|  |
| --- |
| **ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ** |
|  |
|  |
|  |
| **Практическое занятие №6.**  **Тема: Перегрузка операторов и функций** |
|  |
|  |
|  |

**Цель:** получение практических навыков перегрузки операторов и разработки перегруженных и виртуальных функций.

**Основные теоретические положения**

**Перегрузка функций**

Если две или более функций имеют одинаковое имя, то говорят, что они перегружены. В этом случае функции должны отличаться типом и (или) числом своих аргументов. Перегрузить функцию не сложно – просто следует объявить и определить все требуемые варианты. На компилятор возлагается задача выбора соответствующей конкретной версии вызываемой функции (а значит и метода обработки данных).

Пример перегрузки функции.

**#include <iostream>**

**using namespace std;**

**void date (char\* date) {cout <<date<< “\n”;}**

**void date (int month, int day, int year)**

**{cout <<day<< “/” <<month<<“/” <<year<<“\n”;}**

**int main ( )**

**{**

**date (“11/12/2020”);**

**date (11, 12, 2020);**

**return 0;**

**}**

В данном примере функция ***date()*** перегружается для получения даты либо в виде строки, либо в виде трех целых чисел. Данный пример показывает, что перегрузка функции может производиться и вне объектно-ориентированного подхода.

**Виртуальные функции**

Виртуальные функции являются инструментом для поддержки динамического полиморфизма. Виртуальная функция — это функция, которая объявляется в базовом классе с использованием ключевого слова **virtual** и переопределяется в одном или нескольких производных классах. При переопределении виртуальной функции в производном классе ключевое слово **virtual** повторять не нужно (хотя это не будет ошибкой).

Основой для использования виртуальных функций выступают указатели на производные классы.

***Указатели на производные классы***

Указатель, объявленный как указатель на базовый класс, может использоваться как указатель на любой класс, производный от этого базового. Следующие инструкции являются правильными:

**base \*p;** *//*указатель базового класса

**derived ob;** *//*указатель производного класса

**p=&ob;** *//*указатель *р* для объекта производного класса.

При использовании указателя базового класса в производном, доступ может быть обеспечен только к тем членам производного класса, которые были унаследованы от базового. Пример.

**#include <iostream>**

**using namespace std;**

**class base**

**{int x;**

**public:**

**void set\_x(int i) {x=i;}**

**int get\_x() {return x;}**

**};**

**class derived: public base**

**{int y;**

**public:**

**void set\_y(int i) {y=i;}**

**int get\_y() {return y;}**

**};**

**int main()**

**{base \*p;**

**base base\_ob;**

**derived der\_ob; p=&base\_ob;**

**p→set\_x(10);**

**cout <<"x of base=" <<p→get\_x() <<"\n";**

**p=&der\_ob;**

**p→set\_x(99);**

**der\_ob.set\_y(88);**

**cout <<"x of derived="<<p→get\_x() <<"\n";**

**cout <<"y of derived="<<der\_ob.get\_y() <<"\n";**

**return 0;**

**}**

В данном примере спроектированы базовый и производный от него классы. Объявляется указатель ***p*** базового класса и устанавливается на объект базового класса. Через него производится запись и чтение переменной ***х***. Затем указатель ставится на объект производного класса и через него осуществляется доступ к переменной ***х***этого объекта. К переменной ***y*** так обратиться нельзя, так как она является особенной (ненаследуемой) переменной производного класса. К ней обращаться можно только через имя объекта производного класса.

***Особенности виртуальных функций***

Виртуальная функция является членом класса. Она объявляется внутри базового класса и **переопределяется** в производном. Динамический полиморфизм поддерживается только в том случае, когда вызов виртуальной функции **производного класса** производится через **указатель базового**. Какая версия виртуальной функции будет вызвана, определяется **типом объекта,** на который ссылается указатель.

**#include <iostream>**

**using namespace std;**

**class base**

**{public:**

**int i;**

**base (int x) {i=x;}**

**virtual void func(){cout <<”virtual of base ” <<i <<’\n’;}**

**};**

**class derived1: public base**

**{public:**

**derived1(int x): base(x) {}**

**void func() {cout <<”virtual of derived1 ” <<i+i <<’\n’;}**

**};**

**class derived2: public base**

**{public:**

**derived2(int x): base(x) {}**

**void func() {cout <<”virtual of derived2” <<i\*i <<’\n’;}**

**};**

**int main()**

**{**

**base\*p;**

**base ob(10);**

**derived1 d\_ob1(10), derived2 d\_ob2(10);**

**p=&ob; p→func();**

**p=&d\_ob1; p→func();**

**p=&d\_ob2; p→func();**

**return 0;**

**}**

**Пространства имен**

Пространства имен (**namespace**) предназначены для локализации имен идентификаторов во избежание конфликтов имен. Конфликты имен имеют место, когда в одной и той же программе используются библиотеки функций и классов разных производителей (с одинаковыми идентификаторами).

Ключевое слово **namespace** путем объявления именованных областей дает возможность разделить глобальное пространство имен. По существу, пространство имен определяет область видимости. Все, что определено внутри инструкции namespace, находится внутри об-ласти видимости данного пространства имен.

Пример объявления пространства имен **MyNameSpace**:

**namespace MyNameSpace {**

**int i, k;**

**void myfunc(int j) { cout << j; }**

**class myclass {**

**public:**

**void seti(int x) { i = x; }**

**int geti( ) { return i; }**

**};**

**}**

Здесь имена переменных i и k, функции myfunc(), а также класса myclass

находятся в области видимости, определенной пространством имен My-

NameSpace.

К идентификаторам, объявленным в пространстве имен, внутри этого пространства можно обращаться напрямую. Например, в пространстве имен

MyNameSpace в инструкции return i переменная i указана явно. Однако поскольку ключевое слово namespace определяет некоторую область видимо-

сти, то при обращении извне пространства имен к объектам, объявленным внутри этого пространства, следует указывать оператор расширения области

видимости. Например, чтобы присвоить значение 10 переменной i в той

части программы, которая не входит в пространство имен MyNameSpace,

необходимо использовать следующую инструкцию:

**MyNameSpace:: i = 10;**

А чтобы объявить объект типа myclass в той части программы, которая не

входит в пространство имен MyNameSpace, нужна такая инструкция:

**MyNameSpace:: myclass ob;**

Таким образом, для доступа к члену пространства имен извне этого пространства перед именем члена следует указать имя пространства имен с

оператором расширения области видимости.

Если в программе обращения к членам пространства имен происходят достаточно часто, необходимость каждый раз указывать имя пространства имен и оператор расширения области видимости может быстро надоесть. Для решения этой проблемы была разработана инструкция **using**. У этой инструкции имеются две основные формы:

**using namespace имя;**

**using имя:: представитель;**

В первой форме параметр ***имя*** задает имя пространства имен, доступ к которому вы хотите получить. При использовании этой формы инструкции

**using** все члены, определенные в указанном пространстве имен, становятся доступными в текущем пространстве имен, и с ними можно работать напрямую, без необходимости каждый раз указывать имя пространства и оператор расширения области видимости. При использовании второй формы инструкции **using** видимым делается только указанный в инструкции представитель пространства имен. Например, пусть у нас имеется описанное выше пространство имен **MyNameSpace**, тогда правильны все представленные ниже инструкции **using** и операторы присваивания:

**using MyNameSpace::k;** // видимой делается только переменная k

**k = 210;** // инструкция правильна, поскольку переменная k видима

**using namespace MyNameSpace;** // видимыми делаются все члены

// пространства имен MyNameSpace

**i = 215;**  // инструкция правильна, поскольку видимы все члены

// пространства имен MyNameSpace

Имеется возможность объявить более одного пространства имен с одним и тем же именем. Это позволяет разделить пространство имен на несколько

файлов или даже разделить пространство имен внутри одного файла. Рас-

смотрим следующий пример:

**namespace NS {**

**int i;}**

**namespace NS {**

**int j ;}**

Здесь пространство имен NS разделено на две части. Несмотря на это, содержимое каждой части по-прежнему остается в одном и том же пространстве имен — пространстве NS.

**Перегрузка операторов**

Перегрузка операторов позволяет использовать знакомые операторы для работы с новыми типами данных, такими, например, как класс. Когда оператор перегружается, то старое его значение сохраняется, и дополнительно к этому он приобретает новое значение, связанное с классом, для которого он определен.

Для перегрузки оператора создается **оператор-функция**, которая является членом класса, для которого она определена. Имя оператор-функции –

**operator#( ),**

где вместо знака **#** следует подставить перегружаемый оператор (например: **operator\***)*.* В скобках указывается список аргументов (параметров).

В С++ можно перегружать большинство операторов, кроме: **. :: ?**.

Оператор-функция (за исключением оператора =) наследуется производным классом. Тем не менее, для производного класса тоже можно перегружать любой оператор (в том числе уже перегруженный в базовом классе).

Если в классе перегружается бинарный оператор, то необходимо указать, где брать оба операнда. Однако в оператор-функции, которая перегружает этот оператор, указывается только один параметр. Этим параметром будет объект, стоящий справа от оператора; объект, стоящий слева генерирует вызов оператор‑функции и передается неявно, с помощью указателя ***this*.**

Пример.

**#nclude <iostream>**

**using namespace std;**

**class coord**

**{int x, y;**

**public:**

**coord() {x=0; y=0;}**

**coord(int i, int j) {x=i; y=j;}**

**int get\_x() {return x;}**

**int get\_y() {return y;}**

**coord operator+(coord ob2);**

**};**

**coord coord:: operator+(coord ob2)**

**{**

**coord temp;**

**temp.x=x+ob2.x;**

**temp.y=y+ob2.y;**

**return temp;**

**}**

**int main ()**

**{**

**coord obj1(10, 5),**

**obj2(5,3),**

**obj3;**

**obj3=obj1+obj2;**

**cout <<”obj1+obj2: \n” <<”x=” <<obj3.get\_x()<<”\t”;**

**cout <<”y=” <<obj3.get\_y() <<”\n”;**

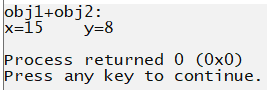
**return 0;**

**}**

В описанном в примере классе: две закрытые переменные, перегруженный конструктор, функции получения значений закрытых переменных и оператор-функция.

Оператор-функция возвращает объект типа ***coord***; чтобы его иметь, он создается внутри функции. У оператора-функции – один параметр, вторым является ***this***и его переменные ***x*** и ***y*** внутри оператор-функции указываются ***открыто***. Локальный объект ***temp*** позволяет хранить результат сложения, оставляя неизменными слагаемые и обеспечивая присваивание результата сложения.

Вид экрана:



Операторы отношений (например, "==" или "<") и логические операторы (например, "&&" или "||") также можно перегружать. Перегруженный оператор отношения или логический оператор возвращает одно из двух возможных значений: **true** или **false**. Это соответствует обычному применению этих операторов и позволяет использовать их в условных выражениях.

Пример перегрузки оператора "**==**".

**#include <iostream>**

**using namespace std;**

**class coord**

**{int x, y;**

**public:**

**coord() {x=0; y=0;}**

**coord(int i, int j) {x=i; y=j;}**

**bool operator==(coord ob2);**

**};**

**bool coord::operator==(coord ob2)**

**{**

**if((x == ob2.x) && (y == ob2.y))**

**return true;**

**else**

**return false;**

**}**

**int main ()**

**{**

**coord obj1(10, 5),**

**obj2(5,3),**

**bool bl;**

**bl = obj1 == obj2;**

**cout <<"obj1=obj2? ";**

**if(bl==0)**

**cout << "false";**

**else**

**cout << "true";**

**return 0;**

**}**

Поскольку операторная функция **operator==()** возвращает результат типа **bool**, ее можно использовать для управления инструкцией **if**.

**Статические члены класса**

Применяя ключевое слово **static** к члену класса, тем самым утверждается, что существует только одна копия этого **static-**члена, которая используется всеми объектами класса. Все статические данные при первом создании объекта инициализируются нулевыми значениями, если не представлено других значений инициализации.

При объявлении статического члена данных в классе программист не должен его определять. Необходимо обеспечить его глобальное определение вне этого класса. Это реализуется путем повторного объявления этой статической переменной с помощью оператора разрешения области видимости, который позволяет идентифицировать, к какому классу она принадлежит.

Только в этом случае для этой статической переменной будет выделена память.

Пример использования static-члена класса.

**#include <iostream>**

**using namespace std;**

**class ShareVar {**

**static int num;**

**public:**

**void setnum(int i) { num = i; };**

**void shownum() { cout << num << " "; }**

**};**

**int ShareVar::num;** // определяется num

**int main()**

**{**

**ShareVar a, b;**

**a.shownum();** // выводится 0

**b.shownum();** // выводится 0

**a.setnum(10);** // устанавливается num равным 10

**a.shownum();** // выводится 10

**b.shownum();** // также выводится 10

**return 0;**

**}**

Обратите внимание на то, что статический целочисленный член num объявлен и в классе ShareVar, и определен в качестве глобальной переменной. Как было заявлено выше, необходимость такого двойного объявления вызвана тем, что при объявлении члена num в классе ShareVar память для него не выделяется. C++ инициализирует переменную num значением 0, поскольку никакой другой инициализации в программе нет.

Таким образом, при объявлении члена класса статическим обеспечивается создание только одной его копии, которая будет совместно использоваться всеми объектами этого класса.

**Практические задания**

*Выберите один из вариантов задания.*

**Вариант А.**

**Задание 1.** Использовать структуру классов, спроектированную на практическом занятии №5. Наследование одного из производных классов осуществить по типу **private**. *Перегрузить оператор* для работы с объектами классов и использовать его для определения *Вычисляемого показателя*.

**Задание 2.** Перегрузить функцию, определяющую *Вычисляемый показатель*. В проект включить *виртуальную функцию,* выводящую на экран характеристики объекта каждого класса (кроме наследника с типом **private**). Добавить в проект статический член в каждый класс, обозначающий общее свойство для всех объектов класса.

Проект разместить в нескольких файлах *в разных пространствах имён.*

**Задание 3.** Создать массивы объектов спроектированных классов и разработать меню для демонстрации содержимого проекта.

**Вариант В.**

**Задание 1**. Использовать структуру классов, спроектированную на практическом занятии №5. Наследование осуществляется по типу **public**. *Перегрузить оператор* для работы с объектами классов и использовать его для определения *Вычисляемого показателя*.

**Задание 2.** Перегрузить функцию, определяющую *Вычисляемый показатель*. Добавить в проект статический член в каждый класс, обозначающий общее свойство для всех объектов класса.

Проект разместить в нескольких файлах *в разных пространствах имён.*

**Задание 3.** Разработать меню для демонстрации содержимого проекта.

**Вариант С.**

**Задание 1**. Использовать структуру классов, спроектированную на практическом занятии №5. Наследование осуществляется по типу **public**.

**Задание 2.** Перегрузить функцию, определяющую *Вычисляемый показатель*. Проект разместить в нескольких файлах *в разных пространствах имён.*

Отчет оформляется по общеустановленным правилам в *электронном виде* со следующим содержанием:

1. титульный лист (см. Приложение),
2. тема и цель практической работы,
3. задание на практическую работу,
4. текст программы с комментариями,
5. результаты работы программы (вид экрана) и
6. выводы по созданному проекту и использованию средств языка программирования.