Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра компьютерных технологий

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С++

Конспект лекций
для студентов специальности
5В070400 – Вычислительная техника и программное обеспечение

Алматы 2016

СОСТАВИТЕЛЬ: Е.С.Турганбаев. Объектно-ориентированное программирование С++. Конспект лекции для студентов очной формы обучения специальности 5В070400 – Вычислительная техника и программное обеспечение. - Алматы: АУЭС, 2007 – 38 с.

Объектно-ориентированное программирование (ООП) - это результат естественной эволюции более ранних методологий программирования. Потребность в ООП связана со стремительным усложнением приложений и отсюда как следствие недостаточной надежностью программ и выразительными способностями языков программирования.

В данном конспекте представлен лекционный материал, позволяющий получить начальные навыки использования объектно-ориентированного подхода на основе языка С++. Лекционный материал прошел апробацию в учебном процессе студентов специальности «Вычислительная техника и программное обеспечение», состоит из восьми лекций, общей продолжительностью 32 часа.

Конспект лекции для студентов очной формы обучения специальности 5В070400 – Вычислительная техника и программное обеспечение.

Библиогр. – 5 назв.

Рецензент: Ю.М.Гармашова

Печатается по плану издания некоммерческого общества «Алматинский институт энергетики и связи» на 2014 г.

Введение

Объектно-ориентированное программирование (ООП) - это результат естественной эволюции более ранних методологий программирования. Потребность в ООП связана со стремительным усложнением приложений и отсюда как следствие недостаточной надежностью программ и выразительными способностями языков программирования.

ООП - это моделирование объектов посредством иерархически связанных классов. Малозначащие детали объекта скрыты от нас, и если мы даем команду, например, переместить объект, то он «знает», как он это делает. Переход от традиционного программирования к ООП на начальном этапе характерен тем, что под объектами в программе подразумеваются конкретные физические объекты. В этом случае легче дается понимание различных действий над ними. В качестве примера можно выбрать простые графические фигуры, поскольку каждая фигура представляет реальный объект на экране. Его всегда можно отобразить и тем самым проверить моделируемые действия над объектом в явном виде. После того как определены простейшие графические объекты, достаточно легко можно формировать более сложные на основе уже имеющихся. В ООП такому сложному графическому объекту соответствует список примитивных объектов, к которому применимы все те же действия, что и к составляющим его элементам: отображение, стирание с экрана, перемещение в заданном направлении.

ООП опирается на три основные понятия: инкапсуляция, наследование, полиморфизм.

Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником или производным классом.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

В данном кратком конспекте представлены лекционный материал, позволяющий получить начальные навыки использования объектно-ориентированного подхода на основе языка C++. Лекционный материал прошел апробацию в учебном процессе студентов специальности «Вычислительная техника и программное обеспечение».

1 Лекция №1. Организация консольного ввода/вывода в С++

Цель: ознакомиться с основными принципами разработки программ на языке С/С++.

1.1 Организация программы в языке С/С++

Программа на языке Си имеет следующую структуру:

```
#директивы препроцессора
. . . . . .
#директивы препроцессора
функция F1 ( ) – заголовок функции
{ оператор1; оператор 2; …};    //- тело функции
функция F2 ( )  //– заголовок функции
{ оператор1; оператор 2; …};    // - тело функции
void main ( )     // - заголовок головной программы
{ оператор1; оператор 2; …} - //тело головной программы
```

Директивы препроцессора управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на СИ в виде текстового файла, проходит 3 этапа обработки:

1) препроцессорное преобразование текста;
2) компиляция;
3) компоновка (редактирование связей или сборка).

После этих трех этапов формируется исполняемый код программы. Задача препроцессора - преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора.

Директива начинается с #. Например,

1) #define – указывает правила замены в тексте.

#define ZERO 0.0 – означает, что каждое использование в программе имени ZERO будет заменяться на 0.0.

2) #include< имя заголовочного файла> предназначена для включения в текст программы текста из каталога «Заголовочных файлов», поставляемых вместе со стандартными библиотеками. Каждая библиотечная функция Си имеет соответствующее описание в одном из заголовочных файлов. Список заголовочных файлов определен стандартом языка. Употребление директивы include не подключает соответствующую стандартную библиотеку, а только позволяет вставить в текст программы описания из указанного заголовочного файла. Подключение кодов библиотеки осуществляется на этапе компоновки, т. е. после компиляции. Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.

После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.

Программа представляет собой набор описаний и определений и состоит из набора функций. Среди этих функций всегда должна быть функция с именем main. Без нее программа не может быть выполнена. Перед именем функции помещаются сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип void: void main ( ). Каждая функция, в том числе и main, должна иметь набор параметров, он может быть пустым, тогда в скобках указывается (void).

За заголовком функции размещается тело функции. Тело функции - это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.

Определения вводят объекты (объект - это именованная область памяти, частный случай объекта - переменная), необходимые для представления в программе обрабатываемых данных. Примером являются

```
int y = 10 ; //именованная константа
float x ;   //переменная
```

Описания уведомляют компилятор о свойствах и именах объектов и функций, описанных в других частях программы.

Операторы определяют действия программы на каждом шаге ее исполнения.

Пример программы на Си:

```
#include <stdio.h>  //препроцессорная директива
void main()         //функция
{                   //начало
printf("Hello! ");  //печать
}                   //конец
```

1.2 Типы данных

Данные отображают в программе окружающий мир. Цель программы состоит в обработке данных. Данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

1) внутреннее представление данных в памяти компьютера;
2) множество значений, которые могут принимать величины этого типа;
3) операции и функции, которые можно применять к данным этого типа.

В зависимости от требований задания программист выбирает тип для объектов программы. Типы Си++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением. В Си++ определено 6 простых типов данных:

int (целый)
char (символьный)
wchar_t (расширенный символьный)
bool (логический)
float(вещественный)
double (вещественный с двойной точностью)

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов

short (короткий)
long (длинный)
signed (знаковый)
unsigned (беззнаковый)

Переменная в СИ++ - именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Перед использованием любая переменная должна быть описана. Примеры:

int a; float x;

Общий вид оператора описания:

[класс памяти][const]тип имя [инициализатор];

Класс памяти может принимать значения: auto, extern, static, register. Класс памяти определяет время жизни и область видимости переменной. Если класс памяти не указан явно, то компилятор определяет его, исходя из контекста объявления. Время жизни может быть постоянным – в течение выполнения программы или временным – в течение блока. Область видимости – часть текста программы, из которой допустим обычный доступ к переменной. Обычно область видимости совпадает с областью действия. Кроме того случая, когда во внутреннем блоке существует переменная с таким же именем.

Const – показывает, что эту переменную нельзя изменять (именованная константа).

При описании можно присвоить переменной начальное значение (инициализация).

Классы памяти:

1) auto – автоматическая локальная переменная. Спецификатор auto может быть задан только при определении объектов блока, например, в теле функции. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют;

2) extern – глобальная переменная, она находится в другом месте программы (в другом файле или долее по тексту). Используется для создания переменных, которые доступны во всех файлах программы;

3) static – статическая переменная, она существует только в пределах того файла, где определена переменная;

4) register – аналогичны auto, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как auto.

Пример

```
int a; //глобальная переменная
void main(){
int b;//локальная переменная
extern int x;//переменная x определена в другом месте
static int c;//локальная статическая переменная
a=1;//присваивание глобальной переменной
int a;//локальная переменная a
a=2;//присваивание локальной переменной
::a=3;//присваивание глобальной переменной
}
int x=4;//определение и инициализация x
```

В примере переменная a определена вне всех блоков. Областью действия переменной a является вся программа, кроме тех строк, где используется локальная переменная a. Переменные b и c – локальные, область их видимости – блок. Время жизни различно: память под b выделяется при входе в блок (т. к. по умолчанию класс памяти auto), освобождается при выходе из него. Переменная c (static) существует, пока работает программа.

Если при определении начальное значение переменным не задается явным образом, то компилятор обнуляет глобальные и статические переменные. Автоматические переменные не инициализируются..

1.3 Ввод, вывод C++

Частью стандартной библиотеки C++ является библиотека iostream, которая реализована как иерархия классов и обеспечивает базовые возможности ввода/вывода. Ввод с терминала, называемый стандартным вводом, “привязан” к предопределенному объекту cin. Вывод на терминал, или стандартный вывод, привязан к объекту cout. Третий предопределенный объект, cerr, представляет собой стандартный вывод для ошибок. Обычно он используется для вывода сообщений об ошибках и предупреждений. Для использования библиотеки ввода/вывода необходимо включить соответствующий заголовочный файл:

```
#include <iostream>
```

Чтобы значение поступило в стандартный вывод или в стандартный вывод для ошибок, используется оператор <<:

```
int v1, v2;
// ...
cout << "сумма v1 и v2 = ";
cout << v1 + v2;
cout << "\n";
```

Последовательность "\n" представляет собой символ перехода на новую строку. Вместо "\n" мы можем использовать предопределенный *манипулятор* endl.

```
cout << endl;
```

2 Лекция №2. Классы. Инкапсуляция

Цель: **о**знакомиться с основными принципами объектно-ориентированного программирования. Разобраться с решением проблемы зашиты данных, а также с интеграцией данных и способов их обработки

2.1 Парадигма объектно-ориентированного программирования

Объе́ктно-ориенти́рованное или объектное программирование (в дальнейшем ООП) – парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс – это тип, описывающий устройство объектов. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. Говорят, что объект – это экземпляр класса. Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

Класс является описываемой на языке терминологии (пространства имён) исходного кода моделью ещё не существующей сущности, т. н. объекта.

Объект – сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции (и линковки) исходного кода на выполнение).

Основные принципы ООП:

1) Инкапсуляция.

Инкапсуляция – это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик – пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

Сокрытие данных.

Сокрытие данных – неотделимая часть ООП, управляющая областями видимости. Является логическим продолжением инкапсуляции. Целью

сокрытия является невозможность для пользователя узнать или испортить внутреннее состояние объекта.

2) Наследование.

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом), добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность.

3) Полиморфизм.

Полиморфизмом называют явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением сигнатуры метода. Это обеспечивает сохранение неизменным интерфейса класса-предка и позволяет осуществить связывание имени метода в коде с разными классами – из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем. Такой механизм называется динамическим (или поздним) связыванием – в отличие от статического (раннего) связывания, осуществляемого на этапе компиляции.

2.2 Классы

Класс – фундаментальное понятие C++, он лежит в основе многих свойств C++. Класс предоставляет механизм для создания объектов. В классе отражены важнейшие концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

С точки зрения синтаксиса, класс в C++ – это структурированный тип, образованный на основе уже существующих типов.

В этом смысле класс является расширением понятия структуры. В простейшем случае класс можно определить с помощью конструкции:

тип_класса имя_класса{список_членов_класса};

где

тип_класса – одно из служебных слов class, struct, union;

имя_класса – идентификатор;

список_членов_класса – определения и описания типизированных данных и принадлежащих классу функций.

Функции – это методы класса, определяющие операции над объектом.

Данные – это поля объекта, образующие его структуру. Значения полей определяет состояние объекта.

Примеры.

```
struct date                    // дата
{int month,day,year;           // поля: месяц, день, год
```

```
  void set(int,int,int);          // метод – установить дату
  void next();                    // метод – установить следующую дату
  void print();                   // метод – вывести дату
};
struct class complex              // комплексное число
{double re,im;
  double real(){return(re);}
  double imag(){return(im);}
  void set(double x,double y){re = x; im = y;}
  void print(){cout<<“re = “<<re; cout<<“im = “<<im;}
};
```

Для описания объекта класса (экземпляра класса) используется конструкция

имя_класса имя_объекта;

```
date today,my_birthday;
date clim[30];                    // массив объектов
```

В определяемые объекты входят данные, соответствующие членам – данным класса. Функции – члены класса позволяют обрабатывать данные конкретных объектов класса. Обращаться к данным объекта и вызывать функции для объекта можно двумя способами. Первый - с помощью “квалифицированных” имен:

имя_объекта. имя_данного
имя_объекта. имя_функции

Например:

```
complex x1,x2;
x1.re = 1.24;
x1.im = 2.3;
x2.set(5.1,1.7);
x1.print();
```

Второй способ доступа использует указатель на объект (см. предыдущую лабораторную работу).

указатель_на_объект–>имя_компонента

```
complex *point = &x1;  //  или  point = new complex;
point –>re = 1.24;
point –>im = 2.3;
point –>print();
```

2.3 Доступность компонентов класса

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где «видно»

определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: public, private, protected.

Общедоступные (public) компоненты доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект->имя_члена_класса

Собственные (private) компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями – членами данного класса и функциями – «друзьями» того класса, в котором они описаны.

Защищенные (protected) компоненты доступны внутри класса и в производных классах.

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова class. В этом случае все компоненты класса по умолчанию являются собственными.

Пример.

```
class complex
{
  double re, im;                  // private по умолчанию
  public:
  double real(){return re;}
  double imag(){return im;}
  void set(double x,double y){re = x; im = y;}
};
```

2.4 Конструктор

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex) либо явным образом присваивать значения данным объекта. Однако для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую конструктором. Формат определения конструктора следующий:

имя_класса (список_форм_параметров){операторы_тела_конструктора}

Имя этой компонентной функции по правилам языка C++ должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора new каждого объекта класса.

Пример.
complex(double re1 = 0.0,double im1 = 0.0){re = re1; im = im1;}.

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

3 Лекция №3. Классы. Наследование

Цель: разобраться с проблемой повторного использования кода. Получить представление о построении диаграмм классов в UML.

3.1 Наследование

Наследование – это механизм получения нового класса на основе уже существующего. Существующий класс может быть дополнен или изменен для создания нового класса.

Существующие классы называются базовыми, а новые – производными. Производный класс наследует описание базового класса; затем он может быть изменен добавлением новых членов, изменением существующих функций-членов и изменением прав доступа. С помощью наследования может быть создана иерархия классов, которые совместно используют код и интерфейсы.

Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах.

В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов (*public, protected*).

Допускается множественное наследование – возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов. В иерархии классов соглашение относительно доступности компонентов класса следующее:

1) *private* – член класса может использоваться только функциями – членами данного класса и функциями – “друзьями” своего класса. В производном классе он недоступен.

2) *protected* – то же, что и *private,* но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями – «друзьями» классов, производных от данного.

3) *public* – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к *public* - членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление friend не является атрибутом доступа и не наследуется.

Синтаксис определения производного класса:

class имя_класса : список_базовых_классов
{список_компонентов_класса};

В производном классе унаследованные компоненты получают статус доступа private, если новый класс определен с помощью ключевого слова class, и статус public, если с помощью struct.

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа – *private, protected и public*, которые указываются непосредственно перед именами базовых классов.

Конструкторы и деструкторы производных классов.

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом происходит передача аргументов от конструктора производного класса конструктору базового класса.

Пример.

```
class Basis
{ int a,b;
public:
Basis(int x,int y){a=x;b=y;}
};
class Inherit:public Basis
{int sum;
public:
Inherit(int x,int y, int s):Basis(x,y){sum=s;}
};
```

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Таким образом, объект производного класса содержит в качестве подобъекта объект базового класса.

Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Таким образом, порядок уничтожения объекта противоположен по отношению к порядку его конструирования.

3.2 Унифицированный язык моделирования

Для создания моделей анализа и проектирования сложных объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 год, эти языки уже имеют представительную историю развития.

В настоящее время различают три поколения языков визуального моделирования, и если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее

популярных языков 2-го поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т. д. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами – претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков 3-го поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (UML), создававшийся в 1994-1997 годах (основные разработчики – Г. Буч, Дж. Рамбо, И. Джекобсон).

UML – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов сложных программных проектов. UML – это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы – это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

В UML имеются четыре разновидности предметов:

- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

Классы в UML.

Структурные предметы являются существительными в UML-моделях. Они представляют статические части модели – понятийные или физические элементы. Среди восьми разновидностей структурных предметов в рамках данной лабораторной работы нас будет интересовать класс:

Класс – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рисунке 3.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

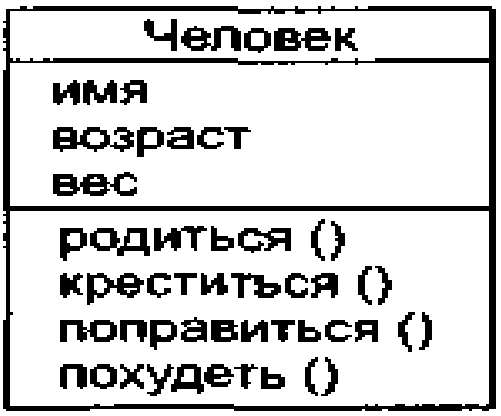


Рисунок 3.1 – Отображение класса

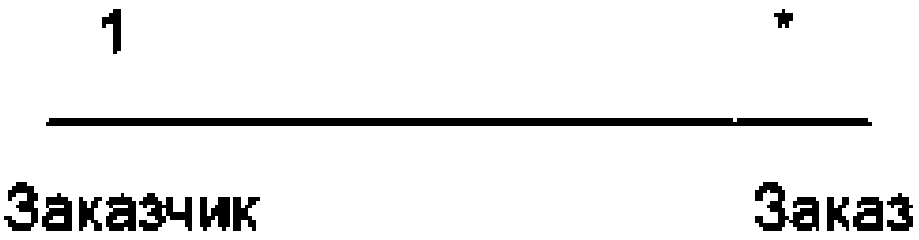
Отношения в UML.

В UML имеются четыре разновидности отношений:

– зависимость;
– ассоциация;
– обобщение;
– реализация.

Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей. Для нас в настоящий момент представляют интерес две из разновидностей отношений:

1) *Ассоциация* – структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация – это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рисунке 3.2, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.



Рисунке 3.2 – Ассоциация

2) *Обобщение* – отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рисунке 3.3, обобщение изображается в виде сплошной стрелки с полым наконечником,

указывающим на родителя.

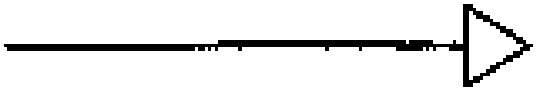


Рисунок 3.3 – Обобщение

Диаграмма – графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. *Диаграмма классов* показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

4 Лекция №4. Массивы

Цель: ознакомиться с использованием массивов.

4.1 Одномерные массивы

В языке Си/Си++, кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых. Стандарт языка определяет три способа получения производных типов:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

Массив – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти. Все элементы имеют одно имя - имя массива и отличаются индексами – порядковыми номерами в массиве. Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

Описание массива.

В С++ можно определить массив любого типа.

int mas[3] - описан массив из 3 целых чисел.

Нумерация в массивах начинается с 0-го элемента, поэтому массив mas содержит:

mas[0], mas[1], mas[2].

Массив можно инициализировать при описании. В этом случае нет необходимости указывать его размер:

int mas[]={23, 25, 81};

Далее создан массив mas из 3-х элементов:

mas[0]= 23,

mas[1]= 25,

mas[2]= 81.

char city [ ] = "Астана";

Массив city будет содержать строку из 7 элементов: 6 букв и 0-символ. 0-символ является признаком конца строки, в программе он обозначается \0.

Такой способ инициализации не подходит, если в дальнейшем в массив city потребуется занести название другого города с длинным названием.

Можно описать название города по-другому.

char city1[ ] = {'А', 'с', 'т' , 'а' , 'н' , 'а' }

Массив city1 - это массив символов, он будет содержать 6 элементов

№	City	№	City1
0	А	0	М
1	с	1	о
2	т	2	с
3	а	3	к
4	н	4	в
5	а	5	а
6	/0		

Вышеописанный способ инициализации возможен только при объявлении массива. В программе используйте поэлементную инициализацию.

city [0]='м';

Кроме стандартного доступа к элементам массива, С++ обеспечивает еще один. В С++ имя массива представляет собой не только имя, которое вы используете в своих программах, но и является адресом, по которому в памяти находится первый элемент массива. Поэтому к элементам массива можно обращаться следующими способами:

int m[6] = {4, 3 , 1, 4, 7, 8 };

m[3] или (m + 3)[0] Обращение к 4-му элементу массива.

Возможны и другие варианты:

(m + 0)[3]

(m + 2)[1]

(m - 2)[5]

Наиболее полезно использовать такой подход к массивам, содержащим символьные строки.

Пример.

char names [ ]={'Иван', 'Петр', 'Елена' } будет выглядеть в памяти следующим образом:

0 И
1 В
2 А
3 Н
4 \0
5 П
6 Е
7 Т
8 Р
9 \0
10 Е
11 Л
12 Е
13 Н
14 А
15 \0

Команда cout << names; - выведет Иван (до признака конца строки);

cout << names+5; - выведет Петр.

4.2 Многомерные массивы

Многомерные массивы - это массивы с более чем одним индексом. Чаще всего используются двумерные массивы. При описании многомерного массива необходимо указать С++, что массив имеет более чем одно измерение.

Пример.

int temp [3] [15] [10]; - резервируется место под 3-х мерный массив.

В памяти многомерные массивы представляются как одномерный массив, каждый из элементов которого, в свою очередь, представляет собой массив.

Рассмотрим на примере двумерного массива.

int a[3][2]={4, l, 5,7,2, 9};

Представляется в памяти:

a[0][0] - заносится значение 4
a[0][1] - заносится значение 1
a[1][0] - заносится значение 5
a[1][1] - заносится значение 7
a[2][0] - заносится значение 2

a[2][1] - заносится значение 9

Второй способ инициализации при описании массива

int a[3][2]={ {4,1}, {5, 7}, {2, 9} };

Обращение к элементу массива производится через индексы.

cout<< a[0][0]; - выдаст значение 4.

cout << a[1][1]; - выдаст значение 7.

Программа, которая инициализирует массив и выводит его элементы на экран:

```
#include <iostream.h>
main ()
{
int a[3] [2]={
{1,2}, {3,4}, {5,6}
};
int i,j;
for (i=0; i<=2; i++)
for(j=0;j<=l;j++)
 cout <<"\n a["<< i <<"," << j <<"] ="<< a[i]|j];
return 0;
}
```

Для того чтобы убрать из программы явные значения размера и массива, можно воспользоваться директивой define.

```
#include < iostream.h>
#define I 3
#define J 2
main()
{ int a[I][J]={ {l,2}, {3,4}, {5,6} };
  int i, j;
   for ( i=0 ;  i< I;  i++)
      for( j=0; j< J;  j++)
      cout <<"\n a["<< i <<"," << j << "] ="<< a[i][j];
return 0;
}
```

При передаче массива в функцию всегда происходит передача его адреса. Таким образом в С++ все массивы передаются по адресу.

Пример.

Вводится квадратная матрица с максимальным размером 10 на 10. Ввод матрицы оформлен в виде отдельной функции vvod. Программа заменяет все отрицательные числа их модулями.

```
#include <iostream.h>

void vvod (int  u[10][10], int n)
```

```
{
 int i,j;
 for (i=0; i<n; i++)
 { cout <<"\nVvedi "<< i<<" stroky";
   for (j=0; j< n; j++)
     cin>> u[i][j];
 }
}
void main()
{
int a[10][10];
int n,i,j,min;
cout<<"\nВведите количество строк и столбцов";
cin>>n;
vvod(a, n);
for (i=0; i< n; i++)
 for (j=0; j< n; j++)
  if (a[i][j] < 0) a[i][j]=-a[i][j];
```

Принимающая функция получает не весь массив, а только адрес первого элемента массива. Несмотря на то что в блоке *main* массив называется *a*, а в функции *vvod* – u, это один и тот же массив.

5 Лекция №5. Динамическое распределение памяти. Использование указателей

Цель: ознакомиться со способами динамического распределения памяти и использованием указателей.

5.1 Использование свободной памяти

Как вы знаете, если ваша программа объявляет массив, компилятор C++ распределяет память для хранения его элементов. Однако представляется возможным, что до некоторого времени размер массива может быть не так велик, чтобы вместить все необходимые данные. Например, предположим, что вы создали массив для хранения 100 акций. Если позже вам потребуется хранить более 100 акций, вы должны изменить свою программу и перекомпилировать ее. С другой стороны, вместо распределения массива фиксированного размера ваши программы могут запрашивать необходимое количество памяти динамически, т.е. во время выполнения. Например, если программе необходимо следить за акциями, она могла бы запросить память, достаточную для хранения 100 акций. Аналогично, если программе необходимы только 25 акций, она могла бы запросить меньше памяти. Распределяя подобным образом память динамически, ваши программы

непрерывно изменяют свои потребности без дополнительного программирования. Если ваши программы запрашивают память во время выполнения, они указывают требуемое количество памяти, а C++ возвращает указатель на эту память. C++ распределяет память из областей памяти, которые называются свободной памятью. В этом разделе рассматриваются действия, которые должна выполнить ваша программа для динамического распределения, а впоследствии освобождения памяти во время выполнения.

Чтобы запросить память во время выполнения, ваши программы должны использовать оператор C++ new.

При использовании оператора new программы указывают количество требуемой памяти. Если оператор new может успешно выделить требуемый объем памяти, он возвращает указатель на начало области выделенной памяти.

Если оператор new не может удовлетворить запрос на память вашей программы (возможно, свободной памяти уже не осталось), он возвращает указатель NULL.

Чтобы позже освободить память, распределенную с помощью оператора new, ваши программы должны использовать оператор C++ delete.

Динамическое распределение памяти во время выполнения является чрезвычайно полезной возможностью. Экспериментируйте с программами, представленными в данном уроке. И вы поймете, что динамическое распределение памяти реально выполняется очень просто.

Оператор C++ new позволяет вашим программам распределять память во время выполнения. Для использования оператора new вам необходимо указать количество байтов памяти, которое требуется программе. Предположим, например, что вашей программе необходим 50-байтный массив. Используя оператор new, вы можете заказать эту память, как показано ниже:

```
char *buffer = new char[50];
```

Говоря кратко, если оператор new успешно выделяет память, он возвращает указатель на начало области этой памяти. В данном случае, поскольку программа распределяет память для хранения массива символов, она присваивает возвращаемый указатель переменной, определенной как указатель на тип char. Если оператор new не может выделить запрашиваемый вами объем памяти, он возвратит NULL-указатель, который содержит значение 0. Каждый раз, когда ваши программы динамически распределяют память с использованием оператора new, они должны проверять возвращаемое оператором new значение, чтобы определить, не равно ли оно NULL.

Зачем необходимо динамически распределять память с использованием new?

Многие программы интенсивно используют массивы для хранения множества значений определенного типа. При проектировании своих программ программисты обычно пытаются объявить массивы с размерами,

достаточными для удовлетворения будущих потребностей программы. К сожалению, если случится так, что потребности программы когда-нибудь превысят подобные ожидания программиста, то кому-то придется редактировать и перекомпилировать такую программу.

Вместо редактирования и перекомпилирования программ, которые просто запрашивают память с запасом, вам следует создавать свои программы таким образом, чтобы они распределяли требуемую им память динамически во время выполнения, используя оператор new. В этом случае ваши программы могут адаптировать использование памяти в соответствии с вашими изменившимися потребностями, избавляя вас от необходимости редактировать и перекомпилировать программу.

Например, следующая программа USE_NEW.CPP использует оператор new для получения указателя на 100-байтный массив:

```
#include <iostream.h>
void main(void)

{
  char *pointer = new char[100];
  if (pointer != NULL) cout << "Память успешно выделена" << endl;
  else cout << "Ошибка выделения памяти" << endl;
}
```

Как видите, программа сразу проверяет значение, присвоенное оператором new переменной-указателю. Если указатель содержит значение NULL, значит new не смог выделить запрашиваемый объем памяти. Если же указатель содержит не NULL, следовательно, new успешно выделил память и указатель содержит адрес начала блока памяти.

Если new не может удовлетворить запрос на память, он возвратит NULL

При использовании оператора new для выделения памяти может случиться так, что ваш запрос не может быть удовлетворен, поскольку нет достаточного объема свободной памяти. Если оператор new не способен выделить требуемую память, он присваивает указателю значение NULL. Проверяя значение указателя, как показано в предыдущей программе, вы можете определить, был ли удовлетворен запрос на память. Например, следующий оператор использует new для распределения памяти под массив из 500 значений с плавающей точкой:

```
float *array = new float[100];
```

Чтобы определить, выделил ли оператор new память, ваша программа должна сравнить значение указателя с NULL, как показано ниже:

```
if (array != NULL) cout << "Память выделена успешно" << endl;
else cout << "new не может выделить память" << endl;
```

Следующая программа NOMEMORY.CPP выделяет каждый раз память для 10000 символов до тех пор, пока оператор new не сможет больше

выделить память из свободной памяти. Другими словами, эта программа удерживает выделенную память, пока не использует всю доступную свободную память. Если программа успешно выделяет память, она извещает об этом сообщением. Если память больше не может быть выделена, программа выводит сообщение об ошибке и завершается:

```
#include <iostream.h>
void main(void)

{
  char * pointer;
  do
  {
    pointer = new char[10000];
    if (pointer != NULL) cout << "Выделено 10000 байт" << endl;
    else cout << "Больше нет памяти" << endl;
  } while (pointer 1= NULL);
}
```

5.2 Освобождение памяти

Каждый раз при запуске вашей программы компилятор C++ устанавливает отдельную область неиспользуемой памяти, которая называется свободной памятью. Используя оператор new, ваша программа может выделить память из этой свободной памяти во время выполнения. Используя свободную память для распределения требуемой памяти, ваши программы не стеснены фиксированными размерами массивов. Размер свободной памяти может изменяться в зависимости от вашей операционной системы и модели памяти компилятора. Если увеличивается количество динамической памяти, требуемой вашими программами, вам необходимо убедиться, что вас не сдерживают ограничения свободной памяти вашей системы.

Как вы знаете, оператор C++ new позволяет вашим программам выделять память динамически во время выполнения. Если вашей программе больше не нужна выделенная память, она должна ее освободить, используя оператор delete. Для освобождения памяти с использованием оператора delete вы просто указываете этому оператору указатель на данную область памяти, как показано ниже:

```
delete pointer;
```

Следующая программа ALLOCARR.CPP выделяет память для хранения массива из 1000 целочисленных значений. Затем она заносит в массив значения от 1 до 1000, выводя их на экран. Потом программа освобождает эту

память и распределяет память для массива из 2000 значений с плавающей точкой, занося в массив значения от 1.0 до 2000.0:

```
#include <iostreain.h>
void main(void)
{
  int *int_array = new int[1000];
  float *float_array;
  int i;
  if (int_array 1= NULL)
  {
    for (i = 0; i < 1000; i++) int_array[i] = i + 1;
    for (i = 0; i < 1000; i++) cout << int_array[i] << ' ';
    delete int_array;
  }
  float_array = new float[2000];
  if (float_array != NULL)
  {
    for (i = 0; i < 2000; i++) float_array[i] = (i + 1) • 1.0;
    for (i = 0; i < 2000; i++) cout << float_array[i] << ' ' ;
    delete float_array;
  }
}
```

Как правило, ваши программы должны освобождать память с помощью оператора delete по мере того, как память становится программам не нужна.

6 Лекция №6. Классы. Полиморфизм. Виртуальные функции. Перегрузка операторов

Цель: ознакомиться с принципом полиморфизма, виртуальными функциями и перегрузкой операторов.

6.1 Полиморфизм

Полиморфизм (polymorphism) (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например, для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трёх различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и

возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В C++ можно использовать одно имя функции для множества различных действий. Это называется перегрузкой функций (function overloading).

В более общем смысле, концепцией полиморфизма является идея «один интерфейс, множество методов». Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в Си, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется перегрузкой операторов (operator overloading).

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путём создания общего для них стандартного интерфейса для реализации похожих действий.

6.2 Перегрузка операторов

С перегрузкой функций тесно связан механизм перегрузки операторов. В языке C++ можно перегрузить большинство операторов, настроив их на конкретные классы.

Перегрузка операторов – одна из самых эффективных возможностей языка C++. Она позволяет полностью интегрировать новые классы в существующую программную среду. После перегрузки операции над объектами новых классов выглядят точно так же , как и операции над переменными встроенных типов. Кроме того, перегрузка операторов лежит в основе ввода-вывода в языке C++.

В каждом из указанных случаев операторная функция объявляется по-разному, поэтому эти случаи должны быть рассмотрены по отдельности.

Перегрузка операторов в форме внешних функций.

Чтобы перегрузить оператор в форме внешней функции, необходимо определить глобальную функцию.

```
class String {
friend String& operator+(const String&, const String&);
private:
char* s;
public:
// Конструкторы и т.д.
}
String& operator+(const String& s1, const String& s2)
{
char* s = new char[strlen(s1.s) + strlen(s2.s) + 1];
strcat(s, s1.s, s2.s);
String newStr(s);
delete s;
return newStr;
}

Main()
{String s1 = "Hello";
String s2 = "Goodbye";
String s3 = s1 + s2;}
```

Перегруженная функция выглядит так же, как и любая глобальная функция (если не считать странного имени). Именно для таких случаев и были придуманы друзья. Если бы мы не объявили функцию operator+ дружественной, то она не имела бы доступа к переменной s, и мы оказались бы перед выбором: то ли разрешить всем на свете доступ к char*, то ли перейти к менее эффективной реализации, при которой строка копируется при каждом обращении к ней. С концептуальной точки зрения operator+ является частью библиотеки String, поэтому нет ничего страшного в том, чтобы объявить эту функцию другом и вручить ей ключи к внутреннему устройству String.

Внешними функциями могут перегружаться любые операторы, кроме операторов преобразования, =, [], () и -> - все эти операторы должны перегружаться только функциями класса.

6.3 Виртуальные функции

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы, включающие такие функции, называются полиморфными и играют особую роль в ООП.

Виртуальные функции предоставляют механизм позднего (отложенного) или динамического связывания. Любая нестатическая функция

базового класса может быть сделана виртуальной, для чего используется ключевое слово virtual.

Пример.

```
class base
{
public:
  virtual void print(){cout<<“\nbase”;}
. . .
};

class dir : public base
{
public:
  void print(){cout<<“\ndir”;}
};
void main()
{
  base B,*bp = &B;
  dir D,*dp = &D;
  base *p = &D;
  bp –>print(); // base
  dp –>print(); // dir
  p –>print(); // dir
}
```

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов.

6.4 Абстрактные классы

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые, по сути, есть абстрактные методы.

Пример.

```
class Base{
public:

Base();                  // конструктор по умолчанию
Base(const Base&);  // конструктор копирования
```

```
virtual ~Base();     // виртуальный деструктор
virtual void Show()=0;     //  чистая виртуальная функция
// другие чистые виртуальные функции
protected: // защищенные члены класса
private:
// часто остается пустым, иначе будет мешать будущим разработкам
};
class Derived: virtual public Base{
public:
Derived();                    // конструктор по умолчанию
Derived(const Derived&);      // конструктор копирования
Derived(параметры);           // конструктор с параметрами
virtual ~Derived();           // виртуальный деструктор
void Show();                  // переопределенная виртуальная функция
// другие переопределенные виртуальные функции
// другие перегруженные операции
protected:
// используется вместо private, если ожидается наследование
private:
// используется для деталей реализации
};
```

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом, мы получаем полиморфные объекты.

Абстрактный метод может рассматриваться как обобщение *переопределения.* В обоих случаях поведение родительского класса изменяется для потомка. Для абстрактного метода, однако, поведение просто не определено. Любое поведение задается в производном классе.

7 Лекция №7. Потоковый ввод-вывод

Цель: ознакомиться с использованием потокового ввода – вывода.

7.1 Понятие потока

Потоковые классы представляют объектно-ориентированный вариант функций ANSI-C. Поток данных между источником и приемником при этом обладает следующими свойствами:

1) Источник или приемник данных определяется объектом потокового класса.
2) Потоки используются для ввода-вывода высокого уровня.

3) Общепринятые стандартные С-функции ввода/вывода разработаны как функции потоковых классов, чтобы облегчить переход от С-функций к С++ классам.

4) Потоковые классы делятся на три группы (шаблонов классов):

– basic_istream, basic_ostream – общие потоковые классы, которые могут быть связаны с любым буферным объектом;

– basic_ifstream, basic_iostream – потоковые классы для считывания и записи файлов;

– basic_istringstream, basic_ostringstream – потоковые классы для объектов-строк.

5) Каждый потоковый класс поддерживает буферный объект, который предоставляет память для передаваемых данных, а также важнейшие функции ввода/вывода низкого уровня для их обработки.

6) Базовым шаблоном классов basic_ios (для потоковых классов) и basic_streambuf (для буферных классов) передаются по два параметра шаблона:

– первый параметр (charT) определяет символьный тип;

– второй параметр (traits) – объект типа ios_traits (шаблон класса), в котором заданы тип и функции, специфичные для используемого символьного типа;

– для типов char и wchar_t образованы соответствующие объекты типа ios_traits и потоковые классы.

Пример шаблона потокового класса.

template <class charT, class traits = ios_traits <charT>> class basic_istream: virtual public basic_ios <charT, traits>;

7.2 Потоковые классы в С++

Библиотека потоковых классов С++ построена на основе двух базовых классов: ios и streambuf .

Класс streambuf обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода. Методы и данные класса streambuf программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих.

Базовые потоки ввода-вывода.

Для ввода с потока используются объекты класса istream, для вывода в поток – объекты класса ostream.

В классе istream определены следующие функции:

– istream& get(char* buffer,int size,char delimiter='\n');

Эта функция извлекает символы из istream и копирует их в буфер. Операция прекращается при достижении конца файла либо при скопировании size символов, либо при обнаружении указанного разделителя. Сам разделитель не копируется и остается в streambuf. Последовательность прочитанных символов всегда завершается нулевым символом.

– istream& read(char* buffer,int size);.

Не поддерживает разделителей, и считанные в буфер символы не завершаются нулевым символом.

– istream& getline(char* buffer,int size, char delimiter='\n');

Разделитель извлекается из потока, но в буфер не заносится. Это основная функция для извлечения строк из потока. Считанные символы завершаются нулевым символом.

– istream& get(streambuf& s,char delimiter='\n');

Копирует данные из istream в streambuf до тех пор, пока не обнаружит конец файла или символ-разделитель, который не извлекается из istream. В s нулевой символ не записывается.

– istream get (char& C);

Читает символ из istream в C. В случае ошибки C принимает значение 0XFF.

– int get();

Извлекает из istream очередной символ. При обнаружении конца файла возвращает EOF.

– int peek();

Возвращает очередной символ из istream, не извлекая его из istream.

– int gcount();

Возвращает количество символов, считанных во время последней операции неформатированного ввода.

– istream& putback(C);

Если в области get объекта streambuf есть свободное пространство, то туда помещается символ C.

– istream& ignore(int count=1,int target=EOF);

Извлекает символ из istream, пока не произойдет следующее:

– функция не извлечет count символов;

– не будет обнаружен символ target;

– не будет достигнут конец файла.

В классе ostream определены следующие функции:

– ostream& put(char C);

Помещает в ostream символ C.

– ostream& write(const char* buffer,int size);

Записывает в ostream содержимое буфера. Символы копируются до тех пор, пока не возникнет ошибка или не будет скопировано size символов. Буфер записывается без форматирования. Обработка нулевых символов ничем

не отличается от обработки других. Данная функция осуществляет передачу необработанных данных (бинарных или текстовых) в ostream.

– ostream& flush();

Сбрасывает буфер streambuf.

Для прямого доступа используются следующие функции установки позиции чтения - записи.

При чтении

– istream& seekg(long p);

Устанавливает указатель потока get (не путать с функцией) со смещением p от начала потока.

– istream& seekg(long p,seek_dir point);

Указывается начальная точка перемещения.

enum seek_dir{beg,curr,end}

Положительное значение p перемещает указатель get вперед (к концу потока), отрицательное значение p – назад (к началу потока).

– long tellg();

Возвращает текущее положение указателя get.

При записи

– ostream& seekp(long p);

Перемещает указатель put в streambuf на позицию p от начала буфера streambuf.

– ostream& seekp(long p,seek_dir point);

Указывается точка отсчета.

– long tellp();

Возвращает текущее положение указателя put.

Помимо этих функций, в классе istream перегружена операция >>, а в классе ostream <<. Операции << и >> имеют два операнда Левым операндом является объект класса istream (ostream), а правым – данное, тип которого задан в языке.

Для того чтобы использовать операции << и >> для всех стандартных типов данных, используется соответствующее число перегруженных функций operator<< и operator>>. При выполнении операций ввода-вывода в зависимости от типа правого операнда вызывается та или иная перегруженная функция operator.

Поддерживаются следующие типы данных: целые, вещественные, строки (char*). Для вывода – void* (все указатели, отличные от char*, автоматически переводятся к void*). Перегрузка операции >> и << не изменяет их приоритета.

Функции operator<< и operator>>возвращают ссылку на тот потоковый объект, который указан слева от знака операции. Таким образом, можно формировать “цепочки ” операций.

```
cout << a << b << c;
cin >> i >> j >> k;
```

7.3 Файловый ввод-вывод

Потоки для работы с файлами создаются как объекты следующих классов:

1) ofstream – запись в файл;
2) ifstream – чтение из файла;
3) fstream – чтение/запись.

Для создания потоков имеются следующие конструкторы:

– fstream();

создает поток, не присоединяя его ни к какому файлу.

– fstream(const char* name,int mode,int p=filebuf::openprot);

создает поток, присоединяет его к файлу с именем name, предварительно открыв файл, устанавливает для него режим mode и уровень защиты p. Если файл не существует, то он создается. Для mode=ios::out, если файл существует, то его размер будет усечен до нуля.

Флаги режима определены в классе ios и имеют следующие значения:

1) in – для чтения;
2) out – для записи;
3) ate – индекс потока помещен в конец файла. Чтение больше не-допустимо, выводные данные записываются в конец файла;
4) app – поток открыт для добавления данных в конец. Независимо от seekp() данные будут записываться в конец;
5) trunc – усечение существующего потока до нуля.

8 Лекция №8. Элементы обобщенного программирования. Стандартная библиотека шаблонов

Цель: получить представление о принципах обобщенного программирования и об использовании стандартной библиотеки шаблонов.

8.1 Обобщенное программирование

Обобщённое программирование – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Возможности обобщённого программирования впервые появились в 70-х годах в языках CLU и Ada, а затем во многих объектно-ориентированных языках таких, как C++, Java, Object Pascal, D, Eiffel, .NET и других.

Средства обобщённого программирования реализуются в языках программирования в виде тех или иных синтаксических средств, дающих возможность описывать данные (типы данных) и алгоритмы (процедуры, функции, методы), параметризуемые типами данных. У функции или типа

данных явно описываются формальные параметры-типы. Это описание является обобщённым и в исходном виде непосредственно использовано быть не может.

В тех местах программы, где обобщённый тип или функция используется, программист должен явно указать фактический параметр-тип, конкретизирующий описание. Например, обобщённая процедура перестановки местами двух значений может иметь параметр-тип, определяющий тип значений, которые она меняет местами. Когда программисту нужно поменять местами два целых значения, он вызывает процедуру с параметром-типом «целое число» и двумя параметрами – целыми числами, когда две строки – с параметром-типом «строка» и двумя параметрами – строками. В случае, с данными программист может, например, описать обобщённый тип «список» с параметром-типом, определяющим тип хранимых в списке значений. Тогда при описании реальных списков программист должен указать обобщённый тип и параметр-тип, получая, таким образом, любой желаемый список с помощью одного и того же описания.

Компилятор, встречая обращение к обобщённому типу или функции, выполняет необходимые процедуры статического контроля типов, оценивает возможность заданной конкретизации и при положительной оценке генерирует код, подставляя фактический параметр-тип на место формального параметра-типа в обобщённом описании. Естественно, что для успешного использования обобщённых описаний фактические типы-параметры должны удовлетворять определённым условиям. Если обобщённая функция сравнивает значения типа-параметра, любой конкретный тип, использованный в ней, должен поддерживать операции сравнения, если присваивает значения типа-параметра переменным – конкретный тип должен обеспечивать корректное присваивание.

Язык C++ поддерживает парадигму обобщенного программирования на основе использования родовых компонентов. Родовые (параметризованные) компоненты обладают свойством адаптивности к конкретной ситуации, в которой такой компонент используется, что позволяет разрабатывать достаточно универсальные и в то же время высокоэффективные компоненты программ (в частности, объекты).

Обобщенное программирование в языке C++ реализовано с помощью шаблонов (template). В C++ определено два вида шаблонов: шаблоны-классы и шаблоны-функции.

Шаблоны-классы могут использоваться многими способами, но наиболее очевидным является их использование в качестве адаптивных объектов памяти. Шаблоны-функции могут использоваться для определения обобщенных алгоритмов. Основное отличие шаблона-функции от шаблона-класса в том, что не нужно сообщать компилятору, к каким типам параметров применяется функция, он сам может определить это по типам ее формальных параметров.

Возможность обобщенного программирования на языке C++ обеспечивается стандартной библиотекой шаблонов STL (Standard Template Library). Она включена в последнюю предварительную версию Стандарта языка C++.

8.2 Стандартная библиотека шаблонов

В данном разделе рассматривается одна из самых важных составляющих языка C++ - стандартная библиотека шаблонов (STL). Ее включение в язык C++ было одним из основных успехов, достигнутых в ходе стандартизации. Библиотека содержит универсальные шаблонные классы и функции, реализующие большое количество разнообразных алгоритмов и структур данных. Например, в ней определены шаблонные классы векторов, списков, очередей и стеков, а также многочисленные процедуры для работы с ними. Поскольку библиотека состоит из шаблонных классов, ее алгоритмы и структуры данных можно применять практически для любых типов данных.

С точки зрения разработки программного обеспечения библиотека стандартных шаблонов представляет собой довольно сложный проект, в котором использованы наиболее ухищренные свойства языка C++. Синтаксис шаблонных классов, описанных в STL, выглядит довольно устрашающе, хотя на самом деле все не так сложно, как кажется на первый взгляд [1].

STL обеспечивает общецелевые, стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных.

STL строится на основе шаблонов классов, и поэтому входящие в нее алгоритмы и структуры применимы почти ко всем типам данных. Ядро библиотеки образуют три элемента: контейнеры, алгоритмы и итераторы.

Контейнеры (containers) - это объекты, предназначенные для хранения других элементов. Например, вектор, линейный список, множество.

Ассоциативные контейнеры (associative containers) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям.

В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип iterator, который определен в различных контейнерах.

В STL определены два типа контейнеров: последовательности и ассоциативные.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться map (ассоциативным массивом). С другой стороны, если преобладают операции, характерные для списков, можно воспользоваться контейнером list. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди queue, очереди с двумя концами deque, стека stack. По умолчанию пользователь должен использовать vector ; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров и, в общем случае, со всеми видами источников информации - унифицированным способом ведет к понятию обобщенного программирования. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров.

В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

- bitset - множество битов <bitset.h>
- vector - динамический массив <vector.h>
- list - линейный список <list.h>
- deque - двусторонняя очередь <deque.h>
- stack - стек <stack.h>
- queue - очередь <queue.h>
- priority_queue - очередь с приоритетом <queue.h>
- map - ассоциативный список для хранения пар ключ/значение, где с каждым ключом связано одно значение <map.h>
- multimap - с каждым ключом связано два или более значений <map.h>
- set - множество <set.h>
- multiset - множество, в котором каждый элемент не обязательно уникален <set.h>

Работа с классом-контейнером «Вектор».

Конструкторы.

В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());

explicit vector(size_type число, const T &значение = T(),
                const Allocator &a = Allocator());
```

```
vector(const vector<T, Allocator> &объект);

template<class InIter> vector(InIter начало, InIter конец,
                              const Allocator &a = Allocator());
```

Первая форма представляет собой конструктор пустого вектора. Во второй форме конструктора вектора число элементов - это число, а каждый элемент равен значению значения. Параметр значения может быть значением по умолчанию. Третья форма конструктора вектор - это конструктор копирования. Четвертая форма - это конструктор вектора, содержащего диапазон элементов, заданный итераторами начало и конец.

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и == .

Список литературы

Основная

1 Дорогов В.Г. Основы программирования на языке С.-М., 2013

2 Пахомов Б. С/С++ и MS Visual С++ 2010 для начинающих.-СПб.: «БХВ-Петербург», 2011

3 Шилдт Г. Полный справочник по С++. - М. : Издательский дом «Вильямс», 2006 - 800 с.

4 Ашарина И.В. Объектно-ориентированииое программирование в С++: лекции и упражнения.-М., 2008

5 Немцова Т.И. Программирование на языке высокого уровня. Программирование на языке С++.-М.: «Форум», 2012

6 Страуструп Б. Язык программирования С++.-М, 2004, 2005, 2011, 2012

7 Хорев П.Б. Объектно-ориентированное программирование.-М.: Академия, 2011

Дополнительная

8 Ашарина И.В. Основы программирования на языках С и С++.-М., 2002, 2012

9 Шлее М. Qt4.5 профессиональное программирование на С++.-СПб: «БХВ-Петербург», 2010

10 Архангельский А.Я. С++ BUILDER работа с документами Excel.-М.: «Бином», 2009

11 Аймұқатов А. Объектілі бағытты бағдарламалау негіздері.-А.: «Фолиант», 2010

12 Выгодский М.Я. Справочник по высшей математике. - М.: АСТ: Астрель, 2006 - 991 с.

13 Выгодский М.Я. Справочник по элементарной математике. М.: АСТ: Астрель, 2006 - 509 с.

14 Куралбаев З.К. Основы информационных систем. Конспект лекций.-А., 2009

Сводный план, 2014, поз. 373

Ерик Сулейменович Турганбаев

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ С++

Конспект лекции
для студентов специальности
5В070400 – Вычислительная техника и программное обеспечение

Редактор Л.Т.Сластихина
Специалист по стандартизации Н.К.Молдабекова

Подписано в печать Формат 60х84/16
Тираж 50 Бумага типографская №1
Объем 2,38 уч. изд. л. Заказ___Цена 1190 тн

Копировально-множительное бюро
некоммерческого акционерного общества
«Алматинский университет энергетики и связи»
050013 Алматы, Байтұрсынұлы, 126