СОДЕРЖАНИЕ

ЛЕКЦИЯ 1. ОСНОВЫ СИ++ 7

1. Несколько замечаний о назначении программирования 7
2. Происхождение языка Си++ 9
3. Стандарт ANSI Си++ 9
4. Среда разработки Microsoft Developer Studio Visual C++ 10
5. Пример программы на Си++ 10
6. Выполнение ввода/вывода данных и присваивание значений 12
7. Управление порядком выполнения команд с помощью оператора if 13
8. Оформление исходного текста 15
9. Сводка результатов 15
10. Упражнения 15

ЛЕКЦИЯ 2. ПЕРЕМЕННЫЕ, ТИПЫ ДАННЫХ И ВЫРАЖЕНИЯ 18

1. Идентификаторы 18
2. Типы данных 18
3. Вывод вещественных чисел на экран 22
4. Описания, константы и перечисления 24
5. Присваивание и выражения 26
6. Сводка результатов 28
7. Упражнения 28
8. Приложения 29

ЛЕКЦИЯ 3. ФУНКЦИИ И ПРОЦЕДУРНАЯ АБСТРАКЦИЯ 31

1. Назначение подпрограмм 31
2. Определение новых функций 31
3. Способы передачи параметров внутрь функций 33
4. Полиморфизм и перегрузка функций 35
5. Процедурная абстракция и "хороший" стиль программирования 36
6. Модульное программирование 36
7. Сводка результатов 38
8. Упражнения 39

ЛЕКЦИЯ 4. ТЕКСТОВЫЕ ФАЙЛЫ И ПОТОКИ ВВОДА/ВЫВОДА 41

1. Назначение файлов 41
2. Потоки ввода/вывода 41
3. Проверка ошибок выполнения файловых операций 43
4. Символьный ввод/вывод 44
5. Проверка достижения конца файла при операциях ввода 45
6. Передача потоков функциям в качестве параметров 47
7. Операторы ввода/вывода ">>" и "<<" 48
8. Сводка результатов 50
9. Упражнения 50

ЛЕКЦИЯ 5. ОПЕРАТОРЫ ВЕТВЛЕНИЯ И ЦИКЛЫ 52

1. Логические значения, выражения и функции 52
2. Циклы "for", "while" и "do...while" 53
3. Множественное ветвление и оператор "switch" 55
4. Блоки и область видимости переменных 56

5. Замечание о вложенных циклах 59
6. Сводка результатов 59
7. Упражнения 60

ЛЕКЦИЯ 6. МАССИВЫ И СИМВОЛЬНЫЕ СТРОКИ 63

1. Назначение массивов 63
2. Передача массивов в качестве параметров функций 66
3. Сортировка массивов 68
4. Двумерные массивы 69
5. Символьные строки 70
6. Сводка результатов 73
7. Упражнения 73

ЛЕКЦИЯ 7. УКАЗАТЕЛИ 75

1. Назначение указателей 75
2. Переменные типа "массив". Арифметические операции с указателями 79
3. Динамические массивы 81
4. Автоматические и динамические переменные 82
5. Связные списки 82
6. Сводка результатов 86
7. Упражнения 87

ЛЕКЦИЯ 8. РЕКУРСИЯ 89

1. Понятие рекурсии 89
2. Простой пример рекурсии 89
3. Как выполняется рекурсивный вызов 90
4. Еще три примера рекурсии 92
5. Рекурсия и циклы 93
6. Рекурсия в структурах данных 94
7. Рекурсивная реализация алгоритма быстрой сортировки 94
8. Сводка результатов 97
9. Упражнения 97

ЛЕКЦИЯ 9. СОСТАВНЫЕ ТИПЫ ДАННЫХ 100

1. Назначение составных типов данных 100
2. Описание и инициализация структур 100
3. Доступ к компонентам структуры через указатель 103
4. Массивы и структуры 104
5. Перегрузка операторов 105
6. Применение структур для реализации стека 107
7. Сводка результатов 111
8. Упражнения 112

ПРИЛОЖЕНИЕ. КРАТКОЕ РУКОВОДСТВО ПО СРЕДЕ РАЗРАБОТКИ DEVELOPER STUDIO VISUAL C++ 113

1. Создание нового проекта 113
2. Добавление в проект нового исходного файла 114
3. Сборка проекта 115
4. Запуск нового приложения 116

ЛИТЕРАТУРА 117

ЛЕКЦИЯ 1. Основы Си++

1. Несколько замечаний о назначении программирования

Программирование – это техническая творческая деятельность, цель которой заключается в решении важных для человека задач или выполнении определенных действий с помощью компьютера. На рис. 1 представлена идеализированная схема решения типичной задачи программирования.

Рис. 1. Схема решения задачи с помощью компьютера.

В рамках такой схемы необходимыми компонентами компьютера являются центральный процессор, устройства ввода/вывода и память (рис. 2).

Рис. 2. Основные компоненты компьютера.

Конечно, в действительности дело обстоит не так просто, как показано на рис. 1. Например, "подробное описание (спецификация) задачи" на естественном языке для компьютера не годится (в настоящее время). Более того, для решения задачи на компьютере недостаточно полного описания задачи, необходимо также снабдить компьютер информацией о том, как именно следует решать задачу – т.е. составить алгоритм. Для описания алгоритмов решения задач или алгоритмов выполнения каких-либо действий (например, управление роботом-манипулятором) с помощью компьютера применяются языки программирования.

На рис. 3 показана более подробная схема решения задачи с помощью компьютера, в которой учтена необходимость использования языка программирования. Иллюстрация этой схемы на конкретном примере приведена в таблице 1.

Существует большое количество различных языков программирования и много способов их классификации. Например, "языками высокого уровня" считаются те языки, синтаксис которых сравнительно близок к естественному языку, в то время как синтаксис "низкоуровневых" языков содержит много технических подробностей, связанных с устройством компьютера и процессора.

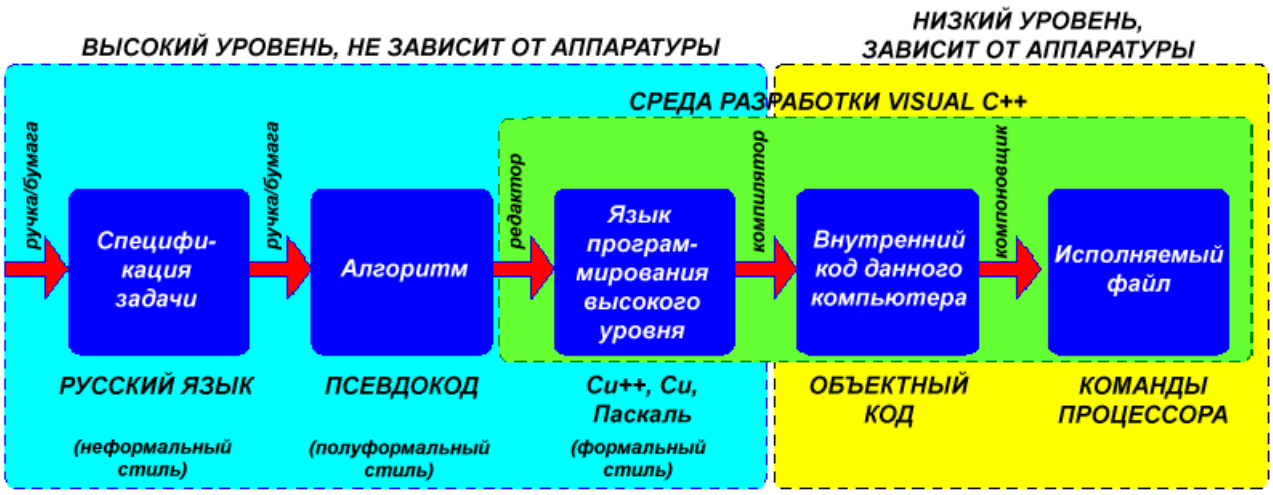

Рис. 3. Схема решения задачи на компьютере с использованием языка программирования.

Таблица 1. Основные этапы решения задачи по проверке числа на простоту.

Спецификация задачи	Требуется определить, является ли данное число простым.
Алгоритм	*Ввести x* *Для каждого целого числа z из диапазоне от 1 до x* *Если остаток от деления x на z равен 0, то* *вывести сообщение "число не простое" и закончить работу* *Если такого числа z не найдено, то* *вывести сообщение "число простое" и закончить работу*

Описание алгоритма на языке высокого уровня	`#include <iostream.h>`
	`int main()`
	`{`
	`  int x;`
	`  cout << "Введите число:\n";`
	`  cin >> x;`
	`  for (int z=2; z<x; z++)`
	`  if (x % z == 0)`
	`  {`
	`  cout << "Это не простое число.\n";`
	`  return 0;`
	`  }`
	`  cout << "Это простое число.\n";`
	`  return 0;`
	`}`

Объектный код (внутренний код конкретного компьютера)	Двоичные команды процессора (частично)
Исполняемый файл для конкретного компьютера	Двоичные команды процессора (полностью)

"Императивные" или "процедурные" языки позволяют программисту описать, в какой последовательности компьютер будет выполнять отдельные шаги алгоритма и, таким образом, решать задачу и выдавать на экран результат. "Декларативные" языки предназначены больше для описания самой задачи и желаемого результата, а не действий компьютера.

"Объектно-ориентированные языки" рассчитаны на применение особого подхода к описанию задач, согласно которому в задаче выделяются некоторые "объекты" с характерным для них "поведением" и взаимодействующие между собой. Один из

первых объектно-ориентированных языков – Смоллток, он предназначен исключительно для объектно-ориентированного программирования. В отличие от него, язык Си++ обладает как объектно-ориентированными возможностями, так и средствами традиционного процедурного программирования.

Радикальные приверженцы различных языков и стилей программирования иногда делают экстравагантные заявления, выделяющие семейство языков или один язык как исключительный и идеально подходящий для любых задач. Например, довольно распространено мнение, что объектно-ориентированный подход наиболее близок к способу решения задач человеком. По этому поводу вы со временем сможете составить собственное мнение, т.к. абсолютно истинного, очевидно, нет.

2. Происхождение языка Си++

Язык Си++ был разработан в начале 1980-х гг. Бьерном Страуструпом из компании AT&T Bell Laboratories. Си++ основан на языке Си. Два символа "++" в названии – это игра слов, символами "++" в языке Си обозначается операция инкремента (увеличение значения переменной на 1). Т.о., Си++ был задуман как язык Си с расширенными возможностями. Большая часть языка Си вошла в Си++ как подмножество, поэтому многие программы на Си можно скомпилировать (т.е. превратить в набор низкоуровневых команд, которые компьютер может непосредственно выполнять) с помощью компилятора Си++.

При классификации языков программирования язык Си вызывает некоторые трудности. По сравнению с ассемблером, это высокоуровневый язык. Однако Си содержит много низкоуровневых средств для непосредственных операций с памятью компьютера. Поэтому язык Си отлично подходит для написания эффективных "системных" программ. Но программы других типов на Си могут оказаться довольно сложными для понимания, и есть ряд ошибок, которым программы на Си особенно подвержены. Дополнительные объектно-ориентированные возможности Си++ были добавлены в Си, в частности, для устранения этих недостатков.

3. Стандарт ANSI Си++

Национальный Институт Стандартизации США (American National Standards Institution, ANSI) разработал "официальные" стандарты для многих языков программирования, в том числе для Си и Си++. Эти стандарты стали общепринятыми и они имеют очень большое значение. Программу, целиком написанную на ANSI Си++, гарантированно можно запустить на любом компьютере, для которого имеется компилятор ANSI Си++. Другими словами, стандарт гарантирует переносимость программ на языке ANSI Си++.

В действительности большинство версий Си++ представляют собой стандартный ANSI Си++, дополненный некоторыми машинно-зависимыми возможностями. Эти специфические средства предназначены для облегчения взаимодействия программ с конкретными операционными системами. Вообще, в программах, которые должны быть переносимыми, подобными специфическими возможностями следует пользоваться как можно реже. В таких случаях части программы на Си++, в которых используются не-ANSI компоненты языка, целесообразно особым образом помечать, так, чтобы их легко можно было отделить от основной части программы и модифицировать для других компьютеров и операционных систем.

4. Среда разработки Microsoft Developer Studio Visual C++

Известно, что лучший способ изучения языка программирования заключается в том, чтобы писать на нем программы и проверять, как они работают на компьютере. Для этого необходимы несколько программ:

- Текстовый редактор, с помощью которого можно набирать и редактировать исходный текст программ на Си++.
- Компилятор. Эта программа выполняет преобразование исходного текста в машинные команды, которые компьютер может непосредственно выполнять.
- Компоновщик, который собирает отдельные скомпилированные части программы в единое целое и, при необходимости, добавляет к ним компоненты из готовых библиотек. В результате компоновки получается готовая к запуску программа – исполняемый файл.
- Отладчик, с помощью которого легче искать ошибки в программе. Ошибки могут обнаружиться как при компиляции, так и во время работы программы.

В данном курсе изучения Си++ практические упражнения предполагается выполнять в среде разработки программ **Microsoft Developer Studio Visual C++** для IBM-совместимых ПК под управлением **Windows 95/NT**. В этом пакете интегрированы редактор, компилятор, компоновщик и отладчик. Все вместе они образуют единую удобную среду программирования. Краткое описание работы со средой **Visual C++** приведено в ***Приложении***.

5. Пример программы на Си++

Ниже приведен исходный текст простой программы на Си++.

```
// В языке Си++ с двойной косой черты начинаются комментарии
// (например, как эта строка). Компилятор игнорирует комментарии,
// начиная от первой черты и до конца строки.
/* Второй способ записи комментариев – после косой черты со звездочкой.
   После текста комментария надо поставить звездочку, а затем – косую
   черту. Комментарии, записанные подобным образом, могут занимать
   больше одной строки. */
/* В программе ОБЯЗАТЕЛЬНО должно быть достаточное количество
   комментариев! */
/* Эта программа запрашивает у пользователя текущий год, возраст
   пользователя и еще один год. Затем программа вычисляет возраст
   пользователя, который будет у него во втором введенном году.*/
#include <iostream.h>

int main()
{
  int year_now, age_now, another_year, another_age;

  cout << "Введите текущий год и нажмите ENTER.\n";
  cin >> year_now;

  cout << "Введите свой возраст (в годах).\n";
  cin >> age_now;
```

```
  cout << "Введите год, для которого вы хотите узнать свой возраст.\n";
  cin >> another_year;

  another_age = another_year - (year_now - age_now);
  if (another_age >= 0)
    {
    cout << "В " << another_year << " году вам будет ";
    cout << another_age << "\n";
    }
  else
    {
    cout << "В " << another_year << " вы еще не родились!\n";
    }

  return 0;
}
```

Программа 5.1.

Некоторые свойства программы 5.1 являются обычными для большинства программ на Си++. Программа начинается (после комментариев) с оператора

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

Этот оператор называется "директивой `include`". До компилятора исходный текст обрабатывается *препроцессором* – специальной программой, которая модифицирует текст программы по специальным командам – директивам. Директивы препроцессора начинаются с символа "`#`". Директива `include` предназначена для включения в исходный текст содержимого другого файла. Например, в программу 5.1 включается файл `iostream.h`, содержащий описания функций стандартной библиотеки ввода/вывода для работы с клавиатурой и экраном. (Стандартные библиотеки языка Си++ будут рассматриваться позже).

Алгоритм, записанный в программе 5.1, очень простой. Поэтому структуру программы легко представить в виде списка последовательно выполняемых команд (операторов). Схематично программу, содержащуюся после директивы `#include`, можно представить в виде:

```
int main()
{
        Первый оператор;
        ...
        ...
        Последний оператор;
        return 0;
}
```

Подобная структура является общей для всех программ на Си++. Каждый оператор в теле программы завершается точкой с запятой. В хорошо разработанной большой программе большинство операторов являются обращениями (вызовами) к подпрограммам, которые записываются после функции `main()` или в отдельных файлах. Каждая подпрограмма (функция) имеет структуру, подобную функции `main()`. Но функция `main()` в каждой программе только одна. Именно с нее начинается выполнение программы. (Подробнее функции будут рассматриваться далее.)

В конце функции `main()` записана строка:

```
return 0;
```

Эта строка значит "вернуть операционной системе в качестве сигнала об успешном завершении программы значение 0". Оператор возврата `return` применяется не только при завершении программы, но и при завершении отдельных подпрограмм. В любом случае этот оператор возвращает определенное значение на более высокий уровень управления.

В программе-примере используются четыре *переменные*:

```
year_now, age_now, another_year и another_age
```

Переменные в программировании отличаются от математических переменных. Они используются как символические имена "фрагментов оперативной памяти компьютера". При выполнении программы в различные моменты времени переменные могут хранить различные значения. В программе 5.1 первое упоминание четырех переменных содержится в строке с оператором описания переменных:

```
int year_now, age_now, another_year, another_age;
```

Этот оператор уведомляет компилятор, что для хранения четырех переменных типа "целое число" (integer – `int`) требуется выделить необходимое количество памяти. Эта область памяти будет зарезервирована в течение выполнения оставшейся части программы. Переменные всегда должны быть описаны до первого использования. В программировании хорошим стилем считается описание всех переменных, используемых в подпрограмме, в начале этой подпрограммы. В Си++ есть несколько различных типов переменных, и они будут обсуждаться немного позже.

6. Выполнение ввода/вывода данных и присваивание значений

После *компиляции* программы ее можно *запустить на выполнение*. Результат выполнения на экране будет выглядеть примерно так:

```
Введите текущий год и нажмите ENTER.
2000
Введите свой возраст (в годах).
21
Введите год, для которого вы хотите узнать свой возраст.
2017
В 2017 году вам будет 38
```

Первая, третья, пятая и седьмая строки выдаются на экран программой с помощью следующего оператора:

```
cout << Выражение1 << Выражение2 << ... << ВыражениеN;
```

Этот оператор выводит на экран сообщение:

*Выражение*1 *Выражение*2 *... Выражение*N

Последовательность операторов

```
cout << Выражение1;
cout << Выражение2;
...
...
cout << ВыражениеN;
```

приводит к аналогичному результату. Если между выражениями требуется вставить пробелы или новые строки, то их нужно указать явно, с помощью символов `" "` и `"\n"` соответственно.

Числа, показанные выше в примере выдачи на экран **полужирным** шрифтом, были напечатаны пользователем. В показанном примере оператор

```
cin >> year_now;
```

приводит к тому, что переменной `year_now` *присваивается* значение **2000**. Это происходит после того, как пользователь напечатает "**2000**" и нажмет клавишу **Enter**. В программе есть еще места, где переменным присваиваются значения, в том числе оператор присваивания:

```
another_age = another_year - (year_now - age_now);
```

Операция "=" означает "присвоить переменной, стоящей слева от знака равенства, значение, указанное справа". Проверка на равенство в Си++ обозначается двойным символом: "==".

7. Управление порядком выполнения команд с помощью оператора if

В нескольких последних строках программы (до строки "`return 0`") записано:

```
if (another_age >= 0)
  {
  cout << "В " << another_year << " году вам будет ";
  cout << another_age << "\n";
  }
else
  {
  cout << "В " << another_year << " вы еще не родились!\n";
  }
```

Оператор ветвления (условный оператор) "`if...else...`" выглядит примерно одинаково во всех процедурных языках программирования. В Си++ он называется просто *оператором if*, и его общая структура такова:

```
if ( условие )
  {
  Оператор1;
  ...
  ...
  ОператорN;
  }
else
  {
  ОператорN+1;
  ...
  ...
  ОператорN+M;
  }
```

Часть "`else` (иначе)" в операторе `if` необязательна. Более того, если после "`if ( условие )`" стоит только один оператор, то можно опустить фигурные скобки и записать оператор так:

```
if ( условие )
  Оператор1;
```

В программах условные операторы часто встречаются группами, например:

```
...
...
if (total_test_score < 50)
  cout << "Вы не прошли тест. Выучите материал как следует.\n";
else if (total_test_score < 65)
  cout << "Вы прошли тест со средним результатом.\n";
else if (total_test_score < 80)
  cout << "Вы хорошо выполнили тест.\n";
else if (total_test_score < 95)
  cout << "Вы показали отличный результат.\n";
else
  {
  cout << "Вы сдали тест нечестно!\n";
  total_test_score = 0;
  }
...
...
```

Приведенный фрагмент программы может показаться довольно сложным. Тем не менее, он соответствует правилам Си++. Это легко понять, если обратиться к синтаксической диаграмме оператора `if` (рис. 4).

В овальных или круговых рамках на синтаксических диаграммах указываются элементы языка, которые буквально так и воспроизводятся в исходном тексте программ. В прямоугольных рамках приведены элементы, требующие дальнейшего определения, возможно, с помощью других синтаксических диаграмм. Набор таких диаграмм служит формальным описанием синтаксиса языка программирования.

Обратите внимание, что на рис. 4 отсутствует символ "`;`" и разделители "`{}`". Эти элементы языка включены в определение (и синтаксическую диаграмму) для обобщенного понятия "оператор языка Си++".

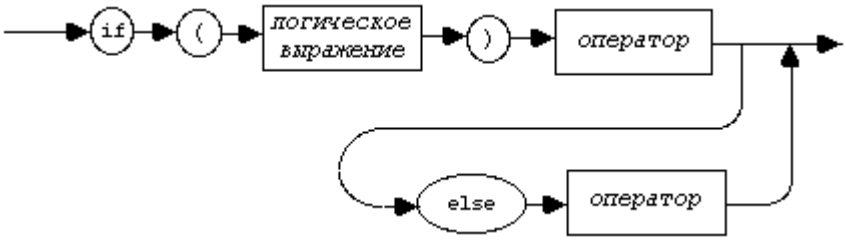

Рис. 4. Синтаксическая диаграмма оператора `if`.

При обработке приведенного фрагмента программы компилятор Си++ трактует весь текст, выделенный ниже полужирным шрифтом, как один оператор после первого слова `else`.

```
...
...
if (total_test_score < 50)
  cout << "Вы не прошли тест. Выучите материал как следует.\n";
else if (total_test_score < 65)
  cout << "Вы прошли тест со средним результатом.\n";
else if (total_test_score < 80)
  cout << "Вы хорошо выполнили тест.\n";
```

```
else if (total_test_score < 95)
  cout << "Вы показали отличный результат.\n";
else
  {
  cout << "Вы сдали тест нечестно!\n";
  total_test_score = 0;
  }
...
...
```

8. Оформление исходного текста

Между текстом программы, приведенным в п.5 и текстом, который показан ниже, для компилятора Си++ нет никаких различий.

```
#include <iostream.h> int main() { int year_now, age_now, another_year,
another_age; cout << "Введите текущий год и нажмите ENTER.\n"; cin >>
year_now; cout << "Введите свой возраст (в годах).\n"; cin >> age_now;
cout << "Введите год, для которого вы хотите узнать свой возраст.\n"; cin
>> another_year; another_age = another_year - (year_now - age_now); if
(another_age >= 0) { cout << "В " << another_year << " году вам будет ";
cout << another_age << "\n"; } else { cout << "В " << another_year << "
вы еще не родились!\n"; } return 0; }
```

Отсутствие *комментариев*, *пробелов*, *пустых строк* и *отступов* делают эту программу практически непригодной для чтения человеком. Для выработки хорошего стиля программирования, конечно, требуется знать не только правила оформления текста программы, но их следует соблюдать с самого начала. При оформлении собственных программ будьте последовательны и делайте так, чтобы отступы и пробелы отражали логическую структуру ваших программ.

Для переменных следует выбирать осмысленные имена: имена "`year_now`", "`age_now`", "`another_year`" и "`another__age`" лучше, чем "`y_n`", "`a_n`", "`a_y`" и "`a_a`" и намного лучше, чем "`w`", "`x`", "`y`" и "`z`". Это особенно важно, если в будущем ваши программы могут потребовать изменения с помощью других программистов.

9. Сводка результатов

В данной лекции кратко и неформально были рассмотрены несколько важных вопросов: переменные и типы данных, ввод и вывод, оператор присваивания и условный оператор ("оператор `if`"). Более строго и подробно эти вопросы будут рассмотрены в последующих лекциях.

10. Упражнения

Для выполнения этих упражнений требуется некоторый опыт работы с ПК под управлением операционной системы **Windows 95/NT**.

Упражнение 1

Изучите краткое руководство по **Visual C++** в ***Приложении***. Создайте проект с именем "`AgeCalculator`". Создайте исходный файл с именем `AgeCalculator.cpp`

и наберите в нем исходный текст программы 5.1. Сохраните файл на диске и добавьте его в проект. Соберите проект и запустите программу на выполнение.

Возможно, вы встретитесь со следующими проблемами:

1) ***В окне программы вместо русских букв выводятся какие-то странные символы.***
Эта проблема объясняется различием таблиц кодировок **Windows** и **DOS**. В этих таблицах русские буквы расположены в разных местах. Консольные программы при работе используют кодировку **DOS**, а текстовый редактор **Visual C++** – кодировку **Windows**. Поэтому вам придется добавить преобразование строк с русскими буквами из кодировки **Windows** в кодировку **DOS**.
Для этого включите в программу, после файла `iostream.h`, файл `windows.h` с описанием функций операционной системы **Windows**:
```
#include <windows.h>
```
Перед функцией `main()` создайте новую функцию с именем rus_str(), которая будет выполнять необходимое преобразование с помощью специальной функции **Windows**:
```
char* rus_str( char* str )
{
  CharToOem( str, str );
  return str;
}
```
Во всех строках программы, где на экран выдаются символьные строки с русскими буквами, укажите преобразование этих строк с помощью новой функции, например:
```
cout << rus_str( "Введите текущий год и нажмите ENTER.\n" );
```

2) ***После завершения работы окно программы закрывается и не удается увидеть результаты.***
Для исправления этого недостатка проще всего предусмотреть в конце программы ввод произвольного символа. Пока пользователь не нажмет какую-нибудь символьную клавишу и потом **Enter**, окно программы будет оставаться на экране. Для этого потребуется завести символьную переменную (строку с описанием этой переменной расположите после строки с описанием целочисленных переменных):
```
char wait_char;
```
Перед строкой с оператором возврата "`return 0`" добавьте оператор для ввода символа с клавиатуры:
```
cin >> wait_char;
```

Сравните результаты работы своей программы с примером из лекции. Поэкспериментируйте над улучшением или изменением формата вывода на экран.

Упражнение 2

Модифицируйте программу 5.1, чтобы при превышении переменной "`another_age`" значения **`150`** на экран выводилось сообщение:

```
Извините, но вы вряд ли доживете до [year] года!
```

Проверьте работу своей программы для нескольких разных лет.

Упражнение 3

Измените программу из упр.2 так, чтобы в ней учитывались и годы, и месяцы. На экран программа должна выводить следующие сообщения:

```
Введите текущий год и нажмите ENTER.
2000
Введите текущий месяц (число от 1 до 12).
10
```

```
Введите свой возраст (в годах).
25
Введите месяц своего рождения (число от 1 до 12).
5
Введите год, для которого вы хотите узнать свой возраст.
2006
Введите месяц этого года.
6
Ваш возраст в 6/2006: 31 год и 1 месяц.
```

Программа должна выдавать корректные сообщения для единственного и множественного числа лет и месяцев, т.е. должна выводить на экран "`25 лет и 1 месяц`", но "`24 года и 2 месяца`".

Подсказка: *В программе вам потребуются дополнительные переменные. Обязательно добавьте их имена в оператор описания переменных. При вычислениях могут пригодиться некоторые стандартные операции Си++:*

Символ	Операция	Пример	Значение
+	Сложение	3 + 5	8
-	Вычитание	43 - 25	18
*	Умножение	4 * 7	28
/	Деление	9/2	4
%	Остаток при делении нацело	20 % 6	2

(Обратите внимание, что в приведенной таблице операция деления "/" применялась к двум целым числам, поэтому результат – тоже целое число.)

Кроме арифметических операций, для проверки условий в операторе `if` *вам могут потребоваться некоторые логические операции.*

Символ	Операция	Пример	Значение
<	меньше, чем	3 < 5	TRUE (истина)
<=	меньше или равно	43 <= 25	FALSE (ложь)
>	больше, чем	4 > 7	FALSE
>=	больше или равно	9 >= 2	TRUE
==	равно	20 == 6	FALSE
!=	не равно	20 != 6	TRUE
&&	Логическое И	5 > 2 && 6 > 10	FALSE
||	Логическое ИЛИ	5 > 2 || 6 > 10	TRUE

ЛЕКЦИЯ 2. Переменные, типы данных и выражения

1. Идентификаторы

В исходном тексте программ на Си++ используется довольно много английских слов и их сокращений. Все слова (идентификаторы), встречающиеся в программах, можно разделить на три категории:

1) **Служебные слова языка.** Например, это слова `if`, `int` и `else`. Назначение этих слов предопределено и его нельзя изменить. Ниже приведен более полный список служебных слов:

```
asm     continue  float     new        signed     try
auto    default   for       operator   sizeof     typedef
break   delete    friend    private    static     union
case    do        goto      protected  struct     unsigned
catch   double    if        public     switch     virtual
char    else      inline    register   template   void
class   enum      int       return     this       volatile
const   extern    long      short      throw      while
```

По назначению эти слова можно разбить на отдельные группы (прил. 8.1).

2) **Библиотечные идентификаторы.** Назначение этих слов зависит от среды программирования. В случае серьезной необходимости программист может изменить их смысл. Примеры таких слов: `cin`, `cout` и `sqrt` (имя функции извлечения квадратного корня).

3) **Идентификаторы, введенные программистом.** Эти слова "создаются" программистом – например, имена переменных (такие, как `year_now` и `another_age` в программе 1.5.1).

Идентификатором не может быть произвольная последовательность символов. По правилам Си++, идентификатор начинается с буквы или символа подчеркивания ("_") и состоит только из английских букв, цифр и символов подчеркивания.

2. Типы данных

2.1 Целые числа

Правила Си++ требуют, чтобы в программе у всех переменных был задан тип данных. Тип данных `int` встречался нам уже неоднократно. Переменные этого типа применяются для хранения целых чисел (integer). Описание переменной, как имеющей тип `int`, сообщает компилятору, что он должен связать с идентификатором (именем) переменной количество памяти, достаточное для хранения целого числа во время выполнения программы.

Границы диапазона целых чисел, которые можно хранить в переменных типа `int`, зависят от конкретного компьютера. В Си++ есть еще два целочисленных типа – `short int` и `long int`. Они представляют, соответственно, более узкий и более широкий диапазон целых чисел, чем тип `int`. Добавление к любому из этих типов префикса `unsigned` означает, что в переменной будут хранится только неотрицательные числа. Например, описание:

```
unsigned short int year_now, age_now, another_year, another_age;
```

резервирует память для хранения четырех относительно небольших неотрицательных чисел.

Приведем несколько полезных правил, касающихся записи целочисленных значений в исходном тексте программ.

1) Нельзя пользоваться десятичной точкой. Значения 26 и 26.0 одинаковы, но "`26.0`" не является значением типа "`int`".
2) Нельзя пользоваться запятыми в качестве разделителей тысяч. Например, число 23,897 следует записывать как "`23897`".
3) Целые значения не должны начинаться с незначащего нуля. Он применяется для обозначения устаревших восьмеричных чисел, так что компилятор будет рассматривать значение "`011`" как число 9 в восьмеричной форме.

2.2 Вещественные числа

Для хранения вещественных чисел применяются типы данных `float` и `double`. Смысл знаков "+" и "–" для вещественных типов совпадает с целыми. Последние незначащие нули справа от десятичной точки игнорируются. Поэтому варианты записи "`+523.5`", "`523.5`" и "`523.500`" представляют одно и то же значение. В Си++ также допускается запись в формате *с плавающей запятой* (в экспоненциальном формате) в виде *мантиссы* и *порядка*. Например, 523.5 можно записать в виде "`5.235e+02`" (т.е. 5.235*10*10), а -0.0034 в виде "`-3.4e-03`".

В большинстве случаев используется тип `double`, он обеспечивает более высокую точность, чем `float`. Максимальную точность и наибольший диапазон чисел достигается с помощью типа `long double`, но он требует больше памяти (в **Visual C++** 10 байт на число), чем `double` (8 байт).

2.3 Преобразование типов в выражениях

При выполнении вычислений иногда бывает нужно гарантировать, что определенное значение будет рассматриваться как вещественное число, даже если на самом деле это целое. Чаще всего это нужно при делении в арифметических выражениях. Применительно к двум значениям типа `int` операция деления "`/`" означает деление нацело, например, `7/2` равно `3`. В данном случае, если необходимо получить результат `3.5`, то можно просто добавить десятичную точку в записи одного или обоих чисел: "`7.0/2`", "`7/2.0`" или "`7.0/2.0`". Но если и в числителе, и в знаменателе стоят переменные, а не константы, то указанный способ не подходит. Вместо него можно применить явное преобразование типа. Например, значение "`7`" преобразуется в значение типа `double` с помощью выражения "`double(7)`". Поэтому в выражении

```
answer = double(numerator) / denominator
```

операция "`/`" всегда будет рассматриваться компилятором как вещественное деление, даже если "`numerator`" и "`denumerator`" являются целыми числами. Для явного преобразования типов можно пользоваться и другими именами типов данных. Например, "`int(14.35)`" приведет к получению целого числа `14`.

2.4 Символьный тип

Для хранения символьных данных в Си++ предназначен тип "char". Переменная типа "char" рассчитана на хранение только одного символа (например, буквы или пробела). В памяти компьютера символы хранятся в виде целых чисел. Соответствие между символами и их кодами определяется таблицей кодировки, которая зависит от компьютера и операционной системы. Почти во всех таблицах кодировки есть прописные и строчные буквы английского алфавита, цифры 0,...,9, и некоторые специальные символы, например, #, £, !, +, - и др. Самой распространенной таблицей кодировки, скорее всего, является таблица символов ASCII.

В тексте программ символьные константы типа "char" надо заключать в одиночные кавычки, иначе компилятор поймет их неправильно и это может привести к ошибке компиляции, или, что еще хуже, к ошибкам времени выполнения. Например, "'A'" является символьной константой, но "A" будет рассматриваться компилятором в качестве имени переменной. Аналогично, "'9'" является символом, а "9" – целочисленной константой.

Т.к. в памяти компьютера символы хранятся в виде целых чисел, то тип "char" на самом деле является подмножеством типа "int". На Си++ разрешается использовать символы в арифметических выражениях. Например, на любом компьютере с таблицей ASCII следующее выражение даст истинное значение (TRUE, или 1):

```
'9'-'0' == 57-48 == 9
```

В таблице ASCII кодом символа '9' является десятичное число 57 (в шестнадцатеричной записи 0x39), а ASCII–код символа '0' равен десятичному числу 48 (шестнадцатеричное значение 0x30). Приведенное выражение можно переписать в виде:

57-48 == 0x39-0x30 == 9

Кодами ASCII удобнее пользоваться в шестнадцатеричной форме. При записи шестнадцатеричных чисел в Си++ применяется двухсимвольный префикс "0x".

Переменные типа "char" существенно отличаются от "int" при выполнении ввода данных с клавиатуры и вывода на экран. Рассмотрим следующую программу.

```
#include <iostream.h>

int main()
{
  int number;
  char character;

  cout << "Напечатайте символ и нажмите Enter:\n";
  cin >> character;

  number = character;

  cout << "Вы ввели символ '" << character;
  cout << "'.\n";
  cout << "В памяти компьютера он хранится в виде числа ";
  cout << number << ".\n";

  return 0;
}
```

Программа 2.1.

Программа 2.1 выдает на экран следующие сообщения:

```
Напечатайте символ и нажмите Enter:
9
Вы ввели символ '9'.
В памяти компьютера он хранится в виде числа 57.
```

Программу 2.1 можно изменить так, чтобы она печатала всю таблицу символов ASCII. Для этого придется применить "оператор цикла `for`". "Цикл `for`" является примером *оператора цикла* – эти операторы будут рассмотрены подробно в одной из следующих лекций. Оператор `for` имеет следующий синтаксис:

```
for (инициализация; условие_повторения; изменение_значений)
  {
  Оператор1;
  ...
  ...
  ОператорN;
  }
```

Цикл `for` выполняется в следующем порядке: (1) Сначала выполняется оператор *инициализации*. (2) Выполняется проверка, является ли *условие_повторения* истинным. Если условие ложно, то оператор `for` завершается. Если условие истинно, то последовательно выполняются операторы тела цикла *Оператор*`1...`*Оператор*`N`, и затем выполняется оператор *изменение_значений*. После этого происходит переход на начало шага (2).

Чтобы код символа вывести на экран в шестнадцатеричной форме, надо сначала послать на экран служебный символ-манипулятор. Программа для печати фрагмента таблицы ASCII (от 32-го символа (пробел) до 126-го (символ '~')), будет выглядеть так:

```
#include <iostream.h>

int main()
{
  int number;
  char character;

  for (number = 32; number <= 126; number = number + 1 )
    {
    character = number;
    cout << "Символ '" << character;
    cout << "' имеет код ";
    cout << dec << number << " (дес.) или ";
    cout << hex << number << " (шестнд.).\n";
    }

  return 0;
}
```

Программа 2.2.

Программа 2.2 напечатает на экране:

```
Символ ' ' имеет код 32 (дес.) или 20 (шестнд.).
Символ '!' имеет код 33 (дес.) или 21 (шестнд.).
...
...
Символ '}' имеет код 125 (дес.) или 7D (шестнд.).
Символ '~' имеет код 126 (дес.) или 7E (шестнд.).
```

2.5 Символьные строки

В большинстве рассмотренных примеров программ для вывода на экран часто используются символьные строки. В Си++ символьные строки заключаются в двойные кавычки. Поэтому в программах часто встречаются операторы вывода вроде:

```
cout << "' имеет код ";
```

На самом деле в Си++ строковый тип ("`string`") не является стандартным типом данных, таким, как, например, "`int`", "`float`" или "`char`". Строки хранятся в памяти в виде символьных массивов, поэтому строки будут рассматриваться позднее, при изучении массивов.

2.6 Типы данных, определяемые пользователем

Вопрос о типах данных, определяемых пользователем, будет обсуждаться намного более подробно в последующих лекциях. Будет показано, как программист может определить собственный тип данных, необходимый для решения конкретной задачи. Средства определения новых типов данных – одна из наиболее мощных возможностей Си++, которые позволяют хранить и обрабатывать в программах на Си++ сложные структуры данных.

3. Вывод вещественных чисел на экран

При выводе на экран численных значений типа "`float`", "`double`" или "`long double`" возможно указание точности представления данных на экране и задание некоторых дополнительных параметров отображения, например, отображение значений в формате с фиксированной или плавающей точкой.

В программе 3.1 вещественное число отображается в формате с фиксированной точкой и двумя десятичными знаками после запятой. Идентификатор "`sqrt`" является именем библиотечной функции извлечения квадратного корня. Описание библиотеки математических функций содержится в заголовочном файле "`math.h`".

```
#include <iostream.h>
#include <math.h>
int main()
{
  float number;
  cout << "Введите вещественное число.\n";
  cin >> number;

  cout << "Корень из ";

  cout.setf(ios::fixed);    // СТРОКА 12
  cout.precision(2);
  cout << number;

  cout << " примерно равен " << sqrt( number ) << ".\n";
  return 0;
}
```

Программа 3.1.

Программа 3.1 напечатает на экране:

```
Введите вещественное число.
200
Корень из 200.00 примерно равен 14.14.
```

Если СТРОКУ 12 заменить на "`cout.setf(ios::scientific);`", то вид результата изменится:

```
Введите вещественное число.
200
Корень из 2.00e+02 примерно равен 1.41e+01.
```

В выходные данные можно включить параметры табуляции. Для этого предназначена функция ширины поля, например, "`cout.width(20)`". Она задает ширину следующего выводимого на экран значения равной, как минимум, 20 символам (при меньшей ширине автоматически будут добавлены пробелы). Эта возможность особенно полезна для печати таблиц.

В компиляторе **Visual C++** при указании ширины поля по умолчанию предполагается, что значения выравниваются по правой границе. Чтобы задать выравнивание по левой границе поля, потребуется использовать еще несколько манипуляторов ввода-вывода. Это специальные функции и операторы, содержащиеся в библиотеке ввода/вывода Си++. Они описаны в заголовочном файле `iomanip.h`. Для задания выравнивания по левой границе надо установить специальный флажок (переключатель) с помощью функции `setiosflags`:

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
  int number;

  cout << setiosflags( ios::left );
  cout.width(20);
  cout << "Число" << "Квадратный корень\n\n";

  cout.setf( ios::fixed );
  cout.precision(2);
  for ( number = 1 ; number <= 10 ; number = number + 1 )
    {
    cout.width(20);
    cout << number << sqrt(number) << "\n";
    }

  return 0;
}
```

Программа 3.2.

Программа 3.2 выдаст на экран следующие сообщения:

```
Число               Квадратный корень
1                   1.00
2                   1.41
3                   1.73
4                   2.00
5                   2.24
6                   2.45
```

```
7                   2.65
8                   2.83
9                   3.00
10                  3.16
```

(*ПРИМЕЧАНИЕ*: во всех примерах идентификатор "`cout`" является именем переменной-объекта класса "`stream`" (поток). Функции "`setf(...)`", "`precision(...)`" и "`width(...)`" являются функциями-членами класса "`stream`". Понятия "объект", "класс", "функция-член" и др. будут подробно рассматриваться в курсе объектно-ориентированного программирования.)

4. Описания, константы и перечисления

Как вы уже знаете, в программах на Си++ переменные обязательно должны быть описаны до первого использования, например, так:

```
float number;
```

После оператора описания до момента выполнения первого оператора присваивания значение переменной "`number`" будет неопределенным, т.е. эта переменная может иметь случайное значение. В Си++ можно (и желательно) инициализировать переменные конкретными значениями непосредственно при описании переменных. Например, возможен следующий оператор описания с инициализацией:

```
float PI = 3.1416;
```

Если значение переменной в программе никогда не изменяется, то ее целесообразно защитить от случайного изменения с помощью служебного слова "`const`" – т.е., превратить в константу.

4.1 Тип "Перечисление"

Для описания набора связанных по смыслу констант типа "`int`" в Си++ есть оператор перечисления. Например, описание вида

```
enum { MON, TUES, WED, THURS, FRI, SAT, SUN };
```

эквивалентно описанию 7 констант-кодов дней недели:

```
const int MON = 0;
const int TUES = 1;
const int WED = 2;
const int THURS = 3;
const int FRI = 4;
const int SAT = 5;
const int SUN = 6;
```

По умолчанию членам перечисления "`enum`" присваиваются значения 0, 1, 2, и т.д.. При необходимости члены перечисления можно инициализировать другими значениями. Неинициализированным явно членам будут присвоены значения по порядку, начиная от предыдущего проинициализированного члена:

```
enum { MON = 1, TUES, WED, THURS, FRI, SAT = -1, SUN };
```

В приведенном примере "`FRI`" имеет значение 5, а "`SUN`" – значение 0.

4.2 Расположение описаний констант и переменных в исходном тексте

В исходном тексте описания констант чаще всего размещаются в заголовке программы перед функцией "`main`". После них, уже в теле функции "`main`", размещаются описания переменных. Для иллюстрации этого порядка ниже приведен фрагмент программы, которая рисует на экране окружность заданного радиуса и вычисляет ее длину (набирать этот пример не надо, поскольку он приведен не полностью.)

```
#include <iostream.h>

const float PI = 3.1416;
const float SCREEN_WIDTH = 317.24;

int drawCircle(float diameter); /* Это "прототип функции" */

int main()
{
  float radius = 0;

  cout << "Введите радиус окружности.\n";
  cin >> radius;

  drawCircle(radius*2);

  cout.setf(ios::fixed);
  cout.precision(2);
  cout << "Длина окружности радиуса " << radius;
  cout << " примерно равна " << 2*PI*radius << ".\n";
  return 0;
}

int drawCircle(float diameter)
{
  float radius = 0;

  if (diameter > SCREEN_WIDTH)
    radius = SCREEN_WIDTH/2.0;
  else
    radius = diameter/2.0;
  ...
  ...
}
```

После определения функции "`main()`" в этой программе содержится определение функции рисования окружности "`drawCircle(...)`". Детали реализации этой функции сейчас не существенны (будем считать, что функция `drawCircle(...)`" реализована корректно и ею можно пользоваться так же, как, например, функцией "`sqrt(...)`"). Обратите внимание, что, хотя переменная "`radius`" используется в обеих функциях "`main()`" и "`drawCircle(...)`", это не одна и та же переменная, а две *разных*.

Если бы переменная "`radius`" была описана до функции "`main`", то в таком случае она была бы *глобальной переменной* (общедоступной). Тогда, предполагая, что внутри функции "`drawCircle(...)`" описания переменной уже нет, если "`drawCircle(...)`" присвоит глобальной переменной значение "`SCREEN_WIDTH/2.0`", то это значение чуть позже функция "`main()`" использует для вычисления длины окружности и получится неверный результат.

В приведенной программе глобальной переменной нет, а есть две *локальных* переменных "`radius`". Например, первая переменная "`radius`" является *локальной* пе-

ременной функции "`main()`", или, говорят, что функция "`main()`" является *областью видимости* этой переменной.

Константы общего назначения, такие, как "`PI`" и "`SCREEN_WIDTH`", принято описывать *глобально*, чтобы они были доступны внутри любой функции.

Для контроля действий программы в приведенном фрагменте предусмотрено повторное отображение данных, введенных пользователем. Другими словами, заданное пользователем значение "`radius`" еще раз печатается на экране перед отображением длины окружности.

5. Присваивание и выражения

5.1 Краткая форма записи операторов присваивания

В программах часто встречаются операторы присваивания, в которых справа стоит выражение, модифицирующее текущее значение переменной, например:

```
number = number + 1;
```

Переменным часто присваиваются значения, вычисленные на основе их старых значений. Поэтому в Си++ была введена краткая форма записи для подобных операторов присваивания. Любую из операций "`+`" (сложение), "`-`" (вычитание), "`*`" (умножение), "`/`" (деление) и "`%`" (остаток от деления нацело) можно указать в качестве префикса оператора присваивания ("`=`") (см. следующую таблицу).

Пример:	**Эквивалентное выражение:**
`number += 1;`	`number = number + 1;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rush_factor;`	`time = time / rush_factor;`
`change %= 100;`	`change = change % 100;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`

Первый пример допускает еще более краткую запись с помощью оператора инкремента "`++`":

```
number++;
```

Оператор "`++`" существует и в префиксной форме:

```
++number;
```

Постфиксная и префиксная форма записи имеют важное различие, которое необходимо помнить. Префиксный оператор применяется **ДО** вычисления остальной части выражения, а постфиксный – **ПОСЛЕ**. Например, посоле выполнения операторов

```
x = 4;
y = x++;
```

переменная "`x`" получит значение 5, а "`y`" – значение 4. В случае операторов

```
x = 4;
y = ++x;
```

обе переменные получат значение 5. Это объясняется тем, что "`++x`" выполняется до того, как значение "`x`" будет использовано в выражении, а "`x++`" – после. В Си++ существует аналогичный оператор декремента "`--`", уменьшающий значение переменной на 1, и у него тоже есть префиксная и постфиксная форма.

Вообще, выражение с оператором присваивания имеет значение, равное значению левой части после выполнения присваивания. Ниже приведено выражение, соответствующее правилам Си++, которое можно использовать для проверки условия:

```
(y = ++x) == 5
```

Это выражение означает следующее: "после присвоения переменной y инкрементированного значения x проверить, не равно ли значение y числу 5".

5.2 Логические выражения и операторы

Интуитивно логические выражения наподобие "2<7", "1.2!=3.7" и "6>=9" воспринимаются человеком как утверждения, которые могут быть "истинными (true)" или "ложными (false)" (операция "!=" означает "не равно"). Допускается объединение нескольких подобных выражений в более сложное выражение с помощью логических операций "&&" ("И"), "||" ("ИЛИ") и "!" ("НЕ") (см. таблицу).

Выражение:	**Истинно или ложно:**
`(6 <= 6) && (5 < 3)`	`false`
`(6 <= 6) || (5 < 3)`	`true`
`(5 != 6)`	`true`
`(5 < 3) && (6 <= 6) || (5 != 6)`	`true`
`(5 < 3) && ((6 <= 6) || (5 != 6))`	`false`
`!((5 < 3) && ((6 <= 6) || (5 != 6)))`	`true`

В таблице в четвертом примере выражение истинно, поскольку приоритет операции "&&" выше, чем у "||". Приоритет (порядок выполнения) различных операций Си++ можно узнать в учебнике или руководстве по языку Си++, а также в справочной системе **Visual C++** (тема *Operator Precedence*). Если у вас возникают сомнения относительно приоритета операций, применяйте круглые скобки (). Применение этих скобок облегчает чтение программ.

Составные логические выражения обычно применяются в качестве условий в операторах if и в циклах for. Например:

```
...
...
if ( total_test_score >= 50 && total_test_score < 65 )
  cout << "Вы прошли тест со средним результатом.\n";
...
...
```

У логических выражений в Си++ есть еще одно важное свойство. В Си++ истинное значение ("true") представляется в виде целого числа 1 (большинство компиляторов любое положительное число считают истинным значением), а ложное значение ("false") в виде значения 0. Это может привести к ошибкам. Например, легко напечатать "=" вместо "==". Поэтому фрагмент

```
...
...
if ( number_of_people = 1 )
  cout << "Есть только один человек.\n";
...
...
```

всегда будет печатать сообщение "Есть только один человек", даже если до оператора if переменная "number_of_people" была больше 1.

6. Сводка результатов

В данной лекции довольно подробно рассматривались переменные языка Си++. У переменных всегда есть определенный тип данных. Переменные применяются для временного или постоянного хранения значений разных типов. Значения переменным можно присваивать различными способами. В выражениях для вычисления новых значений переменных можно использовать различные арифметические и логические операции.

7. Упражнения

Упражнение 1

Для преобразования температуры из шкалы Цельсия в абсолютную шкалу температур (шкалу Кельвина) надо добавить к температуре по Цельсию значение 273.15. В шкалу Фаренгейта температура по Цельсию преобразуется $t_f = 1.8t^\circ + 32$.

Напишите программу преобразования значений температуры, которая будет печатать на экране следующую таблицу:

```
Цельсий   Фаренгейт   Абсолютная температура
      0       32.00                   273.15
     20       68.00                   293.15
     40      104.00                   313.15
    ...         ...                      ...
    ...         ...                      ...
    300      572.00                   573.15
```

Упражнение 2

Измените программу из упражнения 1 так, чтобы она запрашивала у пользователя минимальную и максимальную температуру по Цельсию, которые должны быть в первой и последней строках таблицы. Программа также должна запросить шаг изменения температуры (на это значение должны отличаться температуры в соседних строках таблицы, в упражнении 1 шаг был равен 20-ти градусам).

Перед таблицей программа должна вывести несколько строк с пояснением своих действий, а также повторить вывод на экран введенных пользователем данных.

Упражнение 3

Напишите программу, которая считывает с клавиатуры символ (ch) и затем выводит одно из следующих сообщений (вместо ch должен выводиться введенный символ, а вместо ... – соответствующая прописная или строчная буква):

а) если символ ch является строчной буквой – сообщение "Букве ch соответствует прописная буква ...",

б) если ch является прописной буквой – сообщение "Букве ch соответствует строчная буква ...",

в) если ch не является буквой – сообщение "Символ ch не является буквой".

Для составления необходимых условий обратитесь к расширенной таблице символов ASCII (см. п.8.3).

Упражнение 4

Напишите программу для возведения произвольного числа `x` в положительную степень `n` с помощью цикла `for`. (Есть ли способы повышения эффективности вашей программы?)

8. Приложения

8.1 Служебные слова Си++

По назначению служебные слова языка Си++ можно разделить на несколько групп. Ниже перечислены эти группы и относящиеся к ним слова. **Полужирным** шрифтом выделены слова, назначение которых вы узнаете в данном вводном курсе.

- Типы данных (определяют типы данных, которые можно хранить в памяти компьютера).

```
char       short      int        long     (целые числа)
enum                                       (тип "перечисление")
double     float                           (вещественные числа)
void
struct     union      typedef              (типы, определяемые
                                           пользователем)
```

- Модификаторы типов данных (позволяют задать некоторые свойства хранения данных в памяти).

```
signed     unsigned
volatile   register
const      static     extern     auto
```

- Управление порядком выполнения операторов.

```
if         else                  (ветвление с двумя вариантами)
switch     case       default    (множественное ветвление)
for        while      do         (циклы)
break      continue
return                           (возврат из функции)
goto                             (безусловный переход)
```

- Динамическое распределение памяти.

```
new        delete
```

- Объектно-ориентированное программирование (эти слова будут подробно рассматриваться в отдельном курсе объектно-ориентированного программирования и проектирования).

```
class      private    protected  public
virtual    this       friend     template
operator
```

- Обработка исключений (особый механизм обработки ошибок в объектно-ориентированных программах).

```
try        throw      catch
```

- Разное.

```
sizeof     inline     asm
```

8.2 Таблица символов ASCII

0 NUL	16 DLE	32	48 0	64 @	80 P	96 `	112 p
1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
2 STX	18 DC2	34 "	50 2	66 B	82 R	98 b	114 r
3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
4 EOT	20 DC4	36 $	52 4	68 D	84 T	100 d	116 t
5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
7 BEL	23 ETB	39 '	55 7	71 G	87 W	103 g	119 w
8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
9 TAB	25 EM	41)	57 9	73 I	89 Y	105 i	121 y
10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124 |
13 CR	29 GS	45 -	61 =	77 M	93]	109 m	125 }
14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
15 SI	31 US	47 /	63 ?	79 O	95 _	111 o	127 ⌂

8.3 Расширенная таблица символов ASCII для кодовой страницы DOS-866

128 А	144 Р	160 а	176 ░	192 └	208 ╨	224 р	240 Ё
129 Б	145 С	161 б	177 ▒	193 ┴	209 ╤	225 с	241 ё
130 В	146 Т	162 в	178 ▓	194 ┬	210 ╥	226 т	242 Є
131 Г	147 У	163 г	179 │	195 ├	211 ╙	227 у	243 є
132 Д	148 Ф	164 д	180 ┤	196 ─	212 ╘	228 ф	244 Ї
133 Е	149 Х	165 е	181 ╡	197 ┼	213 ╒	229 х	245 ї
134 Ж	150 Ц	166 ж	182 ╢	198 ╞	214 ╓	230 ц	246 Ў
135 З	151 Ч	167 з	183 ╖	199 ╟	215 ╫	231 ч	247 ў
136 И	152 Ш	168 и	184 ╕	200 ╚	216 ╪	232 ш	248 °
137 Й	153 Щ	169 й	185 ╣	201 ╔	217 ┘	233 щ	249 •
138 К	154 Ъ	170 к	186 ║	202 ╩	218 ┌	234 ъ	250 ·
139 Л	155 Ы	171 л	187 ╗	203 ╦	219 █	235 ы	251 √
140 М	156 Ь	172 м	188 ╝	204 ╠	220 ▄	236 ь	252 №
141 Н	157 Э	173 н	189 ╜	205 ═	221 ▌	237 э	253 ¤
142 О	158 Ю	174 о	190 ╛	206 ╬	222 ▐	238 ю	254 ■
143 П	159 Я	175 п	191 ┐	207 ╧	223 ▀	239 я	255

ЛЕКЦИЯ 3. Функции и процедурная абстракция

1. Назначение подпрограмм

Естественный и интуитивно понятный подход к решению больших сложных задач состоит в том, чтобы разбить большую задачу на набор меньших, которые можно решить более или менее независимо и затем скомбинировать полученные решения для получения полного решения. На таком подходе основана методология структурного программирования, которое господствовало в разработке программного обеспечения до появления объектно-ориентированного подхода.

При структурном программировании большая программа разделяется на набор более или менее независимых *подпрограмм*. В Си++ подпрограммы называются *функциями* (в Паскале и некоторых других языках программирования есть два типа подпрограмм – "процедуры" и "функции").

Подпрограммы уже неоднократно встречались в предыдущих лекциях. Например, в программе 2.3.2 для построения таблицы квадратных корней был применен следующий цикл `for`:

```
...
#include<math.h>
...
...
  for ( number=1 ; number<=10 ; number=number+1 )
  {
    cout.width(20);
    cout << number << sqrt(number) << "\n";
  }
...
```

Функция "`sqrt(...)`" – это подпрограмма, описание которой хранится в заголовочном файле "`math.h`", а реализация – в библиотечном файле "`math.lib`". При вызове функции "`sqrt(...)`" ей передается числовой параметр "`number`", функция применяет алгоритм вычисления квадратного корня из этого числа, и затем возвращает вычисленное значение обратно в место вызова. Для применения этой функции программисту совсем необязательно знать, какой именно алгоритм реализован внутри нее. Главное, чтобы функция гарантированно возвращала верный результат. Было бы довольно нелепо включать в явном виде алгоритм извлечения квадратного корня (и, возможно, делать это неоднократно) в главную функцию программы "`main`".

В данной лекции описывается, как программист может определять свои собственные функции. Сначала предполагается, что эти функции размещаются в одном файле с функцией "`main`". В конце лекции показывается, как распределять функции программы по нескольким файлам.

2. Определение новых функций

Простым примером определения и использования новой функции является программа 2.1 (в ней пользовательская функция называется "`area(...)`"). Эта программа вычисляет площадь прямоугольника заданной длины и ширины.

```
#include<iostream.h>

int area(int length, int width);     /* Описание функции */
```

```
// ГЛАВНАЯ ФУНКЦИЯ:
int main()
{
  int this_length, this_width;

  cout << "Введите длину: ";         /* <--- строка 10 */
  cin >> this_length;
  cout << "Введите ширину: ";
  cin >> this_width;
  cout << "\n";                      /* <--- строка 14 */

  cout << "Площадь прямоугольника с размерами ";
  cout << this_length << "x" << this_width;
  cout << " равна " << area(this_length, this_width) << "\n";

  return 0;
}
// КОНЕЦ ГЛАВНОЙ ФУНКЦИИ

// ФУНКЦИЯ ВЫЧИСЛЕНИЯ ПЛОЩАДИ:
int area(int length, int width) /* Начало определения функции */
{
  int number;

  number = length * width;
  return number;
} /* Конец определения функции */
// КОНЕЦ ФУНКЦИИ
```

Программа 2.1.

Программа 2.1, конечно, допускает запись в более сжатой форме, но в данном виде она служит хорошей иллюстрацией некоторых свойств функций Си++:

- Структура определения (реализации) функции подобна структуре функции "`main()`" – в теле функции есть описания локальных переменных и исполняемые операторы.
- У функции могут быть параметры, которые указываются в списке внутри круглых скобок после имени функции. У каждого параметра задается тип.
- Если вызов функции встречается ранее ее определения, то в начале программы должно содержаться описание функции (прототип). Прототип функции описывает ее параметры и тип возвращаемого значения. Обычно прототипы функций размещаются после описания глобальных констант.

Внутри функции может быть несколько операторов возврата "`return`". Функция завершается после выполнения любого оператора "`return`". Например:

```
double absolute_value(double number)
{
  if (number >= 0)
    return number;
  else
    return -number;
}
```

3. Способы передачи параметров внутрь функций

Во всех рассмотренных до сих пор примерах параметры функций передавались *по значению*. При вызове из функции "`main()`" вызываемой функции передаются *копии* указанных переменных. Например, в программе 2.1 функции "`area(...)`" передаются текущие значения переменных "`this_length`" и "`this_width`". Затем функция "`area(...)`" сохраняет переданные значения в собственных локальных переменных, и именно эти переменные участвуют в последующих вычислениях внутри функции.

3.1 Передача параметров по значению

Функции, принимающие параметры *по значению*, "безопасны" в том смысле, что они не могут случайно изменить переменные вызывающей функции (т.е. у функций нет скрытых *побочных эффектов*). Большинство функций проектируются именно таким образом.

Программа 3.1 поясняет, почему важно гарантировать "сохранность" переменных вызывающей функции. Эта программа должна выводить на экран факториал и корень из числа, введенного пользователем:

```
Введите положительное число:
4
Факториал 4! равен 24, а квадратный корень из 4 равен 2.
```

Для извлечения квадратного корня применяется библиотечная функция "`sqrt(...)`". Библиотечной функции для вычисления факториала нет, поэтому придется написать собственную функцию (вычисляющую для любого положительного целого числа `n` значение `n!=(1*2*3*...*n)`).

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

int factorial(int number);

// ГЛАВНАЯ ФУНКЦИЯ:
int main()
{
  int whole_number;

  cout << "Введите положительное число:\n";
  cin >> whole_number;
  cout << "Факториал " << whole_number << "! равен ";
  cout << factorial(whole_number);
  cout << ", а квадратный корень из " << whole_number;
  cout << " равен " << sqrt(whole_number) << ".\n";

  return 0;
}
// КОНЕЦ ГЛАВНОЙ ФУНКЦИИ

// ФУНКЦИЯ ДЛЯ ВЫЧИСЛЕНИЯ ФАКТОРИАЛА:
int factorial(int number)
{
  int product = 1;

  for ( ; number > 0 ; number--)
    product *= number;
```

```
  return product;
}
// КОНЕЦ ФУНКЦИИ
```

Программа 3.1.

Если бы функция "`factorial(...)`" изменяла переменную вызывающей функции, то программа 3.1 выдавала бы следующий ответ (формально правильный, но по смыслу задачи некорректный):

```
Введите положительное число:
4
Факториал 4! равен 24, а квадратный корень из 0 равен 0.
```

3.2 Передача параметров по ссылке

Все-таки иногда бывает необходимо, чтобы функция изменила значение переданного ей параметра. Рассмотрим программу 2.1. С 10-й по 14-ю строку в ней выполняется запрос размеров прямоугольника, а затем вычисляется его площадь.

При структурном программировании независимые по смыслу части программы принято оформлять в виде отдельных функций. Для получения данных от пользователя создадим функцию "`get_dimensions`". В данном случае необходимо, чтобы эта функция изменяла значения переменных "`this_length`" и "`this_width`" (переданных ей в качестве параметров) – помещала в них значения, введенные пользователем с клавиатуры. Изменение параметров функции возможно при передаче параметров *по ссылке*. У таких параметров в заголовке функции после имени типа указывается символ "`&`".

```
#include<iostream.h>

int area( int length, int width );
void get_dimensions( int& length, int& width );

// ГЛАВНАЯ ФУНКЦИЯ:
int main()
{
  int this_length, this_width;

  get_dimensions( this_length, this_width );
  cout << "Площадь прямоугольника с размерами ";
  cout << this_length << "x" << this_width;
  cout << " равна " << area( this_length, this_width ) << "\n";

  return 0;
}
// КОНЕЦ ГЛАВНОЙ ФУНКЦИИ

// ФУНКЦИЯ ВВОДА РАЗМЕРОВ ПРЯМОУГОЛЬНИКА:
void get_dimensions( int& length, int& width )
{
  cout << "Введите длину: ";
  cin >> length;
  cout << "Введите ширину: ";
  cin >> width;
  cout << "\n";
```

```
}
// КОНЕЦ ФУНКЦИИ

// ФУНКЦИЯ ВЫЧИСЛЕНИЯ ПЛОЩАДИ:
int area( int length, int width )
{
  return length*width;
}
// КОНЕЦ ФУНКЦИИ
```

Программа 3.2.

Функция "`get_dimensions`" изменяет значения параметров "`this_length`" и "`this_width`", но не возвращает никакого значения (т.е. не является "функцией" в математическом смысле). Этот факт отражается и в прототипе, и в определении функции – в качестве возвращаемого значения указан тип "`void`" ("пустой" тип).

4. Полиморфизм и перегрузка функций

Одним из характерных свойств объектно-ориентированного языка, в том числе и Си++, является *полиморфизм* – использование одного имени для выполнения различных действий над различными объектами. Применительно к функциям это называется *перегрузкой*. Для основных операций Си++ перегрузка уже встроена в язык: например, у сложения существует только одно имя, "+", но его можно применять для сложения как целых, так и вещественных значений. Эта идея расширяется на обработку операций, определенных пользователем, т.е., функций.

Перегруженные функции имеют одинаковые имена, но разные списки параметров и возвращаемые значения (см. программу 4.1).

```
#include<iostream.h>

int average( int first_number, int second_number,
             int third_number );
int average( int first_number, int second_number );

// ГЛАВНАЯ ФУНКЦИЯ:
int main()
{
  int number_A = 5, number_B = 3, number_C = 10;

  cout << "Целочисленное среднее чисел " << number_A << " и ";
  cout << number_B << " равно ";
  cout << average(number_A, number_B) << ".\n\n";

  cout << "Целочисленное среднее чисел " << number_A << ", ";
  cout << number_B << " и " << number_C << " равно ";
  cout << average(number_A, number_B, number_C) << ".\n";

  return 0;
}
// КОНЕЦ ГЛАВНОЙ ФУНКЦИИ

// ФУНКЦИЯ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 3-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number,
             int third_number )
```

```
{
  return ((first_number + second_number + third_number)/3);
}
// КОНЕЦ ФУНКЦИИ

// ФУНКЦИЯ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 2-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number )
{
  return ((first_number + second_number)/2);
}
// КОНЕЦ ФУНКЦИИ
```

Программа 4.1.

Программа 4.1. выводит на экран сообщения:

```
Целочисленное среднее чисел 5 и 3 равно 4.
Целочисленное среднее чисел 5, 3 и 10 равно 6.
```

5. Процедурная абстракция и "хороший" стиль программирования

Функции помогают применять для разработки программ структурный метод проектирования "*сверху вниз*". При этом решаемая задача делится на подзадачи (и затем на под-подзадачи и т.д.). Для решения каждой подзадачи программист реализует отдельную функцию, при этом ему не нужно знать детали реализации остальных функций.

Чтобы функцией мог воспользоваться другой программист, она должна иметь осмысленное имя и комментарий с описанием назначения функции, ее параметров и возможных возвращаемых значений.

Опытные программисты на начальных этапах разработки часто применяют пустые функции (*заглушки*), которые содержат только оператор возврата значения соответствующего типа. Эти функций облегчают отладку главной функции или просто функции более высокого уровня.

Выделение в решаемой задаче функций методом "сверху вниз" часто называется *функциональной* или *процедурной абстракцией*. При проектировании независимых друг от друга функций широко применяется передача параметров по значению и локальные переменные внутри функций. После реализации программист может рассматривать подобные функции как "черные ящики". Для их использования знать детали реализации не обязательно.

6. Модульное программирование

Помимо метода "сверху вниз", вторым важным методом структурного проектирования является метод модульного программирования. Он предполагает разделение текста программы на несколько файлов, в каждом из которых сосредоточены независимые части программы (сгруппированные по смыслу функции).

В программах на Си++ часто применяются библиотечные функции (например, "`sqrt(...)`"). Для использования большинства функций, в том числе и библиотечных, необходимы два файла (в скобках примеры даны для "`sqrt(...)`"):

- *Заголовочный файл* ("`math.h`") с прототипом функции ("`sqrt(...)`" и многих других математических функций). Поэтому в программах, вызывающих

"sqrt(...)", есть строка "#include <math.h>", а не явное объявление этой функции.

- *Файл реализации* (для пользовательских функций это файлы с исходным текстом на Си++, а библиотечные функции обычно хранятся в скомпилированном виде в специальных библиотечных файлах, например, "libcmtd.lib"). Файлы реализации пользовательских функций (обычно с расширением ".cpp") содержат определения этих функций.

Разделение исходного текста на заголовочные файлы и файлы реализации показано в программе 6.1, которая выполняет те же действия, что и программа 4.1. Теперь программа состоит из трех файлов: главного файла, заголовочного файла с описаниями двух функций расчета среднего значения, и соответствующего файла реализации.

В главном файле содержится следующий текст:

```
#include <iostream.h>
#include "averages.h"

int main()
{
  int number_A = 5, number_B = 3, number_C = 10;

  cout << "Целочисленное среднее чисел " << number_A << " и ";
  cout << number_B << " равно ";
  cout << average(number_A, number_B) << ".\n\n";

  cout << "Целочисленное среднее чисел " << number_A << ", ";
  cout << number_B << " и " << number_C << " равно ";
  cout << average(number_A, number_B, number_C) << ".\n";

  return 0;
}
```

Главный файл программы 6.1.

Обратите внимание, что имя файла стандартной библиотеки "iostream.h" в директиве препроцессора "include" заключено в угловые скобки ("<>"). Файлы с именами в угловых скобках препроцессор ищет в библиотечных каталогах, указанных в настройках компилятора. Имена пользовательских заголовочных файлов обычно заключаются в двойные кавычки, и препроцессор ищет их в текущем каталоге программы.

Далее приведено содержимое файла "averages.h". В нем есть идентификатор препроцессора "AVERAGES_H" и служебные слова препроцессора "ifndef" ("если не определено"), "define" ("определить") и "endif" ("конец директивы if"). Идентификатор "AVERAGES_H" является глобальным символическим именем заголовочного файла. Первые две строки файла служат защитой от повторной обработки текста заголовочного файла препроцессором, на случай, если в исходном тексте программы строка "#include "averages.h"" встречается несколько раз.

В заголовочных файлах, кроме прототипов функций, часто размещаются описания глобальных констант и пользовательских типов. Подробнее об этом говорится в курсе объектно-ориентированного программирования.

```
#ifndef AVERAGES_H
#  define AVERAGES_H
```

```
// (Определения констант и пользовательских типов)

// ПРОТОТИП ФУНКЦИИ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 3-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number,
             int third_number );
// ПРОТОТИП ФУНКЦИИ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 2-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number );

#endif
```

Заголовочный файл averages.h.

Ниже показано содержимое файла "`averages.cpp`" с исходным текстом пользовательских функций:

```
#include <iostream.h>
#include "averages.h"

// ФУНКЦИЯ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 3-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number,
             int third_number )
{
  return ((first_number + second_number + third_number)/3);
}

// ФУНКЦИЯ ДЛЯ ВЫЧИСЛЕНИЯ ЦЕЛОЧИСЛЕННОГО СРЕДНЕГО
// ЗНАЧЕНИЯ 2-Х ЦЕЛЫХ ЧИСЕЛ:
int average( int first_number, int second_number )
{
  return ((first_number + second_number)/2);
}
```

Файл реализации averages.cpp.

Программа 6.1 демонстрирует основное достоинство модульного подхода: при изменении деталей реализации в файле "`averages.cpp`" не обязательно вносить изменения в файл "`averages.h`" или в главный файл программы.

7. Сводка результатов

В данной лекции описано, как в Си++ можно создавать новые функции. Есть два способа передачи параметров внутрь функции: по значению и по ссылке. Функции облегчают применение процедурной абстракции при разработке программ методом "сверху вниз". При модульном подходе описание и реализация функций размещаются в отдельных файлах, в таком случае для вызова функции необходимо включать в текст программы заголовочный файл.

8. Упражнения

Упражнение 1

В программе из упражнения 2 лекции 2 (файл ex2_2.cpp) выделите 6 функций, имена и назначение которых перечислены ниже:

`fahrenheit_of`
: Возвращает значение температуры по шкале Фаренгейта для переданного значения по шкале Цельсия.

`absolute_value_of`
: Возвращает значение температуры в абсолютной шкале для переданного значения по шкале Цельсия.

`print_preliminary_message`
: Печать сообщения, поясняющего назначение программы.

`input_table_specifications`
: Запрос параметров таблицы с клавиатуры.

`print_message_echoing_input`
: Повторное отображение параметров, введенных пользователем.

`print_table`
: Печать таблицы температур.

Проверьте свою программу для различных исходных данных. В качестве контрольного примера можете использовать следующие выходные данные:

```
Эта программа печатает значения температур в разных шкалах.

Введите минимальное (целое) значение температуры
по Цельсию, которое будет в первой строке таблицы: 0
Введите максимальное значение температуры: 100
Введите разницу температур между соседними строками таблицы: 20

Преобразование значений температуры от 0 градусов Цельсия
до 100 градусов Цельсия, с шагом 20 градусов:

Цельсий   Фаренгейт   Абсолютная температура

      0       32.00                   273.15
     20       68.00                   293.15
     40      104.00                   313.15
    ...         ...                      ...
    ...         ...                      ...
    100      212.00                   485.15
```

Упражнение 2

Разделите программу из упражнения 1 на три файла:

1) главный файл программы;
2) заголовочный файл "`conversions.h`" с прототипами функций "`fahrenheit_of(...)`" и "`absolute_value_of(...)`";
3) файл реализации с определением этих двух функций.

Снова проверьте свою программу для различных исходных данных.

Упражнение 3

(а) Создайте заголовочный файл "`statistics.h`" и соответствующий файл реализации "`statistics.cpp`" с функциями "`average(...)`" и "`standard_deviation(...)`". Эти функции должны вычислять среднее значение и среднеквадратическое отклонение для последовательности из 1, 2, 3 или 4 вещественных чисел. Среднеквадратическое отклонение чисел r_1, ..., r_N определяется как корень из среднего значения квадратов отклонений чисел от своего среднего:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(r_i - m)^2}\ , \text{где}\ m = \frac{1}{N}\sum_{i=1}^{N} r_i$$

Подсказки: *(1) Примените средства перегрузки функций Си++. (2) Функции можно вызывать изнутри друг друга. (3) Максимально используйте возможности текстового редактора по копированию фрагментов исходного текста.*

(б) Напишите тестовую программу для проверки функций из файла "`statistics.h`", которая в цикле запрашивает исходные данные до тех пор, пока пользователь не сообщит о завершении работы (некоторым специально оговоренным числом). Ваша тестовая программа должна выдавать на экран сообщения, подобные приведенным ниже:

```
Эта программа предназначена для тестирования функций из
заголовочного файла "statistics.h".

Сколько чисел будет в тестовой последовательности – 1, 2, 3
или 4? (для завершения работы введите 0): 3

Введите первое число: 5
Введите второе число: 7
Введите третье число: 9
Среднее значение: 7. Среднеквадратическое отклонение: 1.63299.

Сколько чисел будет в тестовой последовательности – 1, 2, 3
или 4? (для завершения работы введите 0): 1

Введите первое число: 5.8
Среднее значение: 5.8. Среднеквадратическое отклонение: 0.

Сколько чисел будет в тестовой последовательности – 1, 2, 3
или 4? (для завершения работы введите 0): 8

Извините, но эта программа может работать только с 1, 2, 3
или 4-мя числами.

Сколько чисел будет в тестовой последовательности – 1, 2, 3
или 4? (для завершения работы введите 0): 0

Программа тестирования функций из заголовочного файла
"statistics.h" завершила работу.
```

Подсказки: *(1) Разрабатывайте свою программу методом "сверху вниз ". Начните с написания короткой главной функции, в которой вызываются функции-заглушки, например, "*`test_three_values()`*". Детали этих функций вы уточните позже, после отладки функции "*`main()`*". (2) В качестве высокоуровневой структуры программы вы можете использовать цикл* `for` *с пустым разделом инициализации и пустым оператором изменения значений (эквивалент цикла* `while`*, который будет рассматриваться в следующих лекциях).*

ЛЕКЦИЯ 4. Текстовые файлы и потоки ввода/вывода

1. Назначение файлов

Во всех рассматривавшихся до сих пор программах ввод данных производился только с клавиатуры, а вывод – только на экран. Если в качестве устройств ввода/вывода ограничиться только клавиатурой и экраном, то в таком случае будет сложно обработать большие объемы входных данных. Выходные данные, отображенные на экране, после выключения компьютера безвозвратно теряются.

Для устранения подобных проблем удобно сохранять данные на запоминающих устройствах, предназначенных для долговременного хранения данных (обычно это магнитные диски). Данные, сгенерированные с помощью одной программы, можно сохранить на диске и в дальнейшем, при необходимости, извлечь и обработать в другой программе.

На дисках данные хранятся в виде структур данных, обслуживаемых операционной системой, – в виде *файлов*. Файл проще всего представить как линейную последовательность символов. Текст этой лекции (если не учитывать специальные символы форматирования) можно сохранить в файле с именем "Lecture_4.txt" в виде (рис. 1):

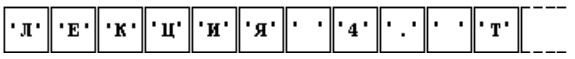

Рис. 1. Начало файла "`lecture_4.txt`".

2. Потоки ввода/вывода

Перед началом изучения файловых операций в Си++, необходимо ознакомиться с понятием *потока ввода/вывода*. Поток напоминает "канал" или "трубу", через которую данные поступают от передатчика к приемнику. Исключительно важная особенность потоковой обработки данных состоит в том, что элементы данных можно посылать или считывать из потока только по одному за раз, т.е. *последовательно*.

В данном курсе рассматриваются только однонаправленные потоки, в которых данные всегда передаются в одном направлении. Из программы данные можно отправить (записать) в *поток вывода*, а получить (прочитать) их в программе из *потока ввода*. Например, сразу после запуска программы, поток стандартного ввода "cin" подключается к клавиатуре, а поток стандартного вывода "cout" – к экрану.

Потоки ввода/вывода, вроде "cin" и "cout", являются примерами *объектов* класса "поток". Поэтому изучение потоков полезно и по той причине, что позволяет ознакомиться с элементами синтаксиса и некоторыми объектно-ориентированными понятиями Си++.

Список функций для работы с файловыми потоками хранится в заголовочном файле "fstream.h". Поэтому во всех рассматриваемых ниже фрагментах программ предполагается, что в начале программы есть соответствующая директива "#include":

```
#include<fstream.h>
```

2.1 Создание потоков

В программе перед первым обращением к потоку ввода или вывода необходимо "создать" поток. Операторы для создания потоков похожи на описания перемен-

ных, и они обычно размещаются в начале программы или функции рядом с описаниями переменных. Например, операторы

```
ifstream in_stream;
ofstream out_stream;
```

создают поток с именем "in_stream", являющийся объектом *класса* (как типа данных) "ifstream" (input-file-stream, файловый поток ввода), и поток с именем "out_stream", являющийся объектом класса "ofstream" (output-file-stream, файловый поток вывода). Аналогию между потоками и обычными переменными (типа "int", "char" и т.д.) не следует понимать слишком буквально. Например, к потокам нельзя применять оператор присваивания (например, нельзя записать "in_stream1 = in_stream2").

2.2 Подключение и отключение потоков от файлов

После создания потока его можно подключить к файлу (открыть файл) с помощью *функции-члена* "open(...)". (В предыдущих лекциях уже использовались несколько функций-членов потоков вывода. Например, во 2-й лекции применялись "precision(...)" и "width(...)".) Функция "open(...)" у потоков ifstream и ofstream работает по-разному (т.е. это полиморфная функция).

Для подключения потока ifstream с именем "in_stream" к файлу с именем "Lecture_4.txt" надо применить следующий вызов:

```
in_stream.open("Lecture_4.txt");
```

Этот оператор подключит поток "in_stream" к началу файла "Lecture_4.txt" (графически состояние программы после выполнения оператора показано на рис. 2).

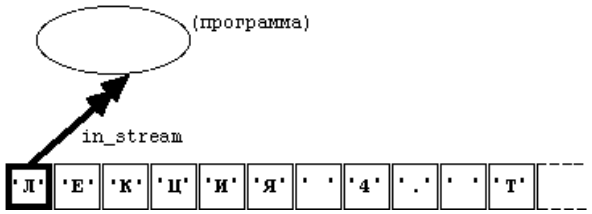

Рис. 2. Состояние программы после подключения потока ввода к файлу.

Чтобы к файлу "Lecture_4.txt" подключить поток вывода ofstream с именем "out_stream", надо выполнить аналогичный оператор:

```
out_stream.open("Lecture_4.txt");
```

Этот оператор подключит поток "out_stream" к файлу "Lecture_4.txt", но при этом прежнее содержимое файла будет удалено. Файл будет подготовлен к приему новых данных (рис. 3).

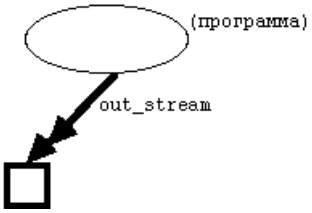

Рис. 3. Состояние программы после подключения потока вывода к файлу.

Для отключения потока "in_stream" от файла, к которому он подключен (для закрытия файла), надо вызвать функцию-член "close()":

```
in_stream.close();
```

После этого состояние программы по отношению к файлу будет таким, как на рис. 4.

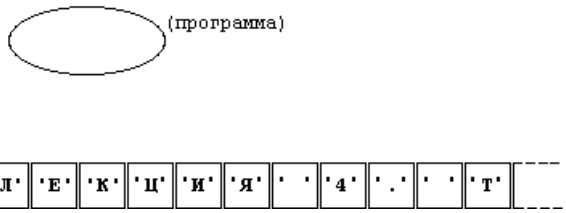

Рис. 4. Состояние программы после отключения потока ввода от файла.

Функция-член отключения от файла у потока вывода:

```
out_stream.close();
```

выполняет аналогичные действия, но, дополнительно, в конец файла добавляется служебный символ "end-of-file (маркер конца файла)". Т.о., даже если в поток вывода не записывались никакие данных, то после отключения потока "out_stream" в файле "Lecture_4.txt" будет один служебный символ (рис. 5). В таком случае файл "Lecture_4.txt" останется на диске, но он будет *пустым*.

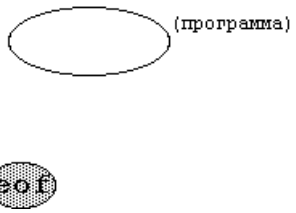

Рис. 5. Состояние программы после отключения потока вывода от файла, в который не было записано ни одного символа..

3. Проверка ошибок выполнения файловых операций

Файловые операции, например, открытие и закрытие файлов, известны как один из наиболее вероятных источников ошибок. В надежных коммерческих программах всегда выполняется проверка, успешно или нет завершилась файловая операция. В случае ошибки вызывается специальная функция-обработчик ошибки.

Простейший способ проверки ошибок файловых операций заключается в вызове функции-члена "fail()". Вызов

```
in_stream.fail();
```

возвращает истинное значение (True), если последняя операция потока "in_stream" привела к ошибке (может быть, была попытка открытия несуществующего файла). После ошибки поток "in_stream" может быть поврежден, поэтому лучше не продолжать работу с ним.

В приведенном ниже фрагменте программы в случае ошибки при открытии файла на экран выдается сообщение и программа завершает работу с помощью библиотечной функции "exit()" (она описана в файле "stdlib.h"):

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

int main()
{
```

```
ifstream in_stream;

in_stream.open( "Lecture_4.txt" );
if ( in_stream.fail() )
  {
  cout << "Извините, открыть файл не удалось!\n";
  exit(1);
  }
...
```

4. Символьный ввод/вывод

4.1 Функция ввода "`get(...)`"

После того, как файл для ввода данных открыт, из него можно считывать отдельные символы. Для этого служит функция "get(...)". У нее есть параметр типа "char&". Если программа находится в состоянии, как на рис. 2, то после вызова:

```
in_stream.get(ch);
```

произойдет следующее: (а) переменной "ch" будет присвоено значение "'Л'", и (б) поток "in_stream" будет подготовлен для чтения следующего символа (рис. 6).

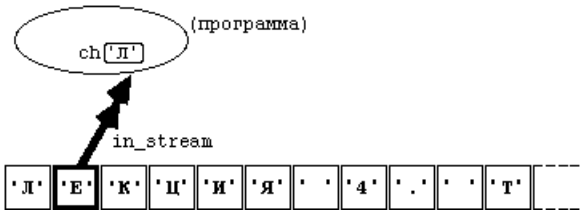

Рис. 6. Состояние программы после чтения из файла первого символа.

4.2 Функция вывода "`put(...)`"

С помощью потока вывода класса ofstream в открытый файл можно *записывать* отдельные символы. Для этого у класса ofstream есть функция-член "put(...)". Записываемый символ передается ей как параметр типа "char". Если программа пребывает в состоянии, представленном на рис. 3, то оператор

```
out_stream.put('Л');
```

изменит состояние на то, которое показано на рис. 7:

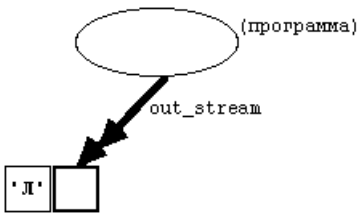

Рис. 7. Состояние программы после записи в файл первого символа.

4.3 Функция "`putback(...)`"

В Си++ у потока `ifstream` есть функция-член "`putback(...)`". На самом деле она не "возвращает символ назад" (т.е. не изменяет содержимого файла ввода), но ведет себя так, как будто это делает. На рис. 8 показано состояние, в которое перейдет программа из состояния рис. 6 после выполнения оператора:

```
in_stream.putback(ch);
```

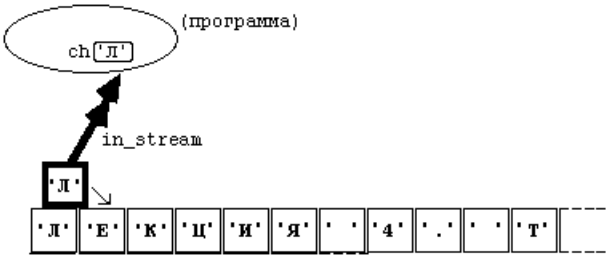

Рис. 8. Состояние программы после вызова функции "`putback('Л')`".

"Вернуть назад" можно любой символ. Состояние программы после вызова

```
in_stream.putback('7');
```

показано на рис. 9.

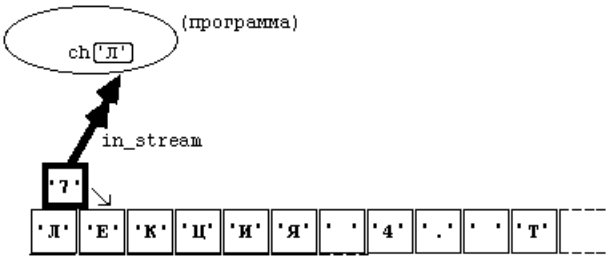

Рис. 9. Состояние программы после вызова функции "`putback('7')`".

5. Проверка достижения конца файла при операциях ввода

5.1 Проверка конца файла с помощью функции "`eof()`"

При работе с потоком ввода надо следить за тем, чтобы не пропустить момент достижения конца файла. В большинстве реализаций Си++ (в том числе и в **Microsoft Visual C++**) в класс "поток ввода" встроен *флаг* "конец файла (end-of-file, EOF)" и функция-член "`eof()`" для чтения этого флага. С помощью функции "`eof()`" можно узнать, находится ли в данный момент флаг в состоянии `True` (конец файла достигнут) или `False` (конца файла пока нет).

При открытии файла, когда поток `ifstream` только подключается к нему, флаг EOF сбрасывается в значение `False` (даже если файл пуст). Но, если `ifstream` "`in_stream`" сейчас расположен в конце файла, и флаг EOF равен `False`, то после вызова

```
in_stream.get(ch);
```

переменная "`ch`" окажется в неопределенном состоянии, а флагу EOF будет присвоено значение `True`. Если флаг EOF равен `True`, то программа не должна пытаться выполнить чтение из файла, поскольку результат чтения будет неопределенным.

Допустим, программа находится с состоянии, показанном на рис. 10.

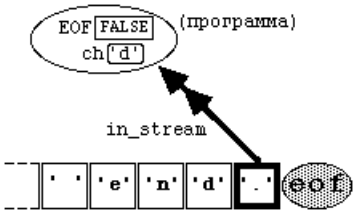

Рис. 10. Состояние программы после чтения предпоследнего символа из файла.

Тогда после выполнения оператора

```
in_stream.get(ch);
```

программа перейдет в состояние, изображенное на рис. 11.

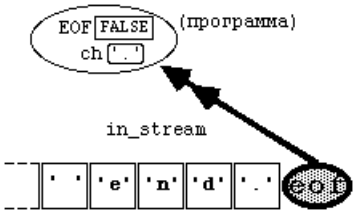

Рис. 11. Состояние программы после чтения последнего символа из файла.

Теперь логическое выражение

```
in_stream.eof()
```

будет иметь истинное значение `True`. Если снова выполнить чтение символа:

```
in_stream.get(ch);
```

то в результате получится состояние, показанное на рис. 12.

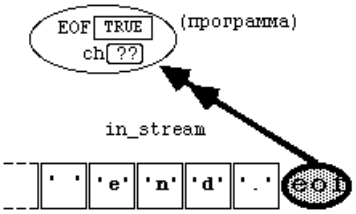

Рис. 12. Состояние программы после операции чтения символа при установленном флаге EOF.

Ниже приведена простая программа для копирования текстового файла "`Lecture_4.txt`" одновременно и на экран, и в другой файл "`Copy_of_4.txt`". Для прекращения копирования применяется вызов функции "`eof()`". Обратите внимание на цикл "`while`". Цикл с префиксным условием (предусловием) "`while`" является упрощенным вариантом цикла "`for`". У цикла "`while`" в круглых скобках "`()`" нет операторов инициализации и изменения значений (подробно эти циклы рассматриваются в следующей лекции).

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

int main()
{
  char character;
  ifstream in_stream;
  ofstream out_stream;
```

```
  in_stream.open( "Lecture_4.txt" );
  out_stream.open( "Copy_of_4.txt" );

  in_stream.get( character );
  while ( !in_stream.eof() )
    {
    cout << character;
    out_stream.put(character);
   in_stream.get(character);
    }

  out_stream.close();
  in_stream.close();

  return 0;
}
```

Программа 5.1.

6. Передача потоков функциям в качестве параметров

Потоки можно использовать в качестве параметров функций, но их *обязательно* надо передавать *по ссылке* (а не по значению). Ниже приведен усовершенствованный вариант программы 5.1, в котором копирование выполняется специальной функцией "`copy_to(...)`".

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

void copy_to( ifstream& in, ofstream& out );

// Главная функция
int main()
{
  ifstream in_stream;
  ofstream out_stream;

  in_stream.open( "Lecture_4.txt" );
  out_stream.open( "Copy_of_4.txt" );
  copy_to( in_stream, out_stream );
  out_stream.close();
  in_stream.close();

  return 0;
}
// Конец главной функции

// Функция для копирования файла в другой файл и на экран
void copy_to( ifstream& in, ofstream& out )
{
  char character;

  in.get( character );
  while ( !in.eof() )
    {
    cout << character;
    out.put( character );
```

```
    in.get( character );
    }
}
// Конец функции
```

Программа 6.1.

7. Операторы ввода/вывода ">>" и "<<"

До сих пор рассматривались способы записи и чтения из файлов отдельных символов. На нижнем уровне, скрытом внутри классов `ofstream` and `ifstream`, объекты этих классов всегда работают с файлами как с последовательностями символов. Поэтому данные других типов ("`int`", "`double`" и др.) для записи в файл должны быть преобразованы в последовательность символов. При чтении из файла эти последовательности должны быть преобразованы обратно.

Некоторые преобразования типов данных автоматически выполняются операторами ">>" и "<<" (в предыдущих лекциях они часто использовались для ввода с клавиатуры и вывода на экран). Например, из состояния, показанного на рис. 13:

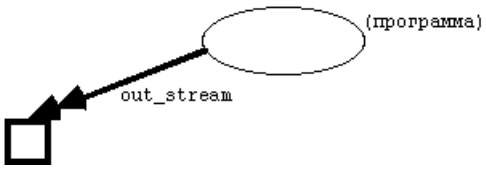

Рис. 13. Состояние программы после открытия файла вывода (после подключения потока вывода к файлу).

с помощью оператора

```
out_stream << 437 << ' ';
```

программа перейдет в новое состояние, изображенное на рис. 14.

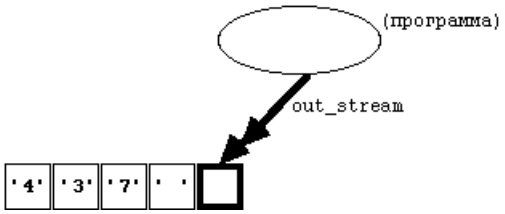

Рис. 14. Состояние программы после записи в поток вывода целого значения "`437`" и пробела.

При использовании операторов ">>" и "<<" обязательно надо после каждого записанного значения записывать еще как минимум один символ `' '` (пробел) или служебный символ `'\n'` (маркер конца строки). Это гарантирует, что элементы данных будут корректно отделены в файле друг от друга, и их можно будет извлекать оттуда с помощью оператора ">>". Например, в состоянии, показанном на рис. 15:

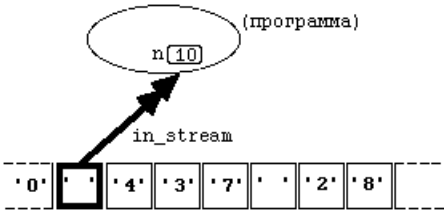

Рис. 15. Состояние программы в некоторый момент времени, когда текущая позиция потока ввода установлена на пробел.

если "`n`" является переменной типа "`int`" и имеет значение `10`, то выполнение оператора

```
in_stream >> n;
```

приведет к состоянию, показанному на рис. 16.

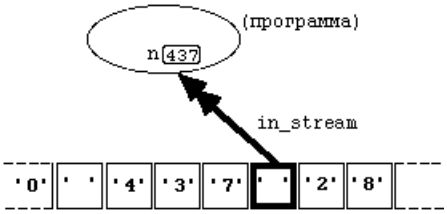

Рис. 16. Состояние программы после чтения из потока ввода целого значения "`437`".

Обратите внимание, что оператор "`>>`" перед числом `437` пропустил пробел `' '`. Этот оператор всегда пропускает пробелы, независимо от типа считываемых данных (даже при чтении символов).

Работа с операторами "`>>`" и "`<<`" продемонстрирована в программе 7.1. Сначала она создает файл "`Integers.txt`", записывает в него целые числа `51`, `52`, `53`, `54` и `55`, а затем считывает эти числа из файла.

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

int main()
{
  char character;
  int number = 51;
  int count = 0;
  ofstream out_stream;
  ifstream in_stream1;   // Поток для подсчета целых чисел.
  ifstream in_stream2;   // Поток для подсчета символов.

  // Создание файла
  out_stream.open( "Integers.txt" );
  for ( count = 1; count <= 5; count++ )
    out_stream << number++ << ' ';
  out_stream.close();

  // Подсчет количества целых чисел в файле
  in_stream1.open( "Integers.txt" );
  count = 0;
  in_stream1 >> number;
  while ( !in_stream1.eof() )
    {
```

```
    count++;
    in_stream1 >> number;
    }
  in_stream1.close();
  cout << "В файле хранится " << count << " целых чисел,\n";

  // Подсчет количества символов, не являющихся разделителями
  in_stream2.open( "Integers.txt" );
  count = 0;
  in_stream2 >> character;
  while ( !in_stream2.eof() )
    {
    count++;
    in_stream2 >> character;
    }
  in_stream2.close();
  cout << "представленных с помощью " << count << " символов.\n";

  return 0;
}
```

Программа 7.1.

Программа 7.1 выведет на экран следующие сообщения:

```
В файле хранится 5 целых чисел,
представленных с помощью 10 символов.
```

При подсчете символов в последней части программы 7.1 снова обратите внимание на то, что, в отличие от функции "`get(...)`", оператор "`>>`" игнорирует в файле пробелы (которые разделяют пять целых чисел).

8. Сводка результатов

В лекции рассмотрены способы работы с текстовыми файлами с помощью потоков ввода/вывода. "Низкоуровневый" ввод/вывод выполняется с помощью функций "`get(...)`" и "`put(...)`", а "высокоуровневый" ввод/вывод значений разных типов – с помощью потоковых операторов "`>>`" и "`<<`".

9. Упражнения

Упражнение 1

Напишите программу, печатающую на экране содержимое собственного исходного файла на Си++.

Упражнение 2

Разработайте программу, которая (1) начинается с оператора вывода тестового сообщения:

```
cout << "Проверка: " << 16/2 << " = " << 4*2 << ".\n\n";
```

и затем (2) копирует собственный исходный файл на Си++ в файл "`WithoutComments.cpp`" и на экран, при этом пропуская все комментарии между маркерами "`/* ... */`" (и маркеры комментариев тоже).

Получившийся файл "`WithoutComments.cpp`" должен компилироваться и работать точно так же, как и исходная программа.
Подсказки: *(1) вам может пригодиться функция "*`putback()`*"; (2) для отслеживания состояния, находитесь ли вы внутри комментария или нет, можете применить логический "флаг".*

Упражнение 3

Напишите программу, которая подсчитывает и выводит на экран количество символов (включая пробелы) в собственном исходном файле.

Упражнение 4

Без использования массива (массивы будут рассмотрены в 6-й лекции) напишите программу, которая печатает на экране собственный исходный файл в обратном порядке.
Подсказки: *(1) Для начальной части этой программы может пригодиться программа из упражнения 3. (2) Будьте внимательны и не используйте поток ввода после ошибки – вместо этого работайте с новым потоком.*
Примечание: *не беспокойтесь, если перед началом печати происходит небольшая задержка – открытие и закрытие файлов требует некоторого времени.*

Упражнение 5

Что выведет на экран следующая программа?

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

int main()
{
  char character;
  int integer;
  ofstream out_stream;
  ifstream in_stream;

  // Создание текстового файла и запись в него двух целых чисел
  out_stream.open( "Integers.txt" );
  out_stream << 123 << ' ' << 456;
  out_stream.close();

  // Попытка чтения из только что созданного файла
  // "Integers.txt" символа, затем целого числа,
  // затем снова символа, затем опять целого числа.
  in_stream.open( "Integers.txt" );
  in_stream >> character >> integer;
  cout << "символ: '" << character << "'\n";
  cout << "целое число: " << integer << "\n";
  in_stream >> character >> integer;
  cout << "символ: '" << character << "'\n";
  cout << "целое число: " << integer << "\n";
  in_stream.close();

  return 0;
}
```

ЛЕКЦИЯ 5. Операторы ветвления и циклы

1. Логические значения, выражения и функции

В этой лекции подробно рассматриваются операторы ветвления ("`if`" и "`switch`") и операторы циклов "`for`" и "`while`". Для применения всех этих операторов необходимо хорошо знать, что такое логические выражения и как они вычисляются.

Язык Си++ унаследовал от языка Си соглашение, согласно которому целое значение 0 считается логическим "`false`" (ложное значение), а ненулевое целое – логическим "`true`" (истинным значением). Но выражения вроде

```
условие1 == 1
```

или

```
условие2 == 0
```

не слишком удобны при чтении теста программ человеком. Было бы лучше записывать логические выражения в интуитивно понятном виде:

```
условие1 == true
```

и

```
условие2 == false
```

Поэтому в Си++ был добавлен специальный логический тип "`bool`". Переменные типа "`bool`" могут принимать значения "`true`" и "`false`", которые при необходимости автоматически преобразуются в выражениях в значения `1` и `0`.

Тип данных "`bool`" можно использовать в программах точно так же, как и типы "`int`", "`char`" и др. (например, для описания переменных или для создания функций, возвращающих значения типа "`bool`").

Программа 1.1 приведена в качестве примера использования типа данных "`bool`". Она запрашивает с клавиатуры возраст кандидата, сдававшего некий тест, и полученную кандидатом оценку в баллах. Затем программа оценивает результат выполнения теста по шкале, зависящей от возраста кандидата и делает вывод о том, сдан тест или нет. Для кандидатов до 14 лет порог сдачи теста составляет 50 баллов, для 15 или 16 лет – 55 баллов, старше 16-ти лет – 60 баллов.

```
#include <iostream.h>

bool acceptable( int age, int score );

int main()
{
  int candidate_age, candidate_score;

  cout << "Введите возраст кандидата: ";
  cin >> candidate_age;
  cout << "Введите результат тестирования: ";
  cin >> candidate_score;

  if ( acceptable( candidate_age, candidate_score ) )
    cout << "Этот кандидат сдал тест успешно.\n";
  else
    cout << "Этот кандидат тест не прошел.\n";
```

```
  return 0;
}

// Функция оценки результата тестирования: тест сдан/не сдан
bool acceptable( int age, int score )
{
  if ( age <= 14 && score >= 50 )
    return true;
  else if ( age <= 16 && score >= 55 )
    return true;
  else if ( score >= 60 )
    return true;
  else
    return false;
}
```

Программа 1.1.

2. Циклы "for", "while" и "do...while"

Циклы "`for`" впервые встречались во 2-й лекции, цикл "`while`" упоминался в 4-й лекции. Любой цикл "`for`" можно переписать в виде цикла "`while`" и наоборот. Рассмотрим программу 2.1 (она похожа на программу 2.2 из 2-й лекции).

```
#include <iostream.h>

int main()
{
  int number;
  char character;

  for ( number = 32; number <= 126; number = number + 1 )
    {
    character = number;
    cout << "Символ '" << character;
    cout << "' имеет код " << number << "\n";
    }

  return 0;
}
```

Программа 2.1.

Программу 2.1 можно переписать с помощью цикла "`while`" (программа 2.2):

```
#include <iostream.h>

int main()
{
  int number;
  char character;

  number = 32;
  while ( number <= 126 )
    {
    character = number;
    cout << "Символ '" << character;
    cout << "' имеет код " << number << "\n";
    number++;
```

```
    }

  return 0;
}
```

Программа 2.2.

Замена цикла "`while`" на цикл "`for`" выполняется совсем просто. Например, в программе 2.2 строку

```
while (number <= 126)
```

можно заменить эквивалентным циклом "`for`" без операторов инициализации и изменения значений:

```
for ( ; number <= 126; )
```

В Си++ есть еще один, третий, вариант цикла – оператор цикла с постфиксным условием (постусловием) *"do {...} while"*. Он отличается от циклов "`for`" и "`while`" тем, что тело цикла внутри скобок "`{}`" обязательно выполняется как минимум один раз, т.к. условие повторения цикла проверяется только после выполнения тела цикла. Циклы "`Do ... while`" используются довольно редко. Например этот цикл можно применить для проверки корректности введенных с клавиатуры данных (программа 2.2a).

```
...
...
do {
  cout << "Введите результат тестирования: ";
  cin >> candidate_score;
  if ( candidate_score > 100 || candidate_score < 0 )
    cout << "Допускается оценка от 0 до 100.\n";
} while ( candidate_score > 100 || candidate_score < 0 );
...
...
```

Фрагмент программы 2.2a.

В программе 2.2a цикл с постусловием позволяет избавиться от дублирования операторов печати приглашения и ввода данных, которое возникает при использовании эквивалентного цикла "`while`" (программа 2.2b).

```
...
...
cout << "Введите результат тестирования: ";
cin >> candidate_score;
while ( candidate_score > 100 || candidate_score < 0 )
  {
  cout << "Допускается оценка от 0 до 100.\n";
  cout << "Введите результат тестирования: ";
  cin >> candidate_score;
  }
...
...
```

Фрагмент программы 2.2b.

3. Множественное ветвление и оператор "switch"

Вложенные операторы "`if`", предназначенные для выполнения "множественного ветвления", уже встречались в 1-й лекции. Упрощенная версия этого примера приведена ниже:

```
...
...
if ( total_test_score < 50 )
  cout << "Вы не прошли тест. Выучите материал как следует.\n";
else if ( total_test_score < 60 )
  cout << "Вы прошли тест со средним результатом.\n";
else if ( total_test_score < 80 )
  cout << "Вы хорошо выполнили тест.\n";
else if ( total_test_score < 100 )
  cout << "Вы показали отличный результат.\n";
else
  {
  cout << "Вы выполнили тест нечестно";
  cout << "(оценка должна быть меньше 100 баллов)!\n";
  }
...
...
```

Вложенные операторы "`if`" выглядят слишком громоздко, поэтому в Си++ реализован еще один способ множественного ветвления – оператор "`switch`". Он позволяет выбрать для выполнения один из нескольких операторов, в зависимости от текущего значения определенной переменной или выражения.

В приведенном выше примере сообщение для печати выбирается в зависимости от значения переменной "`total_test_score`". Это может быть произвольное целое число в диапазоне от 0 до 100. Диапазон проверяемых значений можно сузить, если учесть, что оценка теста проверяется с точностью до 10-ти баллов. Введем дополнительную целочисленную переменную "`score_out_of_ten`" и присвоим ей значение:

```
score_out_of_ten = total_test_score/10;
```

Теперь проверку в программе можно сформулировать так: (1) если переменная "`score_out_of_ten`" равна 0, 1, 2, 3 или 4, то печатать сообщение "`Вы не прошли тест. Выучите материал как следует.`", (2) если "`score_out_of_ten`" равна 5, то печатать сообщение "`Вы прошли тест со средним результатом.`" и т.д. В целом оператор "`switch`" будет выглядеть так:

```
...
...
score_out_of_ten = total_test_score / 10;
switch ( score_out_of_ten )
  {
  case 0 :
  case 1 :
  case 2 :
  case 3 :
  case 4 :
    cout << "Вы не прошли тест. Выучите материал как следует.\n";
    break;
  case 5 :
    cout << "Вы прошли тест со средним результатом.\n";
    break;
  case 6 :
  case 7 :
```

```
      cout << "Вы хорошо выполнили тест.\n";
      break;
    case 8 :
    case 9 :
    case 10 :
      cout << "Вы показали отличный результат.\n";
      break;
    default :
      cout << "Вы выполнили тест нечестно\n";
      cout << "(оценка не должна быть больше 100 баллов)!\n";
    }
...
...
```

Фрагмент программы 3.1.

Оператор "switch" имеет следующий синтаксис:

```
switch (селектор)
  {
  case метка1 :
    <операторы1>
    break;
  ...
  ...
  ...
  case меткаN :
    <операторыN>
    break;
  default :
    <операторы>
  }
```

Сделаем несколько важных замечаний относительно оператора "switch":

- Внутри "switch" выполняются операторы, содержащиеся между меткой, совпадающей с текущим значением селектора, и первым встретившимся после этой метки оператором "break".
- Операторы "break" необязательны, но они улучшают читабельность программ. С ними сразу видно, где заканчивается каждый вариант множественного ветвления. Как только при выполнении операторов внутри "switch" встречается "break", то сразу выполняется переход на первый оператор программы, расположенный после оператора "switch". Иначе продолжается последовательное выполнение операторов внутри "switch".
- Селектор (переменная или выражение) может быть целочисленного (например, "int" или "char") или любого перечислимого типа, но не вещественного типа.
- Вариант "default" ("по умолчанию") необязателен, но для безопасности лучше его предусмотреть.

4. Блоки и область видимости переменных

В Си++ фигурные скобки "{}" позволяют оформить составной оператор, который содержит несколько операторов, но во всех конструкциях языка может подстав-

ляться как один оператор. На описания переменных фигурные скобки также оказывают важное влияние.

Составной оператор, внутри которого описана одна или несколько переменных, называется *блоком*. Для переменных, объявленных внутри блока, этот блок является *областью видимости*. Другими словами, переменные "создаются" каждый раз, когда при выполнении программа входит внутрь блока, и "уничтожаются" после выхода из блока.

Если одно и то же имя используется для переменной внутри и снаружи блока, то это две разных, независимых переменных. При выполнении внутри блока программа по умолчанию полагает, что имя относится к внутренней переменной. Обращение к внешней переменной происходит только в том случае, если переменная с таким именем не описана внутри блока. Действие этого правила продемонстрировано в программе 4.1.

```
#include <iostream.h>

int integer1 = 1;
int integer2 = 2;
int integer3 = 3;

int main()
{
  int integer1 = -1;
  int integer2 = -2;
    {
    int integer1 = 10;
    cout << "integer1 == " << integer1 << "\n";
    cout << "integer2 == " << integer2 << "\n";
    cout << "integer3 == " << integer3 << "\n";
    }
  cout << "integer1 == " << integer1 << "\n";
  cout << "integer2 == " << integer2 << "\n";
  cout << "integer3 == " << integer3 << "\n";

  return 0;
}
```

Программа 4.1.

Программа 4.1 выводит на экран сообщения:

```
integer1 == 10
integer2 == -2
integer3 == 3
integer1 == -1
integer2 == -2
integer3 == 3
```

Применение локальных переменных иногда объясняется экономией памяти, а иногда необходимостью использования в различных частях программы разных переменных с одинаковыми именами. См. в качестве примера программу 4.2, которая печатает таблицу умножения для чисел от 1 до 10.

```
#include <iostream.h>

int main()
{
  int number;

  for ( number = 1; number <= 10; number++ )
```

```
    {
    int multiplier;

    for ( multiplier = 1; multiplier <= 10; multiplier++ )
      {
      cout << number << " x " << multiplier << " = ";
      cout << number * multiplier << "\n";
      }
    cout << "\n";
    }

  return 0;
}
```

Программа 4.2.

Программу 4.2 можно переписать в более понятном виде с помощью функции (см. программу 4.3).

```
#include <iostream.h>

void print_times_table( int value, int lower, int upper );

int main()
{
  int number;

  for ( number = 1; number <= 10; number++ )
    {
    print_times_table( number, 1, 10 );
    cout << "\n";
    }

  return 0;
}

void print_times_table( int value, int lower, int upper )
{
  int multiplier;

  for ( multiplier = lower; multiplier <= upper; multiplier++ )
    {
    cout << value << " x " << multiplier << " = ";
    cout << value * multiplier << "\n";
    }
}
```

Программа 4.3.

Далее, программу 4.3 можно усовершенствовать, исключив описания всех переменных из "`main()`" и добавив две функции (см. программу 4.4).

```
#include <iostream.h>

void print_tables( int smallest, int largest );
void print_times_table( int value, int lower, int upper );

int main()
{
  print_tables( 1, 10 );
  return 0;
}

void print_tables( int smallest, int largest )
```

```
{
  int number;

  for ( number = smallest; number <= largest; number++ )
    {
    print_times_table( number, 1, 10 );
    cout << "\n";
    }
}

void print_times_table( int value, int lower, int upper )
{
  int multiplier;

  for ( multiplier = lower; multiplier <= upper; multiplier++ )
    {
    cout << value << " x " << multiplier << " = ";
    cout << value * multiplier << "\n";
    }
}
```

Программа 4.4.

5. Замечание о вложенных циклах

В первоначальном варианте программы "таблица умножения" (программа 4.2) есть вложенные циклы. В последующих вариантах программы читабельность исходного текста улучшается с помощью процедурной абстракции. Преобразование тела цикла в вызов функции позволяет производить разработку этого алгоритма независимо от остальной части программы. Поэтому уменьшается вероятность ошибок, связанных с областью видимости переменных и перегрузкой имен переменных.

Недостаток выноса тела цикла в отдельную функцию заключается в уменьшении быстродействия, поскольку на вызов функции тратится больше времени, чем на итерацию цикла. Если цикл выполняется не очень часто и не содержит большого количества итераций (больше нескольких десятков), то временными затратами на вызов функции вполне можно пренебречь.

6. Сводка результатов

Тип данных "`bool`" предназначен для использования в логических выражениях и в качестве возвращаемого значения логических функций. Такие функции можно применять в качестве условий в условных операторах и операторах циклов. В Си++ есть три варианта циклов: "`for`", "`while`" и "`do ... while`".

Вложенные операторы "`if`" в некоторых случаях можно заменить оператором множественного ветвления "`switch`".

Внутри составного оператора (блока), ограниченного фигурными скобками "`{}`", допускается описание локальных переменных (внутренних переменных блока).

7. Упражнения

Упражнение 1

Разработайте функцию, которая принимает целочисленный параметр и возвращает логическое ("`bool`") значение "`true`", только если переданное ей число является простым числом из диапазона от 1 до 1000 (число 1 простым не считается). Проверьте свою функцию на различных входных данных с помощью тестовой программы.
Подсказка: *(1) если число не является простым, то оно имеет как минимум один простой множитель, меньший или равный квадратному корню из числа.*
*(2) (32*32)=1024 и 1024>1000.*

Упражнение 2

Напишите функцию "`print_pyramid(...)`", которая получает целочисленный параметр "`height` (высота)" и отображает на экране "пирамиду" заданной высоты из символов "`*`". Проверьте функцию с помощью простой тестовой программы, которая должна воспроизводить следующий примерный диалог с пользователем:

```
Эта программа печатает на экране "пирамиду"
заданной высоты.

Введите высоту пирамиды: 37
Введите другое значение (из диапазона от 1 до 30): 6

             **
            ****
           ******
          ********
         **********
        ************
```

Упражнение 3

Цикл "`for`" всегда можно переписать в форме цикла "`while`", и наоборот. Являются ли две показанных ниже программы эквивалентными? Какие сообщения они печатают на экране? Объясните свой ответ и проверьте его опытным путем.

Программа 3a:

```
#include <iostream.h>

int main()
{
  int count = 1;
  for ( ; count <= 5; count++ )
    {
    int count = 1;
    cout << count << "\n";
    }
  return 0;
}
```

Программа 3b:

```
#include <iostream.h>

int main()
{
  int count = 1;
  while ( count <= 5 )
    {
    int count = 1;
    cout << count << "\n";
    count++;
    }
  return 0;
}
```

Упражнение 4

Приведенная ниже программа должна печатать время закрытия магазина в различные дни недели (в виде таблицы). В программе объявлен новый перечислимый тип данных "День" и определена функция "`closing_time(...)`", которая должна возвращать час закрытия магазина в заданный день (пока эта функция не слишком сложна – для любого дня возвращает значение 17). Программа демонстрирует, как можно использовать типы "`int`" и "`Day`" в преобразованиях типов (в заголовке цикла "`for`" и при вызове функции "`closing_time(...)`").

```
#include <iostream.h>

enum Day { Monday, Tuesday, Wednesday, Thursday,
           Friday, Saturday, Sunday };

int closing_time( Day day_of_the_week );

// Главная функция
int main()
{
  int count;

  // Печать заголовка таблицы
  cout.width(17);
  cout << "ДЕНЬ";
  cout.width(19);
  cout << "ВРЕМЯ ЗАКРЫТИЯ \n\n";

  // Печать таблицы от понедельника (Monday) до
  // воскресенья (Sunday)
  for ( count = int(Monday); count <= int(Sunday); count++ )
    {
    cout.width(19);
    switch ( count )
      {
      case 0 : cout << "Понедельник"; break;
      case 1 : cout << "Вторник"; break;
      case 2 : cout << "Среда"; break;
      case 3 : cout << "Четверг"; break;
      case 4 : cout << "Пятница"; break;
      case 5 : cout << "Суббота"; break;
      case 6 : cout << "Воскресенье"; break;
      default : cout << "ОШИБКА!";
      }
    cout.width(9);
    cout << closing_time( Day(count) ) << ":00\n";
    }

  return 0;
}
// Конец главной функции

// Функция, возвращающая время закрытия магазина
// в заданный день недели
int closing_time( Day day_of_the_week )
{
  return 17;
}
```

(а) Что произойдет (и почему), если заменить оператор "switch" на строку

```
cout << Day(count); ?
```

Вместо этого замените "switch" на строку

```
print_day( Day(count), cout );
```

и добавьте описание и определение функции "print_day(...)", внутри которой разместите удаленный из главной функции оператор "switch" (поток стандартного вывода "cout" передавайте в функцию как параметр по ссылке типа "ostream&").

(б) Магазин закрывается в воскресенье в 13:00, в субботу в 17:00, в среду в 20:00 и в остальные дни в 18:00. С помощью оператора "switch" внесите соответствующие изменения в функцию "closing_time(...)" и проверьте работу программы.

Упражнение 5

Напишите программу, которая отображает в виде таблицы количество строчных английских букв (от 'a' до 'z') в собственном исходном файле "ex5_5.cpp" (сохраните исходный текст программы именно в этом файле).

При разработке программы предположите, что у компьютера очень мало памяти – используйте только одну переменную типа "ifstream", одну переменную типа "char" и две переменных типа "int". Программа должна выдавать на экран сообщения, похожие на следующие:

```
СИМВОЛ     КОЛИЧЕСТВО ВХОЖДЕНИЙ

  a                38
  b                5
  c                35
  d                7
  e                58
  f                8
...
...
  w                4
  x                4
  y                0
  z                1
```

ЛЕКЦИЯ 6. Массивы и символьные строки

1. Назначение массивов

В программировании часто возникают задачи, связанные с обработкой больших объемов данных. Для постоянного хранения этих данных удобно пользоваться файлами. Например, в программе для ввода и сортировки длинных числовых списков данные можно ввести с клавиатуры один раз и сохранить в файле для последующего многократного использования. Но до сих пор не было рассмотрено удобного способа представления больших объемов данных внутри программ. Для этой цели в Си++ часто применяются *массивы* – простейшая разновидность *структурных типов данных* (о более сложных структурах данных будет говориться в следующих лекциях).

Массив – это набор переменных одного типа ("`int`", "`char`" и др.). При объявлении массива компилятор выделяет для него *последовательность* ячеек памяти, для обращения к которым в программе применяется одно и то же имя. В то же время массив позволяет получить прямой доступ к своим отдельным элементам.

1.1 Объявление массивов

Оператор описания массива имеет следующий синтаксис:

```
<тип данных> <имя переменной>[<целое значение>];
```

Допустим, в программе требуется обрабатывать данные о количестве часов, отработанных в течении недели группой из 6-ти сотрудников. Для хранения этих данных можно объявить массив:

```
int hours[6];
```

или, лучше, задать численность группы с помощью специальной константы:

```
const int NO_OF_EMPLOYEES = 6;
int hours[NO_OF_EMPLOYEES];
```

Если подобные массивы будут часто встречаться в программе, то целесообразно определить новый тип:

```
const int NO_OF_EMPLOYEES = 6;
typedef int Hours_array[NO_OF_EMPLOYEES];
Hours_array hours;
Hours_array hours_week_two;
```

В любом из трех перечисленных вариантов, в программе будет объявлен массив из 6 элементов типа "`int`", к которым можно обращаться с помощью имен:

```
hours[0]  hours[1]  hours[2]  hours[3]  hours[4]  hours[5]
```

Каждое из этих имен является именем *элемента* массива. Числа `0`, `...`, `5` называются *индексами* элементов. Отличительная особенность массива заключается в том, что его элементы – однотипные переменные – занимают в памяти компьютера последовательные ячейки памяти (рис. 1).

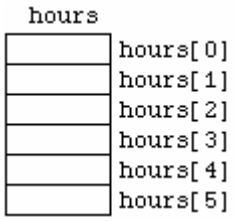

Рис. 1.. Расположение элементов массива в оперативной памяти (направление сверху вниз соответствует возрастанию адресов ячеек памяти).

1.2 Использование элементов массивов в выражениях

С элементами объявленного массива можно выполнять все действия, допустимые для обычных переменных этого типа (выше был приведен пример целочисленного массива, т.е. типа "`int`"). Например, возможны операторы присваивания наподобие:

```
hours[4] = 34;
hours[5] = hours[4]/2;
```

или логические выражения с участием элементов массива:

```
if (number < 4 && hours[number] >= 40) { ...
```

Присвоить значения набору элементов массива часто бывает удобно с помощью циклов "`for`" или "`while`". В программе 1.1 в цикле у оператора запрашивается количество часов, отработанных каждым сотрудником группы за неделю. Хотя более естественной может показаться нумерация сотрудников от 1 до 6, а не от 0 до 5, но необходимо помнить, что индексация массивов в Си++ начинается с 0. Поэтому программа 1.1 вычитает 1 из порядкового номера сотрудника, чтобы вычислить индекс соответствующего элемента массива.

```
#include <iostream.h>

const int NO_OF_EMPLOYEES = 6;
typedef int Hours_array[NO_OF_EMPLOYEES];

int main()
{
  Hours_array hours;
  int count;

  for ( count = 1; count <= NO_OF_EMPLOYEES; count++ )
    {
    cout << "Сколько часов отработал сотрудник";
    cout << " номер " << count << "?: ";
    cin >> hours[count - 1];
    }

  return 0;
}
```

Программа 1.1.

В типичном сеансе работы программа 1.1 выведет на экран подобные сообщения:

```
Сколько часов отработал сотрудник номер 1?: 38
Сколько часов отработал сотрудник номер 2?: 42
Сколько часов отработал сотрудник номер 3?: 29
Сколько часов отработал сотрудник номер 4?: 35
```

```
Сколько часов отработал сотрудник номер 5?: 38
Сколько часов отработал сотрудник номер 6?: 37
```

На рис. 2. показано состояние целочисленного массива после ввода этих данных.

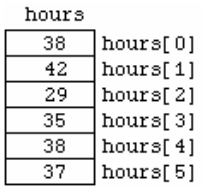

Рис. 2.. Состояние массива после присвоения значений его элементам.

Полезно разобраться, что произошло бы, если бы в программе 1.1 внутри цикла "`for`" в операторе "`cin ...`" отсутствовало бы вычитание 1 из переменной "`count`". Компилятор Си++ (в отличие, например, от Паскаля) не обнаруживает ошибки выхода за пределы массива, поэтому участок памяти компьютера с массивом и сразу за ним оказался бы в состоянии, показанном на рис. 3.

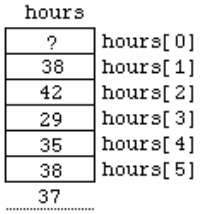

Рис. 3.. Ошибка выхода за пределы массива.

Другими словами, значение "`37`" было бы размещено в ячейке памяти, достаточной для хранения целого числа, которая расположена сразу после массива "`hours`". Это чрезвычайно нежелательная ситуация, потому что компилятор может зарезервировать эту ячейку памяти для другой переменной (например, для переменной "`count`").

Массивы могут быть любого типа, не обязательно типа "`int`". Ниже приведена программа 1.2, в которой символьный ("`char`") массив применяется для печати собственного исходного файла на экране в обратном порядке.

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

const int MAX_LEN = 1000;
typedef char File_array[MAX_LEN];

int main()
{
  char character;
  File_array file;
  int count;
  ifstream in_stream;

  in_stream.open("prg6_1_2.cpp");
  in_stream.get(character);
  for ( count = 0; !in_stream.eof() && count < MAX_LEN; count++ )
    {
    file[count] = character;
```

```
    in_stream.get(character);
    }
  in_stream.close();

  while (count > 0)
    cout << file[--count];

  return 0;
}
```

Программа 1.2.

В заголовке цикла "`for`" обратите внимание на условие "`... && count < MAX ; ...`", специально предусмотренное для предотвращения выхода за пределы массива.

2. Передача массивов в качестве параметров функций

Чтобы у программ была понятная человеку структура, допускающая модификацию и повторное использование исходного текста, отдельные алгоритмы следует реализовывать в виде самостоятельных функций. Т.к. для хранения данных часто применяются массивы, то бывают необходимы и функции для их обработки. Этим функциям массивы можно передавать в качестве параметров. В показанном ниже фрагменте программы 2.1 содержится определение функции, которая принимает массив типа "`Hours_array`" (см. программу 1.1) и возвращает среднее количество часов, отработанных сотрудниками группы.

```
float average( Hours_array hrs )
{
  float total = 0;
  int count;
  for ( count = 0; count < NO_OF_EMPLOYEES; count++ )
    total += float(hrs[count]);
  return ( total/NO_OF_EMPLOYEES );
}
```

Фрагмент программы 2.1.

Эту функцию можно сделать более универсальной, если размер массива не фиксировать в ее определении, а передать в качестве второго параметра:

```
float average( int list[], int length )
{
  float total = 0;
  int count;
  for ( count = 0 ; count < length; count++ )
    total += float(list[count]);
  return ( total/length );
}
```

В этом примере показан очень распространенный способ оформления функций, работающих с массивами: в одном параметре передается длина массива, а в другом – сам массив, причем в описании параметра-массива не указывается длина массива (например, "`int list[]`").

Параметры-массивы всегда передаются по ссылке (а не по значению), хотя при их описании в заголовке функции символ "`&`" не указывается. Правило "массивы все-

гда передаются по ссылке" введено для того, чтобы функции не делали собственных внутренних копий переданных им массивов – для больших массивов это могло бы привести к нерациональным затратам памяти. Следовательно, как и параметры по ссылке, рассматривавшиеся в 3-ей лекции, все изменения элементов параметров-массивов внутри функций будут видны и в вызывающей функции.

Далее показана функция (фрагмент программы 2.2), принимающая в качестве параметров три массива. После ее вызова каждый элемент третьего массива будет равен сумме двух соответствующих элементов первого и второго массивов.

```
void add_lists( int first[], int second[], int total[],
                int length )
{
  int count;
  for ( count = 0; count < length; count++ )
    total[count] = first[count] + second[count];
}
```

Фрагмент программы 2.2.

В целях безопасности, для защиты от случайного изменения элементов массива, в описание первых двух параметров функции добавим модификатор типа "`const`":

```
void add_lists( const int fst[], const int snd[], int tot[],
                int len )
{
  int count;
  for ( count = 0; count < len; count++ )
    tot[count] = fst[count] + snd[count];
}
```

Теперь компилятор не будет обрабатывать ни один оператор в определении функции, который пытается модифицировать элементы константных массивов "`fst`" или "`snd`". Фактически, ограничение, накладываемое модификатором "`const`" в данном контексте, в некоторых ситуациях оказывается слишком строгим. Например, компилятор выдаст ошибку при обработке следующих двух функций:

```
void no_effect( const int list[] )
{
  do_nothing( list );
}

void do_nothing( int list[] )
{
  ;
}
```

Фрагмент программы 2.3.

Ошибка компиляции программы 2.3 объясняется тем, что, хотя фактически функция "`do_nothing(...)`" не выполняет никаких действий, но в ее заголовке отсутствует модификатор "`const`". Когда компилятор в теле функции "`no_effect(...)`" встретит вызов функции "`do_nothing(...)`", то он обратится только к заголовку функ-

ции "`do_nothing(...)`", и выдаст ошибку о невозможности передачи константного массива в эту функцию.

3. Сортировка массивов

По отношению к массивам часто возникает задача сортировки по убыванию или возрастанию. Разработано много различных алгоритмов сортировки, например, *пузырьковая сортировка* и *быстрая сортировка*. В этом параграфе кратко рассматривается один из простейших алгоритмов сортировки – *сортировка методом выбора наименьшего элемента*.

Основные действия алгоритма для сортировки массива длиной *L* заключаются в следующем:

Для каждого значения индекса *i* выполнить два действия:

1) Среди элементов с индексами от *i* до (*L*-1) найти элемент с минимальным значением.
2) Обменять значения *i*-го и минимального элементов.

Работу этого алгоритма рассмотрим на примере сортировки массива из 5 целых чисел:

```
int a[5];
```

Значения элементов неотсортированного массива показаны на рис. 4.

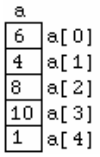

Рис. 4.. Начальное состояние массива.

В процессе сортировки методом выбора наименьшего элемента массив будет последовательно переходить в состояния, показанные на рис. 5 (слева направо). Каждое состояние получается из предыдущего путем перестановки двух элементов, помеченных на рис. 5 кружками.

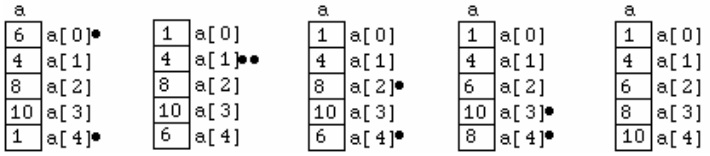

Рис. 5.. Последовательные шаги сортировки массива методом выбора наименьшего элемента.

На Си++ алгоритм сортировки можно реализовать в виде трех функций. Функция высокого уровня будет называться "`selection_sort(...)`" (у нее два параметра – сортируемый массив и его длина). Сначала эта функция вызывает вспомогательную функцию "`minimum_from(array,position,length)`", которая возвращает индекс минимального элемента массива "`array`", расположенного в диапазоне между индексом "`position`" и концом массива. Затем для обмена двух элементов массива вызывается функция "`swap(...)`".

```
void selection_sort( int a[], int length )
```

```
{
  for ( int count = 0; count < length - 1; count++ )
    swap( a[count], a[minimum_from(a,count,length)] );
}

int minimum_from( int a[], int position, int length )
{
  int min_index = position;

  for ( int count = position + 1; count < length; count++ )
    if ( a[count] < a[min_index] )
      min_index = count;

  return min_index;
}

void swap( int& first, int& second )
{
  int temp = first;
  first = second;
  second = temp;
}
```

Фрагмент программы 3.1.

4. Двумерные массивы

Массивы в Си++ могут иметь более одной размерности. В данном параграфе кратко описаны двумерные массивы. Они широко применяются для хранения двумерных структур данных, например, растровых изображений и матриц.

При обработке изображений часто используются битовые маски – вспомогательные двуцветные (черно-белые) изображения, состоящие только из 0 (белая точка) и 1 (черная точка) (или, логических значений "`false`" и "`true`"). Предположим, что требуется маска размером 64x32 пиксела. Для описания соответствующего массива возможны операторы:

```
const int BITMAP_HEIGHT = 64;
const int BITMAP_WIDTH = 32;
bool bitmap[BITMAP_HEIGHT][BITMAP_WIDTH];
```

При обращении к элементам массива "`bitmap`" необходимо указывать два индекса. Например, чтобы изменить значение 2-го пиксела слева в 4-й строке надо записать оператор:

```
bitmap[3][1] = true;
```

Все сказанное во 2-м параграфе относительно одномерных массивов как параметров функций верно и для двумерных массивов, но у них есть еще одна особенность. В прототипах и заголовках функций можно опускать первую размерность многомерного массива-параметра (т.е. записывать пустые квадратные скобки "`[]`"), но все остальные размерности надо обязательно указывать. Далее в качестве примера приведена функция, заполняющая битовую карту черными пикселами.

```
void clear_bitmap( bool bitmap[][BITMAP_WIDTH],
                   int bitmap_height )
{
  for ( int row = 0; row < bitmap_height; row++ )
```

```
    for ( int column = 0; column < BITMAP_WIDTH; column++ )
      bitmap[row][column] = false;
}
```

5. Символьные строки

Во многих уже рассматривавшихся программах для вывода на экран часто использовались символьные строки, например, "`"Введите возраст:"`". В Си++ строки хранятся и обрабатываются в виде символьных массивов, на которые накладывается дополнительное ограничение (см. п.5.1).

5.1 Завершающий нуль-символ '\0'

Символьный массив для хранения строки должен иметь длину на 1 символ больше, чем длина строки. В последнем элементе массива хранится специальный нуль-символ (символ с кодом 0, часто обозначается `'\0'`). Этот служебный символ обозначает конец строки, и это общепринятое правило представления строк, которое соблюдают все библиотечные функции для работы со строками.

На рис. 6 показаны два массива, в которых хранятся символы строки "`"Введите возраст:"`", но из них только массив "`phrase`" является правильным представлением строки. Хотя и "`phrase`", и "`list`" являются символьными массивами, но только "`phrase`" имеет достаточную длину для хранения символьной строки "`"Введите возраст:"`" вместе с завершающим символом "`'\0'`".

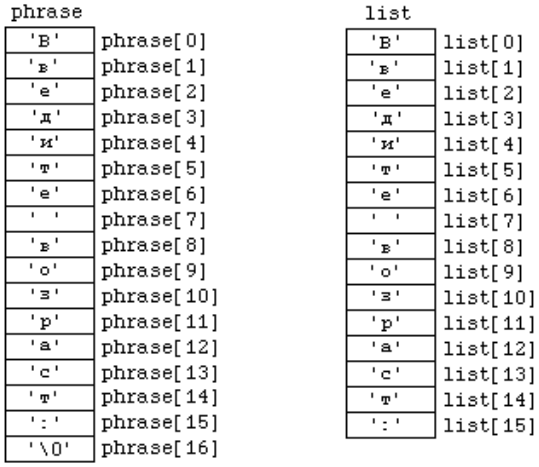

Рис. 6.. Правильное ("phrase") и неправильное ("list") представление символьной строки.

5.2 Объявление и инициализация символьных строк

Символьные строки описываются как обычные массивы:

```
char phrase[17];
```

Массивы можно полностью или частично проинициализировать непосредственно в операторе описания. Список значений элементов массива заключается в фигурные скобки "`{}`" (это правило верно для любых массивов, а не только для символьных). Например, оператор

```
char phrase[17] = { 'В','в','е','д','и','т','е',' ',
                    'в','о','з','р','а','с','т',
                    ':', '\0' };
```

одновременно и объявляет массив "phrase", и присваивает его элементам значения согласно рис. 6. То же самое делает оператор:

```
char phrase[17] = "Введите возраст:";
```

Если не указывать размер "17", то размер массива будет выбран в соответствии с количеством инициализирующих значений (с учетом завершающего нуль-символа). Т.е. строку можно описать с помощью любого из двух следующих операторов:

```
char phrase[] = { 'В','в','е','д','и','т','е',' ',
                  'в','о','з','р','а','с','т',
                  ':', '\0' };
char phrase[] = "Введите возраст:";
```

Очень важно запомнить, что символьные строки хранятся в виде массивов. Поэтому их нельзя приравнивать и сравнивать с помощью операций "=" и "==". Например, нельзя записать оператор присваивания:

```
phrase = "Вы напечатали строку:";
```

Для копирования и сравнения строк следует применять специальные библиотечные функции.

5.3 Библиотечные функции для работы с символьными строками

В стандартном библиотечном файле "cstring.h" хранятся прототипы большо-го количества полезных функций для обработки символьных строк. Будем полагать, что в рассматриваемые далее программы этот заголовочный файл включается с по-мощью директивы препроцессора:

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

Если в программе есть строковая переменная "a_string", то скопировать в нее содержимое другой строки можно с помощью функции "strcpy(...)":

```
strcpy( a_string, "Вы напечатали строку:" );
```

Этот оператор скопирует в массив a_string символы константной строки "Вы напечатали строку:" и добавит в конец массива (т.е. присвоит 21-му элементу) завершающий нуль-символ "'\0'". Вызов оператора

```
strcpy( a_string, another_string );
```

приведет к копированию строки, хранящейся в массиве "another_string", в массив "a_string". При копировании строк важно, чтобы длина массива-приемника ("a_string") была, как минимум, (L+1), где L – длина копируемой строки ("another_string"). В противном случае вызов функции приведет к ошибке выхода за пределы массива, которая не обнаруживается компилятором, но может привести к серьезным сбоям во время выполнения программы.

Для вычисления длины строки предназначена функция "strlen(...)". Вызов "strlen(another_string)" возвращает длину строки "another_string" (без учета нуль-символа).

Функция "strcmp(...)" выполняет сравнение двух переданных ей строк. Если строки одинаковы, то эта функция возвратит 0 (т.е. "false"), а если строки различаются, то функция возвратит ненулевое значение.

Функция конкатенации (соединения) строк "strcat(...)" получает два параметра-строки и копирует вторую строку в конец первой. При работе с "strcat(...)", как и с функцией "strcpy(...)", надо обязательно следить за размером массива-приемника. Си++ не проверяет, достаточен ли размер первого массива, переданного

этой функции, для хранения соединенных строк. При малом размере массива произойдет необнаруживаемая на этапе компиляции ошибка выхода за границы массива.

Работа с библиотечными строковыми функциями продемонстрирована в программе 5.1.

5.4 Чтение символьных строк из потока ввода с помощью функции "getline(...)"

Для ввода строк (например, с клавиатуры) пригоден оператор "`>>`", но его применение ограничено, поскольку этот оператор считает пробелы разделителями. Допустим, в программе содержатся операторы:

```
...
...
cout << "Введите имя: ";
cin >> a_string;
...
...
```

После сеанса работы со следующими экранными сообщениями:

```
...
...
Введите имя: Вася Иванов
...
...
```

Переменной "`a_string`" будет присвоено значение "`"Вася"`", т.к. оператор "`>>`" считает пробел разделителем, который сигнализирует о завершении ввода значения.

Для ввода символьных строк часто более удобной оказывается функция "`getline(...)`", имеющая 2 параметра. Например, оператор:

```
cin.getline(a_string, 80);
```

позволяет получить от пользователя строку с пробелами длиной до 79 символов (последний, 80-й символ строки – служебный нуль-символ).

В программе 5.1 демонстрируется действие функций "`getline(...)`", "`strcmp(...)`", "`strcpy(...)`" и "`strcat(...)`".

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

const int MAXIMUM_LENGTH = 80;

int main()
{
  char first_string[MAXIMUM_LENGTH];
  char second_string[MAXIMUM_LENGTH];

  cout << "Введите первую строку: ";
  cin.getline(first_string,MAXIMUM_LENGTH);
  cout << "Введите вторую строку: ";
  cin.getline(second_string,MAXIMUM_LENGTH);

  cout << "До копирования строки были ";
  if ( strcmp(first_string,second_string) )
    cout << "не ";
  cout << "одинаковыми.\n";
```

```
  strcpy( first_string, second_string );

  cout << "После копирования строки стали ";
  if ( strcmp(first_string,second_string) )
    cout << "не ";
  cout << "одинаковыми.\n";

  strcat( first_string, second_string);

  cout << "После конкатенации первая строка равна: ";
  cout << first_string;

  return 0;
}
```

Программа 5.1.

Программа 5.1 в типичном сеансе работы выводит сообщения:

```
Введите первую строку: Строка 1.
Введите вторую строку: Строка 2.
До копирования строки были не одинаковыми.
После копирования строки стали одинаковыми.
После конкатенации первая строка равна: Строка 2.Строка 2.
```

6. Сводка результатов

Набор однотипных переменных можно представить в виде массива. Массивы можно передавать в качестве параметров внутрь функций. В лекции описан простой алгоритм сортировки массивов методом выбора наименьшего элемента. Кратко рассмотрены приемы работы с двумерными массивами.

Символьные строки в Си++ хранятся в виде символьных массивов, последний символ в которых является служебным нуль-символом. В стандартной библиотеке Си++ есть специальные функции для обработки символьных строк.

7. Упражнения

Упражнение 1

Напишите библиотеку функций для работы с целочисленными массивами. Прототипы функций поместите в заголовочный файл "`IntegerArray.h`", а определения – в файл реализации "`IntegerArray.cpp`". Библиотека должна содержать следующие функции:

- "`input_array(a,n)`" для ввода с клавиатуры первых `n` значений массива `a`.
- "`display_array(a,n)`" для вывода на экран первых `n` значений массива `a`.
- "`copy_array(a1,a2,n)`" для копирования первых `n` элементов массива `a2` в соответствующие элементы массива `a1`.
- "`standard_deviation(a,n)`", которая вычисляет и возвращает среднеквадратическое отклонение первых `n` элементов массива `a`. (Для реализации этой функции может пригодиться функция "`average(a,n)`" из программы 2.1.) Формула для вычисления среднеквадратического отклонения дана в упражнении 3 из лекции 3.

Проверьте разработанные функции с помощью подходящей тестовой программы.

Упражнение 2

Из функции "`selection_sort(...)`", рассматривавшейся в лекции, сделайте функцию сортировки символьной строки "`string_sort(...)`". У этой функции один параметр – строка, которая должны быть отсортирована в алфавитном порядке (точнее, по возрастанию ASCII-кодов, при этом все прописные буквы будут помещены до строчных). Позиция завершающего нуль-символа при сортировке не меняется. Проверьте работу функции с помощью тестовой программы, которая должна выводить на экран сообщения:

```
Введите строку: Вася Иванов
Отсортированная строка равна: ВИааввнося
```

Упражнение 3

Напишите функцию "`no_repetitions(...)`", которая удаляет из строки все повторяющиеся символы. Напишите тестовую программу для проверки этой функции, которая воспроизводит следующий диалог с пользователем:

```
Введите строку: Эта строка содержит повторяющиеся символы
Строка без повторяющихся символов равна: Эта сродежипвющмлы
```

Подсказка: *Как и многие задачи программирования, эту задачу легче решить с помощью процедурной абстракции.*

Упражнение 4

С использованием двумерных массивов напишите функцию (и соответствующую тестовую программу) для умножения целочисленной матрицы размером `mxn` на матрицу размером `nxr`. В тестовой программе для задания значений `m`, `n` и `r` заведите глобальные константы. Ниже приведен пример сообщений, выдаваемых программой:

```
Введите первую матрицу (размер 2x2):
  Введите 2 значения для 1-й строки (через пробелы): 3 4
  Введите 2 значения для 2-й строки (через пробелы): 5 7
Введите вторую матрицу (размер 2x2):
  Введите 2 значения для 1-й строки (через пробелы): 1 1
  Введите 2 значения для 2-й строки (через пробелы): 2 2

           3     4
           5     7
УМНОЖИТЬ
           1     1
           2     2
РАВНО
          11    11
          19    19
```

ЛЕКЦИЯ 7. Указатели

1. Назначение указателей

Язык Си++ унаследовал от языка Си мощные средства для работы с оперативной памятью: динамическое выделение и освобождение блоков памяти, доступ к отдельным ячейкам памяти по их адресам. Эти возможности делают язык Си++, как и Си, удобным для разработки системного программного обеспечения и прикладных программ, в которых применяются динамические структуры данных (т.е. такие, размер которых не известен на этапе компиляции и может меняться во время выполнения программы).

Во всех программах из предыдущих лекций переменные объявлялись так, что компилятор резервировал для каждой из них некоторое количество памяти (в соответствии с типом данных) еще на этапе компиляции. В начале выполнения блока операторов, внутри которого объявлена переменная, автоматически выполняется выделение памяти для этой переменной, а при выходе из блока – освобождение памяти.

В данной лекции подробно рассматривается понятие *указателя*, – средства, которое дает программисту наиболее гибкий способ контроля операций выделения/освобождения памяти во время выполнения программ.

1.1 Объявление указателей

Указателем называется адрес переменной в оперативной памяти. *Переменная указательного типа* (часто она называется переменная-указатель или просто указатель) – это переменная, размер которой достаточен для хранения адреса оперативной памяти. Переменные-указатели объявляются с помощью символа "`*`", который добавляется после названия обычного типа данных. Например, оператор описания (его можно прочитать как "указатель на целочисленную переменную"):

```
int* number_ptr;
```

объявляет переменную-указатель "`number_ptr`", которая может хранить адреса переменных типа "`int`".

Если в программе используется много однотипных указателей, то для их объявления можно ввести новый тип. Это делается с помощью оператора описания нового типа "`typedef`". Например, если записать оператор:

```
typedef int* IntPtrType;
```

то в программе можно будет объявлять сразу несколько переменных-указателей, не применяя для каждой из них символ "`*`":

```
IntPtrType number_ptr1, number_ptr2, number_ptr3;
```

1.2 Применение символов "*" и "&" в операторах присваивания значений указателям

В операторах присваивания допускается совместное использование обычных и указательных переменных одного типа (например, "`int`"). Для получения значения переменной по ее указателю предназначена операция *разыменования указателя* "`*`". У нее есть обратная операция – операция *взятия адреса* "`&`". Упрощенно, операция "`*`" означает "получить значение переменной, расположенной по этому адресу", а операция "`&`" – "получить адрес этой переменной". Для пояснения смысла этих операций далее приводится программа 1.1.

```
#include <iostream.h>

typedef int* IntPtrType;

int main()
{
  IntPtrType ptr_a, ptr_b;
  int num_c = 4, num_d = 7;

  ptr_a = &num_c;     /* СТРОКА 10 */
  ptr_b = ptr_a;      /* СТРОКА 11 */
  cout << *ptr_a << " " << *ptr_b << "\n";
  ptr_b = &num_d;     /* СТРОКА 13 */
  cout << *ptr_a << " " << *ptr_b << "\n";
  *ptr_a = *ptr_b;    /* СТРОКА 15 */
  cout << *ptr_a << " " << *ptr_b << "\n";
  cout << num_c << " " << *&*&*&num_c << "\n";
  return 0;
}
```

Программа 1.1.

Программа 1.1 выдает на экран следующие сообщения:

```
4 4
4 7
7 7
7 7
```

Графически состояние программы 1.1 после выполнения операторов присваивания в 10-11-й строках, 15-й и 19-й показано, соответственно, на рис. 1, 2 и 3.

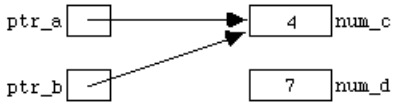

Рис. 1.. Состояние программы 1.1 после выполнения 10-й и 11-й строк.

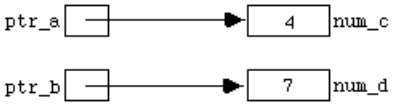

Рис. 2.. Состояние программы 1.1 после выполнения 13-й строки.

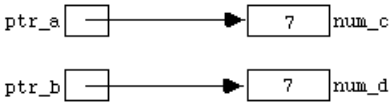

Рис. 3.. Состояние программы 1.1 после выполнения 15-й строки.

Операции "`*`" и "`&`", в некотором смысле, являются взаимно обратными, так что выражение "`*&*&*&num_c`" означает то же самое, что и "`num_c`".

1.3 Операторы "new" и "delete". Константа "NULL"

Рассмотрим оператор присваивания в 10-й строке программы 1.1:

```
ptr_a = &num_c;
```

Можно считать, что после выполнения этого оператора у переменной "`num_c`" появляется еще одно имя – "`*ptr_a`". Часто в программах бывает удобно пользоваться переменными, у которых есть только такие имена – *динамическими переменными*. Независимых имен у них нет. К динамическим переменным можно обращаться только через указатели с помощью операции разыменования (например, "`*ptr_a`" и "`*ptr_b`").

Динамические переменные "создаются" с помощью оператора распределения динамической памяти "`new`", а "уничтожаются" (т.е. занимаемая ими память освобож-

дается для дальнейшего использования) с помощью оператора "`delete`". Действие этих операторов показано в программе 1.2 (она очень похожа на программу 1.1).

```
#include <iostream.h>

typedef int* IntPtrType;

int main()
{
  IntPtrType ptr_a, ptr_b;    /* СТРОКА 7 */

  ptr_a = new int;            /* СТРОКА 9 */
  *ptr_a = 4;                 /* СТРОКА 10 */
  ptr_b = ptr_a;              /* СТРОКА 11 */

  cout << *ptr_a << " " << *ptr_b << "\n";

  ptr_b = new int;            /* СТРОКА 15 */
  *ptr_b = 7;                 /* СТРОКА 16 */

  cout << *ptr_a << " " << *ptr_b << "\n";

  delete ptr_a;
  ptr_a = ptr_b;              /* СТРОКА 21 */

  cout << *ptr_a << " " << *ptr_b << "\n";

  delete ptr_a;               /* СТРОКА 25 */

  return 0;
}
```

Программа 1.2.

Программа 1.2 печатает на экране следующие сообщения:

```
4 4
4 7
7 7
```

На рисунках 4-8 показаны состояния программы 1.2 после выполнения 7-й, 9-11-й, 15-16-й, 21-й и 25-й строк.

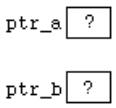

Рис. 4.. Состояние программы 1.2 после выполнения 7-й строки с описаниями переменных.

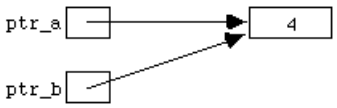

Рис. 5.. Программа 1.2 после выполнения операторов присваивания в 9-й, 10-й и 11-й строках.

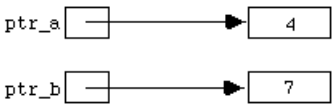

Рис. 6.. Программа 1.2 после выполнения операторов присваивания в 15-й и 16-й строках.

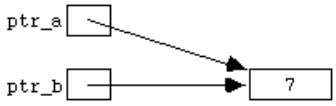

Рис. 7.. Программа 1.2 после выполнения оператора присваивания в 21-й строке.

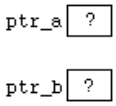

Рис. 8.. Программа 1.2 после выполнения оператора "delete" в 25-й строке.

Состояния на рис. 4 и рис. 8 похожи тем, что значения указателей "ptr_a" и "ptr_b" не определены, т.е. они указывают на несуществующие объекты. Обратите внимание, что указатель "ptr_b" в конце программы оказывается в неопределенном состоянии, хотя при вызове оператора "delete" этот указатель явно не передавался.

Если указатель "ptr" указывает на несуществующий объект, то использование в выражениях значения "*ptr" может привести в непредсказуемым (и часто катастрофическим) результатам. К сожалению, в Си++ нет встроенного механизма проверки несуществующих указателей. Программист может сделать свои программы более безопасными, если всегда будет стараться присваивать несуществующим указателям нулевой адрес (символическое имя нулевого адреса – "NULL"). Для хранения переменных в Си++ нулевой адрес не используется.

Константа "NULL" (целое число 0) описана в библиотечном заголовочном файле "stddef.h". Значение "NULL" можно присвоить любому указателю. Например, в программе 1.3, являющемся усовершенствованным вариантом программы 1.2, для защиты от использования неопределенных указателей "*ptr_a" и "*ptr_b" были добавлены следующие строки:

```
#include <iostream.h>
#include <stddef.h>
  ...
  ...
  delete ptr_a;
  ptr_a = NULL;
  delete ptr_b;
  ptr_b = NULL;
  ...
  ...
  if ( ptr_a != NULL )
    {
    *ptr_a = ...
    ...
    ...
```

Фрагмент программы 1.3.

В случае, если для создания динамической переменной не хватает свободной оперативной памяти, то после вызова "new" Си++ автоматически присвоит соответствующему указателю значение "NULL". В показанном ниже фрагменте программы 1.4 этот факт используется для организации типичной проверки успешного создания динамической переменной.

```
#include <iostream.h>
#include <stdlib.h>   /* "exit()" описана в файле stdlib.h */
#include <stddef.h>
  ...
  ...
```

```
ptr_a = new int;
if ( ptr_a == NULL )
  {
  cout << "Извините, недостаточно оперативной памяти";
  exit(1);
  }
...
...
```

Фрагмент программы 1.4.

Указатели можно передавать в качестве параметров функций. В программе 1.5 проверка на корректность указателя выполняется в отдельной функции, выполняющей создание динамической целочисленной переменной:

```
void assign_new_int( IntPtrType& ptr )
{
  ptr = new int;
  if ( ptr == NULL )
    {
    cout << "Извините, недостаточно оперативной памяти";
    exit(1);
    }
}
```

Фрагмент программы 1.5.

2. Переменные типа "массив". Арифметические операции с указателями

В 6-й лекции были рассмотрены массивы – наборы однотипных переменных. В Си++ понятия массива и указателя тесно связаны. Рассмотрим оператор описания:

```
int hours[6];
```

Этот массив состоит из 6-ти элементов:

```
hours[0]  hours[1]  hours[2]  hours[3]  hours[4]  hours[5]
```

Массивы в Си++ реализованы так, как будто имя массива (например, "`hours`") является указателем. Поэтому, если добавить в программу объявление целочисленного указателя:

```
int* ptr;
```

то ему можно присвоить адрес массива (т.е. адрес первого элемента массива):

```
ptr = hours;
```

После выполнения этого оператора обе переменные – "`ptr`" и "`hours`" – будут указывать на целочисленную переменную, доступную в программе как "`hours[0]`". Фактически, имена "`hours[0]`", "`*hours`" и "`*ptr`" являются тремя различными именами одной и той же переменной. У переменных "`hours[1]`", "`hours[2]`" и т.д. также появляются новые имена:

```
*(hours + 1)   *(hours + 2)   ...
```

или

```
*(ptr + 1)   *(ptr + 2)   ...
```

В данном случае "`+2`" означает "добавить к адресу указателя смещение, соответствующее 2-м целым значениям". Из арифметических операций к указателям часто применяется сложение и вычитание (в том числе операции инкремента и декремента

"++" и "--"), а умножение и деление не используются. Значения однотипных указателей можно вычитать друг из друга.

Главное, что нужно запомнить относительно сложения и вычитания значений из указателя – в выражениях Си++ указывается не число, которое нужно вычесть (или добавить) из адреса, а количество переменных заданного типа, на которые нужно "сместить" адрес.

Арифметические выражения с указателями иногда позволяют более кратко записать обработку массивов. В качестве примера см. функцию для преобразования английской строки в верхний регистр (фрагмент программы 2.1).

```
void ChangeToUpperCase( char phrase[] )
{
  int index = 0;
  while ( phrase[index] != '\0' )
    {
    if ( LowerCase(phrase[index]) )
      ChangeToUpperCase( phrase[index] );
    index++;
    }
}

bool LowerCase( char character )
{
  return ( character >= 'a' && character <= 'z');
}

void ChangeToUpperCase( char& character )
{
  character += 'A' - 'a';
}
```

Фрагмент программы 2.1.

Обратите внимание на полиморфизм функции "`ChangeToUpperCase(...)`" – при обработке вызова компилятор различает две перегруженных функции, т.к. у них разные параметры (у одной – параметр типа "`char`", а у другой – параметр типа "символьный массив"). Имя массива "`phrase`" является переменной-указателем, поэтому функцию с параметром-массивом можно переписать короче, если использовать арифметические выражения с указателями:

```
void ChangeToUpperCase( char* phrase )
{
  while ( *phrase != '\0' )
    {
    if ( LowerCase(*phrase) )
      ChangeToUpperCase(*phrase);
    phrase++;
    }
}
```

Фрагмент программы 2.2.

Эта модификация функции не влияет на остальные части программы – вызовы вариантов функций с параметром-указателем или параметром-массивом записываются одинаково, например:

```
char a_string[] = "Hello World";
...
...
ChangeToUpperCase( a_string );
```

3. Динамические массивы

Правила создания и уничтожения динамических переменных типов "`int`", "`char`", "`double`" и т.п. (см. п.1.3) распространяются и на динамические массивы. По отношению к массивам динамическое распределение памяти особенно полезно, поскольку иногда бывают массивы больших размеров.

Динамический массив из 10-ти целых чисел можно создать следующим образом:

```
int* number_ptr;
number_ptr = new int[10];
```

Переменные-массивы в Си++ являются указателям, поэтому к 10-ти элементам динамического массива допускается обращение:

```
number_ptr[0]   number_ptr[1]   ...   number_ptr[9]
```

или:

```
number_ptr   *(number_ptr + 1)   ...   *(number_ptr + 9)
```

Для уничтожения динамического массива применяется оператор "`delete`" с квадратными скобками "`[]`":

```
delete[] number_ptr;
```

Скобки "`[]`" играют важную роль. Они сообщают оператору, что требуется уничтожить все элементы массива, а не только первый.

Работа с динамическими массивами показана в программе 3.1. В приведенном фрагменте программы у пользователя запрашивается список целых чисел, затем вычисляется и выводится на экран их среднее значение.

```
...
...
int no_of_integers, *number_ptr;

cout << "Введите количество целых чисел в списке: ";
cin >> no_of_integers;

number_ptr = new int[no_of_integers];
if ( number_ptr == NULL )
  {
  cout << "Извините, недостаточно памяти.\n";
  exit(1);
  }

cout << "Наберите " << no_of_integers;
cout << " целых чисел, разделяя их пробелами:\n";
for ( int count = 0; count < no_of_integers; count++ )
  cin >> number_ptr[count];
cout << "Среднее значение: ";
cout << average( number_ptr, no_of_integers );

delete[] number_ptr;
...
```

. . .

Фрагмент программы 3.1.

Динамические массивы, как и обычные, можно передавать в качестве параметров функций. Поэтому в программе 3.1 без изменений будет работать функция "`average()`" из предыдущей лекции (лекция 6, п. 2).

4. Автоматические и динамические переменные

В некоторых случаях без динамических переменных не удается обойтись, но их количество можно уменьшить за счет тщательного проектирования структуры программы и применения процедурной абстракции. Большинство переменных в программах из предыдущих лекций являются *автоматическими* переменными. Они автоматически создаются, когда начинает выполнятся блок (или функция), где эти переменные объявлены. При выходе из блока (или функции) переменные автоматически уничтожаются. Поэтому в хорошо структурированной программе не слишком много операторов для создания и уничтожения переменных.

(***ПРИМЕЧАНИЕ.*** В Си++, кроме автоматических и динамических, существуют *статические* переменные. Для объявления статической переменной в ее операторе описания надо перед названием типа данных добавить служебное слово "`static`". Статические переменные существуют в течение всего времени выполнения программы. Вместо статических переменных чаще используются глобальные константы, которые объявляются в заголовочных файлах или в начале исходных файлов.)

5. Связные списки

В этом параграфе кратко рассматривается одна из разновидностей *абстрактных типов данных* (abstract data type, ADT) – *связный список*. Связный список интересен тем, что при его реализации применяются указатели. О других абстрактных типах данных более подробно будет говориться в последующих лекциях.

Связный список состоит из *узлов*. В каждом узле хранятся некоторые данные и указатель на следующий узел списка (рис. 9). В программе, работающей со списком, обычно есть еще один отдельный указатель, ссылающийся на первый узел списка ("`pointer`" на рис. 9). В указатель последнего узла принято записывать значение "`NULL`".

Размер связных списков, в отличие от массивов, можно изменять динамически (во время выполнения программы можно добавлять/удалять отдельные узлы в любом месте списка).

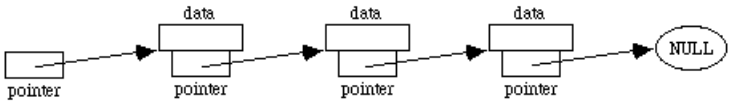

Рис. 9.. Структура связного списка.

Далее будем рассматривать реализацию на Си++ связного списка для хранения символьных строк. Сначала определим на Си++, из чего состоит "узел" нашего списка. Для этого надо объединить строку и указатель в тип данных "узел". Это делается с помощью оператора описания *структуры*. Оператор "`struct`" позволяет создать новый тип данных, в отличие от оператора "`typedef`", который, по существу, предназначен для присвоения нового имени уже существующему типу данных.

Структура – это набор разнотипных переменных (в противоположность массиву, состоящему из элементов одного типа). Применительно к нашей задаче, тип "node" – это структура, состоящая из символьного массива размером MAX_WORD_LENGTH и указателя на такую же структуру:

```
struct node
{
  char word[MAX_WORD_LENGTH];
  node* ptr_to_next_node;
};
```

Обратите внимание на точку с запятой после закрывающей фигурной скобки "}". Слово "struct" является служебным словом Си++ (это аналог "record" в Паскале). Возможно описание узла с применением нового типа данных "указатель на узел":

```
struct node;
typedef node* node_ptr;

struct node
{
  char word[MAX_WORD_LENGTH];
  node_ptr ptr_to_next_node;
};
```

В строке "struct node;" имя "node" является пустым описанием. Оно напоминает прототип (описание) функции – детали структуры "node" определяются в последующих строках. Пустое определение позволяет в следующей строке с помощью оператора "typedef" объявить новый тип данных "указатель на структуру node".

5.1 Операторы "." и "->"

После определения структуры "node (узел)" в программе можно объявлять переменные этого типа:

```
node my_node, my_next_node;
```

Для доступа к полям (внутренним переменным) структуры "my_node" надо пользоваться оператором "." (точка):

```
cin >> my_node.word;
my_node.ptr_to_next_node = &my_next_node;
```

Допустим, что в программе были объявлены указатели на узлы, а сами узлы списка были созданы динамически:

```
node_ptr my_node_ptr, another_node_ptr;
my_node_ptr = new node;
another_node_ptr = new node;
```

В таком случае для доступа к полям узлов можно пользоваться совместно операторами "звездочка" и "точка":

```
cin >> (*my_node_ptr).word;
(*my_node_ptr).ptr_to_next_node = another_node_ptr;
```

Более кратко эти выражения записываются с помощью специального оператора "->", который предназначен для доступа к полям структуры через указатель:

```
cin >> my_node_ptr->word;
my_node_ptr->ptr_to_next_node = &my_next_node;
```

Другими словами, имена "`my_node_ptr->word`" и "`(*my_node_ptr).word`" обозначают одно и то же – поле "`word`" структуры типа "`node`", на которую ссылается указатель "`my_node_ptr `".

5.2 Создание связного списка

Ниже приведена функция создания связного списка для хранения символьных строк, которые пользователь вводит с клавиатуры. Указатель "`a_list`" ссылается на первый узел нового списка (на "голову" списка). Для завершения ввода данных пользователь должен ввести специальный завершающий символ (точку).

```
void assign_list( node_ptr& a_list )
{
  node_ptr current_node, last_node;

  assign_new_node( a_list );
  cout << "Введите первое слово";
  cout << "(или '.' для завершения списка): ";
  cin >> a_list->word;
  if ( !strcmp( ".", a_list->word ) )
    {
    delete a_list;
    a_list = NULL;
    }
  current_node = a_list;                    /* СТРОКА 13 */

  while ( current_node != NULL )
    {
    assign_new_node( last_node );
    cout << "Введите следующее слово";
    cout << "(или '.' для завершения списка): ";
    cin >> last_node->word;
    if ( !strcmp( ".", last_node->word ) )
      {
      delete last_node;
      last_node = NULL;
      }
    current_node->ptr_to_next_node = last_node;
    current_node = last_node;
    }
}
```

Фрагмент программы 5.1.

Функция "`assign_new_node(...)`" для создания нового узла аналогична функции "`assign_new_int(...)`" из программы 1.5.

Порядок действий функции "`assign_list(...)`" поясняется на рис. 10-16. На рис. 10 показано состояние программы после выполнения вызова:

```
assign_new_node( a_list );
```

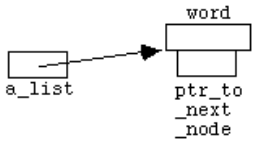

Рис. 10.. Состояние программы 5.1 после создания нового списка.

Предположим, что пользователь напечатал слово "мой". Тогда после 13-й строки программа окажется в состоянии, показанном на рис. 11.

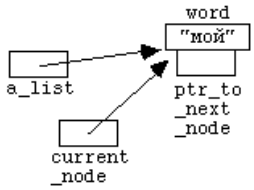

Рис. 11.. Состояние программы после заполнения данными первого узла.

После первой строки в теле цикла "while" программа перейдет в состояние, показанное на рис. 12.

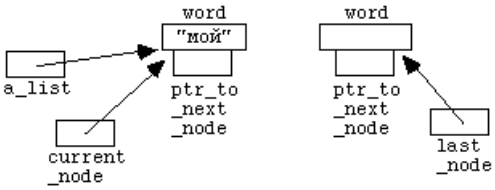

Рис. 12.. В хвост списка был добавлен новый узел.

Далее, если пользователь напечатал слово "список", то после итерации цикла "while" программа будет в состоянии, как на рис. 13.

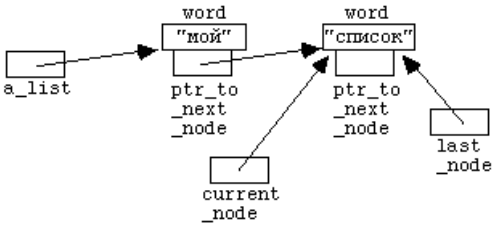

Рис. 13. Последний узел списка заполнен данными и в предыдущий узел помещен соответствующий указатель.

После выполнения первой строки во второй итерации цикла "while" состояние программы см. рис. 14.

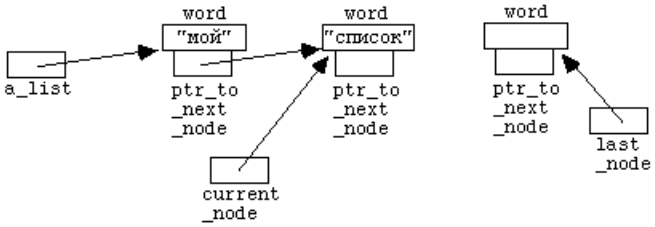

Рис. 14.. В хвост списка был добавлен новый узел.

Допустим, в ответ на следующий запрос пользователь напечатает ".". Тогда после завершения цикла "while" программа будет в состоянии, как на рис. 15.

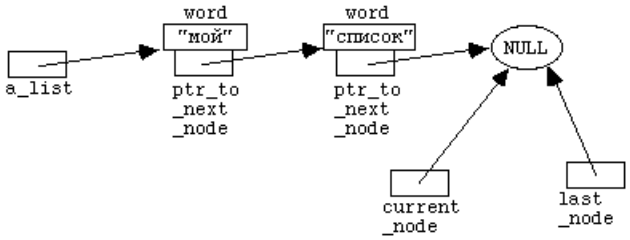

Рис. 15.. Был удален последний узел списка.

Символ "`.`" сигнализирует о том, что пользователь захотел прекратить ввод данных для списка. Поэтому функция "`assign_list(...)`" завершается и при выходе из нее автоматически уничтожаются локальные переменные-указатели "`current_node`" и "`last_node`" (которые были объявлены в теле функции). После возврата из функции состояние программы будет таким, как на рис. 16.

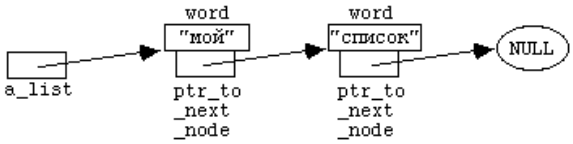

Рис. 16.. Состояние программы 5.1 после выхода из функции ввода списка с клавиатуры.

5.3 Печать связного списка

Напечатать содержимое связного списка на экране несложно. Ниже приведена функция для печати строк, хранящихся в узлах списка:

```
void print_list( node_ptr a_list )
{
  while ( a_list != NULL )
    {
    cout << a_list->word << " ";
    a_list = a_list->ptr_to_next_node;
    }
}
```

Фрагмент программы 5.1.

6. Сводка результатов

В лекции описаны способы динамического распределения оперативной памяти и работа с динамическими переменными с помощью указателей. В Си++ имена массивов можно рассматривать как указатели. Поэтому массивы можно создавать и уничтожать динамически. В выражениях с указателями можно применять арифметические операции сложения и вычитания. В конце лекции кратко рассмотрены основные свойства связного списка и приведен пример реализации этого абстрактного типа данных с помощью указателей.

7. Упражнения

Упражнение 1

Не запуская приведенную далее программу, определите, какие сообщения она выводит на экран. Для проверки своего ответа запустите программу.

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

typedef int* IntPtrType;

int main()
{
  IntPtrType ptr_a, ptr_b, *ptr_c;

  ptr_a = new int;
  *ptr_a = 3;
  ptr_b = ptr_a;
  cout << *ptr_a << " " << *ptr_b << "\n";

  ptr_b = new int;
  *ptr_b = 9;
  cout << *ptr_a << " " << *ptr_b << "\n";

  *ptr_b = *ptr_a;
  cout << *ptr_a << " " << *ptr_b << "\n";

  delete ptr_a;
  ptr_a = ptr_b;
  cout << *ptr_a << " " << *&*&*&*&*ptr_b << "\n";

  ptr_c = &ptr_a;
  cout << *ptr_c << " " << **ptr_c << "\n";

  delete ptr_a;
  ptr_a = NULL;

  return 0;
}
```

Упражнение 2

Напишите логическую функцию с двумя параметрами-строками. Функция должна возвращать "`true`", если ее первый параметр-строка по алфавиту расположена раньше, чем вторая строка. В противном случае функция должна возвращать "`false`". Можете считать, что обе строки содержат только строчные буквы, и в них нет ни пробелов, ни служебных символов. Проверьте работу функции с помощью тестовой программы. После отладки функции напишите ее вариант с применением арифметических выражений с указателями. Убедитесь, что функция работает так же, как и первый вариант.

Упражнение 3

В программу 5.1 добавьте 3 новых функции и измените программу так, чтобы проверить их действие.

- Функция

  `void add_after(node_ptr& list, char a_word[], char word_after[])`

  Вставляет в связный список "`list`" после первого встретившегося узла со словом "`a_word`" новый узел со словом "`word_after`". Если в списке "`list`"

нет узла со словом "a_word", то функция не должна модифицировать список.

- Функция
  ```
  void delete_node(node_ptr& a_list, char a_word[])
  ```
  Удаляет в связном списке "a_list" первый встретившийся узел со словом "a_word".
- Функция
  ```
  void list_selection_sort(node_ptr& a_list)
  ```
  Выполняет сортировку узлов списка в алфавитном порядке (см. Упражнение 2).

Пример экранного ввода/вывода текстовой программы в типичном сеансе работы:

```
1
Введите первое слово (или '.' для завершения списка): это
Введите следующее слово (или '.' для завершения списка): тестовое
Введите следующее слово (или '.' для завершения списка): сообщение
Введите следующее слово (или '.' для завершения списка): из
Введите следующее слово (или '.' для завершения списка): нескольких
Введите следующее слово (или '.' для завершения списка): слов
Введите следующее слово (или '.' для завершения списка): на
Введите следующее слово (или '.' для завершения списка): русском
Введите следующее слово (или '.' для завершения списка): языке
Введите следующее слово (или '.' для завершения списка): .

ТЕКУЩЕЕ СОДЕРЖИМОЕ СПИСКА:
это тестовое сообщение из нескольких слов на русском языке

ПОСЛЕ КАКОГО СЛОВА ВЫ ХОТИТЕ ВСТАВИТЬ НОВОЕ СЛОВО? это
КАКОЕ СЛОВО ВЫ ХОТИТЕ ВСТАВИТЬ? небольшое

ТЕКУЩЕЕ СОДЕРЖИМОЕ СПИСКА:
это небольшое тестовое сообщение из нескольких слов на русском языке

КАКОЕ СЛОВО ВЫ ХОТИТЕ УДАЛИТЬ? тестовое

ТЕКУЩЕЕ СОДЕРЖИМОЕ СПИСКА:
это небольшое сообщение из нескольких слов на русском языке

СОДЕРЖИМОЕ СПИСКА ПОСЛЕ СОРТИРОВКИ:
из на небольшое нескольких русском слов сообщение это языке
```

Подсказки. *Основные черты алгоритма добавления нового узла заключаются в следующем:*

1) *Для поиска и запоминания узла, после которого надо вставить новый узел, применяется дополнительный указатель.*
2) *Для создания нового узла применяется еще один дополнительный указатель.*
3) *После выполнения действий 1) и 2) следует модифицировать соответствующие указатели.*

Возможный алгоритм удаления узла:

1) *Заведите дополнительный указатель на узел перед удаляемым узлом и указатель на удаляемый узел.*
2) *Измените указатель внутри узла, расположенном до удаляемого узла, так, чтобы он указывал на узел, расположенный после удаляемого.*
3) *Удалите лишний узел.*

Для сортировки связного списка несложно адаптировать алгоритм сортировки массива из 6-й лекции.

ЛЕКЦИЯ 8. Рекурсия

1. Понятие рекурсии

В хорошо спроектированной программе на Си++ в определениях функций часто встречаются вызовы других функций этой же программы или библиотеки Си++. Например, в предыдущей (7-й) лекции, в определении функции "`assign_list(...)`" есть вызов "`assign_new_node(...)`". Если в определении функции содержится вызов ее самой, то такая функция называется *рекурсивной*.

Понятие рекурсии хорошо известно в математике и логике. Например, можно привести следующее рекурсивное определение натуральных чисел:

- 1 является натуральным числом
- целое число, следующее за натуральным, есть натуральное число.

В данном контексте понятие рекурсии тесно связано с понятием *математической индукции*. Обратите внимание, что в приведенном определении натуральных чисел есть нерекурсивная часть (базовое утверждение о том, что 1 является натуральным числом).

Еще одним известным примером рекурсивного определения является определение функции факториала "!" для неотрицательных целых чисел:

- 0! = 1
- если `n>0`, то `n! = n*(n-1)!`

В этом определении факториала "`!`" есть базовое утверждение (определение 0!) и рекурсивная часть. Согласно определению, факториал 6 вычисляется следующим образом:

6! = 6*5! = 6*5*4! = 6*5*4*3! = 6*5*4*3*2! = 6*5*4*3*2*1! = 6*5*4*3*2*1*1 = 720

2. Простой пример рекурсии

Рассмотрим рекурсивную функцию "`print_backwards()`" из программы 2.1. Эта функция предназначена для ввода с клавиатуры последовательности символов. Для прекращения ввода пользователь должен напечатать специальный символ (точку). После этого функция печатает введенные символы в обратном порядке.

```
#include<iostream.h>

void print_backwards();

int main()
{
  print_backwards();
  cout << "\n";
  return 0;
}

void print_backwards()
{
  char character;

  cout << "Введите символ (или '.' для завершения ввода): ";
  cin >> character;
  if ( character != '.' )
    {
```

```
        print_backwards();
        cout << character;
        }
}
```

Программа 2.1.

Программа 2.1 печатает на экране подобные сообщения:

```
Введите символ (или '.' для завершения ввода): H
Введите символ (или '.' для завершения ввода): i
Введите символ (или '.' для завершения ввода): .
iH
```

Порядок выполнения функции "`print_backwards()`" подробно описан в следующем параграфе. Пока обратите внимание на то, что вызов "`print_backwards()`" (в ее собственном определении) находится внутри оператора "`if`".

В определениях рекурсивных функций обычно есть некоторый оператор ветвления, у которого, как минимум, одна нерекурсивная ветвь, реализующая "базовое утверждение" определения функции. Если такой ветви нет, то функция будет вызывать себя бесконечно (в действительности, до тех пор, пока не будет занята вся память).

В программе 2.1 базовое утверждение реализовано неявно – это возврат из функции, если был введен символ "`.`" (точка).

3. Как выполняется рекурсивный вызов

Порядок выполнения программы 2.1 легко понять с помощью нескольких схем. Главная функция начинается с вызова "`print_backwards()`". Для выполнения вызова функции в памяти компьютера выделяется некоторое количество памяти (необходимое для запоминания адреса возврата, для создания копий параметров по значению и для передачи параметров-указателей). Свободная область памяти в момент первого вызова "`print_backwards()`" на рис. 1 изображена в виде пустого прямоугольника. Внутри прямоугольника показано содержимое экрана в соответствующие моменты времени (на схемах считается, что направление "сверху-вниз" соответствует увеличению времени, прошедшему с момента запуска программы).

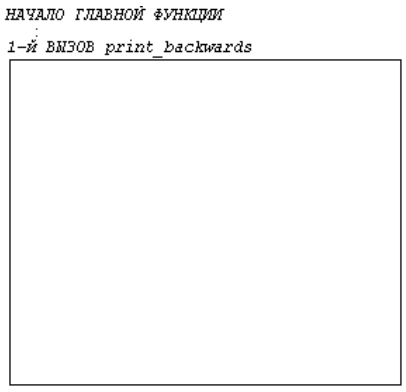

Рис. 1.. Свободная область памяти перед первым вызовом "`print_backwards()`"

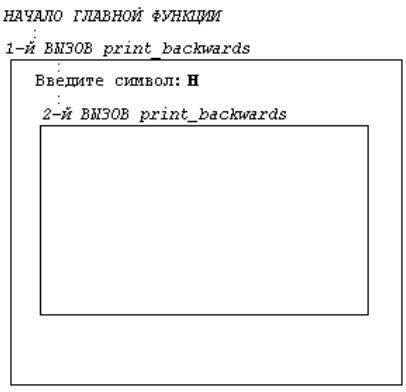

Рис. 2. Свободная область памяти перед вторым вызовом "`print_backwards()`".

Выполнение функции "`print_backwards()`" начинается со ввода символа, а затем происходит второй вызов "`print_backwards()`" (в этот момент программа еще не

начинала обратную печать символов). Для второго вызова также выделяется некоторое количество памяти, поэтому объем свободной памяти уменьшается (рис. 2).

Далее процесс повторяется, но, допустим, при третьем вызове "`print_backwards()`" пользователь ввел завершающий символ (точку). Поэтому после третьего вызова происходит возврат в вызывающую функцию (т.е. во второй экземпляр "`print_backwards()`"), см. рис. 3.

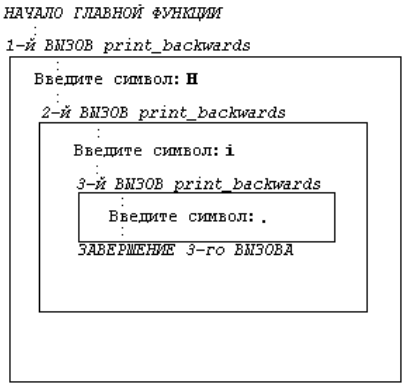

Рис. 3. Возврат из 3-го экземпляра функции "`print_backwards()`" во второй.

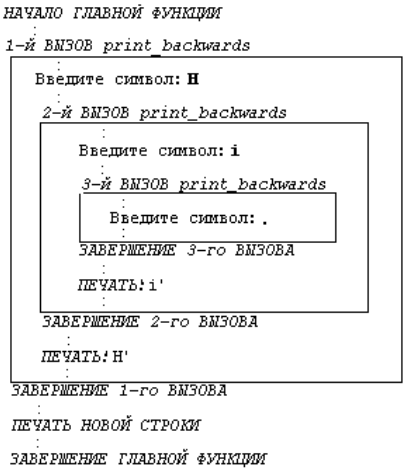

Рис. 4.. Завершение выполнения программы.

Второй экземпляр "`print_backwards()`" завершается, но перед завершением выводит на экран символ "`i`". В свою очередь, первый экземпляр функции перед завершением работы напечатает на экране символ "`H`" (рис. 4).

Для организации вызовов функций и создания автоматических переменных в программах на Си++ отводится специальная область памяти – *стек*. Память, необходимая для очередного вызова функции, выделяется в верхней части стека (в старших адресах). При завершении функции размер стека уменьшается на соответствующую величину. Изменение состояния программного стека для рассмотренного выше примера показано на рис. 5.

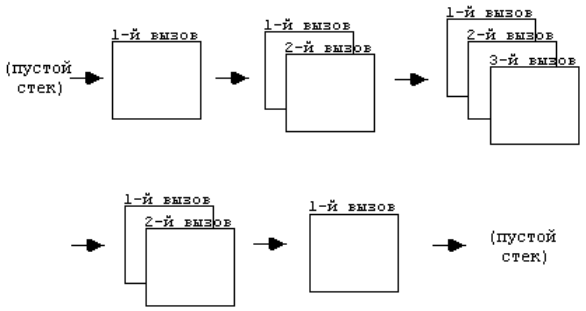

Рис. 5.. Последовательность состояний стека программы 2.1 применительно к тестовому примеру.

Стек в программах на Си++ используется при вызове всех функций, а не только рекурсивных. Стек, как и связный список, является одной из разновидностей абстрактных типов данных. Характерная особенность стека – соответствие принципу "последним прибыл – первым обслужен" (в отличие от стека, *очередь* является примером абстрактного типа данных, действующего по принципу "последним прибыл – последним обслужен").

4. Еще три примера рекурсии

В этом параграфе приводятся три примера рекурсивных функций. В 3-й лекции рассматривалась функция для вычисления факториала целого положительного числа (лекция 3, программа 3.1). Ниже приведен рекурсивный аналог этой функции:

```
int factorial( int number )
{
  if ( number < 0 )
    {
    cout << "\nОшибка - отрицательный аргумент факториала\n";
    exit(1);
    }
  else if ( number == 0 )
    return 1;

  return number*factorial( number - 1 );
}
```

Фрагмент программы 4.1.

В качестве второго примера (фрагмент программы 4.2) дана функция, возводящая свой первый параметр (типа "`double`") в степень с показателем, который передается в качестве второго параметра (неотрицательное целое число).

```
double raised_to_power( double number, int power )
{
  if ( power < 0 )
    {
    cout << "\nОшибка - отрицательный показатель степени\n";
    exit(1);
    }
  else if ( power == 0 )
    return 1.0;

  return number*raised_to_power( number, power - 1 );
}
```

Фрагмент программы 4.2.

В рекурсивных функциях из программ 4.1 и 4.2 особое внимание уделено защите от "бесконечного вызова ". Эта защита основана на проверке параметров, обеспечивающей выполнение вычислений только для корректных значений переданных параметров. Т.о. гарантируется, что будут произведены корректные вычисления и в конечном счете произойдет возврат из функции. При недопустимых значениях параметров выдается сообщение об ошибке и выполняется возврат из функций или завершение работы программы.

Третий пример (фрагмент программы 4.3) – это рекурсивная функция для вычисления суммы первых `n` элементов целочисленного массива "`a[]`".

```
int sum_of( int a[], int n )
{
  if ( n < 1 || n > MAXIMUM_NO_OF_ELEMENTS )
    {
```

```
    cout << "\nОшибка - допускается суммирование от 1 до ";
    cout << MAXIMUM_NO_OF_ELEMENTS << " элементов\n";
    exit(1);
    }
  else if ( n == 1 )
    return a[0];

  return a[n-1]+sum_of( a, n-1 );
}
```

Фрагмент программы 4.3.

5. Рекурсия и циклы

При программировании на Си++ рекурсию применять совсем не обязательно, и на практике она используется довольно редко. Любую рекурсивную функцию можно реализовать итеративно, т.е. с помощью циклов "`for`", "`while`" или "`do...while`". В некотором смысле, какое определение функции выбрать (рекурсивное или итерационное) – вопрос пристрастий программиста. Исходный текст рекурсивных функций иногда легче для понимания, но итерационные функции практически всегда работают быстрее. Далее приводятся итерационные аналоги двух рекурсивных функций, которые ранее рассматривались в данной лекции.

```
void print_backwards()
{
  char chars[MAX_ARRAY_LENGTH];
  int no_of_chars_input = 0;

  do {
    cout << "Введите символ (или '.' для завершения ввода): ";
    cin >> chars[no_of_chars_input];
    no_of_chars_input++;
  } while ( chars[no_of_chars_input - 1] != '.'
            && no_of_chars_input < MAX_ARRAY_LENGTH );

  for ( int count = no_of_chars_input - 2; count >= 0; count-- )
    cout << chars[count];
}
```

Фрагмент программы 2.1b.

```
int factorial( int number )
{
  int product = 1;

  if ( number < 0 )
    {
    cout << "\nОшибка - отрицательный аргумент факториала\n";
    exit(1);
    }
  else if ( number == 0 )
    return 1;

  for ( ; number > 0; number-- )
    product *= number;
  return product;
}
```

Фрагмент программы 4.1b.

Конечно, вопрос о том, какая реализация конкретной функции более понятна, довольно спорный. Обычно в итерационную функцию приходится включать больше локальных переменных, например, в первом примере был добавлен массив "`chars[MAX_ARRAY_LENGTH]`", а во втором примере – целочисленная переменная "`product`". Другими словами, в итеративных функциях память тратится на локальные переменные, а рекурсивных функциях – на организацию вызовов.

6. Рекурсия в структурах данных

На практике рекурсия чаще встречается в структурных типах данных, а не в функциях. Рекурсивная структура данных уже использовалась в предыдущей лекции для определения узла связного списка. В структуре "узел" хранится указатель структуру такого же типа:

```
struct node
{
  char word[MAX_WORD_LENGTH];
  node *ptr_to_next_node;
};
```

В последующих лекциях будут более подробно рассматриваться другие примеры рекурсивных структур данных и особенности их описания на Си++.

7. Рекурсивная реализация алгоритма быстрой сортировки

В завершающей части этой лекции кратко рассмотрим известный пример рекурсивного алгоритма – алгоритм быстрой сортировки *QuickSort*. Этот рекурсивно определенный алгоритм предназначен для упорядочения значений, хранящихся в массиве, по возрастанию или по убыванию.

Предположим, что 11 элементов массива имеют следующие значения (рис. 6).

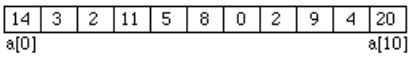

Рис. 6.. Начальное состояние массива.

Идея алгоритма основана на рекурсивном выполнении разбиения массива на две части и переупорядочении элементов в этих частях. Разбиение выполняется путем выбора некоторого элемента, который будем называть *граничным*. После разбиения две части массива обрабатываются так, чтобы в одной части располагались значения, меньшие или равные граничному элементу, а в другой части – только большие или равные.

Например, если выбрать в качестве граничного элемента значение 8, то после переупорядочения массив окажется в состоянии, показанном на рис. 7. Затем тот же самый процесс независимо выполняется для левой и правой частей массива.

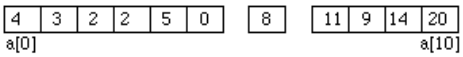

Рис. 7.. Массив после разделения на две части и переупорядочения элементов в них.

Процесс разбиения и переупорядочения частей массива можно реализовать рекурсивно. Индекс граничного элемента вычисляется как индекс среднего элемента:

```
(first + last)/2
```

где "first" и "last" являются индексами первого и последнего элементов обрабатываемой части массива.

Далее подробно рассматривается процедура переупорядочения элементов массива.

Индексы "left_arrow" и "right_arrow" определим как индексы крайних элементов обрабатываемой части массива (которая подвергается разбиению, см. рис. 8).

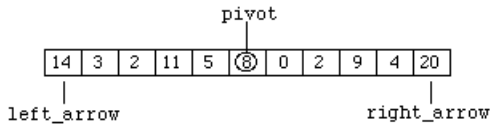

Рис. 8.. Значения индексов "left_arrow" и "right_arrow" перед началом процедуры переупорядочения.

В процессе переупорядочения индексы "left_arrow" и "right_arrow" смещаются в направлении граничного элемента. Так, "right_arrow" перемещается влево, пока значение элемента не окажется меньше или равно граничному значению (рис. 9).

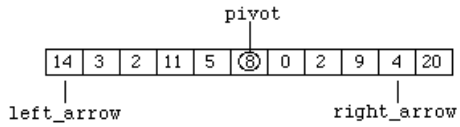

Рис. 9.. Обнаружение левого и правого элементов для обмена.

Аналогично, "left_arrow" смещается вправо, пока не встретится элемент, больший или равный граничному. В нашем примере этот элемент расположен в начале массива (рис. 9). Теперь значения двух найденных элементов массива меняются местами (рис. 10).

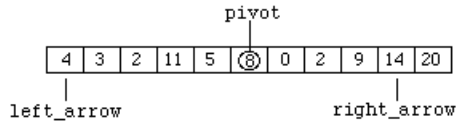

Рис. 10.. Состояние массива после обмена двух элементов.

После обмена двух элементов "right_arrow" продолжает смещаться влево. Смещение прекращается, когда находится очередной элемент, меньший или равный граничному (рис. 11).

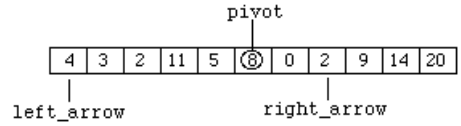

Рис. 11.. Справа обнаружен элемент для обмена.

Индекс "left_arrow" смещается вправо, пока не встретится элемент, требующий перестановки (рис. 12).

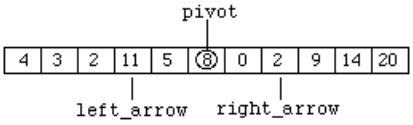

Рис. 12.. Слева обнаружен элемент для обмена.

Значения найденных элементов меняются местами (рис. 13).

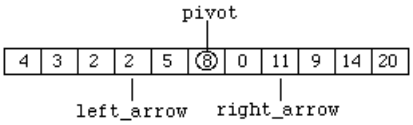

Рис. 13.. Состояние массива после обмена второй пары элементов.

Переупорядочение частей массива прекращается после выполнения условия "`left_arrow > right_arrow`". В состоянии на рис. 13 это условия ложно, поэтому "`right_arrow`" продолжает смещаться влево, пока не окажется в положении, показанном на рис. 14.

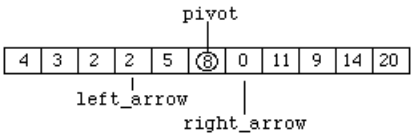

Рис. 14.. Справа обнаружен элемент для обмена.

Перемещение "`left_arrow`" приведет к состоянию, показанному на рис. 15. Поскольку при перемещении вправо надо найти элемент, больший или равный "`pivot`", то "`left_arrow`" прекращает перемещение после достижения граничного элемента.

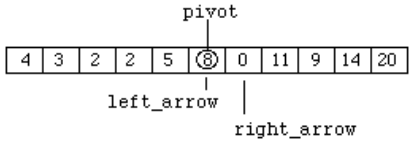

Рис. 15.. Левый элемент для обмена совпадает с граничным.

Теперь выполняется обмен, включающий граничный элемент (такой обмен допустим), и массив переходит в состояние, показанное на рис. 16.

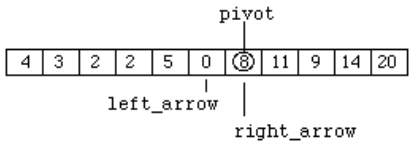

Рис. 16.. Состояние массива после обмена третьей пары элементов.

После обмена элементов "`right_arrow`" перемещается влево, а "`left_arrow`" – вправо (рис. 17).

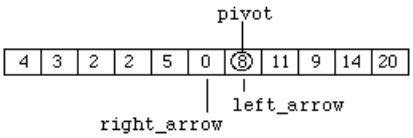

Рис. 17.. Переупорядочение прекращается после прохождения индексами середины массива.

В состоянии, показанном на рис. 17, становится истинным условие завершения процедуры переупорядочения "`left_arrow > right_arrow`". Поэтому первую процедуру разбиения и переупорядочения массива можно считать выполненной.

Ниже приведена функция на Си++, в которой рекурсивно реализован алгоритм сортировки QuickSort:

```
void quick_sort( int list[], int left, int right )
{
  int pivot, left_arrow, right_arrow;
```

```
  left_arrow = left;
  right_arrow = right;
  pivot = list[(left + right)/2];

  do {
    while ( list[right_arrow] > pivot )
      right_arrow--;
    while ( list[left_arrow] < pivot )
      left_arrow++;
    if ( left_arrow <= right_arrow )
      {
      swap( list[left_arrow], list[right_arrow] );
      left_arrow++;
      right_arrow--;
      }
  } while ( right_arrow >= left_arrow );

  if ( left < right_arrow )
    quick_sort( list, left, right_arrow );
  if ( left_arrow < right )
    quick_sort( list, left_arrow, right );
}
```

Программа 7.1.

8. Сводка результатов

На Си++ можно писать рекурсивные функции. В лекции кратко описан порядок выполнения рекурсивных функций и назначение стека. Для любой рекурсивной функции можно разработать итерационный аналог. В лекции указаны преимущества и недостатки рекурсивной реализации. В качестве примера приведен известный алгоритм быстрой сортировки QuickSort и его рекурсивная реализация на Си++.

9. Упражнения

Упражнение 1

Последовательность чисел Фибоначчи может быть определена следующим образом:

- $a(1) = 1$
- $a(2) = 1$
- для всех $n>2$ $a(n) = a(n\text{-}1) + a(n\text{-}2)$

Это определение позволяет сгенерировать такую последовательность:

1, 1, 2, 3, 5, 8, 13, 21, ...

Напишите на Си++ функцию "`fibonacci(...)`", которая вычисляет числа Фибоначчи по их порядковому номеру, например, `fibonacci(7)==13`.

Упражнение 2

Для выполнения этого упражнения используйте программу 5.1 из 7-й лекции.

а) Напишите две рекурсивных функции "`print_list_forwards(...)`" и "`print_list_backwards(...)`", которые должны печатать на экране содержимое связного списка, рассмотренного в 7-й лекции, соответственного в прямом и

обратном порядке. (Функция "`print_list_forwards(...)`" должна вести себя точно так же, как и функция "`print_list(...)`" из 7-й лекции). Измените программу 5.1 из 7-й лекции так, чтобы проверить свои функции. Ваша программа должна выдавать на экран следующие сообщения:

```
Введите первое слово (или '.' для завершения списка): это
Введите следующее слово (или '.' для завершения списка): короткое
Введите следующее слово (или '.' для завершения списка): тестовое
Введите следующее слово (или '.' для завершения списка): сообщение
Введите следующее слово (или '.' для завершения списка): .

ПЕЧАТЬ СОДЕРЖИМОГО СПИСКА В ПРЯМОМ ПОРЯДКЕ:
это короткое тестовое сообщение

ПЕЧАТЬ СОДЕРЖИМОГО СПИСКА В ОБРАТНОМ ПОРЯДКЕ:
сообщение тестовое короткое это
```

Подсказка: *при разработке функции "*`print_list_backwards()`*" еще раз посмотрите на программу 2.1.*

б) Напишите и протестируйте итерационный (т.е. нерекурсивный) вариант функции "`print_list_backwards(...)`". (Какой вариант функции вам показался проще в разработке?)

Упражнение 3

Даны два положительных целых числа m и n таких, что $m<n$. Известно, что наибольший общий делитель чисел m и n совпадает с наибольшим общим делителем чисел m и $(n-m)$. С учетом этого свойства напишите рекурсивный вариант функции "`greatest_common_divisor(...)`", которая принимает два положительных целых параметра и возвращает их наибольший общий делитель. Проверьте свою функцию с помощью подходящей тестовой программы.

Упражнение 4

Бинарный поиск – это рекурсивно определенный алгоритм поиска заданного значения в отсортированном массиве. Он особенно эффективен при обработке больших массивов, т.к. не требует последовательного просмотра всех элементов массива.

Основные действия алгоритма заключаются в следующем. Предположим, требуется определить в массиве `a[]` индекс элемента со значением `v`. Массив `a[]` предварительно отсортирован по возрастанию. Сначала проверим значение среднего элемента массива `a[]`. Если это значение равно `v`, то работа алгоритма завершается, и функция возвращает в качестве результата индекс среднего элемента. Если же оказывается, что средний элемент меньше, чем `v`, то требуется выполнить поиск только во второй половине массива. При этом повторяется процедура деления массива пополам. Аналогично, если среднее значение оказывается больше, чем `v`, то требуется продолжить поиск только в первой половине массива.

Работу этого алгоритма легко понять, если мысленно применить его к поиску заданного слова в словаре. Применяя бинарный поиск, вы должны начать с середины словаря и затем ограничить поиск половиной словаря или до, или после средней страницы. Затем деление просматриваемых страниц словаря повторяется и т.д..

Для выполнения бинарного поиска напишите функцию с прототипом:

```
int binary_search( int value, int list[], int first, int last );
```

Эта функция должна искать значение "value" в массиве "list[]" в интервале индексов от "list[first]" до "list[last]". Если значение "value" обнаружено в заданном диапазоне, то функция должна возвратить индекс соответствующего элемента массива "list[]". В противном случае функция должна вернуть -1. Например, для массива

```
list = { 2, 2, 3, 5, 8, 14, 16, 22, 22, 24, 30 }
```

функция

```
binary_search( 24, list, 0, 10 );
```

должна вернуть значение 9, функция

```
binary_search( 24, list, 2, 6 );
```

должна вернуть -1, а функция

```
binary_search( 22, list, 0, 10 );
```

должна вернуть значение 7 или 8.

Проверьте свою функцию с помощью подходящей тестовой программы. (Для этого вы можете модифицировать программу 7.1).

ЛЕКЦИЯ 9. Составные типы данных

1. Назначение составных типов данных

В программах часто приходится обрабатывать информацию, описывающую более сложные объекты, чем числа и символы. Например, в библиотечной базе данных требуется обрабатывать данные об объектах "Книги", в системе кадрового учета – "Сотрудники", "Отделы" и т.п. В зависимости от решаемой задачи, программист определяет, какие характеристики (свойства) объектов нужно учитывать. Для хранения этих свойств в программе выделяются переменные подходящих типов, например, символьный массив для имени сотрудника и вещественное число для его зарплаты. Оказывается, если свойства объектов хранить в отдельных переменных, то при большом количестве различных свойств и при наличии большого количества экземпляров однотипных объектов программисту становится довольно сложно следить за корректным использованием переменных.

В Си++ для построения новых типов данных используются классы, объединяющие в себе и свойства объектов, и действия (алгоритмы), которые эти объекты способны выполнять. Но, поскольку проблема обработки сложных типов данных стала актуальной еще до распространения объектно-ориентированного программирования, уже в язык Си было введено понятие структуры. ***Структура*** *– это составной тип данных, который получается путем объединения компонент, принадлежащих к другим типам данных (возможно, тоже составным).* Впоследствии в Си++ понятие структуры было расширено до класса.

Пример структур в математике – комплексные числа, состоящие из двух вещественных чисел, и координаты точек, состоящие из двух или более вещественных чисел в зависимости от размерности координатного пространства. Пример из обработки данных – это описание людей с помощью нескольких существенных характеристик, таких, как имя и фамилия, дата рождения, пол и семейное положение.

Понятие структуры встречается во многих языках программирования и в области баз данных, только вместо термина "структура", специфичного для Си++, в других языках обычно применяется термин "запись" (подразумевается, что переменные составных типов предназначены для записи существенных характеристик объектов, обрабатываемых в программе, например, людей или материальных предметов).

2. Описание и инициализация структур

Структура предназначена для объединения нескольких переменных в один тип данных. Сначала тип структуры надо описать, а затем можно создавать переменные этого нового типа. Описание типа структуры "`T`" имеет следующий синтаксис:

```
struct T {
  T1 var1;
  T2 var2;
  ...
  T3 var3;
};
```

где "`T1`", "`T2`", "`T3`" – имена встроенных типов данных ("`int`", "`char`" и др.) или других составных типов. "`var1`", "`var2`", "`var3`" – это имена внутренних переменных структуры (они также называются *компонентами*, *полями* или *членами* структуры).

Далее приведены три примера описания структур для представления комплексных чисел, дат и сведений о людях:

```
struct Complex {
  double re;
  double im;
};

struct Date {
  int day;
  int month;
  int year;
};

struct Person {
  char name[50];
  Date birthdate;
  double salary;
};
```

Обратите внимание на точку с запятой в конце описания типа структуры. Это одно из очень немногих мест в Си++, где необходимо ставить точку с запятой после фигурной скобки. На примере типа "`Person`" видно, что компоненты структуры могут, в свою очередь, тоже быть составными. Переменные типа структуры описываются аналогично переменным встроенных типов:

```
Complex z;
Date d;
Person p;
```

В операторе описания переменные можно инициализировать путем перечисления значений компонент в фигурных скобках (аналогично инициализации массивов):

```
Complex z = { 1.6, 0.5 };
Date d = { 1, 4, 2001 };
Person p = { "Сидоров", {10, 3, 1978}, 1500.48 };
```

Для обращения к отдельным компонентам структуры применяется операция "`.`" (точка). Сначала указывается имя переменной-структуры, а затем – имя компоненты. Например:

```
z.im = 3.2;
d.month = 7;
p.birthdate.year = 1980;
p.salary = 2000.00;
```

Важно отметить, что не все возможные комбинации значений компонент структуры могут иметь смысл применительно к конкретной задаче. Например, тип "`Date`", определенный выше, включает значения {50, 5, 1973} и {100, 22, 1815}, хотя дней с такими датами не существует. Т.о., определение этого типа не отражает реального положения вещей, но все же оно достаточно близко к практическим целям. Ответственность за то, чтобы при выполнении программы не возникали подобные бессмысленные значения, возлагается на программиста.

В программе 2.1 демонстрируется, как в исходном тексте на Си++ располагаются описание типа структуры, объявление переменных и обращение к компонентам структуры.

```
struct SimpleStructure {
  char c;
  int i;
  float f;
  double d;
};

void main()
{
  SimpleStructure s1, s2;

  s1.c = 'a';
  s1.i = 1;
  s1.f = 3.14f;       // Буква 'f' в конце вещественной константы
                      // означает, что это константа типа float,
                      // а не double
  s1.d = 0.00093;

  s2.c = 'b';
  s2.i = 2;
  s2.f = 6.28f;
  s2.d = 0.15;
}
```

Программа 2.1.

В программе 2.1 в функции "`main()`" создаются две переменные типа "`SimpleStructure`" с именами "`s1`" и "`s2`". У каждой из этих переменных есть собственный набор компонент с именами "`c`", "`i`", "`f`", и "`d`". Т.е. "`s1`" и "`s2`" являются наборами независимых переменных. Для выбора компонент внутри "`s1`" или "`s2`" применяется операция "`.`" (точка). Подобная запись применялась в предыдущих лекциях для обращения к функциям-членам объектов "`cin`" и "`cout`". Обращения к компонентам классов в объектно-ориентированном Си++ очень похожи на обращения к компонентам структур.

Переменные типа структуры можно присваивать, передавать, как параметры функции, и возвращать из функции в качестве результата. Например:

```
Person current;
Person set_current_person( Person& p )
{
  Person prev = current;
  current = p;
  return prev;
}
```

Остальные операции, такие, как сравнение ("==" и "`!=`"), для структур по умолчанию не определены, но программист может их определить при необходимости.

Имя типа структуры можно использовать еще до того, как этот тип будет определен, вплоть до момента, когда потребуется, чтобы стал известен размер структуры. Например, допустимы следующие прототипы функций:

```
struct S; // S - имя некоторого типа
S f();
void g(S v1);
```

Но эти функции нельзя вызывать, если тип "S" не определен:

```
void h()
{
  S a;     // ошибка: S не объявлено
  f();     // ошибка: S не объявлено
  g(a);    // ошибка: S не объявлено
}
```

3. Доступ к компонентам структуры через указатель

Во всех предыдущих примерах компоненты структур использовались в выражениях подобно обычным переменным. Аналогично обычным переменным, можно создавать указатели на переменные-структуры. Для доступа к компонентам структуры через указатель применяется операция "->". Например:

```
void print_person( Person* p )
{
  cout << p->name << '\n';
  cout << p->birthdate.day << '\n';
  cout << p->birthdate.month << '\n';
  cout << p->birthdate.year << '\n';
  cout << p->salary << '\n\n';
}
```

Функцию "print_person()" можно переписать в эквивалентном виде с помощью операции разыменования указателя "*" и операции доступа к компонентам структуры ".". Обратите внимание на скобки в записи "(*p).", которые необходимы, поскольку приоритет операции "." выше, чем у "*":

```
void print_person( Person* p )
{
  cout << (*p).name << '\n';
  cout << (*p).birthdate.day << '\n';
  cout << (*p).birthdate.month << '\n';
  cout << (*p).birthdate.year << '\n';
  cout << (*p).salary << '\n\n';
}
```

Использование указателей на структуры показано в программе 3.1. В функции "main()" этой программы указателю "sp" сначала присваивается адрес "s1", и затем с помощью операции "->" компонентам "s1" присваиваются начальные значения. Затем указателю "sp" присваивается адрес "s2", и компоненты этой структуры также инициализируются. Возможность динамического перенаправления на различные объекты является одним из важнейших свойств указателей, которые, в частности, полезны для реализации динамических структур данных (например, связного списка или стека).

```
struct ExStruct {
```

```
  char c;
  int i;
  float f;
  double d;
};

void main()
{
  ExStruct s1, s2;
  ExStruct* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14f;
  sp->d = 0.00093;
  sp = &s2;
  sp->c = 'b';
  sp->i = 2;
  sp->f = 6.28f;
  sp->d = 2.5;
}
```

Программа 3.1.

В 7-й лекции (для реализации связного списка) и в 8-й лекции уже рассматривалось понятие рекурсивных структур данных. Для создания в структуре ссылки на структуру такого же типа необходимо пользоваться указателем. В программе 3.2 создаются две структуры, содержащие ссылки друг на друга.

```
struct SelfReferential {
  int i;
  SelfReferential* sr;
};

void main()
{
  SelfReferential sr1, sr2;
  sr1.sr = &sr2;
  sr2.sr = &sr1;
  sr1.i = 47;
  sr2.i = 1024;
}
```

Программа 3.2.

Описание без указателя является недопустимым, т.к. в нем при описании компоненты структуры тип структуры используется в тот момент, когда этот тип еще не определен полностью:

```
struct SelfReferential {
  int i;
  SelfReferential sr;   // Недопустимое описание компоненты
};
```

4. Массивы и структуры

У массива и структуры есть общее свойство: оба этих типа данных являются типами с произвольным доступом. Но структура более универсальна, поскольку не

требуется, чтобы типы всех ее компонент были одинаковы. С другой стороны, в некоторых случаях массив предоставляет бóльшие возможности, т.к. индексы его элементов могут вычисляться, а имена компонент структуры – это фиксированные идентификаторы, задаваемые в описании типа.

Массивы и структуры могут комбинироваться различными способами. Например, *i*-я компонента массива "a", который является компонентой структуры "r", обозначается как

```
r.a[i]
```

Напротив, компонента с именем "s", входящая в *i*-ю компоненту-структуру массива структур "a" обозначается как

```
a[i].s
```

В качестве примера далее приведено описание переменной "screen", которая является двумерным массивом структур типа "Cell". Этот массив предназначен для хранения содержимого текстового экрана размером 80x25 знакомест:

```
struct Cell {
  unsigned char character; // Символ
  int foreground;          // Цвет символа
  int background;          // Цвет фона
  bool blink;              // Мигание включено/выключено
};

void main()
{
  Cell screen[25][80];
  ...
}
```

5. Перегрузка операторов

В 3-й лекции (п. 4) были описаны средства перегрузки функций. Си++ допускает перегрузку не только функций, но и операторов, таких, как "+", "-", "*" (и большинство других – "+=", "->" и даже "()"). Средства перегрузки операторов полезны тогда, когда желательно, чтобы пользовательские типы данных в исходном тексте программ выглядели в выражениях подобно встроенным типам Си++. Поясним это на нескольких примерах.

Для сложения комплексных чисел (описание структуры см. п.2) можно применить функцию:

```
Complex C_add( const Complex& x, const Complex& y )
{
  Complex t;
  t.re = x.re + y.re;
  t.im = x.im + y.im;
  return t;
}
```

Параметры функции "C_add(...)" передаются по ссылке, и, кроме того, описаны как константы, чтобы запретить изменения параметров внутри функции. Передача по ссылке обеспечивает эффективный вызов функции, без копирования параметров, а константное описание защищает параметры от изменений.

Допустим, в программе реализована также функция "C_mult(...)" для умножения двух комплексных чисел. Ниже приведен пример использования этих функций:

```
Complex u, v, w, z, t;
...
t = C_add( u, v );
w = C_mult( z, t );
```

Конечно, более естественной является запись:

```
Complex u, v, w, z;
...
w = z * ( u + v );
```

Тип "Complex" является типом, введенным пользователем, и, естественно, в языке Си++ арифметические операторы для этого типа не определены. Поэтому для естественной записи выражений с числами в виде структур "Complex" надо перегрузить операторы сложения и умножения:

```
Complex operator +( const Complex& x, const Complex& y )
{
  Complex t;
  t.re = x.re + y.re;
  t.im = x.im + y.im;
  return t;
}

Complex operator *( const Complex& x, const Complex& y )
{
  Complex t;
  t.re = x.re*y.re - x.im*y.im;
  t.im = x.re*y.im + x.im*y.re;
  return t;
}
```

Правила размещения перегруженных операторов в исходном тексте программ аналогичны правилам размещения функций: прототипы отдельно, определения отдельно (возможно, прототипы операторов расположены в заголовочном файле, а определения – в файле реализации).

Аналогично арифметическим операторам, в Си++ допускается перегружать операторы потокового ввода/вывода ">>" и "<<". Например, вывести комплексное число на экран можно так:

```
void main()
{
  Complex u, v;
  u.re = 2.1; u.im = 3.6;
  v.re = 6.5; v.im = 7.8;
  cout << "Числа равны (" << u.re << " + " << u.im << "i) и ";
  cout << " (" << v.re << " + " << v.im << "i).\n";
}
```

В результате выполнения этой программы на экране будет напечатана строка:

```
Числа равны (2.1 + 3.6i) и (6.5 + 7.8i).
```

Для упрощения записи операции печати комплексного числа на экране можно перегрузить оператор вывода в поток:

```
ostream& operator <<( ostream& s, const Complex& x )
{
  s << "(" << x.re << " + " << x.im << "i)";
  return s;
}

void main()
{
  Complex u, v;
  u.re = 2.1; u.im = 3.6;
  v.re = 6.5; v.im = 7.8;
  cout << "Числа равны " << u << " и " << v << ".\n";
}
```

В определении оператора фигурирует тип "`ostream`". Это класс "поток вывода", объектом которого является стандартный поток вывода на экран "`cout`". Для потокового ввода/вывода в файл необходимо перегружать операторы применительно к классам "`ofstream`" и "`ifstream`" (см. 4-ю лекцию).

6. Применение структур для реализации стека

Абстрактный тип данных "стек" кратко рассматривался в 8-й лекции (компилятор использует стек для организации вызовов функций). Стек встречается не только в компиляторах, но и во многих численных алгоритмах. В данном параграфе рассматриваются два варианта реализации стека на Си++ с помощью структур.

Среди значений английского слова "stack" есть значения "пачка" и "стопа". Абстрактный тип данных, соответствующий принципу "последним прибыл – первым обслужен" получил свое название из-за своего сходства со стопкой тарелок (рис. 1).

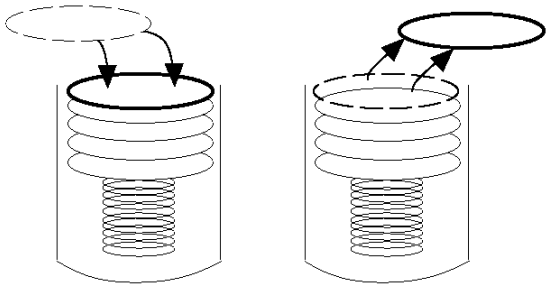

Рис. 1..Поведение стека при записи и чтении элементов.

Основные свойства стека:

1) Элементы добавляются или удаляются только сверху стека.
2) Для добавления элемента применяется функция "`push()`".
3) Для удаления элемента применяется функция "`pop()`".

6.1 Реализация стека на основе статического массива

В качестве задачи поставим разработку стека для хранения вещественных чисел. Свойства стека очень просты, поэтому для работы со стеком, например, из 20-ти вещественных чисел достаточно несложно написать программу, похожую на программу 6.1.

```
#include <iostream.h>

struct Stack {
  double v[20];
  int top;
};

void push( Stack& s, double val )
{
  s.v[s.top] = val;
  s.top++;
}

double pop( Stack& s )
{
  return s.v[--(s.top)];
}

void init( Stack& s )
{
  s.top = 0;
}

bool full( Stack& s )
{
  return s.top >= 20;
}

void main()
{
  Stack s;
  init( s );                // Инициализация стека
  push( s, 2.31);           // Помещение в стек первого элемента
  push( s, 1.19 );          // Помещение в стек второго элемента
  cout << pop( s ) << '\n';      // Извлечение верхнего элемента
                             // и вывод его на экран
  push( s, 6.7 );           // Помещение в стек элемента
  push( s, pop(s)+pop(s) );      // Замена двух верхних элементов
                              // стека их суммой
}
```

Программа 6.1.

В программе 6.1 стек реализован на базе статического массива, один из элементов которого играет роль верхушки стека. Индекс этого элемента хранится в компоненте `top`. (рис. 2).

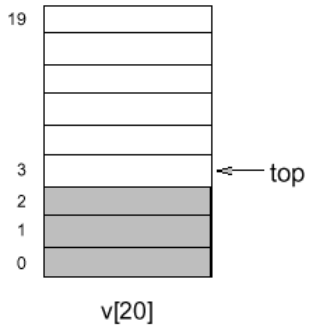

Рис. 2..Стек на основе массива с фиксацией верхушки стека. Серым цветом обозначены помещенные в стек элементы.

6.2 Недостатки структуры данных Stack

Приведенному в п. 6.1 простейшему варианту реализации стека (в виде структуры "`Stack`") присущи ряд недостатков, которые можно разбить на две группы:

1) *Малая гибкость применения*

- У стека ограниченный размер (20 элементов).
- В стеке могут храниться только значения типа `double`.
- Имена функций наподобие "`full()`" и "`init()`" очень распространены и может возникнуть конфликт этих имен с именами функций из других библиотек.

2) *Безопасность использования стека*

- Внутренние переменные стека не защищены от изменений извне. Поэтому стек легко повредить путем изменения значений компонент структуры. Это может привести к сложно обнаружимым ошибкам.
- Назначение компонент стека тесно связано с особенностями реализации обслуживающих функций ("`init()`", "`push()`", "`pop()`" и др.).
- В обслуживающих функциях не предусмотрена обработка типичных ошибок: переполнение стека, попытка извлечения элемента из пустого стека, повторная инициализация стека, нет способа проверки состояния стека (поврежден/не поврежден).
- Присвоение структур (напр., "`A=B`") приводит к возникновению висячих указателей, т.к. присвоение компонент-массивов означает присвоение указателей, а не копирование отдельных элементов.

Перечисленные недостатки показаны во фрагменте программы 6.2.

```
void Stack_Print( Stack& A )
{
  if ( A.top = 0 )    // Ошибка: присваивание вместо сравнения
    cout << "Стек пуст.\n";
  else
    cout << "Стек не пуст.\n";
}

void main()
{
  Stack A;
  double x;
  push( A, 3.141 );   // Стек A не был проинициализирован
```

```
  init( A );
  x = pop( A );       // Ошибка: A в данный момент пуст,
                      // поэтому стек окажется в поврежденном
                      // состоянии, а значение x не определено
  A.v[3] = 2.13;           // Так помещать элементы в стек нельзя
  A.top = -42;        // Теперь стек окажется в логически
                       // некорректном состоянии
  Stack C, D;
  push( C, 0.9 );
  push( C, 6.1 );
  init( D );
  D = C;              // Такое присвоение допускается по
                       // правилам языка, но не обеспечивает
                       // копирования элементов стека
  init( C );          // Эта инициализация очищает содержимое
                      // и C, и D, поскольку они обращаются
                       // к одному массиву
}
```

Фрагмент программы 6.2.

6.3 Реализация стека с использованием динамической памяти

Для реализации стека можно предложить структуру, позволяющую создать стек произвольного размера и работать с элементами стека через указатели. Далее приведен один из вариантов этой структуры:

```
struct DStack {
  double* bottom;
  double* top;
  int size;           // Это максимальный размер стека,
};                    // а не количество элементов в нем
```

Массив для хранения элементов стека должен создаваться динамически на этапе инициализации. Два внутренних указателя стека ссылаются на начало массива ("`bottom`") и на элемент-верхушку стека ("`top`"). Устройство стека на базе динамического массива показана на рис. 3.

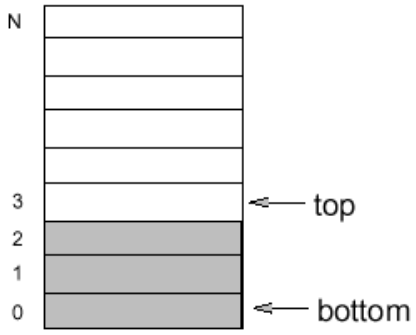

Рис. 3..Стек с хранением элементов в динамической памяти.

Для работы со стеком целесообразно сделать несколько обслуживающих функций:

1) *Инициализация стека.*

```
bool DStack_init( DStack& s, int N );
```

Эта функция динамически создает целочисленный массив размером N элементов и сохраняет его указатель в компонентах структуры "DStack". Если операция инициализации прошла успешно, то функция возвращает "true", а если памяти недостаточно – то "false".

2) *Добавление элемента в стек.*
```
bool DStack_push( DStack& s, double val );
```
Помещает элемент на верх стека. Возвращает "true", если это удалось сделать, или "false", если стек целиком заполнен.

3) *Извлечение верхнего элемента.*
```
double DStack_pop( DStack& s );
```
Возвращает верхний элемент из стека или вохвращает 0.0 (или некоторое другое специально оговоренное значение), если стек пуст.

4) *Проверка на пустоту.*
```
bool DStack_empty( Dstack& s );
```
Возвращает "true", если стек пуст, или "false" в противном случае.

5) *Удаление стека.*
```
void DStack_free();
```
Освобождает динамическую память, занятую при создании стека.

По сравнению со стеком из п.6.1, в описанном варианте реализации достигается ряд улучшений:

1) Есть возможность создания стеков разного размера.
2) В обслуживающих функциях предусмотрена обработка элементарных ошибок переполнения и обращения к пустому стеку.
3) Имена обслуживающих функций, начинающиеся с "DStack_", менее вероятно обнаружить в других библиотеках.

К сожалению, опасность несанкционированного изменения внутренних переменных стека сохраняется и в этой реализации.

7. Сводка результатов

Составной тип данных позволяет объединить под одним именем несколько переменных уже существующих типов. В Си++ для описания составного типа применяется служебное слово "struct". Структуры могут быть вложенными. Для обращения к компонентам структуры используются операции точка "." или "->" (если обращение выполняется через указатель на структуру).

Внутрь функций структуры чаще передаются по ссылке. Если параметры-структуры внутри функции заведомо не изменяются, то такие параметры обычно описываются как константные параметры.

Можно создавать и использовать массивы структур, или использовать массивы в качестве компонент структур. Для более краткой и естественной записи выражений со структурами допускается перегрузка операторов.

В лекции рассмотрено два варианта реализации стека на базе статического массива и с использованием динамической памяти. Явным недостатком обоих вариантов стека является открытость внутренних переменных структуры для логически некорректных изменений извне.

8. Упражнения

Упражнение 1

Напишите функцию для печати на экране содержимого стека, представленного в виде структуры "`Stack`" из п. 6.1. Перегрузку операторов не используйте. Проверьте работу функции с помощью подходящей тестовой программы.

Упражнение 2

Разработайте статический стек для хранения целых чисел и для хранения символьных строк длиной до 100 символов. Для каждого из стеков сделайте отдельные заголовочные файлы ("`stack_int.h`" и "`stack_string.h`") и файлы реализации ("`stack_int.cpp`" и "`stack_string.cpp`"). Проверьте работу стеков с помощью соответствующей тестовой программы.

Упражнение 3

Согласно описанию из п. 6.3 реализуйте стек с использованием динамической памяти (примените операторы "`new`" и "`delete`").

Упражнение 4

Напишите функции для записи содержимого массива структур "`Person`" (см. п.2) в файл, для чтения структур "`Person`" из файла и для печати этих структур на экране. Для ввода/вывода из файла и для вывода на экран примените перегруженные операторы ">>" и "<<".

ПРИЛОЖЕНИЕ. Краткое руководство по среде разработки Developer Studio Visual C++

Microsoft Developer Studio – это интегрированная среда разработки программ, объединяющая текстовый редактор, компилятор, компоновщик и отладчик. Эта среда позволяет разрабатывать программы на нескольких языках программирования, в том числе на Си++ и Java.

В этом приложении подробно описаны действия, необходимые для написания простой программы на Си++, ее компиляции и запуска с помощью **Developer Studio Visual C++** на ПК под управлением операционной системы **Windows 95/NT**.

Visual C++ выполняет компиляцию и запуск программ в соответствии с *проектом*. Проект – это структура данных, содержащая всю информацию, необходимую для компиляции исходных файлов программы и ее компоновки со стандартными библиотеками (например, библиотекой ввода/вывода).

Компиляция и компоновка исходных файлов называется *сборкой* проекта. В результате успешной сборки **Visual C++** создает *приложение* (двоичный исполняемый файл программы).

В данном учебном курсе все проекты рассчитаны на создание 32-битных консольных приложений. Консольные приложения общаются с пользователем через простейшее окно ввода/вывода, которое называется *консольным окном*.

1. Создание нового проекта

Проект состоит из набора файлов с исходным текстом программы (исходных файлов) и набора параметров, определяющих компиляцию и компоновку этих файлов в исполняемый файл приложения. У проекта должно быть уникальное имя. Параметры проекта хранятся в файлах с расширениями "`.DSW`" и "`.DSP`" в *папке проекта*.

Далее подробно описаны действия по созданию проекта для простого консольного приложения `hello_world`, которые вы можете воспроизвести самостоятельно.

Сначала с помощью главного меню **Windows** запустите **Visual C++**. Затем проделайте перечисленные ниже действия.

1) Выберите команду верхнего меню ***File⇒New*** (рис. 1).

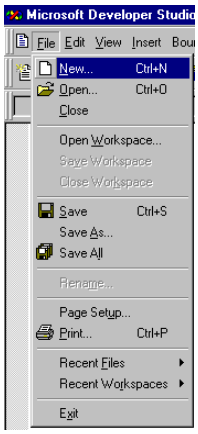

Рис. 1. Выбор команды ***File⇒New*** (***Файл⇒Новый***).

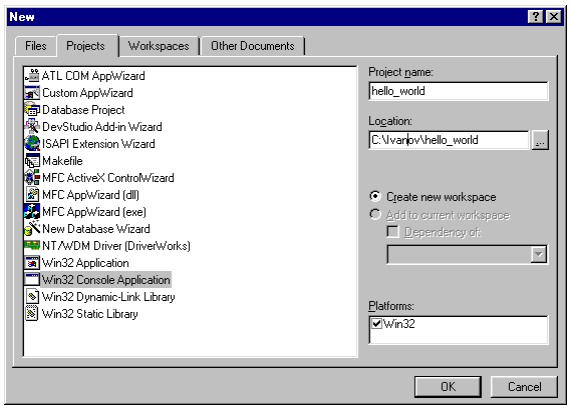

Рис. 2. Закладка **Projects** (**Проекты**) в окне создания нового файла.

2) Перейдите на закладку **Projects** (рис. 2).

3) Выберите проект типа **Win32 Console application** (консольное приложение для платформы **Win32**, т.е. **Windows 95/98** и **NT/2000**).

4) В строке **Location** (**Местоположение**) укажите папку диска `C:\`, имя которой совпадает с вашей фамилией (например, "`C:\Ivanov`"). В строке **Project Name** (**Имя проекта**) введите "`hello_world`". По умолчанию **Developer Studio** сделает новую папку проекта `C:\Ivanov\hello_world`. В ней будут сохраняться все файлы, относящиеся к данному проекту.

5) По умолчанию в качестве целевой платформы проекта указывается **Win32**. Не изменяйте этот параметр.

6) Нажмите **OK** для создания проекта с заданными параметрами.

2. Добавление в проект нового исходного файла

Чтобы включить в проект исходный текст программы, надо создать новый текстовый файл с текстом программы на Си++ и добавить его в проект. Для этого выполните следующие действия:

1) Выберите команду меню ***File⇒New***.
2) В окне создания нового файла перейдите на закладку **Files** (рис. 3).
3) В списке выберите тип нового файла: **C++ Source code** (исходный файл Си++).
4) По умолчанию новый файл будет добавлен в текущий проект `hello_world` (т.к. установлен флажок **Add to project**).
5) В строке **File name** наберите имя нового файла – `hello` (расширение "`.CPP`" будет добавлено автоматически).
6) Нажмите **OK**.

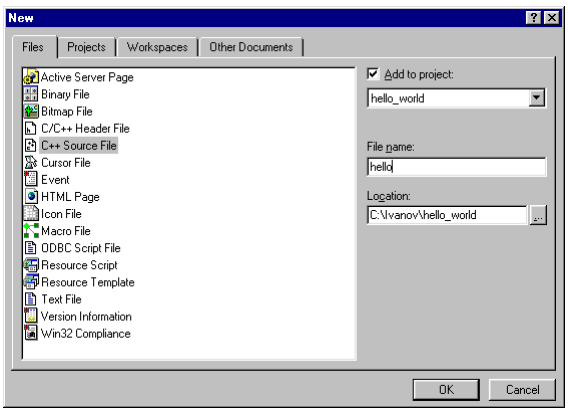

Рис. 3. Закладка **Files** (**Файлы**) в окне создания нового файла.

Developer Studio создаст новый файл `hello.cpp` в папке `C:\Ivanov\hello_world` и добавит его в проект `hello_world`. Новый файл автоматически будет открыт в окне текстового редактора (рис. 4). Наберите в нем текст программы, печатающей на экране короткое сообщение:

```
#include <iostream.h>
int main()
```

```
{
  cout << "Hello world!\n";
  return 0;
}
```

Чтобы сохранить набранный текст на диске, выберите команду меню ***File⇒Save*** (***Файл⇒Сохранить***).

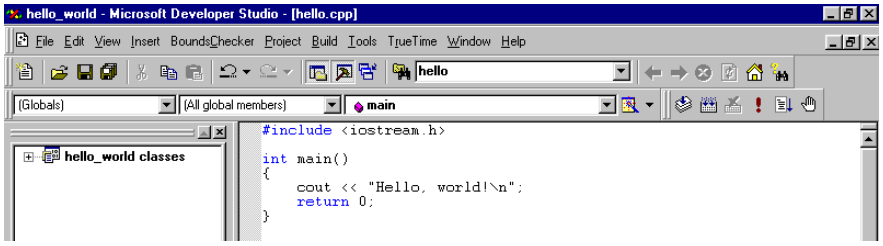

Рис. 4. Окно текстового редактора с открытым файлом `hello.cpp` расположено в правой части окна **Developer Studio**.

3. Сборка проекта

Результатом сборки проекта является исполняемый файл программы, который может работать независимо от **Developer Studio**.

Для сборки проекта выберите команду меню ***Build⇒Build hello_world.exe*** (рис. 5). В нашем примере проект содержит только один исходный файл (`hello.cpp`). В процессе сборки он будет скомпилирован и скомпонован со стандартной библиотекой ввода/вывода.

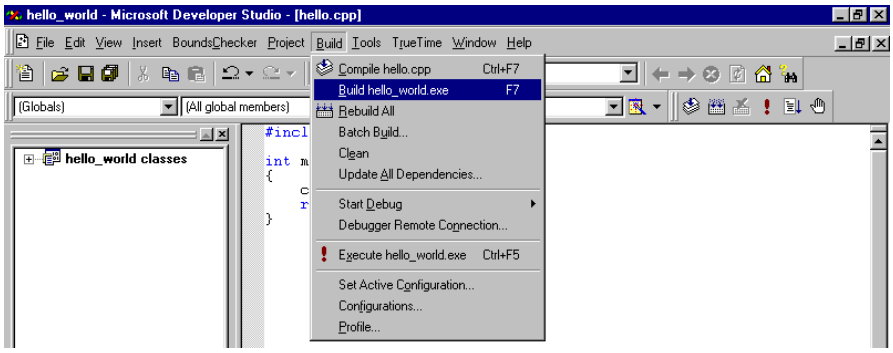

Рис. 5. Выбор команды ***Build⇒Build hello_world.exe*** (***Сборка⇒Сборка приложения hello_world.exe***).

Информация о выполнении сборки отображается в окне **Output window** (рис. 6). В нем выводятся сообщения, выдаваемые программами, работающими при сборке проекта: препроцессором, компилятором и компоновщиком. Среди этих сообщений могут встретиться сообщения об ошибках (errors) и предупреждения о возможных ошибках (warnings). Если таких сообщений не возникло, значит, сборка успешно завершена (рис. 6).

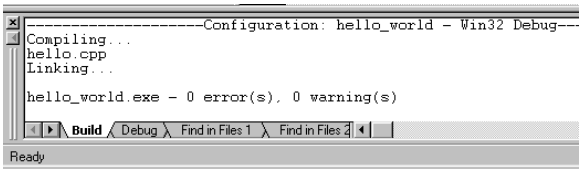

Рис. 6. Окно **Output window** (**Окно вывода**) расположено в нижней части окна **Developer Studio**.

Если есть ошибки, их надо устранить (в нашем случае просто внимательно сверьте исходный текст с образцом) и снова попытаться собрать проект.

4. Запуск нового приложения

В результате сборки было создано консольное приложение. Такие приложения широко использовались до появления систем **Windows**. Они удобны для учебных целей, т.к. простая структура консольных программ позволяет на начальном этапе изучения языка программирования не отвлекаться на системные особенности программ для **Windows**.

Для запуска приложения выберите команду меню ***Build⇒Execute hello_world.exe*** (рис. 7). Для удобства **Developer Studio** помещает имя исполняемого файла в название команды меню.

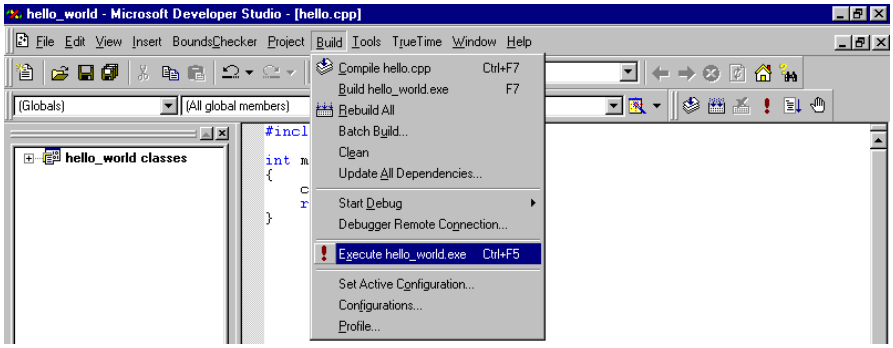

Рис. 7. Выбор команды ***Build⇒Execute hello_world.exe*** (***Сборка⇒Запуск приложения hello_world.exe***).

После выбора команды запуска **Developer Studio** создаст консольное окно – окно, напоминающее экран компьютера, работающего под управлением **MS-DOS**, а не **Windows**. Консольное приложение осуществляет ввод/вывод данных в пределах этого окна (рис. 8).

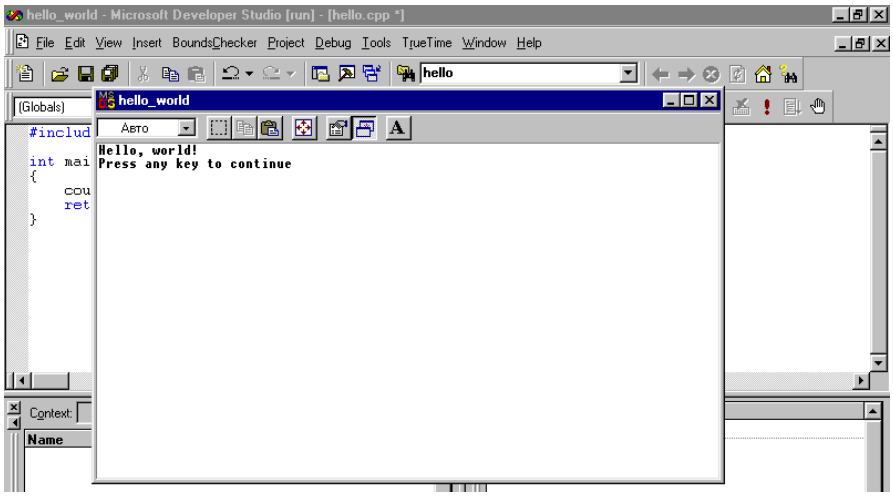

Рис. 8. Окно консольного приложения `hello_world.exe`.

Более подробные сведения об использовании среды разработки содержатся в справочной системе **Developer Studio**. В Приложении 2 описаны некоторые способы отладки программ и служебные клавиши отладчика **Developer Studio**.

Литература

1) Miller R., Clark D, White B., Knottenbelt W. An Introduction to the Imperative Part of C++. Imperial College of Science, Technology and Medicine, London. 1996-1999. (Вводное описание программирования на Си++, послужившее основой данной части учебного курса.)
2) Savitch W., Problem Solving with C++: The Object of Programming, 2nd Edition, Addison Wesley Publishing Company, Inc., 1999. (Учебник начального уровня по программированию, языку Си++ и объектно-ориентированному программированию.)
3) Вирт Н. Алгоритмы+структуры данных=программы. М.:Мир, 1985. (В этой монографии подробно рассматриваются алгоритмы сортировки, рекурсивные алгоритмы и динамические типы данных. Изложение базируется на языке Паскаль, но излагаемый материал во многом применим и к процедурному программированию на Си++.)
4) Страуструп Б. Язык программирования C++. Вторая редакция. К.:"ДиаСофт", 1993. ("Классическое" справочное руководство по языку Си++, написанное автором языка. Эта книга пригодится тем, кто собирается в будущем серьезно заниматься программировать на Си++.)
5) Уэйт М., Прата С., Мартин Д. Язык Си. Руководство для начинающих. М.:Мир, 1988. (Учебник начального уровня по языку Си без объектно-ориентированных возможностей. В отличие от данных лекций, в этой книге используются библиотечные функции ввода-вывода языка Си, а не потоковые объекты Си++.)