design with Application
Авторы: Гради Буч (Grady Booch), Роберт А. Максимчук (Robert A. Maksimchuk), Майкл У. Энгл (Michael W. Engle), Бобби Дж. Янг (Bobbi J. Young), Джим Коналлен (Jim Conallen), Келли А. Хьюстон (Kelli A. Houston)
Год издания: 2010
Издательство: Вильямс
Переплёт: твёрдый
Количество страниц: 720
ISBN: 978-5-8459-1401-9, 0-201-89551-X

Авторы описывают объектные методы решения сложных проблем, связанные с разработкой систем и программного обеспечения. Используя многочисленные примеры, они иллюстрируют основные концепции объектно-ориентированного подхода на примере разработки систем управления, сбора данных и искусственного интеллекта. Читатели найдут в книге практичные советы, касающиеся важных вопросов анализа, проектирования, реализации и оптимального управления проектами.

Книга будет полезна системным аналитикам и архитекторам, программистам, преподавателям и студентам высших учебных заведений, а также всем специалистам по информационным технологиям.