**Лекция 1**

ООП

Объект = структура + данные.

ООП представляет собой технологию программирования, в основе которой лежит подход, позволяющий людям формировать модели объектов реального мира. Сложная проблема раскладывается на составные части и каждая составляющая становится самостоятельным объектом, содержащим свои ресурсы и механизмы, относящиеся к этому объекту.

Все языки ООП основаны на трёх концепциях, называемых инкапсуляцией, наследованием и полиморфизмом. С точки зрения системологии, объект определяется структурой и поведением или, говоря программным языком: объединением данных и процедур, обрабатывающих их в одно целое.

Инкапсуляция – это механизм, который объединяет данные и код, манипулирующий этими данными вместе, а также защищает и то и другое от внешнего вмешательства. Целью инкапсуляции является сокрытие данных от несанкционированного доступа и изменения и работа с этими данными только через специально разработанные для этой цели процедуры.

Наследование – это процесс, посредством которого один объект может приобретать свойства другого. Наследование позволяет одним объектам приобретать атрибуты и поведение других объектов. Наследование позволяет сделать разработку более экономной и обозримой, так как объекты пользуются одними и теми же атрибутами и формами поведения без дублирования реализующих их программных кодов.

Определение объекта и затем – использование его для построения целой иерархии производных объектов, причём каждый производный объект (потомок) наследует доступ к коду и данным всех своих прародителей называется наследованием.

ООП – это процесс разработки иерархии объектов.

При наследовании один объект может сохранять основные свойства другого объекта - предка и добавлять к ним черты, характерные только для него. Целью наследования является использование существующих наработок. Наследование – это процесс внесения уточнений.

Пример:  
Рассмотрим рулевое колесо автомобиля – оно работает одинаково независимо от того, какой используется привод. Главным здесь является то, что рулевое колесо везде выглядит одинаково, а механизмы, выполняющие поворот, в каждом случае свои. То есть одно и то же имя используется для решения схожих, но технически разные задач.

Придавание действию одного имени, которое совместно используется объектами всей иерархии, причём каждый объект иерархии использует это действие своим собственным подходящим для него образом, называется полиморфизмом. Целью полиморфизма применительно к ООП является использование одного имени для задания близких по сути, но разных по реализации действий. В общем случае концепцией полиморфизма является идея – один интерфейс и множество методов.

В C++ идея полиморфизма используется так же для перегрузки функций и операторов.

В C++ инкапсуляция состоит в объединении данных и кода в единой упаковке, называемой классом, который определяет способ хранения и использования данных.

В C++ отдельный класс определяется посредством типов struct, union, class и включает в себя данные и функции (методы) для работы именно с этими данными.

Модификаторы доступа

public – ресурс, доступный всем  
protected – по иерархии  
private – личное

Структуры, объединения и классы могут содержать не только объявления данных, но и функции, с ними работающие. В C++ ключевые слова struct, unit и class могут использоваться для определения классов – типов даных, обладающих инкапсуляцией. Класс, определённый с помощью union – это класс, в котором все компоненты являются открытыми для всех и этот уровень не может быть изменён. Не может использоваться для построения иерархии объектов. Класс, определённый с помощью struct – это класс, в котором все компоненты являются public по умолчанию, но имеется возможность менять эту установку. Может использоватьсмя для построения иерархии объектов. Класс, определённый с помощью class – это класс, в котором все компоненты являются закрытыми по умолчанию, но в случае необходимости имеется возможность изменить уровень доступа. Используется для построения иерархии объектов.

Когда мы говори о классах в C++, то включаем в это понятие те данные и функции, которые определены с помощью ключевых слов struct, union и class. Обычно ограничения на уровне доступа касаются элементов данных. Данные имеют атрибут private или protected, а функция, работающая с ними, public.Если не будет процедур, имеющих доступ к закрытым данным, то эти данные останутся невостребованными и никто не сможет с ними работать. Объединение сокрытых данных и процедур их обслуживающих реализуется через инкапсуляцию.

Создание классов

Struct point  
{  
int x;  
int y;  
};

Тип point даёт необходимую гибкость мышления. Когда нужно думать о координатах x и y отдельно, мы рассматриваем их как независимые элементы или поля структуры point. С другой стороны, когда нужно думать о координатах x и y, как о работающих вместе для определения положения точки на экране, мы можем думать о них как о едином целом, то есть как о структуре point. Предположим, что необходимо отобразить светящуюся точку на экране. В дополнение к координатам имеет смысл добавить ещё один элемент, определяющий, будет ли эта точка видимой на экране или нет.

Struct point  
{  
int x;  
int y;  
bool visible;  
};

Класс в C++ - это тип данных, который может содержать как данные членов, так и функции членов – методы. Метод или функция-член класса – это функция, объявленная внутри определяемого класса и непосредственно с ним связанная. Данные класса – это то, что класс знает, а методы класса – это то, что класс делает с этими данными.

Добавим простые методы: getx setx:

Существуют два способа включения методов в классы:

1. Объявление внутри функции.  
   Struct point  
   {  
   int x;  
   int y;  
   bool visible;  
   int GetX() {return x;}  
   void SetX(int NewX) {x = new x;}  
   };  
   Такая форма определения используется для маленьких функций
2. Объявление функции внутри класса с последующим её определением вне класса. Этот способ чаще всего используется в профессиональной деятельности, потому что в больших проектах удобно иметь в одном файле .h иметь только объявления классов, а в файле .cpp иметь реализацию классов. Файл с объявлением классов доступен всем участникам проекта, а файл .cpp доступен только программисту, отвечающему за методы класса.

При втором способе метод просто объявляется внутри struct Point, а определение метода выполняется в произвольном месте программы или в другом файле вне тела объявления класса.

Point.cpp  
Point::GetX() {return x;}  
Point::SetX(int NewX) {x = new x;}

Операция : - это расширение области видимости. Она указывает компилятору, какой области принадлежит функция. Так как, рядом может быть другие функции GetX, принадлежащие другим классам, то требуется явно указать имя класса.

Методы выполняют операции над полями или компонентами своего класса. Если бы GetX была бы не методом, а обычной функцией C++, то она вызывалась бы простым обращением. Несмотря на то, что конструкция struct в C++ обеспечивает связь между элементами данных и методами, она не очень надёжна. Доступ данных является public, то есть любой оператор из программы может читать или менять элементы данных этого класса. В данном случае возникают те же самые проблемы, что и с глобальными переменными: они отовсюду доступны, но имеется опасность несанкционированного изменения переменных.

Модификаторы доступа

Три ключевых слова определяют правила доступа к элементам класса. Соответствующее ключевое слово, сопровождаемое двоеточием, располагается перед объявляемыми методами класса.

Private

Элементы, стоящие за этим словом, доступны только методам, объявленным внутри класса. Снаружи до них не добраться, по наследству доступ не передаётся.

Protected

Элементы, стоящие за этим словом доступны только методам, объявленным внутри класса и методам потомков, то есть производным класса. Снаружи до них не добраться.

Public

Элементы, стоящие за этим словом, доступны из любой точки программы, имеющие ту же область действия, что и определения классов.

Элементы могут иметь дополнительное описание static. Элемент класса с описанием static будет общим для всех объектов данного класса. Класс типа struct является public по умолчанию, поэтому для объявления некоторой части элементов закрытыми, необходимо использовать private. Если после этого остаются элементы, которые нужно сделать общедоступными, используется private. Хорошим стилем программирования является сокрытие данных во избежание несанкционированного изменения информации и для сохранения её целостности. Общим правилом является закрытие прямого доступа к данным и возможность доступа к ним только через методы (функции этого класса, которые в общем случае могут быть private или protected).

В ООП хорошим стилем является априорное объявление всех элементов, как private или protected с последующим тщательным отбором элементов public. Такими свойствами обладает конструкция class в отличие от struct.

Struct Point  
{ private:  
int x;  
int y;  
bool visible;  
public:  
int GetX();  
void SetX(int NewX);   
};

class Point  
{ private:  
int x;  
int y;  
bool visible;  
public:  
int GetX();  
void SetX(int NewX);   
};

Единственным различием между struct и class являются правила доступа, используемые по умолчанию. Де факто программисты используют class с явным описанием модификатора доступа.

Конструкторы и деструкторы

Существуют специальные функции, получившие названия конструкторов и деструкторов, которые зарезервированы за каждым классом. Стандартной проблемой при создании объектов является их инициализация. Перед тем, как использовать элементы данных, следует под них выделить память и установить корректные начальные значения. Рассмотрим описание и объявление

struct SPoint  
{  
int x;  
int y;  
bool visible;  
public:  
int GetX();  
void SetX(int NewX);   
};

class CPoint  
{  
int x;  
int y;  
bool visible;  
public:  
int GetX();  
void SetX(int NewX);   
};

SPoint SP = {10, 20, false};  
CPoint CP = {10, 20, false};

Причина невозможности инициализации CP в том, что переменные в нём private.

SPoint SP1;  
SP1.X = 20;  
SP1.Y = 30;  
SP1.Visible = false;

Такой способ обеспечивает создание одного объекта. Если будут другие, то потребуется повторить эти действия необходимое число раз. Естественным выходом является создание инициализирующей функции.

Метод, определяющий начальные значения объекта называется конструктором. Конструктор вызывается при создании нового объекта класса, то есть, когда под него отводится память и тогда же она инициализируется. Имя конструктора всегда совпадает с именем класса, именно таким путём компилятор узнаёт, что имеет дело с конструктором. Конструктор, как и другая функция, может иметь аргументы. Тело конструктора строится так же, как и тело любого другого метода, поэтому конструктор может обращаться к любому методу и любому элементу класса. Отличительной чертой конструктора является то, что он не возвращает никакого значения, даже значения типа void. Конструкторов в классе может быть несколько.

**Лекция 2**

Определение класса:  
class CPoint  
{  
private:  
int x;  
int y;  
bool visible;  
public:  
int GetX();  
CPoint(int InitX, int InitY = 20);   
~CPoint();  
};

Определение конструктора:  
CPoint::CPoint(int InitX, int InitY)  
{  
X = InitX;  
Y = InitY;  
visible = false;  
cout << “Constructor CPoint\n”  
}

Создание экземпляра:  
CPoint CPa(1;1)  
CPoint CPb(10);

Программист не обязан писать код, вызывающий конструктор классов. Всю работу за него сделает компилятор.

Деструкторы уничтожают объекты классов, предварительно созданные конструктором, путём очистки значений и освобождения памяти. Деструктор имеет имя, совпадающее с именем класса, но впереди стоит знак «~» и, в отличие от конструктора, он не может иметь параметров. Незачем передавать параметры удаляемому объекту. Деструктор не имеет типа и ничего не возвращает. Объявляется в методах класса, как правило, после конструктора. Деструктор в классе всегда один, но деструкторы могут быть виртуальными.

CPoint::~CPoint()  
{  
cout << “Destructor CPoint\n”  
}

Момент вызова деструктора выбирает сам компилятор, поэтому программный код не должен содержать явных обращений к деструктору. Деструктор вызывается автоматически при выходе из блока программного кода, в котором объект был определён. В случае отсутствия своего деструктора сработает деструктор, заданный по умолчанию.

Пример:  
//ExOOP\_01.cpp  
#include <iostream>  
using namespace std;  
class CPoint  
{  
private:  
int x;  
int y;  
public:  
Point(int InitX, int InitY);   
{  
static int iConst = 0;  
iConst ++;  
X = InitX; Y = InitY;  
cout << “/Point():Call #” << iConst << “Constructor Point X= “ << X << “ Y = “ << Y << endl;  
};  
int GetX()  
{return x;}  
int GetY()  
{return Y;}  
~Point();  
}  
Point::~Point()  
{static iDestr = 0;  
iDestr++;  
cout << “\nPoint():Call #” << iDestr << “Destructor Point“;  
cout << iDestr << “/~Point(); X = “ << X << endl;  
cout << “/~~Point().GetX() = “ << GetX() << endl;  
}  
// основная программа  
int main()  
{  
int NewX, NewY;  
cout << “/main: Enter X Cord:”;  
cin >> NewX;  
cout << “/main: Enter Y Cord:”;  
cin >> NewY;  
// создание экземпляра  
Point Point\_1 = Point(NewX; NewY); // явный вызов  
// Point Point\_1(NewX; NewY); // скрытый вызов  
// Point Point\_1; // память под поля выделяется но значения не определены  
// обращение к данным класса (полям) через методы  
cout << “/Point\_1: Point\_1.GetX() = “ << Point\_1.GetX() << endl;  
cout << “/Point\_1: Point\_1.GetY() = “ << Point\_1.GetY() << endl;  
Point \*pP  
pP = &Point\_1;  
// обращение к методам через указатель  
cout << “/pP = &Point\_1: pP -> GetX() = “ << pP -> GetX() << endl;  
cout << “/pP = &Point\_1: pP -> GetY() = “ << pP -> GetY() << endl;  
// второй вызов конструктора – создан ещё один объект  
Point Point\_2 (NewX + 10, NewY + 20);  
cout << “/Point\_2: Point\_1.GetX() = “ << Point\_2.GetX() << endl;  
cout << “/Point\_2: Point\_1.GetY() = “ << Point\_2.GetY() << endl;

Пересмотр класса Point:

Точка – это местонахождение, которое светится. В определении точки скрыто местонахождение. Так как, все точки должны иметь местонахождение, то мы можем сказать, что тип Point является производным от типа местонахождения. Тип Point наследует всё, что имеет тип Location и дополнительно имеет то своё, что делает тип Point отдельным типом. Процесс, в результате которого один тип получает свойства другого типа, называется наследованием. Класс, от которого наследует, называется базовым классом, а класс, наследующий от базового – производным. Базовый класс определяет все те качества, которые будут общими для всех производных классов. Производный класс может быть базовым для других классов. Допускается для одного производного класса иметь несколько базовых, это называется множественное наследование, но это повышает конфликтность в программах, например, когда поля x и y имеются у обоих предков.

Вертикальное наследование имеет место быть в ситуации, когда у одного предка имеется один потомок. Веерное наследование появляется в том случае, когда у одного предка несколько потомков. В общем случае иерархия классов имеет вертикальное и веерное наследование.

В общем случае классы Location и Point определены следующим образом:

// QTOOP\_02  
// Point\_02.h – объявление двух классов, наследование  
class Location  
{  
protected:  
int X;  
int Y;  
public: // открытый доступ извне к методам класса  
Location (int InitX, int InitY);  
~Location()  
// для работы с закрытыми полями X и Y  
int GetX();  
int GetY();  
void SetX(int NewX){ X = NewX};  
void SetY(int NewY){ Y = NewY};  
} // class Location  
class Point : public Location  
{ // по умолчанию, всё – private  
protected:  
bool visible; // светится точка или нет  
public:  
Point(int InitX, int InitY);  
~Point();  
// для работы с закрытым полем visible  
bool GetVisible();  
void SetVisible(bool NewVisible);  
void Show();  
void Hide();  
void MoveTo(int NewX, int NewY);  
} // class Point  
  
// Point\_02.cpp – реализация методов класса  
#include Point\_02.h  
extern HDC hdc;  
// class Location  
// конструктор  
Location::Location(int InitX, int InitY)  
{ x = InitX; Y = InitY;} // Location()  
// деструктор  
Lcation::~Location(){}  
int Location::GetX() {return X;}  
int Location::GetY() {return Y;}  
// class Location  
// реализация методов класса Point  
// конструктор  
Point::Point(int InitX, int InitY):Location(InitX, InitY)  
{  
visible = false;  
} // Point()  
// деструктор  
Point::~Point(){}  
bool Point::GetVisible(){return Visible)  
void Point::SetVisible(bool NewVisible)  
{Visible = NewVisible;}  
  
void Point::Show()  
{  
Visible = true;  
SetPixel(hdc, X, Y, RGB(255, 0, 0));  
} //Point::Show  
  
void Point::Hide()  
{  
Visible = false;  
SetPixe(hdc, X, Y, RGB(0, 0, 255));  
} // Pixel::Hide  
  
void Point::MoveTo(int NewX, int NewY)  
{  
Hide();  
X = NewX;  
Y = NewY;  
Show();  
} // Point::MoveTo()  
  
//ExOOP\_02.cpp  
#include <iostream>  
#include “Point\_02.h”  
using namespace std;  
HDC hdc;  
int main()  
{   
int X0 = 300;  
int Y0 = 100;  
int i;  
…  
Point APoint(X0, Y0);  
APoint.Show(); // показать точку  
for(i = 0; i < 6; i++)  
{  
Apoint.MoveTo(X0 + 50, Y0 + 35\*i);  
cout << i << “Press ENTER\n”;  
getchar();  
}  
while(1)  
{  
for(i = 0; i < 100; i++)  
{  
Apoint.MoveTo(X0 + 100+ i, Y0 + i);  
Sleep(50);  
}  
}

**Лекция 3**

Добавление нового элемента в иерархии классов

Начнём развивать нашу иерархию геометрических объектов, добавив в наследники точки новый элемент – круг. Когда мы добавляем потомка в иерархию мы должны понимать, как потомок связан с предком, что он от него наследует и что нового будет у потомка. Круг – это толстая точка. Он имеет всё, что есть у точки и радиус. Класс Circle имеет своё собственное поле Radius плюс все те поля, которые он унаследовал от класса Point. Класс Circle имеет поля X, Y и Visible, даже если они не видны в определении класса Circle. Класс Circle описывает новое поле – радиус, поэтому инициализация экземпляра класса потребует инициализации как унаследованных полей, так и нового поля Radius. Вместо того, чтобы непосредственно присваивать величины полям X, Y, Visible в классе Circle, используем инициализацию для полей X, Y, Visible из класса Point, а поле Radius, про которое не знает Point, инициализируем в классе Circle. Такая инициализация делается в конструкторе Circle с использованием конструктора Point.

Circle::Circle(int InitX, int InitY, int InitRad):Point(InitX, InitY)  
{  
Radius = InitRad;  
}  
Вызов конструктора, который обеспечивает доступ к полям X, Y, Visible в Point.

Можно было бы всю инициализацию провести в конструкторе Circle, не обращаясь к Point, но это плохой стиль программирования. Возможно, Point и Location выполняют ещё какую-нибудь скрытую инициализацию, поэтому, если сопровождающий программист получил иерархию с закрытым доступом к исходному коду, то отказавшись от инициализации через конструктор предка имеется риск что-нибудь испортить. Если мы обращаемся к конструктору прародителя, то мы имеем корректную инициализацию и то, что изменения, сделанные в методе прародителя, затронут всех потомков.

Круг, как и любая геометрическая фигура иерархии должен уметь показывать себя на экране и скрывать своё изображение. За это отвечают функции show и hide. Взять их реализацию от предка Point мы не можем, потому что его функции show и hide умеют показывать и скрывать точку. О том, как выглядит круг, знает только потомок Circle, поэтому реализация унаследованных функций show и hide должна быть своя. Это называется переопределением функций. Имя функции и сигнатура взяты от предка, а реализация соответствует логике потомка.

К функции перемещения фигуры MoveTo для класса Circle возникает вопрос: нужно ли иметь свою функцию перемещения круга или можно воспользоваться функцией предка? В теле функции PointMoveTo стоят вызовы функции show и hide. Функции с такими же именами есть в потомке – классе Circle. Попробуем, может быть удастся при работе с кругом обратиться к функции предка PointMoveTo и её средствами переместить фигуру потомка – круга. Поэтому в данной реализации класса Circle мы откажемся от своей реализации функции Circle::MoveTo() и возьмём её из предка. Добавление нового поля в класс не проходит бесследно. Оно тянет за собой добавление новых функций, обслуживающих это поле.

Circle::Expand(int DeltaRad)  
Circle::Reduce(int Deltarad)

В общем случае, появление нового поля требует добавления в класс двух относительно стандартных открытых методов:  
GetValue()  
SetNewValue(int NewVal)  
которые отвечают за чтение и изменение нового закрытого поля. Необходимость наличия этих двух функций обусловлена требованием инкапсуляции работы с полями класса только через методы класса. В C# например, методы Get и Set уже встроены в класс и называются геттеры и сеттеры. В наших примерах, из методических соображений, мы откажемся от использования этих функций, чтобы не усложнять структуру класса, хотя в общем случае, методы GetValue и SetValue безусловно нужны.

//ExOOP\_03  
//Point\_03.h  
//Объявление трёх классов, наследование, статические правила, раннее связывание  
class Location  
{ X, Y  
…  
} // class Location  
class Point   
{  
void Show();  
void Hide();  
void MoveTo();  
}  
  
//Point\_02.h  
class Circle:public Point  
{  
protected:  
int Radius;  
public:  
Circle(int InitX = 100, int InitY = 100, int InitRad = 30);  
~Circle();  
void Show();  
void Hide();  
…  
void Expand(int DeltaRad);  
void Reduce(int DeltaRad);  
} // class Circle

Реализация методов класса:

//Point\_03.cpp  
Circle::Circle(int InitX, int InitY, int InitRad):Point(InitX, InitY)  
{  
Radius = InitRad;  
} // Circle::Circle()  
void Circle::Show()  
{ visible = true;  
HPEN Pen = CreatPen(PS\_SOLID, 2, RGB(255, 0, 0));  
SelectObject(hdc, Pen);  
Ellipse(hdc, X – Radius, Y-Radius, X+Radius, Y+Radius);  
DeleteObject(Pen);  
} // Circle::Show  
void Circle::Hide()  
{  
…RGB(0, 0, 255);  
} // Circle::Hide()  
Void Circle::Expand(int DeltaRad)  
{  
Hide();  
Radius = Radius + DeltaRad;  
if (Radius < 2)  
Radius = 2;  
Show();  
} // Circle::Expand()  
void Circle::Reduce(int DeltaRad)  
{  
Expand(-DeltaRad);  
} // Circle::Reduce()

//ExOOP\_03.cpp  
  
// Наследование рисования окружности, перемещение точки/окружности, изменение радиуса  
#include “Point\_03.h”  
int main()  
{  
int X0 = 300, Y0 = 300;  
int Radius0 = 50, DeltaRad = 30;  
// Точка  
point Apoint(X0, Y0);  
Apoint.Show();  
while(1)  
{ if(KEY\_DOWN(49)) // 1  
break;  
// Круг  
Circle ACircle(X0, Y0, Radius);  
ACircle.Show()  
[(50)]  
ACircle.MoveTo(X0, Y0 – 2Radius);  
[ (51))  
ACircle.Show()  
[(52)) // увеличить радиус круга  
ACircle.Expand(DeltaRad)  
[(53))  
ACircle.Reduce(DeltaRad)  
[(54))

Вернёмся к объявлению и определению в нашей иерархии классов и создадим свою функцию MoveTo для класса Circle. В предыдущем примере для перемещения круга мы вызывали функцию MoveTo предка. Добавим в объявление классов функцию MoveTo и в файле Point\_03.cpp скопируем MoveTo из Point и поменяем префикс на Circle.

Запустим программу ещё раз. Изменения в результатах будут касаться перемещения круга вместо точки. Дальнейшие результаты программы остались прежними.

Наследование статических методов

Статическими называются методы, вызов которых определён на этапе компиляции и адреса вызываемых функций не меняются. При размещении ссылок на функции во время компиляции происходит раннее связывание функций. Ключевое слово static, которое может встречаться перед именем функции, к данному определению не имеет никакого отношения.

Функция Point::MoveTo и Circle::MoveTo полностью идентичны. Никаких изменений не произошло, кроме копирования тела функции MoveTo и изменения префикса с Point на Circle.

Все методы, показанные в связи с Location, Point и Circle являются статическими, то есть определёнными заранее на этапе компиляции. Возникает вопрос: если методы MoveTo классов Point и Circle одинаковы, зачем в Circle копировать MoveTo? Дело в том, что если из ACircle.MoveTo делается вызов PointMoveTo из предка, то на экране будет передвигаться не круг, а точка. Если ACircle вызывает свой MoveTo, то Hide и Show будут прятать и рисовать круг на новом месте. Почему это происходит? Эта ситуация имеет отношение к способу вызова методов компилятором. Когда компилятор компилирует методы класса Point, он сначала встречает Point::Hide и Point::Show и компилирует их код. Немного позднее в файле он встречает Point::MoveTo, который вызывает Point::Hide и Point::Show. В этом случае, компилятор замещает указатели в исходном коде на Point::Hide и Point::Show адресами их сгенерированного кода в сегменте кода. Поэтому, когда вызывается код для Point::MoveTo, то он в свою очередь вызывает код для Point::Hide и Point::Show. Таким образом, когда вызывается метод класса ACircle.MoveTo, то он ссылается на метод своего предка Point::MoveTo, который вызывает Point::Hide и Point::Show, а не Circle::Hide и Circle::Show. В первом примере вызов ACircle.MoveTo идёт в ближайшую реализацию MoveTo по иерархии. ACircle.MoveTo будет вызывать ближайшую реализацию MoveTo из Point, так как класс Point находится ближе всего. Если никакой производный класс не определил свой собственный метод MoveTo для замены им метода MoveTo из Point, то любой производный класс от класса Point будет вызывать только MoveTo из Point. Решение принято во время компиляции и это и есть раннее связывание. Пока метод Point.MoveTo будет вызывать Point::Hide и Point::Show, метод Point::MoveTo не может быть унаследован. Вместо этого в классе Circle должен быть реализован метод Circle::MoveTo, который полностью копируется из Point::MoveTo, тогда Hide и Show, определённые в классе Circle будут вызываться из Circle::MoveTo и перемещаться будет круг, а не точка. При компиляции методов класса Circle вызываются Circle::Hide и Circle::Show, с которыми будет работать Circle::MoveTo.

Логика компилятора в режиме вызова методов такова: когда вызывается метод, компилятор ищет метод с таким же именем, определённом в текущем классе. Локальное имя берётся первым. Класс Circle определяет методы: конструктор, деструктор, MoveTo, Hode, Show, Expand, Reduce. Если должен быть вызван один из этих методов. То компилятор заменит этот вызов методом класса Circle. Если метод с таким именем нем определён в классе, то компилятор переходит наверх к классу прародителя и ищет метод в этом классе. Если метод с таким именем найден, то адрес метода прародителя заменяет в исходном коде адрес метода потомка. Если метод с таки именем не найден, то компилятор продолжает поиск вверх до следующего прародителя. Когда компилятор попадает в самый верхний класс иерархии и метод не найден, то выдаётся сообщение об ошибке, показывающее, что такой метод не определён. Таким образом, метод должен быть реализован либо на том же уровне, либо у предков. Когда статический унаследованный метод будет найден и использован, то следует помнить, что вызванный метод – это тот самый метод, который был определён и скомпилирован для того уровня, на котором был найден. Если метод прародителя вызывает другие методы, то вызываемые методы будут методами уровня прародителя, даже если потомки имеют методы, которые заменяют методы прародителя.

То есть с уровня Point::MoveTo методы Circle::Hide и Circle::Show не могут быть вызваны, только методы Point::Hide и Point::Show.

**Лекция 4**

Если мы хотим реализовать метод применительно к данному классу, то должны скопировать текст этого метода от предка на уровень данного класса. Наследование статических методов – это мощное решение, но не очень удобное, потому что при появлении новых фигур надо:

1. Скопировать соответствующие методы на уровень нового потомка, а это дублирование кода, что снижает надёжность программы и увеличивает её объём.
2. Приходится выпускать из своих рук исходный тест методов, в которые разными программистами могут быть внесены дополнения, возможно с ошибками. Теряется контроль над разработкой программы.
3. Если у вас нет доступа к исходному коду, разработчик по каким-то причинам перевёл все библиотеки, то развить дальше иерархию не удастся.

Поэтому ситуация, когда ссылки на методы устанавливаются во время компиляции, имеет серьёзные недостатки. Выходом из данной ситуации является использование виртуальных функций.

Виртуальные функции

Можно ли, находясь на текущем уровне, вызывать метод предка, который работал бы с методами не своего уровня, а текущего, причём уровень вызова каждый раз может быть разным. Можно ли MoveTo брать из Point, а Hide и Show из Circle. Ответ – да. Если мы позволим устанавливать ссылки на Hide и Show во время исполнения, то есть MoveTo из Point будет вызывать Hide и Show именно той фигуры, которая сейчас активна. Это называется поздним связыванием и реализуется через механизм виртуальных функций.

Термины раннее связывание и позднее связывание относятся к этапу, на котором обращение к функции связывается с её адресов. В случае раннего связывания, адреса всех функций известны в тот момент, когда происходит компиляция и компоновка программы. Это позволяет приписать каждому обращению к функции соответствующий адрес, который будет зафиксирован за каждой функцией, то есть будет статичным. Адреса всех вызываемых функций определены заранее, до запуска программы. В противоположность этому, в случае позднего связывания, адрес функции не связывается с обращением к ней до того момента, пока обращение не произойдёт фактически, то есть во время выполнения программы.

Виртуальный метод – это метод, который будучи описан и реализован в потомках, замещает собой соответствующий метод везде, даже в методах для предка, если они вызываются для потомка. Программирование отображает реальность, поэтому понятие виртуальной функций не есть изобретение человечества. Разница между вызовом статических и виртуальных методов – это разница между решениями, принимаемыми сейчас, и решением, принятие которого откладывается до прояснения ситуации в последний момент:

1. Я сейчас знаю и могу сделать – статические функции. В этом случае, ещё до запуска программы, заранее известно, кто, что будет делать и в какой момент.
2. Я сейчас не знаю, как буду решать задачу, но по месту соображу – виртуальные функции.

Метод делается виртуальным, когда перед его объявлением в классе ставится ключевое слово virtual. Виртуальными могут быть объявлены только методы, функции и члены класса. Если функция повторно объявляется с той же сигнатурой и типом возвращаемого значения, но на другом уровне (переопределение функции), то новый метод становится виртуальным автоматически, вне зависимости от того, используется ли модификатор virtual или нет. Переопределение очень похоже на перегрузку функции, но перегрузкой не является, так как сигнатура одинаковая. Переопределённые функции отличаются только реализацией.

Предположим, у нас есть целая иерархия объектов: точка, круг, кольцо, эллипс, прямоугольник и каждый из них нужно перемещать. Если пользоваться статическими функциями, то MoveTo придётся копировать в реализацию каждой фигуры. Это не очень хорошо, однако, когда Point::MoveTo вызывает виртуальные Hide и Show, всё происходит по-другому. Каждый тип фигуры имеет свою собственную реализацию Show и то, какая именно реализация Show будет вызвана из Point::MoveTo полностью зависит от того, объект какого класса (какая фигура) вызывает Point::MoveTo в данный момент. Поэтому вызов метода Show в пределах реализации Point::MoveTo должен быть отложенным решением: при компиляции кода Point::MoveTo нельзя принять решение, метод Show какой именно фигуры показывать. Эта информация на момент компиляции недоступна, поэтому принятие решения должно быть отложено до момента выполнения, когда активизируется и станет известен объект (фигура), вызывающий Point::MoveTo. Это называется поздним связыванием. В этом случае, вызываемая и вызывающая функция связываются позже, когда вызов фактически уже сделан, то есть при выполнении программы и делается это через конструктор, который создаёт таблицу виртуальных методов, где указаны адреса Hide и Show объекта. То есть экземпляры текущего класса. Если для базового класса нужно указать методы, которые будут иметь различную реализацию в производных классах (сигнатура метода прежняя, а реализация своя), то такие функции делают виртуальные. Виртуальные методы производных классов заменяют методы базового класса своей реализацией. Для виртуальных функций Hide и Show класса Point не устанавливаются их адреса.

В момент создания конкретного объекта иерархии, конструктор создаст таблицу виртуальных методов, в которых будет вся иерархия классов, начиная от базового до текущего. В одном случае – от Location до Circle, во втором – от Location до Triangle и т.д. Там, где в местах обращений из Point::MoveTo к виртуальным функциям Hide и Show стояло пусто, конструктор установит конкретные адреса виртуальных функций с уровня конкретной фигуры потомка. Таким образом, функции базового класса могут вызывать функции потомка, причём уровень потомка каждый раз может быть разным.

Тип конкретной фигуры при вызове Hide и Show из Point::MoveTo точно не известен во время компиляции и может быть любой фигурой производных классов, то есть Hide и Show вызываются, а какой фигуры – неизвестно, а это в чистом виде полиморфизм.

При создании иерархии объектов, виртуальными надо делать те функции, которые будут вызываться с уровня предка. Сочетание наследования и полиморфизма даёт программисту возможность расширять программу не имея её исходного кода. Это можно сделать, потому что наследование действует после компиляции исходной программы. В терминах С++: если программист располагает объявлением базового класса (header), он может определить производный класс, наследующий все свойства базового. После этого, программист может выборочно переопределить поведение некоторых методов, так как ему нужно. Это возможно даже в том случае, если доступа к исходному коду базового класса нет, и базовый класс скомпилировался раньше, чем производный класс был написан или даже задуман. Поскольку позднее связывание происходит во время выполнения программы, изнутри объектного модуля могут полиморфно вызываться виртуальные функции потомков, которых ещё не существовало во время компиляции.

Предположим, что есть программа, которая поддерживает различные типы фигур. Мы хотим написать процедуру буксировки фигур Drag(), которая перемещала бы любую из фигур созданных сейчас и впоследствии. При процедурном программировании мы бы сделали это через switch(), но при добавлении новой фигуры нужно было бы вносить изменения в программу и изменять её код. Если исходного кода нет, то развивать иерархию объектов мы не сможет. Чтобы функция Drag() двигала ещё и те объекты, которые появятся в будущем, нужно использовать виртуальные методы и таким образом мы получим семейство полиморфных объектов. Методы Hide и Show нового потомка будут вызываться из метода Point::MoveTo, который, быть может, был написан несколько лет назад, а Point::MoveTo будет их знать. Если метод Drag() призван работать с полиморфными объектами, то она будет работать с теми объектами, которые уже есть и с теми, которые появятся в будущем. Функцию Drag() имеет смысл поместить туда, где впервые появилась возможность перемещения объекта – в класс Point.

//ExOOP\_04  
Point\_04.h  
Point\_04.cpp  
ExOOP\_04.cpp  
  
//Point\_04.h  
// К Point\_03.h добавим виртуальные функции. За счёт виртуальных функций Show и Hide метод Point::MoveTo работает не с конкретным классом, а вообще с любой фигурой. Добавим в Point функцию Drag()  
class Location  
{ …  
} // Location  
class Point::publicLocation  
{  
private:  
…  
public:  
…  
void MoveTo(int NewX, int NewY); // переместить фигуру  
void Drag(in tint Step);  
// Виртуальные функции  
virtual void Show();  
virtual void Hide();  
}  
class Circle:public Point  
{  
protected:  
int Radius;  
public:  
Circle(…);  
~Circle();  
virtual void Show();  
void Hide();  
void Expand(…);  
void Reduce(…);  
// void MoveTo(…)  
} // class Circle  
  
// Point\_04.cpp  
// Методы Location  
…  
// Методы Point  
Point(), ~Point(), MoveTo(), Show(), Hide()  
void Point::Drag(int Step) // буксировка фигуры  
{ int FigX, FigY;  
FigX = GetX();  
FigY = GetY();  
// Бесконечный цикл буксировки фигуры  
while(1)  
{  
// конец буксировки  
if (KEY\_DOWN(VK\_ESCAPE)) // Esc, 27  
break; // конец работы  
// направление движения объекта  
if(KEY\_DOWN(VK\_LEFT))  
{  
FigX -= Step;  
MoveTo(FigX, FigY);  
Sleep(500);  
}  
if(KEY\_DOWN(VK\_RIGHT))  
{  
FigX += Step;  
MoveTo(FigX, FigY);  
Sleep(500);  
}  
if(KEY\_DOWN(VK\_UP))  
{  
FigY -= Step;  
MoveTo(FigX, FigY);  
Sleep(500);  
}  
if(KEY\_DOWN(VK\_DOWN))  
{  
FigY+= Step;  
MoveTo(FigX, FigY);  
Sleep(500);  
}  
} // while  
} // drag  
  
//ExOOP\_04.cpp  
#include “Point\_04.h”  
#define KEY\_DOWN(…)  
int main()  
{ …  
// точка  
Point::APoint(x0, y0);  
APoint.Show(); // 1  
cout << “Press 1\n”;  
while(1)  
{  
if(KET\_DOWN(49)) break;  
APoint.Drag(40);  
}  
while(1)  
{  
if(KET\_DOWN(50)) break;  
Circle.ACircle(x0, y0, Radius0);  
ACircle.Show();  
ACircle.MoveTo(x0, y0 + 2\*Radius0, Radius0);  
}  
while(1)  
{  
if(KET\_DOWN(51)) break;  
ACircle.Expand(DeltaRad);  
}  
while(1)  
{  
if(KET\_DOWN(52)) break;  
ACircle.Drag(40);  
}  
while(1)  
{  
if(KET\_DOWN(53)) break;  
ACircle.Reduce(DeltaRad);  
}  
while(1)  
{  
if(KET\_DOWN(54)) break;  
}

**Лекция 5**

Виртуальные функции потомков лучше продолжать объявлять с ключевым словом virtual. Функция MoveTo ушла из класса Circle. Circle больше не нуждается в замене MoveTo класса Point неизменённой копии на своём уровне. Вместо этого, метод MoveTo может быть теперь унаследован от Point, причём все вложенные вызовы метода MoveTo, за счёт виртуальности, переходят к методам класса Circle, а не к методам класса Point, в отличие от полностью статической иерархии объектов. Таким образом, MoveTo берётся из Point, а процедуры Hide и Show, которые вызывает Point, берутся из текущего класса Circle. Функция Drag предназначена для перемещения произвольной фигуры по экрану в ответ на нажатие клавиши. Drag перемещает точку – фигуру своего уровня, а так же круг – фигуру потомка. В какое место поместить Drag – это зависит от выполняемых действий и их универсальности. Перемещение фигуры включает в себя изменение её координаты, как реакцию на нажатие клавиш. С точки зрения наследования, этот метод Drag должен быть расположен рядом с MoveTo, так как любой объект, кода меняет координаты, должен перетаскиваться. Поэтому лучше, чтобы метод Drag принадлежал Point, и тогда все производные от Point классы фигур смогут перемещаться. Надо ли метод Drag делать виртуальным? Если предполагается, что на других уровнях иерархии действия могут изменяться, например, двигать фигуру с помощью мыши, то лучше такой метод делать виртуальным. В нашем случае этого сделано не было. В общем случае, надо все функции делать виртуальными.

Динамический полиморфизм

Указатели на производные классы. Основой виртуальных функций и динамического полиморфизма являются указатели на производные классы, которые работают следующим образом: указатель, объявленный в качестве указателя на базовый класс, также может использоваться как указатель на любой класс, производный от этого базового. В такой ситуации Point \*pPoint – указатель на базовый класс. Point APoint – экземпляр.

Естественно, что указатель pPoint может указывать на объект базового класса:  
pPoint = &APoint; // pPoint указывает на объект базового класса  
…  
pPoint = &ACircle;

Указатель базового класса может указывать на объект любого класса, производного от этого базового и, при этом, ошибка несоответствия типов генерироваться не будет. Для указания на объект производного класса можно воспользоваться указателем базового класса, но при этом доступ может быть обеспечен только к тем методам производного класса, которые были унаследованы от базового. Объясняется это тем, что базовый указатель знает только о базовом классе и ничего не знает о членах, добавленных в производных классах. Указатель базового класса можно использовать для указания на объект производного класса, но обратный порядок не действителен. Указатель производного класса нельзя использовать для доступа к объектам базового класса. Чтобы обойти это ограничение можно использовать приведение типов, но на практике так действовать не рекомендуется. Арифметика указателей связана с типом данных, то есть с классом, который задан при объявлении указателя. Таким образом, если указатель базового класса указывает на объект производного класса, а затем инкрементируется, то он всё равно будет указывать на следующий объект базового класса, а не на следующий объект производного класса.

Пример:  
ExOOP\_11.cpp  
#include <iostream>  
using namespace std;  
class Base  
{  
private:  
int BNum  
public:  
int Get\_BNum()  
{return BNum;}  
void Set\_BNum(int NewBNum)  
{BNum = NewBNum}  
class Next: public Base  
{  
private:  
int NNum;  
public:  
int Get\_NNum()  
{return NNum;}  
void Set\_NNum()  
{NNum = NewNNum;}  
int SumAll()  
{return BNum + NNum;  
return Get\_BNum() + Get\_NNum();  
} // SumAll()  
} // class Next  
  
int main()  
{  
Base \*pBase;  
Base ABase;  
Next ANext;  
cout << “Предок \*pBase = &ABase\n”;  
pBase = &ABase;  
cout << “Адрес экз БК pBase = &ABase” << pBase << endl;  
pBase -> Set\_BNum(11);  
cout << “Поле БК pBase -> Get\_BNum() BNum=” << pBase -> Get\_BNum() << endl;  
…  
cout << “Потомок Base \*pBase=&ANext\n”;  
pbase = &ANext;  
cout << “Адрес экз ПК pBase = &ANext pBase =” << pBase << endl;  
// pBase ->Set\_NNum(177); // нельзя  
// потомок видит предка  
cout << “видит предка\n”;  
cout << “из потомка поменяли поле БК pBase -> Set\_BNum(12)\n”;  
pBox -> Set\_BNum(12);

Так как указатель базового класса pBase не видит нового у потомка – это Get\_NNum, Set\_NNum, так как этого нет в базовом классе, то pBase нельзя использовать для установки NNum. Экземпляр потомка ANext знает всё о потомке.

Объявим указатель на производный класс и получим доступ к методам базового класса.

Через указатель на производный класс доступны все методы иерархии – свои и предка.

Виртуальная функция является членом класса. Она объявляется внутри базового класса и переопределяется в производном классе. Для того, чтобы функция стала виртуальной, перед ней ставится слово virtual. Если класс, содержащий виртуальную функцию, наследуется, то в производном классе переопределяется. По существу виртуальная функция реализует идею – один интерфейс и множество методов, который лежит в основе полиморфизма. Каждое переопределение виртуальной функции в производном классе определяет её реализацию, связанную со спецификой производного класса. Таким образом, переопределение создаёт конкретный метод. При переопределении виртуальной функции в производном классе ключевое слово virtual не требуется, но рекомендуется. Для наглядности, чтобы напоминать о переопределении функции, в заголовок можно добавить ключевое слово override. Виртуальная функция может вызываться, так же как и любая другая функция-член класса.

Рассмотрим вызов виртуальной функции через указатель, благодаря чему поддерживается динамический полиморфизм. В предыдущем примере мы выяснили, что указатель базового класса можно использовать в качестве указателя на объект производного класса. Если указатель базового класса ссылается на объект производного класса, который содержит виртуальную функцию и для которого виртуальная функция вызывается через указатель базового класса, то компилятор определяет, какую версию виртуальной функции базового или производного класса вызвать, основываясь на типе объекта, на который ссылается указатель. При этом, определение конкретной версии виртуальной функции имеет место не в процессе компиляции, а в процессе выполнения программы. Тип объекта, на который ссылается указатель и определяет ту версию виртуальной функции, которая будет выполняться, поэтому, если два и более различных классов являются производными от базового, содержащего виртуальную функцию, и если указатель базового класса ссылается на разные производные базовых классов, то выполняются различные версии виртуальных функций. Этот процесс является реализацией принципа динамического полиморфизма. Управление объектами осуществляется через указатель на базовый класс. Необходимость в динамическом полиморфизме появляется, когда имеется много схожих объектов и с ними надо работать.

Пример:  
ExOOP\_12.cpp

**Лекция 6**

Пример: использование динамического полиморфизма для иерархии из четырёх классов.  
Point\_04.h  
Point\_04.cpp  
Point\_05.h  
Point\_05.cpp  
ExOOP\_07.cpp  
  
// Point\_05.h  
#include “Point\_04.h”  
class Ring: public Circle  
{  
protected:  
int RingW;  
public:  
Ring(int InitX, int InitY, int InitRad, int InitW);  
virtual ~Ring();  
virtual void Hide();  
virtual void Show();  
void Wide(int DeltaWide);  
void Slim(int DeltaWide);  
}; // class Ring  
  
// Point\_05.cpp  
#include “Point\_04.h”  
#include “Point\_05.h”  
externHDC hdc;  
KEY …  
// реализация методов класса Ring  
Ring::Ring (int InitX, int InitY, int InitRad, int InitW): Circle(InitX, InitY, InitRad)  
{  
RingW = InitW;  
} // Ring()  
Ring::~Ring(){}  
void Ring::Show()  
{ visible = true;  
HPEN Pen = CreatePen(PS\_SOLID, RingW, RGB(255, 0, 0));  
}  
SelectObject(hdc, Pen);  
Ellipse(hdc, X-Radius, Y-Radius, X+Radius, Y+Radius);  
DeleteObject(Pen);  
}  
void Ring::Hide()  
{ visible = true;  
HPEN Pen = CreatePen(PS\_SOLID, RingW, RGB(0, 0, 255));  
SelectObject(hdc, Pen);  
Ellipse(hdc, X-Radius, Y-Radius, X+Radius, Y+Radius);  
DeleteObject(Pen);  
} // Hide()  
void Ring::Wide(int DeltaWide)  
{ while(1)// цикл опроса клавиатуры  
{ if (KEY\_DOWN(VK\_ESCAPE)) // 27  
break;  
if(KEY\_DOWN(187)); // +  
{ Hide();  
RingW += DeltaWide;  
Show();  
Sleep(500);  
}}}  
void Ring::Slim(int DeltaWide)  
{while(1)  
if (KEY\_DOWN(VK\_ESCAPE)) break;  
if(KEY\_DOWN(189))  
{ Hide();  
RingW -= DeltaWide;  
if (RingW < 5) RingW = 5;  
Show();  
Sleep(500);  
}}}  
  
ExOOP\_07.cpp  
#include “Point\_05.h”  
#define KEY\_DOWN  
void PressKey(int vkCode); // задержка при нажатии клавиши с кодом vkCode  
int main()  
{ // объявление и инициализация переменных  
int X0 =, Y0 =, Radius0 =, DeltaRad =, RingW = 30, DeltaWide = 10;  
// точка  
Point xpPoint;  
Point APoint(X0, Y0);  
// инициализируем указатель на базовый класс адресом экземпляра базового класса  
pPoint = &APoint;  
pPoint ->Show(); // показать фигуру - точка  
pPoint ->MoveTo(X0, Y0 +2\*Radius0) // переместить фигуру – точка  
PressKey(32); // пробел  
// круг  
Circle ACircle(x0 + 3\*Radius0, Y0, Radius0);  
// инициализируем указатель на базовый класс адресом экземпляра потомка  
pPoint = &ACircle;  
pPoint ->Show(); // показать фигуру – круг  
PressKey(49); // 1  
ACircle.Expand(DeltaRad); // увеличить радиус круга  
pPoint ->Drag(40); // двигать фигуру – круг  
PressKey(50); // 2  
ACircle.Reduce(DeltaRad); // уменьшить радиус круга  
PressKey(51);  
// кольцо  
Ring ARing (2\*X0, Y0, Radius0, RingW0);  
pPoint = &ARing;  
pPouint -> Show(); // показать фигуру – кольцо  
pPoint –> Drag(40); // двигать фигуру – кольцо  
PressKey(52); // 4  
pPoint ->Hide(); // скрыть фигуру – кольцо  
ARing.Expand(3\*DeltaRad); // большое кольцо  
// двигаем фигуру – большое кольцо  
pPoint -> Drag(80); // двигаем фигуру – кольцо  
PressKey(54); // 6  
ARing.Reduce(DeltaRad);  
// увеличить радиус кольца  
ARing.Wide(10);  
PressKey(55); // 7  
Aring.Drag(80);  
pPoint -> Drag(100); // передвинуть кольцо  
PressKey(56); // 8  
// уменьшение толщины кольца  
ARing. Slim(10);  
PressKey(57); // 9

Для каждого полиморфного типа данных компилятор создаёт таблицу виртуальных функций и встраивает в каждый объект такого класса скрытый указатель на эту таблицу. Она содержит адреса виртуальных функций соответствующего объекта. Компилятор автоматически встраивает в начало конструктора фрагмент кода, который инициализирует указатель на таблицу виртуальных функций. Если вызывается виртуальная функция, код, сгенерированный компилятором, находит указатель на таблицу виртуальных функций, затем просматривает эту таблицу и извлекает из неё адрес соответствующей функции. После этого происходит переход на указанный адрес и вызов функции. Переопределение виртуальной функции внутри производного класса может оказаться похожим на перегрузку функций, однако эти два процесса совершенно различны. Во первых, перегружаемая функция должна отличаться типом или числом параметров в сигнатуре, а переопределяемая функция должна иметь точно такой же тип параметров, то же их число, совпадая с прототипом. На самом деле, если при переопределении виртуальной функции изменить число или тип параметров. Она становится просто перегружаемой функцией и её виртуальная природа теряется. Во вторых, виртуальная функция должна быть членом класса, это не относится к перегружаемым функциям. Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными, для описания переопределения виртуальной функции используется термин подмена – overriding.

Виртуальные деструкторы

В том случае, когда ожидается передача в функцию указателя на объект базового класса, часто передают указатель на объект производного класса. Конструкторы не могут быть виртуальными, однако деструкторы могут быть виртуальными и часто ими являются. В том случае, когда ожидается указатель на объект базового класса, вполне допустимо и часто используется передача указателя на объект производного класса. При удалении объекта производного класса, на который ссылается указатель базового класса, если деструктор объявлен как виртуальный, то сначала будет вызван деструктор соответствующего производного класса. Затем, деструктор производного класса вызовет деструктор базового класса и указанный объект будет удалён целиком. В противном случае, будет вызван только деструктор только базового класса и произойдёт утечка памяти. Поэтому, если в классе объявлены виртуальные функции, то и деструктор должен быть объявлен виртуальным.

Интерфейсы и абстрактные базовые классы

Абстрактный метод (чистая виртуальная функция)

Иногда, при создании базового класса, приходится объявлять функцию, конкретную реализацию которой указать не можем. Просто известно, что эта функция в базовом классе должна быть, например класс фигуры и метод Show(). Что будет в методе Show() сказать нельзя, потому что пока неизвестно, о какой фигуре идёт речь. Известно только, что у каждой фигуры иерархии должна быть своя функция Show(). Функция в базовом классе объявляется как виртуальная, но тело её не определяется, потому что нет никакой конкретики о фигуре. В таких случаях базовый класс содержит набор полей и объявлений методов, часть которых определяется позже только в производном классе. Когда в виртуальной функции базового класса отсутствует реализация, в любом классе, производном от этого базового (прямом потомке), такая функция обязательно должна быть переопределена. Для этого в С++ предусмотрены так называемые чистые виртуальные функции. Чистая виртуальная функция ничего не делает и не доступна для вызова. Она служит только основой для подменяющих (override) её функций производных классов. Чистые виртуальные функции не определяются в базовом классе. Он содержит только прототипы этих функций. Для определения чистой виртуальной функции используются формы:  
virtual ТипФ ИмяФ(список параметров) = 0;

**Лекция 7**

virtual ТипФ ИмяФ(список параметров) = 0, принципиальным здесь является приравнивание функции нулю. Это означает, что в базовом классе не существует тела функции. Если функция объявляется как чисто виртуальная, то это гарантирует, что в каждом производном классе будет реализация этой функции. Поскольку потомки обязательно будут иметь свою реализацию, унаследованной чистой виртуальной функцией, то получаем один интерфейс и множество реализаций, то есть полиморфизм. Чистые виртуальные функции обеспечивают принудительный полиморфизм.

Абстрактный базовый класс.

Когда мы конструируем класс, то предполагаем создавать экземпляры этого класса. В общем случае, возможность непосредственно создавать экземпляры базового класса является недостатком и плохим стилем программирования. Не всегда можно создать базовый класс, в котором все методы известны и реализованы. Единственной целью базового класса является определение общих полей и методов для всех его потомков. Изначально не предполагается, что кто-то будет непосредственно создавать экземпляры класса, поскольку базовый класс является слишком общим. Он определяет только состав – общие поля и методы для всей иерархии классов. Например, у плоских фигур должны быть формулы расчёта площади и периметра, но ничего конкретного про формулы расчётов сказать нельзя, пока не станет точно известно, о какой фигуре идёт речь. В дальнейшем, потомки расширят функционал базового класса. Например, из абстрактного класса ABC\_Figure можно получить классы Rect, Ring и т.д. со своими формулами расчёта площади и периметра. Для трёхмерных фигур, кроме функций площади и периметра добавится функция вычисления объёма, но как она будет выглядеть – неизвестно. Известно только то, что эта функция должна быть. В некоторых случаях мы не можем сказать, что должно быть в реализации метода, но мы точно знаем, что метод должен быть у потомка и иметь свою реализацию. Иногда требуется создать наследуемый (базовый) класс, в котором определены лишь некоторые характеристики методов: имя метода, тип результата и список параметров (прототип). Тела этих методов пустые, поскольку трудно предвидеть, какие переменные и операторы понадобятся в объектах в наследующих классах. Поэтому в наследующем классе программист должен реализовать эти методы, то есть переменные и тело метода, необходимые для объекта этого класса. То есть, в базовом классе определяется лишь общее предназначение методов, которые должны быть реализованы в последующих классах, но сам по себе этот класс не реализует один или несколько подобных методов. Такие объявленные методы без реализации называются абстрактными или чистыми виртуальными. Чтобы класс стал абстрактным, в нём должна быть хотя бы одна чистая виртуальная функция. В базовом классе метод можно определить лишь частично, указав только его прототип. В базовом классе метод можно определить лишь частично, указав только его прототип, поэтому для более эффективного использования наследования, в базовом классе можно определить абстрактные методы, у которых отсутствует тело, или написать пустое тело. Поскольку в абстрактном классе имеется хотя бы одна функция, в которой отсутствует тело, такой класс не определён полностью и ни одного объекта этого класса создать невозможно. Поэтому абстрактные методы могут быть только наследуемыми. Когда базовым является абстрактный класс, то только прямой потомок должен реализовать все абстрактные методы предка. Если от абстрактного базового класса имеется производный класс с принудительно реализованными чистыми виртуальными функциями и если этот класс является базовым для другого производного класса, то виртуальная функция, при необходимости, может подменяться в последнем производном классе, а может и не подменяться, то есть здесь появляется свобода выбора. Тогда будет вызываться реализация виртуальной функции полноценного базового класса. Если класс является абстрактным базовым классом, он может определять любое число абстрактных методов. Абстрактный базовый класс может быть в середине иерархии классов. Механизм абстрактных классов разработан для задания единой структуры иерархии, содержащей методы, которые в дальнейшем должны быть конкретизированы в классах-наследниках. В некоторых случаях абстрактные классы составляют несколько верхних уровней иерархии. Например, класс ABC\_ФИГУРА может стать ABC\_ФИГУРА\_ПЛОСКАЯ или ABC\_ФИГУРА\_ОБЪЁМНАЯ. Для динамического полиморфизма можно использовать указатели на абстрактный базовый класс.

Пример: чистые виртуальные функции, абстрактный базовый класс, полиморфизм.

//OOPEx08\_01.cpp  
// Абстрактный базовый класс  
class ABC\_Figure  
{  
protected:  
double mDim1;  
double mDim2;  
public:  
void GetDimention(double &xd1, double &xd2)  
{  
xd1 = mDim1;  
xd2 = mDim2;  
} // GetDimention()  
virtual double CalcPerim() = 0;  
virtual double CalcSquare() = 0;  
} // ABC\_Figure  
  
// Прямоугольник  
class Rect:public ABC\_Figure  
{  
public:  
Rect(double InitSide1, double InitSide2)  
{  
mDim1 = InitSide1;  
mDim2 = InitSide2;  
} // Rect()  
  
// Принудительная реализация чистой виртуальной функции  
double CalcPerim() // рассчитать периметр прямоугольника  
{  
double d1, d2;  
GetDimention(d1, d2);  
return 2\*(d1+d2);  
} // CalcPerim();  
  
// Принудительная реализация чистой виртуальной функции  
double CalcSquare()  
{  
double d1, d2;  
GetDimention(d1, d2);  
return d1\*d2;  
} // CalcSquare();  
  
// Прямоугольный треугольник  
class RightTriangle:public ABC\_Figure  
{  
public  
RightTriangle(double InitLeg1, double InitLeg2)  
{  
mDim = InitLeg1; mDim2 = InitLeg2;  
} // RightTriangle();  
  
// Принудительная реализация чистой виртуальной функции  
double CalcPerim()  
{  
double Leg1, Leg2, Hypot;  
double Perim;  
GetDimention(Leg1, Leg2);  
Hypot = sqrt(Leg1\*Leg1 + Leg2\*Leg2);  
Perim = Leg1 + Leg2 + Hypot;  
return Perim;  
} // CalcPerim()  
  
// Принудительная реализация чистой виртуальной функции  
double CalcSquare()  
{  
double Leg1, Leg2;  
GetDimention(Leg1, Leg2);  
return (Leg1\*Leg2)/2;  
} // CalcSquare();  
  
int main()  
{  
// Прямоугольник  
cout << “Прямоугольник\n”;  
Rect ARect(4,5);  
cout << “Периметр = ” << ARect.CalcPerim() << endl;  
cout << “Площадь = ” << ARect.CalcSquare() << endl;  
// Прямоугольный треугольник  
cout << “ Прямоугольный треугольник \n”;  
RightTrianle.ARightTriangle(3,4);  
cout << “Периметр = ” << ARightTriangle.CalcPerim() << endl;  
cout << “Площадь = ” << ARightTriangle.CalcSquare() << endl;

Абстрактные (чистые виртуальные) методы и виртуальные методы.

Абстрактный или чистый виртуальный метод может быть описан только в абстрактном классе. Абстрактный метод – это метод класса, который не имеет собственной реализации. Тело его является пустым, поэтому его необходимо будет определить в наследнике, иначе будет ошибка компиляции. Абстрактные методы могут использоваться тогда, когда требуется определить метод без реализации, заданный по умолчанию. Реализация абстрактных методов принудительно происходит в наследнике. То есть происходит переопределение абстрактного метода в классе-наследнике. В абстрактных класса производным классам придётся принудительно использовать полиморфизм, поскольку они обязаны создавать свои реализации абстрактных методов. Экземпляр абстрактного класса создать нельзя, потому что нет реализации чисто виртуальных методов. Другой тип методов – виртуальные. Виртуальный метод имеет тело при определении, а чистый – нет.

Виртуальный метод – это метод класса, который может быть переопределён в классе-наследнике, а может и не быть. Класс-наследник может переопределить базовый функционал метода, дополнив его или определив заново, не меняя своей сигнатуры. Процесс определения виртуального метода происходит внутри наследуемого класса, при котором частично или полностью изменяется тело метода. Переопределять виртуальный метод не обязательно.

Интерфейс

В ООП иногда возникает необходимость описать состав функционала класса без конкретной реализации методов. Действия, выполняемые классом, объявлены, а способ выполнения этих действий не указан. Например, как это происходит в чистом виртуальном методе. Абстрактный тип метода определяет сигнатуру метода, но не поддерживает реализацию. В этом случае, наследующий класс должен иметь свою собственную реализацию для каждого абстрактного метода. Таким образом. Абстрактный метод определяет интерфейс метода, а не его реализацию.

Интерфейс состоит только из чистых виртуальных функций, не имеет полей, так как не понятно, какой метод будет ими пользоваться. Сам по себе интерфейс не может реализовывать методы, он является конструкцией, которая описывает набор поддерживаемых классом методов, не указывая при этом способ реализации. Интерфейс – это набор методов, которые принудительно реализуются с помощью класса-наследника. Синтаксис интерфейсов подобен синтаксису абстрактных классов – интерфейсный метод не имеет тела. Интерфейс определяет, что нужно делать, но не показывает как именно нужно это делать.

class IDom  
{  
Фундамент();  
Стена1();  
Стена\_с\_окном();  
Крыша();  
}  
class Дом1:public IDom  
{  
…  
}

Один класс может наследоваться от нескольких интерфейсов. При наследовании интерфейса, каждый класс (прямой потомок) обязан реализовать тело каждого интерфейсного метода, причём каждый класс определяет свою реализацию этого интерфейсного метода. Например, два класса могут реализовывать интерфейс различными способами, но набор методов в каждом классе будет одинаковым. Базовый класс, содержащий только абстрактные методы, становится интерфейсом. Если все методы имеют реализацию, то это обычный базовый класс. Если связанные действия реализуются через общий интерфейс, программа становится легче для понимания и сопровождения.

//OOPEx\_21.cpp  
//Интерфейс и веерное наследование, чистые виртуальные функции, полиморфизм и динамический полиморфизм  
// Объявление интерфейса  
class IFigure  
{  
public:  
virtual void Show() = 0;  
virtual void Hide () = 0;  
} // IFigure  
// Точка  
class point:public IFigure  
protected:  
bool Visible;  
public:  
void Show(); // принудительная реализация интерфейсного метода  
{  
cout << “/Point Show()” << “Я маленькая точка/n”;  
} // Show()  
void Hide();  
{  
cout << “/Point Hide()” << “Маленькая точка скрылась/n”;  
} // Hide()  
}  
  
class Circle:public IFigure  
{  
private:  
int Radius;  
public:  
void Show();  
{  
cout << “/Circle Show(): Я большой круг/n” ;  
}  
void Hide()  
{  
cout << “/Circle Hide(): Большой круг исчез/n”;  
}  
void Expand(int DeltaRad)  
{  
<пусто>  
} // Expand()  
void Reduce(int DeltaRad)  
{  
<пусто>  
} // Reduce()  
int main()  
{  
// Точка  
cout << “Фигура точка\n”;  
Point APoint;  
APoint.Show();  
APoint.Hide();  
// Круг  
Circle ACircle;  
ACircle.Show();  
ACircle.Hide():  
// динамический полиморфизм через интерфейс  
cout << “IFigure \*pIFig\n”;  
IFigure \*pIFig;  
// Точка  
pIFig = &APoint;  
// через указатель на базовый класс можно вызывать виртуальные функции производно класса  
pIFig->Show();  
pIFig->Hide();  
// круг  
pIFig = &ACircle;  
// через указатель на базовый класс можно вызывать виртуальные функции базового класса  
pIFig->Show();  
pIFig->Hide();  
} // main()

По своей сути абстрактный класс подобен интерфейсу, однако между ними существуем минимум два различия. Во-первых, абстрактный класс может содержать в себе и конструкторы и поля и тела методов, в то время, как интерфейс лишь представляет набор прототипов, приравненных к нулю. Во-вторых, абстрактный класс может быть как базовым для иерархии, так и наследоваться от интерфейса. Интерфейс от абстрактного класса наследоваться не может.

Один из принципов ООП гласит: при создании системы классов, надо программировать на уровне интерфейсов, а не их конкретных реализаций. Под интерфейсом понимается определение функционала без конкретной реализации, то есть под данное определение попадают как собственные интерфейсы, так и абстрактные базовые классы, которые имеют только абстрактные методы без конкретной реализации. В этом плане у абстрактных базовых классов и интерфейсов есть много общего, нередко, при проектировании программ, мы можем заменять абстрактные базовые классы на интерфейсы и наоборот.

Когда следует использовать абстрактные классы:

1. Если надо определить общий функционал для родственных объектов.
2. Если мы проектируем довольно большую функциональную единицу, которая содержит много базовых функционалов.
3. Если нужно, чтобы все производные классы на всех уровнях наследования имели некую общую реализацию.

При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.

Пример: допустим есть система транспортных средств: легковой автомобиль, трамвай, поезд и т.д. Поскольку данные объекты являются родственными, то мы можем использовать абстрактные классы.

**Лекция 8**

Если нам нужно определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой, то в этих случаях используется полиморфизм. Предположим, что наша система транспорта не ограничивается машиной, автобусом и трамваем. Например, мы можем добавить самолёты и лодки, также мы можем добавить лошадь – это животное, которое также выполняет роль транспортного средства, ещё можем добавить дирижабль, ракету, подводную лодку. Получается довольно широкий круг объектов, который связан только тем, что объекты являются транспортным средством и должны реализовывать метод Move, выполняющий перемещение. Так как объекты мало связаны между собой, то для определения общего для всех них функционала лучше определить интерфейс, тем более, что некоторые из этих объектов могут существовать в рамках параллельных систем классификации, например, лошадь может быть классом в структуре системы классов животного мира.

Пример:  
//класс  
class IVehicle // транспортное средство  
{ public:  
// умение двигаться есть, а реализация нас не интересует  
virtual void Move() = 0;  
} // Vehicle  
class Car: public IVehicle  
{ public:  
void GeneralProperty()  
{  
cout << “Общее свойство иерархии – движение на колёсах \n”;  
} // GeneralProperty()  
void Move() // принудительная реализация интерфейсного метода  
{  
cout << “/Car: Машина едет\n”;  
} // Move()  
class Bus: public Car  
{ private:  
int NPass; // число пассажиров  
public:  
Bus(int InitNPass)  
{ NPass=InitNPass;  
}// Bus()  
void BusInfo()  
{  
cout << “Количество пассажиров =” << endl;  
} // BusInfo()  
void Move()  
{  
cout << “/Bus: Автобус едет \n”;  
} / Move()  
} // class Bus  
  
// самолёт  
class Plane: public IVehicle  
{  
void GeneralProperty()  
{  
cout << “Общее свойство иерархии = Летать по воздуху”  
} // GeneralProperty()  
// принудительная реализация интерфейсного метода  
void Move()  
{  
cout << “/Plane Самолёт летит\n”;  
} // Move()  
} // class Plane  
  
int main()  
{  
Car ACar = Car();  
Bus ABus = Bus(20);  
Plane APlane = Plane();  
ACar.GeneralProperty();  
ACar.Move(); cout <<endl;  
ABus.GeneralProperty();  
ABus.Move(); cout << endl;  
APlane.GeneralProperty();  
APlane.Move(); cout << endl;  
  
IVehicle\*pIVehicle = 0;  
pIVehicle = &ACar;  
pIVehicle -> Move();  
cout << endl;  
pIVehicle = &ABus;  
pIVehicle -> Move();  
pIVehicle = &Plane;  
pIVehicle -> Move();

Если классы относятся к единой системе классификации, то выбирается абстрактный класс, иначе выбирается интерфейс. Говоря об использовании абстрактных классов и интерфейсов, можно ещё привести такую аналогию, как состояние и действие. Как правило, абстрактные классы фокусируются на общем состоянии классов наследников, в то время, как интерфейсы строятся вокруг какого-либо общего действия. Например, солнце, костёр, батарея отопления, электронагреватель выполняют функцию (общее действие) – выделение тепла, а это интерфейс. Выделение тепла – их единственная схожая функция. Можно ли создать для рыб, животных и насекомых общий класс? Это возможно, но это не оптимально, тем более, что там могут быть родственные сущности, которые мы захотим использовать. Для каждой сущности мы можем определить свою систему классификации. Например, в одной системе класса, которая наследуется от общего абстрактного класса, были бы звёзды, планеты, астероиды и т.д., то есть все те объекты, которые могут иметь какое-то схожее с солнцем состояние. В рамках другой системы классов мы могли бы определить эдлетроприборы. И так далее, для каждой разноплановой сущности можно было бы составить свою систему классов, исходящую от одного абстрактного класса. А для общего действия – нагрева можно сделать интерфейс IHeatable Heat(), и реализовать его везде. Таким образом, если разноплановые классы обладают каким-то общим действием, то это действие лучше выносить в интерфейс, а для одноплановых классов, которые имеют общее состояние, лучше определять абстрактный класс.

Перегрузка операторов

Это реализация своего собственного функционала этого оператора для конкретного класса. Перегрузка операций представляет собой пример полиморфизма в C++ и перегрузка операций позволяет наделять операции в C++ многозначностью. С++ позволяет распространить перегрузку операций на типы, определяемые пользователем. Компилятор воспользуется числом и типом операндов, чтобы определить, какое определение операций применить. Для перегрузки операторов используются функции специальной формы, которые называются операторные:  
operator <op> (Arg List)

Задача: если турист в течении какого-то времени шёл пешком, ехал на велосипеде, отдыхал, то как узнать, сколько времени занял весь переход. В данном случае, операция сложения имеет смысл, однако величины, которые мы складываем, не является встроенными типом. Чтобы сложить время – часы и минуты, надо перегрузить операцию сложения.  
  
struct Time  
{ int Hours;  
int Minutes;  
} // Time  
// перегрузка оператора сложения для типа time  
Time operator + (Time a, Time b)  
{  
Time SumTime;  
SumTime.Minutes = a.Minutes + b.Minutes;  
SumTime.Hours = a.Hours + b.Hours + SumTime.Minutes/60;  
SumTime.Minutes = SumTime.Minutes%60;  
return SumTime;  
} // oper +  
  
int main()  
{  
Time ATime = {4, 40};  
Time BTime = {3, 55};  
Time CTime;  
cout << “Исходные данные\n”;  
cout << “ATime: Hours = ” << ATime.Hours << “ATime: Minutes = ” << ATime.Minutes << endl;  
cout << “BTime: Hours = ” << BTime.Hours << “BTime: Minutes = ” << BTime.Minutes << endl;  
// операторная форма вызова  
CTime = ATime + BTime;  
cout << “Операторная форма вызова CTime = ATime + BTime\n”;  
cout << “CTime: Hours = ” << CTime.Hours << “CTime: Minutes = ” << CTime.Minutes << endl;  
// функциональная форма вызова  
CTime = operator + (ATime, BTime);  
cout << “Функциональная форма вызова CTime = ATime + BTime\n”;  
cout << “CTime: Hours = ” << CTime.Hours << “CTime: Minutes = ” << CTime.Minutes << endl;

Перегрузка функций

Это один из типов полиморфизма в C++. В С++ несколько функций могут иметь одно и то же имя. Перегрузка функций – это объявление функций с одним и тем же именем несколько раз. Таким образом, в некоторой области видимости имя перегруженной функции объявляется несколько раз. Чтобы компилятор мог отличать перегруженные функции между собой, эти функции должны отличаться списком входных параметров. В С++ две или более функций могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим. Если две и более функции имеют одинаковое имя, то говорят, что они перегружены. Чтобы перегрузить функцию, необходимо объявить и определить все требуемые варианты.

В общем случае, объявление перегруженной функции выглядит следующим образом:   
RetType1 FunName (ParamList1) {…}  
RetType2 FunName (ParamList2) {…}

Классический пример перегрузки функций даёт набор библиотечных функций abs(), fabs(), labs(), которые возвращают абсолютные значения для int, float и double. Однако, из-за того, что для трёх типов данных требуется три функции. Во всех трёх случаях возвращается абсолютное значение числа, отличие только в типе данных. В то же время, на С++ можно исправить эту ситуацию, путём перегрузки одно имени для трёх типов данных. Перегруженные функции отличаются по списку параметров. Списки параметров перегруженных функций должны отличаться по следующим признакам:

1. Количество параметров.
2. Если количество параметров одинаковое, то по типу параметров.

Перегружать можно только функции, отличающиеся сигнатурой, то есть аргументы должны отличаться либо типом, либо количеством. Перегружать функции, которые отличаются только типом возвращаемого значения нельзя.

Могут ли считаться перегруженными функции, которые имеют одинаковые имена, одинаковое количество и типы параметров, но которые возвращают значения разных типов? Ответ – нет, потому что компилятор распознаёт перегруженные функции по параметрам. Если функции имеют одинаковые имена, одинаковое количество параметров, но возвращают разные значения, то такие функции считаются одинаковыми, и в этом случае компилятор выдаст ошибку. Перегруженные функции дают возможность упростить программу, допуская обращение к одному имени для выполнения близких по смыслу действий над различными типами данных. Чтобы перегрузить функцию нужно сначала объявить прототип, а потом – определить все требуемые варианты её вызова. Компилятор автоматически выберет правильный вызов на основании числа и/или типа используемого аргумента.

Пример: перегрузка функции поиска максимума двух чисел для разных типов аргументов:  
// прототипы  
int Max(int, int);  
float Max(float, float);  
double Max (double, double);  
int main()  
{  
int iA = 10, iB = 20;  
float f1 = 1,4142f, f2 = 1,712f;  
double dE = 2,71828, dPi = 3,1457;  
cout << “INT iA = ” << iA << “\t iB” << iB << “\t MaxInt = ” << Max(iA, iB) << endl;  
cout << “FLOAT f1 = ” << f1 << “\t f2” << f2 << “\t MaxFloat = ” << Max(f1, f2) << endl;  
cout << “DOUBLE …  
return 0;  
} // main  
// реализация функций  
int Max (int x, int y)  
{  
if (x > y) return x;  
return y;  
} // max()  
float Max (float x, float y)  
{  
if (x > y) return x;  
return y;  
} // max()  
double Max (double x, double y)  
{  
if (x > y) return x;  
return y;  
} // max()

Иметь одинаковое имя для функций, выполняющих схожие действия удобно, так как имя говорит о характере выполняемой функциями работы. Необходимо помнить только одно имя – то, которое описывает общие действия. В данном случае, благодаря использованию полиморфизма, из трёх имён iMax, fMax и dMax получилось одно имя – Max. Использование перегрузки упрощает код, предоставляя программисту возможность пользоваться одной функцией, которой передаются аргументы разных типов.

Шаблоны функций

Шаблоны функций представляют собой обобщённое описание функций. Шаблоны определяют функцию на основе обобщённого типа, вместо которого может быть подставлен конкретный тип данных. Программирование на основе шаблонов называют обобщённым программированием, а шаблоны функций, в которых типы представлены параметрами, называют параметризованными типами. Шаблонами удобно пользоваться, когда семейство перегружаемых функций имеет одинаковый код, применяемый к различным типам данных.

Рассмотрим функцию Swap(), которая осуществляет перестановку значений int, double и struct:  
struct Point  
{ int X;  
int Y;  
} // Struct

Можно создать перегружаемы функции Swap, сдублировав программный код и поменяв тип int на double и struct. Это решение потребует времени и имеется вероятность допустить ошибку. Средства использования шаблонов в С++ автоматизируют этот процесс, обеспечивая высокую надёжность и экономию времени. Шаблоны функций предоставляют возможность давать определение функций на основе некоторого произвольного типа данных.

template <typename xType>  
void Swap(xType &a, xType &b)  
{ xType &Temp;  
Temp = a;  
a = b;  
b = Temp;  
} // swap()   
// прототип шаблона функций  
template <typename xType>   
void Swap (xType &a, xType &b);  
int main()  
{ // 1  
int N = 10; M = 20;  
cout << “Before intSwap: N = ” << N << “M = “ << M << endl;  
Swap(N, M);  
cout << “After intSwap: N = ” << N << “M = “ << M << endl;  
// 2  
double E = 2.71828, Pi = 3.145;  
cout << “Before doubleSwap: E = ” << E << “Pi = “ << Pi << endl;  
Swap(E, Pi);  
cout << “After doubleSwap: E = ” << E << “Pi = “ << Pi << endl;  
// 3  
Point A = {11, 111};  
Point B = {22, 222};  
cout << “Before PointSwap: A = ( ” << A.x << “ , “ << A.y << “ )\t B.x = ( ” << B.x << “ , “ << B.y << “ )” << endl;  
Swap(A, B);  
cout << “After PointSwap: A = ( ” << A.x << “ , “ << A.y << “ )\t B.x = ( ” << B.x << “ , “ << B.y << “ )” << endl;

Первая строка указывает, что устанавливается шаблон и произвольному типу присваивается имя xType, которое выбирается программистом. В описание обязательно входят ключевые слова template и typename, а также угловые скобки. Шаблон не создаёт никаких функций (если создавать, то сколько вариантов?), вместо этого он представляет компилятору указания, относительно того, как определить функцию для конкретного типа, соответствующего образцу шаблона. По типу передаваемых аргументов компилятор создаст соответствующую функцию.

Компилятор генерирует и использует три версии функции Swap для типов int, double и struct соответственно. При использовании шаблонов, объём программы не уменьшается. Окончательный код программы не содержит никаких шаблонов, только три реальные функции, сгенерированные для программы. Преимущество шаблонов заключается в том, что они упрощают создание множественных определений функций и повышают надёжность этого процесса.