

|  |
| --- |
| **«ТОО «НПО «Группа Компаний «DOSTI»** |
| **Современное программирование на C/C++** |
| Целью метакурса является необходимость правильного обучения программирования на языках C/C++ в версиях от 98 до 2X, преимущественно для студентов и слушателей образовательных учреждений. |

|  |
| --- |
| **Мольганов А.А.**  24.6.2022 |

**СОДЕРЖАНИЕ**

|  |  |
| --- | --- |
| **Предисловие** |  |
| **Немного истории или краткое вводное слово о C/C++** |  |
| **Настройка программного окружения** |  |
|  |  |
| **Раздел I. Язык программирования C** |  |
| **Часть 1. Введение в программирование** |  |
| Глава 1.1. Основные возможности языка |  |
| Глава 1.2. Компиляция проектов (+3 глава) |  |
| Глава 1.3. Структура памяти процесса (+ 5 глава) |  |
| **Часть 2. Немного ООП** |  |
| Глава 2.1. Введение в ООП |  |
| Глава 2.2. Композиция данных |  |
| Глава 2.3. Наследование |  |
| Глава 2.4. Абстракция данных |  |
| **Часть 3. Взаимодействие с операционной системой** |  |
| Глава 3.1. Внутренняя архитектура Unix |  |
| Глава 3.2. Системные вызовы |  |
| **Часть 4. Нововведения** |  |
| Глава 4.1. Конкурентность |  |
| Глава 4.2. Введение в синхронизацию |  |
| Глава 4.3. Многопоточное программирование |  |
| Глава 4.4. Синхронизация потоков |  |
| Глава 4.5. Процессы |  |
| Глава 4.6. Синхронизация процессов |  |
| Глава 4.7. Сокеты и программирование |  |
| Глава 4.8. Межязыковое взаимодействие |  |
| Глава 4.9. Тестирование и отладка кода |  |
| Глава 4.10. Автоматизированная сборка |  |
|  |  |
| **Раздел II. Язык программирования C++** |  |
| Часть 1. |  |
|  |  |
|  |  |
| **Взгляд в будущее** |  |
|  |  |
| **Сборник задач** |  |
| **Список использованной литературы** |  |
| **Список использованных графических элементов** |  |

**ПРЕДИСЛОВИЕ**

**Дорогие студенты!**

В ваших руках пособие, созданное для получения практических знаний и умений по программированию и алгоритмизации. Алгоритмизация, а затем и программирование стали одними из областей математики в период научно-технической революции в середине XX века. Изначально, данные области науки использовались и продолжают использоваться как решение самых трудных фундаментальных и прикладных математических задач.

Однако, со временем алгоритмизация и программирование перестали быть чисто математическими областями науки и стали полноправными участниками современной вычислительной и информационно-коммуникационной картины современного мира.

В данном пособие и материалах к нему, упор сделан на практическое программирование на языках C и C++, а также использование современных методик, программного обеспечения и утилит. Данное пособие состоит из двух частей – программирование на языке C и программирование на языке C++.

Мы надеемся, что вы будьте усердно и честно выполнять задания, а также придерживаться академического кодекса. Списывание заданий и выдача чужой работы за свою – очень тяжкое образовательное преступление, так как этим вы лишаетесь практических знаний и что самое главное – своего времени и времени того человека, у которого вы списали, а также делаете труд преподавателя наименьшим, ведь в таком случае преподаватель потратил свое время впустую.

Повествование книги ведется в свободной манере общения, язык повествования относительно легкий, все термины, абстракции и листинги с программным кодом будут подробно объяснены.

Надеемся, что на каждый непонятный для вас термин, вопрос, задачу или тезис вы сможете задать вопрос и получить качественный ответ, который будет способствовать получению практических знаний!

*Желаем Вам успехов в этом увлекательном, интересном и веселом процессе!*

**НЕМНОГО ИСТОРИИ**

***или краткое вводное слово о C/C++***

Язык программирования C++ был первым языком, с которым автор метакурса стал общаться на «ТЫ» с компьютером. Тогда же, в период летних каникул между восьмым и девятым классом, автор наконец-таки понял, что языки программирования таят в себе безграничную силу. Тогда еще не было так модно понятие «*войти в IT*», поэтому на всех IT-специалистов смотрели как на древнегреческих богов и старались хоть немного быть похожими на них.

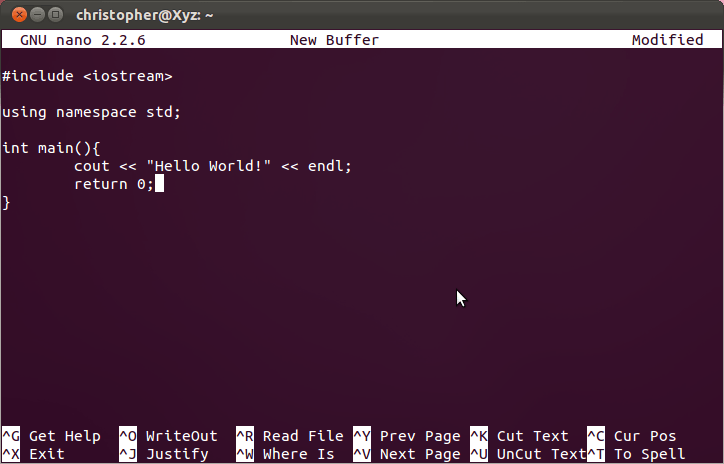


Рисунок 1.1.1 – Редактор GNU Nano

Мотивация выбора языка программирования C++ была очень проста – такой синтаксис позволял в дальнейшем без труда освоить языки более высокого уровня, например C# или Java, но

при этом C++ также является очень легким языком программирования по сравнению с C.

Но для лучшего понимания самого процесса программирования на C++, следует сначала изучить его предка – язык C. Язык C является очень простым, но одновременно и одним из самых сложных языков программирования, ведь шанс сделать ошибку намного выше, чем в других языках программирования.

C стал своего рода одним из первых языков программирования общего назначения. На момент его изобретения, в 1970-х годах, абсолютно все языки программирования делились на две группы – низкоуровневые (в основном символьная обертка над машинными кодами – прим. *Assembler*) и высокоуровневые (в основном научные языки программирования – прим. *Fortran, LISP*).

Первая группа языков программирования сочетала в себе быстроту выполнения программ, но при этом требовала от программиста высокого уровня знания архитектуры компьютера, в особенности памяти.

Вторая группа языков программирования сочетала в себе человекоподобную лексику и расширенные возможности (прим. – возможности машинных языков ограничивались архитектурными возможностями компьютеров того времени), но при этом требовали огромное количество памяти для хранения программ.

Сторонники двух групп постоянно соперничали друг с другом в разных сферах – промышленное, научное, прикладное, системное программирование, упорно доказывая друг другу только положительные стороны их «идеологии».



Рисунок 1.1.2 – Главный корпус MIT

В 1963 году в MIT (*Massachusetts Institute of Technology*, с англ. – *Массачусетский технологический институт, город Кембридж, штат Массачусетс, США*) совместно с Bell Telephone Laboratories и General Electric был начат научно-исследовательский проект под названием MAC (*Multiple Access Computer, Machine-Aided Cognition, Man and Computer*, с англ. – *Компьютер с множественным доступом, машинное познание, человек и компьютер*).

Проект представлял собой унификацию и развитие самого понятия «компьютер» как единой системы для самых различных людей, проводились также эксперименты по созданию различного опыта взаимодействия между человеком и компьютером.

В рамках данного проекта также была разработана операционная система Multics (*Multiplexed Information and Computing Service*, с англ. – *Мультиплексная информационно-вычислительная служба*) в 1965 году, которая сделала упор на выполнение задач с разделением времени. Однако позже, Bell Telephone Laboratories вышли из проекта по финансовым причинам.

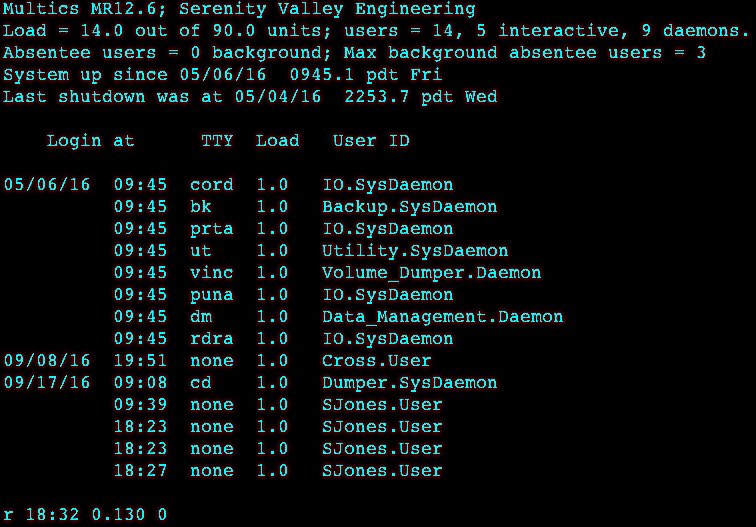


Рисунок 1.1.3 – Интерфейс операционной системы Multics OS

Некоторые сотрудники (*Кен Томпсон, Деннис Ритчи, Стью Фельдман, Дуг МакИлрой, Боб Моррис, Джо Оссанна*) Bell Telephone Laboratories, захотели продолжить работу самостоятельно. В то время, сэр Томпсон работал над игрой Space Travel для компьютера GE-635 компании General Electric. Данная видеоигра была сначала написана для операционной системы Multics, а затем была переписана на язык Fortran под экспериментальную операционную систему GECOS.

Однако при портировании игры Space Travel, аппаратное и программное обеспечение не было готово для полноценного запуска игры, написанной на языке Fortran из-за системных требований самого языка Fortran. Сэр Томпсон нашел альтернативу в виде портировании игры под популярный в то время PDP-7, один из самых лучших компьютеров 1960-х годов в соотношении цена/качество/производительность.

Томас и Ритчи полностью создали игру на кросс-ассемблере GE (прим. – для последующего портирования игры под разные аппаратные и программные платформы), а затем переносили программный код на перфоленты. Томпсону это активно не нравилось, и он начал писать операционную систему для PDP-7, начиная с файловой системы, считая, что перфоленты «доживают» свой век. Таким образом появилась легендарная операционная система UNIX, основа любой современной операционной системы.

|  |  |
| --- | --- |
|  |  |
| Рисунок 1.1.4 – PDP-7 и PDP-11 | |

UNIX также положил начало высокоуровневому языку B, который создавался под влиянием языка BCPL (прим. – один из языков программирования середины 1960-х годов). Деннис Ритчи сказал, что В — это C без типов. BCPL помещался в 8 Кб памяти и был тщательно переработан Томпсоном. В постепенно вырос в С.

К 1973 году язык C стал достаточно силён, и большая часть ядра UNIX, первоначально написанная на ассемблере PDP-11/20, была переписана на C. Это было одно из самых первых ядер операционных систем, написанное на языке, отличном от ассемблера.

Таким образом, язык C является побочным продуктом более грандиозного проекта под названием UNIX.

----------------------------------------------------------------------------

… (прошло 10 лет) …

----------------------------------------------------------------------------

Спустя почти десять лет, примерно в начале 1980-х годов, микроэлектроника превратилась из увлечения военных, космоса и государства в видеоигры и консольные приставки, аркадные автоматы, видео и аудио плееры, появились первые домашние компьютеры, впоследствии названные персональными. Количество компьютеров стало расти, а проект ARPANET (прим. – *первая компьютерная сеть*) от DARPA (*Defense Advanced Research Projects Agency*, с англ. – *Управление перспективных исследовательских проектов Министерства обороны США*) стал привлекать все больше внимания, что позже стало «всемирной паутиной» или Интернетом.

Работая над теорией очередей (прим. – *один из разделов математической теории вероятностей, целью которого является создание рациональной системы обслуживания на основе изучения потоков требования на обслуживание*) и моделируя распределение вызовов по автоматической телефонной станции, Бьёрн Страуструп из Bell Labs столкнулся с серьёзной проблемой. Язык программирования BCPL и его потомки работают быстро, но из-за близости к низкоуровневым языкам не подходят для разработки больших программ. С другой стороны, языки на основе Алгола или Fortran имеют все нужные функции, но работают очень медленно.

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгол-60** | 'BEGIN'  'COMMENT' Hello World in Algol 60;  OUTPUT(4,'(''('Hello World!')',/')')  'END' | BEGIN  OutText("Hello World!");  OutImage;  END | **Simula** |
| **Алгол-68** | ( # Hello World in Algol 68 # print(("Hello World!",newline))) | #include <stdio.h>  #include <stdlib.h>  int main(void)  {  puts("Hello World!");  return EXIT\_SUCCESS;  } | **C** |
| **BCPL** | GET "libhdr"  LET start() = VALOF  $( writes("Hello world\*N")  RESULTIS 0  $) | #include <iostream>  int main()  {  std::cout << "Hello World!" << std::endl;  } | **C++** |
| Листинг 1.1.5 – Hello World на различных языках программирования | | | |

Бьёрн решает скрестить языки первой со второй категории – дополнить С функциональностью основанного на Алголе языка Simula. C оказывается быстрым, легко портируемым и имеет много функций. Бьёрн добавляет классы, возможность их инкапсуляции и наследования, строгую проверку типов, встраиваемые функции и аргументы по умолчанию. Очень быстро «C с классами» получает собственное имя С++, где ++, – оператор приращения в С. Получившийся в результате язык годится для широкого применения, однако, остается таким же быстрым, как С, при этом имея расширенные возможности и функциональность, не уступающую другим языкам высокого уровня.

«C позволяет легко выстрелить себе в ногу; с C++ это сложнее, но, когда вы это делаете, ногу отрывает целиком», – Бьёрн Страуструп. "Выстрелить себе в ногу" — это идиома англоязычного мира разработчиков программного обеспечения, означающая причинение себе вреда необдуманными действиями без злого умысла. Языки C и C++ достаточно низкоуровневые языки, чтобы на них можно было делать опасные вещи. При этом C вообще никак не контролирует действия программиста и позволяет написать абсолютно любой код, а у C++ такое количество тёмных уголков, в которых можно «опасный» программный код сделать случайно, что не все знают, даже специалисты с десятилетиями опыта. Эти два совокупных фактора превращают программирование на C и C++ в изготовление приспособлений для стрельбы по собственным ногам.

Развитие интернета заставило использовать C++ для критичных серверных инфраструктур. В 1998 году был выпущен первый официальный стандарт языка C++ – «C++98». Стандарт состоял из ядра и стандартной библиотеки, которая включала в себя *Standard Template Library* (STL) и немного измененную стандартную библиотеку языка C.

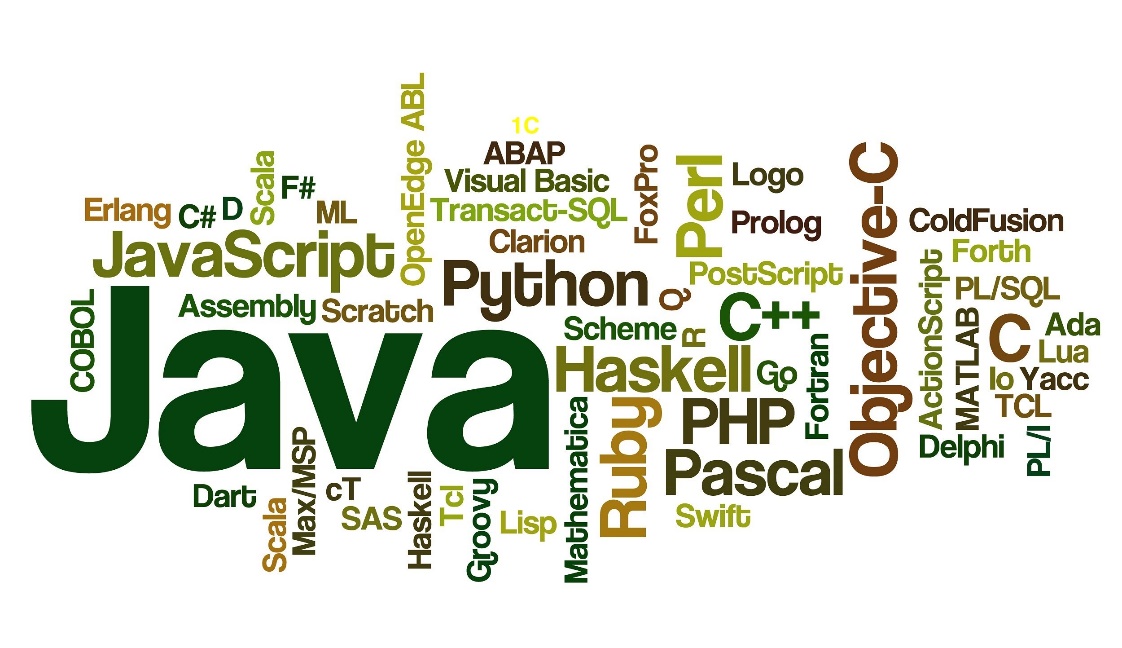


Рисунок 1.1.6 – Основные языки программирования

В 2000-е были созданы новые языки программирования – Java, C#, Go, D и другие. Однако, С++ стал расти не горизонтально, а в ширь – исправляя ошибки и сделав фокус на исправлении UB (*Undefined Behavior* – прим. *неопределенное поведение*) в ядре и библиотеках.

В 2003 году вышел C++03, который имел всего одну новую функцию – инициализацию значения, однако имел тонны исправлений ошибок, которые были сделаны в предыдущие года.

И тут язык взял непонятную никем паузу, пока в 2011 году не выходит C++11, в котором совершенствуется ядро языка, добавляется многопоточность, улучшается поддержка обобщенного программирования, унифицируются инициализации. Многие удобные вещи, которые были у языков-конкурентов, были добавлены только в C++11.

В 2015 году вышел C++14, который добавил следующие возможности: автоматическое определение типа значения, лямбда-выражения (блоки кода с параметрами, которые можно вызвать из любого другого места программы), шаблоны переменных, обособленные строки, пользовательские литералы для типов стандартной библиотеки и много чего еще.

В 2017 году вышел C++17, который добавил свертку параметров шаблона, авто-шаблоны, constexpr, constexpr-лямбды, inline-переменные, структурное связывание блоков кода, атрибуты, std::byte, std::filesystem, и другие возможности.

В 2020 году вышел C++20, который добавил следующие возможности: явные константы, концепты, модули, диапазоны, форматирование строк, библиотека синхронизации, прерывание потоков, using enum, частичный отказ от volatile, корутины (подпрограммы).

В начале 2021 года началась активная разработка стандарта C++23, вот лишь неполный список добавленных возможностей: std::stacktrace, улучшения стандарта, string::contains, executors (исполнители), networking (сетевая библиотека), метапрограммирование, рефлексия.

Автор же видит будущее C++ как смесь из функциональных языков программирования и общего назначения – Scala, Haskell и C#, Go. В настоящий момент (середина 2022 года), языки программирования стремятся к единению, целостности и созданию экосистемы вокруг себя, при этом благодаря сообществу и открытому программному обеспечению, люди со всего мира используют языки программирования в самых нестандартных условиях. Наверное, это и есть единение людей с одной общей благой целью для общества – сделать что-то лучше для общества с помощью ИТ-технологий ❤.

В пособии весь текст написан с помощью шрифта Segoe UI 12, а программные листинги написаны преимущественно на свободно распространяемом шрифте Fira Code 8.

**Глава 1. Создаем и запускаем 🚀**

Знакомство с языком программирования C++ вы начнете с настройки интегрированной среды разработки, которая представляет собой набор инструментов, позволяющих разрабатывать программное обеспечение на C++. Вы будете использовать интегрированную среду разработки Microsoft Visual Studio 2022 Community для компиляции первого консольного приложения на C++, программы, которую можно **запустить** из командной строки.

После первоначальной настройки вы изучите основные компоненты интегрированной среды разработки, а также роль, которую они играют в создании вашего первого приложения. В следующих главах будет описано достаточно основ языка программирования C++ для создания полезных программ и применение их в жизни.

Язык C++ имеют репутацию очень трудного языка программирования в обучении. Это действительно так, C++ – большой, сложный, амбициозный и очень противоречивый язык программирования.



Рисунок 1.1.1 – Microsoft Visual Studio 2022

**1.1. Структура базовой программы на C++**

В этом разделе мы напишем простую программу на C++, а затем скомпилируем и запустим её. Вы превратите исходный код C++ в удобные программы, которую можно будет запустить на любом компьютере. Для создания программ, первоначально требуется провести определенные действия с Microsoft Visual Studio 2022 Community:

**Шаг 1.** Скачайте и установите Microsoft Visual Studio 2022 Community, воспользовавшись следующей ссылкой: <https://visualstudio.microsoft.com/ru>

**Шаг 2.** Откройте Visual Studio Installer, и выберите необходимые компоненты для загрузки. Дождитесь окончания загрузки и установки необходимых компонентов (см. Рисунок 1.1.2):

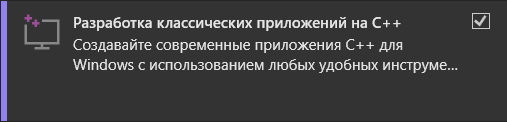


Рисунок 1.1.2 – Установка компонентов для C++

**Шаг 3.** После успешной установки, откройте Visual Studio 2022 и нажмите «**Создание проекта**» (см. Рисунок 1.1.3):

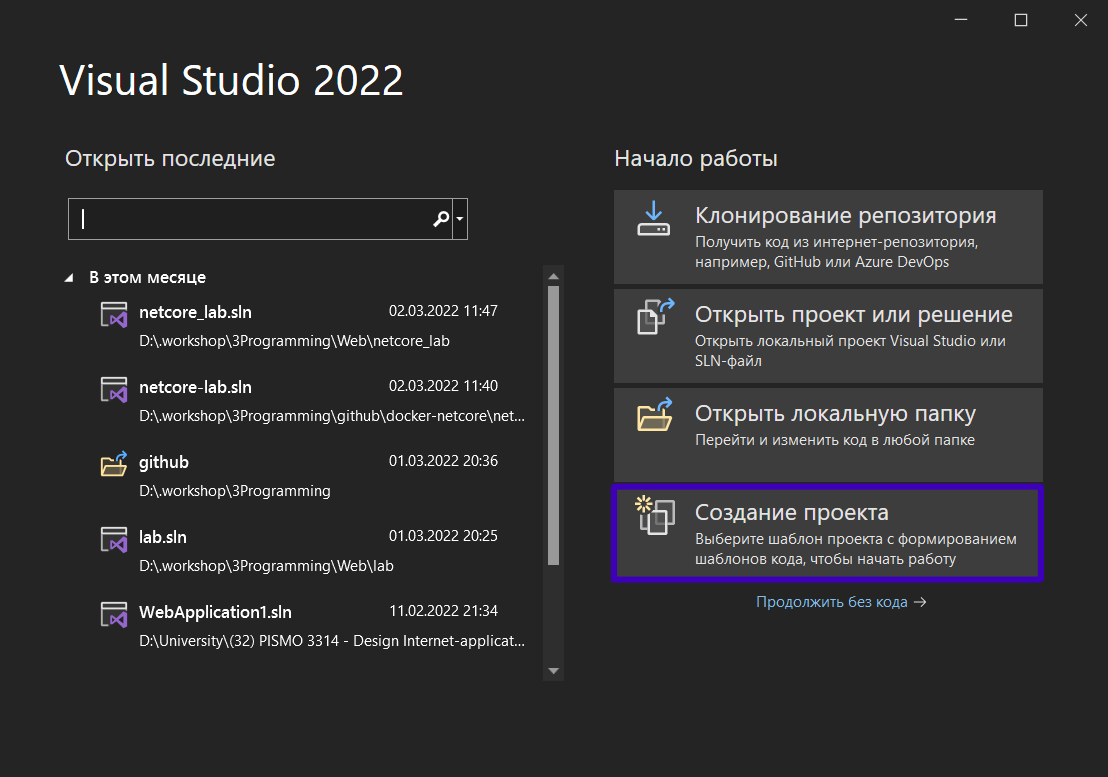


Рисунок 1.1.3 – Создание проекта

**Шаг 4.** Далее, на странице «**Создание проекта**», выберите тип проекта – «**Пустой проект**» с параметрами **C++, Windows, Консоль**. Данные параметры можно найти в меню под поиском. Можно также воспользоваться поиском для быстрого создания проекта по заготовленному шаблону (см. Рисунок 1.1.4):

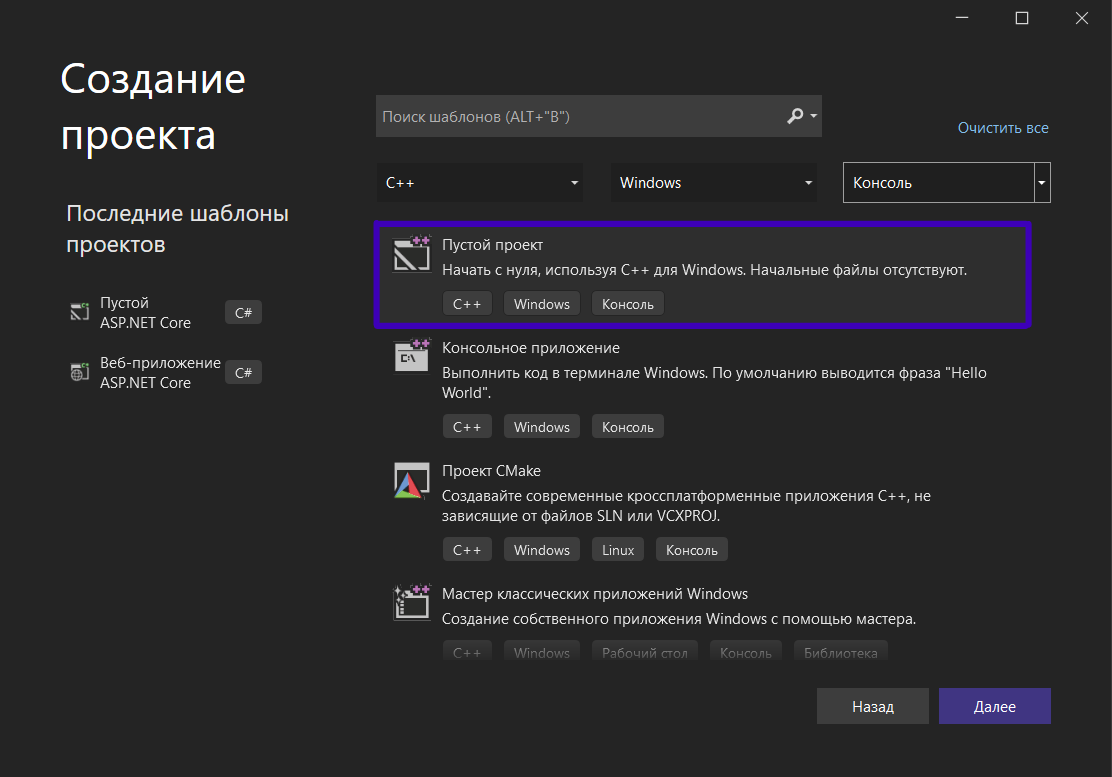


Рисунок 1.1.4 – Тип проекта

**Шаг 5.** После выбора типа проекта, откроется панель настройки проекта (см. Рисунок 1.1.5):

* Поле «**Имя проекта**» - название вашего проекта;
* Поле «**Расположение**» - путь, где будет храниться ваш проект;
* Поле «**Имя решения**» - не трогайте, поставьте галочку напротив текста «**Поместить решение и проект в одном каталоге**».

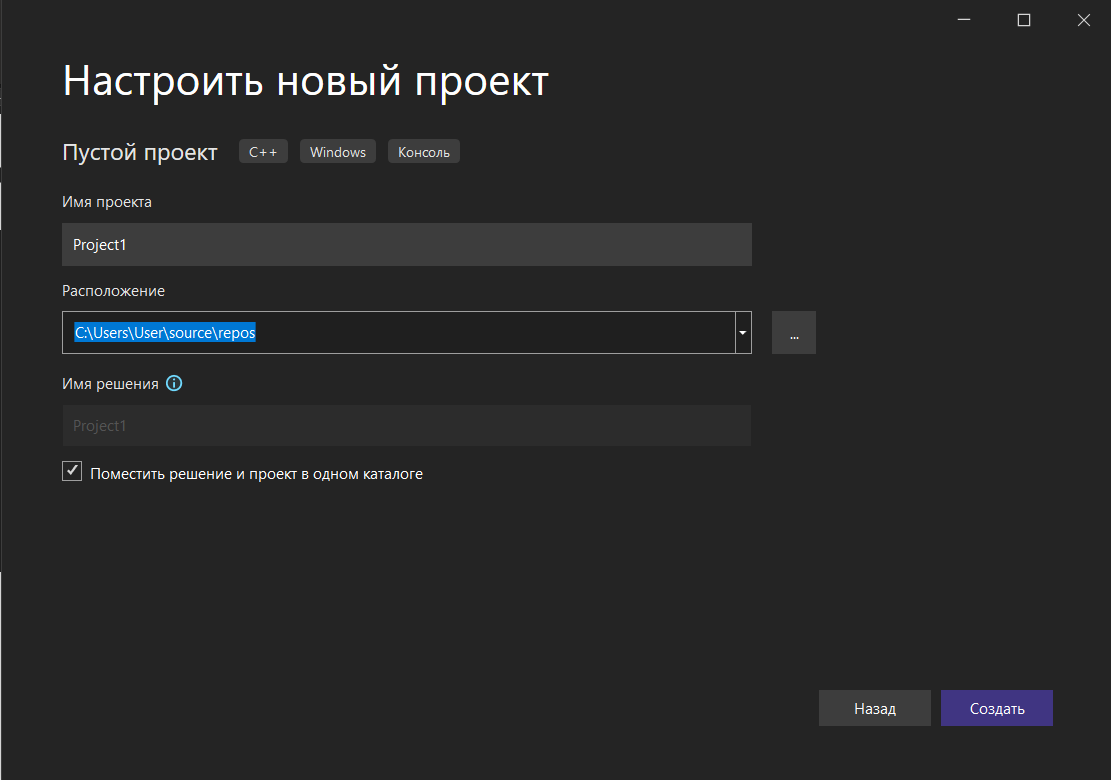


Рисунок 1.1.5 – Настройка проекта

**Шаг 6.** После создания проекта, необходимо добавить наш исходного файл, который будет компилироваться в исполняемую программу. Для этого, правой кнопкой мыши щелкните на папку «**Исходные файлы**», далее выберите меню «**Добавить**», подменю «**Создать элемент**» (см. Рисунок 1.1.6).

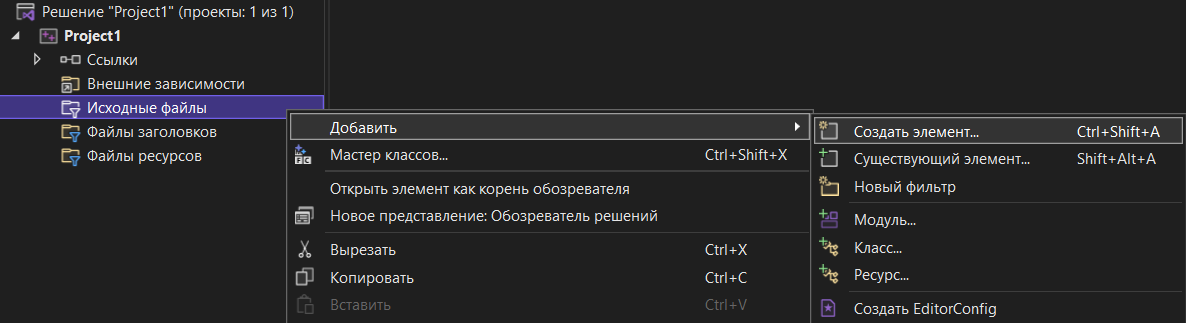


Рисунок 1.1.6 – Создание исходного файла

**Шаг 7.** В открывшемся меню, выберите «**Файл C++ (.cpp)**», выберите имя для файла – обычно это main.cpp. Нажмите кнопку «Добавить» (см. Рисунок 1.1.7).

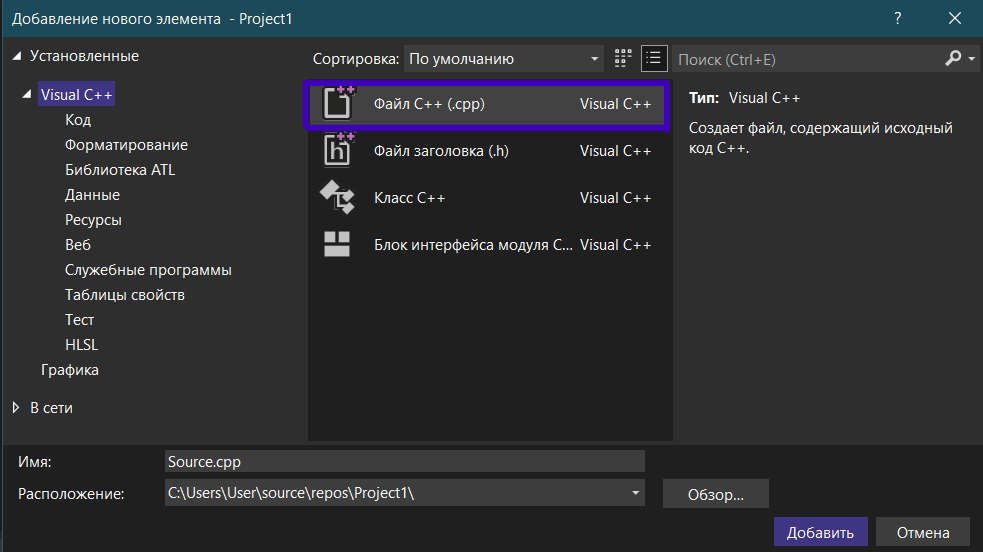


Рисунок 1.1.7 – Тип исходного файла

**Шаг 8.** Поздравляем! Вы успешно создали свой первый проект. В дальнейшем, все проекты будут созданы по такому же алгоритму.

**1.2. Немного о компиляции**

Ниже, в Листинге 1.2.1 представлен исходный код с нашей первой программой на C++. Введите данный код в наш только что созданный проект в Microsoft Visual Studio 2022, и сохраните его. Исходный файл будет иметь расширение .cpp. В данной книге, в нижней части листинга, будут находиться выходные данные, которые будут получены вами после компиляции и запуска вашей программы.

Листинг 1.2.1 – Первая программа на C++

#include <iostream>

**int** main (){

cout << "Hello, world!";

**return** 0;

}

-------------------------------------------------------------------

Hello, World!

Как показано в листинге 1.2.1, программы на C++, имеют единственную точку входа, называемую функций main. **Точка входа** – функция, которая выполняется, когда пользователь выполняет программу. **Функции** – блок кода, которые могут принимать входные данные, выполнять некоторые инструкции и возвращать результаты (выходные данные).

В main вы вызываете функцию cout, которая выводит символы Hello, world! в консоли. Затем программа завершается, возвращая код выхода () в операционную систему. **Коды входа** – целочисленные значения, которые операционная система использует для определения того, насколько хорошо работает программа. Как правило, код завершения ноль, означает что программа успешно запущена и функционирует. Наличие данного оператора возврата не обязательно, код выхода по умолчанию равен 0. Функция cout не определена в программе, она взята из библиотеки iostream.

**Библиотеки** – полезные наборы кода, которые можно импортировать в программы, чтобы избежать необходимости изобретать «велосипед». Все библиотеки – системные и пользовательские для подключения к проекту используют структуру #include. Системные библиотеки заключаются в знаки отношений <iostream>, а пользовательские библиотеки в кавычки “library.h”.

После написания исходного кода для программы на C++ следующий шаг – превратить исходный код в исполняемую программу. **Цепочка инструментов компилятора** (или цепочка инструментов) представляет собой набор из трех элементов, которые запускаются один за другим для преобразования исходного кода в программу.

1. **Препроцессор** – выполняет базовые операции с кодом. Например, библиотечные функции. Когда препроцессор заканчивает обработку исходного файла, он генерирует один блок перевода. Каждая единица трансляции затем передается компилятору для дальнейшей обработки.
2. **Компилятор** – читает единицу трансляции и генерирует объектный файл. **Объектный файл** – промежуточный формат, содержит код в виде промежуточного формата, которые не понять большинству людей. Каждая трансляция равняется каждому объектному файлу.
3. **Редактор связей** – генерирует программы из объектных файлов. Редакторы связей также несут ответственность за поиск библиотек, включенных в исходный код.

⚠ При создании проектов на C++, желательно выбирать самые новые инструменты и версии языка, а для нас именно это:

* Стандарт языка C++ – «Стандарт ISO C++ 20 (/std::c++20)»;
* Стандарт языка C – «Стандарт ISO C17 (2018) (/std::c17)»;
* Режим совместимости – «Да (/permissive-)»

**1.3. Инициализация в C++**

С++ – объектно-ориентированный язык программирования. **Объекты** – абстракции с **состоянием** и **поведением**. Пример из реальной жизни – выключатель света. У него есть свое состояние и поведение. Так вот, коллекция поведений и состояний, которые описывают объект, называются его **типом**. C++ также является **строго типизированным** языком программирования, то есть каждый объект имеет **предопределенный тип** данных. C++ имеет встроенный целочисленный тип под названием **int**. Объект **int** может хранить целые числа (состояния) и поддерживает множество математических операций (поведение).

Переменные объявляются путем предоставления типа, имени и точки с запятой. В следующей примере объявляется переменная hello с типом **int**. За типом **int** следует имя переменной hello.

Листинг 1.3.1 – Объявление целочисленной переменной

**int** hello;

**Переменные инициализируются при объявлении**. Инициализация объекта устанавливает начальное состояние объекта, например его значение. Для того, чтобы задать значение используем **знак равенства**.

Листинг 1.3.2 – Объявление целочисленной переменной со значением

**int** hello = 500;

После запуска этой строки кода, вы получите переменную со значением 5. Для того чтобы так сделать, мы используем операцию деления числа на число прямо в объявлении переменной. Однако так делать нельзя, и следующий пример будет являться тоном «**плохого кода**». Вторая строка в листинге 1.3.3, вычисляет значение второй переменной с помощью деления значения первой переменной на число 100.

Листинг 1.3.3 – Объявление целочисленной переменной со значением

**int** hello = 500;

**int** hello2 = hello / 100;

**Условные выражения** позволяют принимать решения в программах. Эти решения основаны на логических выражениях, которые оцениваются как true или false. Некоторые основные операторы сравнения, такие как «больше» или «не равно» работают с целочисленными переменными, например:

Листинг 1.3.4 – Основные операторы сравнения

**int** main() {

**int** x = 0;

42 == x; // Равенство

42 != x; // Неравенство

100 > x; // Больше

123 >= x; // Больше или равно

-10 < x; // Меньше

-99 <= x; // Меньше или равно

}

Эта программа ничего не возвращает, однако ее компиляция позволит нам убедиться в правильности написания программы на C++. Для создания **комментариев** в коде используется двойной слэш // комментарий, а для создания **многострочных комментариев** используется - \*/ комментарий /\*.

**Оператор if** содержит логическое выражение и один или несколько **вложенных операторов**. В зависимости от того, принимает ли выражение значение true или false, программа автоматически решает, какой вложенный **оператор** выполнить. Существует несколько форм операторов if, но в основном они используются следующим образом:

Листинг 1.3.5 – Объявление условного выражения if

**if** (логическое-выражение) оператор

Если логическое-выражение истинно, вложенный оператор выполняется, в противном случае – не выполняется. Иногда нам требуется использовать несколько операторов. Группы операторов называются **составными операторами**. Для этого используем фигурные скобки:

Листинг 1.3.6 – Составные операторы

**if** (логическое-выражение) {

оператор1;

оператор2;

}

Можно расширить оператор if, добавив операторы if и else. Это необязательные дополнения, но они помогут нам создать более **сложные конструкции управления**, например:

Листинг 1.3.7 – Составные операторы

**if** (логическое-выражение-1) оператор-1

**else** **if** (логическое-выражение-2) оператор-2

**else** оператор-3

Сначала выполняется логическое-выражение-1. Если логическое-выражение-1 равно true, выполняет оператор-1, и оператор if прекращает выполнение и оставшиеся конструкции не выполняются из-за логического подтверждения. Мы можем добавлять **любое количество выражений else** **if** или полностью исключить их. Рассмотрим специальный листинг, направленный на закрепление вами операторов управления.

Листинг 1.3.8 – Составные операторы

#include <iostream>

**int** main() {

**int** x = 0;

**if** (x > 0) cout << "Positive\n";

**else** **if** (x < 0) cout << "Negative\n";

**else** cout << "Zero\n";

}

-------------------------------------------------------------------

Zero

**Функции** – блоки кода, которые принимают любое количество входных объектов, которые называются **параметрами** или **аргументами** и могут возвращать **выходные объекты функциями**, вызвавшими их. Функции объявляются следующим образом:

Листинг 1.3.9 – Функции

возвращаемый-тип имя-функции (тип-параметра1 имя-параметра1, тип-параметра2

имя-параметра2) {

--пропуск--

**return** возвращаемое-значение;

}

Первая часть данного объявления функции – тип **возвращаемой переменной**, когда функция возвращает **значения**, тип этого значения должен соответствовать типу возвращаемого значения. После объявления возвращаемого типа объявляется **имя-функции**. Ряд скобок после имени функции содержит любое количество разделенных запятыми **входных параметров**, необходимых для **функции**. Каждый параметр также имеет **тип** и **имя**.

В листинге 1.3.9 параметры имеют название и тип параметра. Параметры представляют объекты, переданные в **функцию**. Фигурные скобки, следующие за списком, содержат **тело функции**. В рамках этой логики функция может решить **вернуть значение** функции вызывающему объекту. **Функции**, которые возвращают значения, обязательно должны **иметь один или несколько операторов возвращения**. Давайте посмотрим на функцию возвращения шага для различных значений.

Листинг 1.3.10 – Пример функции

**int** step\_function(**int** x) {

**int** result = 0;

**if** (x < 0) {

result = -1;

} **else** **if** (x > 0) {

result = 1;

}

**return** result;

}

Данная функция step\_function принимает один параметр – **int** x. Переменная result объявляется и инициализируется значением 0. Затем оператор if устанавливает result в -1, если x меньше 0. Если же x больше 0, то оператор if устанавливает result в 1. Наконец же, result возвращается вызывающему объекту.

Чтобы вызвать **функцию**, нужно использовать **имя** нужной функции, скобки и список обязательных параметров через **запятую**. Компилятор читает файлы **сверху вниз**, поэтому объявление функции должно появляться перед точкой ее первого использования. Например:

Листинг 1.3.11 – Передача аргументов в функцию

**int** step\_function(**int** x) {

// some code

}

**int** main() {

**int** value1 = step\_function(100); // значение1 равно 1

**int** value2 = step\_function(0); // значение2 равно 0

**int** value3 = step\_function(-10); // значение3 равно -1

}

С помощью различных переменных мы передаем значения этих переменных в функцию, а обратно можем получать результат вычисления. Давайте совместим листинг 1.3.10 и 1.3.11 и увидим, что можно сделать с помощью функций.

Листинг 1.3.12 – Полный код исходной программы

#include <iostream>

**int** step\_function(**int** x) {

**int** result = 0;

**if** (x < 0) {

result = -1;

} **else** **if** (x > 0) {

result = 1;

}

**return** result;

}

**int** main() {

**int** num1 = 42;

**int** result1 = step\_function(num1);

**int** num2 = 0;

**int** result2 = step\_function(num2);

**int** num3 = -32767;

**int** result3 = step\_function(num3);

cout << "Num1:" << num1 << "; " << "Result1:" << result1 << endl;

cout << "Num2:" << num2 << "; " << "Result2:" << result2 << endl;

cout << "Num3:" << num3 << "; " << "Result3:" << result3 << endl;

**return** 0;

}

-------------------------------------------------------------------

Num1: 42, Step: 1

Num2: 0, Step: 0

Num3: -32767, Step: -1

**1.4. Отладка нашей программы**

Одним из наиболее важных навыков для разработчика является эффективная **отладка**. Большинство интегрированных сред разработки имеют в своем составе **инструменты отладки**. Научиться правильно их использовать, значит облегчить себе жизнь, ведь нет ничего лучше, чем программа без ошибок.

Важное примечание, в языке программирования C++ существуют так называемые **undefined behavior (неопределенное поведение)**. Даже опытные разработчики, в том числе автор частенько попадают в такие ситуации, когда программа может вести себя не так, как изначально было задумано из-за особенностей языка программирования. **Неопределенное поведение** – поведение, которое может возникать в результате использования ошибочных программных конструкций или некорректных данных. **Неопределенное поведение** также может возникать в ситуациях, которые не описаны в стандарте языка. Иными словами, неопределенное поведение означает что угодно, что может произойти, начиная от простого указателя, заканчивая многопоточностью.

Очень важно применять инструменты отладки для тестирования своих программ.

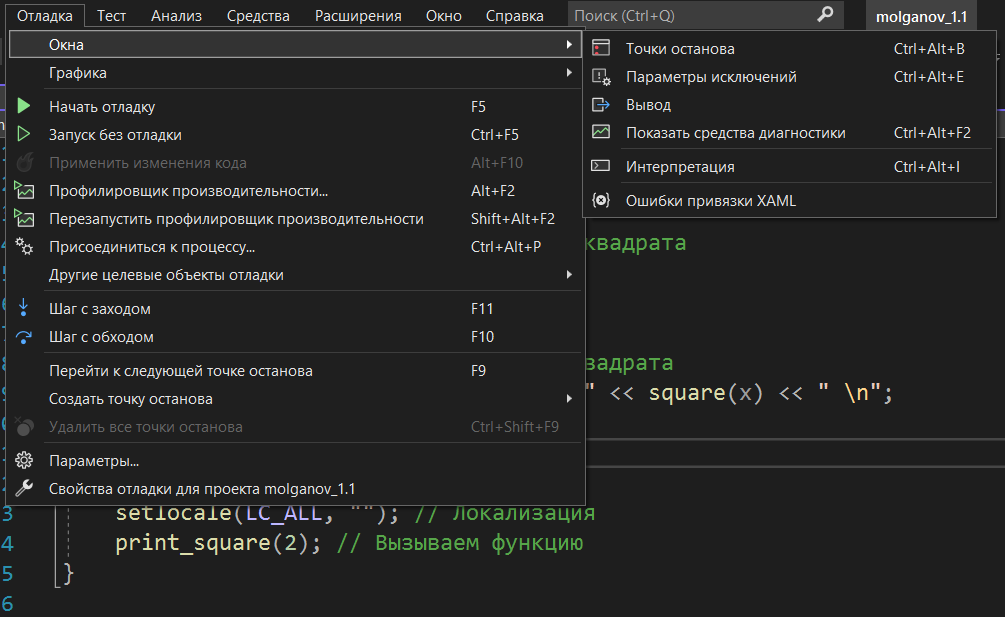


Рисунок 1.4.1 – Отладка в Microsoft Visual Studio 2022

Microsoft Visual Studio 2022 имеет в своем составе различные инструменты отладки, мы можем сразу же увидеть потребление памяти, составить отчет о производительности или построчно выполнить программу и найти ошибку в каждой строке и символе. С помощью инструментов диагностики можно выявить аномалии в работе приложения и посмотреть логи (сервисная информация) которые были созданы приложением во время работы.

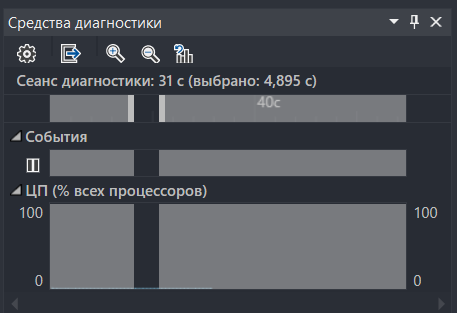


Рисунок 1.4.2 – Средства диагностики в Microsoft Visual Studio 2022

**Глава 2. Типы** ⚙

Как обсуждалось в Главе 1, тип определяет, как объект будет интерпретироваться и использовать компилятором. Каждый объект в программе имеет тип. Эта глава начинается с подробного обсуждения основных типов, а затем познакомит вас с пользовательскими типами.

**2.1. Основные типы**

**Основные типы** являются базовыми типами объектов и включают в себя **целые числа, числа с плавающей точкой, символы, логические значения и пустые значения**. Основные типы являются **фундаментом**, поэтому они будут работать на любой платформе и в любой операционной системе, но их функции – размер и расположение в памяти зависят от **реализации** программиста.

**Целочисленные типы** – хранят целые числа, те, что можно записать без запятой. Существуют четыре варианта целочисленных типов – short int, int, long int, long long int. Каждый из них может быть знакомы или беззнаковым. Знаковая переменная может быть положительной, отрицательной или нулевой, а беззнаковая переменная должна быть неотрицательной. В **таблице 2.1.1**, определены основные размеры и спецификаторы целочисленных типов.

Таблица 2.1.1 – Размеры целочисленных типов

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип** | **Знаковый** | **Размер в байтах** | | | |
| **32-битная ОС** | | **64-битная ОС** | |
| **Windows** | **\*nix** | **Windows** | **\*nix** |
| short | **Да** | 2 | 2 | 2 | 2 |
| unsigned short | **нет** | 2 | 2 | 2 | 2 |
| int | **Да** | 4 | 4 | 4 | 4 |
| unsigned int | **Нет** | 4 | 4 | 4 | 4 |
| long | **Да** | 4 | 4 | 4 | 8 |
| unsigned long | **Нет** | 4 | 4 | 4 | 8 |
| long long | **Да** | 8 | 8 | 8 | 8 |
| unsigned long long | **Нет** | 8 | 8 | 8 | 8 |

Обратите внимание, что размер в байтах в рамках одинакового типа на разных операционных системах не всегда одинаков! Это один из примеров UB (**Неопределенное поведение**), когда программист может перенести программу на другую операционную систему и с удивлением обнаружить что программа не корректно функционирует.

**Литерал** – жестко закодированной значение в программе. Можно использовать одно из четырех жестко запрограммированных целочисленных литеральных представлений – binary, octal, decimal, hexadecimal.

Если вы хотите точное **целое число со знаком** равное 8, 16, 32 или 64 битам, вы можете использовать int8\_t, int16\_t, int32\_t, int64\_t. Но данные числа доступны не на **всех платформах**, поэтому перед их использованием убедитесь в том, что на вашей целевой платформе где вы будет испытывать или выпускать в релиз приложение, будет доступна **поддержка данных чисел**. Подробнее об этом можно узнать из источников, которые были использованы при написании данного пособия.

**Целочисленные литералы** могут содержать любое количество одинарных кавычек (‘) для удобства чтения. Это полностью игнорируется компилятором. Например: 1000000 и 1’000’000 – одинаковые числа (миллион). Рекомендуем использовать данное преимущество для уменьшения проблем с читаемостью кода.

**⚠** Обращаем ваше внимание, что есть вы хотите использовать восьмеричный префикс, то будет очень осторожны. **Восьмеричный префикс** является пережитком языка программирования **B (1969 год)**. Языки C и C++ продолжают эту дурную традицию, поэтому следует **проявлять повышенную осторожность** при написании, например почтовых кодов. Язык программирования B очень сильно похож на C/C++.

Листинг 2.1.2 – Язык программирования B

main()

{

**auto** a, b, c, sum;

a = 1; b = 2; c = 3;

sum = a+b+c;

putnumb(sum);

}

По умолчанию тип целочисленного литерала – один из следующих: **int, long или long long.** Тип целочисленного литерала является **наименьшим** из подходящих из этих трех типов. Для большего контроля вы можете применять **суффиксы** к целочисленному литералу, чтобы указать его тип: unsigned – u (U), long – l (L), long long – ll (LL).

**Тип с плавающей точкой** – хранят целое число с частью после запятой – 0,33333 или 98,6. Устройство всех современных компьютеров не позволяет хранить точно число с плавающей точкой, однако компьютер хранит максимально возможное число с плавающей точкой. Например, число π, мы все знаем лишь 3,14, некоторые знают 3,14159. Компьютер хранит такие число точно также – не слишком точно.

Как и все типы, типы с плавающей точкой занимают **ограниченный объем памяти**, что называется точностью типа. Чем выше **точность**, тем точнее он будет приближен к **действительному числу**. Язык программирования C++предполагает три уровня точности (Таблица 2.1.3):

Таблица 2.1.3 – Точность типа с плавающей точкой

|  |  |  |
| --- | --- | --- |
| **Тип** | **Точность** | **Значение** |
| float | Одинарная точность | 3,1 |
| double | Двойная точность | 3,14 |
| long double | Повышенная точность | 3,1459 |

В распространённых операционных системах уровень float обычно имеет точность в **4 байта**. Уровни double и long double обычно имеют **8 байтов** точности (двойная точность). Мы опустим некоторые детали по типам с плавающей точкой, однако если вы заинтересованы научными вычислениями или вычислительными науками, то вам будет интересен стандарт IEEE 754.

Существуют литералы для типов с плавающей точкой – double (0.2), float (0.1F), long double (0.3L).

**Символьные типы** – хранят данные о естественном языке, в большинстве своем представляют символы, однако для строк существует другой специальный тип. Существует шесть типов символов (Таблица 2.1.4):

Таблица 2.1.4 – Символьные типы

|  |  |  |
| --- | --- | --- |
| **Тип** | **Использование** | **Пример** |
| char | Тип по умолчанию в 1 байт, может быть знаковый или беззнаковый | ASCII |
| char16\_t | 2-байтовые наборы символов | UTF-16 |
| char32\_t | 4-байтовые наборы символов | UTF-32 |
| signed char | char, но гарантированно знаковый | - |
| unsigned char | char, но гарантированно знаковый | - |
| wchar\_t | Имеет неограниченную память | Unicode |

**Символьный литерал** – один постоянный символ. Символьные литералы заключают в одинарные кавычки (‘literal’). Также существуют экранированные последовательности, которые можно использовать в работе с данными (Таблица 2.1.5). Символы Unicode можно представить как ‘\u0041’.

Таблица 2.1.5 – Экранированные последовательности

|  |  |
| --- | --- |
| **Значение** | **Последовательность** |
| Новая строка | \n |
| Табуляция (горизонтальная) | \t |
| Табуляция (вертикальная) | \v |
| Возврат на одну позицию | \b |
| Возврат каретки | \r |
| Прогон страницы | \f |
| Оповещение | \a |
| Обратная косая | \\ |
| Знак вопроса | ? или \? |
| Одинарная кавычка | \’ |
| Двойная кавычка | \” |
| Нулевой символ | \0 |

**Логические типы** – имеют два состояния – true (истина) или false (ложь). Единственный логический тип – bool. Целочисленный типы и типы bool легко преобразуются. Любое ненулевое целое число преобразуется в true, а 0 в false.

**Операторы сравнения** – функции, которые выполняют вычисления над операндами. Операнды – это просто объекты (Таблица 2.1.6).

Таблица 2.1.6 – Операторы сравнения

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Символ** | **Пример** | **Операция** |
| Больше | > | x>y | true, если x больше y, иначе false |
| Меньше | < | x<y | true, если x меньше y, иначе false |
| Больше или равно | >= | x>=y | true, если x больше/равно y, иначе false |
| Меньше или равно | <= | x<=y | true, если x меньше/равно y, иначе false |
| Равно | == | x==y | true, если x равно y, иначе false |
| Не равно | != | x!=y | true, если x не равно y, иначе false |

**Логические операторы** – вычисляют логические выражения для типов bool (Таблица 2.1.7).

Таблица 2.1.7 – Логические операторы

|  |  |  |  |
| --- | --- | --- | --- |
| **Операции** | **Обозначения** | **Условие** | **Описание** |
| И | && | a == 3 **&&**b > 4 | Составное условие истинно, если истинны оба простых условия |
| ИЛИ | || | a == 3 **||**b > 4 | Составное условие истинно, если истинно, хотя бы одной из простых условий |
| НЕ | ! | **!**( a == 3) | Условие истинно, если а не равно 3 |

**std::byte** – системные программисты иногда работают напрямую с необратимой памятью, которая представляет собой набор битов без типа. В таких ситуациях использовать тип std::byte. Данный тип разрешает побитовые логические операции и кое-что другое. Использование данного типа позволяет избежать общих трудных для отладки ошибок программирования.

Листинг 2.1.8 – Использование std::byte

#include <iostream>

#include <cstddef>

#include <bitset>

std::ostream& **operator**<< (std::ostream& os, std::byte b) {

**return** os << std::bitset<8>(std::to\_integer<**int**>(b));

}

**int** main()

{

std::byte b{42};

std::cout << "1. " << b << '\n';

// b \*= 2 compilation error

b <<= 1;

std::cout << "2. " << b << '\n';

b >>= 1;

std::cout << "3. " << b << '\n';

std::cout << "4. " << (b << 1) << '\n';

std::cout << "5. " << (b >> 1) << '\n';

b |= std::byte{0b11110000};

std::cout << "6. " << b << '\n';

b &= std::byte{0b11110000};

std::cout << "7. " << b << '\n';

b ^= std::byte{0b11111111};

std::cout << "8. " << b << '\n';

}

-------------------------------------------------------------------

1. 00101010

2. 01010100

3. 00101010

4. 01010100

5. 00010101

6. 11111010

7. 11110000

8. 00001111

**Sizeof** – оператор применяемый для определения размера переменной, которая занимает память.

**Void** – имеет пустой набор значений. Поскольку void-объект не может содержать значение, C++ запрещает void-объекты. Void-объекты используются в особых ситуациях, таких как тип возврата для функций, которые не возвращают никакого значения:

Листинг 2.1.9 – void-функция

#include <iostream>

**void** taunt() {

cout << "Hello, void function!"; }

**2.2. Массивы**

**Массивы** – последовательности переменных одинакового типа. Типы массивов включают в себя содержащийся тип и количество элементов в массиве. Эта информация объединена в синтаксисе объявления: тип элемента предшествует квадратным скобкам, в которых определяется размер массива.

Листинг 2.2.1 – Объявление массива

**int** MyArray[100]; // Массив из 100 элементов типа int

Существует быстрый способ для инициализации массивов со значениями с использованием **фигурных скобок**. Длину элементов массива можно опустить, она будет определена в процессе **компиляции**:

Листинг 2.2.2 – Объявление массива

**int** MyArray[] = { 1, 2, 3, 4 };

**Доступ к элементам** массива можно получить, используя **квадратные скобки,** в которые указывается нужны индекс. **Индексирование** массива начинается с **нуля**, поэтому первый элемент имеет индекс ноль, а десятый элемент имеет индекс 9.

Листинг 2.2.3 – Доступ к массиву

#include <iostream>

**int** main() {

**int** arr[] = { 1, 2, 3, 4 };

cout << "The third element is" << arr[2];

arr[2] = 100;

cout << "The third element is" << arr[2];

}

-------------------------------------------------------------------

The third element is 3

The third element is 100

**Цикл for** позволяет повторять выполнение **оператора указанное количество раз**. **Оператор инициализации** выполняется до прохождения первой итерации, поэтому в нем можно инициализировать переменные, используемые в цикле. **Условное выражение** – выражение, которое выполняет каждый раз перед использованием цикла. **Оператор цикла** выполняется после каждого раза.

Листинг 2.2.4 – Цикл For

**for**(оператор-инициализации; условное-выражение; оператор-цикла) {

// code

}

В циклах желательно используйте тип size\_t, так как оно сможет индексировать любое значение внутри него, а int – нет. На практике это не принесет большой выгоды, однако это правило хорошего кода и отличное техническое решение.

Оператор sizeof можно также использовать для получения общего размера массива в байтах, например:

Листинг 2.2.5 – Массивы и sizeof

**short** array[] = { 104, 105, 32, 98, 105, 108, 108, 0 };

size\_t n\_elements = **sizeof**(array) / **sizeof**(**short**);

**2.3. Пользовательские типы**

**Пользовательские типы** – типы, которые пользователь может определить самостоятельно. Существуют три категории пользовательских типов:

1. Перечисления – простейший из определяемых пользователем типов. Значения, которые может принимать перечисление, ограничены набором возможных значений. Перечисления отлично подходят для моделирования категориальных понятий;
2. Классы – более полнофункциональные типы, которые позволяют гибко объединять данные и функции;
3. Объединения – Специализированный пользовательский тип. Все члены объединения разделяют одну и ту же ячейку памяти. Объединения опасны, и ими легко злоупотреблять.

**Типы перечислений**. Перечисления объявляются при помощи ключевых слов enum class. Значения являются произвольными буквенно-цифровыми строками, они будут определять любые категории. Например:

Листинг 2.3.1 – Перечисления

**enum** **class** Race {

Dinan,

Teklan,

Ivyn,

Moiran,

Camite,

Julian,

Aidan

};

Для инициализации переменной перечисления значением используйте имя типа, за которым следуют два двоеточия :: и требуемое значение. Например, так можно объявить переменную langobard\_race и инициализировать ее значением Aidan.

Листинг 2.3.2 – Перечисления

Race langobard\_race = Race::Aidan;

**Оператор ветвления switch** передает управление одному из нескольких операторов в зависимости от значения условия, которое вычисляется как целое число или тип перечисления. Операторы switch обеспечивают условное ветвление. Когда выполняется оператор switch, управление переходит к случаю, соответствующему условию или к условию по умолчанию, если ни один случай не соответствует условному значению. Ключевое слово case обозначает случай, а ключевое слово default обозначает условие по умолчанию.

Выполнение будет выполняться до конца оператора switch или ключевого слова остановки вычисления break.

Листинг 2.3.2 – Оператор ветвления switch

#include <iostream>

**enum** **class** Race {

Dinan,

Teklan,

Ivyn,

Moiran,

Camite,

Julian,

Aidan

};

**int** main() {

Race race = Race::Dinan;

**switch**(race) {

**case** Race::Dinan: {

printf("You work hard.");

} **break**;

**case** Race::Teklan: {

printf("You are very strong.");

} **break**;

**case** Race::Ivyn: {

printf("You are a great leader.");

} **break**;

**case** Race::Moiran: {

printf("My, how versatile you are!");

} **break**;

**case** Race::Camite: {

printf("You're incredibly helpful.");

} **break**;

**case** Race::Julian: {

printf("Anything you want!");

} **break**;

**case** Race::Aidan: {

printf("What an enigma.");

} **break**;

**default**: {

printf("Error: unknown race!"); �'

}

}

}

-------------------------------------------------------------------

You work hard

**Глава 3. Ссылочные типы** 📇

Ссылочные типы хранят адреса памяти объектов. Эти типы обеспечивают эффективное программирование, многие из них имеют элегантный дизайн. В этой главе мы рассмотрим два вида ссылочных типов – указатели и ссылки.

**3.1. Указатели**

**Указатели** – механизм, используемый для обращения к адресам памяти. Указатели кодируют обе части информации, необходимые для взаимодействия с другим объектом – адрес и тип объекта. Указатели являются объектами очень низкого уровня. Указатели объявляются с помощью звездочки к указанному типу. Например:

Листинг 3.1.1 – Объявление указателей

**int**\* my\_ptr;

**Адрес переменной** можно получить, предварительно добавив **оператор вычисления** адреса (& - амперсанд). Работа с указателями и ссылками будет разной в разных операционных системах, на это влияет **разрядность операционной системы**. Например:

Листинг 3.1.2 – Адрес переменной

#include <iostream>

**int** main()

{

**int** a = 7;

std::cout << a << '\n'; // выводим значение переменной a

std::cout << &a << '\n'; // выводим адрес памяти переменной a

**return** 0;

}

**Оператор разыменования** – оператор, обращающийся к объекту, на который ссылается указатель. Это операция обратна операции вычисления адреса. По указанному адресу можно получить находящийся там **объект**. Однако, будьте внимательны, поскольку операторы **разыменования используется тоже \* (звездочку)** для своих действий, однако он ставит звездочку рядом с именем, а не типом. Например:

Листинг 3.1.3 – Оператор разыменования

#include <iostream>

**int** main()

{

**int** a = 7;

std::cout << a << '\n'; // выводим значение переменной a

std::cout << &a << '\n'; // выводим адрес переменной a

std::cout << \*&a << '\n'; /// выводим значение ячейки памяти переменной a

**return** 0;

}

**Указатели** имеют несколько общих характеристик с массивами. При малейшей провокации массив автоматические превращается в указатель. **Разрушенный массив** теряет информацию о размере и преобразуется в указатель на первый элемент массива. **Указатель** невозможно преобразовать в **массив**, что хорошо. Это вам никогда не понадобится, да и компилятор не сможет восстановить данные массива из указателя. Однако это не значит, что компилятор не может уберечь вас от всех опасных вещей. **Язык C++** полон неоднозначности и противоречивости, поэтому при программировании на этом языке стоит быть внимательным и не «**стрелять себе в ноги**». Например:

Листинг 3.1.4 – Указатели и массивы

**int** key\_to\_the\_universe[]{ 3, 6, 9 };

**int**\* key\_ptr = key\_to\_the\_universe; // Указывает на 3

**3.2. Ссылки**

**Ссылки** – безопасная и удобная версия указателя. Ссылки могут быть объявлены путем добавления (& – амперсанда) к имени типа. Ссылки не могут быть **присвоены нулю** и не могут быть повторно установлены. Например:

Листинг 3.2.1 – Ссылки

**int** value = 7; // обычная переменная

**int** &ref = value; // ссылка на переменную value

**Указатели и ссылки** в значительной степени взаимозаменяемы, но и те и другие имеют компромиссные решения, поэтому если в программе используются указатели или ссылки, то желательно продолжить «традицию» и не добавлять различных новых возможностей.

**Связный список** – простая структура данных, состоящая из ряда элементов. Каждый элемент содержит указатель на следующий элемент. Последний элемент содержит nullptr. Вставка элементов в список очень эффективна, и элемент могут иметь разрывы в памяти. Например:

Листинг 3.2.2 – Реализация связного списка

#include <iostream>

**using** **namespace** std;

**struct** Node {

string val;

Node\* next;

Node(string \_val) : val(\_val), next(**nullptr**) {}

};

**struct** list {

Node\* first;

Node\* last;

list() : first(**nullptr**), last(**nullptr**) {}

**bool** is\_empty() {

**return** first == **nullptr**;

}

**void** push\_back(string \_val) {

Node\* p = **new** Node(\_val);

**if** (is\_empty()) {

first = p;

last = p;

**return**;

}

last->next = p;

last = p;

}

**void** print() {

**if** (is\_empty()) **return**;

Node\* p = first;

**while** (p) {

cout << p->val << " ";

p = p->next;

}

cout << endl;

}

Node\* find(string \_val) {

Node\* p = first;

**while** (p && p->val != \_val) p = p->next;

**return** (p && p->val == \_val) ? p : **nullptr**;

}

**void** remove\_first() {

**if** (is\_empty()) **return**;

Node\* p = first;

first = p->next;

**delete** p;

}

**void** remove\_last() {

**if** (is\_empty()) **return**;

**if** (first == last) {

remove\_first();

**return**;

}

Node\* p = first;

**while** (p->next != last) p = p->next;

p->next = **nullptr**;

**delete** last;

last = p;

}

**void** remove(string \_val) {

**if** (is\_empty()) **return**;

**if** (first->val == \_val) {

remove\_first();

**return**;

}

**else** **if** (last->val == \_val) {

remove\_last();

**return**;

}

Node\* slow = first;

Node\* fast = first->next;

**while** (fast && fast->val != \_val) {

fast = fast->next;

slow = slow->next;

}

**if** (!fast) {

cout << "This element does not exist" << endl;

**return**;

}

slow->next = fast->next;

**delete** fast;

}

Node\* **operator**[] (**const** **int** index) {

**if** (is\_empty()) **return** **nullptr**;

Node\* p = first;

**for** (**int** i = 0; i < index; i++) {

p = p->next;

**if** (!p) **return** **nullptr**;

}

**return** p;

}

};

**int** main()

{

list l;

cout << l.is\_empty() << endl;

l.push\_back("3");

l.push\_back("123");

l.push\_back("8");

l.print();

cout << l.is\_empty() << endl;

l.remove("123");

l.print();

l.push\_back("1234");

l.remove\_first();

l.print();

l.remove\_last();

l.print();

cout << l[0]->val << endl;

**return** 0;

}

**3.3. Вывод типов с помощью auto**

Будучи строго типизированным языком, C++ предоставляет компилятору много информации. При инициализации элементов или возврате значения из функций компилятор может предугадывать информацию о типе из контекста. Ключевое слово auto указывает компилятору выполнить такое предсказание, освобождая вас от ввода избыточной информации о типе. Например:

Листинг 3.3.1 – Использование auto

**auto** the\_answer { 42 }; // int

**auto** foot { 12L }; // long

**auto** rootbeer { 5.0F }; // float

**auto** cheeseburger { 10.0 }; // double

**auto** politifact\_claims { **false** }; // bool

**auto** cheese { "string" }; // char[7]

Обычно в auto добавляются такие модификаторы, как &, \* и const. Такие модификации добавляют предполагаемые значения (ссылка, указатель):

Листинг 3.3.2 – Auto и ссылочные типы

**auto** year { 2021 }; // int

**auto**& year\_ref = year; // int&

**const** **auto**& year\_cref = year; // const int&

**auto**\* year\_ptr = &year; // int\*

**const** **auto**\* year\_cptr = &year; // const int\*

**Глава 4. Выражения 🌌**

**Выражения** — это вычисления, которые дают результаты и побочные эффекты. Как правило, выражения содержат операнды и операторы, которые работают с ними. Многие операторы включены в основной язык, и вы увидите большинство из них в этой главе.

**4.1. Операторы**

**Операторы** работают с аргументами, называемыми **операндами**. Существует несколько типов операторов: **логические, арифметические, присваивания, увеличения/уменьшения, сравнения, доступа к элементам, тернарные условные операторы и операторы запятой**.

**Логические операторы** – выполняют логические функции, согласно законам дискретной математики (Таблица 4.1.1).

Таблица 4.1.1 – Логические операторы

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Имя** | **Пример** | **Результат** |
| x & y | Побитовое И | 0b1100 & 0b101 | 0b1000 |
| x | y | Побитовое ИЛИ | 0b1100 | 0b101 | 0b1110 |
| x ^ y | Побитовое исключающее ИЛИ | 0b1100 ^ 0b1010 | 0b0110 |
| ~x | Побитовое дополнение | ~0b1010 | 0b0101 |
| x << y | Побитовый сдвиг влево | 0b1010 << 2 0b0011 << 4 | 0b101000  0b110000 |
| x >> y | Побитовый сдвиг вправо | 0b1010 >> 2 0b10110011 >> 4 | 0b10  0b1011 |
| x && y | И | true && false true && true | false  true |
| x | | y | ИЛИ | true | | false  false | | false | true  false |
| !x | НЕ | ! true  ! false | false  true |

**Арифметические операторы** – выполняют арифметические функции (Таблица 4.1.2)

Таблица 4.1.2 – Логические операторы

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Имя** | **Пример** | **Результат** |
| +x | Унарный плюс | +10 | 10 |
| -x | Унарный минус | -10 | -10 |
| x + y | Бинарное сложение | 1 + 2 | 3 |
| x – y | Бинарное вычитание | 1 – 2 | -1 |
| x \* y | Бинарное умножение | 10 \* 20 | 200 |
| x / y | Бинарное деление | 300 / 15 | 20 |
| x % y | Бинарное деление по модулю | 42 % 5 | 2 |

**Операторы присваивания** – сочетают в себе сразу две функции – вычисление результат и присваивание (Таблица 4.1.3).

Таблица 4.1.3 – Операторы присваивания

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Имя** | **Пример** | **Результат** |
| x = y | Простое присваивание | x = 10 | 10 |
| x += y | Присваивание сложения | x += 10 | 15 |
| x -= y | Присваивание вычитания | x -= 10 | -5 |
| x \*= y | Присваивания умножения | x \*= 10 | 50 |
| x /= y | Присваивания деления | x /=2 | 2 |
| x %= y | Присваивания деления по модулю | x %= 2 | 1 |
| x &= y | Присваивая побитового И | x &= 0b1100 | 0b0100 |
| x |= y | Присваивание побитового ИЛИ | x |= 0b1100 | 0b1101 |
| x ^= y | Присваивания побитового исключающего ИЛИ | x ^= 0b1100 | 0b1001 |
| x <<= y | Присваивание побитового сдвига влево | x <<= 2 | 0b10100 |
| x >>= y | Присваивание побитого сдвига вправо | x >>= 2 | 0b0001 |

**Операторы увеличения и уменьшения** – уменьшают или увеличивают значение переменной (Таблица 4.1.4)

Таблица 4.1.4 – Операторы увеличения и уменьшения

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Имя** | **Пример** | **Результат** |
| ++x | Префиксное увеличение | 6 | 6 |
| x++ | Постфиксное увеличение | 6 | 5 |
| --x | Префиксное уменьшение | 4 | 4 |
| x-- | Постфиксное уменьшение | 4 | 5 |

**Операторы сравнения** – сравнивают элементы (Таблица 4.1.5)

Таблица 4.1.5 – Операторы сравнения

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Имя** | **Результат** |
| x == y | Оператор равенства | 100 == 100 |
| x != y | Оператор неравенства | 100 != 101 |
| x < y | Оператор меньше | 10 < 20 |
| x > y | Оператор больше | -10 > -20 |
| x <= y | Оператор меньше или равно | 10 <= 10 |
| x >= y | Оператор больше или равно | 20 >= 10 |

**4.2. Инструкции-выражения и составные операторы**

**Инструкция-выражение** – выражение, за которым следует точка с запятой. Такие инструкции составляют большинство операторов в программе. Любой исходный код в программе имеет инструкцию-выражение, так как без них невозможно нормальное функционирование программы.

**Составные операторы** – блоки, которые представляют собой последовательность операторов, заключенных в фигурные скобки. Такие операторы полезны в управляющих структурах, потому что может понадобиться выполнить сразу несколько операторов вместо одного.

Листинг 4.2.1 – Составные операторы

**int** add(**int** x, **int** y)

{ // начало блока

**return** x + y;

} // конец блока

**int** main()

{ // начало блока

// Несколько блоков

**int** value(0);

add(3, 4);

**return** 0;

} // конец блока (без точки с запятой)

**4.3. Операторы выбора и перебора**

**Операторы выбора** выражают условный поток управления. Существуют два варианта операторов выбора – if и switch. **Оператор выбора if** проверяет соответствует ли выражение в скобках логической истине, если соответствует, то выполняется первая строка, если же не соответствует, то выполняется строка, следующая за **else**. **Оператор if** имеет уже знакомую нам форму, показанную в листинге 4.3.1:

Листинг 4.3.1 – Оператор выбора if

#include <iostream>

**int** main()

{

std::cout << "Enter a number: ";

**int** a;

std::cin >> a;

**if** (a > 15)

std::cout << a << " is greater than 15\n";

**else**

std::cout << a << " is not greater than 15\n";

**return** 0;

}

**Оператор switch** используется для управления программой, если структуры оператора if не хватило. Если логические условий больше трех, рекомендуется использовать оператор switch, которые более гибок и универсален в разработке программ, а также их обслуживании и поддержке.

Листинг 4.3.2 – Оператор выбора switch

#include <iostream>

**using** **namespace** std;

**int** main(**int** argc, **char**\* argv[])

{

**int** count; // переменная для выбора в switch

**double** a,b; // переменные для хранения операндов

cout << "Enter first number: ";

cin >> a;

cout << "Enter second number: ";

cin >> b;

cout << "Get action: 1 +; 2 -; 3 x; 4 /";

cin >> count;

**switch** (count) // начало оператора switch

{

**case** 1: // если count = 1

{

cout << a << " + " << b << " = " << a + b << endl; // выполнить сложение

**break**;

}

**case** 2: // если count = 2

{

cout << a << " - " << b << " = " << a - b << endl; // выполнить вычитание

**break**;

}

**case** 3: // если count = 3

{

cout << a << " \* " << b << " = " << a \* b << endl; // выполнить умножение

**break**;

}

**case** 4: // если count = 4

{

cout << a << " / " << b << " = " << a / b << endl; // выполнить деление

**break**;

}

**default**: // если count равно любому другому значению

cout << "Incorrect input" << endl;

}

system("pause");

**return** 0;

}

**Цикл while** – создан для того, чтобы повторить одно и то же действие несколько раз подряд без остановки. В отличие от других структур он не управляет потоком, а наоборот, заставляет программу сделать определенное действие столько раз, сколько надо программисту или другим программам. **Существует также цикл do-while**, однако условный оператор вычисляется после завершения цикла, а не до. Такая особенность не позволяет широко использовать do-while в современных программах.

Листинг 4.3.3 – Цикл while

#include <iostream>

**using** **namespace** std;

**int** main()

{

**int** i; // счетчик цикла

**int** sum = 0; // сумма чисел от 1 до 1000.

setlocale(0, "");

**for** (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.

{

sum = sum + i;

}

cout << "Сумма чисел от 1 до 1000 = " << sum << endl;

**return** 0;

}

**Глава 5. Контейнеры 🚚**

**Стандартная библиотека шаблонов (STL)** – это часть стандартной библиотеки языка программирования C++, которая предоставляет контейнеры и алгоритмы для управления ими, а итераторы служат интерфейсом между ними. Следующие главы посвящены описанию и применению алгоритмов с помощью упрощенного программирования.

**5.1. Контейнеры последовательностей**

**Контейнер** – специальная структура данных, которая хранит объекты организованным образом, следуя определенным правилам доступа. Существует три вида контейнеров:

1. **Контейнеры последовательности** – хранят элементы последовательно;
2. **Ассоциативные контейнеры** – хранят отсортированные контейнеры;
3. **Неупорядоченные ассоциативные контейнеры** – хранят хешированные объекты.

Ассоциативные и неупорядоченные ассоциативные контейнеры обеспечивают **быстрый поиск** отдельных элементов. Современные программы на C++ постоянно используют контейнеры.

**Контейнеры последовательностей** – контейнеры STL, которые позволяют использовать последовательный доступ к элементам. То есть можно начать с одного конца контейнера и перейти к другому концу. Разные контейнеры при этом, имеют разные характеристики и функции, а также разные способы доступа к элементам контейнера.

**5.1.1. Массив**

**Массив (std::array)** – последовательный контейнер, содержащий непрерывную последовательность элементов фиксированного размера. Сочетает в себе высокую производительность и эффективность. Данный контейнер следует использовать везде, взамен старых массивов, логика и работа которых полностью основывается на языке C.

Листинг 5.1.1 – Объявление массива std::array

#include <array>

std::array<**int**, 3> myArray; // объявляем массив значений int длиной 3

В отличие от старых массивов, вы можете не указывать размер и тип массива. **Доступ к значениям массива** с помощью операторов работает так:

Листинг 5.1.2 – Доступ к элементам массива std::array

std::cout << myArray[1] << '\n';

myArray[2] = 6;

**Размер массива** можно получить с помощью такой структуры:

Листинг 5.1.3 – Размер массива

std::array myArray { 9.0, 7.2, 5.4, 3.6, 1.8 };

std::cout << "length: " << myArray.size() << '\n';

**Сортировка массива** выполняется следующим образом:

Листинг 5.1.4 – Сортировка массива

#include <algorithm> // для std::sort

#include <array>

#include <iostream>

**int** main()

{

std::array myArray { 7, 3, 1, 9, 5 };

std::sort(myArray.begin(), myArray.end()); // сортируем массив по возрастанию

// std::sort(myArray.rbegin(), myArray.rend()); // сортируем массив по убыванию

**for** (**int** element : myArray)

std::cout << element << ' ';

std::cout << '\n';

**return** 0;

}

Вот полная таблица всех функций, связанных с операциями - https://docs.microsoft.com/ru-ru/cpp/standard-library/array-class-stl?view=msvc-160

**5.1.2. Векторы**

**Вектор в C++** — это замена стандартному динамическому массива, память для которого выделяется вручную. Разработчики языка рекомендуют использовать именно **vector** вместо ручного выделения памяти для массива. Это позволяет избежать утечек памяти и облегчает работу программисту.

Листинг 5.1.5 – Создание вектора

#include <iostream>

#include <vector>

**int** main()

{

// Вектор из 10 элементов типа int

std::vector<**int**> v1(10);

// Вектор из элементов типа float

// С неопределенным размером

std::vector<**float**> v2;

// Вектор, состоящий из 10 элементов типа int

// По умолчанию все элементы заполняются нулями

std::vector<**int**> v3(10, 0);

**return** 0;

}

Создадим вектор, в котором будет содержаться **произвольное количество** фамилий студентов:

Листинг 5.1.6 – Управление элементами вектора

#include <iostream>

#include <vector>

#include <string>

**int** main()

{

// Поддержка кириллицы в консоли Windows

setlocale(LC\_ALL, "");

// Создание вектора из строк

std::vector<std::string> students;

// Буфер для ввода фамилии студента

std::string buffer = "";

std::cout << "Вводите фамилии студентов. "

<< "По окончание ввода введите пустую строку" << std::endl;

**do** {

std::getline(std::cin, buffer);

**if** (buffer.size() > 0) {

// Добавление элемента в конец вектора

students.push\_back(buffer);

}

} **while** (buffer != "");

// Сохраняем количество элементов вектора

**unsigned** **int** vector\_size = students.size();

// Вывод заполненного вектора на экран

std::cout << "�'аш вектор." << std::endl;

**for** (**int** i = 0; i < vector\_size; i++) {

std::cout << students[i] << std::endl;

}

**return** 0;

}

Для добавления **нового элемента** в конец вектора используется метод push\_back(). Количество элементов определяется методом size(). Для доступа к элементам вектора можно использовать квадратные скобки [], также, как и для обычных массивов.

* pop\_back() — удалить последний элемент;
* clear() — удалить все элементы вектора;
* empty() — проверить вектор на пустоту;

**5.1.3 – Двусвязный список**

**Контейнер** **list** представляет двухсвязный список. Для его использования необходимо подключить заголовочный файл list. Создание списка:

Листинг 5.1.7 – Создание списка

std::list<**int**> list1; // пустой список

std::list<**int**> list2(5); // список list2 состоит из 5 чисел, каждый элемент имеет значение по умолчанию

std::list<**int**> list3(5, 2); // список list3 состоит из 5 чисел, каждое число равно 2

std::list<**int**> list4{ 1, 2, 4, 5 }; // список list4 состоит из чисел 1, 2, 4, 5

std::list<**int**> list5 = { 1, 2, 3, 5 }; // список list5 состоит из чисел 1, 2, 4, 5

std::list<**int**> list6(list4); // список list6 - копия списка list4

std::list<**int**> list7 = list4; // список list7 - копия списка list4

**5.2. Ассоциативные контейнеры**

**Ассоциативные контейнеры** обеспечивают очень быстрый поиск элементов. Последовательные контейнеры имеют некоторые естественное упорядочение, которое позволяет выполнять итерации от начала контейнера до конца в точно указанном порядке.

std::set, является ассоциативный контейнером, который содержит отсортированные уникальные элементы, называемые ключами. Поскольку set хранит отсортированные элементы, можно эффективно вставлять, удалять и искать элементы. Кроме того, set поддерживает отсортированную итерацию по его элементам, и можно полностью контролировать, как ключи сортируются с использованием объектов строения. (Листинг 5.2.1.1)

Листинг 5.2.1.1 – Множества

#include <iostream>

#include <ctime>

#include <set>

**using** **namespace** std;

**int** main() {

setlocale(LC\_ALL, "Russian");

srand(time(NULL));

multiset <**int**> mst;

cout << "Объявление случайных значений: " << endl;

**for** (**int** i = 0; i < 10; i++) {

**int** random = rand() % 10 + 1;

mst.insert(random);

cout << i + 1 << ") " << random << endl;

}

multiset <**int**> :: iterator it = mst.begin();

cout << "Отсортированный вариант: " << endl;

**for** (**int** i = 1; it != mst.end(); i++, it++) {

cout << \*it << " ";

}

system("pause");

**return** 0;

}

**Глава 6. Алгоритмы ⚗**

**Алгоритм** – процедура для решения класса задач. Стандартная библиотека C++ содержит множество алгоритмов, которые вы можете использовать в своих программах.

**6.1. Сложность алгоритмов**

**Сложность алгоритма** описывает **сложность вычислительной задачи**. Один из способов измерить эту сложность — использовать обозначения **Бахманна—Ландау**, или «**О большое**» (О большое). Обозначение О большое характеризует функции в соответствии с ростом вычислений относительно размера ввода. Это обозначение включает только **главный член функции сложности**. **Главный член** — тот, что растет быстрее всего при увеличении размера ввода.

Например, **алгоритм**, сложность которого увеличивается примерно на фиксированную величину для каждого дополнительного входного элемента, имеет обозначение О большое O(N), в то время как алгоритм, сложность которого не изменяется при заданном дополнительном входе, имеет обозначение О большое, равное O(1).

В этой главе описываются **алгоритмы**, которые подразделяются на пять **классов сложности**, как показано в следующем списке. Чтобы дать некоторое представление о том, как масштабируются эти алгоритмы, каждый класс указан в нотации О большое и с предположением о том, сколько дополнительных операций потребовалось бы из-за главного члена, когда ввод увеличивается с 1000 до 10 000 элементов. Каждый пример предоставляет операцию с **данным классом сложности**, где N — количество элементов, участвующих в операции:

1. **Постоянное время (О1)** – никаких дополнительных вычислений;
2. **Логарифмическое время (OlogN**) – примерно один дополнительный шаг в процессе вычисления;
3. **Линейное время (ON)** – примерно 9000 дополнительных шагов в процессе вычисления;
4. **Квазилинейное время (O\*NlogN)** – примерно 37 000 дополнительных шагов в процессе вычисления;
5. **Квадратичное время (O N2)** – примерно 99 000 000 дополнительных шагов в процессе вычислений.

**6.2. Список алгоритмов**

**Алгоритмы, не изменяющие последовательные операции** (Таблица 6.2.1)

Таблица 6.2.1 – Алгоритмы, не изменяющие последовательные операции

|  |  |
| --- | --- |
| **Название** | **Описание** |
| count | Возвращает количество элементов, значение которых равно заданному числу |
| equal | Возвращает значение true, если все соответствующие пары объектов из диапазонов равны |
| search | Проверяет, содержится ли второй диапазон внутри первого, возвращает начало совпадения или последний элемент первого диапазона, если нет совпадения |

**Алгоритмы, изменяющие последовательные операции** (Таблица 6.2.2)

Таблица 6.2.2 – Алгоритмы, изменяющие последовательные операции

|  |  |
| --- | --- |
| **Название** | **Описание** |
| fill | Присваивает заданное значение всем объектам из диапазона |
| remove | Удаляет из диапазона все значения, которые равны заданному значению |
| reverse | Обращает последовательность объектов из диапазона |
| replace | Заменяет все объекты равные old, объектам равными new |
| rotate | Отражает зеркально последовательность элементов |
| swap | Заменяет один объект другим |
| swap\_ranges | Обменивает соответствующие объекты в двух диапазонах |
| unique | Удаляет все эквивалентные объекты в последовательность, кроме первого |

**Алгоритмы сортировки данных** (Таблица 6.2.3)

Таблица 6.2.3 – Алгоритмы сортировки данных

|  |  |
| --- | --- |
| **Название** | **Описание** |
| nth\_element | Перемещает N-ый объект в позицию, которую он бы занимал после сортировки всего диапазона данных |
| sort | Сортирует объекты в диапазоне |
| stable\_sort | Сортирует объекты в диапазоне. Если два объекта равны, их порядок не изменится. |

**Алгоритмы бинарного поиска** (Таблица 6.2.4)

Таблица 6.2.4 – Алгоритмы бинарного поиска

|  |  |
| --- | --- |
| **Название** | **Описание** |
| binary\_search | Возвращает true, если заданное значение входит в интервал |
| equal\_range | Возвращает пару объектов, представляющих собой нижнюю и верхнюю границы, между которыми можно вставить значение value без изменения порядка сортировки |

**Алгоритмы слияния данных** (Таблица 6.2.5)

Таблица 6.2.5 – Алгоритмы слияния данных

|  |  |
| --- | --- |
| **Название** | **Описание** |
| merge | Соединяет отсортированные диапазоны 1 и 2 в диапазон 3 |

**Алгоритмы с отношениями** (Таблица 6.2.6)

Таблица 6.2.6 – Алгоритмы с отношениями

|  |  |
| --- | --- |
| **Название** | **Описание** |
| lexicographical\_compare | Возвращает true, если последовательность в диапазоне 2 следует в алфавитном порядке за последовательностью диапазона 1 |
| max | Возвращает наибольшее из a, b |
| max\_element | Возвращает итератор, указывающий на наибольший объект в диапазоне |
| min | Возвращает наименьшее из a, b |
| min\_element | Возвращает итератор, указывающий на наименьший объект в диапазоне |
| next\_permutation | Выполняет одну перестановку в последовательности данного диапазона |
| prev\_permutation | Выполняет одну обратную перестановку в последовательности данного диапазона |

**Глава 7. Мета программирование алгоритмов** 📟

Данная глава будет лишь обзорной экскурсией в область современного применения языка программирования C++, и в данном случае будут показаны примеры вычисления различных математических функций с помощью продвинутого программирования на C++ с использованием парадигмы мета программирования.

**7.1. Введение**

Под мета программированием в языке программирования C++ подразумевается вычисления на **стадии компиляции** нашей программы (проекта). Основным инструментом для этого являются шаблоны классов, называемые в этом случае – **мета функциями**.

Язык мета программирования C++ является **полным** в том смысле, в каком полной является **машина Тьюринга** или **лямбда-исчисление**, т.е. на нем можно реализовать любую технические вычислимую функцию. Язык C++ одновременно является еще и функциональным языком, потому что в нем присутствуют **операторы присваивания**.

**Метапрограммирование** – вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы, либо программ, которые меняют себя во время выполнения. Данный вид программирования позволяет получать программы при меньших затратах времени и усилий на кодирования и создание алгоритмов исходного кода, чем если бы программист писал их вручную целиком.

**7.2. Базовые мета функции**

В данной под главой показаны простейшие (базовые) мета функции, которые в дальнейшем будут использоваться для более сложных вычислений. Как уже говорилось, мета функция с точки зрения языка программирования C++ — это шаблон класса, параметры мета функции – это параметры шаблона класса, в качестве них могут передаваться или целочисленные константы, значения которых известны во время компиляции или различные типы.

Например факториал, исходный код которого показан в листинге 7.2.1. (Листинг 7.2.1):

Листинг 7.2.1 – Факториал

**template**<**unsigned** N>

**struct** factorial {

**static** **const** **unsigned** value = N \* factorial<N - 1>::value;

};

**template**<>

**struct** factorial<0> {

**static** **const** **unsigned** value = 1;

};

// Пример обращения:

**const** **unsigned** f5 = factorial<5>::value; // 1

**Бином Ньютона** – формула для разложения на отдельные слагаемые неотрицательной степени двух переменных, имеющих вид (Рисунок 7.2.2):

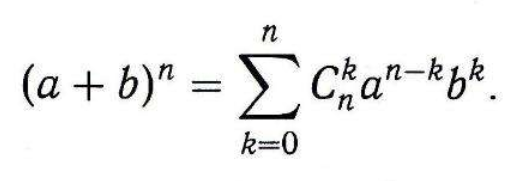


Рисунок 7.2.2 – Бином Ньютона

**Биноминальный коэффициент** – коэффициенты в разложении Бинома Ньютона по степенях x. (Рисунок 7.2.3)



Рисунок 7.2.3 – Биноминальный коэффициент

Листинг 7.2.4 – Биноминальный коэффициент

**template**<**unsigned** N, **unsigned** K>

**struct** C {

**static** **const** **unsigned** value =

factorial<N>::value / factorial<K>::value / factorial<N-K>::value;

};

// Пример обращения:

const unsigned i = C<5, 2>::value; //10

Теперь допустим, что нам нужно найти интеграл от одночлена на плоскости с тремя осями – 3D-измерение – Ось X, Ось Y, Ось Z. (Листинг 7.2.5).



Рисунок 7.2.4 – Интеграл от одночлена на плоскости с тремя осями

Листинг 7.2.4 – Биноминальный коэффициент

**template**<**unsigned** alpha, **unsigned** beta, **unsigned** gamma>

**struct** monomial\_3d {

**static** **const** **unsigned** value =

factorial<alpha + beta + gamma + 3>::value /

(factorial<alpha>::value \* factorial<beta>::value

\* factorial<gamma>::value);

};

**7.3. Сложные мета функции**

На этапе компиляции несложно проверить целое число на простоту. Такие функции являются обыденностью в физико-математические науках, тем более что исходный код, написанный в данной главе (Глава 7), полностью создан для одной или нескольких переменных. Но что, если таких переменных тысячи или миллионы, вот тут и пригодится умение мета программирования, параллельность данных и многопоточность выполнения программ.

Листинг 7.3.1 – Биноминальный коэффициент

**template**<**int** n, **int** i>

**struct** is\_prime0 : std::conditional<(n % i) == 0,

std::false\_type, is\_prime0<n, i - 1>

>::type {};

**template**<**int** n>

**struct** is\_prime0<n, 1> : std::true\_type {};

**template**<**int** n>

**struct** is\_prime : is\_prime0<n, n / 2>{};

**ЭПИЛОГ**

**Дорогие студенты!**

Вот и подошел к концу ваше маленькое путешествие по языку программирования C++. Надеемся, что вам понравилось это приключение, ведь преподаватели и авторы пособия сделали все возможное, чтобы вы получили максимум качественных и современных практических знаний за минимальное время.

Если же вам не понятны некоторые конструкции, у вас есть вопросы или предложения, рекомендуем обратиться по электронной почте, указанной на обложке пособия для получения обратной связи с авторами и преподавателями.

Мы надеемся, что навыки, полученные вами, пригодятся вам в дальнейшей профессиональной жизни и программирование, будет занимать большую часть вашей профессиональной жизни.

Технологический процесс идет быстрее, чем пишется данное пособие и курс к нему, поэтому ошибки, неточности и упущения могут быть частым явлением. Советуем вам искать дополнительную информацию в сети интернет, ведь нельзя уместить всю информацию об языке программирования в 50 с лишним страниц. Рекомендуем хоть немного, но заглядывать в книги, которые написаны в списке литературы, так как это действительно самые лучшие книги для старта в языке программирования C++.

**Спасибо вам за этот путь, который вы проделали вместе с преподавателями! Никогда не останавливайтесь на пути своего обучения!**

**Сборник задач**

**Глава 1. Создаем и запускаем** 🚀

**Упражнение 1.1**. Создайте функцию, которая возвращает абсолютное значение одного аргумента. Абсолютное значение целого числа A является следующим: A (само по себе), если A больше или равно 0; в противном случае это A в степени -1. Можно использовать предыдущие листинге в качестве шаблона.

**Упражнение 1.2**. Попробуйте запустить программу с разными значениями. Совпадают ли полученные значения с ожидаемыми?

**Упражнение 1.3.** Запустите программу с отладчиком, шаг за шагом выполняя каждую инструкцию.

**Упражнение 1.4.** Напишите другую программу, которая принимает два целочисленных аргумента и возвращает их сумму. Нельзя использовать предыдущие коды, только теория, которая была написана в первой главе, без использования листингов.

**Упражнение 1.5.** Напиши программу, которая будет вычислять площадь и периметр треугольника. Не используйте одинаковые переменные. Чем сложнее формула и программа, тем выше ваша оценка.

**Упражнение 1.6.** Напишите программу, которая будет вычислять синус, косинус, тангенс и котангенс угла по сторонам и углам треугольника. Используйте новые возможности C++ 20, в частности число pi\_v.

**Глава 2. Типы ⚙**

**Упражнение 2.1.** Составить программу, которая требует ввести два числа. Если первое число больше второго, то программа печатает слово больше. Если первое число меньше второго, то программа печатает слово меньше. А если числа равны, программа напечатает сообщение Эти числа равны.

**Упражнение 2.2**. Напишите программу с помощью структур, пройденных во второй главе подобие калькулятора с несколькими операциями. Используйте функции из первой главы.

**Упражнение 2.3.** Напишите программу с помощью массивов и циклов, которые будут заполнять массив различными числами.

**Упражнение 2.4.** Напишите программу, которая будет вычислять год – високосный или нет.

**Упражнение 2.5.** Напишите программу, которая будет использовать std::byte лучше всего.

**Упражнение 2.6**. Напишите программу «Операционная система», где с помощью оператора ветвления в зависимости от выбранного параметра будут запускать различные программы – «Калькулятор», «Календарь», «Часы».

**Глава 3. Ссылочные типы** 📇

**Упражнение 3.1.** Создайте программу используя ссылочные типы. Чем больше функций будет в программе, и чем она сложнее, тем выше ваша оценка. Используйте все доступные возможности.

**Глава 4. Выражения и инструкции** 🌌

**Упражнение 4.1.** Создайте программу используя выражения и инструкции. Чем больше функций будет в программе, и чем она сложнее, тем выше ваша оценка. Используйте все доступные возможности.

**Глава 5. Контейнеры** 🚚

**Упражнение 5.1.** Создайте программу используя один из контейнеров. Чем больше функций будет в программе, и чем она сложнее, тем выше ваша оценка. Используйте все доступные возможности.

**Глава 6. Алгоритмы** ⚗

**Упражнение 6.1.** Создайте программу используя один из алгоритмов. Чем больше функций будет в программе, и чем она сложнее, тем выше ваша оценка. Используйте все доступные возможности.

**Глава 7. Мета программирование алгоритмов** 📟

**Упражнение 7.1**. Напишите программу, вычисляющую факториал числа от 0 до n. Используйте все доступные возможности.

**Упражнение 7.2**. Напишите программу, вычисляющую Бином Ньютона и биноминальный коэффициент. Используйте все доступные возможности.

**Упражнение 7.3.** Напишите программу, вычисляющую интеграл от любой математической формулы или уравнения. Используйте все доступные возможности.

**Список использованной литературы**

1. Лоспинозо Джош – «C++ для профи» – СПб: Питер, 2021. – 816 с.: ил. – (Серия «Для профессионалов»), ISBN 978-5-4461-1730-7
2. М.М. Краснов – «Метапрограммирование шаблонов C++ в задачах математической физики». – М.: ИПМ им. М.В. Келдыша, 2017. – 84 с.
3. Т.А. Жданова, Ю.С. Бузыкова – «Основы алгоритмизации и программирования: учеб. пособие» – Хабаровск: Изд-во Тихоокеанского Государственного Университета, 2011. – 56 с.
4. Интернет-ресурс «<https://ru.cppreference.com/w/Заглавная_страница>»
5. Интернет-ресурс «https://habr.com»

**Список использованных графических элементов**

* Рисунок 1.1.1 - <https://visualstudio.microsoft.com/wp-content/uploads/2021/06/preview-video.png>
* Рисунок 7.2.2 - <https://mypresentation.ru/documents_6/1979886aecc93259b6eb1db500d1abf4/img34.jpg>
* Рисунок 7.2.3 - <https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%BD%D0%BE%D0%BC%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D1%8D%D1%84%D1%84%D0%B8%D1%86%D0%B8%D0%B5%D0%BD%D1%82>