МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Программной инженерии

Специальность 1-40 01 01 Программное обеспечение информационных технологий

**РЕФЕРАТ**

на тему:

«Ранее или позднее связывание»

Выполнил:

студент I курса 6 группы

специальности ПОИТ Шимко А.А

Преподаватель:

Белодед Николай Иванович

**Оглавление**

Введение.Полиморфизм…………….…….………………………………..…..3-5

Достижение позднего связывания с помощью кода …………………….…...6-7

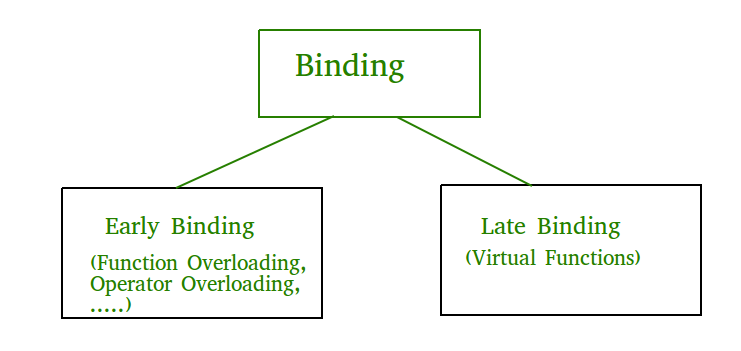
Статическое или раннее связывание………………………………………...…..8

Таблица виртуальных функций……………….……………………………..…..9

Заключение……………………………………………………………………….10

**Введение**

**Связывание** относится к процессу, который используется для преобразования идентификаторов (например, имен переменных и функций) в адреса. Хотя связывание используется как для переменных, так и для функций, в этом реферате мы сосредоточимся на связывании функций. Привязка выполняется для каждой переменной и функций. Для функций это означает, что компилятор сопоставляет вызов с правильным определением функции. Это происходит либо во время компиляции, либо во время выполнения.



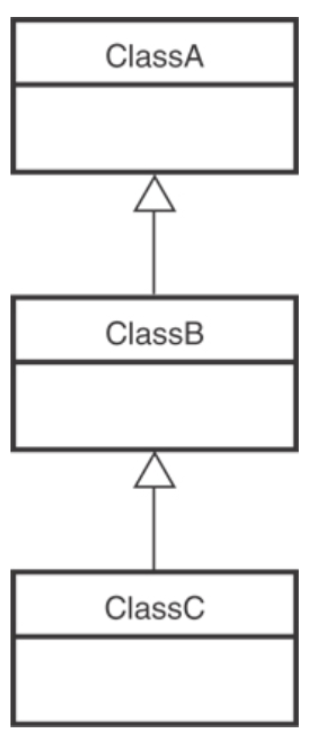
Виртуальные функции реализуют так называемый полиморфизм. Термин «полиморфизм» происходит от греческих слов poly (много) и morphos (форма). *Полиморфизм* – это свойство программного кода изменять свое поведение в зависимости от ситуации, возникающей при выполнении программы. В контексте реализации *полиморфизм* – это технология вызова виртуальных функций, реализуемых в иерархически связанных классах. Иерархия классов формируется на базе механизма наследования.

С понятием полиморфизма тесно связано понятие *виртуальная функция*. Это специальным образом оформленная функция, которая может быть в так называемом полиморфном состоянии – состоянии, при котором вызов нужной функции из набора виртуальных формируется на этапе *позднего связывания*. Понятие позднее связывание означает, что код вызова нужной функции формируется при выполнении программы. Иными словами, в исходном коде вызов функции только обозначается без точного указания того, какая именно функция должна быть вызвана. Объект, для которого вызывается виртуальная функция, имеет общее значение. Конкретный объект и соответствующая ему функция будут сформированы на этапе выполнения программы.

Механизм виртуальных функций реализует основополагающий *принцип полиморфизма*: «один интерфейс, несколько реализаций» или «один интерфейс, несколько методов».

Как известно, существует также и *раннее связывание*. При раннем связывании известно, какие объекты используются при вызове функции в каждом случае. Это накладывает ограничения на возможности программного кода. Изменение объектов для функций с одинаковыми именами невозможно в процессе выполнения программы, это изменение нужно программировать каждый раз вручную (это и есть ограничение кода).

Не все задачи нуждаются в использовании позднего связывания. Выбор того, какой вид связывания будет использоваться в программе, зависит от специфики решаемой задачи.



Для реализации позднего связывания требуется следующее:

* классы обязаны образовывать иерархию с помощью механизма наследования;
* в иерархии классов были определены функции, имеющие одинаковое имя и список параметров;
* функции с одинаковым именем и параметрами должны быть отмечены как виртуальные (с ключевым словом virtual).

Относительно парадигмы классов и объектов для полиморфизма можно выделить следующие характерные особенности:

* полиморфизм не характеризует объект;
* реализацию полиморфизма определяет архитектура класса;
* полиморфизм является характеристикой функций-членов класса;
* весь класс не может быть полиморфным, полиморфными могут быть только функции члены класса.

**Достижение позднего связывания с помощью кода**

По умолчанию раннее связывание выполняется в C ++. Позднее связывание (обсуждается ниже) достигается с помощью ключевого слова virtual:

#include<iostream>

using namespace std;

class Base

{

public:

void show() { cout << “ In Base \n”; }

};

class Derived : public Base

{

public:

void show() { cout << “In Derived \n”; }

};

int main(void)

{

Base\* bp = new Derived;

bp->show();

return 0;

}

Также рассмотрим пример вызова полиморфизма без использования виртуальной функции :

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

// parameterized constructor

Shape(int l, int w)

{

length = l;

width = w;

}

int get\_Area()

{

cout << “This is call to parent class area\n”;

// Returning 1 in user-defined function means true

return 1;

}

protected:

int length, width;

};

// Derived class

class Square : public Shape {

public:

Square(int l = 0, int w = 0)

: Shape(l, w)

{

} // declaring and initializing derived class

// constructor

int get\_Area()

{

cout << “Square area: “ << length \* width << ‘\n’;

return (length \* width);

}

};

// Derived class

class Rectangle : public Shape {

public:

Rectangle(int l = 0, int w = 0)

: Shape(l, w)

{

} // declaring and initializing derived class

// constructor

int get\_Area()

{

cout << “Rectangle area: “ << length \* width

<< ‘\n’;

return (length \* width);

}

};

int main()

{

Shape\* s;

// Making object of child class Square

Square sq(5, 5);

// Making object of child class Rectangle

Rectangle rec(4, 5);

s = &sq; // reference variable

s->get\_Area();

s = &rec; // reference variable

s->get\_Area();

return 0; // too tell the program executed

// successfully

}

В приведенном выше примере:

В приведенном выше примере:

Мы сохраняем адрес каждого дочернего объекта Rectangle и Square класса в s и  
Затем мы вызываем для него функцию get\_Area(),  
В идеале, он должен был вызывать соответствующие функции get\_Area() дочерних классов, но  
Вместо этого он вызывает get\_Area(), определенный в базовом классе.  
Это происходит из-за статической привязки, которая означает, что вызов get\_Area() устанавливается компилятором, который находится в базовом классе, только один раз.

## Статическое или раннее связывание

Вызов обычных функций и методов происходит через механизм, называемый статическим (статичным) связыванием (static binding) или ранним связыванием (early binding).

Раннее связывание использовалось во всех функциях и методах наших программ за исключением тех случаев, где мы использовали указатели на функции.

Когда мы запускаем сборку (building) программы, компилятор просматривает исходный код и превращает все операторы в команды процессора. Допустим, в коде встречается вызов какой-нибудь функции:

someFunction(arg); // some - какой-то

Если это обычная функция (не указатель на функцию), то при вызове используется механизм раннего связывания.

Во время компиляции для кода (определения) функции выделяется память, и назначаются адреса для каждого оператора. Первый адрес в определении (теле функции) является адресом функции. При вызове someFunction, процессор будет переходить на адрес функции и начнёт выполнять тело функции. Самое важное здесь то, что адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции. Это и есть раннее или статичное связывание. Т.е. имя функции крепко привязано к адресу функции.

#include <iostream>

void printValue(int value)

{

std::cout << value;

}

int main()

{

printValue(5); // это прямой вызов функции

return 0;

}

### **Таблица виртуальных функций**

Таблица виртуальных функций — это массив указателей на методы, поддерживаемые объектом. Если вы используете C, объект отображается как структура, первый член которой является указателем на таблицу виртуальных функций (lpVtbl); то есть первый элемент указывает на массив, содержащий указатели функций. Все методы принимают указатель на таблицу функций в качестве первого параметра. Таким образом, в следующем примере вызывается метод Read объекта pStream :

pStream->lpVtbl->Read(pStream, parameters)

В C+ + указатель на таблицу виртуальных функций( этот указатель) неявен. Следующий пример эквивалентен предыдущему примеру при использовании C+ +:

pStream->Read(parameters)

**Заключение**

Работу функций трудно объяснить, используя язык программирования высокого уровня. Наиболее просто это сделать с помощью ассемблера, которого мы пока ещё не знаем.

В заключении я попытаюсь кратко резюмировать материал урока.

При вызове обычной функции во время выполнения программы, подставляется её адрес, который был присвоен на этапе компиляции. Это раннее или статическое связывание (early/static binding).

При использовании указателя на функцию, в нём хранится адрес фактического местоположения реальной функции. Этот адрес был назначен на этапе компиляции (абзац выше), но указатель может менять своё значение во время выполнении программы. Это позволяет вызывать с помощью указателя разные функции. Это пример позднего/динамического связывания (late/dynamic binding). Ещё одним примером позднего связывания являются виртуальные функции.

Виртуальные функции объявляются с помощью ключевого слова virtual в базовом классе. При этом для базового класса и для всех производных создаётся таблица указателей на функции – виртуальная таблица методов/функций (virtual function table или vtable). Для каждого класса создаётся своя таблица. Количество элементво в таблице равно количеству виртуальных методов. В таблице хранятся фактические адреса методов, определённых в классах. Также в базовом классе объявляется дополнительное поле \_\_vfptr (наследуется всеми производными классами) – указатель на таблицу виртуальных функций класса. Т.е. когда создаётся объект самого класса или любого производного, в нём \_\_vfptr присваивается адрес таблицы виртуальных функций этого класса (или производных).

Виртуальные функции нужны в C++ для поддержки полиморфизма.