Федеральное агентство по образованию
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ГОУВПО «АмГУ»
Факультет математики и информатики

УТВЕРЖДАЮ
Зав. кафедрой МАиМ
Т.В. Труфанова
«___»_____ 2007г.

Учебно – методический комплекс дисциплины

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

для специальности 010501 – прикладная математика

Составитель: Труфанов В.А.

Благовещенск
2007

Печатается по решению
редакционно-издательского совета
факультета математики и
информатики
Амурского государственного
университета

Труфанов В.А.

Языки программирования и методы трансляции. Учебно – методический комплекс дисциплины для студентов очной формы обучения специальности 010501–"Прикладная математика". – Благовещенск: Амурский гос. ун–т, 2007. − 94 с.

ОГЛАВЛЕНИЕ

1. Государственный образовательный стандарт

Специальность 010501–"Прикладная математика и информатика".

Квалификация – математик ,системный программист.

Дисциплина «Языки программирования и методы трансляции» входит в блок дисциплин федерального компонента для специальности 010501 – «Прикладная математика».

ОПД.Ф.05*:* **Языки программирования и методы трансляции** (153 ч.)

Основные понятия языков программирования; синтаксис, семантика, формальные способы описания языков программирования; типы данных, способы и механизмы управления данными; методы и основные этапы трансляции; конструкции распределенного и параллельного программирования.

2. Рабочая программа

по дисциплине **" Языки программирования и методы трансляции "**

для специальности 010501–"Прикладная математика"

Курс 1 Семестр 1,2

Лекции 72 (36+36) час.

Экзамен 010501 –"Прикладная математика" 2 семестр

Практические (семинарские) занятия 36 (18+18) час.

Лабораторная работа 18 час.

Зачет 1 семестр.

Самостоятельная работа 45 (20+25) час.

Всего часов 153 (74+79) час.

1. Цели и задачи дисциплины, ее место в учебном процессе.

1.1. Цель преподавания дисциплины.

Основной целью дисциплины является введение в проблематику, связанную с изучением языков программирования, методов разработки алгоритмов и программ и методов реализации языков программирования. Формирование у студентов системных знаний о существующих теориях вычислительных процессов, а также структурах, предназначенных для их реализации; ознакомление с началами программирования на C++.

.

1.2. Задачи изучения дисциплины.

Основными задачами являются: знание методов описания синтаксических конструкций языков программирования; освоение основных концепций языков программирования; знание жизненного цикла программного обеспечения и понимание работ, выполняемых на каждом из его этапов; знание структуры транслятора и понимание стадий трансляции программы.

1.3. Перечень дисциплин с указанием разделов, усвоение которых студентами необходимо при изучении данной дисциплины.
Курс поддержан соответствующим компьютерным практикумом в рамках дисциплин «Информатика», «Практикум на ЭВМ». При изучении дисциплины привлекаются понятия и методы математического анализа, дискретной математики, линейной алгебры.

2. Сводные данные об основных разделах дисциплины и распределение часов по видам занятий.

Тематический план.

на I семестр

№ темы	Наименование темы	Лекции	Практические занятия	Самост. работа
1	Теория формальных языков и трансляций. Введение в проблематику языков программирования. Технология программирования.	4	4	4
2	Языки программирования (на примере языка C++)	32	14	16
Итого		**36**	**18**	**20**

на II семестр

№ темы	Наименование темы	Лекции	Практические занятия	Самост. работа
1	Парадигмы языков программирования	2	1	1
2	Критерии оценки языков прогр-я	2	1	1
3	Объекты данных в языках прогр-я	2	1	1
4	Механизмы типизации	2	1	1
5	Время жизни переменных. Область видимости переменных	2	1	1
6	Типы данных	2	1	1
7	Выражения и операторы присваивания.	2	1	1
8	Структуры управления на уровне операторов	2	1	1
9	Подпрограммы	2	1	1
10	Описание языка программирования	2	1	1
11	Теоретические основы трансляции	16	8	15
Итого		**36**	**18**	**25**

3. Самостоятельная работа студентов

3.1 Знакомство с рекомендуемой литературой.

3.2 Подготовка к практическим занятиям.

4. Наименование тем лекций и их содержание

I семестр

Тема 1

Начальные сведения о языке.

История создания языка и его эволюция. Международный стандарт языка. Сферы применения языка Си++. Пример простой программы. Процесс построения программы. Компилирование в среде Windows.

История и назначение языка Си++.

Разработчиком языка Си++ является Бьерн Страуструп. В своей работе он опирался на опыт создателей языков Симула, Модула2, абстрактных типов данных. Основные работы велись в исследовательском центре компании Bell Labs.

Непосредственный предшественник Си++ – язык Си с классами – появился в 1979 г., а в 1997 г. был принят международный стандарт Си++, который фактически подвел итоги его 20-летнего развития. Принятие стандарта обеспечило единообразие всех реализаций языка Си++. Не менее важным результатом стандартизации стало то, что в процессе выработки и утверждения стандарта язык был уточнен и дополнен рядом существенных возможностей.

На сегодня стандарт утвержден Международной организацией по стандартизации ISO. Его номер ISO/IEC 14882. ISO бесплатно стандарты не распространяет. Его можно получить на узле американского национального комитета по стандартам в информационных технологиях : www.ncits.org.

В России следует обращаться в ВНИИ. Сертификации: www.vniis.ru.

Язык Си++ является универсальным языком программирования, а в дополнение к которому разработан набор разных библиотек. Поэтому, строго говоря, он позволяет решить практически любую задачу программирования. Тем не менее, в силу разных причин (не всегда технических) для каких-то типов задач он употребляется чаще, а для каких-то реже.

С++ как приемник языка Си широко используется в системном программировании. На нем можно писать высоко эффективные программы, в том числе операционные системы (ОС), драйверы и т. п. Язык Си++ один из основных языков разработки трансляторов. Поскольку системное программное обеспечение часто бывает написано на языке Си или Си++, то и программные интерфейсы к подсистемам ОС тоже часто пишут на Си++. Соответственно, те программы, даже и прикладные, которые взаимодействуют с операционными системами, написаны на языке Си++.

Обработка сложных структур данных – текста, бизнес–информации, Internet-страниц и т.п. – одна из наиболее распространенных возможностей применения языка. В прикладном программировании, наверное, проще назвать те области, где язык Си++ применяется мало.

Разработка графического пользовательского интерфейса на языке Си++ выполняется, в основном, тогда, когда необходимо разрабатывать сложные, нестандартные интерфейсы. Простые программы чаще пишутся на языках Visual Basic, Java и т.п.

Программирование для Internet в основном производится на языках Java, VBScript, Perl.

В целом надо сказать, что язык Си++ в настоящее время является одним из наиболее распространенных языков программирования в мире.

Простейшая программа на языке Си++.

Самая короткая программа на языке Си++ выглядит так:

```
// простейшая программа
 void main ( ) { }
```

Первая строка в программе – комментарий, который служит лишь для пояснении. Признаком комментария является два знака деления подряд.

main – это имя главной функции программы. С функции main всегда начинается выполнение. У функции есть имя (main), после имени в круглых скобках перечисляются аргументы или параметры функции (в данном случае у функции main аргументов нет). У функции может быть результат или возвращаемое значение. Если функция не возвращает никакого значения, то это обозначается ключевым словом void. В фигурных скобках записывается тело функции – действия, которые она выполняет. Пустые фигурные скобки означают, что никаких действий не производится. Таким образом, приведенная программа ничего не делает!

Если мы говорим об объектно-ориентированной программе, то она должна создать объект какого-либо класса и послать ему сообщение. Чтобы не усложнять программирование, воспользуемся одним из готовых, предопределенных классов – классом ostream (поток ввода-вывода). Этот класс определен в файле заголовков “iostream.h”. Поэтому первое что надо сделать – включить файл заголовков в нашу программу:

```
# include <iostream.h>
void main ( )
{ cout <<”Welcome to C++! ”  << endl;
}
```

Кроме класса, файл заголовков определяет глобальный объект этого класса cout. Объект называют глобальным, поскольку доступ к нему возможен из любой части программы. Этот объект выполняет вывод на консоль. В

функции main мы можем к нему обратится и послать ему сообщение:
cout<<”Welcome to C++! ” << endl;
Операция сдвига << для класса iostream определена как «ввести ». Таким образом, программа посылает объекту cout сообщение «вывести строку Welcome to C++» и «вывести перевод строки» (endl обозначает новую строку). В ответ на эти сообщения объект cout выведет строку <<”Welcome to C++! ”>> на консоль и переведет курсор на следующую строку.

—

Процесс построения программы.

Здесь мы рассмотрим процесс подготовки и трансляции программы на языке высокого уровня (в нашем случае это Си++) в исполняемый файл, содержащий машинные инструкции и все остальное, что необходимо для работающей программы системы Windows .

Создание программы на языке C++ выглядит примерно так.

Прежде всего, программист с помощью того или иного текстового редактора готовит файлы исходного кода на C/C++. После этого происходит построение программы, в котором выделим такие этапы:

Компиляцию исходных файлов в файлы объектного кода (с расширением .obj).

Компоновку объектных файлов с присоединением необходимых библиотек, в результате получится уже машинный код.

Компоновку ресурсов (этап, специфический для Windows; ресурсы включают в себя битовые матрицы, курсоры, строковые таблицы, пиктограммы и т.п.). Это завершающий шаг, на котором формируется конечный exe-файл, запускаемый на выполнение.

В настоящее время популярны интегрированные схемы разработки программного обеспечения (IDE). К их числу относят Microsoft Visual C++, Delphi т.п. Такая среда скрывает от программиста детали вышеупомянутого процесса. Создается впечатление, что он имеет дело с чем-то цельным. Однако не следует забывать, что в наиболее существенной части работу

такой системы обеспечивают те же самые конвенциональные инструменты (компилятор, компоновщик, компилятор ресурсов), а то что мы видим – это лишь их графическая оболочка.)

Компилирование и выполнение программ в среде Windows.

Пусть у нас будет Visual C++ с версией 5.0 и выше. Этот компилятор представляет собой интегрированную среду разработки.

В этой среде прежде всего необходимо создать новый проект. Для этого нужно выбрать в меню File атрибут New. Появиться новое диалоговое окно. В закладке Projects в списке различных типов выполняемых файлов выберите Win 32 Consol Application. Убедитесь, что отмечена кнопка Create new workspace. Затем следует набрать имя проекта в поле Project name (например, test) и имя каталога, в котором будут храниться все файлы, относящиеся к данному проекту, в поле Location. Например: C:\Work. После этого нажмите кнопку OK.

Теперь необходимо создать файл. Опять в меню File выберете атрибут New. В появившемся диалоге в закладке File отметьте C++ Source File. По умолчанию новый файл будет добавлен к текущему проекту text, в чем можно убедиться, взглянув на поле Add to project. В поле File name нужно ввести имя файла. Пусть это будет main.cpp. Расширение cpp – это стандарт для файлов с исходными текстами на языке C++. Поле Location должно показывать на каталог: C:\Work. Нажмите кнопку OK. На экране появится пустой файл. Наберите текст программы.

Компиляция выполняется с помощью меню Build. Выберите пункт Build text.exe (этому пункту меню соответствует функциональная клавиша F7). В нижней части экрана появятся сообщения компиляции. Если была сделана ошибка (Например, опечатка), двойной щелчок мыши по строке с ошибкой переведет курсор в окне текстового редактора на соответствующую строку кода. После исправления всех ошибок и повтора компиляции система выдаст сообщение об успешной компиляции и компоновке (увидите сообщение

Linking). Готовую программу можно выполнить с помощью меню Build, пункт Execute text.exe. То же самое можно сделать, нажав одновременно клавиши Ctrl – F5. На экране монитора появится консольное окно и в нем будет выведена строка ”Welcome to C++! ”. Затем появится надпись “Press any key to continue”; что означает; программа выполнилась и лишь ожидает нажатия произвольных клавиш, чтобы закрыть консольное окно.

Тема 2

Имена, переменные и константы.

Правила именования переменных и функций языка, правила записи констант. Понятие ключевого или зарезервированного слова.

Имена

Для символического обозначения величин, имен функций и т. п. используются имена или идентификаторы.

Идентификаторы языка C++ требуют соблюдение следующих правил:

1. Первым символом должна быть буква или знак подчеркивания _.
2. Последующие символы могут быть буквами, цифрами или знаками подчеркивания.
3. Длинна идентификаторов произвольная.
4. Идентификаторы в языке Си++ являются зависимыми от регистра, т.е. заглавные и строчные буквы различаются, поэтому имена *авс* и *Abc* – два разных идентификатора.
5. Идентификаторами не могут быть зарезервированные слова.

Рекомендации. *Используйте несущие смысловую нагрузку имена разумной длины. Не используйте слишком длинных и слишком коротких имен идентификаторов. Короткие имена неинформативны, а длинные часто являются источником ошибок.*

Ряд слов в языке Си++ имеет особое значение и не могут использоваться в качестве идентификаторов. Такие зарезервированные слова называются ключевыми. Например: int, double, static.

В следующем примере:

```
int max (int x, int y)
{
 f (x>y)
        return x;
 else
        return y;
}
```

Здесь max, x, y – имена или идентификаторы. Слова int, if, return, else – ключевые слова, они не могут быть именами переменных или функций и используются для других целей.

Переменные.

Программа оперирует информацией, представленной в виде различных объектов и величин. Переменная – это символическое обозначение величины в программе.

Как ясно из названия, значение переменной (или величина, которую она обозначает) во время выполнения программы может изменяться.

С точки зрения архитектуры, переменная – это символическое обозначение ячейки оперативной памяти (ОП) программы, в которой хранятся данные. Содержимое этой ячейки – это текущее значение переменной.

В языке Си++ прежде чем использовать переменную, ее необходимо объявить. Объявить переменную с именем *x* можно так:

```
int x;
```

В объявлении первым стоит название типа переменной int (целое число), а затем идентификатор *x* – имя переменной. У переменной *x* есть тип – в данном случае целое число. Тип переменной определяет, какие возможные значения эта переменная может принимать и какие операции можно выполнять над данной переменной. Тип переменной изменить нельзя, т.е. пока переменная *x* существует, она всегда будет целого тип, но возможно принудительное приведение типа к какому-либо оператору в выражении.

Язык Си++ - это строго типизированный язык. Любая величина, используемая в программе, принадлежит к какому-либо типу. При любом использовании переменных в программе проверяется, применимо ли выражение или операция к типу переменных. Довольно часто смысл выражения зависит от типа участвующих в нем переменных.

Например, если запишем *x+y*, где *x* – объявленная выше переменная, то переменная *y* должна быть одного из числовых типов.

Соответствие типов проверяется во время компиляции программы. Если компилятор обнаруживает несоответствие типа переменной и ее использования, он выдаст ошибку (или предупреждение). Однако во время выполнения программы типы не проверяются. Такой подход, с одной стороны, позволяет обнаружить и исправить большое количество ошибок на стадии компиляции, а , с другой стороны, не замедляет выполнение программы.

Переменной можно присвоить какое – либо значение с помощью операции присваивания. Присвоить – это значит установить текущее значение переменной. По-другому можно объяснить, что операция присваивания запоминает новое значение в ячейке память, которая обозначена переменной.

```
int x ;      //объявление целой переменную x.
int y;
x = 0;       //присвоить x значение 0.
```

```
y = x+1;   // присвоить y значение x+1, т.е. 1.
```

Приведение типа.

Если в операторе присваивания тип результата, полученного при оценке выражения в правой части, отличен от типа переменной слева, компилятор выполнит автоматическое приведение типа результата к типу переменной. Например, если оценка выражения дает вещественный результат, который присваивается целой переменной, то дробная часть результата будет отброшена, после чего будет выполнено присваивание. Рассмотрим обратный случай приведения:

```
int p;
double pReal=2.718281828;
p=pReal;                    // p получает значение 2.
pReal=p;                    // pReal теперь равно 2.0.
```

Возможно и принудительное приведение типа, который выполняется посредством операции приведения и может применяться к любому операнду в выражении, например:

```
p=p0+(int)(pReal+0.5);          // округление pReal.
```

Замечание: *Следует иметь в виду, что операция приведения типа может работать двояким образом. Во-первых она может производить действительное преобразование данных, как это происходит при проведении целого типа к вещественному и наоборот. Получаются совершенно новые данные, физически отличные от исходных. Во-вторых операция может никак не воздействовать на имеющиеся данные, а только изменять их интерпретацию.*

Например, если переменная типа short со значением -1 привести к типу insigned short, то данные останутся теми же самыми, но будут интерпретироваться по другому (как целое без знака), в результате чего будет получено максимальное значение этого типа данных 65535.

Константы.

В программе можно явно записать величину – число, символ и т.п. Например, мы можем записать выражение *x*+4 – сложить текущее значение переменной *x* и число 4. В зависимости от того, при каких условиях мы будем выполнять программу, значение переменной *x* может быть различным. Однако целое число четыре всегда останется прежним. Это неизменяемая величина или константа.

Таким образом, явная запись значения в программе – это константа. Далеко не всегда удобно записывать константы в тексте программы явно. Гораздо чаще используются символические константы. Например, если записать const BITS_IN_WORD=32; то за тем имя BITS_IN_WORD можно будет использовать вместо целого числа 32.

Преимущества такого подхода очевидны. Во-первых, имя (битов в машинном слове) дает хорошую подсказку, для чего используется данное число. Тогда и без комментариев понятно, что выражение b/ BITS_IN_WORD (значение *b* разделить на число 32) вычисляет количество машинных слов, необходимых для хранения *b* битов информации. Во-вторых, если по каким либо причинам нам надо изменить эту константу., потребуется изменить только одно место в программе – определение константы, оставив все случаи ее использования как есть. (Например, мы переносим программу на компьютер с другой длиной машинного слова).

Таким образом, константы являются типизированными и значение констант, которые заданы при инициализации, нельзя изменить.

Тема 3

Операции и выражения.

Правила формирования выражений в языке С++. Все операции языка.

Выражения.

Программа оперирует с данными. Числа можно складывать, вычитать, умножать, делить. Знаки можно сравнивать и т.д. То есть из разных величин можно составлять выражения, результат вычисления которых – новая величина. Приведем примеры выражений:

```
x*12+y      // значение x умножается на 12 и к результату прибавить
            // значение y
val < 3     // сравниваем значение val с 3
-9          // константное выражение
```

Выражение после которого стоит ‘;’ – это оператор – выражение. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение.

```
x+y-12;     // сложить значение x и y, а затем вычесть 12
a = в+1;    // прибавить к 1 к значению в и запомнить результат в
            // переменной a
```

Выражение – это переменные, функции и константы, называемые операндами, объединенные знаками выражений. Операции могут быть унарными – с одним операндом, например, минус (приоритет операции 14); могут быть бинарные – с двумя операндами, например сложение (12) и деление (13). В С++ есть даже одна операция с тремя операндами – условное выражение. Позже приведем список всех операций языка С++ для встроенных типов данных. Подробно каждая операция будет разбираться при описании соответствующего типа данных. Кроме того, ряд операций будет

рассмотрен в разделе, посвященном определителю операторов для классов. Пока мы ограничимся лишь общим описанием способов записи выражений.

В типизированном языке, которым является Си++, у переменных и констант есть определенный тип. Есть он и у результата выражения. Например, операции сложения (+), умножения (*) (13), вычитания (-) (12) и деления (/), примененные к целым числам, выполняются по общепринятым математическим правилам и дают в результате целое значение. Те же операции можно применить к вещественным числам и получить вещественное значение.

Операции сравнивания: больше (>) (10), меньше (<) (10), равно (= =) (9), не равно (!=) (9) сравнивают значения чисел и выдают логическое значение: истина (true) или ложь (false).

Операция присваивания.

Присваивания – это тоже операция, являющаяся частью выражения, и которая сама (и, соответственно, все выражение участвует в целом) возвращает значение. Значение правого операнда присваивается левому операнду. В этом отличие Си++ от других языков, в частности Pascal, где присвоение является оператором, а не операцией. Оператором выражение станет, если поставить после него ‘;‘.

Например:

```
x=2;          //переменной x присвоить значение 2
bool   cond;
cond = x < 2; // переменной cond присвоить значение true, если
                // x<2, в противном случае присвоить значение false.
3=5;          //ошибка, число 3 неспособно изменять свое значение
```

Последний пример иллюстрирует требование к левому операнду операции присваивания. Он должен быть способен хранить и изменять свое значение.

У операции присваивания тоже есть результат. Он равен значению правого операнда. Таким образом, операция присваивания может участвовать в более сложном выражении: z = (x = y + 3);

В этом примере переменной *x* и *z* присваивается значение *y*+3. Очень часто в программе приходится значение переменной увеличивать или уменьшать на 1. Для того, чтобы сделать эти действия наиболее эффективными и удобными для использования, применяются предусмотренные в Си++ специальные знаки операций: ++(увеличить на 1) и -- (уменьшить на 1). Существуют две формы этих операций: префиксная (++ | -- переменная) (14) и постфиксная (переменная ++ | --) (15). Рассмотрим их на примерах.

```
int x=0;
++x;          // аналогично x++; то есть когда операция ++ (--) является
              // единственной.  Значение x увеличивается на единицу и
              // становится равным 1.
--x;           // значение x  уменьшается на 1 и становится  = 0.
int y = ++x; // значение x опять увеличивается на 1.
```

Результат операции ++ – новое значение *x* = 1, то есть переменной *y* присваивается значение 1.

```
int z=x++;
```

Здесь используется постфиксная запись операции увеличения на 1. Значение переменной *x* до выполнения операции равно 1. Однако результат постфиксной операции – значение аргумента до увеличения. Таким образом, переменная *z* присваивает значение 1, а затем значение *x* увеличивается на 1 и становится равным 2.

Аналогично, результатом постфиксной операции уменьшения на единицу является начальное значение операнда, а префиксной – его конечное значение. Другими словами, в первом случае результатом является значение операнда до изменения на 1, а во втором случае – после изменения на единицу.

Подобными мотивами оптимизации и сокращения записи руководствовались создатели Си (а за тем и Си++), когда вводили новые знаки операций типа «выполнить операцию и присвоить». Поскольку довольно часто одна и та же переменная используется в левой и правой части операции присваивания, например: x=x+5; y=y*3; z=z-(x+y);.

То в Си++ эти выражения можно записать короче: x += 5; y *= 3; z -= x+y. То есть запись *операция*= означает, что левый операнд вначале использует как левый операнд операции *операция*, а затем как левый операнд операции присваивания результата операции *операция*. Кроме краткости выражения, такая запись облегчает оптимизацию программы компиляции.

Другие операции языка С++.

Наряду с общепринятыми арифметическими и логарифмическими операциями, в языке Си++ имеется набор операций для работы с битами – поразрядные И, ИЛИ, Исключающее. ИЛИ и НЕ, а также сдвиги.

Особняком стоит операция sizeof (приоритет 14), которая позволяет определить, сколько памяти занимает то или иное значение в байтах. Например:

```
sizeof (long);      // сколько байтов занимает тип long
sizeof b;           // сколько байтов занимает переменная b
```

Операция sizeof в качестве аргумента берет имя типа или выражение. Аргумент заключается в скобки (если аргумент – выражение, то скобки не обязательны). Результат операции – целое число, равное количеству байтов, которое необходимо для хранения в памяти заданной величины.

Арифметические операции

Арифметические операции: + (сложение), - (вычитание), *(умножение), / (деление). Операции сложения, вычитания, умножения и деления целых и вещественных чисел. Результата операции – число, по типу соответствующее

большему по разрядности операнду. Например, сложение чисел типа short и long в результате дает число типа long.

Операция нахождения остатка от деления % (13) одного целого числа на другое. Тип результата – целое число.

Операция «минус»(-) – это унарная операция, при которой знак числа изменяется на противоположный. Она применима к любым числам со знаком.

Операция «плюс» существует для симметрии. Она ничего не делает, то есть примененная к целому числу, его же и выдает.

Операции сравнения

Операции сравнения: = = (равно) (9), != (неравно) (9), < (меньше), > (больше), <= (меньше или равно), >= (больше или равно) (все по 10).

В этих операциях сравнивать можно операнды любого типа, но либо они должны быть оба одного и того же встроенного типа (сравнение на равенство и неравенство работает для величин любого типа), либо между ними должна быть определена соответствующая операция сравнения. Результат – логическое значение true или false.

Логические операции

Логические операции конъюнкции &&(логическое И) (5), дизъюнкции – || (логическое ИЛИ) (4), отрицания – ! (логическое НЕ) (14). В качестве операндов выступают логические значения, результат – тоже логическое значение true или false.

Битовые операции

& – битовое И (8), | – битовое ИЛИ (6), ^ – битовое исключающее ИЛИ (синоним сложение по модулю два) (7), ~ – битовое НЕ. Эти операции выполняются над целыми числами. Соответствующая операция выполняется над каждым битом операндов. Результатом является целое число.

<< – сдвиг в лево (11), побитовый сдвиг левого операнда на количество разрядов в сторону старших битов,

>> – сдвиг в право (11), побитовый сдвиг правого операнда. Старшие биты, оказавшиеся за пределами разрядной сетки, теряются; справа результат добавляется нулями. Результат сдвига вправо зависит от того, является ли операнд знаковым или беззнаковым. Биты операнда перемещаются вправо на заданное число позиций. Младшие биты теряются. Если операнд – целое со знаком, то производится расширение знакового бита (старшего), т. е. освободившиеся позиции принимают значение 0 в случае положительного числа и 1 – в случае отрицательного. При беззнаковом операнде старшие биты заполняются нулями. Сдвиг влево эквивалентен умножению на соответствующую степень двойки, сдвиг вправо – делению. Например:

```
aNumber = aNumber <<4;        // умножает aNumber на 16=  .
```

Условная операция(? :)

Эта операция позволяет составить условное выражение, то есть выражение, принимающее различные значения в зависимости от некоторого условия. Эта операция трехарная (трехместная) (3); если значение первого операнда истина, то результат – второй операнд; если ложь – результат третий операнд. Первый операнд должен быть логическим значением, второй и третий операнды могут быть любого, но одного и того же типа.
Классический пример:

```
max_ab = a > b ? a : b;
```

Последовательность (запятая)

Выполнить выражение до запятой, затем выражение после запятой. Два произвольных выражения можно поставить рядом, разделив их запятой (приоритет 1). Они будут выполняться последовательно, и результатом всего выражения будет результат последнего выражения. Например:

```
res = (j=4, j+=n, j++);     // res присваивает n+4, j равно n+5
```

Операция запятая применяется довольно редко, обычно только в управляющих выражениях циклов.

Порядок вычисления выражений

У каждой операции имеется приоритет. Если в выражении несколько операций, то первой будет выполняться операция с более высоким приоритетом. Если же операция одного того же приоритета, они вычисляются слева направо. Например: 2+3*6 (=20) – сначала выполняется умножение, а затем сложение. В выражении x=y+3 вначале выполняется сложение, а затем присваивание, поскольку приоритет операции присваивания ниже, чем приоритет операции сложения.

Если в выражении несколько операций присваивания, то они выполняются справа налево, например, x=y=2 сначала будет выполняться операция присваивания значения 2 переменной y, затем результат этой операции – значение 2 – присваивается переменной x.

Для того, чтобы изменить последовательность вычисления выражений, можно воспользоваться круглыми скобками. Часть выражения, заключенного в скобки, вычисляется в первую очередь.

Скобки могут быть вложенными, соответственно, самые внутренние выполняются первыми.

Тема 4

Операторы

Что такое оператор

Запись действий, которые должен выполнить процессор компьютера, состоит из операторов. При выполнении программы операторы выполняются один за другим, если оператор не является оператором управления, который может изменить последовательное выполнение программы.

Различают операторы объявления имен, операторы управления и операторы – выражения.

Операторы выражения

Выражения были рассмотрены в предыдущей лекции. Выражение, после которого стоит ; – это *оператор – выражение*. Его смысл состоит в том, что компьютер должен выполнить все действия, записанные в данном выражении, иначе говоря, вычислить выражение. Чаще всего в операторе – выражении стоит операция присваивании или вызов функции. Операторы выполняются последовательно, и все изменения значений переменных, сделанные в предыдущем операторе, используются в последующих.

```
a=1; b=3;
m=max(a,b); // вызывается функция max с параметрами 1 и 3, и ее
                    // результат присваивается переменной m
```

Как уже отмечали ранее, присваивание – необязательная операция в операторе – выражении. Следующие операторы тоже вполне корректны:

```
x+y-12; // сложение значения x и y затем вычесть 12
func(d,12,x); // вызвать функцию func с заданными параметрами.
```

Операторы объявления имен

Эти операторы объявляют имена, то есть делают их известными программе. Все идентификаторы или имена, используемые в программе на языке Си++, должны быть объявлены.

Оператор объявления состоит из названия типа и объявляемого имени:

```
int x; // объявляется целая переменная x
double f; // объявить переменную f  типа double – вещественный с
            // двойной точностью.
const float PI=3.141592; // объявляет константу PI типа float –
                                    // (вещественный) со значением
```

Оператор объявления заканчивается ; .

Операторы управления

Операторы управления определяют, в какой последовательности выполняется программа. Если бы их не было, операторы программы всегда выполнялись бы последовательно, в том порядке, в каком они записаны.

Условные операторы. Условные операторы позволяют выбрать один из вариантов выполнения действий в зависимости от каких – либо условий. Условие – это логическое выражение, результатом которого является логическое значение true(истина) или false(ложь).

Оператор if выбирает один из двух вариантов последовательности вычислений.

```
if (условие)
   оператор 1
else
    оператор 2
```

Если условие истинно, выполняется оператор 1, если ложно, то оператор 2.

Пример. В данном примере переменная *a* присваивает значение max из двух величин *x* и *y*

```
if (x > y)
   a = x;
else
   a = y;
```

Конструкция else необязательна. Можно записать:

Пример:

```
if (x<0)
   x = -x;
abs = x;
```

В данном примере оператор x=-x; выполняется только в том случае, если значение переменной *x* было отрицательным. Присваивание переменной abs выполняется в любом случае

Если в случае истинности условия необходимо выполнить несколько операторов, их можно заключить в фигурные скобки:

Пример:

```
if (x < 0)
{  x = -x;
   cout << “Изменяется значение x на противоположное по знаку”;
}
abs=x;
```

Теперь, если значение *x* отрицательно, то не только его значение изменяется на противоположное, но и будет выведено соответствующее сообщение. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Прием заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.

Условный оператор можно расширить для проверки нескольких условий:

```
If (x<0)
   cout << “Отрицательная величина”;
else if (x>0)
   cout << “Положительная величина”;
else
   cout << “Ноль”;
```

Конструкций else if может быть несколько.

Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной. Оператор выбора switch используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия. Например, предположим, что переменной code хранится целое число от 0 до 2, и нам нужно выполнить различные действия в зависимости от ее значения:

```
switch (code)
{
 case 0:
      cout << “код ноль”;
      x=x+1;
      break;
 case 1:
      cout << “код один”;
      y+=1;
      break;
 case 2:
      cout << “код два”;
      z=z+1;
      break;
 default:
      cout << “Необрабатываемое значение”;
}
```

В зависимости от значения code управление передается на одну из меток case. Выполнение оператора заканчивается по достижению либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится сообщение «код один», а затем переменная *y* увеличивается на 1. Если бы после этого не стоял оператор break, то управление «провалилось»

бы дальше, была бы введена фраза «код два», и переменная z тоже увеличилась бы на единицу.

Если значение переключателя не совпадает ни с одним из значений меток case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи:

```
default:
    ;              // пустой оператор, не выполняющий никаких действий.
```

Очевидно, что приведенный пример можно переписать с помощью оператора if.

Особенно часто переключатель switch используется, когда значение выражения имеет тип набора.

Операторы цикла

Предположим, нам нужно вычислить сумму всех целых чисел от 0 до 100 . Для этого воспользуемся оператором цикла for:

```
int sum = 0;
for (int i = 1; I <= 100; I += 1)  // Заголовок цикла
    sum += i;                      // тело цикла
```

Оператор цикла состоит из заголовка цикла и тела цикла. Тело цикла – это оператор, который будет повторно выполняться (в данном случае – увеличение значения переменной sum на величину переменной *i*). Заголовок – это ключевое слово for, после которого в круглых скобках записаны три выражения, разделенные символом ;. Первое выражение вычисляется один раз до начала выполнения цикла. Второе – это условие цикла. Тело цикла будет повторяться до тех пор, пока условие цикла истинно. Третье выражение вычисляется после каждого повторения тела цикла.

Оператор for реализует фундаментальный принцип вычислений в программировании – итерацию. Тело цикла повторяется для разных , в

данном случае последовательных, значений переменной *i*. Повторение иногда называется итерацией. Мы как бы проходим по последовательности значений переменной *i*, выполняя с текущим значением одно и тоже действие, тем самым постепенно вычисляя нужное значение. С каждой итерацией подходим к нему все ближе и ближе. С другим принципом вычислений в программировании – рекурсией – познакомимся в разделе, описывающем функции.

Любое из трех выражений в заголовке цикла может быть опущено (в том числе и все три). То же самое можно записать следующим образом:

```
int i = 1;
for (; i <= 100;)
{
    sum = sum+1;
    i = i+1;
}
```

Заметим, что вместо одного оператора цикла мы записали несколько операторов, заключенных в фигурные скобки

Другой вариант:

```
int i=1;
for ( ; ; )
{
if (i > 100;)
   break;
sum += i; i += 1;
}
```

В этом примере мы опять встречаем оператор break. Здесь он завершает выполнение цикла. Еще одним вспомогательным оператором при выполнении циклов служит оператор продолжения continue. Он заставляет

пропустить остаток тела цикла и перейти к следующей итерации (повторению). Например, если нам нужно найти сумму целых чисел от 0 до 100, которые не делятся на 7, то можно это записать так:

```
int sum = 0;
for ( int i = 1; i <= 100; i = i + 1)
{   if ( i % 7 == 0)
    continue;
    sum += i;
}
```

Еще одно полезное свойство цикла for: в первом выражении заголовка цикла можно объявить переменную. Эта переменная будет действительна только в пределах цикла.

Другой формой оператора цикла является оператор while. Его форма:

```
while (условие)
        оператор
```

Условие – как и в условном операторе if – это выражение, принимающее логическое значение истина или ложь. Выполнение оператора повторяется до тех пор, пока значение условия является true (истина). Условие вычисляется заново после каждой итерации. Посчитать, сколько десятичных цифр нужно для записи целого положительного числа N, можно с помощью следующего фрагмента:

```
int digits = 0;
while (N > 0) // Если число N меньше либо =0, тело цикла не будет
                // выполнено
{   digits += 1;
    N = N / 10;
}
```

Третьей формой оператора цикла является цикл do while. Он имеет форму: do {операторы} while (условие);. Отличие от предыдущей формы цикла while заключается в том, что условие проверяется после выполнения тела цикла. Предположим, требуется прочитать символы с терминала до тех пор, пока не будет введен символ ‘*’.

```
char ch;
do
{   ch = getch();      // функция возвращает символ, введенный с
                       // клавиатуры
}   while(ch != ’*’);
```

В операторах while и do также можно использовать операторы break и continue. Как легко заметить, операторы цикла взаимозаменяемы. Оператор while соответствует оператору for:

```
for ( ; условие; )
    оператор
```

Пример чтения символов с терминала можно переписать в виде:

```
ch = getch();
while (ch != ’*’)
{    ch=getch ();
}
```

Разные формы нужны для удобства и наглядности записи.

Операторы возврата

Оператор возврата return завершает выполнение функции и возвращает управление в ту точку, откуда она была вызвана. Его форма:

```
return выражение;
```

где выражение – это результат функции. Если функция не возвращает никакого значения, то оператор возврата имеет форму: return;

Пример:

```
int fact (int n);
{ int k;
  if (n == 1)
        k=1;
  else
        k = n * fact(n-1);
return k;                    //возврат функции
}
```

Это функция вычисления факториала. Первый оператор в ней – это объявление переменной k, в которой будет храниться результат вычисления. Затем выполняется условие оператора if. Если n=1, то вычисление факториала закончено, и выполняется оператор – выражение, который присваивает переменной значение 1. В противном случае выполняется другой оператор – выражение.

Оператор перехода

Последовательность выполнения операторов в программе можно изменить с помощью оператора перехода goto. Он имеет вид: goto метка;. Метка ставится в программе, записывая ее имя и затем двоеточие. Например, вычислить абсолютную величину значения переменной *x* можно следующим способом:

```
if (x >= 0)
   goto positiv;
x = -x;
positive:
   abs = x;          // присваивание переменной abs положительное
                     // значение.
```

При выполнении goto вместо следующего оператора выполняется оператор, стоящий после метки positiv.

В настоящее время считается, что оператор goto очень легко запутывает программу. Без него, вообще говоря, можно обойтись, поэтому лучше его не использовать, ну разве лишь в самом крайнем случае.

Тема 5

Функции

Функции – это основные единицы построения программ при процедурном программировании на языке Си++.

Вызов функции

Функция вызывается при вычислении выражений. При вызове ей передаются определенные аргументы, функция выполняет необходимые действия и возвращает результат.

Программа на языке Си++ состоит, по крайней мере, из одной функции – функции main. С нее всегда начинается выполнение программы. Встретив имя функции в выражении, программа вызовет эту функцию, то есть передаст управление на ее начало и начнет выполнять операторы. Достигнув конца функции или оператора return – выхода из функции, управление вернется в ту точку, откуда функция была вызвана, подставив вместо нее вычисленный результат.

Прежде всего функцию необходимо объявить. Объявление функции, аналогично объявлению переменной, определяет имя функции и ее тип, типы и количество ее аргументов и тип возвращаемого значения.

Например:

```
double sqrt (double x);    // функция с одним аргументом –
                           // вещественным числом двойной точности,
                           // возвращает результат типа double.
int sum (int a, int b, int c); // функция трех целых аргументов, возвращает
                                  // целое число
```

Объявление функции называется иногда прототипом функции. После того, как функция объявлена, ее можно использовать в выражениях:

```
double x=sqrt(3)+1;
sum(k, l , m)/15;
```

Если функция не возвращает никакого результата, т. е. она объявлена как void, ее вызов не может быть использован как операнд более сложного выражения, и должен быть записан сам по себе: fun (a,b,c);.

Следующий шаг – определение функции, которое описывает, как она работает, т. е. какие действия надо выполнить, чтобы получить искомый результат. Для функции sum, объявленной выше, определение может быть следующим образом:

```
int sum (int a, int b, int c)
{
   int result;
   result = a+b+c;
   return result;
}
```

Первая строка – это заголовок функции, он совпадает с объявлением функции, за исключением того, что объявление заканчивается точкой с запятой. Далее в фигурных скобках заключено тело функции – действия, которые выполняет данная функция.

Аргументы a, b, c называются формальными параметрами. Это переменные, которые определены в теле функции (то есть к ним можно

обращаться только внутри фигурных скобок). При написании определения функций программа не знает их значения. При вызове функции вместо них подставляются фактические параметры – значения, с которыми функция вызывается. В примере вызова функции sum фактическими параметрами (или фактическими аргументами) являлись значения переменных *k, l* и *m*. Формальные параметры принимают значения фактических аргументов, заданных при вызове, и функция выполняется.

Первое, что мы делаем в теле функции – объявляем внутреннюю переменную result типа целое. Переменные, объявляющиеся в теле функции, также называются локальными. Это связано с тем, что переменная result существует только во время выполнения операторов тела функции sum. После завершения выполнения функции она уничтожается – ее имя становится неизвестным, и память, занимаемая этой переменной, освобождается.

Вторая строка определения функции – вычисление результата. Отметим, что до присваивания значения result было неопределенным (то есть значение переменной было неким произвольным числом (мусором)).

Последняя строка функции возвращает в качестве результата вычисленное значение.

```
int k = 2, l = 3, m = 5;
int s=sum(k,l,m);      // s = 10
```

Имена функций

В языке Си++ допустимо иметь несколько функций с одним и тем же именем, потому что функции различаются не только по именам, но и по типам аргументов. Если в дополнение к определенной выше функции sum мы определим еще одну функцию с тем же именем:

```
double sum (double a, double b, double c)
{ double result;
  result = a+b+c;
```

```
    return result;
}
```

это будет считаться новой функцией.

Иногда говорят, что у этих функций разные подписи. В следующем фрагменте программы в первый раз будет вызвана первая функция, а во второй раз – вторая:

```
int x, y, z, ires;
double p, q, s, dres;
…
ires = sum (x, y, z);   // вызвать первое определение функции
dres = sum (p, q, s);   // вызвать второе определение функции
```

При первом вызове функции sum все фактические аргументы имеют тип int. Поэтому вызывается первая функция. Во втором вызове все аргументы имеют тип double, соответственно, вызывается вторая функция.

Важен не только тип аргументов, но и их количество. Можно определить функцию sum, суммирующую два аргумента:

```
int sum (int x1, int x2)
{
return x1+x2;
}
```

Отметим, что при определении функций имеют значение тип и количество аргументов, но не тип возвращаемого значения. Попытка определения двух функций с одними и теми же аргументами, но различными возвращаемыми значениями, приведет к ошибке компиляции:

```
int fer(int x);
doble fer(int x);    // ошибка – двукратное определение   функции.
```

Необязательные аргументы функции

При объявлении функций в языке Си++ имеется возможность задать значения аргументов по умолчанию. Первый случай применения этой возможности языка – сокращение записи. Если функция вызывается с одним и тем же значением аргумента в 99% случаев, и это значение достаточно очевидно, можно задать его по умолчанию. Предположим, функция step возводит вещественное число в произвольную целую положительную степень. Чаще всего она используется для возведения в квадрат. Тогда ее объявление можно записать так:

```
double step (double x, unsigned int st=2);
```

Определение функции:

```
double step (double x, unsigned int st )
{ double result = 1;
  for (int i=0; i< st; i++)
       result *= x;
  return result;
}
main()
{ double y = step(3.141592);  //возведение числа пи в квадрат
  double x = step(2.9,5);        // 2.9 в пятой степени
}
```

Использовать аргумент по умолчанию удобно при изменении функции. Если при изменении программы нужно добавить новый аргумент, то для того, чтобы не изменять все вызовы этой функции, можно новый аргумент объявить со значением по умолчанию. В таком случае старые вызовы будут использовать значение по умолчанию, а новые – значения, указанные при вызове.

Необязательных аргументов может быть несколько. Если указан один необязательный аргумент, то либо он должен быть последним в прототипе, либо все аргументы после него должны также иметь значение по умолчанию.

Если для функции задан необязательный аргумент, то фактически задано несколько подписей этой функции. Например, попытка определения двух функций:

```
double step(double x, unsigned int  st=2);
double step(double x);
```

приведет к ошибке компиляции – неоднозначности определения функции. Это происходит потому, что вызов double x=step(4,1); подходит как для первой, так и для второй функции.

Рекурсия

Определение функций не могут быть вложенными, то есть нельзя внутри тела одной функции определить тело другой. Разумеется, можно вызвать одну функцию из другой. В том числе функция может вызвать сама себя.

Рассмотрим функцию вычисления факториала целого числа. Ее можно реализовать двумя способами. Первый способ использует итерацию:

```
int fact(int n)
{   int result = 1;
    for (int i = 1; i <= n; i++)
    result = result*i;
    return result;
}
```

Второй способ:

```
int fact(int n)
{   if (n == 1)
       return 1;              // факториал = 1
    else
       return n*fact(n-1);
}
```

Функция fact вызывает сама себя с модифицированными аргументами. Такой способ вычислений называется рекурсией. Рекурсия – это очень мощный метод вычислений. Значительная часть математических функций определяется в рекурсивных терминах. В программировании алгоритмы обработки сложных структур данных также часто бывают рекурсивными. Рассмотрим, например, структуру двоичного дерева. Дерево состоит из узлов и направленных связей. С каждым узлом могут быть связаны один или два узла, называемые сыновьями этого узла. Соответственно, для «сыновей» узел, из которого к ним идут связи, называется «отцом». Узлы, у которых нет сыновей, называются листьями. Рассмотрим пример дерева на рисунке:

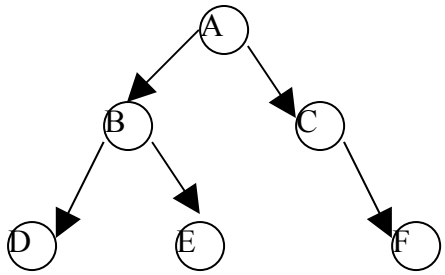


Рис. Пример дерева.

В этом дереве узел A – корень дерева. Узлы B и C – сыновья узла A, узлы D и E – сыновья узла B, узел F – сын узла C. Узлы D,E и F – листья. Обход дерева (прохождение по всем его узлам) можно описать таким образом:

1. Посетить корень дерева.

2. Обойти поддеревья с корнями – сыновьями данного узла, если у узла есть сыновья.

3. Если у узла нет сыновей – обход закончен.

Очевидно, что реализация такого алгоритма с помощью рекурсии не составит труда.

Довольно часто рекурсия и итерация взаимозаменяемы(как в примере с факториалом). Выбор между ними может быть обусловлен разными

факторами. Чаще рекурсия более наглядна и легче реализуется. Но в большинстве случаев итерация более эффективна.

Тема 6

Встроенные типы данных.

Рассматриваются все встроенные типы данных языка C++: целые числа разной разрядности, вещественные числа, логические величины, перечисляемые значения, символы и их кодировка.

Встроенные типы данных предопределены в языке. Это самые простые величины, из которых составляют все производные типы, в том числе и классы.

Различные реализации и компиляторы могут определять различные диапазоны значений целых и вещественных чисел.

Целые числа

Для представления целых чисел в языке C++ существует несколько типов – char, short, int и long (полное название типов: short int, long int, unsigned long int и т.д.). Поскольку описатель int можно опустить, то используется сокращенные названия. Они отличаются друг от друга диапазоном возможных значений. Каждый из этих типов может быть знаковым и беззнаковым. По умолчанию, тип целых величин – знаковый. Если перед определением типа стоит ключевое слово unsigned, то тип целого числа – беззнаковый. Для того, чтобы определить переменную *x* типа короткого целого числа, нужно записать

```
short x;          // диапазон значений -32768 до +32767.
```

Число без знака принимает только положительные значения и значение ноль. Число со знаком положительные значения, отрицательные значения и значение ноль.

Целое число может быть непосредственно записано в программе в виде констант: 0, 125, -37. По умолчанию целые константы принадлежат к типу int. Если необходимо указать, что целое число – это константа типа long, можно добавить символ *L* или *l* после числа. Если константа беззнаковая, то есть относится к типу unsigned long, short, int, после числа записывается символ *U* или *u*. Например: 34*U*, 7654*ul*.

Кроме стандартной десятичной записи, числа можно записывать в восьмеричной или шестнадцатеричной системе счисления. Признаком восьмеричной системы счисления является цифра 0 в начале числа. Признаком шестнадцатеричной – 0*x* или 0*X* перед числом. Для 16-х цифр используют латинские буквы A и F (неважно, большие или маленькие). Таким образом, фрагмент программы:

```
const int x =240;
const int y=0360;
const int z=0xF0;
```

определяет три целые константы x, y и z с одинаковыми значениями. Отрицательные числа предваряются знаком минус (-). Приведем ещё несколько примеров:

```
const unsigned long k=0678;   // ошибка в записи восьмеричного числа
const short a=0xa4;           // правильная запись
const int x=23F3;             // ошибка в записи десятичного числа
```

Для целых чисел определены стандартные арифметические операции: сложение +, вычитание -, умножение *, деление /, нахождение остатка от деления %, изменение знака -. Результатом этих операций также является целое число. При делении остаток отбрасывается.

Примеры выражений с целыми величинами: x+4; 30-x; x*2; -x; 10/x; x%3;

Кроме стандартных арифметических операций, для целых чисел определен набор битовых (или поразрядных) операций. В них целое число рассматривается как строка битов (нулей и единиц при записи числа в двоичной системе счисления или разрядов машинного представления).

К этим операциям относятся поразрядные операции: И, ИЛИ, исключающее ИЛИ, поразрядные отрицания и сдвиги. Поразрядная операция ИЛИ, обозначается символом |, выполняет операцию ИЛИ над каждым индивидуальным битом двух своих операндов. Например 1|2 в результате дают 3, так как 1 – 01, 2 – 10, соответственно операция ИЛИ дает 11 – 3 в десятичной системе счисления (нули слева были опущены).

Аналогично выполняется поразрядные операции И (&), исключающее ИЛИ (^) и отрицание (~). Пример 3\1 = 3, 4&7 = 4, 0&0xF = 0, ~0x00F0 = 0xFF0F

Операция сдвига перемещает двоичное представление левого операнда на количество битов, соответствующее значению правого операнда. Например двоичное представление короткого целого числа 3 – 0000000000000011. Результатом операции 3<<2 (сдвиг влево), будет двоичное число 0000000000001100 или в десятичной записи, 12. Аналогично, сдвинув число 9 (в двоичном виде 0000000000001001) на 2 разряда вправо (записав 9>>2) получим 0000000000000010, то есть 2.

При сдвиге влево число дополняется нулями справа. При сдвиге вправо бит, которым дополняется число, зависит от того, знаковое оно или беззнаковое. Для беззнаковый чисел при сдвиге вправо они всегда дополняются нулевым битом. Если же число знаковое, то значение самого левого бита используется для дополнения. Это объясняется тем, что самый левый бит как раз и является знаком – 0 означает + и 1 означает -. Таким образом, если short x = 0xFF00; unsigned short y = 0xFF00; то результатом

x>>2 будет 0xFFC0 (в двоичном представлении 1111 1111 1100 0000), а результатом y>>2 будет уже 0x3FC0 (0011 1111 1100 0000).

Рассмотренные арифметические и поразрядные операции выполняются над целыми числами и в результате дают целое число. В отличие от них операции сравнения выполняются над целыми числами, но в результате дают логическое значение истина (true) или ложь (false).

Для целых чисел определены операции сравнения: равенства (==), неравенства (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=).

Последний вопрос, который мы рассмотрим в отношении целых чисел, – это преобразование типов. В языке Си++ допустимо смешивать в выражении различные целые типы. Например, вполне допустимо записать $x + y$, где x типа short , а y – long . При выполнении операции сложения величина переменной x преобразуется к типу long . Такое преобразование можно произвести всегда, и оно безопасно, т.е. мы не теряем никаких значащих цифр. Общее правило преобразования целых типов состоит в том, что более короткий тип при вычислениях преобразуется в более длинный. Только при выполнении присваивания длинный тип может преобразовываться в более короткий. Например:

```
short x;
long y = 15;
. . .
x = y;            // преобразование длинного типа в более короткий
```

Такое преобразование не всегда безопасно, поскольку могут потеряться значащие цифры. Обычно компиляторы, встречая такое преобразование, выдают предупреждение или сообщение об ошибке.

Вещественные числа

Вещественные числа в Си++ могут быть одного из трех типов: с одинарной точностью — float , с двойной точностью – double , и с расширенной точностью – long double.

В большинстве реализаций языка представление и диапазоны значений соответствуют стандарту IEEE (Institute of Electrical and Electronics Engineers) для представления вещественных чисел. Точность представления чисел составляет 7 десятичных значащих цифр для типа float , 15 значащих цифр для double и 19 — для типа long double .

Вещественные числа записываются либо в виде десятичных дробей, например 1.3, 3.1415, 0.0005, либо в виде мантиссы и экспоненты: 1.2E0, 0.12e1. Отметим, что обе предыдущие записи изображают одно и то же число 1.2.

По умолчанию вещественная константа принадлежит к типу double . Чтобы обозначить, что константа на самом деле float , нужно добавить символ f или F после числа: 2.7f. Символ l или L означает, что записанное число относится к типу long double .

```
const float pi_f = 3.14f;
double pi_d = 3.1415;
long double pi_l = 3.1415L;
```

Для вещественных чисел определены все стандартные арифметические операции: сложения (+), вычитания (-), умножения (*), деления (/) и изменения знака (-). В отличие от целых чисел, операция нахождения остатка от деления для вещественных чисел не определена. Аналогично, все битовые операции и сдвиги к вещественным числам неприменимы; они работают только с целыми числами. Примеры операций:

```
2 * pi;
(x – e) / 4.0
```

Вещественные числа можно сравнивать на равенство (==), неравенство (!=), больше (>), меньше (<), больше или равно (>=) и меньше или равно (<=). В результате операции сравнения получается логическое значение истина или ложь.

Если арифметическая операция применяется к двум вещественным числам разных типов, то менее точное число преобразуется в более точное, т.е. float преобразуется в double и double преобразуется в long double . Очевидно, что такое преобразование всегда можно выполнить без потери точности.

Если вторым операндом в операции с вещественным числом является целое число, то целое число преобразуется в вещественное представление. Хотя любую целую величину можно представить в виде вещественного числа, при таком преобразовании возможна потеря точности (для больших чисел).

Логические величины

В языке Си++ существует специальный тип для представления логических значений bool . Для величин этого типа существует только два возможных значения: true (истина) и false (ложь). Объявление логической переменной выглядит следующим образом:

```
bool condition;
```

Соответственно, существуют только две логические константы – истина (true) и ложь (false).

Для типа bool определены стандартные логические операции: логическое И (&&), ИЛИ (||) и НЕ (!). Например,

```
cond1 && cond2     // истинно, если обе переменные,
                   // cond1 и cond2, истинны
```

```
cond1 || cond2        // истинно, если хотя бы одна из переменных
                      // истинна
!cond1                // результат противоположен значению cond1
```

Как мы уже отмечали ранее, логические значения получаются в результате операций сравнения. Кроме того, в языке Си++ принято следующее правило преобразования чисел в логические значения: ноль соответствует значению false , и любое отличное от нуля число преобразуется в значение true . Поэтому можно записать, например:

```
int k = 100;
while (k)     // выполнить цикл 100 раз
{   k--;
}
```

Символы и байты

Символьный или байтовый тип в языке Си++ относится к целым числам, однако мы выделили их в особый раздел, потому что запись знаков имеет свои отличия.

Итак, для записи знаков в языке Си++ служат типы char и unsigned char . Первый – это целое число со знаком, хранящееся в одном байте, второй – беззнаковое байтовое число. Эти типы чаще всего используются для манипулирования символами, поскольку коды символов как раз помещаются в байт.

Пояснение. *Единственное, что может хранить компьютер, это числа. Поэтому для того чтобы можно было хранить символы и манипулировать ими, символам присвоены коды – целые числа. Существует несколько стандартов, определяющих, какие коды каким символам соответствуют. Для английского алфавита и знаков препинания используется стандарт ASCII. Этот стандарт определяет коды от 0 до 127. Для представления*

русских букв используется стандарт КОИ-8 или СР-1251. В этих стандартах русские буквы кодируются числами от 128 до 255. Таким образом, все символы могут быть представлены в одном байте (максимальное число символов в одном байте – 255). Для работы с китайским, японским, корейским и рядом других алфавитов одного байта недостаточно, и используется кодировка с помощью двух байтов и, соответственно, тип wchar_t:

Чтобы объявить переменную байтового типа, нужно записать:

```
char c;                 // байтовое число со знаком

unsigned char u;   // байтовое число без знака
```

Поскольку байты – это целые числа, то все операции с целыми числами применимы и к байтам. Стандартная запись целочисленных констант тоже применима к байтам, т.е. можно записать:

```
c = 45;
```

где c — байтовая переменная. Однако для байтов существует и другая запись констант. Знак алфавита (буква, цифра, знак препинания), заключенный в апострофы, представляет собой байтовую константу, например: 'S', '&', '8', 'ф'.

Числовым значением такой константы является код данного символа, принятый в вашей операционной системе.

В кодировке ASCII два следующих оператора эквивалентны:

```
char c = 68;
char c = 'D';
```

Первый из них присваивает байтовой переменной с значение числа 68. Второй присваивает этой переменной код латинской буквы D, который в кодировке ASCII равен 68.

Для обозначения ряда непечатных символов используются так называемые экранированные последовательности – знак обратной дробной черты, после которого стоит буква. Эти последовательности стандартны и заранее предопределены в языке:

\f перевод страницы
\n новая строка
\r перевод каретки
\t горизонтальная табуляция
\v вертикальная табуляция
\? вопросительный знак
и другие.

Для того чтобы записать произвольное байтовое значение, также используется экранированная последовательность: после обратной дробной черты записывается целое число от 0 до 255.

```
char zero = '\0';
const unsigned char bitmask = '\0xFF';
char tab = '\010';
```

Следующая программа выведет все печатные символы ASCII и их коды в порядке увеличения:

```
int i;
for (char c = 32; c < 127; c++)
{  i = int (c);
   cout << c << " - " << i << "; ";
```

```
}
```

Однако напомним еще раз, что байтовые величины – это, прежде всего, целые числа, поэтому вполне допустимы выражения вида

```
'F' + 1
'a' < 23
```

и тому подобные.

Тип char был придуман для языка Си, от которого Си++ достались все базовые типы данных. Язык Си предназначался для программирования на достаточно "низком" уровне, приближенном к тому, как работает процессор компьютера, именно поэтому символ в нем – это лишь число.

В языке Си++ в большинстве случаев для работы с текстом используются специально разработанные классы строк.

Кодировка, многобайтовые символы

Мы уже упоминали о наличии разных кодировок букв, цифр, знаков препинания и т.д. Алфавит большинства европейских языков может быть представлен однобайтовыми числами (т.е. кодами в диапазоне от 0 до 255). В большинстве кодировок принято, что первые 127 кодов отводятся для символов, входящих в набор ASCII: ряд специальных символов, латинские заглавные и строчные буквы, арабские цифры и знаки препинания. Вторая половина кодов – от 128 до 255 отводится под буквы того или иного языка. Фактически, вторая половина кодовой таблицы интерпретируется по-разному, в зависимости от того, какой язык считается "текущим". Один и тот же код может соответствовать разным символам в зависимости от того, какой язык считается "текущим".

Однако для таких языков, как китайский, японский и некоторые другие, одного байта недостаточно – алфавиты этих языков насчитывают более 255 символов.

Перечисленные выше проблемы привели к созданию многобайтовых кодировок символов. Двухбайтовые символы в языке Си++ представляются с помощью типа wchar_t:

```
wchar_t wch;
```

Тип wchar_t иногда называют расширенным типом символов, и детали его реализации могут варьироваться от компилятора к компилятору, в том числе может меняться и количество байт, которое отводится под один символ. Тем не менее, в большинстве случаев используется именно двухбайтовое представление.

Константы типа wchar_t записываются в виде L'ab'.

Наборы перечисляемых значений

Достаточно часто в программе вводится тип, состоящий лишь из нескольких заранее известных значений. Например, в программе используется переменная, хранящая величину, отражающую время суток, и мы решили, что будем различать ночь, утро, день и вечер. Конечно, можно договориться обозначить время суток числами от 1 до 4. Но, во-первых, это не наглядно. Во-вторых, что даже более существенно, очень легко сделать ошибку и, например, использовать число 5, которое не соответствует никакому времени дня. Гораздо удобней и надежнее определить набор значений с помощью типа enum языка Си++:

```
enum DayTime { morning, day, evening, night };
```

Теперь можно определить переменную

DayTime current;

которая хранит текущее время дня, а затем присваивать ей одно из допустимых значений типа DayTime:

current = day;

Контроль, который осуществляет компилятор при использовании в программе этой переменной, гораздо более строгий, чем при использовании целого числа.

Для наборов определены операции сравнения на равенство (==) и неравенство (!=) с атрибутами этого же типа, т.е.

```
if (current != night)
        // выполнить работу
```

Вообще говоря, внутреннее представление значений набора – целые числа. По умолчанию элементам набора соответствуют последовательные целые числа, начиная с 0. Этим можно пользоваться в программе. Во-первых, можно задать, какое число какому атрибуту набора будет соответствовать:

```
enum { morning = 4, day = 3, evening = 2, night = 1 };
enum { morning = 1, day, evening, night }; // последовательные числа, начиная с 1.
enum { morning, day = 2, evening, night }; // используются числа 0, 2, 3, 4
```

Во-вторых, атрибуты наборов можно использовать в выражениях вместо целых чисел. Преобразования из набора в целое и наоборот разрешены.

Однако, этого делать не рекомендуется. Для работы с целыми константами лучше применять символические обозначения констант, а наборы использовать по их прямому назначению.

Тема 7

Классы и обьекты.

Способы описания классов. Создание объектов. Обращение к атрибутам и методам объектов.

Понятие класса

До сих пор мы говорили о встроенных типах, т.е. типах, определенных в самом языке. Классы – это типы, определенные в конкретной программе. Определение класса включает в себя описание, из каких составных частей или атрибутов он состоит и какие операции определены для класса.

Предположим, в программе необходимо оперировать комплексными числами. Комплексные числа состоят из вещественной и мнимой частей, и с ними можно выполнять арифметические операции.

```
class Complex
{
  public:
    int real;              // вещественная часть
    int imaginary;     // мнимая часть
    void Add(Complex x);          // прибавить комплексное число
```

```
};
```

Приведенный выше пример – упрощенное определение класса под именем Complex, представляющее комплексное число. Комплексное число состоит из вещественной части – целого числа real и мнимой части, которая представлена целым числом imaginary. real и imaginary - это атрибуты класса. Для класса Complex определена одна операция или метод – Add. Определив класс, теперь мы можем создать переменную типа Complex:

```
Complex number;
```

Переменная с именем number содержит значение типа Complex, то есть содержит объект класса Complex. Имея объект, мы можем установить значения атрибутов объекта:

```
number.real = 1;
number.imaginary = 2;
```

Операция точка "." обозначает обращение к атрибуту объекта. Создав еще один объект класса Complex, мы можем прибавить его к первому:

```
Complex number2;
number.Add(number2);
```

Как можно заметить, метод Add выполняется с объектом. Имя объекта (или переменной, содержащей объект, что, в сущности, одно и то же), в данном случае, number, записано первым. Через точку записано имя метода – Add с аргументом – значением другого объекта класса Complex, который прибавляется к number. Методы часто называются сообщениями. Но чтобы послать сообщение, необходим получатель. Таким образом, объекту number

посылается сообщение Add с аргументом number2. Объект number принимает это сообщение и складывает свое значение со значением аргумента сообщения.

Определение методов класса.

Чтобы данные рассуждения стали яснее, нужно определить, как будет выполняться операция сложения:

```
void
Complex::Add(Complex x)
{
        this -> real = this -> real + x.real;
        this -> imaginary = this -> imaginary + x.imaginary;
}
```

Первая строка говорит о том, что это метод Add класса Complex. В фигурных скобках записано определение операции или метода Add. Это определение означает следующее: для того чтобы прибавить значение объекта класса Complex к данному объекту, надо сложить вещественные части и запомнить результат в атрибуте вещественной части текущего объекта. Точно так же следует сложить мнимые части двух комплексных чисел и запомнить результат в атрибуте текущего объекта, обозначающем мнимую часть.

Запись this -> говорит о том, что атрибут принадлежит к тому объекту, который выполняет метод Add (объекту, получившему сообщение Add). В большинстве случаев this -> можно опустить. В записи определения метода какого-либо класса упоминание атрибута класса без всякой дополнительной информации означает, что речь идет об атрибуте текущего объекта.

Теперь приведем этот небольшой пример полностью:

```
// определение класса комплексных чисел
class Complex
{
 public:
    int real;              // вещественная часть
    int imaginary;     // мнимая часть
    void Add(Complex x);      // прибавить комплексное число
};
// определение метода сложения
void Complex::Add(Complex x)
{
    real = real + x.real;
    imaginary = imaginary + x.imaginary;
}
void main()
{
Complex number;            // первый объект класса Complex

    number.real = 1;
    number.imaginary = 3;
    Complex number2;        // второй объект класса Complex
    number2.real = 2;
    number2.imaginary = 1;
    number.Add(number2);  // прибавить значение второго
                                  // объекта к первому
}
```

Переопределение операций.

В языке Си++ можно сделать так, что класс будет практически неотличим от предопределенных встроенных типов при использовании в

выражениях. Для класса можно определить операции сложения, умножения и т.д. пользуясь стандартной записью таких операций, как например, x + y. В языке Си++ считается, что подобная запись – это также вызов метода с именем **operator+** того класса, к которому принадлежит переменная x. Перепишем определение класса Complex:

```
class Complex
{
  public:
        int real;
        int imaginary;
        Complex operator+ (const Complex x) const;
};
```

Здесь вместо метода Add появился метод operator+. Изменилось и его определение. Во-первых, этот метод возвращает значение типа Complex (операция сложения в результате дает новое значение того же типа, что и типы операндов). Во-вторых, перед аргументом метода появилось ключевое слово const. Это слово обозначает, что при выполнении данного метода аргумент изменяться не будет. Также const появилось после объявления метода. Второе ключевое слово const означает, что объект, выполняющий метод, не будет изменен. При выполнении операции сложения x + y над двумя величинами x и y сами эти величины не изменяются. Теперь запишем определение операции сложения:

```
Complex
Complex::operator+(const Complex x) const
{
  Complex result;
  result.real = real + x.real;
```

```
  result.imaginary = imaginary + x.imaginary;
  return result;
}
```

Подписи методов и необязательные аргументы.

Как и при объявлении функций, язык Си++ допускает определение в одном классе нескольких методов с одним и тем же именем, но разными типами и количеством аргументов. (Определение методов или атрибутов с одинаковыми именами в разных классах не вызывает проблем, поскольку пространства имен разных классов не пересекаются).

```
class Complex          // определение класса комплексных чисел
{
public:
  int real;
  int imaginary;
    Complex operator+(const Complex x) const;   // прибавить комплексное
                                                // число
    Complex operator+(long x) const;            // прибавить целое число
};
```

В следующем примере вначале складываются два комплексных числа, и вызывается первая операция +. Затем к комплексному числу прибавляется целое число, и тогда выполняется вторая операция сложения.

```
Complex c1;
Complex c2;
long x;
c1 + c2;
c2 + x;
```

Запись классов.

Как уже отмечалось раньше, выбор имен – это не праздный вопрос. Существует множество систем именования классов. Опишем ту, которой мы придерживаемся в данном курсе лекций.

Имена классов, их методов и атрибутов составляются из английских слов, описывающих их смысл, при этом, если слов несколько, они пишутся слитно. Имена классов начинаются с заглавной буквы; если название состоит из нескольких слов, каждое слово начинается с заглавной буквы, остальные буквы маленькие:

Complex, String, StudentLibrarian;

Имена методов классов также начинаются с большой буквы:

Add, Concat

Имена атрибутов класса начинаются с маленькой буквы, однако если имя состоит из нескольких слов, последующие слова начинаются с большой:

real, classElement

При записи определения класса мы придерживаемся той же системы расположения, что и при записи функций. Ключевое слово class и имя класса записываются в первой строке, открывающаяся фигурная скобка – на следующей строке, методы и атрибуты класса – на последующих строках с отступом.

Тема 8

Производные типы данных.

Создание и использование массивов, структур, объединений, указателей. Адресная арифметика. Строки и литералы.

Массивы.

Массив – это совокупность нескольких величин одного и того же типа, объединенных под одним именем и различающихся своими индексами. Примером массива может служить набор из двенадцати целых чисел, соответствующих числу дней в каждом календарном месяце:

```
int days[12];
days[0] = 31;     // январь
days[1] = 28;     // февраль
…
days[11] = 31;    // декабрь
```

В первой строчке мы объявили массив из 12 элементов типа int и дали ему имя days. Остальные строки примера – присваивания значений элементам массива. Для того, чтобы обратиться к определенному элементу массива, используют операцию индексации []. Как видно из примера, первый элемент массива имеет индекс 0, соответственно, последний – 11.

При объявлении массива его размер должен быть известен в момент компиляции, поэтому в качестве размера можно указывать только целую константу или константное выраженик. При обращении же к элементу массива в роли значения индекса может выступать любая переменная или выражение, которое вычисляется во время выполнения программы и преобразуется к целому значению.

Предположим, мы хотим распечатать все элементы массива days. Для этого удобно воспользоваться циклом for.

```
for (int i = 0; i < 12; i++)
```

```
cout << days[i] << “  “;
```

Следует отметить, что при выполнении программы границы массива не контролируются. Если мы ошиблись и вместо 12 в приведенном выше цикле написали 13, то компилятор не выдаст ошибку и при выполнении программа попытается напечатать 13-е число. Что при этом случится, вообще говоря, не определено. Быть может, произойдет сбой программы. Более вероятно, что будет напечатано какое-то случайное 13-е число. Выход индексов за границы массива – довольно распространенная ошибка, которую иногда очень трудно обнаружить.

Отсутствие контроля индексов налагает на программиста большую ответственность. С другой стороны, индексация – настолько часто используемая операция, что наличие контроля, несомненно, повлияло бы на производительность программ.

Рассмотрим еще один пример. Предположим, что имеется массив из 100 целых чисел, и его необходимо отсортировать, т.е. расположить в порядке возрастания. Сортировка методом "пузырька" – наиболее простая и распространенная – будет выглядеть следующим образом:

```
int array[100];
. . .
for (int i = 0; i < 99; i++ )
{    for (int j = i + 1; j < 100; j++)
     {    if (array[j] < array[i] )
          {  int tmp = array[j];
             array[j] = array[i];
             array[i] = tmp;
          }
     }
}
```

В приведенных примерах у массивов имеется только один индекс. Такие одномерные массивы часто называются векторами. Имеется возможность определить массивы с несколькими индексами или размерностями. Например, объявление

```
int m[10][5];
```

представляет матрицу целых чисел размером 10 на 5. По-другому интерпретировать приведенное выше объявление можно как массив из 10 элементов, каждый из которых – вектор целых чисел длиной 5. Общее количество целых чисел в массиве m равно 50.

Обращение к элементам многомерных массивов аналогично обращению к элементам векторов: m[1][2] обращается к третьему элементу второй строки матрицы m.

Количество размерностей в массиве может быть произвольным. Как и в случае с вектором, при объявлении многомерного массива все его размеры должны быть заданы константами.

При объявлении массива можно присвоить начальные значения его элементам (инициализировать массив). Для вектора это будет выглядеть следующим образом:

```
int days[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

При инициализации многомерных массивов каждая размерность должна быть заключена в фигурные скобки:

```
double temp[2][3] = {
    { 3.2, 3.3, 3.4 },
    { 4.1, 3.9, 3.9 } };
```

Особенностью инициализации многомерных массивов является возможность не задавать размеры всех измерений массива, кроме самого последнего. Приведенный выше пример можно переписать так:

```
double temp[][3] = {
    { 3.2, 3.3, 3.4 },
    { 4.1, 3.9, 3.9 } };
```

Можно вычислить размер пропущенной размерности:

```
const int size_first = sizeof (temp) / sizeof (double[3]);
```

Структуры.

Структуры – это не что иное, как классы, у которых разрешен доступ ко всем их элементам (доступ к определенным атрибутам класса может быть ограничен). Рассмотрим пример структуры:

```
struct Record
{   int number;
    char name[20];
};
```

Так же, как и для классов, операция точка (".") обозначает обращение к элементу структуры.

В отличие от классов, можно определить переменную-структуру без определения отдельного типа:

```
struct {
    double x;
    double y;
} coord;
```

Обратиться к атрибутам переменной coord можно coord.x и coord.y.

Битовые поля

В структуре можно определить размеры атрибута с точностью до бита. Традиционно структуры используются в системном программировании для описания регистров аппаратуры. В них каждый бит имеет свое значение. Не менее важной является возможность экономии памяти – ведь минимальный тип атрибута структуры это байт (char), который занимает 8 битов. До сих пор, несмотря на мегабайты и даже гигабайты оперативной памяти, используемые в современных компьютерах, существует немало задач, где каждый бит на счету.

Если после описания атрибута структуры поставить двоеточие и затем целое число, то это число задает количество битов, выделенных под данный атрибут структуры. Такие атрибуты называют битовыми полями. Следующая структура хранит в компактной форме дату и время дня с точностью до секунды.

```
struct TimeAndDate
{
  unsigned hours    :5; // часы от 0 до 24 (5 битов)
  unsigned mins     :6; // минуты (6 битов)
  unsigned secs     : 6; // секунды от 0 до 60
  unsigned weekDay  :3; // день недели
  unsigned monthDay :6; // день месяца от 1 до 31
  unsigned month    :5; // месяц от 1 до 12
  unsigned year     :  8; // год от 0 до 100
};
```

Одна структура TimeAndDate требует всего 39 битов, т.е. около 5 байтов (1 байт – 8 бит). Если бы мы использовали для каждого атрибута этой структуры тип char, нам бы потребовалось 7 байт.

Объединения.

Особым видом структур данных является объединение. Определение объединения напоминает определение структуры, только вместо ключевого слова struct используется unio:

```
union number
{
        short sx;
        long lx;
        double dx;
};
```

В отличие от структуры, все атрибуты объединения располагаются по одному адресу. Под объединение выделяется столько памяти, сколько нужно для хранения наибольшего атрибута объединения. Объединения применяются в тех случаях, когда в один момент времени используется только один атрибут объединения и, прежде всего, для экономии памяти. Предположим, нам нужно определить структуру, которая хранит «универсальное» число, т. е. число одного из предопределенных типов и признак типа. Это можно сделать следующим образом:

```
struct Value
{ enum NumberType { ShortType, LongType, DoubleType};
        NumberType  type;
        short sx;                   // если type = ShortType
        long lx;                    // если type = LongType
        double dx;                  // если type = DoubleType
};
```

Атрибут type содержит тип хранимого числа, а соответствующий атрибут структуры – значение числа.

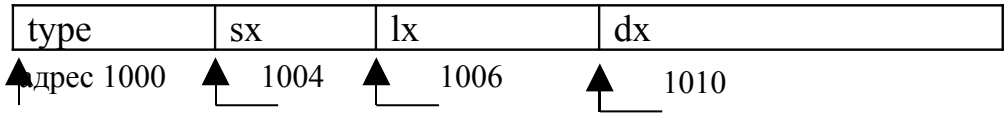
```
Value shortVal;
shortVal.type = Value::ShortType;
shortVal.sx = 15;
```

Хотя память выделяется под все три атрибута sx, lx, dx, реально используется только один из них. Сэкономить память можно, используя объединение:

```
struct Value
{ enum NumberType {ShortType, LongType, DoubleType};
  NumberType  type;
  union number
  { short sx; long lx; double dx; } val;
};
```

Теперь память выделена только для max из этих атрибутов (в данном случае dx). Однако обращаться с объединением надо осторожно (внимательно). Поскольку все три атрибута делят одну и ту же область памяти, изменение одного из них означает изменение всех остальных. Рассмотрим на рисунке выделение памяти под объединение. Будем предполагать, что структура расположена по адресу 1000. Объединение располагает все три своих атрибута по одному и тому же адресу.

Расположение при первом варианте структуры



Расположение при втором варианте структуры

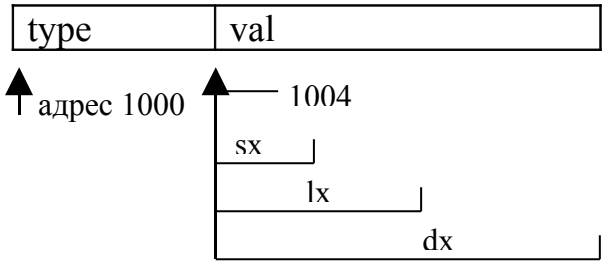


Рис. Использование памяти в объединении

Замечание: *объединения существовали в языке Си, откуда без изменений перешли в Си++. Использование наследования классов, позволяет во многих случаях добиться того же эффекта без использования объединений, причем программа будет более надежной.*

Указатели.

Указатель – это производный тип, который представляет собой адрес какого-либо значения. В языке Си++ используется понятие адреса переменных. Работа с адресами досталась Си++ в наследство от языка Си. Предположим, что в программе определена переменная типа int :

```
int x;
```

Можно определить переменю типа «указатель» на целое число: int* xptr; и присвоить переменной xptr адрес переменной x: xptr = &x;

Операция &, примененная к переменной – это операция взятия адреса.

Операция *, примененная к адресу – это обращение по адресу. Таким образом, два оператора эквивалентны:

```
int y = x;
int y = *xptr;    // присвоить переменной y значение, находящееся по
                  // адресу xptr
```

С помощью операций обращения по адресу можно записывать значения:

```
*xptr = 10;          // записать число 10 по адресу xptr
```

После выполнения этого оператора значение переменной *x* станет равным 10, поскольку xptr указывает на переменную *x*.

Указатель – это не просто адрес, а адрес величины определенного типа. Указатель xptr – адрес целой величины. Определить адреса величины других типов можно следующим образом:

```
unsigned long* lptr;      // указатель на целое число без знака
char* cp;                 // указатель на байт
Complex* p;               // указатель на объект класса Complex (комплекс)
```

Если указатель ссылается на объект, некоторого класса, то операция обращения к атрибуту класса вместо точки обозначается “->”, например p->real. Вспомним один из предыдущих примеров:

```
void Complex::Add(Complex x)
{
        this -> real = this -> real + x.real;
        this -> imaginary = this -> imaginary + x.imaginary;
}
```

Таким образом this – это указатель на текущий объект, то есть объект, который выполняет метод Add. Запись this -> означает обращение к атрибуту текущего объекта.

Можно определить указатель на любой тип, в том числе на функцию или метод класса. Если имеется несколько функций одного и того же типа:

```
int foo (long x);
int bar (long x);
```

можно определить переменную типа указатель на функцию и вызывать эти функции не напрямую, а косвенно, через указатель:

```
int (*functptr)(long x);
functptr = &foo;
(*functptr)(2);
functptr = &bar;
```

```
(*functptr)(4);
```

Для чего нужны указатели? Указатели появились, прежде всего, для нужд системного программирования. Поскольку язык Си предназначался для "низкоуровневого" программирования, на нем нужно было обращаться, например, к регистрам устройств. У этих регистров вполне определенные адреса, т.е. необходимо было прочитать или записать значение по определенному адресу. Благодаря механизму указателей, такие операции не требуют никаких дополнительных средств языка. Например,

```
int* hardwareRegiste =0x80000;
*hardwareRegiste =12;
```

Однако использование указателей нуждами системного программирования не ограничивается. Указатели позволяют существенно упростить и ускорить ряд операций. Предположим, в программе имеется область памяти для хранения промежуточных результатов вычислений. Эту область памяти используют разные модули программы. Вместо того, чтобы каждый раз при обращении к модулю копировать эту область памяти, мы можем передавать указатель в качестве аргумента вызова функции, тем самым упрощая и ускоряя вычисления.

```
struct TempResults
{   double x1;
    double x2;
} tempArea;
  // Рассматриваемая далее функция calc возвращает истину, если
  // вычисления были успешны, и ложь – при
  // наличии ошибки. Вычисленные результаты
  // записываются на место аргументов по
  // адресу, переданному в указателе trPtr
bool calc(TempResults* trPtr)
{
    if (noerrors)          // вычисления
```

```
    {  trPtr -> x1 = res1;
       trPtr -> x2 = res2;
       return true;
    }
     else {
        return false;
    }
}
void fun1(void)
{   . . .
    TempResults tr;
    tr.x1 = 3.4;
    tr.x2 = 5.4;
    if (calc(&tr) == false)
    {                   // обработка ошибки
    }
    . . .
}
```

В приведенном примере проиллюстрированы сразу две возможности использования указателей: передача адреса общей памяти и возможность функции иметь более одного значения в качестве результата. Структура TempResults используется для хранения данных. Вместо того чтобы передавать эти данные по отдельности, в функцию calc передается указатель на структуру. Таким образом достигаются две цели: большая наглядность и большая эффективность (не надо копировать элементы структуры по одному). Функция calc возвращает булево значение – признак успешного завершения вычислений. Сами же результаты вычислений записываются в структуру, указатель на которую передан в качестве аргумента.

Упомянутые примеры использования указателей никак не связаны с объектно-ориентированным программированием. Казалось бы, объектно-

ориентированное программирование должно уменьшить зависимость от низкоуровневых конструкций типа указателей. На самом деле программирование с классами нисколько не уменьшило потребность в указателях, и даже наоборот, нашло им дополнительное применение, о чем будет рассказано по ходу изложения лекций.

Адресная арифметика

С указателями можно выполнять не только операции присваивания и обращения по адресу, но и ряд арифметических операций. Прежде всего, указатели одного и того же типа можно сравнивать с помощью стандартных операций сравнения. При этом сравниваются значения указателей, а не значения величин, на которые данные указатели ссылаются. Так, в приведенном ниже примере результат первой операции сравнения будет ложным:

```
int x = 10;
int y = 10;
int* xptr = &x;
int* yptr = &y;
// сравниваем указатели
if (xptr == yptr)
   cout << "Указатели равны" << endl;
else
   cout << "Указатели неравны" << endl;
// сравниваем значения, на которые указывают
// указатели
if (*xptr == *yptr)
    cout << "Значения равны" << endl;
else
    cout << "Значения неравны" << endl;
```

Результат второй операции сравнения будет истинным, поскольку переменные *x* и *y* имеют одно и то же значение.

Кроме того, над указателями можно выполнять ограниченный набор арифметических операций. К указателю можно прибавить целое число или вычесть из него целое число. Результатом прибавления к указателю единицы является адрес следующей величины типа, на который ссылается указатель, в памяти. Поясним это на рисунке. Пусть xPtr – указатель на целое число типа long, а cp – указатель на тип char (char* cp). Начиная с адреса 1000, в памяти расположены два целых числа. Адрес второго — 1004 (в большинстве реализаций Си++ под тип long выделяется четыре байта). Начиная с адреса 2000, в памяти расположены объекты типа char.

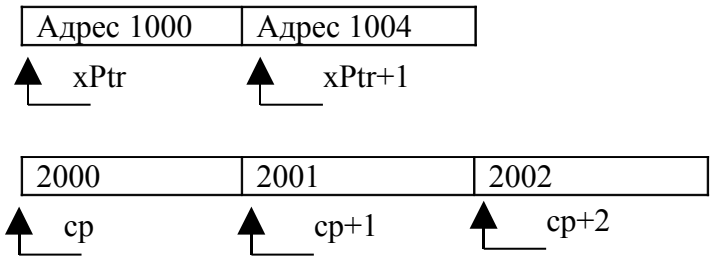


Рис. Адресная арифметика

Размер памяти, выделяемой для числа типа long и char, различен. Поэтому адрес при увеличении xPtr и cp тоже изменяется по-разному. Однако и в том, и в другом случае увеличение указателя на единицу означает переход к следующей в памяти величине того же типа. Прибавление или вычитание любого целого числа работает по тому же принципу, что и увеличение на единицу. Указатель сдвигается вперед (при прибавлении положительного числа) или назад (при вычитании положительного числа) на соответствующее количество объектов того типа, на который показывает указатель. Вообще говоря, неважно, объекты какого типа на самом деле находятся в памяти – адрес просто увеличивается или уменьшается на необходимую величину. Таким образом, значение указателя ptr всегда изменяется на число, равное sizeof(*ptr).

Указатели одного и того же типа можно друг из друга вычитать. Разность указателей показывает, сколько объектов соответствующего типа может поместиться между указанными адресами.

Связь между массивами и указателями

Между указателями и массивами существует определенная связь. Предположим, имеется массив из 100 целых чисел. Запишем двумя способами программу суммирования элементов этого массива:

```
long array[100];
long sum = 0;
for (int i = 0; i < 100; i++)
    sum += array[i];
```

То же самое можно сделать с помощью указателей:

```
long array[100];
long sum = 0;
for (long* ptr = &array[0]; ptr < &array[99] + 1; ptr++)
    sum += *ptr;
```

Элементы массива расположены в памяти последовательно, и увеличение указателя на единицу означает смещение к следующему элементу массива. Упоминание имени массива без индексов преобразуется в адрес его первого элемента:

```
for (long* ptr = array; ptr < array[99] + 1; ptr++)
    sum += *ptr;
```

Хотя смешивать указатели и массивы можно, но такой стиль не рекомендуется, особенно начинающим программистам.

При использовании многомерных массивов указатели позволяют обращаться к срезам или подмассивам. Если мы объявим трехмерный массив exmpl:

```
long exmpl[5][6][7];
```

то выражение вида exmpl[1][1][2] – это целое число, exmpl[1][1] – вектор целых чисел (адрес первого элемента вектора, т.е. имеет тип *long), exmpl[1] – двухмерная матрица или указатель на вектор (тип (*long)[7]). Таким образом, задавая не все индексы массива, мы получаем указатели на массивы меньшей размерности.

Бестиповый указатель

Особым случаем указателей является бестиповый указатель. Ключевое слово void используется для того, чтобы показать, что указатель означает просто адрес памяти, независимо от типа величины, находящейся по этому адресу:

```
void* ptr;
```

Для указателя на тип void не определена операция ->, не определена операция обращения по адресу *, не определена адресная арифметика. Использование бестиповых указателей ограничено работой с памятью при использовании ряда системных функций, передачей адресов в функции, написанные на языках программирования более низкого уровня, например на ассемблере.

В программе на языке Си++ бестиповый указатель может применяться там, где адрес интерпретируется по-разному, в зависимости от каких-либо динамически вычисляемых условий. Например, приведенная ниже функция будет печатать целое число, содержащееся в одном, двух или четырех байтах, расположенных по передаваемому адресу:

```
void printbytes(void* ptr, int nbytes)
{
 if (nbytes == 1)
 {
   char* cptr = (char*) ptr;
   cout << *cptr;
 }
```

```
  else if (nbytes == 2)
        { short* sptr = (short*) ptr;
         cout << *sptr;
        }
        else if (nbytes == 4)
              { long* lptr = (long*)ptr;
                cout << *lptr;
              }
              else cout << "Неверное значение аргумента";
}
```

В примере используется операция явного преобразования типа. Имя типа, заключенное в круглые скобки, стоящее перед выражением, преобразует значение этого выражения к указанному типу. Разумеется, эта операция может применяться к любым указателям (int, char и т.д.)

Нулевой указатель

В программах на языке Си++ значение указателя, равное нулю, используется в качестве "неопределенного" значения. Например, если какая-то функция вычисляет значение указателя, то чаще всего нулевое значение возвращается в случае ошибки.

```
long* foo(void);
. . .
long* resPtr;
if ((resPtr = foo()) != 0)
{       // использовать результат
} else
  {     // ошибка
  }
```

В языке Си++ определена символическая константа NULL для обозначения нулевого значения указателя.

Такое использование нулевого указателя было основано на том, что по адресу 0 данные программы располагаться не могут, он зарезервирован операционной системой для своих нужд. Однако во многом нулевой указатель – просто удобное соглашение, которого все придерживаются.

Строки и литералы

Для того чтобы работать с текстом, в языке Си++ не существует особого встроенного типа данных. Текст представляется в виде последовательности знаков (байтов), заканчивающейся нулевым байтом. Иногда такое представление называют Си-строки, поскольку оно появилось в языке Си. Кроме того, в Си++ можно создать классы для более удобной работы с текстами (готовые классы для представления строк имеются в стандартной библиотеке шаблонов).

Строки представляются в виде массива байтов:

```
char string[20];
string[0] = 'М';
string[1] = 'и';
string[2] = 'р';
string[3] = 0;
```

В массиве string записана строка "Мир". При этом были использованы только 4 из 20 элементов массива.

Для записи строковых констант в программе используются литералы. Литерал – это последовательность знаков, заключенная в двойные кавычки:

"Это строка", "012345678910", "*".

Заметим, что символ, заключенный в двойные кавычки, отличается от символа, заключенного в апострофы. Литерал "*" обозначает два байта: первый байт содержит символ звездочки, второй байт содержит ноль. Константа '*' обозначает один байт, содержащий знак звездочки.

С помощью литералов можно инициализировать массив:

```
char alldigits[] = "0123456789";
```

Размер массива явно не задан, он определяется исходя из размера инициализирующего его литерала, в данном случае 11 (10 символов плюс нулевой байт).

При работе со строками особенно часто используется связь между массивами и указателями. Значение литерала – это массив неизменяемых байтов нужного размера. Строковый литерал может быть присвоен указателю на char:

```
const char* message = "Сообщение программы";
```

Значение литерала – это адрес его первого байта, указатель на начало строки. В следующем примере функция CopyString копирует первую строку во вторую:

```
void CopyString(char* src, char* dst)
{
    while (*dst++ = *src++);
    *dst = 0;
}
void main()
{
    char first[] = "Первая строка";
    char second[100];
    CopyString(first, second);
}
```

Указатель на байт (тип char*) указывает на начало строки. Предположим, нам нужно подсчитать количество цифр в строке, на которую показывает указатель str:

```
#include <ctype.h>
. . .
char* str = “proba1,2 3end”;
int count = 0;
while (*str != 0)       // признак конца строки – ноль
```

```
{     if (isdigit(*str++))       // функция isdigit проверяет байт, на который
                                 // указывает str, и указатель сдвигается на
                                 // следующий байт
      count++;                   // счетчик цифр
}
```

При выходе из цикла while переменная count содержит количество цифр в строке str, а сам указатель str указывает на конец строки – нулевой байт. Чтобы проверить, является ли текущий символ цифрой, используется функция isdigit. Это одна из многих стандартных функций языка, предназначенных для работы с символами и строками.

С помощью функций стандартной библиотеки языка реализованы многие часто используемые операции над символьными строками. В большинстве своем в качестве строк они воспринимают указатели. Приведем ряд наиболее употребительных. Прежде чем использовать эти указатели в программе, нужно подключить их описания с помощью операторов #include <string.h> и #include <ctype.h>.

char* strcpy(char* target, const char* source);

Копировать строку source по адресу target, включая завершающий нулевой байт. Функция предполагает, что памяти, выделенной по адресу target, достаточно для копируемой строки. В качестве результата функция возвращает адрес первой строки.

char* strcat(char* target, const char* source);

Присоединить вторую строку с конца первой, включая завершающий нулевой байт. На место завершающего нулевого байта первой строки переписывается первый символ второй строки. В результате по адресу target получается строка, образованная слиянием первой со второй. В качестве результата функция возвращает адрес первой строки.

int strcmp(const char* string1, const char* string2);

Сравнить две строки в лексикографическом порядке (по алфавиту). Если первая строка должна стоять по алфавиту раньше, чем вторая, то

результат функции меньше нуля, если позже – больше нуля, и ноль, если две строки равны.

size_t strlen(const char* string);

Определить длину строки в байтах, не считая завершающего нулевого байта.

Рассмотрим пример, использующий приведенные функции, в массиве result будет образована строка "1 января 2007 года, 00 часов":

```
. . .
char result[100];
char* date = "1 января 2007 года";
char* time = "00 часов";
strcpy(result, date);
strcat(result, ", ");
strcat(result, time);
```

Как видно из этого примера, литералы можно непосредственно использовать в выражениях.

Определить массив строк можно с помощью следующего объявления:

```
char* StrArray[5] = { "one", "two", "three", "four", "five" };
```

Тема 9

Распределение памяти

Проблемы при явном распределении памяти в Си++, способы их решения. Ссылки и указатели. Распределение памяти под переменные, управление память. с помощью переопределенных операторов new и delete.

В языке Си++ существует три способа выделения памяти для используемых в программе данных: автоматический, статический и динамический.

Автоматические переменные

Самый простой метод – это объявление переменных внутри функций. Если переменная объявлена внутри функции, каждый раз, когда функция вызывается, под переменную автоматически отводится память. Когда функция завершается, память, занимаемая переменными, освобождается. Такие переменные называют автоматическими.

При создании автоматических переменных они никак не инициализируются, т.е. значение автоматической переменной сразу после ее создания не определено, и нельзя предсказать, каким будет это значение. Соответственно, перед использованием автоматических переменных необходимо либо явно инициализировать их, либо присвоить им какое-либо значение.

```
int func()
{
  double f;  // значение f не определено
  f = 1.2;    // теперь значение f определено
  bool result = true;  // явная инициализация автоматической переменной
  . . .
}
```

Аналогично автоматическим переменным, объявленным внутри функции, автоматические переменные, объявленные внутри блока (последовательности операторов, заключенных в фигурные скобки) создаются при входе в блок и уничтожаются при выходе из блока.

Замечание. *Распространенной ошибкой является использование адреса автоматической переменной после выхода из функции. Конструкция типа:*

```
int* func()
{
      int x;
      . . .
      return  &x; // дает непредсказуемый результат.
}
```

Статические переменные

Другой способ выделения памяти – статический.

Если переменная определена вне функции, память для нее отводится статически, один раз в начале выполнения программы, и переменная уничтожается только тогда, когда выполнение программы завершается. Можно статически выделить память и под переменную, определенную внутри функции или блока. Для этого нужно использовать ключевое слово static в его определении:

```
double globalMax;                  // переменная определена вне функции
void func(int x)
{
    static bool visited = false;
    if (!visited)
    {   . . .
        visited = true;        // инициализация
    }
    . . .
}
```

В данном примере переменная visited создается в начале выполнения программы. Ее начальное значение – false. При первом вызове функции func условие в операторе if будет истинным, выполнится инициализация, и переменной visited будет присвоено значение true. Поскольку статическая переменная создается только один раз, ее значения между вызовами функции сохраняются. При втором и последующих вызовах функции func инициализация производиться не будет. Если бы переменная visited не была объявлена static, то инициализация происходила бы при каждом вызове функции.

Динамическое выделение памяти

Третий способ выделения памяти в языке Си++ – динамический. Память для величины какого-либо типа можно выделить, выполнив операцию new. В качестве операнда выступает название типа, а результатом является адрес выделенной памяти.

```
long* lp;
lp = new long;          // создать новое целое число
Complex* cp;
cp = new Complex; // создать новый объект типа Complex
```

Созданный таким образом объект существует до тех пор, пока память не будет явно освобождена с помощью операции delete. В качестве операнда delete должен быть задан адрес, возвращенный операцией new:

```
delete lp;
delete cp;
```

Динамическое распределение памяти используется, прежде всего, тогда, когда заранее неизвестно, сколько объектов понадобится в программе и понадобятся ли они вообще. С помощью динамического распределения памяти можно гибко управлять временем жизни объектов, например выделить память не в самом начале программы (как для глобальных переменных), но, тем не менее, сохранять нужные данные в этой памяти до конца программы.

Если необходимо динамически создать массив, то нужно использовать немного другую форму new:

```
int *address = new int[100];
```

в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов целого типа.

В отличие от определения переменной типа массив, размер массива в операции new может быть произвольным, в том числе вычисляемым в ходе выполнения программы. (Напомним, что при объявлении переменной типа массив размер массива должен быть константой.)

Освобождение памяти, выделенной под динамический массив, должно быть выполнено с помощью операции delete:

```
delete [] address;
```

Выделение памяти под строки

В следующем фрагменте программы динамически выделяется память под строку переменной длины и копируем туда исходную строку. Стандартная функция strlen подсчитывает количество символов в строке:

```
int length = strlen(src_str);
// выделить память и добавить один байт для завершающего строку нуля
char* buffer = new char[length + 1];
strcpy(buffer, src_str);    // копирование строки
```

Операция new возвращает адрес выделенной памяти. Однако нет никаких гарантий, что new обязательно завершится успешно. Объем оперативной памяти ограничен, и может случиться так, что найти еще один участок свободной памяти будет невозможно. В таком случае new возвращает нулевой указатель (адрес 0). Поэтому результат new необходимо проверять:

```
char* newstr;
newstr = new char[length];
if (newstr == NULL)        // проверить результат
{         // обработка ошибок
}
// память выделена успешно
```

Рекомендации по использованию указателей и динамического распределения памяти

Указатели и динамическое распределение памяти – очень мощные средства языка. С их помощью можно разрабатывать гибкие и весьма

эффективные программы. В частности, одна из областей применения Си++ – системное программирование – практически не могла бы существовать без возможности работы с указателями. Однако возможности, которые получает программист при работе с указателями, накладывают на него и большую ответственность. Наибольшее количество ошибок в программу вносится именно при работе с указателями. Как правило, эти ошибки являются наиболее трудными для обнаружения и исправления.

Приведем несколько примеров.

- Использование неверного адреса в операции delete. Результат такой операции непредсказуем. Вполне возможно, что сама операция пройдет успешно, однако внутренняя структура памяти будет испорчена, что приведет либо к ошибке в следующей операции new, либо к порче какой-нибудь информации.
- Пропущенное освобождение памяти, т.е. программа многократно выделяет память под данные, но "забывает" ее освобождать. Такие ошибки называют утечками памяти. Во-первых, программа использует ненужную ей память, тем самым, понижая производительность. Во-вторых, вполне возможно, что в 99 случаях из 100 программа будет успешно выполнена. Однако если потеря памяти окажется слишком большой, программе может не хватит памяти под какие-нибудь данные и, соответственно, произойдет сбой.
- Запись по неверному адресу. Скорее всего, будут испорчены какие-нибудь данные. Как проявится такая ошибка – неверным результатом, сбоем программы или иным образом – предсказать трудно.

Примеры ошибок можно приводить бесконечно. Общие их черты, обуславливающие сложность обнаружения, это, во-первых, непредсказуемость результата и, во-вторых, проявление не в момент совершения ошибки, а позже, быть может, в том месте программы, которое само по себе не содержит ошибки.

Поэтому подчеркнем, что их использование требует внимания и дисциплины. Несколько общих рекомендаций.

1. Используйте указатели и динамическое распределение памяти только там, где это действительно необходимо. Проверьте, можно ли выделить память статически или использовать автоматическую переменную.
2. Старайтесь локализовать распределение памяти. Если какой-либо метод выделяет память (в особенности под временные данные), он же и должен ее освободить.
3. Там, где это возможно, вместо указателей используйте ссылки.
4. Проверяйте программы с помощью специальных средств контроля памяти (Purify компании Rational, Bounce Checker компании Nu-Mega и т.д.)

Ссылки.

Ссылка – это еще одно имя переменной. Если имеется какая-либо переменная, например

```
Complex x;
```

то можно определить ссылку на переменную *x* как

```
Complex &y = x;
```

и тогда *x* и *y* обозначают одну и ту же величину. Если выполнены операторы

```
x.real = 1;
x.imaginary = 2;
```

то y.real равно 1 и y.imaginary равно 2. Фактически, ссылка – это адрес переменной (поэтому при определении ссылки используется символ & – знак операции взятия адреса), и в этом смысле она сходна с указателем, однако у ссылок есть свои особенности.

Во-первых, определяя переменную типа ссылки, ее необходимо инициализировать, указав, на какую переменную она ссылается. Нельзя определить ссылку

```
int &xref;
```

можно только

```
int &xref = x;
```

Во-вторых, нельзя переопределить ссылку, т.е. изменить на какой объект она ссылается. Если после определения ссылки xref мы выполним присваивание

```
xref = y;
```

то выполнится присваивание значения переменной y той переменной, на которую ссылается xref. Ссылка xref по-прежнему будет ссылаться на *x*. В результате выполнения следующего фрагмента программы:

```
int x = 10;
int y = 20;
int &xref = x;
xref = y;
x += 2;
cout << "x = " << x << endl;
cout << "y = " << y << endl;
cout << "xref = " << xref << endl;
```

будет выведено:

```
x = 22
y = 20
xref = 22
```

В-третьих, синтаксически обращение к ссылке аналогично обращению к переменной. Если для обращения к атрибуту объекта, на который ссылается указатель, применяется операция ->, то для подобной же операции со ссылкой применяется точка ".".

```
Complex a;
Complex* aptr = &a;
Complex& aref = a;
aptr -> real = 1;
aref.imaginary = 2;
```

Как и указатель, ссылка сама по себе не имеет значения. Ссылка должна на что-то ссылаться, тогда как указатель должен на что-то указывать.

II семестр

Тема 1

Парадигмы языков программирования

1. Императивные (процедурные) языки.

2.Языки функционального программирования.

3. Декларативные языки.

4. Объектно-ориентированные языки.

Тема 2

Критерии оценки языков программирования

5. Понятность, надежность, гибкость, простота, естественность, мобильность, стоимость.

Тема 3

Объекты данных в языках программирования

6. Имена.

7. Константы.

8. Переменные..

Тема 4

Механизмы типизации

9. Статические и динамические типы данных.

10. Слабая, строгая типизация.

11. Производные типы. Эквивалентность типов.

12. Наследование атрибутов. Ограничения. Подтипы.

Тема 5

Время жизни переменных. Область видимости переменных

13. Время жизни переменных. Область видимости переменных.

Тема 6

Типы данных

14. Элементарные типы данных.

15. Символьные строки. Перечисляемые типы. Ограниченные типы.

16. Векторы и массивы Записи. Объединения. Множества. Списки.

Тема 7

Выражения и операторы присваивания

17. Арифметические и логические выражения.

18. Операторы присваивания.

Тема 8

Структуры управления на уровне операторов

19. Составные и условные операторы. Операторы цикла.

Тема 9

Подпрограммы

20. Определение подпрограммы. Процедуры и функции. Методы передачи параметров. Сопрограммы.

Тема 10

Описание языка программирования

21. Определение синтаксиса языка. Описание контекстных условий. Описание динамической семантики.

Тема 11

Теоретические основы трансляции

22. Формальные языки и грамматики.

23. Автоматные грамматики и языки.

24. Контекстно-свободные грамматики и языки.

25. Трансляция выражений.

5. Учебно-методическое обеспечение

5.1 **Основная литература**

1. Кнут Д. Искусство программирования. Т.1 Основные алгоритмы. – М.: Издат. дом “Вильямс”, 2000.
2. Пратт Т., Зельковиц М. Языки программирования: реализация и разработка. – СПб.:Питер, 2001.
3. Себеста Р.У. Основные концепции языков программирования. – М.: Издат. дом “Вильямс”, 2001.
4. Свердлов С.З. Языки программирования и методы трансляции. – СПб.:Питер, 2007.
5. Павловская Т.А. C/C++ программирование на языке высокого уровня. – СПб.: Питер, 2002. – 464 с.
6. Павловская Т.А., Щупак Ю.А. C/C++. Структурное программирование: практикум. – СПб.: Питер, 2003. – 240 с.
7. Хопкрофт Дж., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений. – М.: Издат. дом “Вильямс”, 2002.

5.2 Дополнительная литература

1. Ахо А., Ульман Д., Сети Р. Компиляторы: принципы, технологии, инструментарий. - М.: Вильямс, 2001.
2. Дейтель Х.М., Дейтель П.Дж. Как программировать на C++. – М.: БИНОМ, 1999.–1000с.
3. Зыков В. В. Введение в системный анализ: моделирование, управление, информация. – Тюмень: ТюмГУ, 1998.
4. Кауфман В.Ш. Языки программирования. Концепции и принципы. - М.: Радио и связь, 1998.
5. Костельцов А.В. Построение интерпретаторов и компиляторов: использование программ BIZON, BYACC, ZUBR. - М.: Наука и техника, 2001.

6. Соммервил И. Инженерия программного обеспечения. - М.: Вильямс, 2002.

6. Перечень и темы промежуточных форм контроля знаний

6.1 Тестовые задания.

Образцы тестовых заданий

№ 1

1. Если n =3, каков будет результат:
switch(n) { case 3 : cout <<"aaa\n";
break; case 3 : cout <<"bbb\n"; break;
default: cout <<"ввв\n"; break;}?

☐ ошибка компилятора
☐ ааа
☐ ббб
☐ ввв
☐ неопределенное поведение

2. Если i = 5, каков будет результат:
do { cout << (--i)--<< "";} while (i >= 2 && i < 5);?

☐ ошибка компиляции
☐ цикл не разу не будет выполнен
☐цикл будет выполнен бесконечно
☐ 4321
☐ 432
☐ 421
☐ 42

3. Что означает запись for (;;)?

☐ бесконечный цикл
☐ цикл, который не выполняется ни разу
☐ ошибка компиляции
☐ аварийный выход из программы

№ 2

1. Если char n = 3, каков будет результат?
switch(n) { case 3 : cout <<"aaa";

case 3 : cout <<"bbb"; default: cout <<"ввв"; break;}?

☐ ошибка компилятора
☐ ааа
☐ ббб
☐ ввв
☐ аааббббвв
☐ аааввв
☐ неопределенное поведение

2. Если i = 5, каков будет результат?
do { cout << (++ i)++<< " ";} while (i < 8 && i >= 5); ?

☐ 6
☐ 6 8
☐ 6 7
☐ 6 7 8

3. Что означает запись while (false)?

☐ бесконечный цикл
☐ цикл, который не выполняется ни разу
☐ ошибка компиляции
☐ аварийный выход из программы

№ 3

1. Если n = 3, каков будет результат?
if (n= = 3) cout <<"aaa"; else if (n = = 3) cout <<"bbb";
else if (n ! = 3) cout <<"ccc";?

☐ aaa
☐ bbb
☐ ccc
☐ aaaccc
☐ bbbccc
☐ ошибка компиляции

2. Если i = 5, каков будет результат:
while (i <= 5){ cout << (--i)--<< ""; if (i<2) break;}?

☐ ошибка компиляции
☐ цикл не разу не будет выполнен

□ цикл будет выполнен бесконечно
□ 4321
□ 432
□ 421
□ 42

3. Что произойдет при выполнении
for (i = 0;;i <5) { continue; i --; func( ); }?

□ функция func выполниться 5 раз
□ функция func не выполниться ни разу
□ цикл будет выполнен бесконечно

7. Вопросы для подготовки к зачету и к экзамену

7.1. К зачету (I семестр).

1. История и назначение языка C++.
2. Компиляция и выполнение программы в среде Windows.
3. Имена. Правила именования переменных и функций языка.
4. Правила записи констант.
5. Формирование и вычисление выражений.
6. Арифметические операции и операции сравнения.
7. Логические и битовые операции.
8. Условная операция, последовательность, операции присваивания.
9. Порядок вычисления выражений.
10.Оператор. Операторы-выражения. Объявление имен.
11.Операторы управления: условные.
12.Операторы управления: цикла.
13.Оператор возврата. Оператор перехода.
14.Вызов функций.
15.Имена функций.
16.Необязательные аргументы функций.
17.Рекурсия.
18.Встроенные типы данных: целые числа.
19.Встроенные типы данных: вещественные числа.
20.Встроенные типы данных: логические величины, символы и байты.

21.Наборы перечисляемых значений.
22.Понятие класса.
23.Определение методов класса.
24.Переопределение классов.
25.Подписи методов и необязательные аргументы.
26.Массивы.
27.Структуры.
28.Объединения.
29.Указатели.
30.Адресная арифметика.
31.Связь между массивами и указателями.
32.Бестиповый указатель. Нулевой указатель
33.Строки и литералы
34.Автоматические переменные.
35.Статические переменные.
36.Динамическое выделение памяти .
37.Выделение памяти под строки.
38.Ссылки.
39.Распределение памяти при передаче аргументов функции.

7.2. К экзамену (II семестр).

1. Императивные языки.
2. Языки функционального программирования.
3. Декларативные языки.
4. Три основные свойства объектно-ориентированных языков программирования.
5. Как можно увеличить надежность языка программирования?
6. Что понимается под естественностью языка программирования.
7. Какие два критерия разработки языка программирования конфликтуют между собой?
8. Из каких составляющих складывается суммарная стоимость языка программирования?
9. Какое свойство языка программирования дает возможность более просто переносить программы с одной платформы на другие?
10. Концептуальная целостность языка программирования.
11. Что понимается под объектом данных в языках программирования?
12. Отличие литерала от именованной константы.

13.С помощью каких атрибутов можно охарактеризовать переменную?

14.Предопределенное имя. Неявное определение типа.

15.Статическое связывание типа. Недостатки слабой типизации языка программирования.

16.Основные признаки строгой типизации.

17.Из каких частей состоит среда ссылок? Какие операции определены для указателей?

18.Представление в памяти многомерных массивов.

19.Какие типы побочных эффектов встречаются в выражениях.

20.Что означает приведение типа? Перегруженный оператор.

21.Как выполняется сокращенное вычисление?

22.Составной оператор присваивания.

23.Типы управляющих структур в языках программирования.

24.Что представляет собой спецификация подпрограммы?

25.Какие существуют режимы и механизмы передачи параметров?

26.Чем сопрограммы отличаются от обычных подпрограмм?

27.Основные понятия формального языка.

28.Примеры языков.

29.Порождающие грамматики (Н.Хомского).

30.Дерево вывода. Задача разбора.

31.Интерпретация задачи разбора (домино Де Ремера).

32.Эквивалентность и однозначность грамматик.

33.Граф автоматной грамматики.

34.Конечные автоматы.

35.Контекстно-свободные (КС) грамматики и языки. Однозначность КС-грамматики.

36.Алгоритмы распознавания КС-языков.

37.Распознающий автомат для КС-языков. Самовложение в КС-граматиках.

38.Синтаксические диаграммы КС-языков.

39.Синтаксический анализ КС-языков методом рекурсивного спуска.

40.Трансляция выражений. Польская запись.

41.Алгоритм вычисления выражений в обратной польской записи.

42.Перевод выражений в обратную польскую запись.

43.Интерпретация выражений.

44.Семантическое дерево выражения.

УЧЕБНО-МЕТОДИЧЕКАЯ (ТЕХНОЛОГИЧЕСКАЯ) КАРТА ДИСЦИПЛИНЫ

Номер недели	Номер темы		Занятия (номера)		лекция, доп. литература	Самостоятельная работа студентов		
			Практич.	Лабораторные		Содержание	часы	
I семестр								
1	]	Начальные сведения о языке.	1		лекция, доп. литература.	подготовка к тесту № 1	2	тест № 1
3	2	Имена, переменные, константы.	2		тоже	подготовка к тесту № 2	2	тест №2
5	3	Операции и выражения	3		тоже	подготовка к тесту №3	2	тест №3
7	4	Операторы	4		тоже	подготовка к тесту №4	2	тест № 4
9	5	Функции.	5		тоже	подготовка к тесту №5	2	тест № 5
11	6	Встроенные типы данных	6		тоже	подготовка к тесту № 6	2	тест № 6
13	7	Классы и объекты	7		тоже	подготовка к тесту № 7	4	тест № 7
15	8	Производные типы данных	8		тоже	подготовка к тесту № 8	2	тест № 8
17	9	Распределение памяти	9		тоже	подготовка к тесту № 9	2	тест № 9
II семестр								
1	1,2	Парадигмы языков программирования	1		лекция, доп. литература.	подготовка к тесту № 1	2	тест № 1
3	3,4	Объекты данных в языках	2		тоже	подготовка к тесту № 2	3	тест № 2
5	5	Время жизни переменных.	3		тоже	подготовка к тесту № 3	2	тест № 3

7	6	Типы данных	4		тоже	подготовка к тесту № 4	4	тест № 4
9	7,8	Структуры управле-ния на уровне операто-ров	5		тоже	подготовка к тесту № 5	2	тест № 5
11	9,10	Подпрог-раммы	6			подготовка к тесту № 6	4	тест № 6
13	11	Формаль¬ные языки и грамма тики.	7		тоже	подготовка к тесту № 7	2	тест № 7
15	11	Автомат ные грам матики и языки	8		тоже	подготовка к тесту № 8	4	тест № 8
17	11	Контекст но-свобод ные грам матики и языки	9		тоже	подготовка к тесту № 9	2	тест № 9