Лекция «Знакомство с языком C++»

СОДЕРЖАНИЕ

1 ИСТОРИЯ ПОЯВЛЕНИЯ ЯЗЫКА С И С++

1.1 Язык С

Первые ЭВМ программировались непосредственно в машинных кодах. Написание, чтение и изменение кода программ, состоящего из чисел, было затруднительно, поэтому появились т. н. языки ассемблера — запись машинных кодов в виде слов. Код на языке ассемблера по структуре не отличался существенно от машинного, кроме того, что был проще для восприятия человеком. Из-за этого перенос программ между ЭВМ оказывался затруднителен, если у них были разные системы команд. Одновременно код оставался невыразительным с точки зрения логики программы. Преодолеть эту проблему удалось путем создания первых высокоуровневых языков программирования (Fortran, COBOL, LISP, BASIC), однако они еще более жестко опирались на платформу (систему кодов) ЭВМ, для которой реализовывались, и не для каждой платформы выпускался компилятор (в силу сложности языков).

В 1972 году инженерами компании Bell Labs Брайаном Керниганом (Brian Kernighan) и Дэннисом Ритчи (Dennis Ritchie) были опубликованы первые версии языка С для машин PDP-11. В его основу были заложены принципы [1]:

а) Высокий уровень, то есть использование конструкций, отражающих логику программы (циклы, ветвления, подпрограммы), а не порядок выполнения машинных кодов (сравнения, переходы).

б) Гибкость и мощность, сиречь возможность на С сделать не менее функциональную программу, чем на чистом ассемблере (пользуясь возможностью вставлять в программу на С части ассемблерного кода).

в) Простота компиляции и независимость от платформы, призванная максимально упростить создание компиляторов С под новые ЭВМ. В язык не вносилось тех средств, которые можно было бы реализовать на нем самом как библиотеки (например, ввод-вывод).

Таким образом, язык С позволял вести разработку любого ПО (будучи «высокоуровневым ассемблером») и удобно для программиста как язык высокого уровня (ЯВУ), а перенос программ на другие платформы был предельно упрощен — достаточно было реализовать сравнительно простой компилятор. Будучи создан с учетом нужд практики, язык С быстро завоевал всемирную популярность.

Реализации языка С зачастую немного отличались, поэтому в 1989 году ANSI установил стандарт на язык С — ANSI C89, чтобы зафиксировать элементы языка и состав

стандартной библиотеки. Следующий аналогичный стандарт, самый популярный, появился через 10 лет: ANSI C99, а в 2011 году — его обновление, стандарт C11.

1.2 Язык C++

Язык C имел процедурную парадигму, то есть предполагалось структурирование кода при помощи подпрограмм (функций и процедур). К началу 80-х выяснилось, что для создания сложных приложений этого недостаточно. Среди предложенных удачных решений было объектно-ориентированное программирование (ООП). Его встроенная поддержка отсутствовала в C, поэтому, начиная с 1984 года, Бьярн Страуструп из Bell Labs разрабатывал новый язык программирования на основе C с поддержкой ООП, в конечном счете названный C++.

Самым привлекательным свойством C++ стала забота об *обратной совместимости (backward compatibility)* с C и собственными предыдущими версиями. Разработчики зачастую могли использовать большое количество готового кода прямо в программах на новом языке и быть уверенными, что его не понадобится переписывать. Очевидная выгода от этого способствовала почти столь же широкому распространению C++, как и C.

В ходе развития язык C++ претерпевал изменения и многочисленные эксперименты, иногда спорные, но большинство из них оставалось в языке. Одновременно появлялись другие языки, основанные на ООП: Objective-C (совместим с C), Java, Object Pascal (Delphi), C#, где многие ошибки проектирования C++ были учтены и не повторялись. Они оказались удачны для новых приложений, однако во многих случаях переписывать с C++ имеющийся код было нецелесообразно, либо новый код активно использовал старый на C++, либо требовалась одновременно производительность и кроссплатформенность. Таким образом, язык C++, в некотором смысле, «проложил дорогу» многим последующим языкам, иногда более удобным.

2 СОВРЕМЕННОЕ ПОЛОЖЕНИЕ C++

2.1 Стандартизация

Первые 15 лет своего существования C++ основывался на описании в книгах Страуструпа [2] и на поддержке отдельных возможностей компиляторами. В 1998 году Международная организация по стандартизации (ISO) утвердила первый стандарт C++, включающий стандарт языка C++ и описание стандартной библиотеки.

Стандарт C++ описывает синтаксис (внешний вид) и семантику (смысл, поведение) конструкций языка, исполняемых на некоей абстрактной машине. Результат компиляции и выполнения реальных программ должен совпадать с теоретическим, описанным в стандарте. Детали конкретных реализаций намеренно оставлены на усмотрение разработчиков компиляторов.

Стандарты C++ принимаются комитетом ISO, в состав рабочих групп которого входят члены коллективов разработчиков компиляторов и представители крупных IT-компаний. Изменения всегда имеют обоснование (rationale), обсуждения ведутся публично, принимаются комментарии общественности, периодически бесплатно публикуются черновики стандарта [3]. Финальная редакция [4] доступна только платно (порядка 150 $ за стандарт), она почти совпадает с последним черновиком.

Таким образом, модель развития C++ такова:

а) Комитетом ISO вырабатывается очередной стандарт или набор поправок.

б) Производители компиляторов реализуют поддержку новой версии стандарта.

в) Разработчики пользуются новой версией языка, выявляют недостатки и выдвигают очередные пожелания. Тогда начинается разработка очередного стандарта.

На практике, все три этапа перекрываются: элементы нового стандарта реализуются в компиляторах еще до официального принятия, разработчики сразу оценивают нововведения и обращаются к комитету с просьбами изменить будущий стандарт.

В 2003 году был принят очередной, актуальный дольше других, стандарт C++: ISO/IEC 14882:2003, сокращенно называемый C++03. В 2007 году к нему были приняты поправки (т. н. Technical Report 1, или TR1), дополняющие стандартную библиотеку языка. В 2011 году принят действующий стандарт ISO/IEC 14882:2011 [5], называемый C++11: язык пополнен новыми конструкциями, а стандартная библиотека существенно расширена. В настоящее время (сентябрь 2013) C++11 поддерживается практически в полной мере всеми распространенными компиляторами, и ведется разработка C++14.

2.2 Степени свободы поведения программы

Говоря о поддержке тем или иным компилятором определенного стандарта C++ (редакции), имеют в виду то, насколько полно поведение скомпилированной программы совпадает с описанным в стандарте. Часть поведения в стандарте намеренно не фиксируется, чтобы не создавать неконструктивные ограничения разработчикам компиляторов. Поведение программы при этом может быть:

Неопределенное (undefined behavior, UB), при котором результат компиляции и запуска программы неизвестен никому и может быть любым: ошибка компиляции, ошибка времени исполнения, «нормальная» (с т. з. программиста) работа программы. Код, поведение которого неопределенное — заведомо ошибочен вне зависимости от того, какое реально поведение наблюдается [5, п. 1.3.24].

Недокументированное (unspecified behavior), когда разработчиком компилятора может быть реализован один из возможных вариантов без указания в документации, какой именно и в каких случаях. Пример: порядок вычисления `g()` и `h()` — аргументов, передаваемых функции `f()` в конструкции вида `f(g(), h())`. Важно не полагаться на наблюдаемую реализацию, даже если удалось её определить, так как это может измениться в будущих версиях компилятора или при иных условиях [5, п. 1.3.25].

Определяемое реализацией (implementation-defined behavior), когда разработчиком компилятора документируется выбранный вариант поведения программы. Встречается редко. Полагаться на такое поведение при разработке программ не возбраняется, но возникает риск, что этом возникает привязка кода к определенному компилятору [5, п. 1.3.10].

2.3 Область применения

В настоящее время C++ используется для создания высокопроизводительных, кроссплатформенных или низкоуровневых приложений и библиотек. Это связано с тем, что данные приложения имеют сложную структуру и нуждаются в высокоуровневых средствах языка (ООП), но в то же время требуется возможность низкоуровневых операций (работа с памятью, ассемблерные вставки) и большая производительность (за счет контроля использования ресурсов). Также C++ используется при поддержке старого, или унаследованного, кода (legacy code). Области применения:

а) Графика. Во-первых, это визуализация в технических приложениях, например, при расчетах конструкций, в медицине, физике. Во-вторых, это индустрия развлечений — кинематограф и видеоигры (для игровых приставок альтернатив C++ зачастую нет).

б) Математические расчеты. Дело в том, что современные компиляторы имеют хорошие средства оптимизации на уровне машинного кода, в частности, код может автоматически быть приспособлен для многопроцессорного параллельного исполнения. Диалекты C и C++ применяются в технологиях CUDA и OpenCL — для высокопроизводительных вычислений на векторных процессорах (например, на видеокартах).

в) Высокопроизводительные системы, действующие в условиях ограниченных ресурсов машины и жестких требований по скорости работы. Это системы управления базами данных (СУБД), web-серверы, обработка звука и видео, особенно потоковая и в реальном времени.

г) Системное и низкоуровневое программирование. Во-первых, это ядра операционных систем и драйверы устройств, во-вторых, на C и C++ часто программируются специальные устройства (микроконтроллеры, платы, сетевое оборудование).

д) Крупные прикладные библиотеки (frameworks), например, библиотеки пользовательского интерфейса — MFC, Qt, wxWidgets, Gtk+, виртуальные машины и интерпретаторы языков программирования.

е) Кроссплатформенные программы, в которых требуется что-либо из вышеперечисленного.

2.4 Инструментальные средства

Язык C++ используется очень широко для создания сложных, важных приложений. Поэтому для него разработано большое количество компиляторов, отладчиков, и других инструментальных средств, включаемых сообща в инструментальные среды разработки (ИСР, англ. integrated development environment — IDE).

Популярные компиляторы перечислены в таблице 1. Под компилятором здесь понимается одновременно транслятор кода на C++ в машинные команды и компоновщик для разрешения зависимости программы от библиотек.

Таблица 1 — Популярные компиляторы C++

Компилятор	**Модель распространения**	**Операционная система**	**Краткая характеристика**
Microsoft Visual C++ Compiler (MSVC++)	коммерческая	Windows	Поставляется вместе с ИСР Visual Studio. Среднего качества поддержка стандартов,

			присутствуют собственные расширения языка для технологий Microsoft (COM).
Intel C++ Compiler (ICC)	коммерческая	Windows, Linux, OS X	Высокая производительность скомпилированного кода на процессорах Intel. Большое количество расширений языка (поддержка параллелизма, наборов инструкций ЦП и т. п.).
GNU C++ Compiler (gcc, правильнее — g++)	свободное ПО	Windows, *nix, OS X, специальные ОС и платформы	Самый широкий охват программно-аппаратных платформ, очень популярен. Хорошая поддержка стандартов.
Clang	свободное ПО	Window, Linux, OS X	Новизна, современная архитектура, очень хорошая поддержка стандартов. Находится в стадии разработки, планируется на замену g++.
Borland C++ Compiler и CodeGear C++ Builder (BCC, C++ Builder)	коммерческая	Windows (Borland), Windows, Linux, OS X, iOS (CodeGear)	Плохая поддержка стандартов. Средства RAD (как в Delphi), акцент на разработке интерфейсов в собственной ИСР.

3 ВВЕДЕНИЕ В ЯЗЫК C++

3.1 Структура программы и пример

Рассмотрим программу, печатающую на экране строку «Hello, world!» (Листинг 1).

```
1  #include <cstdio> /* Директива препроцессора. */
2
3  int main() {
4        puts("Hello, world!");
5        return 0;
6        puts("Вот скачет Неуловимый Джо!");   // Никогда не выполнится.
7  }
```

Листинг 1 — Программа, печатающая на экране строку «Hello, world!»

В коде присутствуют комментарии: текст, заключенный в `/*  */` (возможно, на несколько строк) — т. н. комментарии в стиле C (C-style comments), а также текст от символов `//` до конца строки [5, п. 2.8].

На строке 3 содержится объявление функции `main()`, с вызова которой начинается выполнение программы [5, п. 3.6.1]. Перед именем функции указан тип ее возвращаемого значения — целое число со знаком (**`int`**, от англ. *integer* — целое). Пустые круглые скобки означают отсутствие у функции аргументов. Тело функции содержится в фигурных скобках (см. «Синтаксис объявления и реализации функции»).

Строка 4 выводит на экран сообщение «Hello, world!». В C++ нет встроенных средств ввода-вывода, наподобие оператора[1] **`WriteLn`** в Pascal, поэтому вывод осуществляется функцией[2] `puts()` стандартной библиотеки ввода-вывода C, которой передается аргумент — строковой литерал в двойных кавычках [5, п. 2.14.5].

Функция `puts()` не является автоматически доступной в любой программе. Она объявлена в т. н. заголовочном файле (header file) `cstdio`. Строка 1 — это не оператор программы, это директива препроцессора (preprocessor instruction), которая выполняется до компиляции и приводит к включению в исходный текст содержимого указанного файла [5, п. 16.2]. Угловые скобки показывают, что файл берется не из каталога с файлом программы, а из каталога общих заголовочных файлов (его расположение зависит от настроек компилятора). Реализация может находиться в самом заголовочном файле, но чаще — в скомпилированной библиотеке времени выполнения (runtime library).

[1] Корректный термин, согласно документации, — intristic subroutine (встроенная подпрограмма), но так как `WriteLn` невозможно реализовать средствами самого Pascal, то, по сути, это оператор языка.

[2] В C++ нет деления подпрограмм на функции и процедуры.

Функция `main()` обязана вернуть значение целого типа со знаком. Это делается оператором **`return`** в строке 5. Возвращаемое значение (`0`) будет использовано как код завершения программы (exit code) и моет быть проанализировано тем, кто ее запустил.

Оператор **`return`** приводит к немедленному выходу из функции [5, п. 6.6.3], поэтому строка 6 не выполнится, и второе сообщение выведено не будет (да никому оно и не нужно :-).

Языки C и C++ *чувствительны к регистру символов (case sensitive),* поэтому `main` и `Main` — разные идентификаторы, может быть переменная `Return`, но **`return`** — ключевое слово, и т. п.

Очевидно, что минимализм устройства C++ делает написание программ на нем сравнительно сложным. Однако, это является «платой» за гибкость языка. Можно было бы воспользоваться не старой библиотекой ввода-вывода («в стиле C»), а новой[3], описанной в заголовочном файле `iostream`, или вовсе не подключать средства ввода-вывода текста, например, если устройства вывода нет (случай приложений для микроконтроллеров).

3.2 Виды конструкций языка C++

Конструкции языка C++ (language constructs) делятся на объявления (declarations), определения (definitions), операторы (statements) и выражения (expressions).

Выражения могут иметь возвращаемое значение и входить в другие выражения, например, в выражении `1 + 2*3` содержатся выражения сложения и умножения.

Операторами называются конструкции, управляющие ходом выполнения программы и не имеющие возвращаемого значения, например, условный оператор. Разумеется, операторы также могут быть вложенными — например, проверка условия в цикле. Не следует путать оператор как конструкцию языка и оператор как *лексему (token)* — неделимый элементы языка — например, **`+`** или **`*=`**.

Объявления — это конструкции для введения в программу имен функций, переменных, типов данных и т. д., а *определения* задают вид или состав объявляемого.

Перечисленные термины часто переводят по-разному. Мы будем всюду пользоваться приведенным переводом и пониманием соответствующих терминов.

[3] Так поступают авторы некоторых учебников по C++. К сожалению, это требует либо объяснения довольно сложного и объемного материала, либо использования средств ввода-вывода без понимания принципов их работы, что, на наш взгляд, неприемлемо в учебном курсе.

3.3 Простые типы данных и модификаторы

Базовые (fundamental) типы данных C++ включают числовые, символьные, логический и пустой тип [5, п. 3.9.1]. Числовые типы данных приведены в таблице 2. В столбце «целые» допускается употреблять соответствующие типы, опуская **`int`** и **`signed`**, то есть **`char`** эквивалентно **`signed char`**, а **`unsigned`** — **`unsigned int`**.

Таблица 2 — Числовые типы данных C++

Целые			**С плавающей запятой (вещественные)**
Знаковые	**Беззнаковые**		
`signed char`	**`unsigned char`**		**`float`**
`short int`	**`unsigned short int`**		**`double`**
`int`	**`unsigned int`**		**`long double`**
`long int`	**`unsigned long`**		
`long long int`	**`unsigned long long int`**		

Точные размеры и диапазоны не фиксируются стандартом, а вместо этого требуется, чтобы каждый следующий (сверху вниз) тип в любом столбце таблицы 2 включал весь диапазон значений предыдущего. Тип **`char`** должен быть достаточного размера, чтобы хранить код любого символа кодировки по умолчанию данной платформы (обычно ASCII, поэтому размер 7—8 бит). Другой символьный тип, **`wchar_t`**, обсуждается в последующих лекциях.

Типичные размеры и диапазоны значений на 32-битной архитектуре x86 в ОС Windows, а также аналоги типов C++ в Delphi (32 бита) приведены в таблице 3.

Таблица 3 — Типичные характеристики числовых типов C++ и их аналоги в Delphi 7

Тип C++	**Диапазон**[4, 5]	**Размер**	**Аналог в Delphi 7 (32 бита)**	
			Знаковый	**Беззнаковый**
`char`	±127	1 байт (8 бит)	`SmallInt`	`Char, Byte`
`short int`	±32 767	2 байта (16 бит)	`ShortInt`	`Word`
`int` / **`long int`**	±2 147 483 647	4 байта (32 бита)	`Integer`	`Cardinal`
`long long int`	±9 223 372 036 854 775 807	8 байт (64 бита)	`LongInt`	`LongWord`

[4] Для знаковых типов верхняя граница на 1 меньше (по модулю), чем нижняя, например, для **`char`** — от −128 до +127. В таблице для краткости это не учтено.

[5] Для беззнаковых типов диапазон от 0 до удвоенной границы беззнакового типа, минус 1, например, для **`unsigned char`** — от 0 до +255.

float	Зависит от точности значения, формат по IEEE 754.	4 байта	Single
double		8 байт	Real
long double		12 байт (96 бит)	Extended

Логический тип **bool** может принимать значения **true** (истина) и **false** (ложь).

Специальный тип **void** используется в случаях, когда тип отсутствует, например, это тип результата функции, которая ничего не возвращает (как процедура в Pascal).

К типам могут применяться *модификаторы* ***const*** *и* ***volatile*** *(cv-qualifiers)*, которые пишутся перед именем типа при объявлении переменной. Переменные, объявленные как **const**, не могут изменять свое значение, хотя оно может быть им сразу присвоено при объявлении [5, п. 3.9.3, п. п. 7.1.6.1]. Смысл и применение **volatile** будет рассмотрено в последующих лекциях.

Примеры объявления переменных см. в листингах 5—7.

В отличие от Pascal, в C++ объявление переменной — это просто выражение, которое не обязано располагаться в каком-либо определенном месте программы или функции. Областью видимости переменной является тот блок, где она объявлена (например, ветвь условного оператора, тело цикла и т. д.).

3.4 Операторы

3.4.1 Оператор-выражение, пустой и составной оператор

Оператор-выражение (expression statement) — это оператор, состоящий из выражения, результат которого отбрасывается, и точки с запятой сразу вслед за ним. Если опустить выражение, получится *пустой оператор (null statement)*, который не выполняет никаких действий и состоит только из точки с запятой (**;**). Он применяется, когда синтаксис конструкции требует оператора, но выполнять никаких действий не нужно [5, п. 6.2]. Далее в тексте мы будем считать точку с запятой частью оператора, указывая «оператор» в описании синтаксиса.

Составной оператор (compound statement, block) — это любое, в том числе нулевое, количество операторов, размещенное в фигурных скобках (**{}**). Он применяется, когда синтаксис конструкции предусматривает один оператор, а требуется выполнить несколько действий. В отличие от составного оператора Pascal **begin**… **end**, точкой с запятой должен заканчиваться каждый оператор составного, даже последний или единственный [5, п. 6.3].

3.4.2 Операторы выбора

К операторам выбора (selection statements) относятся **if**… **else** и **switch** [5, п. 6.4].

Оператор **if**… **else** [5, п. 6.4.1] реализует алгоритмическую конструкцию «если-то-иначе». Синтаксис оператора приведен ниже:

```
if (выражение-условие-1)
      оператор-1,-выполняющийся,-если-условие-1-истинно
else if (выражение-условие-2)
      оператор-2,-выполняющийся,-если-условие-2-истинно
else
      оператор-n,-выполняющийся,-если-все-условия-ложны
```

Частей **else if** может быть много, а может не быть вовсе; часть **else** может отсутствовать. *Условия* должны быть приводимы к логическому или целочисленному типу и считаются истинными, если результат отличается от 0 (такое поведение совместимо с C, где не было логического типа данных). Круглые скобки вокруг *условий* обязательны.

Оператор **switch** [5, п. 6.4.2] реализует выбор одного из конечного числа вариантов и используется подобно **case**… **of** в Pascal. Синтаксис оператора **switch** таков:

```
switch (выражение) {
case вариант-1:
      операторы-варианта-1
case вариант-2:
      операторы-варианта-2
…
case вариант-n:
      операторы-варианта-n
default:
      операторы-по-умолчанию
}
```

Выражение вычисляется, и управление передается метке **case**, *вариант* которой равен значению *выражения*, либо к метке **default**, если значение *выражения* не равно ни одному из *вариантов* (если метка **default** есть).

Однако, механизм действия оператора C и C++ существенно отличается от аналогичных в других языках программирования. Метки case не являются границами кода, который будет выполнен для значения выражения, равного варианту. Например, если значение *выражения* равно *варианту-1*, будут выполнены *операторы-вараинта-1*, затем *операторы-варианта-2* и так далее до *операторов-по-умолчанию* включительно. Это поведение называется *проваливающимся управлением (fall-through)* и часто нежелательно. Чтобы немедленно покинуть оператор **switch** после того, как соответствующие *операторы* будут выполнены, нужно воспользоваться оператором

```
break;
```

С другой стороны, fall-through позволяет перечислить несколько меток **`case`** подряд, чтобы одни и те же *`операторы`* срабатывали для заданного множества *`вариантов`*.

3.4.3 Операторы циклов

Детерминированные (с условием продолжения) циклы в C++ — это циклы **`while`** и **`do`**… **`while`** [5, п. п. 6.5.1, 6.5.2], которые эквивалентны циклам **`while`** и **`repeat`**… **`until`** в Pascal соответственно с той разницей, что **`do`**… **`while`** использует не условие окончания, а условие продолжения. Синтаксис этих конструкций таков:

а) **`while`** (*`выражение-условие-продолжения`*) *`оператор-тело-цикла`*

б) **`do`** *`оператор-тело-цикла`* **`while`** (*`выражение-условие-продолжения`*)**`;`**

Итерационный цикл (со счетчиком итераций) в C++ реализуется оператором **`for`** [5, п. 6.5.3] со следующим синтаксисом:

`for` (*`оператор-инициализации`***`;`** *`выражение-условие-продолжения`***`;`** *`выражение`*)
    `оператор-тело-цикла`

Перед началом цикла (один раз) исполняется *`оператор-инициализации`*. Пока *`выражение-условие-продолжения`* истинно, *`выполняется оператор-тело-цикла`*, а вслед за ним — *`выражение`* из заголовка **`for`**. Можно опускать *`выражение-условие-продолжения`*, которое считается в этом случае всегда истинным. Допускается также опускать *`выражение`*. Типичный цикл **`for`**, в котором **`i`** принимает значения от 0 до 9 выглядит так: **`for (int i = 0; i < 10; ++i) { … }`**, а бесконечный цикл — **`for (;;) { … }`**.

В стандарте C++11 введен новый вид цикла **`for`** для перебора элементов коллекций (массивов и более сложных структур данных), называемый *range-based* *`for`* *statement* [5, п. 6.5.4], который будет рассмотрен в последующих лекциях.

В теле любого цикла можно использовать оператор **`break`**, чтобы прервать цикл, и аналогичный оператор **`continue`**, чтобы перейти к следующей итерации. Для цикла **`for`** в последнем случае будет очередной раз вычислено *`выражение`*, как будто пройден конец итерации.

3.4.4 Операторы перехода

К операторам перехода (jump statements) относятся **`break`**, **`continue`**, **`return`** и **`goto`**. Использование первых двух см. выше в п. п. 3.4.2—3.4.3, об операторе **`return`** см. подраздел 3.6 ниже.

Оператор **goto** [5, п. 6.6.4] используется совместно с оператором объявления именованной метки (identifier label statement) для безусловного перехода. Синтаксис объявления метки:

наименование-метки:

Синтаксис оператора **goto**:

goto *наименование-метки*;

Метка, на которую совершается переход оператором **goto**, должна находиться в той же функции (см. 3.6) и в ее же пределах иметь уникальное имя среди меток (но имя метки может совпадать с именем переменной — это будут два независимых идентификатора).

Оператор **goto** остается в C++, в основном, в целях обратной совместимости. Его использование противоречит принципам структурного программирования, зачастую порицается на практике и всегда может быть заменено на другие конструкции. Среди исключений — переписывание на C++ кода с **goto** других языков программирования и случаи, когда вариант без **goto** оказывается значительно сложнее и менее понятен.

3.5 Выражения

3.5.1 Присваивание

Оператор присваивания в C++ — знак равенства (=). В отличие от Pascal, присваивание в C++ имеет результат (присваиваемое значение), поэтому можно присвоить значение сразу нескольким переменным (типа **int**):

```
x = y = z = 1;  // Переменные x, y и z теперь имеют значение 1.
```

Присваивание выполняется справа налево (говорят: имеет *правую ассоциативность*), то есть вышеприведенное выражение следует читать так: «присвоить z значение 1, присвоить y значение z (1), присвоить x значение y (1)».

В языках C и C++ имеются операторы совмещенного присваивания (compound assignment): *=, /=, %=, +=, -=, >>=, <<=, &=, ^= и |=. Например, эквивалентны следующие выражения: x = x + y и x += y, и аналогично для остальных операторов группы. Подробнее см. [5, п. 5.17].

3.5.2 Арифметические операции

Арифметические операции в C++ подобны одноименным в Pascal [5, п. п. 5.6—5.7].

Взятие остатка от деления обозначается оператором % и аналогично **mod** в Pascal.

Операторы инкремента (увеличения на 1) ++ и декремента (уменьшения на 1) -- имеют префиксную (перед операндом) и постфиксную (после операнда)

форму [5, п. п. 5.2.6, 5.3.2]. В первом случае результат операции — новое значение операнда, во втором — копия старого:

```
int subject = 5;
int prefixResult = ++subject;   // prefixResult = 6, subject = 6
int postfixResult = subject++;  // postfixResult = 6, subject = 7
```

3.5.3 Битовые операции

Битовые операции в C++ аналогичны имеющимся в Pascal, но имеют иные лексемы:

а) `~` (инверсия, логическое «не», **`not`**);

б) `&` (конъюнкция, логическое «и», **`and`**);

в) `|` (дизъюнкция, логическое «или», **`or`**);

г) `^` (исключающее «или», сложение по модулю 2, **`xor`**);

д) `>>` (сдвиг вправо, **`shr`**);

е) `<<` (сдвиг влево, **`shl`**).

Подробности их поведения см. в [5, п. 5.3.1, подразделы 5.8, 5.11—5.13]. Заметим, что основные трудности и ограничения составляет работа со знаковыми типами и отрицательными аргументами.

3.5.4 Логические операции

К логическим операциям относятся сравнения [5, подразделы 5.9—5.10] и операции над логическим типом [5, подразделы 5.14—5.15].

Сравнения чисел осуществляются теми же операторами, что и в Pascal, кроме проверки на равенство, лексема которой — двойное равенство (`==`), и на неравенство (`!=`). Распространенной ошибкой является употребление в условиях (в операторе **`if`**, **`while`** и т. п.) одного знака равенства вместо двух, что превращает сравнение в присваивание, компилируется без ошибок, но чаще всего неправильно (но бывает и намеренным).

При сравнении к типам применяются стандартные приведения (standard conversions) и преобразования к наиболее вместительному из типов (promotions), как это описано в [5, раздел 4, раздел 5 абзац 10]. Основную проблему представляет сравнение знаковых и беззнаковых целых, так как преобразование зависит от архитектуры ЦП [5, подраздел 4.7], и для x86 ведет к побитовому сравнению, а не сравнению значений. Таких сравнений рекомендуется избегать.

Операции над логическим типом — это конъюнкция `&&` (логическое «и», **`and`**), дизъюнкция `||` (логическое «или», **`or`**) и инверсия `!` (логическое «не», **`not`**). Для первых двух применяется *правило сокращенного вычисления (short-circuit evaluation):* если после вычисления первого операнда ясен результат всего выражения, второй операнд

не вычисляется. Исключающее «или» над логическим типом производится обычным оператором исключающего «или» (^).

3.5.5 Приведение типов

Преобразование типов бывает явным (explicit) и неявным (implicit). Подвид первого, *приведение типов (type cast)* используется, чтобы явно указать тип некоторого выражения, например, для обеспечения нужного размера переменной при битовых операциях. Приведение типов бывает следующих видов [5, подраздел 5.4]:

а) Оператор приведения типа **static_cast**<*тип*>(*выражение*) для попытки преобразовать *выражение* к *типу* на этапе компиляции с выдачей сообщения об ошибке, если это невозможно. Запрещается при помощи **static_cast** менять константность *выражения*.

б) Оператор приведения типа **const_cast**<*тип*>(*выражение*) для изменения константности *выражения*; менять что-либо еще, кроме модификатора, **const_cast** запрещается.

в) Формат приведения (*тип*)*выражение* для одновременного изменения типа выражения и снятия любых модификаторов (у *типа* они игнорируются).

г) Функциональная форма *тип*(*выражение*), которая для встроенных типов эквивалентна объявлению «невидимой» переменной *типа* с присваиванием ей значения *выражения*.

Существуют также другие способы приведения типов и аспекты названных, которые будут рассмотрены в последующих лекциях.

3.5.6 Тернарное выражение

Тернарное, или условное, выражение (conditional operator) имеет синтаксис: *условие* ? *выражение-1* : *выражение-2*. Результатом является значение *выражения-1*, если значение *условия* истинно, иначе — значение *выражения-2*. Вычисляется только то выражение, результат которого служит результатом тернарного оператора [5, подраздел 5.16]. Пример — вычисление силы сопротивления движению одним выражением:

```
bool considerAirResistance = false;   // Учитывать ли сопр. воздуха?
bool considerFriction = true;         // Учитывать ли вязкое трение?
float resistanceForce = -gravity +    // Сила сопротивления падению.
     considerAirResistance ? velocity*velocity*airFactor : 0.0f +
     (considerFriction ? velocity*frictionFactor : 0.0f);
```

3.5.7 Приоритет операций

Приоритет операций C++ следует из формальной грамматики языка [5, приложение A]. В таблице указаны приоритеты только тех операторов, которые были рассмотрены.

Полную версию таблицы см. (C++ Operator Precedence. — Электронный ресурс. — Режим доступа: http://en.cppreference.com/w/cpp/language/operator_precedence, свободный).

Таблица 4 — Приоритет рассмотренных операторов C++

Приоритет	Операторы	Значение	Ассоциативность
1	`++ --`	Инкремент и декремент (постфиксные)	Левая
	`()`	Вызов функции	
2	`++ --`	Инкремент и декремент (префиксные)	Правая
	`+ -`	Унарное отрицание	
	`! ~`	Логическое и побитовое отрицание	
	`(type)`	Приведение типов (любое)	
3	`* / %`	Умножение, деление, взятие остатка	Левая
4	`+ -`	Сложение и вычитание	
5	`<< >>`	Битовый сдвиг	
6	`< <=` `> >=`	Сравнение	
7	`== !=`	Проверка на (не)равенство	
8	`&`	Побитовое логическое «и»	
9	`^`	Побитовое исключающее «или»	
10	`|`	Побитовое логическое «или»	
11	`&&`	Логическое «и»	
12	`||`	Логическое «или»	
13	`?:`	Тернарное выражение	Правая
	`=`	Присваивание	

	`+= -=`
	`*= /=`
	`%=`
	`<<= >>=`
	`&= ^=`
	`|=`
		Совмещенное присваивание	

14	`,`	Запятая (выражения через запятую, результат — значение последнего)	Левая

3.6 Функции

3.6.1 Синтаксис объявления и реализации функции

Язык C++ поддерживает процедурную парадигму программирования, при которой программа делиться на подпрограммы — функции (термина «процедура» в C и C++ нет). Синтаксис определения функции [5, п. 8.4.1]:

```
тип-возвращаемого-значения имя-функции (
      тип-аргумента-1 аргумент-1, …, тип-аргумента-N аргумент-N) {
      тело-функции
}
```

Например, функция `power()`, принимающая два аргумента: вещественное основание и целочисленный показатель степени — может быть описана так:

```
1  double power(const double base, const unsigned int exponent) {
2        double result = 1.0;
3        for (unsigned int i = 0; i < exponent; ++i) {
4              result *= base;
5        }
6        return result;
7  }
8  // …
9  // Где-либо в программе:
10 double x = 5.0;
11 double y = power(x, 3); // y = 125.0
```

Листинг 2 – Функция вычисления целой степени числа

Оператор **`return`** приводит к выходу из функции с возвратом указанного значения. Если возвращаемого значения нет, в качестве его типа указывается **`void`**, а после **`return`** никакого значения не следует [5, п. 6.6.3].

При вызове функции (строки 10—11) ей передаются значения выражений-параметров, которые вычисляются до передачи управления телу функции в порядке, определяемом реализацией компилятора [5, п. 5.2.2].

Аргументы функции не обязаны быть константными, но хорошей практикой считается так делать, поскольку это защищает от их случайной перезаписи внутри функции, и программист уверен, что обращаясь к аргументу, он получит именно переданное значение. При этом, аргументы функции — отдельные переменные, и присваивание им не изменяет значения переменных вне функции.

Объявление функции *(прототип)* и ее реализацию можно разделять, и даже между несколькими файлами, как это показано в листинге 3:

```
1  // Прототип функции:
```

```
2  double power(const double base, const int exponent);
3  // Реализация функции (возможно, в другом файле):
4  double power(const double base, const int exponent) { … }
```

Листинг 3 — Пример разделения прототипа и реализации функции

Иногда отделение прототипа неизбежно: если функция `f()` должна вызвать функцию `g()`, а `g()` — `f()`, необходимо перед реализацией одной из них описать прототип другой.

Важный случай использования прототипов — описание функций, реализация которых находится во внешних библиотеках, или в других единицах трансляции (translation unit, по сути — в файлах *.cpp). Например, реализация стандартных функций находится в отдельной т. н. библиотеке времени выполнения (C run-time library, CRT). В этом случае компилятор только проверяет правильность их вызова, основываясь на прототипе, а с исполняемым кодом этих функций программу связывает *компоновщик (linker)*. Обычно группу прототипов размещают в одном заголовочном файле (`*.h`, `*.hpp`), а их реализацию — в одноименном компилируемом (`*.c`, `*.cpp`). В этом случае при смене реализации функции потребуется только перекомпилировать файл с ней и выполнить заново компоновку программы, что гораздо быстрее, чем полная перекомпиляция всех файлов, использующих изменившуюся функцию.

Если какие-либо из аргументов функции не используются, их имена можно опустить. Это часто применяется, чтобы избежать предупреждений компилятора.

Аргументам функций можно указывать значения по умолчанию [5, п. 8.3.6]. Например, если функция объявлена как `void f(int x = 5)`, то вызовы `f(5)` и `f()` эквивалентны. Значение аргумента по умолчанию допускается указывать только в одном месте области видимости (из всех прототипов и реализаций данной функции). На практике принято указывать значения по умолчанию только для последних аргументов в списке (возможно, нескольких).

3.6.2 Перегрузка функций

Перегрузкой функции называется определение и реализация нескольких функций с одинаковыми именами, но различными списками аргументов, и таким образом, по вызову функции компилятор определяет, какую из перегруженных функций использовать. Очевидно, что если для некоторых параметров заданы значения по умолчанию, выбрать правильную перегрузку может оказаться невозможно. Типы возвращаемых значений при перегрузке не учитываются [5, подразделы 13.1—13.3].

```
1  int power(int base, int exponent) { … }
2  int power(int base, unsigned int exponent) { … }
```

```
3  double power(double base, int exponent) { … }
4  // double power(int base, int exponent) { … }
5  // …
6  power(2, 5);    // int power(int, int)
7  power(2, 5u);   // int power(int, unsigned int)
8  power(2.0, 5);  // double power(double, int)
```

Листинг 4 — Перегрузка функции вычисления степени

В строках 1—3 описана перегрузка функции `power()` для двух наборов аргументов: или двух целых чисел, или вещественного и целого числа. Если бы строка 4 была раскомментирована, произошла бы ошибка компиляции: у функций на строках 1 и 4 были бы одинаковые наборы аргументов. В строках 6—8 в комментариях указано, какие варианты перегрузки будут выбраны компилятором.

3.6.3 Рекурсия

В теле функции допускается вызов других функций и той же функции [5, п. 5.2.2]. Последнее называется рекурсией. Рекурсия `main()` запрещена. Листинг 5 содержит улучшенную рекурсивную реализацию функции `power()` с возможностью вычислять отрицательные степени:

```
1  double power(const double base, const int exponent) {
2      double result = 1.0;
3      if (exponent < 0) {
4          return power(1.0 / base, -exponent);
5      } else {
6          for (unsigned int i = 0; i < exponent; ++i)
7              result *= base;
8      }
9      return result;
10 }
```

Листинг 5 — Функция вычисления целой степени числа

Рекурсивный вызов находится в строке 4: $base^{exponent} = \left(\frac{1}{base}\right)^{-exponent}$ для отрицательного показателя (exponent) преобразуется в вызов с положительным показателем.

Заметим, что количество вложенных вызовов `power()` никогда не превзойдет двух (говорят: *глубина рекурсии* не более двух). Действительно: если $exponent \geq 0$, сработает ветвь **`else`** — первый уровень, иначе будет вновь вызвана `power()`, с заведомым $exponent \geq 0$ — второй уровень. На практике глубина рекурсии часто зависит от данных и теоретически не ограничена, а также может возникать в разных местах тела функции. Листинг 6 показывает возможную рекурсивную реализацию функции `power()`:

```
1  double power(const double base, const int exponent) {
2        if (exponent < 0) {
3              return power(1.0 / base, -exponent);
4        } else if (exponent > 0) {
5              return base * power(base, exponent - 1);
6        }
7        return 1.0;
8  }
```

Листинг 6 — Рекурсивная функция вычисления целой степени числа

Рекурсивные вызовы расположены теперь в строках 4 и 6, при вызове функции производится один из них, но глубина рекурсии теперь достигает до $(|exponent| + 1)$ вложенных вызовов. Таблица 5 иллюстрирует процесс вычисления $2^{-2} = \left(\frac{1}{2}\right)^2 = \frac{1}{2}\left(\frac{1}{2}\right)^1 = \frac{1}{2} \cdot \frac{1}{2} \cdot \left(\frac{1}{2}\right)^0 = \frac{1}{2} \cdot \frac{1}{2} \cdot 1$ как серию вложенных вызовов.

Таблица 5 — Стек вызовов функции `power()` из листинга 6

Вызов	**Возвращаемое значение**
`power(2.0, -2)`	0,25
`power(1.0 / 2.0, 2)`	0,25
`power(0.5, 1)`	0,5
`power(0.5, 0)`	1,0

Большая глубина рекурсии нежелательна, и вот почему. При каждом вложенном вызове функции компилятор помещает значения аргументов в конец специальной области памяти — *стека* (от англ. *stack* — стопка). Разумеется, в реальной ЭВМ память, отведенная под стек, конечна, и переполнение стека в результате множества вызовов ведет к неопределенному поведению (UB), а де-факто — к аварийному завершению работы программы. Стек как область памяти не следует путать со *стеком вызовов (call stack)* — списком вложенных вызовов наподобие таблицы 5.

Очевидно, что любой рекурсивный алгоритм можно переписать без рекурсии (в итеративном виде), реализовав вручную подобие стека. Кроме того, современные оптимизирующие компиляторы способны в некоторых случаях преобразовывать рекурсивные вызовы в циклы.

3.6.4 Соглашения о вызове функций

При возврате из функции компилятор очищает часть стека, занятую ее аргументами. Это называется *раскруткой стека (stack unwinding).* Имеет значение, где расположен код, выполняющий очистку — внутри функции или вовне. Это особенно важно при вызовах

функций из внешних библиотек. Некоторые применяемые для платформы x86 соглашения о вызове приведены в таблице 6.

Таблица 6 — Некоторые соглашения о вызове платформы x86

Соглашение о вызове	**Ключевое слово (пишется перед именем функции)**	**Стек очищает**	**Порядок помещения аргументов в стек**	**Способ возврата значения**
stdcall	**`__stdcall`**	подпрограмма	справа налево (обратный)	регистр ЦП
cdecl	**`__cdecl`**	вызывающий код	справа налево (обратный)	регистр ЦП
pascal	**`__pascal`**	подпрограмма	слева направо (прямой)	стек
fastcall	**`__fastcall`**, **`__msfastcall`** или **`register`**	подпрограмма	определяется компилятором, используются регистры ЦП	

Детальное рассмотрение каждого соглашения о вызове выходит за рамки данного занятия. Соглашения *stdcall* и *pascal* используются обычно для функций в библиотеках, которые могут быть использованы из других языков программирования. Соглашение *fastcall* используется по умолчанию компиляторами для вызовов в пределах программы.

Результат неправильного обращения со стеком — например, если вызывающий код и функция используют разные соглашения о вызове, — называется *дисбалансом стека.* Так как на стеке часто сохраняются не только значения аргументов, но и сведения о том, куда должно перейти выполнение кода после возврата из функции (адрес возврата), дисбаланс стека может привести к серьезным проблемам — может оказаться выполнен, фактически, произвольный код.

3.6.5 Функции с переменным числом аргументов

Соглашение *cdecl* принято, в основном, в компиляторах C и C++ и позволяет создавать эффективные функции с переменным числом аргументов. Например, прототип стандартной функции `printf()` выглядит так:

```
void printf(char* format, ...);
```

Здесь «`...`» — не сокращение, а ключевое слово, указывающее на переменное число аргументов. Узнав положение на стеке последнего из именованных аргументов (в данном случае — `format`), можно утверждать, что ниже расположены неименованные. При этом корректное извлечение из стека всех переданных аргументов — задача вызывающего кода,

так как `printf()` это самостоятельно сделать не в состоянии — нет данных о том, сколько и каких аргументов было реально передано. Рассмотрение конкретных механизмов реализации функций с переменным числом аргументов выходит за рамки данного занятия; подчеркнем, что это выполняется сугубо средствами языка, не встроено в компилятор и может быть выполнено прикладным программистом [5, п. 8.3.5, 18.10].

Функция `printf()` стандартной библиотеки C и C++ аналогична функции `fprintf()` в MATLAB за исключением возможности обрабатывать векторы и матрицы.

3.6.6 Статические переменные в функциях

Иногда требуется, чтобы внутренняя переменная функции сохраняла свое значение между вызовами на протяжении всей программы. Для этого необходимо объявить такую переменную с ключевым словом **`static`**, как показано в листинге 7.

```
1  int triggerCount() {
2        static int count = 0;
3        count++;
4        return count;
5  }
6  // …
7  int one = triggerCount();  // one == 1
8  int two = triggerCount();  // two == 2
```

Листинг 7 — Применение статической переменной в функции

Статическая переменная `count` объявлена на строке 2 с начальным значением 0, а затем увеличивается на 1 при каждом вызове функции [5, п. 7.1.1, 3.7.1, подраздел 6.7].

Статические переменные считаются сомнительной практикой в C++, и вот почему. Во-первых, они нарушают независимость функции от внешнего кода, запутывая логику работы программы. Во-вторых, в современных многопоточных программах статические переменные нарушают принцип *reenterability*, то есть препятствуют одновременному выполнению функции (с разными аргументами) параллельно, так как статическая переменная одна и та же для всех потоков. Наконец, использование статических переменных принадлежит процедурному стилю программирования и служит для имитации возможностей объектно-ориентированного подхода, предпочитаемого в C++.

Мы приводим описание этой возможности сугубо потому, что ее применение встречается на практике и бывает оправданно в процедурно-ориентированной программе.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Брайан Керниган, Деннис Ритчи. Язык программирования C / пер. с англ. В. Бродовой (The C Programming Language) — М.: Вильямс. — 2012 г. — 304 с.
2. Бьерн Страуструп. Дизайн и эволюция языка C++ / пер. с англ. (The Design and Evolution of C++) — ДМК Пресс. — 2009 г. — 448 с.
3. JTC1/SC22/WG21 - The C++ Standards Committee [Электронный ресурс.] — Последнее изменение: 7 июля 2013 г. — Режим доступа: http://open-std.org/JTC1/SC22/WG21/, свободный.
4. ISO/IEC 14882:2011 - Information technology -- Programming languages -- C++ [Страница в интернете.] — 2011 г. — Режим доступа: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372, свободный.
5. ISO/IEC 14882:2011 Programming Languages — C ++. — Введен в 2011 г. — 1338 с.
6. Стивен Прата. Язык программирования C++. Лекции и упражнения, 6-е издание / пер. с англ. (C++ Primer Plus, Sixth Edition) — М.: Вильямс. — 2012 г. — 1248 с.