тольяттинский
государственный
университет

Е.С. Глибин

ЯЗЫКИ ВЫСОКОГО УРОВНЯ В СИСТЕМАХ УПРАВЛЕНИЯ

Учебное пособие

Тольятти
Издательство ТГУ
2025

Министерство науки и высшего образования
Российской Федерации
Тольяттинский государственный университет

Е.С. Глибин

ЯЗЫКИ ВЫСОКОГО УРОВНЯ В СИСТЕМАХ УПРАВЛЕНИЯ

Учебное пособие

Тольятти
Издательство ТГУ
2025

УДК 004.432:004.31(075.8)
ББК 32.973.22я73
Г 54

Рецензенты:
д-р техн. наук, профессор Поволжского государственного университета сервиса *Б.М. Горшков*;
канд. техн. наук, доцент, доцент кафедры «Промышленная электроника» Тольяттинского государственного университета *А.В. Прядилов*.

Г 54 Глибин, Е.С. Языки высокого уровня в системах управления : учебное пособие / Е.С. Глибин – Тольятти : Издательство ТГУ, 2025. – 143 с. – ISBN 978-5-8259-1765-8.

В учебном пособии изложены сведения о программировании на языке Си++ микроконтроллеров, миникомпьютеров, а также программ для ЭВМ в целом. Рассмотрены основные синтаксические конструкции языка Си++ с точки зрения написания программного обеспечения современной встраиваемой электроники. Представлены примеры применения рассмотренных конструкций языка для решения практических задач. Определены ключевые аспекты опроса датчиков, управления данными в памяти процессора, управления исполнительными устройствами.

Предназначено для студентов, обучающихся по направлению подготовки бакалавров 11.03.04 «Электроника и наноэлектроника» (профили «Промышленная электроника» и «Электроника и робототехника») очной и заочной форм обучения, в том числе с использованием ДОТ.

УДК 004.432:004.31(075.8)
ББК 32.973.22я73

Рекомендовано к изданию научно-методическим советом Тольяттинского государственного университета.

ISBN 978-5-8259-1765-8

ПРЕДИСЛОВИЕ

В настоящее время сердцем многих электронных схем является микроконтроллер. Эта микросхема работает по-разному в зависимости от загруженной в нее программы. Одна и та же модель микроконтроллера может быть использована для управления, например, и автомобилем, и бытовым кондиционером. Программное обеспечение, которое напрямую взаимодействует с оборудованием, называется встраиваемым. И пишется оно чаще всего на языке программирования Си.

Основная цель изучения дисциплины «Языки высокого уровня в системах управления» – формирование профессиональных компетенций, необходимых для разработки встраиваемого программного обеспечения на языках Си и Си++.

Поставленная цель решается с помощью следующих задач:

1) познакомиться с синтаксисом языков Си и Си++;
2) научиться использовать их при разработке электронных устройств.

Учебное пособие предназначено для студентов бакалавриата направления подготовки 11.03.04 «Электроника и наноэлектроника» (профили «Промышленная электроника» и «Электроника и робототехника»), в том числе с использованием ДОТ.

В учебном пособии рассматриваются основные возможности языков Си и Си++ с точки зрения разработки встраиваемого программного обеспечения. Дисциплины и практики, на освоении которых базируется данная дисциплина: «Электронные измерительные приборы и датчики информации», «Информатика», учебная (ознакомительная) практика. Дисциплины, для которых освоение данной дисциплины необходимо как предшествующее: «Информационная электроника», «Системы компьютерного зрения», «Программируемые контроллеры».

ВВЕДЕНИЕ

Учебное пособие включает в себя четыре главы.

В первой главе «Разработка встраиваемого программного обеспечения» рассматриваются простые программы на языках Си и Си++ как для ОС Windows, так и для микроконтроллеров семейства AVR. Также описывается, какие этапы разработки программ существуют и какой набор инструментов понадобится для их написания.

Во второй главе «Основы языка Си» рассматриваются основы синтаксиса языка, в частности описывается работа с переменными и константами, с операциями и операторами, с массивами и функциями, с указателями. Основное внимание посвящено тем вопросам, которые обычно встречаются при программировании микроконтроллеров, например применению битовых масок при работе с регистрами.

В третьей главе с названием «Си++» изучаются основы объектно-ориентированного программирования. В современной электронике язык Си++ часто используют в мини-компьютерах, например Raspberry Pi. Здесь программы решают задачи компьютерного зрения и создания искусственного интеллекта.

Четвертая глава называется «Стандартная библиотека Си++». Здесь рассматриваются решения типичных задач программирования, например работа с динамическими массивами и файлами.

Глава 1. РАЗРАБОТКА ВСТРАИВАЕМОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Простые программы на языке Си++

В настоящем учебном пособии будут рассмотрены ключевые возможности языка Си и его наследника Си++. Язык Си++ является высокоуровневым компилируемым языком программирования.

Высокоуровневый язык программирования означает, что программы на нем отличаются высокой степенью абстракции. То есть оперируют такими смысловыми конструкциями для краткого описания структур данных и операций над ними, описание которых на низкоуровневом языке программирования или машинном коде было бы длинным и трудным для понимания. Другими словами, простой оператор сложения на языке Си++, например такой:

```
int sum = 4 + 5 + 8;
```

на низкоуровневом языке ассемблере для AVR-микроконтроллера выглядел бы примерно так:

```
ldi R16, 4 ; регистру R16 присваивается значение «4»
ldi R17, 5 ; регистру R17 присваивается значение «5»
ldi R18, 8 ; регистру R17 присваивается значение «8»
add R17, R16 ; сложить содержимое R17 и R16 и результат поместить в R17
add R18, R17 ; сложить содержимое R18 и R17 и результат поместить в R18
```

То, что Си++ является компилируемым языком, означает, что каждый файл с исходным текстом (.cpp) должен быть обработан с помощью программы-компилятора (compiler) для получения объектных файлов (.obj), файлов промежуточных представлений отдельных модулей (.cpp), которые собираются программой-компоновщиком (linker) в выполнимую программу (например, .exe).

Для создания выполняемых программ используются другие программы, которые относятся к инструментальному программному обеспечению (выделяют системное ПО, например ОС Windows или драйверы устройств, прикладное, или приложения, и инструментальное). Исторически сложилось, что для создания испол-

няемого файла требуется минимум две программы – компилятор и компоновщик. Первые ЭВМ не могли одновременно держать в памяти компилятор, текст большой программы и выполняемый код. Поэтому приходилось компилировать по частям: весь исходный код разбивался на отдельные файлы, каждый файл являлся отдельной так называемой единицей трансляции, которая обрабатывалась компилятором, получался более компактный объектный файл. Однако в нем имелись ссылки или связи (link) на код из других единиц трансляции. Поэтому на финальном этапе другая программа-компоновщик, она же редактор связей, собирала весь код программы в единое целое.

Сегодня данный подход остался, но применяется для повторного использования кода (использования библиотек кода), для разбиения программы на отдельные логические части, или модули, для ускорения сборки программы. Так как изменения в тексте одной единицы трансляции не затрагивают другие, нет необходимости перекомпилировать все файлы с кодом в случае изменения кода в одном отдельном файле .cpp.

В программном обеспечении набор инструментов программирования (toolchain) – это набор инструментальных программ, который используется для выполнения сложной задачи разработки программного обеспечения. Обычно набор инструментов выполняется последовательно, поэтому выходные данные каждого инструмента становятся входными данными для следующего.

Набор инструментов программирования включает (но не ограничивается):

1) компилятор;
2) компоновщик;
3) библиотеки кода;
4) отладчик (используется для тестирования и отладки получившейся программы).

Сегодня обычно нет необходимости запускать вручную компилятор для компиляции отдельных файлов – на практике используются интегрированные среды разработки (Integrated development environment – IDE), обеспечивающие удобный графический интерфейс для доступа к набору инструментов.

Среда разработки включает в себя:

1) редактор текста;
2) транслятор (это компилятор и/или интерпретатор);
3) средства автоматизации сборки;
4) отладчик.

При первом знакомстве с IDE может показаться, что для создания выполняемой программы используется всего лишь одна программа, но это не так. Например, одинаковый редактор текста используется в Visual Studio для создания программ для персонального компьютера и в Microchip Studio для создания прошивок AVR микроконтроллеров, но компиляторы очевидно различаются.

Изучение Си и Си++ начнем с простых программ. Для их создания используем Visual Studio Community 2019 с установленным Visual C++.

Загрузить последнюю версию Visual Studio Community с официального сайта Microsoft можно по ссылке (рис. 1.1): https://visualstudio.microsoft.com/ru/free-developer-offers/

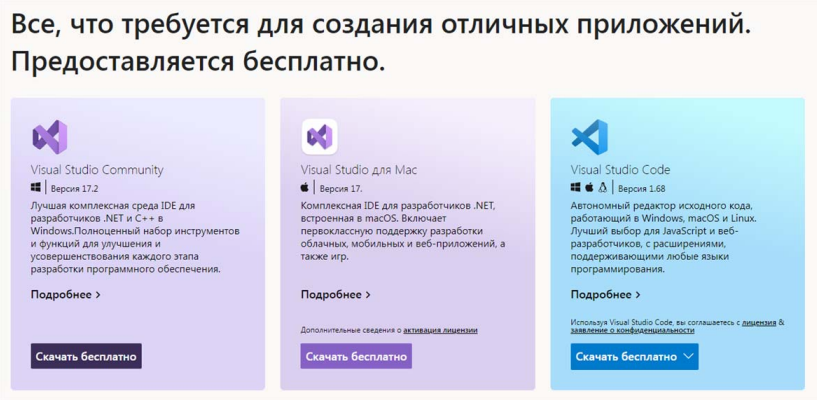

Рис. 1.1. Загрузка Visual Studio

При установке не забудьте установить именно Си++ (*Разработка классических приложений на C++*), а не другой язык программирования, как показано на рис. 1.2.

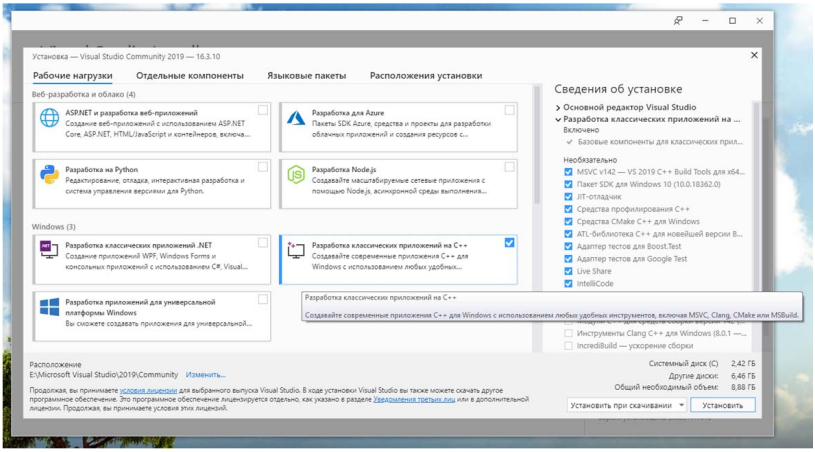

Рис. 1.2. Настройки установки

После установки и запуска интегрированной среды разработки вы увидите стартовое окно, похожее на то, что показано на рис. 1.3.

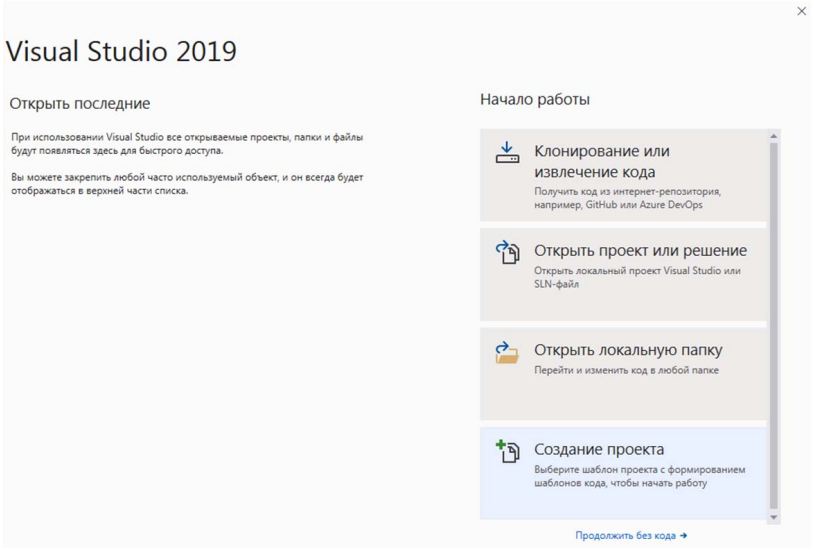

Рис. 1.3. Стартовое окно среды разработки Visual Studio

Нажмите *Создание проекта* и в следующем окне выберите пункт *Консольное приложение* (рис. 1.4).

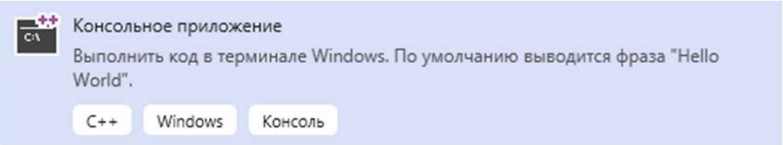

Рис. 1.4. Выбор типа проекта

В Microsoft Visual Studio для организации работы служат проекты и решения. Обычно программа содержится не в одном файле, а в нескольких. Данная совокупность файлов называется проектом. Решение может содержать несколько проектов, например библиотеку DLL и ссылающийся на нее исполняемый EXE-файл. Чаще одному решению соответствует один проект.

Назовите проект HelloWorld1 и создайте его, оставив все другие настройки без изменений. Если все выполнено корректно, то появится окно, подобное представленному окну на рис. 1.5.

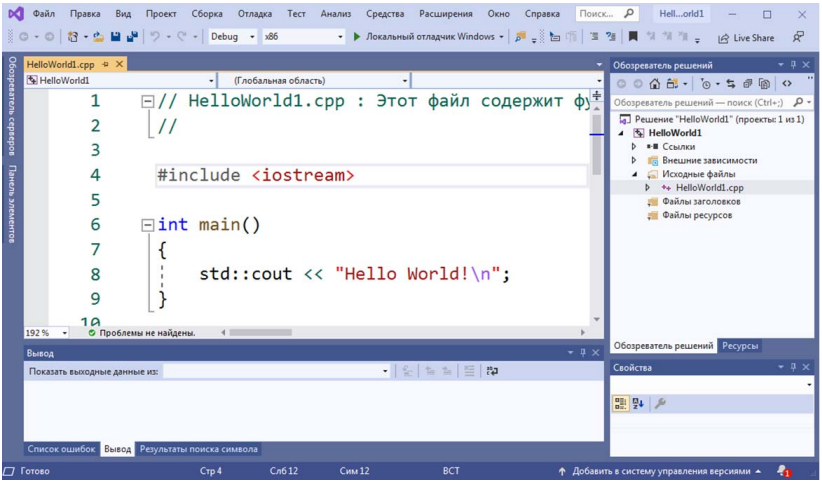

Рис. 1.5. Рабочая область интегрированной среды разработки Visual Studio с проектом HelloWorld1

Текст зеленого цвета, начинающийся с // и до конца строки, является комментарием в языке Си++ и на работу программы никак не влияет. Обозреватель решений справа показывает отсортированные файлы проекта, в нашем случае только один файл с исходным

кодом HelloWorld1.cpp, который и будет компилироваться, а затем собираться в исполняемую программу HelloWorld1.exe. Если случайно вы закрыли *Обозреватель решений*, его всегда можно открыть, используя соответствующий пункт меню *Вид*.

Собственно, сама среда разработки создала проект с одним файлом и добавила туда код, показанный в листинге 1.1. Сгенерированные Visual Studio комментарии в листинге не показаны.

Листинг 1.1

```
#include <iostream>

int main()
{
    std::cout << “Hello World!\n”;
}
```

Чтобы скомпилировать и собрать программу, выполните команду *Сборка* → *Собрать решение*. Будет выполнен запуск компилятора, а затем автоматически – компоновщика. Файлы настроек проекта, решения, файлы с исходным кодом, получившиеся объектные файлы, другие файлы, а также собственно исполняемый файл программы можно найти в папке с решением на диске. Путь к папке с решением указывается при создании проекта, а также ее можно открыть из *Обозревателя решений*. Для этого правой кнопкой мыши щелкните на *Решение 'HelloWorld1'* (проекты 1 из 1) и выберите *Открыть папку в проводнике*. Также запустить получившуюся программу можно сразу из среды разработки, выполнив пункт меню *Отладка* → *Начать отладку* или нажав F5. Можно сразу нажать F5, будет произведена сборка решения и, если нет ошибок, запущена программа.

Результат работы программы HelloWorld1 показан на рис. 1.6.

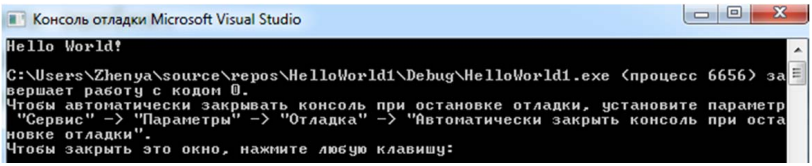

Рис. 1.6. Программа HelloWorld1

При изучении основ программирования обычно используются консольные приложения с текстовым интерфейсом.

Кроме Visual Studio, в случае простых программ с одним файлом исходного кода, которых в настоящем курсе будет большинство, можно использовать онлайн-компилятор. Например, показанный на рис. 1.7. Он поддерживает много языков программирования, при переключении языка будет сгенерирован код простой программы, выводящий надпись Hello World в консоль. В случае использования онлайн-компилятора достаточно набрать текст программы в редакторе и нажать на кнопку *Run*.

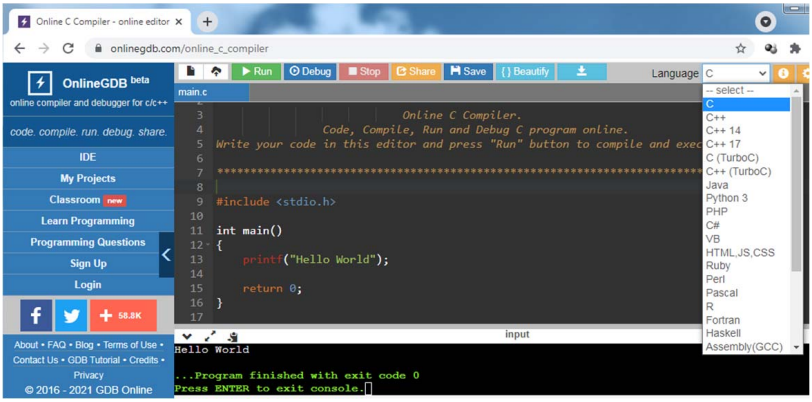

Рис. 1.7. Онлайн-компилятор

Вернемся к программе в листинге 1.1. Первая ее строчка:

```
#include <iostream>
```

является директивой препроцессора. Директивы препроцессора начинаются с символа # и являются не командами вашей программы процессору, который ее будет выполнять, а указанием компилятору, как обрабатывать исходный код программы. Упрощенно пока примем, что данная директива подключает к программе библиотеку кода для вывода текста в консоль и ввода текста с клавиатуры. Без нее не будет работать

```
std::cout << «Hello World!\n»;
```

Далее описывается главная функция main().

Вообще, минимальная программа C++ выглядит следующим образом:

```
int main()
{
}
```

Однако она ничего не делает и ничего не выводит на экран. Выполняемые команды в Си располагаются в функциях, подпрограммах с определенным именем (main в примере), которым передаются определенные данные, называемые аргументами, в параметры функции, описанные в круглых скобках (). Функция выполняет некоторые вычисления и/или действия, описанные внутри фигурных скобок {}, и возвращает результат определенного типа, например int, принимающий целочисленные значения определенного диапазона.

int main() – заголовок функции, код внутри {} – тело функции.

Компилятор компилирует исходный текст, основываясь на синтаксических конструкциях языка, и дополнительные пробелы, символы табуляции, новые строки в тексте программы на ее компиляцию не влияют. Так, например, минимальную программу можно записать так:

```
int main() {}
```

или так

```
int main() {
}
```

Отличия стилистические, программа будет работать одинаково.

Функция с именем main – особенная, называется главной, и она должна быть в программе. С нее начинается выполнение программы. Языки Си и Си++ являются регистр-зависимыми: main не то же самое, что Main, а int не то же самое, что INT, и т. д. В примере функция не имеет параметров, в круглых скобках ничего нет. Функция возвращает значение типа int. Возврат значения из функции осуществляется оператором return:

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

Оператор, то есть наименьшая автономная часть языка программирования, или команда, заканчивается знаком «;». Возвращает main() функция 0, что операционной системой интерпретируется как нормальное завершение работы. Согласно стандарту языка Си++ в главной функции main() оператор «return 0;» можно не указывать. Она особенная и неявно возвращает 0, если вы не указали явно «return 0;».

```
std::cout << “Hello World!\n”;
```

Программа выводит на экран строку «Hello World!\n». «std» является пространством имен, а «std::» указывает, что элемент cout относится к стандартной библиотеке языка Си++ (не Си!). На самом деле в основе понимания работы данной конструкции лежат такие понятия объектно-ориентированного программирования, как классы и объекты, перегрузка операций, а также потоки ввода-вывода. Вернемся к ней в последнем параграфе данного учебника, но пока придется принять без объяснений, что так выводится текст на экран. Далее аналогично рассмотрим, как выводить значения переменных на экран и считывать их значения с клавиатуры.

Два символа «\n» при выводе текста интерпретируются как переход на новую строку и не выводятся. Собственно, если потребуется вывести «\», в строке необходимо записать «\\».

В литературе можно встретить такой вариант программы HelloWorld1:

```
#include <iostream>

using namespace std;

int main()
{
    cout << “Hello World!\n”;
}
```

Отличие заключается в том, что мы указываем, что будем использовать пространство имен стандартной библиотеки, и можем использовать далее cout, а не std::cout.

Вариант программы HelloWorld для языка Си показан в листинге 1.2.

Листинг 1.2

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Данная программа скомпилируется и компилятором Си++, но для микроконтроллеров часто используется только язык Си, в котором нет стандартной библиотеки Си++ (std). Для ввода-вывода в консоль в языке Си подключается файл stdio.h и вызывается функция printf().

Рассмотрим пример, в котором пользователь вводит значения в программу с клавиатуры, программа выполняет вычисления и выводит результат на экран. Пример показан в листинге 1.3.

Листинг 1.3

```
#include <iostream>

int main()
{
    double x;
    std::cout << "Enter x: ";
    std::cin >> x;
    double y = x * x;
    std::cout << "x * x = " << y << "\n";
}
```

Пользователь вводит с клавиатуры число, программа выводит это число, возведенное в квадрат. Используем Си++. В программе объявляется переменная типа double и с именем x. В переменных типа double можно хранить вещественные числа, например 4.5.

Выводим на экран предложение пользователю ввести x:

std::cout << «Enter x: «;

Ожидаем от пользователя, когда он введет число с клавиатуры, и нажимаем на клавишу Enter:

std::cin >> x;

Введенное значение помещается в переменную x.

Далее вводим новую переменную с именем y, тоже типа double, и сразу присваиваем ей значение выражения x * x (* – умножение). И выводим результат на экран:

```
std::cout << «x * x = « << y << «\n»;
```

Это же действие можно было бы выполнить в три строчки:

```
std::cout << “x * x = “;
std::cout << y;
std::cout << «\n»;
```

Результат работы программы показан на рис. 1.8.

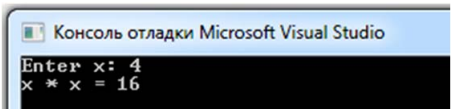

Рис. 1.8. Возведение числа в квадрат

Аналогичная программа на языке Си приведена в листинге 1.4.

Листинг 1.4

```
#include <stdio.h>

int main()
{
    double x;
    printf(“Enter x: “);
    scanf(“%lf”, &x);
    double y = x * x;
    printf(“x * x = %lf\n”, y);
    return 0;
}
```

Здесь для ввода с клавиатуры используется стандартная функция языка Си, объявленная в файле stdio.h:

```
scanf(«%lf», &x).
```

«%lf» означает, что во введенном пользователем тексте будет осуществлен поиск вещественного числа и его значение будет сохранено в переменную типа double. Адрес этой переменной в памяти x передается вторым аргументом. Операция & вычисляет адрес переменной в оперативной памяти и будет рассмотрена позже. Почему передается адрес переменной, а не ее значение? Удобно

считать, что функции Си похожи на математические функции, то есть запись y = sin(x) в математике не предполагает, что x может измениться в результате вычисления синуса. Поэтому в функцию передаются значения или аргументы и присваиваются параметрам функции, то есть функция работает с копиями аргументов и их изменить не может. Но в данном случае требуется именно изменить значение аргумента. Передается адрес ячейки в памяти, где хранится переменная, функция меняет содержимое этой ячейки, и переменная x меняется. Почему нельзя использовать возвращаемое функцией значение и присвоить его x, как в случае sin? Иногда функции может потребоваться вернуть несколько значений, мы просто рассмотрели простой вариант использования scanf().

В вызове функции «printf(«x * x = %lf\n», y);» текст «%lf» заменен на значение y. Поэтому функция printf называется форматированной печатью (print formatted).

Значение нескольких переменных можно вывести так:

printf("x = %lf, y = %lf \n", x, y);

Обе переменные x и y должны быть типа double. Для целочисленных переменных типа int необходимо вместо «%lf» использовать «%d», для float – «%f», а для строк – «%s».

Программа, показанная в листинге 1.4, может работать в онлайн-компиляторе. Но Visual C++ последних версий выдаст ошибку, сообщая о небезопасности функции scanf(). Ее некорректное использование может привести к ошибкам в процессе работы программы. Можно использовать безопасную реализацию функции от Microsoft. Она не является частью стандарта Си и в других компиляторах работать не будет. Пример использования показан в листинге 1.5.

Листинг 1.5

```
#include <stdio.h>

int main()
{
    double x;
    printf("Enter x: ");
    scanf_s("%lf", &x);
```

```
    double y = x * x;
    printf("x * x = %lf\n", y);
    return 0;
}
```

При попытке заменить выводимый текст кириллическим результат в Visual Studio может быть некорректным, как показано на рис. 1.9.

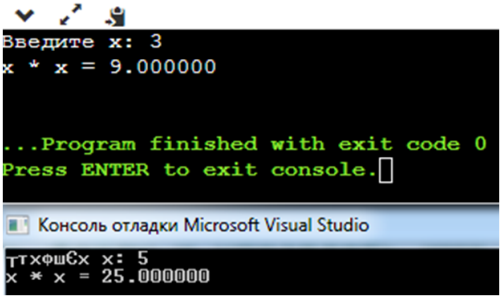

Рис. 1.9. Проблемы с отображением кириллицы

Необходимо изменить локальные настройки, как показано в листинге 1.6.

Листинг 1.6

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    setlocale(LC_ALL, "Russian");
    cout << «Введите x: «;
    cin >> x;
    double y = x * x;
    cout << "x в квадрате = " << y << "\n";
}
```

В завершение темы простых программ на Си++ рассмотрим вариант HelloWorld для микроконтроллеров – мигающий светодиод.

Пример такой программы для Arduino показан в листинге 1.7.

Листинг 1.7

```
void setup()
{
    pinMode(13, OUTPUT);
}

void loop()
{
    digitalWrite(13, HIGH);
    delay(1000); // Wait for 1000 millisecond(s)
    digitalWrite(13, LOW);
    delay(1000); // Wait for 1000 millisecond(s)
}
```

В программе, казалось бы, нет главной функции main() согласно стандарту Си++. На самом деле скетч Arduino является лишь одной единицей трансляции. А главная функция скрыта внутри библиотеки Wiring и выглядит примерно так (немного упрощенно, файл arduino-1.0.5-r2\hardware\arduino\cores\arduino\main.cpp):

```
int main(void)
{
    setup();

    for (;;)
    {
            loop();
    }
}
```

Очевидно, что без описания функций setup() и loop() в скетче компоновщик не сможет собрать полноценную прошивку микроконтроллера. Заголовок функции int main(void) аналогичен int main(), просто более наглядно показывает, что функция не имеет параметров.

Для написания прошивки без использования Arduino воспользуемся Atmel Studio 7.

Создадим новый проект аналогично проекту для Visual Studio, как показано на рис. 1.10. Назовем его Blink.

В следующем диалоговом окне необходимо выбрать модель микроконтроллера. Arduino UNO R3 используется микроконтроллер Atmega328p, выберем его (рис. 1.11).

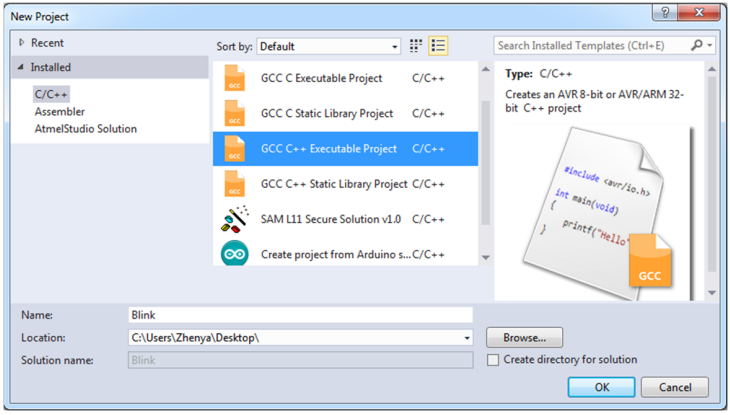

Рис. 1.10. Создание проекта в Atmel Studio 7

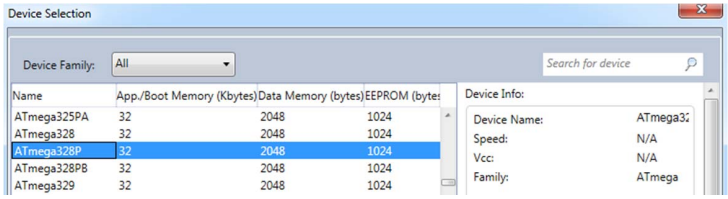

Рис. 1.11. Выбор микроконтроллера

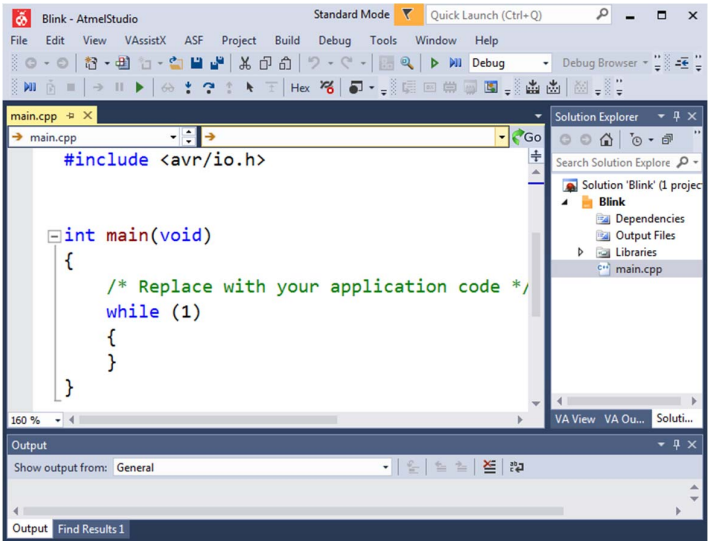

Рис. 1.12. Рабочая область Atmel Studio 7
с загруженным проектом Blink

После создания проекта будет сгенерирован шаблон простой программы, как показано на рис. 1.12.

for (;;) и while (1) являются операторами цикла, код внутри {} будет выполняться, пока истинно условие цикла. В первом случае оно отсутствует, во втором оно всегда истинно (0 – ложь, не ноль – истина). Оба цикла будут выполняться бесконечно, пока работает микроконтроллер.

Изменим начальную программу, которая не делает ничего, на свою, приведенную в листинге 1.8.

Листинг 1.8

```
#include <avr/io.h>
#define F_CPU 1000000UL
#include <util/delay.h>

int main(void)
{
    DDRB = 32;
 /* Replace with your application code */
 while (1)
 {
            PORTB = 32;
            _delay_ms(500);
            PORTB = 0;
            _delay_ms(500);
 }
}
```

В первой строчке подключается файл avr/io.h с объявлением регистров цифрового порта ввода-вывода, в том числе DDRB и PORTB:

#define *F_CPU* 1000000UL

Здесь вводится определение препроцессора *F_CPU,* равное 1000000. Это выбранная частота работы микроконтроллера, которая требуется для корректной работы функции задержки:

_delay_ms(500);

По сути, это означает, что любой встреченный текст «*F_CPU*» в программе будет заменен (только в памяти, не в файле с кодом!) перед компиляцией на текст «1000000UL». Текст для замены находится в файле util/delay.h. Он позволяет корректно подсчитывать

число циклов паузы в *_delay_ms*() для создания задержки нужной длительности.

Далее в главной функции устанавливается значение регистра DDRB, которое упрощенно пока можно представить как переменную. Каждый бит регистра DDRB имеет назначение (табл. 1.1).

Таблица 1.1

Биты регистров DDRB и PORTB

PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0	0	1	0	0	0	0	0

Число 32 в десятичной системе счисления – это 00100000 в двоичной, или, другими словами, только бит PB5 будет установлен в 1, остальные биты будут равны 0. Если взглянуть на принципиальную схему платы Arduino Uno R3 (найдите самостоятельно на официальном сайте), то можно узнать, что встроенный светодиод подключен к выводу 13 платы и выводу PB5 микроконтроллера. Последний управляется битом PB5 регистров DDRB и PORTB. Установка 1 в бит регистра DDRB устанавливает вывод в режим выхода подобно тому, как это делает функция pinMode(). А запись 1 в соответствующий бит PORTB уже устанавливает высокий логический уровень PB5.

В этом можно убедиться, используя скетч для Arduino (листинг 1.9). Самостоятельно проверьте его работу в онлайн-эмуляторе.

Листинг 1.9

```
void setup()
{
    DDRB = 32;
}

void loop()
{
    PORTB = 32;
    _delay_ms(500);
    PORTB = 0;
    _delay_ms(500);
}
```

Для отладки программы в Atmel Studio 7 можно использовать встроенный эмулятор. Его необходимо выбрать в качестве инструмента отладки в свойствах проекта (рис. 1.13).

Для отладки необходимо установить точки остановки, щелкнув на сером поле слева или нажав на F9. Появится красный кружок. Далее можно запустить отладку с помощью F5.

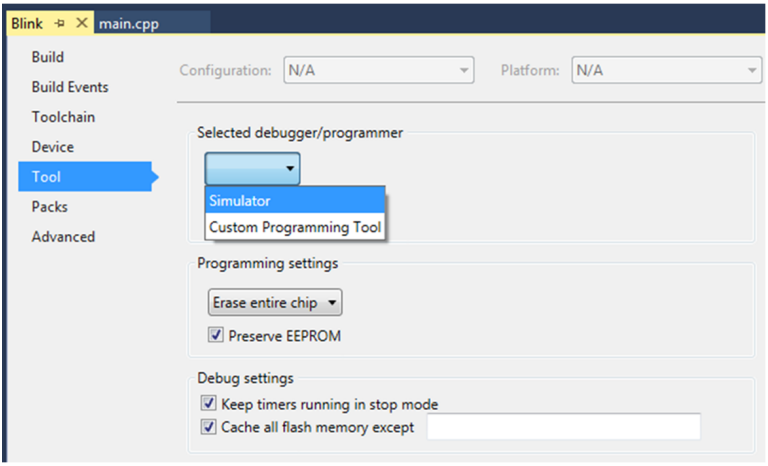

Рис. 1.13. Выбор эмулятора

Как только выполнение программы дойдет до точки остановки, она остановится, а в информационных окнах можно будет увидеть состояние выводов микроконтроллера. Продолжить выполнение можно, нажав на F5 (рис. 1.14).

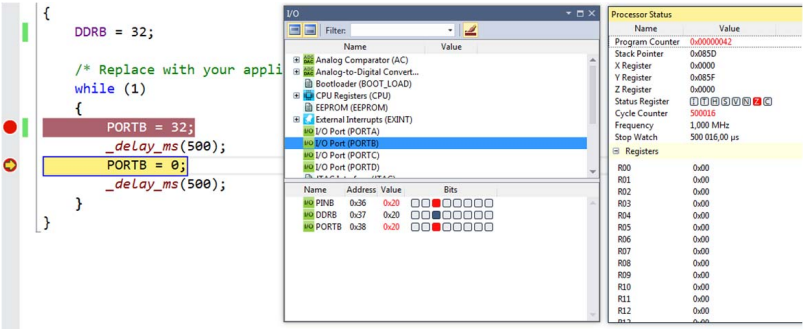

Рис. 1.14. Отладка мигающего светодиода

1.2. Основные этапы проектирования программ. Набор инструментов программирования

Выделяют следующие основные этапы проектирования программ:

1. Разработка спецификации.
2. Проектирование программы.
3. Доказательство правильности проекта.
4. Кодирование.
5. Отладка и тестирование.
6. Доработка и улучшение программ.
7. Производство окончательной программы.
8. Поддержка программы в процессе ее использования.

Начинается проектирование с разработки спецификации.

Спецификация программы – подробное описание всех действий, которые она должна выполнять.

На этом этапе необходимо ответить на следующие вопросы:

– Какими должны быть входные данные?
– Какие данные корректные, какие ошибочные?
– Кто будет использовать программу?
– Каким будет пользовательский интерфейс?
– Какие ошибки в работе программы нужно будет выявлять?
– Какие упрощения будут приняты в работе программы?
– Какие могут возникнуть особые ситуации?
– Какие должны быть выходные данные?
– Как будет оформлена документация?
– Как в будущем развивать и улучшать программу?

Далее необходимо спроектировать программу действий в следующем порядке:

1. Построить структуру программы.
2. Разработать алгоритмы.
3. Решить все вопросы по организации данных.

На данном этапе определяются модули программы.

Модульность – способность разделения.

Каждый модуль программы обладает следующими свойствами:

1. Интерфейс – средство общения с объектом.
2. Потоки данных – часть интерфейса.
3. Реализация – внутреннее устройство объекта.

Например, программа, осуществляющая управление нагревателем, может быть разбита на следующие модули:

- модуль датчика температуры (TmpSensor) – может использоваться цифровой или аналоговый, с простым или сложным интерфейсом;
- модуль системы управления нагревателем (heater) – может использоваться ПИД-регулирование или двухпозиционное;
- модуль пользовательского интерфейса, с помощью которого он задает температуру (UI), – может использоваться дисплей или приложение на телефоне;
- основная программа, реализующая работу нагревателя (main).

При этом проект программы может включать файлы исходного кода, показанные в *Обозревателе решений* на рис. 1.15.

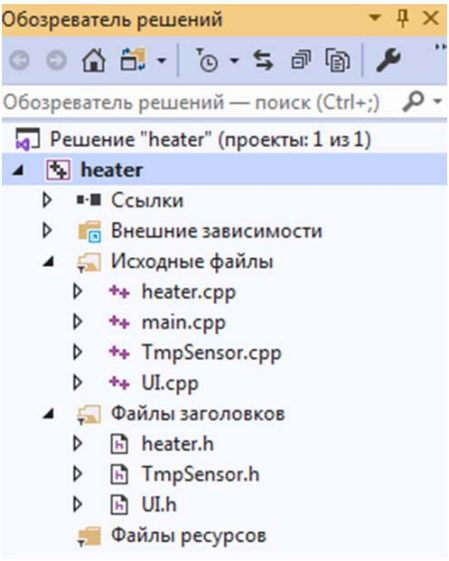

Рис. 1.15. Проект нагревателя

На этом этапе необходимо ответить на следующие вопросы:

– Какие данные можно передать в модуль до начала работы?
– Какие упрощения будут приняты в работе программ?
– Что будет с данными по завершении работы?

После разработки общей структуры программы необходимо выполнить следующие действия:

1) определение библиотечных средств, которые можно использовать;
2) разработка новых структур данных и алгоритмов выполнения новых процедур;
3) доказательство правильности проекта;
4) кодирование;
5) отладка и тестирование;
6) доработка и улучшение программ;
7) производство окончательной программы;
8) поддержка программы в процессе ее использования.

Обычно программы на языке Си++ состоят их многих файлов с исходными текстами (обычно именуемыми просто исходными файлами).

Главный файл main.cpp:

```
// файл main.cpp
#include “heater.h”
#include “UI.h”

int main()
{
        init(); // начальная инициализация
        while (true)
        {
                double userTemp = getUserTemp();
                controlHeater(userTemp); // управление нагревом
        }
}
```

Главная функция выполняет всего несколько действий: инициализирует систему управления вначале с помощью init(), далее в бесконечном цикле получает от модуля пользовательского интерфейса температуру UI с помощью getUserTemp() и передает

ее в модуль системы управления нагревателем heater, в функцию controlHeater().

В свою очередь, файл heater.cpp выглядит так:

```
// файл heater.cpp
#include “TmpSensor.h”

#define DELTA_TEMP 2

bool heaterOn = false;

void init()
{
        // инициализация
}

void controlHeater(double targetTemp)
{
        double temp = readTemp();
        if (temp < targetTemp – DELTA_TEMP)
                heaterOn = true;
        if (temp > targetTemp + DELTA_TEMP)
                heaterOn = false;
}
```

Функция init() ничего не делает, реальная система управления, конечно, устроена сложнее. Для управления температурой происходит опрос датчика температуры «double temp = readTemp();». Далее используется условие: если температура меньше, чем целевая – минус 2 градуса, то нагрев включается.

Оператор условия if имеет следующий синтаксис:

```
if (условие)
    один_оператор;
```

Если условие истинно, один следующий оператор выполняется, если ложно, то не выполняется. Операторы можно группировать в один составной оператор с помощью {}, что будет рассмотрено далее.

В файле heater.cpp не присутствует код, отвечающий за работу датчика температуры. Он находится в файле TmpSensor.cpp и пока для упрощения просто возвращает число:

```
// файл TmpSensor.cpp
double readTemp()
{
```

```
        return 36.6;
}
```

Аналогично устроены четвертый и последний файл с исходным кодом UI.cpp:

```
// файл UI.cpp
double getUserTemp()
{
        return 100.0;
}
```

Взглянем на схему подготовки исполняемой программы, которая показана на рис. 1.16.

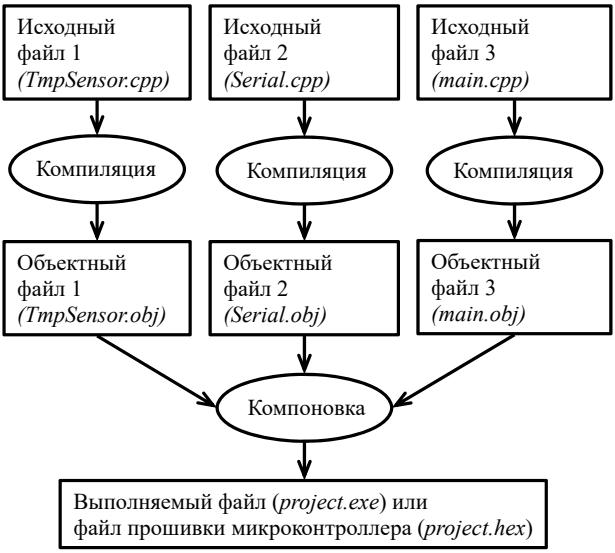

Рис. 1.16. Схема подготовки исполняемой программы

Файлы TmpSensor.cpp и UI.cpp компилятор может скомпилировать без затруднений, они независимы. Но вот при компиляции главной функции возникает проблема: компилятор ничего не знает про функцию init(), существует ли такая функция, какие у нее параметры. Ведь эта функция находится в файле heater.cpp, который, возможно, еще даже не обрабатывался и будет обрабатываться независимо от всех других единиц трансляции! Компилятор выдаст ошибку.

Чтобы избежать этого, необходимо сообщить компилятору, что да, такая функция где-то существует в проекте. У нее определенные параметры, и возвращает она значение определенного типа. Для этого используется так называемый прототип функции.

Прототипом функции в языке Си или C++ называется объявление функции, не содержащее тела функции, но указывающее имя функции, количество аргументов, типы аргументов и возвращаемый тип данных. В то время как определение функции описывает, что именно делает функция, прототип функции может восприниматься как описание ее интерфейса.

Определение функции:

```
double readTemp()
{
        return 36.6;
}
```

Прототип этой же функции:

```
double readTemp();
```

По сути, это заголовок функции без ее тела, но в конце ставится точка с запятой. Прототип функции должен быть введен в единице трансляции до первого вызова функции. А вот описание функции может располагаться где угодно. В этом случае компилятор при вызове функции включит в код «заглушку», которая будет заменена выполняемым кодом-компоновщиком.

В таком варианте файл main.cpp мог бы скомпилироваться следующим образом:

```
// файл main.cpp
void init();
void controlHeater(double targetTemp);
double getUserTemp();

int main()
{
        init(); // начальная инициализация

        while (true)
        {
                double userTemp = getUserTemp();
```

```
                controlHeater(userTemp); // управление нагревом
        }
}
```

Но здесь есть неудобство – приходится перечислять прототипы функции в каждом подобном файле с исходным кодом, где функции могут понадобиться. Хотя относятся они к модулю. Поэтому обычно прототипы выносят в заголовочные файлы с расширением .h.

```
// файл heater.h
void init();
void controlHeater(double targetTemp);
```

Теперь вместо перечисления всех требуемых функций в каждом файле с исходным кодом достаточно включить содержимое заголовочного файла соответствующего модуля:

```
#include «heater.h»
```

Директива #include буквально включает содержимое файла heater.h вместо себя. Если название файла заключено в двойные кавычки, то его поиск будет сначала осуществляться в директории проекта, затем в системных директориях компилятора. В случае использования кавычек <> файл будет искаться только в системных директориях.

Схема обработки единицы трансляции показана на рис. 1.17. Следует отметить, что заголовочные файлы реализуют интерфейс модуля. Для использования системы управления нагревателем необходимо знать о существовании функции controlHeater(), знать ее назначение, следующее из ее названия (лучший вариант), или из сопроводительной документации, или из комментариев в программе, также необходимо знать ее параметры, их типы. Но абсолютно не требуется знать ее внутреннюю реализацию, как она устроена. В том числе она может быть написана другим человеком.

При использовании заголовочных файлов необходимо добавить защиту от повторного включения, чтобы один и тот же файл не включался дважды в одну единицу трансляции. Это делается или с помощью директивы #pragma once в начале заголовочного файла:

```
// файл UI.h
#pragma once
```

```
double getUserTemp();
```

или с помощью условной компиляции:

```
// файл TmpSensor.h
#ifndef _TMP_SENSOR_H
#define _TMP_SENSOR_H
double readTemp();
#endif
```

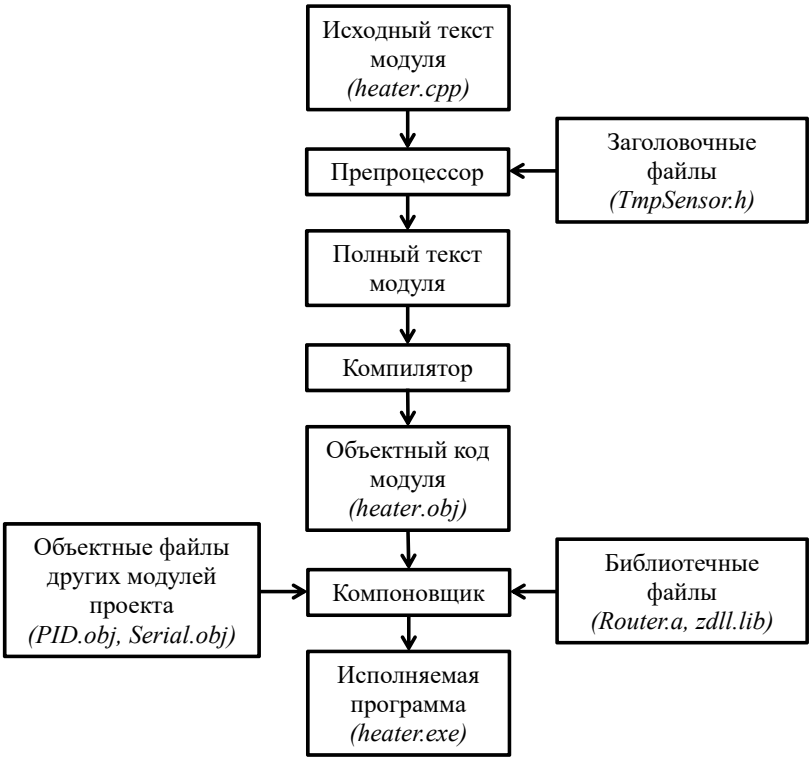

Рис. 1.17. Схема обработки единицы трансляции

Если файл включается первый раз, то символ _TMP_SENSOR_H не определен (название произвольное, обычно связано с названием файла). В этом случае текст от #ifndef (if not defined) до #endif включается. В том числе и объявляется определение «#define _TMP_SENSOR_H». При попытке повторно включить файл в исходный код он целиком будет пропущен.

Выводы

Язык Си++ является высокоуровневым компилируемым языком программирования, что означает высокую степень абстракции программ и необходимость преобразования исходного кода в машинный.

Исторически сложилось, что для создания исполняемого файла требуется минимум две программы – компилятор и компоновщик.

Программа может быть разбита на модули, код которых размещается в разных cpp-файлах, включенных в проект. Компилятор обрабатывает их независимо, поэтому в проектах используются заголовочные файлы для связи файлов с исходным кодом. Альтернативой заголовочным файлам является получение информации об объявленных типах, функциях и т. д. напрямую из уже откомпилированного модуля. Так поступают языки Паскаль, Java и другие.

Контрольные вопросы

1. Что такое инструментальное программное обеспечение?
2. Что такое компилятор, транслятор, компоновщик?
3. Что такое единица трансляции?
4. Зачем используются заголовочные файлы в Си++?
5. Что такое препроцессор? Какие директивы препроцессора вам известны?
6. Что такое описание функции? Что такое определение функции?
7. Что включается в заголовок функции?
8. Что такое главная функция?
9. Какие функции осуществляют ввод данных с клавиатуры и вывод на экран в Си?
10. Что означает %d в параметре функции printf()?
11. Что означает \n в строке Си?

Глава 2. ОСНОВЫ ЯЗЫКА СИ

2.1. Типы, переменные и константы, арифметика

Все данные в программе на языке Си имеют тип. Например, при объявлении

```
int counter;
```

указывается, что переменная с именем counter (с англ. – счетчик) имеет целочисленный тип int. Это означает, что переменная будет содержать только целые числа в некотором диапазоне (от –32768 до 32767 в программах микроконтроллеров серии ATmega, например в контроллере Arduino UNO). Тип задается один раз при объявлении переменной, и далее в процессе работы программы поменять его нельзя.

В более общем случае тип – это множество значений и операций для объекта. В свою очередь объект – некоторая область в памяти вычислительной системы, в которой хранится значение определенного типа. **Значение** представляет собой множество битов (к примеру, 00101010), которые интерпретируются программой в соответствии с типом.

Переменная является частным случаем объекта и представляет собой именованный объект. В выражении

```
double y = 5.5 + sin(x);
```

sin(x) – математическая функция синус, подпрограмма, которая возвращает некоторое значение вещественного типа double. Это значение перед выполнением сложения с числом 5.5 должно храниться в оперативной памяти, однако с ним не связана переменная. Другими словами, функция sin() возвращает объект.

Тип указывает, как интерпретировать данные, которые хранятся в памяти в виде последовательности нулей и единиц. Следующий фрагмент кода выведет на экран число 42 (в этом примере и в нескольких далее для вывода на экран для кратности будет использована библиотека ввода-вывода C++, а не функция C printf()):

```
int var1 = 42;
std::cout << var1;
```

С другой стороны такой фрагмент выведет на экран символ *:

```
char var2 = 42;
std::cout << var2;
```

В первом (или младшем) байте переменной var1 будут храниться биты 00101010 (42 в десятичной системе счисления), остальные байты будут равны 0, так как переменная занимает 2 байта и даже больше, например в программах для ОС Windows 10.

Первый и единственный байт var2 будет содержать биты 00101010, такие же, как в var1. Однако значения типа char интерпретируются подпрограммой вывода на экран (*std::cout << var2*) не как числа, а как коды символа или номера символов в таблице ASCII, а под номером 42 находится символ *.

Также тип определяет число бит, которое занимает объект в памяти.

Типы данных делят на базовые и производные. В Си достаточно много базовых типов, некоторые более распространенные из них приведены в табл. 2.1 и 2.2.

Таблица 2.1

Размер некоторых целочисленных типов данных в битах на разных платформах

Тип	Стандарт Си	ATmega, Arduino	16-битные версии Windows	32-битные версии Windows, Linux, macOS	64-битные версии Windows	64-битные версии Linux, macOS
short	Не менее 16	16	16	16	16	16
int	Не менее 16	16	16	32	32	32
long	Не менее 32	32	32	32	32	64
long long	Не менее 64	64	64	64	64	64

Каждый базовый тип соответствует аппаратным возможностям вычислительной платформы и имеет фиксированный размер в программах для нее.

Как видно, размер данных типа short на всех платформах одинаков и составляет 16 бит, или 2 байта. Размер данных других указанных типов варьируется. Обычно их размер определяет исполь-

зуемую модель данных и обозначается как, например, LP32 (long и размер указателей, англ. pointer, 32 бита) или 2/4/4 (размер типа int / размер типа long / размер указателей).

Указатели являются объектами, которые хранят номера ячеек в памяти. Соответственно, если указатель занимает 2 байта, то хранить он может числа от 0 до 65535 или, другими словами, в программе можно адресовать и использовать 64 кБ памяти.

Модель данных 2/4/2 используется в Arduino; 2/4/4 (или LP32) использовалась в Windows 3.1; 4/4/4 ILP32 – в 32-битных ОС Windows, Linux, macOS; 4/4/8 или LLP64 – в 64-битных ОС Windows; 4/8/8 или LP64 – в 64-битных ОС linux, macOS. Другие модели используются очень редко.

По умолчанию переменные указанных типов могут хранить как положительные числа, так и отрицательные:

```
short var3 = 32767;
var3 = var3 + 1; // var3 станет равной -32 768, а не +32 768
```

То есть при увеличении максимального числа из диапазона на единицу значение становится минимальным.

С каждым из указанных типов можно использовать ключевое слово unsigned в виде префикса, чтобы самый старший бит в данных интерпретировался как разряд числа, а не использовался как признак наличия знака минус в числе:

```
unsigned short var4 = 32767;
var4 = var4 + 1; // var4 станет равной +32 768
```

В объекте типа unsigned short можно хранить числа в диапазоне от 0 до 65535. В следующем примере обратите внимание на результат переполнения целого числа:

```
unsigned short var5 = 65530; // var5 равно 65530
var5 = var5 + 10; // var5 равно 4
```

Для записи целочисленных констант можно использовать префиксы и суффиксы. Префиксы означают используемую систему счисления:

```
int var1 = 0x1A; // 0x – префикс шестнадцатеричной системы счисления
int var2 = 0b11101; // 0b – префикс двоичной системы счисления
int var3 = 016; // 0 – префикс восьмеричной системы счисления
```

Обратите внимание, что 016 – это 14 в десятичной системе счисления, которая используется по умолчанию без префикса.

Суффиксы задают тип констант. Они также занимают место в памяти, как и переменные, и если суффикса нет, то компилятор сам выбирает тип констант в зависимости от значения и сколько байт под нее выделить:

от 0 до 32767 – int;

от 32768 до 2147483647 – long;

от 2147483648 до 4294967295 – unsigned long.

Если по каким-то причинам необходимо сделать константу 37 типа long, то следует использовать суффиксы L или l: 37L. Для unsigned-типов – суффиксы U или u: 37U – unsigned int, 37ul – unsigned long.

В табл. 2.2 приведены некоторые другие базовые типы данных.

Для констант вещественного float-типа следует использовать суффиксы F или f. Без суффиксов тип констант будет double.

3.3f – имеет тип float и занимает 4 байта в памяти;

3.3 – имеет тип double и занимает 8 байт в памяти;

33E-1 – double;

33e-1 – double;

Таблица 2.2

Некоторые базовые типы данных

Тип	Типичный размер в битах	Назначение переменных типа
bool	8	Используется в логических выражениях, хранит только два возможных значения: true – истина, false – ложь
char	8	Используется для хранения символов. Массив данного типа используется для хранения текста
float	32	Хранит вещественные числа в формате стандарта IEEE-754
double	64	Хранит вещественные числа двойной точности (больший диапазон значений или цифр после запятой, чем в переменных типа float)

Тип bool относится к Си++ (в Си доступен аналог в заголовочном файле stdbool.h стандарта C99), зависит от платформы и может отличаться от одного байта. А вот значения bool могут быть преобразованы к простому целочисленному типу int, и стандартом C++11 гарантируется, что true равно 1, а false равно 0.

Переменные типа char также могут хранить целые 8-битные числа. Их размер всегда 1 байт вне зависимости от платформы. Знаковость типа char по умолчанию зависит от платформы. Для ARM-микроконтроллеров или PowerPC переменные этого типа обычно только положительные и хранят числа в диапазоне от 0 до 255. Для ПК x86, x64 тип char обычно знаковый, диапазон значений этого типа от 128 до 127.

С этим типом может быть использован префикс unsigned для хранения чисел в диапазоне от 0 до 255. Очень часто тип unsigned char используется для хранения одного байта информации и манипуляции отдельными его битами, например в регистрах управления периферией микроконтроллеров. Если же тип, наоборот, необходимо сделать знаковым, то используется ключевое слово signed:

```
signed char var1 = -100;
signed int var2 = -100;
```

Размер любого типа на конкретной платформе можно всегда узнать, используя стандартную операцию sizeof:

```
std::cout << sizeof(int); // на экране появится 4 на большинстве платформ
```

Операция возвращает размер указанного типа int в байтах. Можно указывать или тип в скобках, или переменную, например:

```
double var3 = -123.45;
std::cout << sizeof(var3); // на экране появится 8
```

Как видно, с одной стороны, базовые типы данных позволяют эффективно использовать ресурсы и возможности конкретной платформы, с другой – получается некоторый зоопарк типов данных, особенно целочисленных. Часто при передаче данных с одного устройства, например от цифрового датчика, к другому, например контроллеру нагревателя, необходимо гарантировать корректную интерпретацию значений вне зависимости от платформы и размера типов данных.

Для этого в стандарте C99 вводится заголовочный файл stdint.h. В этом файле среди прочего определяются типы данных с фиксированным размером, как показано в табл. 2.3.

В Си существует ключевое слово typedef, с помощью которого можно вводить синонимы типам данным. Например, тип unsigned char используется для хранения 1 байта и имеет довольно громоздкое написание.

Таблица 2.3

Целочисленные типы данных с фиксированным размером

Тип	Описание
int8_t	Знаковое 8-битное число
int16_t	Знаковое 16-битное число
int32_t	Знаковое 32-битное число
int64_t	Знаковое 64-битное число
uint8_t	Беззнаковое 8-битное число
uint16_t	Беззнаковое 16-битное число
uint32_t	Беззнаковое 32-битное число
uint64_t	Беззнаковое 64-битное число

Можно ввести синоним byte, который не является ключевым словом языка:

```
typedef unsigned char byte;
```

Далее возможно объявлять и использовать переменные типа byte:

```
byte var1 = 10; // по сути тип var1 – unsigned char
```

Вот фрагмент файла stdint.h из Arduino (arduino-1.0.5-r2\hardware\tools\avr\avr\include):

```
typedef signed int int16_t;
```

А так выглядит это же объявление в файле Visual Studio 2019 Community (Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.23.28105\include):

```
typedef short int16_t;
```

Аналогично для хранения данных, обозначающих размер участка памяти в байтах, следует использовать тип **size_t**, так как размер

может варьироваться от нескольких килобайт до гигабайт и более. Операция sizeof возвращает значение именно этого типа. Тип **size_t** определяется аналогично типу int8_t и другим в заголовочном файле stddef.h (найдите самостоятельно).

Операция – это конструкция в Си, которая аналогична по записи математическим операциям, таким как сложение, вычитание и т. д. Операции сообщают компилятору о необходимости выполнения различных действий. В Си операции бывают не только арифметическими, но и логическими (обозначаются знаками &&, ||, !), сравнения (используются знаки ==, !=), присваивания (используются знаки =, +=, -=) и многими другими.

Операции обозначаются специальными знаками операций, иногда состоящими из одного символа (такими как +), иногда из двух (==). Реже применяются слова, например sizeof, new, delete, typeid.

Знаки операций формируют и обеспечивают вычисление выражений. **Выражение** – это правило для получения значения. Следующая запись является выражением:

```
5 + var1 – var2 * 10
```

В русском языке возникла путаница между понятиями «операция» и «оператор». Английское слово operator, которое соответствует термину «операция», иногда ошибочно переводят как «оператор». С другой стороны, statement соответствует термину «инструкция» и по историческим причинам обозначает то же самое, что и «оператор».

Оператор – это наименьшая исполняемая единица программы. Для обозначения конца оператора в Си используется символ точка с запятой (;). Исключение – составной оператор, то есть набор операторов, помещенных между фигурными скобками, открывающей ({) и закрывающей (}). В конце составного оператора точка с запятой не ставится.

Другими словами, 5 + var1 – это операция, а следующая запись является оператором:

```
var2 = 5 + var1;
```

Операция может быть частью оператора, наоборот – нет.

По сути, операция – это просто функция, которая записывается особым способом. Операция сложения в примере выше теоретически могла быть заменена вызовом функции:

var2 = sum(5, var1);

Аргументы операций называют операндами. В операции сложения – два операнда, которые записываются слева и справа от знака операции. Операции делятся по количеству операндов на унарные (один операнд), бинарные (два), тернарные (три).

Унарные операции приведены в табл. 2.4.

Таблица 2.4

Некоторые унарные операции

Знак операции	Назначение	Пример использования

–	Унарный минус изменяет знак операнда	int var1 = 5;
		int var2 = -var1;
		// var2 равна -5

+	Унарный плюс. Введен для симметрии с унарным минусом –	int var1 = +5; // var1 равна 5

&	Получение адреса операнда	int var1 = 5;
		int* p = &var1;
		/* p содержит адрес в памяти, где хранится переменная var1 */

*	Операция обращения по адресу, операция разыменовывания. Операндом должен быть адрес (указатель)	int var1 = 5;
		int* p = &var1;
		*p = 10;
		/* изменено содержимое ячейки памяти, где хранится var1, var1 теперь равна 10 */

~	Побитовое отрицание. Инвертирует все разряды внутреннего двоичного кода операнда. Инверсия осуществляется во всех байтах, если операнд занимает не один байт	uint8_t var1 = 5;
		// var1 = 5 = 00000101
		uint8_t var2 = ~var1;
		// var2 = 11111010 = 250

!	Логическое отрицание. Отрицание любого ненулевого числа будет 0, а отрицание 0 будет 1	int var1 = !2; // var1 = 0
		int var2 = !0 // var2 = 1

Окончание табл. 2.4

Знак операции	Назначение	Пример использования

++	Увеличение на 1, или инкремент. Операндом не может быть что-то, значение чего поменять нельзя (например, ++5). Операция может быть префиксной (++var1) или постфиксой (var1++). Префиксная операция увеличивает значение операнда до его использования в выражении, постфиксная – после	int var1 = 5;
		++var1; // var1 = 6
		var1++; // var1 = 7
		
		int var2 = ++var1;
		// var2 = 8, var1 = 8
		var2 = var1++;
		// var2 = 8, var1 = 9

––	Уменьшение на 1, или декремент. Аналогичная инкременту операция	int var1 = 5; --var1; // var1 = 4

sizeof	Вычисление размера в байтах объекта того типа, который имеет операнд.
	Допустимы форматы:
	sizeof(унарное_выражение)
	sizeof(тип)
		float var1 = 5;
		size_t var2 = sizeof(var1);
		// var2 = 4
		var2 = sizeof(double);
		// var2 = 8
		var2 = sizeof(5.5)
		// var2 = 8
		var2 = sizeof(5.5f)
		// var2 = 4

Бинарных операций большинство, они делятся на следующие группы:

1) аддитивные (сложение +, вычитание –);
2) мультипликативные (умножение *, деление /, получение остатка от деления %);
3) побитовых сдвигов (сдвиг влево <<, сдвиг вправо >>);
4) поразрядные (И &, ИЛИ |, исключающее ИЛИ^);
5) сравнения (меньше чем <, больше чем >, меньше или равно <=, больше или равно >=, равно ==, не равно !=);
6) логические (И &&, ИЛИ ||);
7) присваивания (=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=);
8) операции с компонентами классов Си++;
9) запятая;
10) скобки () и [].

Эти операции, а также тернарная операция «?:» будут рассмотрены в примерах в последующих разделах.

2.2. Проверки

При вычислении выражений некоторые операции требуют, чтобы операнды имели соответствующий тип, например, бинарная операция сдвига не работает с вещественным типом float. В операции сложения целого числа и вещественного результат по стандарту Си будет вещественным, и, чтобы его получить, целый операнд будет принудительно неявно преобразован в формат вещественного типа. То же самое возникает при инициализации, когда тип инициализирующего объекта приводится к типу определяемого объекта.

Преобразования типов можно разделить на две группы:

1) не меняющие внутреннее битовое представление данных;
2) меняющие внутреннее битовое представление данных.

К первой группе относится преобразование вида:

```
short var1 = 40000;
```

40000 (10011100 01000000) больше максимального положительного числа типа short 32767 (01111111 11111111), а старший 16-й бит предназначен для отрицательных чисел. Код вида:

```
short var1 = 40000;
std::cout << var1;
```

выведет на экран число –25536 (10011100 01000000). При обратном преобразовании данные сохранятся, так как обе переменные имеют идентичное значение, которое по-разному интерпретируется:

```
short var1 = 40000;
std::cout << var1 << "\n"; // -25536 на экране
unsigned short var2 = var1;
std::cout << var2; // 40000 на экране
```

При преобразовании из вещественного типа float в тип short недостаточно изменить только интерпретацию, необходимо изменить длину участка в памяти для внутреннего представления (с 4 байт до 2) и способ кодирования (отбрасывается дробная часть целиком). Даже возможен выход за допустимый диапазон значений short, например:

```
short var3 = 50000.5f;
```

Что именно произойдет в этом случае, будет зависеть от конкретной реализации языка. Поэтому с целью сохранения переноси-

мости программ следует применять преобразования типов с осторожностью.

В приведенных примерах преобразование типов было неявным. Часто компилятор делает предупреждение о таких преобразованиях при компиляции. Однако иногда преобразования типов данных необходимы. Например, требуется отбросить дробную часть числа путем преобразования float в int. В этом случае следует использовать явное преобразование типов, указав компилятору, что так и задумано. Для явного преобразования в Си используется каноническая операция приведения (cast) к требуемому типу: (int)3.14, (float)2 / 5, (int)'A'.

```
int var4 = (int)3.14; // var4 = 3
float var5 = (float)var4 / 2; // var5 = 1.5f
```

То есть преобразование осуществляется унарной операцией в формате (тип)операнд. Существуют другие форматы явного преобразования данных, их следует изучить самостоятельно.

Перед чтением переменной в программе ей необходимо присвоить какое-то значение. Это можно выполнить или с помощью оператора присваивания:

```
int var6; // значение переменной неопределенно
var6 = 5;
```

или сразу инициализировать значением (рекомендуемый вариант):

```
int var7 = 5;
```

В стандарте C++14 (не в Си) появилась универсальная форма, основанная на фигурных скобках:

```
double var8 = { 5.4 };
```

или

```
double var9 { 5.4 };
```

В случае использования фигурных скобок неявное преобразование типа будет ошибкой компиляции:

```
int var1 = 4.6; // неявное преобразование, var1 = 4
int var2 { 4 }; // все OK, var2 = 4
int var3 = { 4.6 }; // ошибка!
```

Также в C++11 в качестве типа переменной можно использовать ключевое слово auto, в этом случае тип переменной будет определен автоматически в зависимости от типа инициализирующего выражения:

```
auto var1 = 4.6; // var1 имеет тип double
auto var2 = 4.6f; // var2 имеет тип float
auto var3 = 4; // var3 имеет тип int, так как константа 4 имеет тип int
```

Си предоставляет обычный набор операторов для выражения выбора и циклов, таких как условный оператор if, переключатель switch, операторы цикла for и while.

Формат условного оператора if следующий:

```
if (выражение)
    один_оператор;
```

Если выражение в скобках истинно, то есть больше или меньше 0 (отрицательные числа тоже считаются истиной), то выполняется один следующий оператор. Если ложно, то есть равно 0, то выполнение одного следующего оператора пропускается:

```
int a = 2, b = 3;
if (a < b)
    printf(«a < b»); // печатает на экране надпись a < b всегда
```

Чтобы выполнить несколько операторов, при выполнении условия их необходимо объединить в составной оператор, используя фигурные скобки:

```
if (2 < 3)
{
        int var1 = 5;
        printf(“%d”, var1);
}
```

Дополнительно в условный оператор if можно добавить необязательную часть else, которая выполняется, если выражение выбора ложно:

```
int a = 2, b = 3;
if (a < b)
    printf(“a < b”);
else
    printf(«b < a»);
```

В части else также используется один оператор, который может быть составным. Зачастую в условие выбора необходимо добавить несколько проверок, это осуществляется с помощью логических операций И, ИЛИ:

```
float temp = 25.7f;
if (temp > 20 && temp < 30)
    printf(«Температура больше 20 и меньше 30»);
else
printf(«Температура или 20 и меньше, или 30 и больше»);
```

В выражении выбора в примере используются три бинарных операции: >, &&, <. В данном выражении сначала выполняется temp > 20, затем temp < 30, а потом только логическая операция И. Почему так, а не, например, сначала temp > 20, затем &&, где результат temp > 20 (0 или 1) как левый операнд и temp как правый, а потом результат логической операции становится левым операндом операции <? Точно так же, как в арифметических операциях, сначала выполняется умножение и деление, а потом сложение и вычитание, у всех операций Си есть приоритет, который определяет порядок вычисления выражения. Приоритеты операций Си приведены в табл. 2.5. Иногда приоритет операций называют рангом операций.

Таблица 2.5

Приоритеты операций

Приоритет	Операция	Описание	Ассоциативность
1	++ --	Суффиксные ++ и –	Слева направо →
	()	Вызов функции	
	[]	Индексация массива	
	.	Доступ к элементу структуры и объединения	
	->	Доступ к элементу структуры и объединения через указатель	
	(type){list}	C99. Составные литералы	
2	++ --	Префиксные ++ и –	Справа налево ←
	+ -	Унарные плюс и минус	
	! ~	Логическое НЕ и побитовое НЕ	
	(type)	Преобразование типа	
	*	Разыменовывание указателя	
	&	Получение адреса	

Окончание табл. 2.5

Приоритет	Операция	Описание	Ассоциативность
	sizeof	Вычисление размера	
	_Alignof	Выравнивание	
3	* / %	Умножение, деление и вычисления остатка от деления	Слева направо →
4	+ -	Сложение, вычитание	
5	<< >>	Побитовый сдвиг влево и вправо	
6	< <= > >=	Операции сравнения	
7	== !=	Операции сравнения РАВНО и НЕРАВНО	
8	&	Побитовая И	
9	^	Побитовая ИСКЛЮЧАЮЩЕЕ ИЛИ	
10	|	Побитовая ИЛИ	
11	&&	Логическая И	
12	||	Логическая ИЛИ	
13	?:	Тернарная условная операция	Справа налево ←
14	=	Простая операция присваивания	
	+= -= *= /= %= <<= >>= &= ^= |=	Операции присваивания	
15	,	Операция «запятая»	Слева направо →

При вычислении любого выражения сначала выполняются операции с наивысшим приоритетом (1 в таблице), затем с меньшим приоритетом (2 и далее). Если в выражении несколько операций с одинаковым приоритетом, то они выполняются в порядке ассоциативности или слева направо, или справа налево.

Необходимо различать логические операции И, ИЛИ, НЕ и побитовые. Последние выполняют соответствующие действия над каждым битом операндов. Например, значение выражения 6 & 3 будет равно 2:

$$6_{10} = 110_2,\ 3_{10} = 011_2,\ 110_2 \,\&\, 011_2 = 2_{10}.$$

Другими словами, первый бит в результате будет равен 1, если первый бит в первом И во втором операнде равен 1, в остальных случаях первый бит в результате будет равен 0. Второй и остальные биты в результате вычисляются аналогично.

Подобным образом вычисляется результат побитовой операции ИЛИ. Первый бит в результате будет равен 1, если первый бит и в первом ИЛИ во втором операнде равен 1. Только если первый бит в обоих операндах равен 0, соответствующий бит в результате будет равен 0. 1 | 4 = 5:

$$1_{10} = 001_2, 4_{10} = 100_2, 001_2 \mid 100_2 = 101_2 = 5_{10}.$$

Побитовая операция НЕ является унарной и просто меняет все биты операнда на противоположные: с 1 на 0, с 0 на 1. Обратите внимание, что операция НЕ применяется ко всем байтам операндов, так же как И, ИЛИ. Но именно в случае операции НЕ старшие биты устанавливаются в 1, если в операнде они были равны 0, и результат может отличаться в зависимости от длины типа:

```
uint8_t a = ~1; // a = 0b11111110 = 254
uint16_t b = ~1; // b = 0b1111111111111110 = 65534
int16_t c = ~1; // c = 0b1111111111111110 = -2
```

Использование побитовых операций получило широкое использование при программировании микроконтроллеров.

Рассмотрим пример. В процессе работы оборудования могут возникать ошибки, обычно их обозначают номерами: 1 (например, не горит лампа), 2 (не работает двигатель), 3, 4 и т. д. Нам необходимо сохранить их в переменную. Если ошибка сохраняется только одна, к примеру, последняя, ее номер можно просто сохранить в переменную целого типа:

```
int error = 0; // нет ошибки
error = 1; // не горит лампа
```

Но что делать, если может возникнуть несколько ошибок сразу и требуется сохранить их все? Можно использовать логические переменные для каждой ошибки:

```
bool error1 = false; // лампа горит
bool error2 = true; // двигатель не работает
...
bool error30 = false;
```

Конечно, можно использовать массивы:

```
bool errors[30]; // массив из 30 объектов для хранения ошибок
```

Но в любом случае для каждой ошибки используется целый байт, что не очень рационально в случае микроконтроллеров с 2 килобайтами оперативной памяти (Arduino на базе МК серии Atmega).

Можно использовать отдельные биты переменной: первый бит – ошибка 1, второй бит – ошибка 2. Если использовать переменную типа uint8_t, то можно закодировать до 8 ошибок в одном байте:

```
uint8_t errors = 0; // нет ошибок
errors = 0b00000001; // errors = 1, не горит лампа
errors = 0b00000010; // errors = 2, двигатель не работает
errors = 0b00000011; // errors = 3, не горит лампа И
// двигатель не работает
```

Теперь нам необходимо установить соответствующий бит в единицу, если возникает ошибка, в ноль – если ошибки нет. При этом остальные биты переменной изменять нельзя. Это реализуется с помощью битовой маски.

Битовая маска – определенные данные, которые используются для маскирования – выбора отдельных битов или полей из нескольких битов из двоичной строки или числа.

Пусть переменная errors равна 0b00011001, то есть обнаружены ошибки 1, 4 и 5. Мы выполнили опрос датчика и установили, что двигатель оборудования не работает. Необходимо добавить ошибку номер 2 к трем имеющимся. Это делается с помощью операции ИЛИ так:

```
errors = errors | 0b00000010;
```

или так, что полностью эквивалентно:

```
errors |= 0b00000010;
```

Второй операнд в примере, в котором только 2-й бит равен единице, а все остальные нули, является битовой маской для установки 2-го бита переменной errors в единицу. Эту битовую маску можно получить, используя операцию сдвига влево <<.

```
uint8_t errors = 0b00011001;
uint8_t newErrorCode = 2;
uint8_t bitmask = 1 << newErrorCode – 1;
```

```
// обратите внимание на приоритет << и -,
// если не уверены, ставьте (): 1 << (newErrorCode – 1)
errors |= bitmask; // 0b00011011
```

Сдвиг влево единицы ноль раз – это единица, один раз – 2, два раза – $100_2 = 4_{10}$ и т. д.

Чтобы сбросить второй бит в ноль, не затрагивая остальные биты, понадобится операция И и другая битовая маска:

```
uint8_t errors = 0b00011011;
errors = errors & 0b11111101; // errors = 0b00011001
```

В этом случае в битовой маске все биты равны единицы, кроме того, который необходимо сбросить в 0. Для сброса выполняется побитовая И. Получить такую маску, зная номер бита, можно следующим образом, также используя сдвиг единицы влево:

```
uint8_t bitmask = ~(1 << newErrorCode – 1);
```

У побитового НЕ более высокий приоритет, чем у сдвига и вычитания, нам потребуется использовать скобки для изменения порядка вычисления выражения.

Установка и сброс отдельных битов часто применяются при работе с регистрами микроконтроллера. Каждый бит регистра может что-то означать. Например, для регистра PORTB это напряжение на соответствующих выводах:

```
// установили 5 вольт на PB5,
// но убрали напряжение с PB0, PB1, PB2, PB3, PB4, PB6, PB7
PORTB = 0b00100000;
// установили 5 вольт на PB5,
// не затронув остальные выводы порта
PORTB = PORTB | 0b00100000;
// установили 0 вольт на PB5,
// не затронув остальные выводы порта
PORTB = PORTB & 0b11011111;
```

Часто для установки и сброса отдельных битов используются макросы препроцессора. Например, в AVR и в Arduino определены следующие макросы:

```
#define sbi(port, bit) (port) |= (1 << (bit))
#define cbi(port, bit) (port) &= ~(1 << (bit))
```

Также для удобства определяются номера битов, начиная с 0:

```
/* PORTB */
#define PB7 7
#define PB6 6
#define PB5 5
#define PB4 4
#define PB3 3
#define PB2 2
#define PB1 1
#define PB0 0
```

Используются в программе подобные макросы следующим образом:

```
sbi(PORTB, PB5); // установить бит PB5 в 1
cbi(PORTB, PB5); // сбросить бит PB5 в 0
```

Чтение отдельного бита осуществляется с помощью операции И & и битовой маски вида 0b00000010.

```
uint8_t errors = 0b00011011;
if (errors & 0b00000001) // проверяем первую ошибку
{
    // действия при ошибке № 1
}
if (errors & 0b00000010) // проверяем вторую ошибку
{
    // действия при ошибке № 2
}
```

Зачастую сами битовые маски, в которых только один бит установлен в единицу, а остальные равны нулю, используются как идентификаторы ошибок, параметров и т. д.

```
#define ERROR1 (1 << 0)
#define ERROR2 (1 << 1)
uint8_t errors = ERROR1; // установлена только ошибка 1
errors |= ERROR2; // установили ошибку номер 2
errors &= ~ERROR2; // сбросили ошибку номер 2

if (errors & ERROR2) // проверили ошибку номер 2
{
```

```
        // обработка ошибки № 2
}
```

Проверки используются в операторах цикла. Их несколько.

```
for (int x = 0; x < 5; ++x)
        один_оператор;
```

Итерационный цикл for включает в себя три части, разделенные точкой с запятой:

1. Инициирующее выражение (int x = 0). Выполняется только один раз при достижении программой оператора цикла for().

2. Условие продолжения (x < 5). Проверяется перед каждым выполнением тела цикла. То есть один_оператор может ни разу не выполниться, если условие после начальных действий ложно. Обычно является критерием завершения цикла.

3. Действие в конце каждой итерации цикла (++x). Выполняется каждый раз после выполнения тела цикла. Обычно используется для приращения индексов массивов.

Цикл с предусловием while:

```
int x = 0;
while (x < 5)
{
                ++x;
}
```

Выполняется, пока условие в круглых скобках () истинно. Тело цикла может не выполниться ни разу.

Цикл с постусловием:

```
int x = 6;
do
{
                ++x;
}
while (x < 5);
```

Тело цикла выполнится хотя бы один раз.

2.3. Указатели и массивы

Память в современных ЭВМ представляется последовательностью ячеек по одному байту, в которых хранятся данные. Память имеет адресацию: первый байт имеет адрес 0, второй 1 и т. д. в зависимости от объема памяти. При использовании микроконтроллеров часто приходится иметь дело с небольшими объемами оперативной памяти, поэтому следует знать, как в ней хранятся объекты, переменные, сколько свободной памяти осталось при работе программы.

Для работы с памятью в Си используются указатели. Указатель (pointer) – переменная, содержащая адрес объекта. Следующий пример демонстрирует инициализацию и использование указателя:

```
int var = 5; // целочисленная переменная равная 5
int* p; // неинициализированный указатель на объект типа int
p = &var; // получение адреса, где хранится var
std::cout << var << «\n»; // на экране появляется 5
std::cout << p << «\n»; // на экране появляется 0x7ffe98d06ffc
```

Здесь при запуске примера в 64-битной операционной системе выводимое значение указателя равно 0x7ffe98d06ffc, но это число может отличаться при повторном запуске: переменная var в другом запуске может храниться в другой ячейке памяти, однако важно, что когда переменная уже размещена в памяти, она не перемещается в процессе работы.

Знак операции & называется амперсандом (ampersand), а сама операция является унарной операцией получения адреса, операнд должен быть левосторонним выражением. Левостороннее (l-value) выражение – выражение, которое может находиться с левой стороны операции присваивания =, правостороннее (r-value) – с правой стороны. Переменная является левосторонним выражением, а, например, константа – правосторонним. Другими словами, запись int* p = &5 будет такой же некорректной операцией, как и 5 = var.

Указатель не хранит данные объекта, на который он указывает. Только адрес объекта. Адрес может быть 16-битным (до 64 кБ ОЗУ), 32-битным (до 4 Гб ОЗУ) или 64-битным, как в примере, позволяя процессору работать с разными максимальными объемами памяти.

Узнать размер указателя в случае необходимости можно с помощью операции sizeof(p).

При объявлении указателя мы выбрали тип int. Это тип объекта, на который ссылается указатель, в примере тип переменной var.

Значение ячеек памяти, на которые указывает указатель, можно изменить:

```
int var1 = 5; // целочисленная переменная равная 5
int var2 = 7; // другая целочисленная переменная равная 7
int* p = &var1; // сразу инициализируем указатель
*p = 4; // *p – объект по адресу, на который указывает p;
// var1 теперь равна 4, а не 5
p = &var2; // теперь указатель p содержит адрес var2
*p = 10; // var2 теперь равна 10
std::cout << var2; // 10 на экране
```

Операция * является унарной операцией обращения по указателю, или операцией разыменовывания указателя, ее результатом является объект, адрес которого хранит указатель. Тип объекта в данном случае определяется типом указателя.

Значение переменных поменялось не в результате операции присваивания с этими переменными, а в результате изменения данных в ячейках оперативной памяти. Простое сопоставление фундаментальных конструкций языка с реальными аппаратными средствами имеет решающее значение для высокой производительности на низком уровне. Модель машины в Си и Си++ основана на компьютерном оборудовании, а не на некой математической абстракции.

Рассмотрим хранение переменных в памяти на следующем примере:

```
long var = 0x11223344; // 287454020 в десятичной системе счисления
std::cout << var; // на экране 287454020
```

В переменной var типа long мы храним достаточно большое значение, для которого используется 4 байта, как показано на рис. 2.1.

Адрес	10	11	12	13
Содержимое ячейки памяти	0x44	0x33	0x22	0x11
Объект	var			

Рис. 2.1. Размещение переменной типа long в памяти

Короткий адрес ячейки памяти 10, где хранится переменная, приведен только для примера и обычно не является реальным. Размещаются многобайтовые переменные в памяти стандартно одинаково – сначала младший байт, затем старшие. Обратите внимание на преимущество шестнадцатеричной системы счисления над десятичной в данном случае: мы легко можем определить, какие значения будут хранить все 4 байта. Попробуем изменить только младший байт 0x44.

```
long var = 0x11223344;
long* p1 = &var; // p1 хранит адрес var
char* p2 = (char*)&var; // p2 хранит тот же адрес, что и p1
```

При попытке выполнить только char* p2 = &var; появится ошибка «cannot convert ‘long int*’ to ‘char*’ in initialization». Попытка выполнения операций присваивания или сравнения с указателями разных типов приведет не к неявному преобразованию типов, а к ошибке. Поскольку обычно это действительно алгоритмическая ошибка или опечатка в коде и компилятор следит за этим. Однако с помощью явного преобразования типа можно указать компилятору, что так задумано разработчиком и это не является ошибкой.

```
long var = 0x11223344; // 287454020
long* p1 = &var; // p1 хранит адрес var
char* p2 = (char*)&var; // p2 хранит тот же адрес, что и p1
*p2 = 0;
std::cout << var; // на экране 287453952 или 0x11223300
```

Если выполнить выражение *p1 = 0, то поменяется значение всех 4 байтов. Так как тип указателя p1 long, то все 4 байта, включая 3 байта за тем, адрес которого содержит p1, будут рассматриваться как один объект. Однако при выполнении *p2 = 0 объект будет интерпретироваться как однобайтовый тип char и в результате поменяется только самый младший байт.

Кстати, чтобы выводить числа сразу в шестнадцатеричной системе счисления с помощью std::cout, можно применить флаг преобразования потока hex:

```
std::cout << hex << var;
```

Обратно вернуть вывод в десятичной системе можно с помощью флага dec:

```
std::cout << dec << var;
```

Над указателями можно выполнять арифметические действия, такие как увеличение или уменьшение на целочисленное значение и нахождение разницы указателей.

```
uint32_t var = 0x11223344;
uint16_t* p1 = (uint16_t*)&var; // p1 содержит адрес с байтом 0x44 var
p1 = p1 + 1; // увеличиваем адрес на размер типа uint16_t (2 байта)
*p1 = 0x55; // var = 0x00553344
std::cout << hex << var << endl; // endl вместо “\n”
*p1 = 0x5566; // var = 0x55663344
std::cout << var;
```

Обратите внимание, что в результате p1 = p1 + 1 изменение адреса, который хранит p1, происходит не на один байт, как может показаться сначала, а на размер типа объекта. Если необходимо изменить только один байт, то указатель должен быть типа char*, uint8_t* и т. д.

```
long var = 0x11223344; // 287454020
char* p2 = (char*)&var;
++p2; // аналогично p2 = p2 + 1
*p2 = 0; // var = 0x11220044
std::cout << var; // на экране 287440964 или 0x11220044
```

Другим способом работы с большими участками памяти являются массивы. В листинге 2.1 показано использование одномерного массива для вычисления среднего значения нескольких элементов.

Листинг 2.1

```
#include <stdio.h>
int main()
{
        int arr[5] = { 1, 2, 3, 4, 5 }; // объявление и инициализация массива
```

```
        arr[0] = 6; // изменение первого элемента с 1 на 6
        float sum = 0; // сумма элементов
        for (int i = 0; i < 5; ++i)
        {
                printf("%d ", arr[i]);
                sum += arr[i]; // добавление к sum i-го
элемента
        }
        float average = sum / 5;
        printf("\nsum=%f, average=%f\n", sum, average);
        return 0;
}
```

Наиболее простыми являются одномерные массивы с одним индексом. Объявляются они следующим образом:

```
int arr[5]; // массив с именем arr из 5 элементов типа int
```

Если происходит сразу инициализация значений элементов, то число элементов в массиве можно пропустить, компилятор сам его вычислит:

```
float arr1[] = { 0, 1, 2 }; // 3 элемента
arr[3] = 10; // ОШИБКА, диапазон индексов от 0 до 2
float arr2[5] = { 0, 1, 2 }; // 5 элементов,
// но значения присвоены только трем первым
```

Имя массива можно использовать как указатель на его начало. Тип такого указателя соответствует типу элементов массива.

```
float arr1[] = { 0, 1, 2 };
*arr1 = 5; // arr1[0] = 5
std::cout << arr1[0] << " " << arr1[1] << " " << arr1[2]; // 5 1 2
```

Но это не совсем указатель в обычном понимании. Во-первых, его нельзя изменить:

```
arr1 = arr1 + 1; // ОШИБКА
```

Однако мы можем создать обычный указатель отдельно и менять элементы через него:

```
float arr1[] = { 0, 1, 2 };
float* p = arr1;
++p;
*p = 4; // 0, 4, 2
```

Возможно даже получить адрес отдельных элементов, используя операцию &:

```
float* p = &arr[2]; // p указывает на последний элемент arr1
```

Во-вторых, при выполнении операции sizeof(arr1) результатом будет не размер указателя, а число байт, которое занимает массив в памяти. Зная это, можно вычислить число элементов массива в местах программы, отличных от объявления массива.

```
int arr2[] = { 10, 11, 12 };
int numOfElements = sizeof(arr2) / sizeof(arr2[0]); // 3
```

В приведенном фрагменте кода размер всего массива в байтах делится на размер первого элемента (а минимум один элемент в массиве будет). Возможно даже использовать макрос для работы с массивами:

```
#define NUM_OF_ELEMENTS(x) (sizeof(x) / sizeof(x[0]))
int arr2[] = { 10, 11, 12 };
int numOfElements = NUM_OF_ELEMENTS(arr2); // 3
```

Приведенные массивы являются статическими – их размер определяется программистом на этапе создания программы. В процессе работы изменить размер массива уже нельзя. Однако часто требуется выделить и использовать массив, размер которого определяется вводом пользователя, показателями датчиков информации и т. д. Например, программа поиска людей на изображении с помощью компьютерного зрения может использовать массив. В таком массиве могут содержаться координаты областей изображения с обнаруженными людьми для последующего анализа, например для идентификации личности, однако заранее нельзя сказать, сколько людей попадут в кадр. В этом случае приходится иметь дело с динамически выделяемой памятью. В Си для этого используются стандартные функции malloc() и free(), в Си++ – операции new и delete. В листинге 2.2 показан пример программы, использующей malloc() и free().

Листинг 2.2

```
#include <stdio.h>
#include <stdlib.h> // необходимо для malloc() и free()
int main()
{
```

```
        int numOfElements;
        scanf("%d", &numOfElements); // вводится число
элементов
        int* p = malloc(numOfElements * sizeof(int));
        for (int i = 0; i < numOfElements; ++i)
        {
                p[i] = 10 * i;
                printf("%d ", p[i]);
        }
    free(p);
        return 0;
    }
```

При вводе числа 4 на экран выводится список элементов массива 0, 10, 20, 30. Как видно, операция [] используется для доступа к отдельным элементам массива.

Функция malloc() ищет свободный участок нужного размера в оперативной памяти, запоминает его и возвращает его адрес. Тип возвращаемого значения – void*. Указатель void* преобразуется к другим типам без ошибок. Функция не знает тип элементов, которые вы хотите хранить в выделяемом участке памяти, поэтому ее аргументом является не число элементов массива, а размер участка в байтах. Его придется вычислить вручную и без ошибок. Далее в программе подобно тому, как имя массива использовалось в качестве указателя, с указателем на выделенную область памяти можно использовать операцию []. Как только память перестает быть нужна, ее необходимо освободить, чтобы она могла использоваться другими подпрограммами. В случае если программа просто закрывается, то операционная система Windows, конечно, освободит выделенные участки автоматически для использования другими программами. Но в целом следует придерживаться правила: любые запрашиваемые ресурсы, такие как память, файл или внешнее устройство, после использования следует освобождать.

Если свободной памяти не хватает и выделить участок требуемого размера не получается, то функция malloc() возвращает особое значение указателя – NULL. Существуют разные реализации значения NULL (приводится в файле stddef.h, он подключается через stdlib.h), например такая:

```
#define NULL 0
```

В программе следует проверять результат работы malloc():

```
int* p = malloc(numOfElements * sizeof(int));
if (p == NULL)
{
        printf("error in malloc()");
}
```

Или так:

```
if (!p)
{
        printf("error in malloc()");
}
```

Кратко рассмотрим на примере микроконтроллера Atmega328p, используемого в контроллере Arduino UNO R3, что означают термины «свободная оперативная память» и «динамически выделяемая память».

На рис. 2.2 показана упрощенная карта оперативной памяти микроконтроллера.

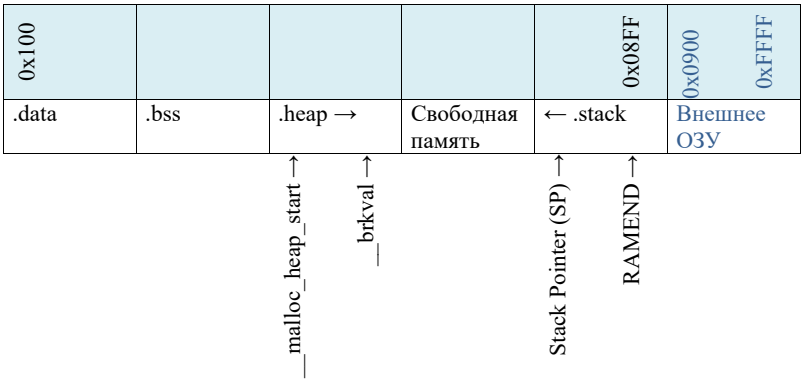

Рис. 2.2. Карта оперативной памяти микроконтроллера Atmega328p

Размер оперативной памяти Atmega328p составляет 2 килобайта, или 2048 байт. Каждая ячейка имеет номер.

Обратите внимание, что память с данными программы начинается с адреса 256 (0x100). Первые 256 байт имеют системное назна-

чение, в частности используются для ввода-вывода через память. Например, запись определенных значений в определенные ячейки оперативной памяти будет приводить к установке определенных уровней напряжения на выводах микроконтроллера.

Память данных программ начитается с адреса 0x100. Всю память делят на так называемые разделы, хранящие данные определенного типа.

.data – этот раздел содержит статические данные, которые были определены в вашем коде. Код, подобный следующему, будет помещен в раздел .data:

char str[] = «Hello, world!»;

То есть коды символов ‘H’, ‘e’, ‘l’ и т. д. будут храниться здесь.

.bss (block starting symbol, название сложилось исторически) – в этом разделе находятся неинициализированные глобальные или статические переменные.

В комментариях к следующему фрагменту программы для Arduino указаны разделы, где хранятся переменные:

```
int a; // .bss
int b = 5; // .data
void setup()
{
    int c = 5; // STACK
    c++;
}
```

Локальные переменные в программе хранятся в разделе стека (STACK). Причем если размер .data и .bss фиксирован и определен на этапе создания прошивки, то размер стека меняется в процессе выполнения программы.

Когда в программе встречается новая локальная переменная, она помещается в стек, но когда выполнение программы выходит за область видимости данной переменной (за }), она исключается из стека и ее место может занимать следующая локальная переменная.

Раздел «куча» (HEAP) используется для работы функции malloc(). Именно в нем выделяются участки для хранения данных произвольного размера в процессе работы. С работой кучи связано понятие «фрагментация кучи».

Фрагментация кучи – это состояние, в котором доступная память разбивается на небольшие не связанные друг с другом блоки. Если куча фрагментирована, выделение памяти может завершиться сбоем, даже если общего объема доступной памяти в куче достаточно для выполнения запроса, так как ни один блок памяти не обладает нужной величиной.

Размер кучи может увеличиваться в процессе работы программы. Что произойдет, если разделы «куча» и «стек» пересекутся? В этом случае работа микроконтроллера далее становится непредсказуемой. Оценить объем свободной памяти для Arduino можно, используя программу, показанную в листинге 2.3.

Листинг 2.3

```
extern char* __brkval;

int freeMemory()
{
    char top;
    return __brkval ? &top – __brkval : &top – __malloc_heap_
start;
}

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    Serial.println(freeMemory());
    delay(2000);
}
```

Для работы используются системные указатели __brkval и __malloc_heap_start, применяемые для работы функции malloc(). Вершину стека можно вычислить, введя новую локальную переменную «char top;» и получив ее адрес с помощью операции &. Конкретное распределение памяти других микропроцессоров отличается, однако понятия «стек» и «куча» используются и для других вычислительных систем.

Выводы

Все данные в Си имеют тип, определяющий множество возможных значений и допустимых операций над ними. В частности, к базовым типам относятся целочисленный, вещественный, символьный и логический типы.

Значения вычисляются с помощью выражений, включающих различные операции. Порядок выполнения операций в выражении определяется рангом и ассоциативностью.

Записывая значения в определенные ячейки памяти, можно управлять периферийными устройствами, подключенными к микроконтроллеру.

Доступ к ячейкам памяти осуществляется с помощью указателей.

Контрольные вопросы

1. Что такое тип данных?
2. Что такое объект?
3. Чем переменная отличается от объекта?
4. Что такое значение?
5. Что такое выражение? Чем операция отличается от оператора?
6. Каков порядок выполнения операций в выражении?
7. Что такое указатель?
8. Что такое стек и куча с точки зрения оперативной памяти?

Глава 3. СИ++

3.1. Пользовательские типы и классы Си++

В отличие от своего предшественника Си, являющегося процедурным языком программирования, язык Си++ относится к объектно-ориентированным языкам.

Чтобы понять, что такое объектно-ориентированное программирование, рассмотрим небольшой пример. Предположим, вам надо написать программу для работы с базой данных книг или отчетов. Для упрощения примера остановимся на задаче представления информации о книге в оперативной памяти и вывода ее на экран. Каждая запись о книге содержит название и год издания. Программа показана в листинге 3.1.

Листинг 3.1

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

char title[50];
int year;

void InputBook()
{
    cout << "Enter title:\n";
    cin.getline(title, sizeof(title));
    cout << "Enter year:\n";
    cin >> year;
}

void PrintBook()
{
    cout << "Title: " << title;
    cout << ", year: " << year << "\n";
}

int main()
{
    InputBook();
    PrintBook();
    return 0;
}
```

Результат работы программы показан на рис. 3.1.

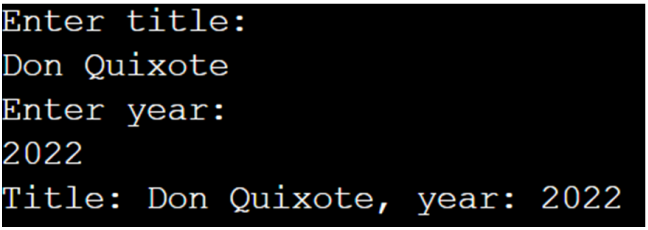

Рис. 3.1. Ввод и вывод информации о книге

Пока пропустим рассмотрение строчки «*cin.getline(title, sizeof (title));*», она работает подобно знакомой вам строчке «*cin >> title;*», но сохраняет в переменную title всю введенную строку, а не только символы до первого пробела, а также ограничивает размер ввода 50 байтами, отведенными под эту переменную.

Проблема программы заключается в том, что она позволяет хранить только информацию об одной книге. Для хранения двух и более книг нам понадобятся или новые переменные:

```
char title1[50], title2[50], title3[50];
int year1, year2, year3;
```

или использование массивов:

```
char title[50][10];
int year[10];
```

В первом случае программа просто нерасширяема и работает только для трех книг, во втором случае мы все равно работаем с отдельными независимыми переменными и связь между ними ясна только из имен и только для небольших программ. И если в нашу программу необходимо будет к книгам добавить больше переменных, например сведения о журналах, отчетах, логи доступа к базе данных, то быстро понять, что относится к книгам, будет сложно:

```
char title[50][10];
int year[10];
int logIDs[30];
int date, userID;
```

Поэтому еще на заре развития вычислительной техники в программах стали использоваться структуры.

Структура – это объединенное в единое целое множество переменных в общем случае разных типов.

В листинге 3.2 приведена модифицированная программа, в которой осуществляется работа с информацией о двух книгах.

Листинг 3.2

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

struct Book
{
    char title[50];
    int year;
};

Book InputBook()
{
    Book newBook;
    cout << "Enter title:\n";
    cin.getline(newBook.title, sizeof(newBook.title));
    cout << "Enter year:\n";
    cin >> newBook.year;
    cin.ignore();
    return newBook;
}

void PrintBook(Book book)
{
    cout << "Title: " << book.title;
    cout << ", year: " << book.year << "\n";
}

int main()
{
    Book book1 = InputBook();
    Book book2 = InputBook();
    PrintBook(book1);
    PrintBook(book2);
    return 0;
}
```

В первую очередь мы объединили переменные title и year в новый структурный тип данных с именем Book следующим образом:

```
struct Book
{
    char title[50];
    int year;
};
```

Обратите внимание на обязательную точку с запятой ; в конце такого определения после закрывающей фигурной скобки } в отличие от функций или составных операторов.

Далее модифицировали функцию InputBook():

```
Book InputBook()
{
    Book newBook;
    cout << “Enter title:\n”;
    cin.getline(newBook.title, sizeof(newBook.title));
...
return newBook;
}
```

Теперь функция возвращает значение нового пользовательского типа Book. В самой функции объявляется новая переменная newBook типа Book и заполняются ее элементы title и year, которые обычно называют полями структуры. Доступ к полю структуры осуществляется с помощью точки:

имя_структурной_переменной.имя_элемента_структуры

Очень важно различать шаблон для создания объектов или тип данных Book и объекты этого типа (анонимные и именованные, или переменные). То есть такая запись будет бессмысленной и является синтаксической ошибкой: Book.year!

«cin.ignore();» удаляет невидимый символ перехода на новую строку, который остается во входном потоке после «cin >> newBook.year;», когда пользователь нажимает на Enter после ввода года. Без этой команды при повторном вызове InputBook() вместо title будет сразу считана пустая строка, как если бы пользователь нажал Enter.

Также модифицировали функцию PrintBook(). Теперь у функции есть аргумент – книга, информацию о которой необходимо

распечатать. Следует отметить, что в функцию передается полная копия объекта, то есть копируется более 50 байт информации при каждом вызове, что на практике иногда неэффективно, и часто передаются указатели и ссылки на объекты пользовательских типов, а не сами объекты, этот подход будет рассмотрен далее.

После проделанной работы по созданию нового типа данных главная функция программы выглядит достаточно просто:

```
int main()
{
    Book book1 = InputBook();
    Book book2 = InputBook();
    PrintBook(book1);
    PrintBook(book2);
    return 0;
}
```

Мы считываем с клавиатуры информацию о первой книге, сохраняем ее в переменную book1, затем о второй и так же последовательно выводим введенные пользователем данные на экран.

Таким образом, в языке Си++ можно выделить базовые типы данных, например:

```
int X;
```

или

```
int X, Y, Z;
```

где int – имя базового типа данных, а X, Y, Z – объекты типа int.

А также можно выделить производные типы данных, сконструированные из объектов других типов:

```
struct Person
{
    char name[50];
    int age;
};
Person student1, student2;
Person students[10];
```

где Person – производный тип данных, структура, а student1, student2, students – объекты.

Стоит отметить, что массивы также являются производным типом данных, объединяющим однородные объекты. Также в состав структуры или другого производного типа данных (далее мы рассмотрим еще классы) могут входить объекты других производных типов, а не только базовых.

Несмотря на то что мы упростили работу с переменными в создаваемой базе данных книг, объединив их в структуру, функции для работы с данными по-прежнему отделены от них, а об их связи с книгами говорят лишь имена InputBook() и ReadBook(). По сути, функции действуют на данные, и они не связаны жестко между собой. В небольших программах это обычно не вызывает проблем, но по мере увеличения объема кода программы его необходимо как-то упорядочивать, для его поддержки необходима определенная архитектура. Стали применяться различные подходы к оформлению кода программ. И в конечном счете появились парадигмы программирования, среди которых выделилась объектно-ориентированная парадигма.

Возникла идея внести функции для работы с данными структуры в саму структуру. Модифицированная в третий раз программа работы с книгами показана в листинге 3.3.

Листинг 3.3

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

struct Book
{
    char title[50];
    int year;

    void Input()
    {
            cout << “Enter title:\n”;
            cin.getline(title, sizeof(title));
            cout << “Enter year:\n”;
            cin >> year;
            cin.ignore();
    }

    void Print()
    {
```

```
            cout << "Title: " << title;
            cout << ", year: " << year << "\n";
        }
};
int main()
{
    Book book1, book2;
    book1.Input();
    book2.Input();
    book1.Print();
    book2.Print();
    return 0;
}
```

Во-первых, код функций внесен внутрь структуры. В главной функции main() теперь нет кода, независимого от объектов. В функции main() объявляются две переменные – book1 и book2. Далее объекту book1 говорится «считай себя», то же самое сообщается объекту book2. В завершение объекту book1 говорится «напечатай себя» и аналогично – book2. Теперь программа представляет собой не просто множество функций с кодом, а множество объектов с действиями над ними, взаимодействующих между собой.

Синтаксис вызова функции структурного объекта аналогичен синтаксису доступа к полям: используется операция «точка». В функции Input() доступ к собственным элементам осуществляется без точки. Следует понимать, что year в операторе «cin >> year;» в функции Input() для объекта book1 – это book1.year, а для book2. Input() – это book2.year. Другими словами, Book.Input() также не имеет смысла и является синтаксической ошибкой.

Чтобы показать отличие функции, принадлежащей какому-то объекту или структуре, от традиционных функций Си, используется специальный термин – «метод». Другими словами, Input() является методом структуры Book и объектов book1, book2.

В объектно-ориентированном программировании используются следующие четыре принципа:

1) инкапсуляция;
2) наследование;

3) полиморфизм;
4) абстракция.

Инкапсуляция (или пакетирование) – соединение в одном объекте данных и функций, которые манипулируют этими данными. Собственно, мы только что частично рассмотрели данный принцип на примере программы из листинга 3.3.

Сокрытие представляет собой принцип проектирования, заключающийся в разграничении доступа различных частей программы к внутренним компонентам друг друга. В языке Си++ (но не во всех языках ООП, например, в Python есть инкапсуляция, но нет сокрытия) принцип тесно пересекается, вплоть до отождествления, с инкапсуляцией. Другими словами, инкапсуляция – это принцип, согласно которому любая часть системы должна рассматриваться как «черный ящик»: пользователь подсистемы должен видеть только интерфейс (то есть список декларируемых переменных и методов) и не вникать во внутреннюю реализацию.

Реализуется в Си++ такое разграничение доступа с помощью трех спецификаторов доступа – **public**, **protected** и **private**. Рассмотрим фрагмент программа для Arduino для управления скоростью вращения мотора (листинг 3.4).

Листинг 3.4

```
class Motor
{
public:
    void init(int pin);
    void start();
    void stop();

protected:
    int _myPin;

public:
    int _speed;
private:
    void calculateSpeed();
protected:
};

void Motor::start()
{
```

```
    calculateSpeed(); // OK
}
Motor myMotor;
void setup()
{
    myMotor.init(2); // OK
    myMotor._speed = 0; // OK
    // ERROR!
    myMotor._myPin = 13;
    // ERROR!
    myMotor.calculateSpeed();
}
```

Вместо struct чаще в Си++ используется ключевое слово class. Синтаксис классов аналогичен синтаксису структур. Отличие состоит лишь в спецификаторе доступа к элементам по умолчанию. Основные термины для классов аналогичны рассмотренным ранее терминам для структур.

Класс – множество объектов, имеющих общую структуру и общее поведение, шаблон для создания объектов, тип данных.

Метод класса в объектно-ориентированном программировании – это функция или процедура, принадлежащая какому-либо классу или объекту.

Объект – сущность в адресном пространстве вычислительной системы, которая появляется при создании экземпляра класса и обладает определенным состоянием и поведением.

Переменная имеет состояние, поведение и может быть однозначно идентифицирована (имеет уникальное имя).

Объявляется класс Motor с четырьмя методами – init(), start(), stop(), calculateSpeed() и двумя переменными целочисленного типа – int _myPin и _speed.

После спецификатора доступа «public:» все, что ниже до следующего спецификатора (или до конца определения класса), является общедоступной секцией, и ко всем элементам в ней можно получить доступ извне. Другими словами, такие обращения в функции setup() являются полностью корректными:

```
void setup()
{
    myMotor.init(2); // OK
    myMotor._speed = 0; // OK
...
```

Так как и метод init(), и переменная _speed являются открытыми, или общедоступными.

А вот такое действие приведен к ошибке компиляции:

```
// ERROR!
myMotor._myPin = 13;
```

Переменная _myPin находится в защищенной (protected) секции класса. Это означает, что доступ к ней могут иметь только методы данного класса (и производные от него, которые рассмотрены в теме «Наследование»), например init():

```
void init(int pin)
{
    _myPin = pin;
}
```

Но извне доступ получить не удастся.

В случае использования спецификатора private доступ к элементам можно получить только из методов данного класса (из производных также нельзя).

Обратите внимание, что одинаковых секций может быть несколько, они могут размещаться в произвольном порядке, могут быть вообще пустые. Это имеет значение исключительно для восприятия кода.

В объявлении класса код методов init(), start(), stop(), calculateSpeed() не приводится, хотя мог бы, как в листинге 3.3 ранее, но приводятся только прототипы (declaration, или объявление, декларация), заканчивающиеся точкой с запятой. Сама реализация (definition, или определение) методов с выполняемым кодом вынесена за объявление класса ниже:

```
void Motor::start()
{
    calculateSpeed(); // OK
}
```

В этом случае в названии функции используется «Motor::», чтобы показать, что она относится к классу Motor (в общем случае это имя_класса::имя_метода, используется ::). Такой подход позволяет выполнить разделение класса на интерфейс и реализацию.

Зная интерфейс (объявление класса Motor), можно его использовать, не вдаваясь в подробности работы внутренних функций. То есть инициализировать работу с мотором, вызвав метод init(), указав номер пина, к которому подключено устройство управления мотором, а затем запускать вращение вызовом start() и останавливать вызовом stop().

На практике объявление класса обычно находится в заголовочном файле, например motor.h, а его реализация (код методов) – в файле исходного кода motor.cpp.

Зачем необходимо ограничивать доступ с помощью protected и private? Предположим, что вы инициализировали управление мотором на выводе номер 2: «myMotor.init(2);». Затем запустили его, метод start() установил бы требуемый уровень напряжения на выводе _myPin. И если бы _myPin была бы открытой, вы могли бы ее изменить. В этом случае метод stop() попытался бы остановить мотор, используя новое значение _myPin. Но состояние вывода, к которому реально подключен мотор, не изменилось бы. Это нарушает принцип инкапсуляции, так как происходит вмешательство во внутреннюю работу объекта. Если мотор вращается, а программист вызывает функцию init() повторно, это еще можно как-то скорректировать, например, остановить вращение, но прямое изменение переменных не позволит этого сделать, что в крупных программных проектах может привести к многочисленным ошибкам.

Обратите внимание, что в листинге 3.4 не приводится код методов init(), stop(), calculateSpeed(). Поэтому это не полностью рабочий код, а только пример. А следующие вызовы являются ошибками компилятора:

```
// ERROR!
myMotor._myPin = 13;

// ERROR!
myMotor.calculateSpeed();
```

Следует отметить, что обычно код пишут так, что буквально все переменные классов являются закрытыми, а доступ к ним осуществляется только через методы класса (например, GetSpeed(), SetSpeed() и т. п.). Также в именовании переменных класса используются различные префиксы или суффиксы для удобства восприятия, чтобы в коде методов легко можно было различать имена локальных переменных, аргументов и переменных самого класса. Обычно используются префиксы m_pin (m – member, элемент класса), _pin, mPin или суффикс pin_, но необязательно.

Рассмотрим спецификатор доступа к элементам класса по умолчанию:

```
class Motor
{
    void someFunc();
    int _someVar;

public:
    void init(int pin);
    void start();
    void stop();
};
```

Для метода someFunc() и переменной _someVar не указан никакой спецификатор доступа, в этом случае используется спецификатор доступа по умолчанию.

Спецификатором по умолчанию для классов (class) является **private**!

Спецификатором по умолчанию для структур (struct) является **public**!

В этом отличие. В структурах C++ также можно использовать спецификаторы доступа public, protected, private. Но для обеспечения принципа инкапсуляции, чтобы случайно не оставить что-то общедоступным, обычно используются классы.

Абстракция данных означает предоставление только важной информации внешнему миру и скрытие фоновых данных, то есть представление необходимой информации в программе без предоставления деталей.

Программа на C++, где вы реализуете класс с общедоступными и частными членами, является примером абстракции данных.

В примере из листинга 3.4 для использования класса Motor пользователю нужно знать, что его можно инициализировать с помощью функции init(), передав номер вывода, запустить мотор и остановить. В процессе работы используется недоступная ему переменная _speed, о назначении которой ему знать не надо, но которая нужна для работы класса.

Наследование – механизм создания нового класса на основе существующего, при этом к классу могут быть добавлены новые элементы (данные, функции), а существующие функции изменены.

Полиморфизм – использование одних и тех же функций для решения разных задач.

Наследование и полиморфизм будут рассмотрены в других темах.

3.2. Конструкторы и деструкторы. Ссылки

В информатике неинициализированная переменная – это переменная, которая была объявлена в коде, но которой не было присвоено конкретное значение перед ее использованием. Какие-то значения такие переменные всегда имеют, однако эти значения непредсказуемы. Использование неинициализированных переменных – ошибка в программе, приводящая к багам. Пример приведен в листинге 3.5.

Листинг 3.5

```
#include <stdio.h>
void count(void)
{
    int k, i;
    for (i = 0; i < 10; i++)
    {
            k = k + 1;
    }
    printf("%d", k);
}
```

```
int main()
{
    count();
    return 0;
}
```

Значение, которое выводится на экран, неопределенно. Предполагая, что в начале цикла for значение k равно 0, ожидается, что на экране появится число 10. Но это не так. Результат работы показан на рис. 3.2.

При повторных запусках программы результат может отличаться. Некоторые компиляторы (Visual C++ 2019) просто выдадут ошибку наподобие «C4700 – использована неинициализированная локальная переменная».

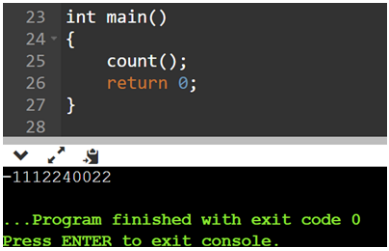

Рис. 3.2. Использование неинициализированной переменной

В случае использования структур или классов при объявлении новой переменной необходимо присвоить начальное значение уже каждому полю, которых может быть много:

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

class Motor
{
public:
    int m_pin;
    float m_speed;
};

int main()
{
```

```
    Motor myMotor;
    cout << myMotor.m_pin; // ОШИБКА!
    return 0;
}
```

Поскольку переменные класса обычно закрыты, а не являются общедоступными, как в примере выше, логичным решением проблемы инициализации является использование функции, например, так:

```
class Motor
{
public:
    void Init();
    void Start()
    {
    // запускаем вращение, подавая сигнал на вывод с номером
m_pin
    }
protected:
    int m_pin;
    float m_speed;
};

void Motor::Init()
{
    m_pin = 4;
    m_speed = 0.0f;
}

int main()
{
    Motor myMotor;
    myMotor.Init(); // здесь объект myMotor инициализирован
    myMotor.Start(); // сейчас с мотором можно работать
    return 0;
}
```

Поскольку проблема инициализации объектов является типовой, то для инициализации переменных класса не создают обыч-

ный метод вроде Init(), так как можно забыть его имя или случайно забыть вызвать после объявления переменной. Для инициализации используется специальная функция, которая называется конструктором.

Конструктор класса – специальная функция, которая автоматически вызывается при создании объекта класса. Имя этой функции по правилам языка Си++ совпадает с именем класса. Никаких значений конструктор не возвращает никогда, поэтому тип возвращаемого значения не указывается (не используется даже void!). Пример выше можно переписать так:

```
class Motor
{
public:
    Motor(); // конструктор
    void Start()
    {
        // запускаем вращение
    }
protected:
    int m_pin;
    float m_speed;
};
Motor::Motor()
{
    m_pin = 4;
    m_speed = 0.0f;
}
int main()
{
    Motor myMotor; // здесь объект myMotor инициализирован
    myMotor.Start(); // сейчас с мотором можно работать
    return 0;
}
```

В строчке «Motor myMotor;» не только объявляется новая переменная myMotor, но и сразу же инициализируется путем вызова функции-конструктора автоматически.

Перед более подробным рассмотрением вариантов использования конструкторов следует сразу разобрать понятие **деструктора**. Как вы знаете, локальные переменные существуют только внутри фигурных скобок, в которых были объявлены, или внутри своей области видимости (scope). Как только программа выходит за закрывающую скобку, все переменные, объявленные в промежутке от соответствующей ей открывающей скобки, разрушаются. В случае глобальных и статических переменных создание и разрушение происходит при запуске и завершении программы. При разрушении объекта класса происходит автоматический вызов специальной функции, называемой деструктором. Пример, показывающий порядок вызова конструкторов и деструкторов, показан в листинге 3.6.

Листинг 3.6

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

class Motor
{
public:
    Motor(); // конструктор
    ~Motor(); // деструктор
};

Motor::Motor()
{
    cout << «вызван КОНСТРУКТОР...\n»;
}

Motor::~Motor()
{
    cout << «вызван ДЕСТРУКТОР...\n»;
}

Motor globalVar;

int main()
{
```

```
    setlocale(LC_ALL, “Russian”);
    cout << «создаем объект a\n»;
    Motor a;
    if (5 > 2)
    {
            cout << “создаем объекты b и c\n”;
            Motor b, c;
            cout << «выходим за область видимости b и c\n»;
    }
    cout << «возвращаемся из функции main()\n»;
    return 0;
}
```

В примере класс Motor очень простой, включает только конструктор, который выводит надпись на экран о том, что он вызван. И аналогичный деструктор. Имя функции-деструктора также совпадает с именем класса, но вначале ставится знак тильда ~. Обратите внимание на синтаксическую конструкцию «Motor::~Motor()», в начале изучения языка легко ошибиться и написать что-то вроде «~Motor::Motor()». Сначала вводится имя класса, к которому относится функция, затем ::, в конце – имя функции.

На рис. 3.3 приведен результат работы программы в Visual Studio 2019 Community Edition.

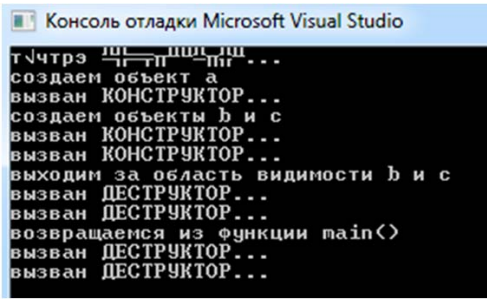

Рис. 3.3. Порядок вызова конструкторов и деструкторов

Первой строчкой идет странная нечитаемая надпись. Это сообщение о том, что вызван конструктор до установки кириллицы в консоли с помощью setlocale() в функции main(). Этот код выполняется до кода в функции main().

Следует заметить, что в целом конструктор подобен обычной функции, в которой можно открывать файлы на диске, выводить информацию на экран и другие устройства, а не только присваивать значения переменным класса. Этот факт используется при проектировании объектно-ориентированных программ.

Далее вызывается конструктор переменной a, затем выполнение программы доходит до условного оператора, внутри которого объявляются две локальные переменные – b и c. Затем происходит выход из области видимости b и c, и вызываются их деструкторы. В завершение сначала разрушается переменная a при выходе из функции main(), и потом – globalVar при завершении программы.

Применяется деструктор, чтобы освободить ресурсы, задействованные объектом в течение своего существования, например, закрыть файл на диске, открытый в конструкторе или в каком-то методе класса, освободить динамически выделенную память под данные и т. д. Пример:

```
#include <stdlib.h>

class Device
{
protected:
    void* _data; // указатель на участок памяти
    int _size; // размер участка памяти в байтах

public:
    Device();
    ~Device();
};

Device::Device()
{
    // Здесь «открываем» файл

    // ...

    // Размер _size может меняться при запуске,
    // например, это может быть вычисленный размер файла,
    // а не константа 50.
    _size = 50;
```

```
        // Выделяем динамически участок памяти
        // под содержимое файла.
        _data = malloc(_size);

        // Копируем содержимое файла и закрываем его

        // ...
}

Device::~Device()
{
        free(_data); // освобождаем использованную память
}
```

Деструктор не имеет каких-либо аргументов и может быть в классе только один. В отличие от конструкторов.

Рассмотрим следующий пример:

```
class Motor
{
public:
        Motor()
        {
                m_Pin = 0;
        }

        Motor(int pin);
protected:
        int m_Pin;
};

Motor::Motor(int pin)
{
        m_Pin = pin;
}
```

В классе два конструктора: один без аргументов, второй имеет один аргумент целого типа int с именем pin. Пока мы использовали только первый вариант без аргументов. Конструктор без аргументов называется конструктором по умолчанию. Следует отметить, что в ранних языках программирования для упрощения процесса трансляции в машинный код существовало ограничение, согласно

которому в программе не могло существовать несколько функций с одним именем. Позднее вводится перегрузка функций – теперь в программах может существовать несколько функций с одним именем, но с разным количеством аргументов или другими их типами, а функции различаются не по имени. Например, так:

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

void func()
{
    cout << “func()\n”;
}

void func(int a)
{
    cout << “func(int a)\n”;
}

int main()
{
    func();
    func(777);
}
```

В результате работы такой программы на экране сначала появится «func()», затем «func(int a)», как если бы функции, вызываемые в main(), имели разные имена. Какая именно из двух одноименных функций будет выполнена, определяется компилятором по аргументам, которые указаны при вызове. Поскольку конструкторы класса являются функциями, все сказанное относится и к ним.

При создании объекта вызывается только один конструктор. Какой именно конструктор будет использован, зависит от способа создания объекта:

```
Motor motor1; // вызывается конструктор по умолчанию Motor()
Motor motor2(13); // используется конструктор Motor(int pin)
Motor motors[4]; // используется конструктор по умолчанию Motor()
```

Конструкторы с несколькими параметрами вызываются аналогично. Далее приведен пример использования библиотеки LiquidCrystal Arduino:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 10, 5, 4, 3, 2);

void setup()
{
    lcd.begin(16, 1);
    lcd.print("hello, world!");
}

void loop() {}
```

Как видно, для работы с дисплеем создается объект lcd, в конструктор которого передаются номера использованных для подключения выводов платы.

Если аргумент конструктора только один, его можно вызвать следующим способом:

```
Motor motor3 = 12; // аналогично Motor motor3(12);
```

Здесь следует заметить разницу между знаком = при инициализации объекта и операцией присваивания.

```
Motor motor3 = 12; // вызывается конструктор, это инициализация
motor3 = 10; // вызывается операция присваивания, сейчас это ошибка
```

Еще пример:

```
Motor motor4; // вызван конструктор по умолчанию
// объект motor4 инициализирован
motor4 = 13; // ОШИБКА
// операция присваивания введена для данного класса
```

Конструктор по умолчанию может отсутствовать:

```
class Motor
{
public:
    Motor(int pin);
protected:
```

```
    int m_Pin;
};

Motor::Motor(int pin)
{
    m_Pin = pin;
}

Motor motor1; // ОШИБКА!
Motor motor2(13); // Все ОК
Motor motors[4]; // ОШИБКА!
```

Ошибка в Visual Studio 2019 будет звучать примерно так: «error C2512 : 'Motor' : no appropriate default constructor available». В таком случае создать объект такого класса, не передав какие-то параметры в конструктор, просто не получится.

Отметим, что если в классе (или структуре) не указано ни одного конструктора, то компилятор сам создает конструктор по умолчанию с пустым телом. Такой конструктор называется неявным.

Мы рассмотрели два типа конструкторов: по умолчанию и с параметрами. Еще одним типом конструкторов являются конструкторы копирования. Рассмотрим следующий пример:

```
class Motor
{
public:
    Motor()
    {
        m_Pin = 0;
    }

    Motor(int pin)
    {
        m_Pin = pin;
    }

protected:
    int m_Pin;
};

Motor motor1, motor2(2);
```

```
void setup()
{
    Motor motor3 = motor2; // неявный конструктор копирования
    // OK?!
    Motor motor4 = 4;
}
```

Теперь нам интересна строчка «Motor motor3 = motor2;», в которой один объект инициализируется существующим объектом этого же типа. В этом случае ошибки не будет – вызывается конструктор копирования. Однако в самом классе Motor данного конструктора не приведено. Он создается компилятором и является неявным. В отличие от неявного конструктора по умолчанию, он не пустой, а выполняет действия по копированию, а именно побитово копирует все значения всех внутренних переменных motor2 (m_Pin в данном конкретном примере) в значения аналогичных переменных motor3.

Иногда этого достаточно, иногда нет. В следующем примере в классе Device содержится указатель на область памяти, динамически выделяемой в процессе работы.

```
#include <stdlib.h>
#include <string.h>

class Device
{
protected:
    void* data;
    int size;

public:
    Device()
    {
        size = 50;
        data = malloc(size);
    }

    Device(const Device& src)
    {
        data = malloc(src.size);
```

```
            memcpy(data, src.data, src.size);
        }
        ~Device()
        {
            free(data);
        }
};
int main()
{
    Device device1;
    // конструктор копирования
    Device device2 = device1;
}
```

Конструктор «Device(const Device& src)» является явным конструктором копирования. Пока не следует обращать внимания на тип аргумента «const Device&». Конструктор по умолчанию динамически выделяет 50 байт информации, например, под содержимое файла на диске. Деструктор освобождает эти 50 байт в оперативной памяти. На выделенный для работы участок памяти указывает указатель data. Теперь, если бы был использован неявный конструктор копирования, то значение data, то есть номер ячейки памяти с 50 байтами информации, device2 совпадало бы со значением data device1. Оба объекта, по сути, использовали бы один и тот же участок памяти. Если один из объектов будет разрушен, участок памяти будет освобожден, работа с ним из второго объекта приведет к ошибкам.

В нашем же конструкторе копирования мы выделяем новый участок памяти: «data = malloc(src.size);». Размер данного участка равен src.size, то есть размеру поля size существующего объекта src (source, исходник). Обратите внимание, что переменная size является защищенной, но конструктор к ней может получить доступ, так как он относится к тому же классу, что и src. Далее копируем всю информацию с исходного участка памяти в новый, используя функцию memcpy() из стандартной библиотеки Си: «memcpy(data, src.data, src.size);». Неявный конструктор копирования так бы не сделал.

Вернемся к типу аргумента конструктора копирования «const Device&».

До этого момента мы успели рассмотреть два основных типа переменных:

1) обычные переменные, которые хранят значения напрямую;
2) указатели, которые хранят адрес другого объекта (или NULL), для доступа к которому выполняется операция разыменования указателя.

Ссылки – это третий базовый тип переменных в языке C++, который работает как псевдоним другого объекта или значения.

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

Пример работы со ссылками приведен в листинге 3.7.

Листинг 3.7

```
int main()
{
    int value = 7;
    int& ref = value;
    value = 8; // value = 8
    ref = 9; // value = 9
    ++ref; // value = 10
    return 0;
}
```

В программе объявляется переменная value целого типа, равная 7. Создается ссылка на нее. Далее меняется значение переменной через ее имя на 8 и через ссылку на нее сначала на 9, а затем в следующей строчке увеличивается до 10. В результате выполнения программы финальное значение value будет равно 10.

Зачем нужны ссылки?

Ссылка – это тот же указатель, который неявно разыменовывается при доступе к значению, на которое он указывает. Ссылки реализованы с помощью указателей и, по сути, являются неявными указателями.

Упрощенно можно сказать, что это также номер ячейки памяти, в которой хранится переменная, на которую указывает ссылка.

Однако для доступа к содержимому ячеек памяти не требуется использовать операцию разыменовывания *. Также ссылка в отличие от указателя обязана быть инициализирована значением. Поэтому ссылки безопаснее указателей, так как их значения всегда корректны, но ссылки ограничены в функциональности (например, при динамическом выделении памяти).

Рассмотрим подробнее, почему аргументом конструктора копирования Device является ссылка на объект «Device(const Device& src)», а не сам объект, например «Device(Device src)». Вы знаете, что при передаче переменных или констант в любую функцию Си или Си++ создаются их копии:

```
void func(int a)
{
    // a – копия x
    a = a + 1; // a = 6
}
int main()
{
    int x = 5;
    func(x);
    // x по-прежнему равен 5
}
```

Для переменных базового типа копирование очевидно, но для создания копии объекта класса нужен конструктор копирования. Который мы создаем и которого пока нет. Поэтому вместо копии исходного объекта в конструктор копирования передается адрес ячейки памяти, где содержится исходный объект.

Применение ссылок не ограничивается конструктором копирования, а используется в любых методах, когда нет необходимости создавать копии объектов. Поэтому рассмотрим синтаксис работы со ссылками немного подробнее.

Типы ссылок бывают следующими:

1) ссылки на неконстантные значения (просто «ссылки» или «неконстантные ссылки»);
2) ссылки на константные значения («константные ссылки»);
3) ссылки r-value, появившиеся в стандарте C++11.

Инициализация ссылок осуществляется так:

```
int value = 123;
int& ref = value; // ссылка инициализирована переменной value
int& invalidRef; // ОШИБКА: ссылка должна указывать на что-либо
```

Примеры использования:

```
int a = 333;
int& ref1 = a; // ок: a – это неконстантное l-value
const int b = 222;
int& ref2 = b; // не ок: b – это константное l-value
int& ref3 = 4; // не ок: 4 – это r-value
```

В процессе работы нельзя изменить, куда указывает ссылка:

```
int value1 = 3;
int value2 = 5;
int& ref = value1;
ref = value2;
```

В последней строчке примера «ref = value2;» ссылка ref по-прежнему указывает на переменную value1 и переменной value1 присваивается значение value2!

Ссылки могут применяться как синонимы полей структуры или класса:

```
struct Sensor
{
    int value1;
    float value2;
    int ADCval;
};
struct Device
{
    Sensor sensor1, sensor2;
    int otherValue;
};
Device myDevice;
```

Тогда в коде функций доступ к элементам myDevice можно получить следующими способами:

```
myDevice.sensor1.value1 = 5;
myDevice.sensor1.value2 = 3;

Sensor& sensor = myDevice.sensor1;
sensor.value1 = 5;
sensor.value2 = 3;

int& val = myDevice.sensor1.value1;
val = 5;
val = 1;
```

Однако чаще всего ссылки используются в качестве аргументов функций:

```
int a = 10;
incr(a);
```

Если функция incr() описана так, то значение переменной a не изменится:

```
void incr(int x)
{
    ++x;
}
```

Изменить переменную a можно, используя такой вариант функции:

```
void incr(int& x)
{
    ++x;
}
```

Некорректный вариант вызова:

```
incr(10); // ОШИБКА! Нельзя изменить константу 10
```

Следует отметить, что подобный функционал функции incr() можно реализовать и с помощью указателей, но код будет более громоздкий, а значения указателя могут быть некорректными:

```
int a = 10;
incr(&a);

void incr(int* ptr)
{
```

```
        if (ptr == NULL)
                return;
        ++* ptr;
}
```

Чтобы иметь возможность передавать константные объекты через ссылку, используют константные ссылки:

```
#include <stdio.h>

const int value = 123;
const int& ref = value;

int a = 7;
const int& ref1 = a;
const int b = 9;
const int& ref2 = b;
const int& ref3 = 5;

void func(const int& a)
{
        printf("%d\n", a);
}
int main()
{
        int x = 3;
        func(x);
        const int y = 4;
        func(y);
        func(5);
        func(3 + y);
        return 0;
}
```

Приведенный пример не содержит синтаксических ошибок и выводит на экран значения 3, 4, 5, 7.

Существуют и другие типы конструкторов. Рассмотрение r-value-ссылок и связанных с ними конструкторов перемещения выходит за рамки данного курса.

3.3. Наследование

Наследование (inheritance) представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса, или базового класса (base class), в другом – производном классе (derived class).

В языке Си++ производный класс объявляется следующим образом:

```
class Имя : [private | protected | public] базовый_класс
{
    тело класса
};
```

В общем случае наследование может осуществляться так:

```
class A { ... };
class B { ... };
class C { ... };
class D : A, protected B, public C
{
    ...
};
```

где класс D наследует функциональность классов A, B, C. Это называется множественным наследованием. Однако зачастую на практике производный класс наследует функциональность одного базового класса.

Например, производный класс TempSensor для работы с датчиком температуры может быть объявлен следующим образом:

```
class Sensor
{
protected:
    int m_pin;
};

class TempSensor : public Sensor
{
protected:
    float m_temp;
};
```

где Sensor – базовый класс, содержащий номер вывода для подключения датчика любого типа, а класс TempSensor содержит функционал для работы с датчиком температуры. Класс TempSensor содержит специфичную для данного типа датчиков информацию о температуре и не содержит общую для любых датчиков информацию о выводе подключения, но наследует ее как переменную m_pin от базового. Всем возможным методам класса TempSensor становится доступна переменная m_pin.

Чтобы понять, что такое наследование классов, рассмотрим другой пример, показанный в листинге 3.8.

Листинг 3.8

```
class Device
{
public:
    void Init(int pin);
protected:
    int _pin;
};
class Sensor : public Device
{
public:
    int readADCvalue();
};
class TempSensor : public Sensor
{
protected:
    float _temperature;
};
class NTCresistor : public TempSensor
{
public:
    void calculateTemp();
};
class TMP36 : public TempSensor
{
public:
    void calculateTemp();
};
```

Базовым для всех остальных классов является класс Device (устройство), обеспечивающий функционал некоторого электронного устройства. В классе имеется переменная, хранящая номер вывода микроконтроллера или платы, к которому устройство подключается. И имеется метод инициализации, сохраняющий переданный номер вывода.

Далее класс Sensor (датчик) включает функционал работы с датчиками информации. В примере это метод readADCvalue(), выполняющий чтение данных с аналогового входа. Номер входа в этом случае может храниться в переменной _pin, которая теперь доступна и методам класса Sensor в результате наследования.

Затем класс Sensor наследует класс TempSensor (датчик температуры) подобно тому, как было рассмотрено ранее в примере.

Потом в нашей иерархии классов встречается некоторая развилка: класс TempSensor наследует NTCresistor (терморезистор с отрицательным температурным коэффициентом, термистор) и независимо TempSensor наследует TMP36 (конкретная модель интегрального датчика температуры). Оба эти класса включают метод calculateTemp(), с помощью которого происходит вычисление температуры, но по разным формулам. Причем в работе метода calculateTemp() может быть использован метод readADCvalue() базового класса Sensor. В этом случае в функционал класса для работы с термистором нет необходимости включать функционал для измерения напряжения на входе, достаточно использовать формулу перевода напряжения в температуру, специфичную для конкретного типа датчиков.

Таким образом, в объектно-ориентированном программировании выделяют следующие цели наследования:

1) исключение из программы повторяющегося кода;
2) упрощение модификации и развития программы;
3) упрощение создания новых программ на основе существующих;
4) единственная возможность изменять объекты, исходный код которых недоступен.

В рассмотренных примерах мы использовали наиболее распространенное на практике открытое наследование, или public наследование, то есть объявляли производный класс как «class Sensor :

public Device». В этом случае все общедоступные (public) элементы класса Device остаются общедоступными (public) и в Sensor, все защищенные (protected) остаются защищенными и в Sensor. А вот все приватные (private) элементы (и методы, и переменные) базового класса полностью недоступны в производном классе! Существуют и другие типы наследования, а именно protected-наследование и private-наследование. Например, при объявлении «class Sensor : protected Device» общедоступный метод Init() из примера станет защищенным при обращении к объекту класса Sensor. Правила доступа к элементам в производных классах при наследовании показаны в табл. 3.1.

Таблица 3.1

Правила наследования

Ключ доступа	Спецификатор в базовом классе	Доступ в производном классе
private	private protected public	нет private private
protected	private protected public	нет protected protected
public	**private** **protected** **public**	**нет** **protected** **public**

Поскольку классы могут содержать объекты других классов, а при инициализации таких объектов вызываются функции-конструкторы, которые могут быть сложными, то необходимо понимать порядок вызова конструкторов, в том числе при наследовании.

Правила, определяющие порядок вызова конструкторов, следующие:

1. Производный класс должен иметь свои конструкторы.
2. Если в конструкторе потомка нет явного вызова конструктора предка, автоматически вызывается конструктор предка **по умолчанию**.
3. Для иерархии, состоящей из нескольких уровней, конструкторы предков вызываются, начиная с **конструктора базового класса**.

После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

4. В случае нескольких предков их конструкторы вызываются в порядке объявления.

Чтобы лучше понять вышесказанное, рассмотрим пример, показанный в листинге 3.9.

Листинг 3.9

```
#include <conio.h>
#include <iostream>

class A
{
public:
    A() { std::cout << "A constructor\n"; }
};

class B
{
public:
    B() { std::cout << "B constructor\n"; }
};

class C
{
public:
    C() { std::cout << "C constructor\n"; }
    C(int a) { std::cout << "C(int a) constructor\n"; }
};
class D : public C
{
public:
    D() { std::cout << "D constructor\n"; }
    D(int a) : C(a) { std::cout << "D(int a) constructor\n"; }

protected:
    A myA;
    B myB;
    A otherA;
};

int main()
{
```

```
    D myD;
    _getch();
    return 0;
}
```

В программе имеются классы A, B, C, независимые друг от друга, и класс D, наследующий класс C. Также в классе D содержатся переменные типа A и B. В A и B есть только конструкторы по умолчанию. Все, что они делают, это выводят сообщение в консоль, что они вызваны. В C и D есть аналогичные конструкторы по умолчанию, но к каждому из них добавлено по второму конструктору с параметром. Конструкторы с параметром также только выводят сообщение в консоль о том, что они вызваны, но другого вида.

В главной функции main() объявляется одна-единственная переменная myD типа D. И запускается цепочка вызова конструкторов для конструирования объекта myD.

Результат работы показан на рис. 3.4.

Рис. 3.4. Результат работы программы из листинга 3.9

Рис. 3.4 наглядно показывает, что при инициализации myD сначала вызывается конструктор по умолчанию его базового класса C. Затем вызываются конструкторы для переменных класса D:

```
protected:
    A myA;
    B myB;
    A otherA;
```

Вызов осуществляется в порядке их объявления. В конце выполняется тело самого конструктора класса D. После этого объект myD сконструирован и готов к работе.

Заменим «D myD;» в главной функции на «D myD(2);», то есть используем конструктор с параметром.

Результат работы программы в этом случае показан на рис. 3.5.

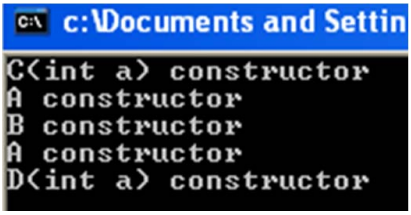

Рис. 3.5. Порядок вызова конструкторов
при выполнении «D myD(2);»

Основное отличие заключается в том, что вместо конструктора по умолчанию класса C вызывается конструктор с параметром. Но почему?

Все дело в списке инициализации конструктора класса D:

D(int a) **: C(a)** { std::cout << “D(int a) constructor\n”; }

С помощью списка инициализации до выполнения тела конструктора определяется, какие конструкторы будут использованы для инициализации базовых классов и всех собственных переменных. Ведь объекты myA, myB и другие класса D в теории могут не иметь конструкторов по умолчанию. Переменные базовых типов также можно инициализировать в списке инициализации. Пример использования списка инициализации показан в листинге 3.10.

Листинг 3.10

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

class Device
{
public:
    Device(int pin)
        : _pin(pin) // _pin = pin
    {
    }
protected:
    int _pin;
};

class Serial
{
```

```
public:
    Serial()
    {
        _speed = 9600;
    }
    Serial(int speed)
    {
        _speed = speed;
    }
    int GetSpeed()
    {
        return _speed;
    }
protected:
    int _speed;
};
class Sensor : public Device
{
public:
    Sensor(int pin, int speed, int value)
        : Device(pin), _serial(speed), _value(value)
    {
        cout << "_pin = " << _pin << "\n";
        cout << "_serial.GetSpeed() = " << _serial.GetSpeed()
<< "\n";
        cout << "_value = " << _value << "\n";
    }
protected:
    Serial _serial;
    int _value;
};
int main()
{
    Sensor mySensor(10, 115200, 42);
    return 0;
}
```

Обратите внимание, что список инициализации начинается после заголовка функции-конструктора в его определении (не в прототипе!) и продолжается до открывающей фигурной скобки.

При этом отдельные параметры могут располагаться на разных строках, форматирование текста может варьироваться.

Результат работы программы из листинга 3.10 показан на рис. 3.6.

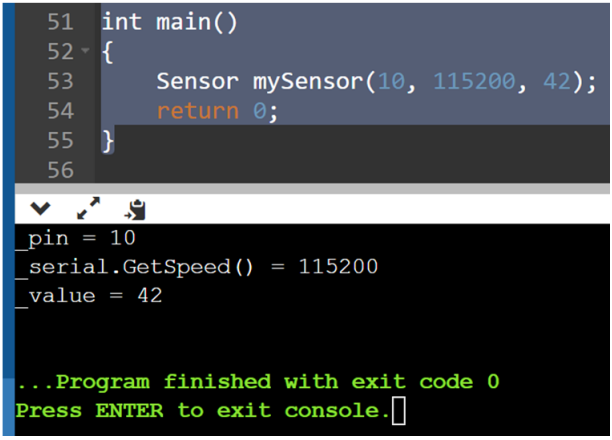

Рис. 3.6. Работа со списками инициализации

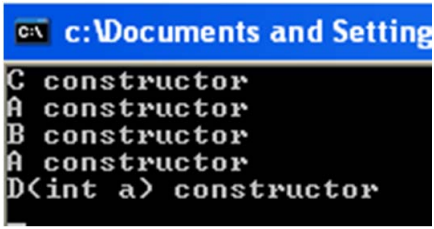

Рис. 3.7. Порядок вызова конструкторов при выполнении D myD(2); после удаления вызова конструктора C из списка инициализации конструктора D

Теперь, если в программе из листинга 3.9 заменить «D(int a) : C(a) { std::cout << «D(int a) constructor\n»; }» на «D(int a) { std::cout << «D(int a) constructor\n»; }», то есть убрать вызов конструктора класса C из списка инициализации конструктора класса D, то результат будет как на рис. 3.7. То есть согласно пункту 2 рассмотренного порядка вызова конструктора будет вызван конструктор по умолчанию. Если он есть и если конструктора по умолчанию не будет, то компилятор выдаст соответствующую ошибку. В любом

случае следует запомнить, что на момент выполнения тела конструктора выполнены конструкторы всех базовых классов и инициализированы все внутренние объекты других классов.

Порядок вызова деструкторов следующий:

1. Деструкторы не наследуются.
2. Если деструктор в производном классе не описан, он формируется **автоматически** и вызывает деструкторы всех базовых классов.
3. Вызывать в деструкторе производного класса явно деструкторы базового класса не надо. **Это будет сделано автоматически**.
4. Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются **строго обратно вызову конструкторов**: сначала деструктор производного класса, затем – элементов класса, затем – базового класса.

В случае деструкторов порядок вызова проще. У класса не может быть два деструктора, и у них нет параметров. Поэтому вызываются они автоматически в порядке, противоположном конструкторам: сначала деструктор производного класса, затем деструкторы всех его компонентов, затем деструктор базового класса. Вышесказанное демонстрирует программа в листинге 3.11, подобная программе из листинга 3.9 (показывающая порядок вызова конструкторов).

Листинг 3.11

```
#include <conio.h>
#include <iostream>

class A
{
public:
    ~A() { std::cout << "A destructor\n"; }
};

class B
{
public:
    ~B() { std::cout << "B destructor\n"; }
};
```

```
class C
{
public:
    ~C() { std::cout << “C destructor\n”; }
};

class D : public C
{
public:
    ~D() { std::cout << “D destructor\n”; }
protected:
    A myA;
    B myB;
    A otherA;
    A moreA;
};
int main()
{
    {
        D myD;
    }
    _getch();
    return 0;
}
```

Здесь использован составной оператор, чтобы разрушить переменную myD до вызова _getch():

```
{
    D myD;
}
```

Иначе сначала будет вызвана функция _getch() для формирования паузы в программе, а деструкторы и вывод информации на экран будут осуществлены при выходе из главной функции. Результат работы показан на рис. 3.8.

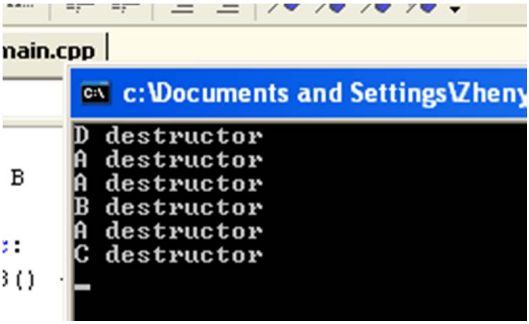

Рис. 3.8. Порядок вызова деструкторов

3.4. Абстрактные классы, виртуальные деструкторы

Полиморфизм – это свойство программного кода изменять свое поведение в зависимости от ситуации, возникающей при выполнении программы. В контексте реализации C++ полиморфизм – это технология вызова виртуальных функций, реализуемых в иерархически связанных классах. Иерархия классов формируется на базе механизма наследования, рассмотренного ранее.

Рассмотрим несколько примеров для демонстрации так называемого раннего связывания. В листинге 3.12 показаны два класса – Device (устройство) и Sensor (датчик). Оба класса имеют метод Run() (выполнить). Sensor является производным от Device.

Листинг 3.12

```
#include <conio.h>
#include <iostream>

class Device
{
public:
    void Run() { std::cout << “Device::Run()\n”; }
};

class Sensor : public Device
{
public:
    void Run() { std::cout << “Sensor::Run()\n”; }
};
```

```
int main()
{
    Device device;
    Sensor sensor;

    device.Run();
    sensor.Run();

    _getch();
    return 0;
}
```

В главной функции создаются два объекта – device и sensor соответствующих классов. И для них вызываются методы Run(). Результат работы показан на рис. 3.9.

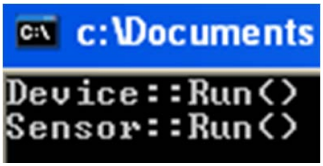

Рис. 3.9. Вывод программы из листинга 3.12

Как видно из рисунка, оба раза вызывается только одна функция Run() именно того класса, для объекта которого она была вызвана.

Теперь несколько изменим главную функцию:

```
int main()
{
    Device* pointer;

    pointer = new Sensor;
    pointer->Run();
    delete pointer;

    _getch();
    return 0;
}
```

Теперь вместо самого объекта мы объявляем указатель на объект типа Device. Указатель – это только номер ячейки памяти, где хранятся данные, и пока у нас нет ни данных, ни выделенной памяти.

Далее используется новая операция, появившаяся в Си++, – new. Она динамически в процессе выполнения программы выде-

ляет место под переменную типа Sensor в куче (heap). С кучей мы сталкивались ранее при рассмотрении функции языка Си malloc().

Операция new Си++ похожа на стандартную функцию malloc() Си за исключением очень важного нюанса. Операция new не только выделяет память в куче под все переменные класса, как это сделала бы и malloc(), но и вызывает конструктор класса для инициализации объекта. Очевидно, что функция malloc() языка Си «не знает» про классы и для выделения памяти под объекты классов не подходит.

Также new является операцией и для ее использования нет необходимости подключать библиотеки. Наконец, в отличие от malloc(), операция new никогда не вернет некорректного значения: если памяти не хватает, то будет сгенерировано исключение. В стандартном случае программа просто завершится. В данном курсе исключения С++ не рассматриваются, но проверять результат операции new не надо.

В примере в операции new вызывается конструктор по умолчанию. Если необходимо вызывать конструктор с параметрами, например 12 и 9600, вызов будет выглядеть так:

```
pointer = new Sensor(12, 9600);
```

Когда объект больше не нужен, его необходимо освободить, используя операцию delete:

```
delete pointer;
```

delete также отличается от фунции free() языка Си вызовом деструктора.

В примере динамически создается объект типа Sensor, а его адрес сохраняется в указатель на объект базового типа. Это допустимо, явного преобразования типа не требуется.

Как видно из рис. 3.10, вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта.

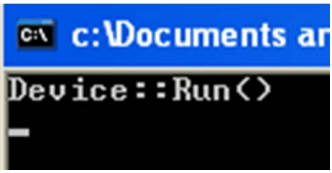

Рис. 3.10. Вызов Run() через указатель для класса из листинга 3.12

Можно использовать явное преобразование типа указателя:

```
((Sensor*)pointer)->Run();
```

В этом случае будет вызвана функция Run() класса Sensor. Но в любом случае вызываемая функция определяется на этапе написания кода.

Функцию Run() базового класса Device можно вызвать из производного Sensor, используя «::»:

```
class Sensor : public Device
{
public:
    void Run()
    {
        std::cout << “Sensor::Run()\n”;
        Device::Run();
    }
};
```

Теперь немного изменим класс, приведенный в листинге 3.12, добавив всего одно ключевое слово virtual в функцию Run(). В остальном код идентичный (листинг 3.13).

Листинг 3.13

```
#include <conio.h>
#include <iostream>

class Device
{
public:
    virtual void Run() { std::cout << “Device::Run()\n”; }
};

class Sensor : public Device
{
public:
    void Run() { std::cout << “Sensor::Run()\n”; }
};
int main()
{
    Device* pointer;

    pointer = new Sensor;
    pointer->Run();
```

```
        delete pointer;
        _getch();
        return 0;
}
```

Результат работы программы показан на рис. 3.11.

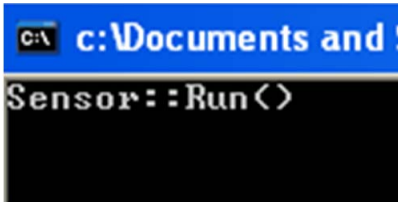

Рис. 3.11. Вызов Run() через указатель для класса из листинга 3.13

Теперь мы можем хранить все устройства в одном массиве, используя указатели на базовые классы, в цикле обрабатывать их, вызывая функцию Run(), однако на этапе выполнения будет вызываться код методов, соответствующих реальному объекту. При использовании виртуальных методов следует учитывать, что:

1) если в предке метод определен как виртуальный, то метод потомка с таким же именем и аргументами автоматически становится виртуальным, а с другими аргументами – обычным;
2) виртуальные методы наследуются, переопределять их нужно, если требуется изменить функционал.

В листинге 3.14 приведен пример использования виртуального метода.

Листинг 3.14

```
#include <conio.h>
#include <iostream>

class Device
{
public:
        virtual void Run() { std::cout << “Device::Run()\n”; }
};

class Sensor : public Device
{
```

```
public:
    void Run() { std::cout << "Sensor::Run()\n"; }
};
int main()
{
    const int config[] = { 0, 1, 1, 0, 1 };
    Device* devices[5];
    for (int i = 0; i < 5; ++i)
    {
        if (config[i] == 0)
            devices[i] = new Device;
        else
            devices[i] = new Sensor;
    }
    for (int i = 0; i < 5; ++i)
    {
        devices[i]->Run();
    }
    for (int i = 0; i < 5; ++i)
    {
        delete devices[i];
    }
    _getch();
    return 0;
}
```

Результат работы программы показан на рис. 3.12.

Рис. 3.12. Результат работы программы, управляемой данными

Главная функция программы состоит из трех частей: инициализация, главный цикл, освобождение объектов. В цикле инициализации создаются объекты в зависимости от конфигурации. Для упрощения примера использован массив с целыми числами, где 0 – базовое устройство, 1 – датчик. Однако в общем случае эти данные могут быть получены с файла на диске, то есть конкретная конфигурация программной системы определяется не на этапе написания кода, а на этапе выполнения программы.

Во втором цикле вызываются виртуальные методы единым для всех устройств способом. А в третьем цикле объекты освобождаются.

С помощью виртуальных методов реализуется программирование, управляемое данными:

1. Программирование, управляемое данными (data-driven prog-ramming) представляет собой метод или даже парадигму программирования, при котором программный код хотя и отделен от входных данных, но спроектирован таким образом, что логика программы определяется входными данными.

2. В программе, управляемой данными, часть или даже все ее свойства устанавливаются во время выполнения, что особенно важно, если программа составляется пользователем или должна им изменяться без перекомпиляции.

Следует отметить, что при таком подходе широко используются подобные конструкции:

```
class Device
{
public:
    virtual void loop();
};
void Device::loop()
{
    // ничего не делать
}
class Motor : public Device
{
```

```
public:
    void loop();
};
void Motor::loop()
{
    // какие-то реальные действия
}
```

То есть в базовом классе устройства Device метод loop() пустой, а реальный код присутствует только в производном классе Motor. Класс Device необходим только как базовый с «пустыми» виртуальными методами в иерархии устройств. По своей сути является интерфейсом.

Интерфейс описывает поведение или возможности класса C++, не связываясь с конкретной реализацией этого класса. Интерфейсы в C++ реализуются с использованием абстрактных классов.

В рассмотренном случае метод loop() базового класса Device можно записать так:

```
class Device
{
public:
    virtual void loop() = 0;
};
```

В этом случае метод loop() является чисто виртуальным методом. Использование чисто виртуальных методов имеет следующие особенности:

1. Вместо тела в прототипе записывается «= 0;». Реализация не приводится вообще.
2. Метод должен переопределяться в производном классе.
3. Если хотя бы один метод класса является чисто виртуальным, то весь класс называется **абстрактным**.
4. **Нельзя создавать объекты абстрактного класса**. Он не содержит кода методов.
5. **Можно создавать указатели и ссылки на абстрактный класс**, если не требуется создавать временный объект. Очевидно, что хранить такие указатели могут объекты производных классов.

6. Если производный класс не переопределяет все чисто виртуальные функции, то он тоже является абстрактным.

Деструкторы классов также могут быть невиртуальными и виртуальными, пока мы сталкивались только с невиртуальными. Если модифицировать классы из листинга 3.14, добавив в них деструкторы, то при разрушении объектов через указатель на них с помощью delete будут вызваны деструкторы базового класса:

```
class Device
{
public:
    ~Device() { std::cout << "~Device()\n"; }
    virtual void Run() { std::cout << "Device::Run()\n"; }
};
class Sensor : public Device
{
public:
    ~Sensor() { std::cout << "~Sensor()\n"; }
    void Run() { std::cout << "Sensor::Run()\n"; }
};
```

Результат показан на рис. 3.13.

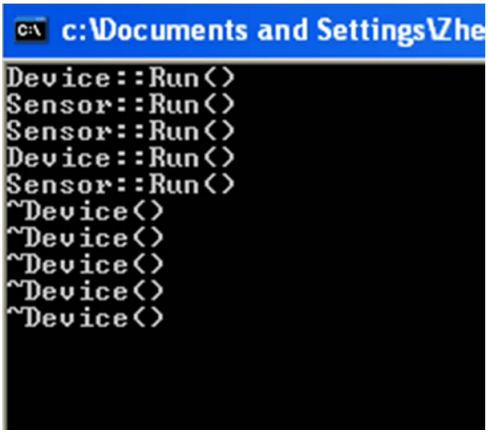

Рис. 3.13. Вызов конструкторов и невиртуальных деструкторов

Это может привести к ошибкам, так как ресурсы объектов класса Sensor не освобождаются. Однако, изменив деструкторы на виртуальные, мы получим корректную работу операции delete.

```
class Device
{
public:
    virtual ~Device() { std::cout << "~Device()\n"; }
    virtual void Run() { std::cout << "Device::Run()\n"; }
};

class Sensor : public Device
{
public:
    ~Sensor() { std::cout << "~Sensor()\n"; }
    void Run() { std::cout << "Sensor::Run()\n"; }
};
```

Результат работы программы в этом случае показан на рис. 3.14.

Рис. 3.14. Вызов конструкторов и виртуальных деструкторов

В этом случае сначала вызывается деструктор производного класса Sensor, затем базового Device.

Выводы

Структура – это объединенное в единое целое множество переменных в общем случае разных типов. Структуры позволяют создавать, в частности, переменные пользовательских типов. В Си++ структура может включать не только данные, но и функции, предназначенные для работы с этими данными.

В Си++ используются почти аналогичные структурам типы данных – классы, а функции классов называются методами.

Классы включают специальные методы – конструкторы, вызываемые автоматически при создании экземпляра класса, и деструктор, вызываемый также автоматически при разрушении объекта.

Классы могут наследоваться. Наследование – один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса в другом – производном классе.

Контрольные вопросы

1. Что такое класс?
2. Что такое метод класса?
3. Чем отличается структура от класса?
4. Какие бывают спецификаторы доступа?
5. Что такое конструктор? Сколько в классе может быть конструкторов?
6. Что такое деструктор? Сколько в классе может быть деструкторов?
7. Что такое производный класс?
8. Чем виртуальный деструктор отличается от невиртуального?
9. Что такое ссылка? Какие типы ссылок вы знаете?
10. Что такое абстрактный класс?

Глава 4. СТАНДАРТНАЯ БИБЛИОТЕКА СИ++

4.1. Шаблоны классов. Работа с контейнерами на примере std::vector<T>

Предположим, требуется написать программу, которая вычисляет среднее значение N-го количества чисел. Будем использовать одномерный массив элементов типа double. Программа показана в листинге 4.1.

Листинг 4.1

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

double mean(const double* values, size_t length)
{
	double sum = 0;
	for (size_t i = 0; i < length; ++i)
	{
		sum += values[i];
	}
	return sum / length;
}
int main()
{
	double arr[] = { 1, 2, 3, 4, 5 };
	cout << mean(arr, sizeof(arr) / sizeof(arr[0]));

	return 0;
}
```

Переменная sum (сумма) в функции mean (среднее) инициализируется нулем, далее в цикле к переменной sum прибавляются значения всех элементов массива values. Число таких элементов – length. Тип переменной values (значения) – const double*, то есть является указателем на первый элемент массива.

Ключевое слово const в типе values является модификатором. Его использование с обычными переменными превращает их в константы. При использовании модификатора const в типе указателя в начале нельзя изменить объект, на который указывает этот указатель, сам указатель изменять можно (не путать с double* const,

в этом случае именно сам указатель нельзя изменить, что на практике встречается нечасто). Другими словами, в функции mean() нельзя изменить элементы массива. Если модификатор убрать, то конкретно в данном примере ничего не изменится, однако const улучшает восприятие кода, так как пользователь функции знает, что данные, которые он передает ей, не будут изменены.

Предположим, что теперь нужно предусмотреть вычисления среднего значения для других числовых типов, например float или long. Использовать приведенную в листинге 4.1 функцию mean() для long не получится – неявные преобразования типов указателей запрещены. При попытке выполнить следующий код в функции main():

```
long arr[] = { 1, 2, 3, 4, 5 };
cout << mean(arr, sizeof(arr) / sizeof(arr[0]));
```

– появится ошибка компиляции вида: error: cannot convert ‘long int*’ to ‘const double*’.

Можно использовать перегрузку функции:

```
long mean(const long* values, size_t length)
{
    long sum = 0;
    for (size_t i = 0; i < length; ++i)
    {
        sum += values[i];
    }
    return sum / length;
}
```

И аналогично для float. Но при этом требуется произвести много действий по копированию и вставке ради очень небольших изменений: меняется тип возвращаемого значения, тип переменной sum и тип указателя values. И этот подход не масштабируется при добавлении новых типов.

Для решения этой проблемы копирования и вставки понадобится так называемое обобщенное программирование. Такой стиль, при котором код программируется с типами, которые еще не определены.

Обобщенное программирование (англ. *generic programming*) – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание.

В языке Си++ обобщенное программирование реализуется с помощью шаблонов.

Модифицированная программа, вычисляющая среднее значение массива с помощью шаблонной функции mean(), показана в листинге 4.2.

Листинг 4.2

```
#include <iostream>
using namespace std;
#define NUM_OF_ELEMENTS(x) (sizeof(x) / sizeof(x[0]))

template<typename T> T mean(const T* values, size_t length)
{
	T sum = 0;
	for (size_t i = 0; i < length; ++i)
	{
		sum += values[i];
	}
	return sum / length;
}

int main()
{
	long longArr[] = { 1, 2, 3, 4, 5 };
	double doubleArr[] = { 1.5, 2.5, 3.5, 4.5, 5.5 };
	cout << "longArrMean = " << mean(longArr, NUM_OF_
ELEMENTS(longArr)) << "\n";
	cout << "doubleArrMean = " << mean(doubleArr, NUM_OF_
ELEMENTS(doubleArr)) << "\n";
	return 0;
}
```

Теперь функцию mean() можно использовать со многими типами.

В случае если алгоритм является общим для типов, с которыми приходится работать, можно определить шаблон функции. Определяется шаблон функции следующим образом:

1. Используется реализация функции для какого-то определенного типа.
2. Добавляется заголовок template<typename Type>, что означает, что в алгоритме используется абстрактный тип данных Type.
3. В реализации функции имя типа заменяется на Type.

В примере вместо Type использовано T, имя типа определяется пользователем. Вместо template<typename Type> в литературе встречается template<class Type>. Это одно и то же. Сосуществование ключевых слов typename и class является неудачным и запутывающим, но поддерживается по историческим причинам.

Следует отметить, что функция mean() буквально становится шаблоном для создания функций, а не полноценной функцией с рабочим транслируемым кодом. Компилятор автоматически создает (то есть инстанцирует) нужную реализацию функции, когда встречает ее вызов. На практике это означает, что код шаблонной функции должен располагаться в той же единице трансляции, что и вызов: или размещаться в том же cpp-файле, или (чаще полностью) в заголовочном файле, который подключается в компилируемый cpp-файл. Нельзя поместить прототип шаблонной функции в mean.h, а ее реализацию – в mean.cpp, и использовать, например, в main.cpp, как обычно происходит в случае использования обычных функций.

В Си++, кроме шаблонов функций, используются шаблоны классов.

Вероятно, первая задача, в которой разработчик программ на языке Си++ сталкивается с шаблонами классов, – это применение динамических массивов.

Пока мы рассматривали статические массивы, размер которых фиксируется на стадии компиляции программы, и динамическое выделение памяти. Во втором случае необходимо указать размер памяти, которая нам требуется, а следовательно, заранее каким-то образом его подсчитать. В случае использования динамического массива мы смогли бы добавлять новые элементы в массив, а выделение необходимой памяти происходило бы автоматически. Простейший пример реализации такого массива показан в листинге 4.3.

Листинг 4.3

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

template<typename T> class DynArray
{
public:
	DynArray();
	~DynArray();

	size_t GetSize() const
	{
		return m_size;
	}

	void Add(const T& newValue);
	T& GetAt(size_t index);
	T& operator[](size_t index);

protected:
	void AddCapacity();

	T* m_values;
	size_t m_capacity, m_size;
};

template<typename T> DynArray<T>::DynArray()
	: m_capacity(1), m_size(0)
{
	m_values = new T[m_capacity];
}

template<typename T> DynArray<T>::~DynArray()
{
	delete[] m_values;
}

template<typename T> void DynArray<T>::Add(const T&
newValue)
{
	if (m_size >= m_capacity)
	{
		AddCapacity();
	}

	m_values[m_size++] = newValue;
}
```

```
template<typename T> void DynArray<T>::AddCapacity()
{
	m_capacity *= 2;
	T* temp = m_values;
	m_values = new T[m_capacity];
	for (size_t i = 0; i < m_size; ++i)
	{
		m_values[i] = temp[i];
	}
	delete[] temp;
}
template<typename T> T& DynArray<T>::GetAt(size_t index)
{
	return m_values[index];
}
template<typename T> T& DynArray<T>::operator[](size_t index)
{
	return m_values[index];
}
int main()
{
	DynArray<int> myArray;
	myArray.Add(10);
	myArray.Add(33);
	myArray.Add(5);
	myArray.GetAt(1) = 44;
	for (size_t i = 0, size = myArray.GetSize(); i < size; ++i)
	{
		cout << myArray[i] << "\n";
	}
	return 0;
}
```

Программа является самой длинной, рассмотренной в данном учебнике, и в некотором смысле суммирующей материал, разберем ее подробно. В начале объявляется новый шаблон класса DynArray (от Dynamic Array, или «динамический массив»). Подобно шаблону функции в начале объявления класса добавлено «template<typename T> class DynArray».

Шаблон класса содержит три защищенных переменных m_values (значения) типа T*, m_capacity (емкость) и m_size (размер) стандартного типа size_t. Емкость – размер области памяти, выделенная под хранимые элементы с некоторым запасом. Размер – число хранимых элементов. Обычно емкость выбирается больше размера, чтобы не выделять новый участок памяти при каждом добавлении нового элемента. Другими словами, мы выделяем, например, память сразу под 5 элементов, а пользователь может внести в массив только 3 элемента, при добавлении 4-го элемента выделения дополнительной памяти не происходит. Но при добавлении 6-го элемента необходимо выделить новый участок памяти, например под 10 элементов, скопировать в него все имеющиеся 5 элементов из старого участка памяти и освободить старый. То есть выполнить достаточно большое число действий. Поэтому используются участки памяти с запасом, чтобы снизить время добавления нового элемента.

В шаблоне класса реализовано 5 методов: 4 общедоступных и 1 защищенный.

size_t GetSize() const – метод возвращает текущее число элементов в массиве. const в данном случае означает, что этот метод не будет изменять значения переменных класса. На практике это означает, что если объект класса является константой или доступ к нему происходит через константную ссылку, то вызываться могут только константные методы, объявленные таким образом. Методы, не являющиеся константными, вызываться для константных объектов не могут!

void Add(const T& newValue) – добавляет новый элемент в массив, аргумент передается через константную ссылку, так как в общем случае это может быть объект класса и нет необходимости создавать лишний экземпляр при вызове функции.

T& GetAt(size_t index) – возвращает ссылку на элемент в массиве. С помощью такой ссылки можно как считать значение элемента, так и установить новое.

T& operator[](size_t index) – вместо обычного имени используется operator[], является перегруженной операцией [] для объектов данного класса, работает аналогично методу **GetAt()**. Перегрузка операций позволяет определить действия, которые будет

выполнять операция для объектов класса. Перегрузка подразумевает создание функции, название которой содержит слово operator и символ перегружаемой операции. Перегрузить можно только те операции, которые уже определены в C++. Создать новые операции нельзя. Другими словами, операция сложения работает для базовых типов, например float. Но мы написали класс для работы с двумерными векторами, включающий переменные x и y. Компилятор «не знает» правила сложения векторов, поэтому просто сложить объекты такого двумерного вектора не получится. Чтобы операция сложения работала с объектами нового класса, необходимо ее перегрузить. Мы уже говорили, что по сути операции подобны обычным функциям, только используются знаки операции вместо традиционных имен. В данном примере перегружается оператор [], с помощью которого обеспечивается доступ к элементам обычного статического массива.

Защищенный метод void AddCapacity() увеличивает емкость массива, выделяя новый участок памяти и копируя в него уже хранимые элементы.

Далее определяется конструктор класса. Обратите внимание, что перед знаком :: ставим имя класса DynArray<T>, а не просто DynArray. Также в заголовке реализации функции пишется template<typename T>:

```
template<typename T> DynArray<T>::DynArray()
    : m_capacity(1), m_size(0)
{
    m_values = new T[m_capacity];
}
```

В конструкторе емкость инициализируется значением 1, а размер – значением 0 в списке инициализации.

Выделяется память под 1 элемент с использованием операции new для массивов. В языке C++ существует два варианта операции new – для одного объекта и для массивов. Чтобы выделить память под 5 элементов типа int, используется следующий синтаксис:

```
int* arr = new int[5];
```

В случае использования классов вместо int для элементов массива для каждого элемента будет вызван конструктор по умолчанию.

Освободить область, выделенную под массив элементов, необходимо с помощью операции delete[]:

```
delete[] arr;
```

Не путать с операцией

```
delete arr; // удаляет объект, выделенный new не для массива!
```

Освобождение выделенной памяти осуществляется в деструкторе:

```
template<typename T> DynArray<T>::~DynArray()
{
    delete[] m_values;
}
```

Следующая функция добавляет один элемент в массив:

```
template<typename T> void DynArray<T>::Add(const T& newValue)
{
    if (m_size >= m_capacity)
    {
            AddCapacity();
    }
    m_values[m_size++] = newValue;
}
```

Сначала с помощью условного оператора проверяется, достаточно ли памяти под новый элемент, и если нет, то увеличивается емкость.

В строчке

```
m_values[m_size++] = newValue;
```

новый элемент помещается с помощью операции копирования (для объектов класса, возможно, потребуется перегрузить данную операцию, чтобы копирование было осуществлено согласно задуманному алгоритму, а не побитово по умолчанию!). Причем новый элемент помещается в позицию m_size (при первом использовании имеет значение 0, см. конструктор в листинге 4.3), а только затем значение m_size увеличивается на 1. Следует различать m_values[m_size++] от m_values[++m_size], где сначала значение m_size увеличилось бы на 1, а затем newValue было бы сохранено под неправильным индексом.

Самый длинный метод увеличивает емкость динамического массива.

```
template<typename T> void DynArray<T>::AddCapacity()
{
    m_capacity *= 2;
    T* temp = m_values;
    m_values = new T[m_capacity];

    for (size_t i = 0; i < m_size; ++i)
    {
        m_values[i] = temp[i];
    }

    delete[] temp;
}
```

Сначала значение переменной m_capacity удваивается, то есть емкость при добавлении элементов будет равна 1, 2, 4, 8, 16 и т. д. Такой подход немного отличается от подхода, рассмотренного выше. Далее указатель на старую область памяти сохраняется во временный указатель temp (T* temp = m_values). Выделяется новый участок памяти, который уже в два раза больше (m_values = new T[m_capacity];). В итерационном цикле for копируем все элементы из старой области памяти в новую (m_values[i] = temp[i];). И освобождаем старый участок памяти (delete[] temp;).

Последние два метода реализуют доступ к элементам массива:

```
template<typename T> T& DynArray<T>::GetAt(size_t index)
{
    return m_values[index];
}

template<typename T> T& DynArray<T>::operator[](size_t index)
{
    return m_values[index];
}
```

Устроены методы идентично, возвращают элемент через указатель на область памяти m_values. Для упрощения кода выход за границы массива не проверяется.

В главной функции main() происходит тестирование шаблона класса.

```
int main()
{
    DynArray<int> myArray;
    myArray.Add(10);
    myArray.Add(33);
    myArray.Add(5);

    myArray.GetAt(1) = 44;

    for (size_t i = 0, size = myArray.GetSize(); i < size; ++i)
    {
            cout << myArray[i] << "\n";
    }
    return 0;
}
```

В начале объявляется новый объект myArray типа DynArray<int>. Компилятор инстанцирует шаблон класса DynArray для работы с типом int.

Далее мы добавляем в массив три элемента: 10, 33 и 5.

Затем элемент с индексом 1 (то есть 33) меняем на 44. Здесь GetAt() возвращает ссылку, по которой меняется значение элемента в массиве. Хоть такая запись и рабочая, но немного запутанная. Поэтому GetAt() можно использовать для чтения элементов, а не для записи. Но такая конструкция показывает, почему аналогичная запись вида:

```
myArray[1] = 44;
```

будет работать. Операция [] перегружена в шаблоне класса DynArray, возвращает ссылку на элемент массива, который мы можем изменить.

Через строчку как раз и используется перегруженная операция [] для вывода элементов массива.

Результат работы программы показан на рис. 4.1.

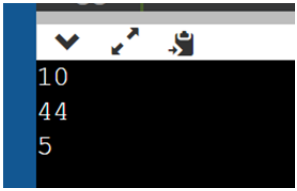

Рис. 4.1. Результат работы программы
из листинга 4.3

Рассмотренные примеры приведены для лучшего понимания идей обобщенного программирования и работы шаблонов классов. В том числе шаблонов классов, объекты которых предназначены для хранения других объектов, так называемых контейнеров.

Шаблоны классов гораздо более сложные и функциональные, чем рассмотренный нами DynArray<T>, они уже входят в состав стандартной библиотеки языка Си++.

Стандартная библиотека шаблонов (Standard Template Library, сокращенно STL) – это часть стандартной библиотеки языка Си++, включающая контейнеры и алгоритмы для управления ими.

Архитектура STL была разработана Александром Александровичем Степановым и Менг Ли.

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

1) последовательные, например массивы (array), векторы (vector), списки (list);
2) ассоциативные, например словарь (map);
3) неупорядоченные ассоциативные контейнеры (unordered_map).

Далее рассмотрим шаблон класса std::vector, являющийся самым популярным последовательным контейнером. Контейнер std::vector ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер std::vector является наиболее предпочтительным последовательным контейнером для большинства областей применения. Если вы сомневаетесь в выборе вида последовательного контейнера, начните с использования вектора.

std::vector подобен рассмотренному динамическому массиву DynArray, но гораздо более функционален. Не следует путать std::vector с контейнером std::array, который является статическим массивом без возможности увеличения длины, по сути, заменой статических массивов языка Си.

Пример использования контейнера vector приведен в листинге 4.4.

Листинг 4.4

```
#include <iostream>
#include <vector> // контейнер std::vector
using namespace std;

int main()
{
    vector<int> myVec; // std::vector,
// но мы использовали «using namespace std;»
    myVec.push_back(10);
    myVec.push_back(33);
    myVec.push_back(5);

    myVec.at(1) = 44;

    for (size_t i = 0, size = myVec.size(); i < size; ++i)
    {
            cout << myVec[i] << "\n";
    }
    return 0;
}
```

Программа очень похожа на программу из листинга 4.3. Для работы с контейнером необходимо включить файл с шаблоном класса с помощью «#include <vector>».

Некоторые методы std::vector приведены в табл. 4.1.

Таблица 4.1

Неполный список методов std::vector

Тип метода	Метод	Описание
Конструкторы	vector::vector	Конструктор по умолчанию. Создает новый экземпляр вектора
	vector::vector(const vector& b)	Конструктор копирования. Создает копию вектора b, копируя все его объекты
	vector::vector(size_t N, const T& val = T())	Создает вектор с N объектами. Если val объявлена, то каждый из этих объектов будет инициализирован ее значением. vector<double> v1(10, 5.0); // 10 элементов, каждый имеет значение 5.0 В противном случае объекты получат значение конструктора по умолчанию типа T
Деструктор	vector::~vector	Уничтожает вектор и его элементы
Операторы	vector::operator=	Копирует значение одного вектора в другой
	vector::operator==	Сравнение двух векторов
Доступ к элементам	vector::at	Доступ к элементу с проверкой выхода за границу
	vector::operator[]	Доступ к определенному элементу
	vector::front	Доступ к первому элементу
	vector::back	Доступ к последнему элементу
Работа с размером вектора	vector::empty	Возвращает true, если вектор пуст, то есть не содержит элементов
	vector::size	Возвращает количество элементов в векторе
	vector::max_size	Возвращает максимально возможное количество элементов в векторе на данной вычислительной платформе

Окончание табл. 4.1

Тип метода	Метод	Описание
	vector::reserve	Устанавливает минимально возможное количество элементов в векторе
	vector::capacity	Возвращает количество элементов, которое может содержать вектор до того, как ему потребуется выделить больше места
	vector::shrink_to_fit	Уменьшает количество используемой памяти за счет освобождения неиспользованной
Модификаторы	vector::clear	Удаляет все элементы вектора
	vector::insert	Вставка элементов в вектор
	vector::erase	Удаляет указанные элементы вектора (один или несколько)
	vector::push_back	Вставка элемента в конец вектора
	vector::pop_back	Удаляет последний элемент вектора
	vector::resize	Изменяет размер вектора на заданную величину
	vector::swap	Обменивает содержимое двух векторов

Отметим, что работа контейнеров сопряжена с динамическим выделением и освобождением памяти, что практически невозможно в случае работы программы на микроконтроллерах с несколькими килобайтами оперативной памяти. С этой точки зрения STL просто недоступна на таких платформах, как Arduino. Однако контейнер vector находит широкое применение в миникомпьютерах, например в реализации библиотеки компьютерного зрения OpenCV на Си++. Так, найденные элементы на анализируемом изображении функциями OpenCV, в том числе контуры, хранятся в контейнере именно этого типа.

4.2. Ввод и вывод на языке Си++

Стандартная библиотека потоков ввода-вывода обеспечивает буферизированный ввод-вывод текстовых и числовых значений. Класс **ostream** преобразует типизированные объекты в набор символов или байтов по следующей схеме:

типизированные значение ('A', 123 или 456.78f) → ostream → последовательность байтов / буфер потока → «где-то».

«Где-то» может быть файлом на диске, экраном монитора или строкой в памяти ЭВМ.

istream, наоборот, преобразует поток символов (или байтов) в типизированные объекты:

«где-то» → последовательность байтов / буфер потока → istream → типизированные значение ('B', 321 или 876.54).

В случае ввода «где-то» может быть файлом на диске, клавиатурой или строкой в памяти ЭВМ. Другие формы взаимодействия с пользователем, такие как графический интерфейс или ввод с помощью мыши, обеспечиваются библиотеками, не входящими в стандарт ISO языка C++.

Типы данных могут быть как базовыми (int, float и т. п.), так и пользовательскими (классы). В случае использования классов операции над потоками программисту необходимо самостоятельно расширить для пользовательских типов данных. Потоки могут использоваться как для текстового ввода-вывода, так и для бинарного. При использовании текстового ввода-вывода также возможно использование локализаций.

Все классы библиотеки потоков ввода-вывода имеют деструкторы, которые освобождают ресурсы, которыми владеют, например, дескрипторы файлов. Иными словами, при уничтожении потока файл закрывается автоматически.

Ранее мы использовали стандартные потоки ввода cin и вывода cout, являющиеся объектами классов istream и ostream соответственно. Пример использования представлен в листинге 4.5.

Листинг 4.5

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

int main()
{
    int x, y;
    cin >> x >> y;
    cout << “x = “ << x << “, y = “ << y;
}
```

>> является перегруженной операцией сдвига вправо для класса istream и теперь выполняет не побитовый сдвиг, а работает в качестве операции ввода. Тип правого операнда определяет, какие входные данные приемлемы и какой объект целевой. На рис. 4.2 показан некорректный ввод и что получается в выводе для рассмотренной программы.

Рис. 4.2. Пример некорректного ввода и его интерпретация

Поскольку x и y являются целочисленными переменными, то ожидается ввод символов вроде 12 34, а любой символ, не являющийся цифрой, прекращает ввод. Разделителем является пробельный символ (пробел, tab, новая строка).

Ввод и вывод можно объединять в цепочки:

```
int x;
double y;
cin >> x >> y;
cout << «x = « << x << «, y = « << y;
```

Приведенный выше код аналогичен такому коду:

```
int x;
cin >> x;
double y;
cin >> y;
cout << “x = “ << x << “, y = “ << y;
```

Корректный ввод показан на рис. 4.3.

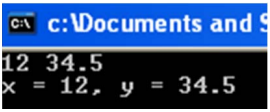

Рис. 4.3. Пример корректного ввода и его интерпретация

Часто требуется считать последовательность символов. Это можно сделать путем помещения введенной информации в переменную типа std::string, как показано в листинге 4.6.

Листинг 4.6

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

int main()
{
    string str;
    cin >> str;
    cout << “entered: “ << str;
}
```

Результат работы программы показан на рис. 4.4.

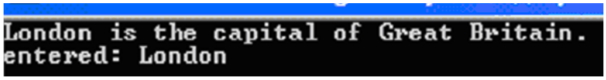

Рис. 4.4. Ввод строки

Как видно из рисунка, ввод заканчивается первым пробелом. Прочесть строку целиком можно с помощью функции getline():

getline(stream, string, separator);

Первый аргумент – поток данных, второй – строка std::string, а третий обозначает разделитель (по умолчанию – символ новой строки). Пример приведен в листинге 4.7.

Листинг 4.7

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

int main()
{
    string str;
    getline(cin, str);
```

```
    cout << «entered: « << str;
}
```

Результат работы показан на рис. 4.5.

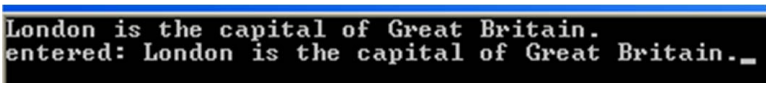

Рис. 4.5. Использование функции getline()

Также данную функцию можно использовать как метод класса:

cin.getline(string, streamsize, separator);

Первый аргумент – строка Си (переменная типа char*), второй – максимальное число символов в строке, третий аргумент необязателен и обозначает разделитель. В листинге 4.8 показан пример.

Листинг 4.8

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

int main()
{
    char str[256];
    cin.getline(str, 256, ';');
    cout << «entered: « << str;

}
```

При вводе последовательности символов «123;456» на экране появится надпись «entered: 123».

В дополнение к вводу-выводу встроенных типов и строк стандартная библиотека позволяет программистам определять ввод-вывод для собственных типов.

Рассмотрим пример ввода-вывода информации о книгах:

```
class Book
{
public:
    Book(const string& title, int year)
    {
        _title = title;
```

```
            _year = year;
        }

        string _title;
        int _year;
};
```

Класс включает в себя две переменные _title и _year для хранения названия книги и года издания, а также конструктор для удобного создания объектов. Для упрощения примера _title и _year являются public-переменными. Теперь попробуем вывести информацию о книге на экран:

```
int main()
{
    Book myBook1("Dunno's Adventures", 2022);
    cout << myBook1; // Ошибка binary '<<' : no operator found which takes a right-hand operand of type 'Book' (or there is no acceptable conversion)
}
```

Необходимо определить операцию << для типа Book:

```
ostream& operator<<(ostream& os, const Book& book)
{
    return os << "{ \"" << book._title << "\", " << book._year << " }\n";
}
```

Пример ввода для пользовательского класса показан в листинге 4.9.

Листинг 4.9

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

class Book
{
public:
    Book(const string& title, int year)
    {
        _title = title;
```

```
            _year = year;
        }
        string _title;
        int _year;
    };
    ostream& operator<<(ostream& os, const Book& book)
    {
        return os << "{ \"" << book._title << "\", " << book._year <<
" }\n";
    }
    int main()
    {
        Book myBook1("Dunno's Adventures", 2022);
        Book myBook2("Cheburashka", 2023);
        cout << myBook1 << myBook2;
    }
```

Результат работы программы показан на рис. 4.6.

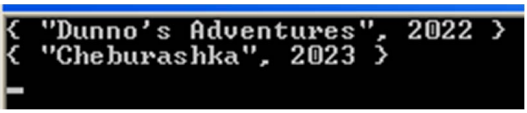

Рис. 4.6. Вывод данных пользовательского класса

Соответствующая операция ввода будет немного более сложной, нам придется проверять корректность ввода и обрабатывать ошибки. Пользователь вводит последовательность символов вроде { «Dunno›s Adventures», 2000 }. Во введенной пользователем последовательности необходимо проверить первый символ {, записать все символы от первых кавычек до вторых в строку _title, а символы после запятой преобразовать в число и сохранить в переменную _year типа int.

Полный код примера использования операции ввода приведен в листинге 4.10.

Листинг 4.10

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

```
class Book
{
public:
	Book()
	{
		_year = 0;
	}
	Book(const string& title, int year)
	{
		_title = title;
		_year = year;
	}
	string _title;
	int _year;
};
ostream& operator<<(ostream& os, const Book& book)
{
	return os << "{ \"" << book._title << "\", " << book._year <<
" }\n";
}
istream& operator>>(istream& is, Book& book)
{
	char c, c2;
	if (is >> c && c == '{' && is >> c2 && c2=='»') // первый
символ в последовательности должен быть {
	{ // пропускаем все пробельные символы до первых « и
сами кавычки
		string title;
		while (is.get(c) && c != '»') // считываем все симво-
лы до вторых «
			title += c; // в строку title
		if (is >> c && c == ',')
		{
			int year;
			if (is >> year >> c && c == '}')
			{
				book._title = title;
				book._year = year;
```

```
                    return is;
                }
            }
        }
        is.setstate(ios_base::failbit);
        return is;
}
int main()
{
        Book myBook;
        cin >> myBook;
        if (cin.fail())
        {
                cout << “FAILED\n”;
        }
        else
        {
                cout << “SUCCESS: “ << myBook;
        }
}
```

В модифицированной программе добавлен конструктор по умолчанию для класса Book. В главной функции main() происходит считывание данных новой книги из потока. Вместо { “название книги”, год издания } пользователь может ввести некорректные данные, в этом случае cin >> myBook не изменяет объект книги, а стандартная функция cin.fail() возвращает истинное значение. Для указания ошибочного состояния потока необходимо использовать cin.setstate(ios_base::failbit). Сбросить состояние ошибки в состояние good можно, выполнив функцию cin.clear() (в примере не используется).

Сам поток можно использовать для индикации ошибки:

```
if (is >> c)
{
}
```

Также операция >> пропускает пробельные символы в отличие от функции is.get(), которую мы используем для чтения заголовка книги, в котором могут быть пробелы. Проверка if (is >> year) в коде вернет ложное значение, если введенные данные не являются чис-

лом, так как year – переменная типа int. На рис. 4.7 показан результат работы программы для корректного и некорректного ввода.

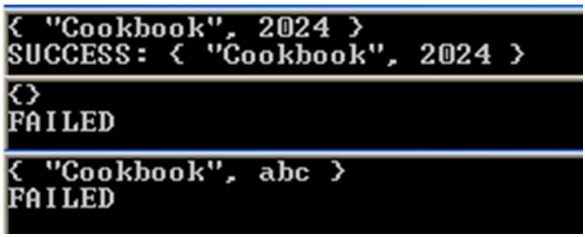

Рис. 4.7. Потоковый ввод для пользовательского типа данных

Теперь можно использовать полиморфизм в полной мере и выполнить чтение из файла или записать данные в файл.

Для этого вместо стандартного выходного потока в консоль следует просто изменить объект с std::cout на объект класса std::ifstream или std::ofstream, производных от std::iostream. Пример работы с файлом показан в листинге 4.11.

Листинг 4.11

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

int main()
{
    ifstream input("input.txt");

    if (!input.is_open())
    {
        cout << "input.txt not found\n";
        return 0;
    }

    ofstream output("output.txt");
    double value;
    while (input >> value)
    {
        cout << "value = " << value << "\n";
        output << value + 1 << "\n";
    }
}
```

Содержимое файла input.txt в папке с проектом:

0 1.5 2.5 5

4

Содержимое файла ouput.txt после выполнения программы:

1

2.5

3.5

6

5

В начале программы создается объект input класса std::ifstream. В конструктор класса передается имя файла, который необходимо открыть. Дальше проверяется, что файл существует и открыт: «if(!input.is_open())». Функция is_open() специфичная для файловых потоков. Подобным образом создается объект класса std::ofstream для сохранения информации. Далее работа осуществляется как с cout и cin. Читаем значение с файла, прибавляем единицу, сохраняем в выходной файл. Закрывать файлы нет необходимости. Они будут автоматически закрыты в деструкторе класса.

Выводы

Никакая серьезная программа не может быть написана просто на языке программирования. Писать на чистом языке программу в большинстве случаев утомительно, долго и экономически неэффективно. Любая задача может быть упрощена благодаря использованию хороших библиотек.

Примерно две трети стандарта ISO C++ – спецификация стандартной библиотеки (stdlib). Она содержит высокопроизводительный общий код, который гарантированно доступен по умолчанию и соответствует стандартам.

Три важнейших компонента стандартной библиотеки – контейнеры, итераторы и алгоритмы.

Контейнеры – структуры данных, отвечающие за хранение последовательностей объектов. Они правильные, безопасные и обычно как минимум настолько же эффективные, как и те, что вы могли бы выполнить вручную.

Контрольные вопросы

1. Что такое сигнатура функции?
2. Что такое шаблон класса? Почему он обычно размещается в заголовочном файле? Может ли шаблон класса находиться в cpp-файле?
3. Чем операция new отличается от функции malloc()?
4. Что такое контейнер?
5. Как объявить вектор, содержащий значения типа int?
6. Как добавить новый элемент в вектор? Как его удалить?
7. Что такое перегрузка операций?
8. Что такое std::cout?
9. Чем функция getline() отличается от метода getline()?

ЗАКЛЮЧЕНИЕ

Представленный в пособии материал позволяет сделать следующие выводы.

В сравнении с процедурным программированием на Си объектно-ориентированное программирование на Си++ требует больше затрат на изучение, однако сложные программы на нем написать легче.

Из-за взрывного роста использования Си++ стало ясно, что неизбежна формальная стандартизации Си++.

Стандарт ISO C++ определяет два типа сущностей:

1) фундаментальные возможности языка, например встроенные типы int, double или циклы while;
2) компоненты стандартных библиотек, такие как контейнеры, например vector, или операции ввода-вывода, например getline().

C++ 11 (ISO/IEC 14882-2011) значительно увеличил стандартную библиотеку.

В настоящее время стандарт Си++ обновляется каждые три года (C++ 11, C++ 14, C++ 17, C++ 20, C++ 23), в результате чего в языке добавляются новые фундаментальные возможности языка, появляются новые ключевые слова и обновляется стандартная библиотека Си++, позволяя, например, составлять многопоточные программы. Это требует постоянной актуализации знаний, навыков и умений.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Свердлов, С. З. Языки программирования и методы трансляции : учеб. пособие / С. З. Свердлов. – Изд. 4-е, стер. – Санкт-Петербург [и др.] : Лань, 2024. – 560 с. – URL: e.lanbook.com/book/362948 (дата обращения: 02.01.2024). – Режим доступа: по подписке. – ISBN 978-5-507-48776-9.
2. Гагарина, Л. Г. Основы программирования на языке С : учеб. пособие / Л. Г. Гагарина, Е. Г. Дорогова. – 2-е изд., испр. и доп. – Москва : ИНФРА-М, 2023. – 268 с. – (Высшее образование – Бакалавриат). – URL: znanium.ru/catalog/document?id=418753 (дата обращения: 02.01.2023). – Режим доступа: по подписке. – ISBN 978-5-16-104798-9.
3. Васильев, А. Н. Объектно-ориентированное программирование на С++ / А. Н. Васильев. – Санкт-Петербург : Наука и Техника, 2016. – 543 с. – (Просто о сложном). – ISBN 978-5-94387-984-5.
4. Страуструп, Б. Язык программирования С++ для профессионалов : учебник / Б. Страуструп. – 3-е изд. (электрон). – Москва : Интернет-Университет Информационных Технологий [и др.], 2021. – 670 с. – URL: www.iprbookshop.ru/102077.html (дата обращения: 02.01.2023). – Режим доступа: по подписке. – ISBN 978-5-4497-0922-6.
5. Боровский, А. С. Программирование микроконтроллера Arduino в информационно-управляющих системах : учеб. пособие / А. С. Боровский, М. Ю. Шрейдер ; Оренбургский государственный университет. – Оренбург : ОГУ, 2017. – 112 с. – URL: www.iprbookshop.ru/78913.html (дата обращения: 02.01.2023). – Режим доступа: по подписке. – ISBN 978-5-7410-1853-8.

Содержание

Учебное издание

Глибин Евгений Сергеевич

ЯЗЫКИ ВЫСОКОГО УРОВНЯ В СИСТЕМАХ УПРАВЛЕНИЯ

Учебное пособие

Редактор *Е.В. Пилясова*
Технический редактор *Н.П. Крюкова*
Компьютерная верстка: *Л.В. Сызганцева*
Дизайн обложки: *Г.В. Карасева*

При оформлении пособия использовано изображение от lcd2020 на Freepik (сайт ru.freepik.com)

Подписано в печать 08.10.2025. Формат 60×84/16.
Печать оперативная. Усл. п. л. 8,31.
Тираж 100 экз. Заказ № 1-13-23.

Издательство Тольяттинского государственного университета
445020, г. Тольятти, ул. Белорусская, 14,
тел. 8 (8482) 44-91-47, www.tltsu.ru