Объектно-ориентированное программирование

Практическое занятие №4.
Виртуальные функции

Автор: И.О. Архипов, к.т.н., доцент

4.1. Указатели на производные классы

Виртуальные функции используются для поддержки динамического полиморфизма (run-time polymorphism).

В C++ полиморфизм поддерживается двумя способами.

1. при компиляции он поддерживается посредством перегрузки операторов и функций;
2. во время выполнения программы он поддерживается посредством виртуальных функций.

Динамический полиморфизм позволяет повысить гибкость программ.

Основой виртуальных функций и динамического полиморфизма являются указатели на производные классы.

4.1. Указатели на производные классы

Особенность указателей в C++: указатель, объявленный в качестве указателя на базовый класс, также может использоваться, как указатель на любой класс, производный от этого базового.

Пример:

```
base *p;           // указатель базового класса
base base_ob;      // объект базового класса
derived derived_ob;  // объект производного класса

   // p может указывать на объект базового класса
p = &base__ob;

   // p может указывать на объект производного класса
p = &derived_ob;
```

4.1. Указатели на производные классы

Особенности использования указателей при наследовании:

1. Для указания на объект производного класса можно воспользоваться указателем базового класса.

При этом доступ может быть обеспечен только к тем объектам производного класса, которые были унаследованы от базового.

Объясняется это тем, что базовый указатель "знает" только о базовом классе и ничего не знает о новых членах, добавленных в производном классе.

2. Указатель базового класса можно использовать для указания на объект производного класса, но обратный порядок недействителен.

3. Указатель производного класса нельзя использовать для доступа к объектам базового класса.

4.1. Указатели на производные классы

Особенности использования указателей при наследовании:

4. Следует помнить, что арифметика указателей связана с типом данных (т. е. классом), который задан при объявлении указателя.

Таким образом, если указатель базового класса указывает на объект производного класса, а затем инкрементируется, то он уже не будет указывать на следующий объект производного класса.

Этот указатель будет указывать на следующий объект базового класса.

4.1. Указатели на производные классы

```
// Демонстрация указателя на объект производного класса
#include <iostream>
using namespace std;

class base {
   int x;
public:
   void setx(int i){x = i;}
   int getx() {return x;}
};

class derived: public base {
   int y;
public:
   void sety(int i) {y = i;}
   int gety() {return y;}
};
```

4.1. Указатели на производные классы

```
int main() {
   base *p;    // указатель базового класса
   base b_ob;     // объект базового класса
   derived d_ob;  // объект производного класса
      // использование указателя p
      // для доступа к объекту базового класса
   p = &b_ob;
   p->setx(10);   // доступ к объекту базового класса
   cout << "Объект базового класса x:  ";
   cout << p->getx()  << '\n';
```

4.1. Указатели на производные классы

```
    // использование указателя p
    // для доступа к объекту производного класса
  p = &d_ob;   // указывает на объект произв. класса
  p->setx(99); // доступ к объекту произв. класса

    // p нельзя использовать для установки y,
    // делаем это напрямую
  d_ob.sety(88);
  cout << "Объект производного класса x:  ";
  cout << p->getx() << ' ';
  cout << "Объект производного класса y:  ";
  cout << d_ob.gety() << '\n';
  return 0;
}
```

4.2. Виртуальные функции

Виртуальная функция (*virtual function*) является членом класса.

Она объявляется внутри базового класса и переопределяется в производном классе.

Для того, чтобы функция стала виртуальной, перед объявлением функции ставится ключевое слово `virtual`.

При переопределении виртуальной функции в производном классе, ключевое слово `virtual` не требуется.

Если класс, содержащий виртуальную функцию, наследуется, то в производном классе виртуальная функция переопределяется.

4.2. Виртуальные функции

Виртуальная функция реализует идею "один интерфейс, множество методов", которая лежит в основе полиморфизма.

Виртуальная функция внутри базового класса определяет вид интерфейса этой функции.

Каждое переопределение виртуальной функции в производном классе определяет ее реализацию, связанную со спецификой производного класса.

Таким образом, переопределение создает конкретный метод.

4.2. Виртуальные функции

Виртуальная функция может вызываться так же, как и любая другая функция-член.

Однако наиболее интересен вызов виртуальной функции через указатель, благодаря чему поддерживается динамический полиморфизм.

Если указатель базового класса ссылается на объект производного класса, который содержит виртуальную функцию и для которого виртуальная функция вызывается через этот указатель, то компилятор определяет, какую версию виртуальной функции вызвать, основываясь при этом 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.

4.2. Виртуальные функции

Тип объекта, на который ссылается указатель, определяет ту версию виртуальной функции, которая будет выполняться.

Поэтому, если два или более различных класса являются производными от базового, содержащего виртуальную функцию, то, если указатель базового класса ссылается на разные объекты этих производных классов, выполняются различные версии виртуальной функции.

Этот процесс является реализацией принципа динамического полиморфизма.

Фактически, о классе, содержащем виртуальную функцию, говорят как о полиморфном классе (*polymorphic class*).

4.2. Виртуальные функции

```
// Простой пример использования виртуальной функции
#include <iostream>
using namespace std;

class base {
public:
int i;
base(int x) {i = x;}
virtual void func() {
    cout << "Выполнение функции базового класса";
    cout << i <<  ‘\n’ ;
}
};
```

4.2. Виртуальные функции

```
class derived1: public base {
public:
derived1(int x): base(x){ }
void func(){
   cout << "Выполнение функции класса derived1: ";
   cout << i * i <<  '\n';
}
};

class derived2: public base {
public:
derived2(int x): base(x){ }
void func(){
   cout << "Выполнение функции класса derived2: ";
   cout << i + i <<  '\n';
}
};
```

4.2. Виртуальные функции

```
int main () {
base *p;
base ob(10);
derived1 d_ob1(10);
derived2 d_ob2(10);
p = &ob;
p->func();  // функция класса base
p = &d_ob1;
p->func();  // функция произв. класса derived1
p = &d_ob2;
p->func();  // функция произв. класса derived2
return 0;
}
```

1. Тип адресуемого через указатель объекта определяет вызов той или иной версии подменяемой виртуальной функции
2. Выбор конкретной версии происходит в процессе выполнения программы.

4.2. Виртуальные функции

Отличия переопределения виртуальной функции внутри производного класса от перегрузки функций.

1. Перегружаемая функция должна отличаться типом и/или числом параметров, а переопределяемая виртуальная функция должна иметь точно такой же тип параметров, то же их число, и такой же тип возвращаемого значения. Если при переопределении виртуальной функции изменить число или тип параметров, она становится перегружаемой функцией и ее виртуальная природа теряется
2. Виртуальная функция должна быть членом класса. Это не относится к перегружаемым функциям.
3. Деструкторы могут быть виртуальными, а конструкторы нет.
4. Чтобы подчеркнуть разницу между перегружаемыми функциями и переопределяемыми виртуальными функциями, для описания переопределения виртуальной функции используется термин подмена (*overriding*).

4.2. Виртуальные функции

Виртуальные функции имеют иерархический порядок наследования.

Если виртуальная функция не подменяется в производном классе, то используется версия функции, определенная в базовом классе.

```
// Иерархический порядок виртуальных функций
#include <iostream>
using namespace std;

class base {
public:
int i ;
base(int x) {i = x;}
virtual void func() {
   cout << "Выполнение функции базового класса: ";
   cout << i << '\n';
}
};
```

4.2. Виртуальные функции

```
class derivedl: public base {
public:
derivedl(int x): base(x) {  }
void func (){
   cout << "Выполнение функции класса derived1: ";
   cout << i * i <<  '\n';
}
};

class derived2: public base {
public:
derived2(int x): base(x) {  }
// в классе derived2 функция func() не подменяется
};
```

4.2. Виртуальные функции

```
int main() {
base *p;
base ob(10);
derivedl d_obl(10);
derived2 d_ob2(10);

p = &ob;
p->func();  // функция базового класса

p = &d_obl;
p->func();  // функция произв. класса derived1

p = &d_ob2;
p->func();  // функция базового класса

return 0;
}
```

4.2. Виртуальные функции

```
/*Работа виртуальной функции при наличии случайных событий во время выполнения программы.*/

#include <iostream>
#include <cstdlib>
using namespace std;

class base {
public:
int i ;
base(int x) {i = x; }
virtual void func(){
   cout << "Выполнение функции базового класса: ";
   cout << i <<  '\n';
}
};
```

4.2. Виртуальные функции

```
class derived1: public base {
public:
derived1(int x): base(x) { }
void func() {
   cout << "Выполнение функции класса derived1: ";
   cout << i * i << '\n';
}
};

class derived2: public base {
public:
derived2(int x): base(x) { }
void func(){
   cout << "Выполнение функции класса derived2: ";
   cout << i + i <<  '\n';
}
};
```

4.2. Виртуальные функции

```
int main() {
   base *p;
   derived1 d_ob1(10);
   derived2 d_ob2(10);

   int i, j;
   for(i=0; i<10;  i++) {
j = rand();
if((j%2)) p = &d_ob1; // если число нечетное
              // использовать объект d_ob1
   else p = &d_ob2;   // если число четное
              // использовать объект d_ob2
   p->func();   // вызов подходящей версии функции
   }
   return 0;
}
```

4.2. Виртуальные функции

Рассмотрим более реальный пример использования виртуальной функции.

Создадим исходный базовый класс `area`, в котором сохраняются две размерности фигуры.

В классе `area` объявим виртуальную функцию `getarea()`, которая, при ее подмене в производном классе, возвращает площадь фигуры, вид которой задается в производном классе.

В данном случае определение функции `getarea()` внутри базового класса задает интерфейс.

Конкретная реализация остается тем классам, которые наследуют класс `area`.

4.2. Виртуальные функции

```
#include <iostream>
using namespace std;
class area {
double dim1, dim2;  // размеры фигуры
public:
void setarea(double d1, double d2) {
   dim1 = d1;
   dim2 = d2;
}
void getdim(double &d1, double &d2) {
   d1 = dim1;
   d2 = dim2;
}
virtual double getarea() {
   cout << “Следует подменить эту функцик\n";
   return 0.0;
}
};
```

4.2. Виртуальные функции

```
class rectangle: public area {
public:
double getarea(){
double dl, d2;
getdim (dl, d2);
   return dl * d2;
}
};

class triangle: public area {
public:
double getarea(){
double dl,  d2;
getdim(dl,  d2);
   return 0.5 * dl * d2;
}
};
```

4.2. Виртуальные функции

```
int main() {
area *p;
rectangle r;
triangle t;
r.setarea(3.3, 4.5);
t.setarea(4.0, 5.0);

p = &r;
cout << "Площадь прямоугольника:  ";
cout << p->getarea() << '\n';

p = &t;
cout << "Площадь треугольника:  ";
cout << p->getarea()  << ‘\n’;
return 0;
}
```

4.2. Виртуальные функции

Определение `getarea()` внутри класса `area` является только "заглушкой" и не выполняет никаких действий.

Поскольку класс area не связан с фигурой конкретного типа, то нет значимого определения, которое можно дать функции `getarea()` внутри класса `area`.

Для того чтобы нести полезную нагрузку, функция `getarea()` должна быть переопределена в производном классе.

4.3. Чистые виртуальные функции

Виртуальная функция может не выполнять значимых действий.

Ситуация, когда в базовом классе законченный тип данных не определяется является типичной.

Вместо этого в базовом классе содержится основной набор функций-членов и переменных, для которых в производном классе определяется все недостающее.

Когда в виртуальной функции базового класса отсутствует значимое действие, в производном классе такая функция обязательно должна быть переопределена.

Для реализации этого положения в C++ поддерживаются так называемые чистые виртуальные функции (*pure virtual function*).

4.3. Чистые виртуальные функции

Чистые виртуальные функции не определяются в базовом классе.

В базовый класс включаются только прототипы этих функций.

Для чистой виртуальной функции используется следующая основная форма:

```
virtual тип имя_функции(список_параметров) = 0;
```

Приравнивание функции нулю сообщает компилятору, что в базовом классе не существует тела функции.

Если функция задается как чистая виртуальная, это предполагает, что она обязательно должна подменяться в каждом производном классе.

Если этого нет, то при компиляции возникнет ошибка.

Таким образом, создание чистых виртуальных функций — это путь, гарантирующий, что производные классы обеспечат их переопределение.

4.3. Чистые виртуальные функции

Если класс содержит хотя бы одну чистую виртуальную функцию, то о нем говорят как об абстрактном классе (*abstract class*).

Технически такой класс неполон, и ни одного объекта этого класса создать нельзя.

Абстрактные классы могут быть только наследуемыми и никогда не бывают изолированными.

Однако, по-прежнему можно создавать указатели абстрактного класса, благодаря которым достигается динамический полиморфизм.

Также допускаются и ссылки на абстрактный класс.

4.3. Чистые виртуальные функции

```
// Создание абстрактного класса
#include <iostream>
using namespace std;
class area {
double diml, dim2; // размеры фигуры
public:
void setarea(double dl, double d2) {
   diml = dl;
   dim2 = d2;
}
void getdim(double &dl, double &d2) {
   dl = diml;
   d2 = dim2;
}
virtual double getarea()=0;   // чистая виртуальная
                  // функция
};
```

4.3. Чистые виртуальные функции

```
class rectangle: public area {
public:
double getarea(){
   double d1, d2;
   getdim (d1, d2);
   return d1 * d2;
}
};

class triangle: public area {
public:
double getarea(){
   double d1, d2;
   getdim (d1,  d2);
   return 0.5 * d1 * d2;
}
};
```

4.3. Чистые виртуальные функции

```
int main() {
area *p;
rectangle r;
triangle t;
r.setarea(3.3, 4.5);
t.setarea(4.0, 5.0);

p = &r;
cout << "Площадь прямоугольника:  ";
cout << p->getarea() << '\n';

p = &t;
cout << "Площадь треугольника: ";
cout << p->getarea()  << '\n';

return 0;
}
```

4.3. Чистые виртуальные функции

Покажем, как при наследовании сохраняется виртуальная природа функции:

```
#include <iostream>
using namespace std;

class base {
public:
virtual void func(){
   cout << "Выполнение функции базового класса\n";
}
};

class derived1: public base {
public:
void func(){
   cout << "Выполнение функции класса derived1\n";
}
};
```

4.3. Чистые виртуальные функции

```
class derived2: public derivedl  {
public:
void func(){
   cout << "Выполнение функции класса derived2\n";
}
};
int main() {
base *p, ob;
derived1 d_ob1;
derived2 d_ob2;
p = &ob;
p->func();  // функция базового класса
p = &d_obl;
p->func();  // функция произв. класса derived1
p = &d_ob2;
p->func();  // функция произв. класса derived2
return 0;
}
```

4.4. Применение полиморфизма

Раннее связывание (*early binding*) относится к событиям, о которых можно узнать в процессе компиляции.

Функции раннего связывания — это обычные функции, перегружаемые функции, невиртуальные функции-члены и дружественные функции.

При компиляции функций этих типов известна вся необходимая для их вызова адресная информация.

Главное преимущество раннего связывания — это высокое быстродействие программ.

Определение нужной версии вызываемой функции во время компиляции программы — это самый быстрый метод вызова функций.

Главный недостаток — потеря гибкости.

4.4. Применение полиморфизма

Позднее связывание (*late binding*) относится к событиям, которые происходят в процессе выполнения программы.

Вызов функции позднего связывания — это вызов, при котором адрес вызываемой функции до запуска программы неизвестен.

В C++ виртуальная функция является объектом позднего связывания.

Если доступ к виртуальной функции осуществляется через указатель базового класса, то в процессе работы программа должна определить, на какой тип объекта он ссылается, а затем выбрать, какую версию подменяемой функции выполнить.

Главное преимущество позднего связывания — это гибкость во время работы программы. Программа может легко реагировать на случайные события.

Основной недостаток — это необходимость выполнения большего количества действий для вызова функции.

4.4. Применение полиморфизма

Рассмотрим пример

Определим исходный базовый класс для связанного списка целых.

Интерфейс списка определяется с помощью чистых виртуальных функций `store()` и `retrieve()`.

Для хранения значения в списке вызывается функция `store()`.

Для выборки значения из списка вызывается функция `retrieve()`.

В каждом производном классе явно определяется, какой тип списка будет поддерживаться.

В программе реализованы списки двух типов: очередь и стек.

Хотя способы работы с этими двумя списками совершенно различны, для доступа к каждому из них применяется один и тот же интерфейс.

4.4. Применение полиморфизма

```
// Демонстрация виртуальных функций
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class list {
public:
list *head; // указатель на начало списка
list *tail; // указатель на конец списка
list *next; // указатель на следующий эл-т списка
int num; // число для хранения
list() {head = tail = next = NULL;}
virtual void store(int i) = 0;
virtual int retrieve() = 0;
};
```

4.4. Применение полиморфизма

```
// Создание списка типа очередь
class queue: public list {
public:
void store(int i);
int retrieve();
};

// Создание списка типа стек
class stack: public list {
public:
void store(int i);
int retrieve();
};
```

4.4. Применение полиморфизма

```
void queue::store(int i) {
list *item;
item = new queue;
if(!item) {
   cout << "Ошибка выделения памяти\n";
   exit (1);
}
item->num = i;

// добавление элемента в конец списка
if(tail) tail->next = item;
tail = item;
item->next = NULL;
if(!head) head = tail;
}
```

4.4. Применение полиморфизма

```
int queue::retrieve() {
int i ;
list *p;
if(!head)   {
   cout << "Список пуст\n";
   return 0;
}

// удаление элемента из начала списка
i = head->num;
p = head;
head = head->next;
delete p;
return i;
}
```

4.4. Применение полиморфизма

```
void stack::store(int i) {
list *item;
item = new stack;
if(!item) {
   cout << "Ошибка выделения памяти\n";
   exit(1);
}
item->num = i;

// добавление элемента в начало списка
if(head) item->next = head;
head = item;
if(!tail)  tail = head;
}
```

4.4. Применение полиморфизма

```
int stack::retrieve() {
int i;
list *p;
if(!head) {
   cout << "Список пуст\n";
   return 0;
}

// удаление элемента из начала списка
i = head->num;
p = head;
head = head->next;
delete p;
return i;
}
```

4.4. Применение полиморфизма

```
int main() {
list *p;

// демонстрация очереди queue q_ob;
p = &q_ob;  // указывает на очередь
p->store(1) ;
p->store(2);
p->store(3) ;

cout  << "Очередь:  ";
cout  << p->retrieve();
cout  << p->retrieve();
cout  << p->retrieve();
cout << '\n' ;
```

4.4. Применение полиморфизма

```
// демонстрация стека
stack s_ob;
p = &s_ob;  // указывает на стек
p->store(1);
p->store(2);
p->store(3);

cout  << "Стек:   ";
cout  << p->retrieve();
cout  << p->retrieve();
cout  << p->retrieve();
cout << '\n' ;
return 0;
}
```

4.4. Применение полиморфизма

Функция `main()` в предыдущей программе только иллюстрирует работу классов.

Для изучения динамического полиморфизма попробуем использовать в предыдущей программе следующую версию функции `main()`:

4.4. Применение полиморфизма

```
int main() {
list *p;
stack s_ob;
queue q_ob;
char ch;
int i;
for(i=0; i<10; i++) {
   cout << "Стек или Очередь? (С/О): ";
   cin >> ch;
   ch = tolower(ch);
   if(ch=='o') p = &q_ob;
      else p = &s_ob;
   p->store(i);
}
```

4.4. Применение полиморфизма

```
cout << "Введите K для завершения работы\n";
for(;;) {
   cout << "Извлечь из Стека или Очереди? (C/O):";
   cin >> ch;
   ch = tolower(ch);
   if(ch=='K')  break;
   if(ch=='o')  p = &q_ob;
   else p = &s_ob;
   cout << p->retrieve() << '\n';
}
cout << '\n';
return 0;
}
```