Объектно-ориентированное программирование (ООП)

ООП – это методология программирования, при которой программа строится в виде совокупности взаимодействующих между собой объектов. Объект представляет собой модель некой сущности, которая содержит совокупность данных и алгоритмов, реализующие методы обработки (преобразование) этих данных. Множество объектов, моделирующих сущности реальных систем, связаны между собой, обмениваются сообщениями и каждый объект на основе сообщения сам выбирает метод его обработки.

Следовательно, объектно-ориентированная программа представляет собой модель некоторого процесса или явления реальной действительности. Модель, по определению, описывает только часть признаков и свойств моделируемого процесса (если все, то тогда это и есть сам процесс). В зависимости от назначения модели (задачи моделирования) определяется, что включить в модель, а что не учитывать в ней, т.е. определяется степень *абстрактности* моделирования.

Существенным моментом является также то, что модель может основываться на иных физических принципах, чем моделируемый объект, например, описание колебательного контура математическим выражением или программой, реализующей это математическое выражение.

Для понимания связи и взаимоотношений ООП с реальной действительностью, моделями которой являются программы для ЭВМ, надо владеть основными понятиями

Теории систем и системного анализа.

Понятия теории систем и системного анализа (ТСиСА)

Система (С.) – совокупность (множество) отдельных объектов и (обязательные) связи между ними, которые придают данной совокупности больше свойств, чем простая сумма свойств составляющих С. объектов при отсутствии связей между ними (С. от греч. σύστημα, "составленный").

***Теория систем (Общая теория систем)* – это наука, предметом которой является разработка логико-концептуального и математического аппарата теоретического исследования объектов, представляющих собой системы.**

Целью исследований в рамках этой теории является изучение:

- **различных видов и типов С.;**
- **основных принципов и закономерностей поведения С.;**
- **функционирования и развития С.**

***Системный анализ* - совокупность понятий, методов, процедур и технологий для изучения, описания, реализации явлений и процессов различной природы и характера, междисциплинарных проблем; это совокупность общих законов, методов, приемов инженерного исследования таких систем.**

Цель системного анализа:

- **полнее понять механизмы функционирования С.;**
- **повысить эффективность функционирования С.;**
- **получить систематические, планомерно-поэтапные методы (способы) анализа и создания (построения) С.**

Системный подход – это подход, при котором любая система (объект) рассматривается как совокупность взаимосвязанных компонентов, имеющая выход (цель), вход (ресурсы), связь с внешней средой, обратную связь.

Любую *предметную область* можно рассматривать как систему.

Понятия теории систем и системного анализа (ТСиСА)

Задачи ТСиСА:

- Декомпозиция – представление С. в виде подсистем и элементов.
- Анализ – определение свойств С. и среды её окружающей.
- Синтез – построение С., обладающей заданными свойствами.
- Оптимизация – нахождение С. с экстремальными значениями её функционалов.

Некоторые термины:

Система – это некоторая совокупность взаимосвязанных объектов (компонентов С.), обладающая свойствами, не сводящимися к свойствам отдельных объектов.

Подсистема – это часть С. с некоторыми связями и отношениями, любая С. состоит из подсистем, подсистема любой С. может быть сама рассмотрена как С. (иерархическая вложенность подсистем).

Надсистема – это более крупная с С., частью которой является рассматриваемая С.

Элемент – компонент нижнего иерархического уровня С. (не имеющий внутренней структуры).

Границы системы – это материальные и нематериальные ограничители, дистанцирующие систему от окружающей среды.

Место С. – место в иерархии, определяет надсистему, в которую входит данная система, системы одного уровня с которыми взаимодействует С., подсистемы входящие в данную С.

Рамки С. – ресурсные, технологические (функциональные), географические и другие границы данной С.

Структура – устойчивая картина взаимоотношений между компонентами С. (картина связей и их стабильностей).

Процесс – динамическое изменение С. во времени.

Функция – процесс, происходящий внутри С. и имеющий определённый результат.

Состояние – положение С. относительно других её положений.

Поведение – целеориентированная активность С., служащая для осуществления контакта с окружающей средой.

Понятия теории систем и системного анализа (ТСиСА)

Основные принципы системного анализа:

- **Целостность – рассмотрение системы одновременно как единое целое и как подсистему для вышестоящих уровней.**
- **Иерархичность строения – наличие множества компонентов, расположенных на основе подчинения компонентов низшего уровня компонентам высшего уровня.**
- **Структуризация – проведение анализа компонентов системы и их взаимосвязи в рамках конкретной организационной структуры. Как правило, процесс функционирования системы обусловлен не столько свойствами ее отдельных компонентов, сколько свойствами самой структуры (связями между компонентами).**
- **Множественность – использование множества математических, кибернетических и экономических моделей для описания отдельных компонентов и системы в целом.**

Понятия теории систем и системного анализа (ТСиСА)

Классификация систем:

- **Физические – абстрактные.** Ф. – реальные С. (объекты, явления, процессы) и А. – отображения (модели реальных С.). Для одной реальной С. может быть построено множество моделей (объективная и субъективная реальности, реальная и виртуальная действительности.
- **Простые – сложные.** Простые – Большие – Сложные: Простые – мало элементов, немного однородных связей; Большие – много элементов, много однородных связей, Сложные – много элементов, много неоднородных связей).
  Сложные С. – структурная и функциональная сложность.
  Отличительные свойства сложных С.: (а) робастность, (б) неоднородные связи: структурные (в т.ч. иерархические), функциональные, отношения истинности, каузальные (причинно-следственные) и казуальные (случайные);
- **Динамические – статические.** С. имеют изменяющиеся во времени процессы или нет
- **Открытые-замкнутые.** По отношению системы к окружающей среде
- **Естественные-искусственные-виртуальные-смешанные.**
- **С управлением – неуправляемые.**
- **Непрерывные – дискретные.**
- **Детерминированные – стохастические.**
- **и др.**

Понятия теории систем и системного анализа (ТСиСА)

Упрощенная последовательность практических действий при системном анализе некого объекта (системы):

1. Определение задач исследования (анализа) или синтеза С.
2. Определение назначения и целей С.
3. Декомпозиция С.: выявление подсистем и элементов.
4. Определение структуры системы (наличия и характеристик связей между компонентами С.)
5. Определение функций системы и её компонентов.
6. Определение задач решаемых С., её подсистемами и элементами.
7. Определение принципов моделирования компонентов С. и С. в целом (в зависимости от задач исследования (анализа) или синтеза).
8. Разработка модели системы и её компонентов (в настоящее время чаще всего программной модели).
9. Исследование модели С. (проведение с ней экспериментов).
10. Анализ результатов исследования модели С.
11-а. Формирование выводов анализа, если исследуется существующая система.
11-б. Синтез (создание) новой системы.

Понятия объектно-ориентированного программирование (ООП)

Имея теперь некоторые базовые представления о системах и системном анализе можно увидеть, что создание парадигмы[1] ООП – это распространение на деятельность по программированию системного подхода, который более полноценно описывает модели реальных систем (процессов, явлений).

Объект в ООП – это реальная сущность, описывающая (моделирующая) некоторый компонент процесса или явления. Он обладает *состоянием* и *поведением,* обменивается *сообщениями* с другими объектами. Это может быть подсистема или элемент С.

Состояние объекта описывается совокупностью параметров и свойств, характерных именно этому объекту. Значения этих параметров и свойств изменяется во времени как у реального объекта.

Поведение объекта определяется совокупностью операций, которые можно выполнять над этим объектом.

Объекты, имеющие сходные (в определенных пределах) наборы свойств и параметров, а также набор операций объединяются в класс однотипных объектов.

---

[1] **Парадигма – устоявшиеся системы научных взглядов, в рамках которых ведутся исследования – комплекс теорий, стандартов и методов, которые представляют способ организации знаний.**
Парадигма программирования — это парадигма, определяющая стиль программирования, иначе говоря – некоторый цельный набор идей и рекомендаций, определяющих стиль написания программ.
Парадигма программирования представляет (и определяет) то, как программист видит выполнение программы. Например, в объектно-ориентированном программировании программист рассматривает программу как набор взаимодействующих объектов, тогда как в функциональном программировании программа представляется в виде цепочки вычисления функций.

Понятия объектно-ориентированного программирование (ООП)

Три главные свойства объектно-ориентированного языка (ООЯ) программирования: Инкапсуляция, Полиморфизм, Наследование.

Инкапсуляция (encapsulation, incapsulation).

Это механизм, связывающий воедино код и данные, которыми код манипулирует, и защищает их от несанкционированного и неправильного использования. В ООЯ код и данные можно "упаковать" в "чёрный ящик" – **объект (object)**. Объект и есть средство инкапсуляции.

Объект представляет собой сложную переменную, тип которой определён программистом.

Внутри объекта код и данные могут быть закрытыми (private) или открытыми (public). Открытая часть объекта доступна извне из любой части программы и обеспечивает управляемое взаимодействие (интерфейс) между объектами.

Полиморфизм (polymorphism).

Это атрибут, позволяющий организовать через один интерфейс доступ к целому классу методов. Выбор конкретного метода определяется компилятором в зависимости от ситуации (например, в типом переданных извне в объект данных).

Наследование (inheritance).

Это процесс, в ходе которого один объект приобретает свойства другого. Тем самым в ООЯ реализуется идея классификации (classification), когда конкретный объект является специфическим экземпляром более общей разновидности.

Понятия объектно-ориентированного программирование (ООП)

Классы.

Класс – это абстрактный тип данных, определяемый программистом, моделирующий реальный объект в виде данных и функций для работы с ними.
Данные класса называются полями, синоним – данные-члены класса (по аналогии с полями структуры), а функции методами, синоним – функции-члены. Поля и методы называются элементами класса.
Конкретные переменные типа "класс" называются объектами (экземплярами или членами класса). Например, class myclass.

Конструкторы.

Конструктор – это особая функция, являющаяся членом класса. Её имя всегда совпадает с именем класса. Например, myclass(). Конструктор предназначен для инициализации нужной части объекта и автоматически вызывается программой в момент создания объекта.
Для глобальных и статических локальных объектов конструкторы вызываются лишь однажды. Для локальных объектов конструкторы вызываются каждый раз при входе в соответствующий блок.

Деструкторы.

Деструктор – это особая функция-антипод конструктора. Её имя совпадает с именем класса с тильдой (~) перед ним. Например, ~myclass(). Деструктор предназначен для удаления объекта. Это может потребоваться для освобождения памяти или закрытия открытого ранее файла. Деструктор вызывается автоматически при выходе объекта из области видимости (для локальных объектов - при выходе из блока, где они были объявлены; для глобальных – при выходе из main; для объектов, заданных через указатели – неявно при использовании операции delete).

ООП. Инкапсуляция, классы и объекты

Инкапсуляция (encapsulation, incapsulation) – это способ формирования объектов в ООП, состоящий в ограничении внешнего доступа к данным и объединении данных с методами (алгоритмами их обработки).

Этот механизм, связывает воедино код и данные, которыми код манипулирует, и защищает их от несанкционированного и неправильного использования. В ООЯ код и данные можно "упаковать" в "чёрный ящик" – **объект (object)**. Объект и есть средство инкапсуляции.

Объект представляет собой сложную переменную, тип которой определён программистом.

Внутри объекта код и данные могут быть закрытыми (private) или открытыми (public). Открытая часть объекта доступна извне из любой части программы и обеспечивает управляемое взаимодействие (интерфейс) между объектами.

Основным механизмом инкапсуляции является логическая абстракция **класс** – модель реального объекта, **объект**, в свою очередь, это конкретный экземпляр класса, его физическая реализация.

Класс – это тип данных, определяемый программистом, в котором объединяются структуры данных и функции их обработки.

Класс содержит константы и переменные, называемые полями, а также операции и функции выполняемые над данными – методы. Доступ к полям класса возможен только через вызов соответствующих методов.

Классы и объекты

Объявление класса:

```
class <имя_класса> {
[<спецификатор_доступа>:]
 <данные и функции>
……..……..
[<спецификатор_доступа>:]
  <данные и функции>
} [<список_объектов>];
```

спецификатор_доступа – это одно из 3-х ключевых слов:

- **public** – открытая секция класса, открывает доступ к функциям и данным этого класса из других частей программы, это интерфейс класса;
- **private** – закрытая секция класса, объявляет функции и данные закрытые от внешнего доступа и открытые только членам этого класса, действует по умолчанию;
- **protected** – защищённая секция класса, этот спецификатор используется только при наследовании классов.
- **список_объектов** – объявляет объекты класса, можно опускать и объявлять объекты позже.

- Методы, расположенные в открытой части (public), формируют интерфейс класса и могут вызываться другим (внешним) объектом.
- Доступ к закрытой секции класса (private) возможен только из собственных методов класса.
- Доступ к защищённой секции класса (protected) -- из собственных методов класса и из методов классов-потомков данного класса (см. наследование).

Количество одинаковых спецификаторов_доступане ограничено, порядок их следования – произвольный.

Классы

Пример: объявление (или спецификация) класса:

```
// Файл Book.h – спецификация класса CBook
#pragma once  /*Чтобы компилятор включал объявление типа только один раз при трансляции программы */
class CBook
{
private:     //  спецификатор доступа private действует по умолчанию,
            //  его  можно опускать
   char m_author [50];          // автор
   char *m_pTitle;              // указатель на название
   int m_year;                  // год издания
public:
// методы установки значений
    void setAuthor  const char*);
    void setTitle (const char*);
    void setYear (const int);
// методы возврата значений
      char* getAuthor  void ;
      char* getTitle (void);
      int getYear (void);
}
```

Классы

Данные и функции класса.

Данные класса называются: **поля** (данные-члены, компонентные данные).

Функции класса называются: **методы** (функции-члены, компонентные функции).

Поля и методы – это элементы класса.

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;
- могут быть описаны с модификатором static, но не как auto, extern и register.

Инициализация полей при описании не допускается.

Классы могут быть *глобальными* (объявленными вне любого блока) и *локальными* (объявленными внутри блока, например, функции или другого класса).

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется *конструктором* и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

Классы и объекты

Конкретные переменные типа класс называются экземплярами класса, или объектами. Время жизни и видимость объектов зависит от вида и места их описания и подчиняется общим правилам C++:

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается конструктор, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается деструктор.

Доступ к элементам объекта **аналогичен доступу к полям структуры. Для этого используются операция . (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель, например:**

```
int n = Vasia.get_ammo(); stado[5].draw;
cout << beavis->get_health();
```

Обратиться таким образом можно только к элементам со спецификаторам public. Получить или изменить значения элементов со спецификатором private можно только через обращение к соответствующим методам.

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Это обеспечивается передачей в функцию скрытого параметра this, в котором хранится константный указатель на вызвавший функцию объект. Указатель this неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (return this;) или ссылки (return *this;) на вызвавший объект.

Указатель this можно также применять для идентификации поля класса в том случае, когда его имя совпадает с именем формального параметра метода.

Классы и объекты

Объявление объектов класса

Описание объекта задает его тип (имя класса) и, возможно, необходимые для инициализации членов-данных значения. При объявлении объекта компилятор получает указание на создание переменной класса на основании заданного типа данных (класса).

Когда объект объявляется, то согласно описанию класса для объекта происходит выделение оперативной памяти, а также при указании значений данных осуществляется инициализация членов-данных указанными значениями. Всю эту работу делает специальный метод класса, называемый конструктором.

Основные форматы объявления объекта:

<имя_класса> <имя_объекта>;
<имя_класса> <имя_объекта> (список параметров);
<имя_класса> <имя_объект > (имя_объекта_копирования);

Пример объявления объектов класса:

- **CBook book, aBook [100];**
- **CBook obj ("Carrol L.", "Alice in Wonderland", 2000);**
- **CBook copy (obj);**

Здесь использованы все три формата для объявления объектов типа класса CBook Согласно первому, объявляется объект book и массив из 100 объектов aBook. По второму формату объявлен объект obj, и по третьему -- объект copy.

Классы

Вложенные классы

Один класс можно определить внутри другого – это вложенные классы. Поскольку объявление класса фактически определяет его область видимости, вложенные классы могут существовать только внутри области видимости внешнего класса. Вложенные используются редко, т.к. в языке C++ существует гибкий и мощный механизм наследования и создавать вложенные классы практически не требуется.

Локальные классы

Класс можно определить внутри функции. Пример:

```
#include <iostream>
using namespace std;
void f ();
int main() {
  f ();
// Класс myclass отсюда не виден
  return 0; }
     void f ()
     {  class myclass
        {
           int i;
            public:
              void put_i(int n) { i=n; }
               int get_i()   { return i; }
        } ob;
     ob.put_i(10);
     cout << b.get_i();  }
```

На локальные классы налагается несколько ограничений. Во-первых, все функции-члены должны определяться внутри объявления локального класса. Во-вторых, локальные классы не могут обращаться к локальным переменным, объявленным внутри функции, за исключением статических локальных переменных (static) и внешних переменных (extern). Однако они имеют доступ к именам типов и перечислениям, определенным во внешней функции. В-третьих, внутри локальных классов нельзя объявлять статические переменные.

Пространство имён

Пространства имен предназначены для локализации имен идентификаторов и предотвращения их конфликтов.

Среда программирования C++ работает с большим количеством переменных, функций и классов. Раньше все их имена пребывали в глобальном пространстве и нередко конфликтовали между собой. Чаше всего конфликты имен возникали, когда программа использовала несколько сторонних библиотек одновременно. Особенно это касается имен классов.

Введение ключевого слова namespace позволило решить эти проблемы.

Поскольку пространство имен позволяет локализовать область видимости объектов, объявленных внутри него, одно и то же имя, упомянутое в разных контекстах, больше не вызывает конфликтов. Наибольшую пользу это нововведение принесло стандартной библиотеке языка C++. До сих пор вся стандартная библиотека языка C++ находилась в глобальном пространстве имен (которое, собственно говоря, было единственным). Теперь стандартная библиотека определена внутри своего собственного пространства имен std, что намного уменьшает вероятность конфликтов.

Программист может создавать свои собственные пространства имен и самостоятельно локализовать имена, которые могут вызывать конфликты. Это особенно важно при разработке классов или библиотек функций.

Пространство имён

Ключевое слово namespace позволяет разделить глобальное пространство имен на декларативные области (declarative region). Пространство имен – это область видимости.

Общий вид объявления пространства имен таков:

```
namespace <имя_пространства_имён>
  {
    // Объявления
  }
```

Все, что объявлено в разделе namespace, находится внутри области видимости этого пространства имен.

Пространство имен должно объявляться вне всех остальных областей видимости за исключением того, что одно пространство имен может быть вложено в другое.
То есть вложенным пространство имен может быть только в другое пространство имен, но не в какую бы то ни было иную область видимости.
Это означает, что нельзя объявлять пространства имен, например, внутри функции.

Пространство имён

Директива using

Текст программы, в которой часто встречаются ссылки на элементы пространства имен станет малопонятным, поскольку будет переполнен квалификаторами и операторами области разрешения видимости.
Чтобы избежать этого, следует применять директиву using:

```
using namespace <имя_пространства_имён>;
using <имя_простр_имён> :: <член_простр_имён>;
```

В первом варианте параметр <name> задает название пространства имен. Все элементы этого пространства становятся частью текущей области видимости и могут использоваться без указания квалификатора.
Во втором варианте в область видимости включается только конкретный элемент пространства имен.

Если в программе, например, объявлено пространство имен CounterNameSpace, то можно применить следующие операторы:

```
using CounterNameSpace::lowerbound;   // Видимым является
                                      // только член lowerbound.
lowerbound =10;          // Правильный оператор, поскольку переменная
                         // lowerbound является видимой.
using namespace CounterNameSpace;  //Все члены видимы.
 upperbound = 100;       // Правильный оператор, поскольку все члены
                         // являются видимыми.
```

ООП. Наследование

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства.

При большом количестве никак не связанных классов управлять ими становится невозможным. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

Класс лежащий в основе иерархии, называется базовый (base class) (предок, суперкласс), а класс, наследующий свойства базового класса, -- производным (derived class) (наследник, подкласс). Производные классы, в свою очередь, могут быть базовыми по отношению к другим классам.

Объявление наследования класса:-

```
class <имя_производного_класса> :
 <уровень_доступа  имя_базового_класса>
{
<тело_класса>
} ;
```

уровень_доступа – это public, private или protected. По умолчанию для производного класса private, а для производной структуры – public. Член класса со спецификатором protected недоступен вне класса, но может наследоваться производным классом.

ООП. Наследование

Пример

```
#include <iostream.h>
class building
{   int rooms;       int floors;      int area;
 public:
    void set_rooms (int num);
    int get_rooms();
    void set_floors (int num);
    int get_floors();
    void set_area (int num);
    int get_area();   };
// Класс house – производный от building
class house : public building
{  int bedrooms;  int baths;
 public:
    void set_bedrooms (int num);
    int get_bedrooms();
    void set_baths (int num);
    int get_baths();  };
// Класс school –  также производный от building
class school : public building
{  int classrooms;  int offices;
 public:
    void set_classrooms (int num);
    int get_classrooms();
    void set_offices (int num);
    int get_offices();  };
void building :: set_rooms (int num)
    { rooms = num;  }
void building :: set_floors (int num)
    { floors = num;  }
```

// см. продолжение

// продолжение

```
void building :: set_area (int num)
    { area = num;  }
int building :: get_rooms ()
    { return rooms;  }
int building :: get_rooms ()
    { return rooms;  }
int building :: get_floors ()
    { return floors;  }
int building :: get_area ()
   { return area;  }
void house :: set_bedrooms (int num)
   { bedrooms = num;   }
void house :: set_baths (int num)
   { baths= num;   }
int house :: get_bedrooms ()
   { return bedrooms;  }
int house :: get_baths ()
   { return baths;  }
void school :: set_classrooms (int num)
   { classrooms= num;   }
void school :: set_offices (int num)
   {offices= num;   }
int school :: get_classrooms ()
   { return classrooms;  }
int school :: get_offices  ()
   { return offices ;  }
```

// см. продолжение

ООП. Наследование

Пример

```
// продолжение
int main ()
{
  house h;
  school s;
h.set_rooms (12);
h.set_floors (3);
h.set_area (4500);
h.set_bedrooms (5);
h.set_baths (3);
  cout << "В доме " << h.get_bedrooms ();
  cout << "  спален\n";
s.set_rooms (200);
s.set_classrooms (180);
s.set_offices (5);
s.set_area (25000);
  cout << "В школе" << s.get_classrooms ();
  cout << "  кабинетов\n";
  cout <<"Её площадь равна  " << s.get.area ();
   return 0;
}
```

Вывод на экран:

В доме 5 спален

В школе 180 кабинетов

Её площадь равна 25000

Замечания:

1. **Открытые члены класса building становятся открытыми членами производных классов house и school.**
2. **НО, методы (функции-члены) классов house и school *не имеют доступа* к закрытым членам класса building.**

То есть, наследование не нарушает инкапсуляцию.

ООП. Наследование

Простое (или одиночное) наследование – это наследование, при котором производный класс имеет только одного родителя.

В Примере на предыдущих слайдах как раз реализовано простое наследование. Формально наследование одного класса от другого можно задать следующей конструкций:

```
class имя_класса_потомка: [модификатор_доступа]
                          имя_базового_класса
  { тело__класса }
```

Класс-потомок наследует структуру (все элементы данных) и поведение (все методы) базового класса. Модификатор доступа определяет доступность элементов базового класса в классе-наследнике. Этот модификатор мы будем называть **модификатором наследования**.

Если в качестве модификатора наследования записано слово **public**, то такое **наследование открытое**. При использовании модификатора **protected** – **защищенное наследование**, а **private** означает **закрытое наследование**.

ООП. Наследование

Множественное наследование отличается от простого (одиночного) наличием нескольких базовых классов:

```
class A {}; class B {};

class D: public A, public B
```

Базовые классы перечисляются через запятую; количество их стандартом не ограничивается. Модификатор наследования для каждого базового класса может быть разный: можно от одного класса наследовать открыто, а от другого – закрыто.

При множественном наследовании выполняется все то же самое, что и при одиночном, то есть класс-потомок наследует структуру (все элементы данных) и поведение (все методы) всех базовых классов.

ООП. Наследование

Конструкторы при наследовании

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы.

Порядок вызова конструкторов определяется приведенными ниже правилами:

- **Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров).**
- **Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.**
- **В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.**

ВНИМАНИЕ!

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации.

ООП. Наследование

Деструкторы при наследовании

Правила наследования деструкторов

- Деструкторы **не наследуются**, и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.
- В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.
- Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем – деструкторы элементов класса, а потом деструктор базового класса.

ООП. Полиморфизм.

Полиморфизм – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы:
один интерфейс, несколько методов.

Полиморфизм в ООП реализуется с помощью нескольких инструментов:

- Два вида перегрузки: перегрузка функций и перегрузка операторов;
- Шаблоны: шаблоны классов и шаблоны функций;
- Переназначение функций при наследовании: виртуальные функции и абстрактные классы.

Пример полиморфизма первой разновидности −перегрузка функций, когда из нескольких заданных вариантов функции выбирается наиболее подходящая по соответствию ее прототипа передаваемым параметрам.

С перегрузкой функций тесно связан механизм перегрузки операторов (операций). Перегруженный оператор сохраняет своё первоначальное предназначение, Просто набор типов данных, к которым его можно применять, расширяется.

Например, в классе stack можно перегрузить оператор "+" для заталкивания элемента в стек, а оператор "-" для выталкивания элемента из стека.

Второй пример – использование шаблонов функций (и шаблонов классов), когда один и тот же алгоритм видоизменяется в соответствии с заданным типом данных. переданным в качестве параметра при вызове. В этой разновидности полиморфизма создаётся шаблон алгоритма с обобщённым(-ми) типами данных некоторых переменных, которые конкретизируются при вызове в программе.

Третья разновидность полиморфизма – виртуальные функции - используется в схеме наследования и позволяет определить в классе-наследнике другой алгоритм выполнения одноимённого члена-метода (функции).

ООП. Полиморфизм.
Перегрузка функций

Перегрузка функций (function overloading) – это использование одного имени для нескольких функций, отличающихся либо другими типами данных параметров, либо другим количеством параметров.
Перегрузка функций является особенностью языка C++, которой нет в языке C, это одна из разновидностей полиморфизма.

Основные концепции перегрузки функций:

- Перегрузка функций предоставляет несколько "взглядов" на одну и ту же функцию внутри вашей программы.
- Для перегрузки функций просто надо определить несколько функций с одним и тем же именем, которые отличаются только количеством и/или типом параметров.
- Компилятор C++ определит, какую функцию следует вызвать, основываясь на количестве и типе передаваемых (фактических) параметров.
- Перегрузка функций упрощает программирование, позволяя программистам работать только с одним именем функции.

Правила описания перегруженных функций:

- Перегруженные функции должны находиться в одной области видимости,
- Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать,
- Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или использованием ссылки (например, int и const int или int и int&).

ООП. Полиморфизм.
Перегрузка функций

ПРИМЕР: надо сделать функции поиска в базе данных, каждая функции имеет осмысленное имя:

```
int Search_int(int Num);
int Search_long(long Num);
int Search_float(float Num);
int Search_double(double Num);
```

Но гораздо проще дать всем этим функциям одно имя для поиска всех типов данных. Например:

```
int Search(int Num);
int Search(long Num);
int Search(float Num);
int Search(double Num);
```

Заметим - имена функций одинаковы, отличие только в типах аргументов. Цель перегрузки функций состоит в том, чтобы функции с одним именем ВЫПОЛНЯЛИСЬ ПО-РАЗНОМУ (и возможно возвращали разные значения) при обращении к ним с разными по типам и количеству параметрами. В языке C++ функции могут иметь одинаковые имена до тех пор, пока они значимо отличаются хотя бы одним параметром. Если значимого различия нет – компилятор предупредит о возникшей неопределенности.

Необходимо отличать перегрузку и переопределение функции.

Если функции возвращают одинаковый тип и список параметров у них абсолютно одинаковый, то второе объявление функции будет обработано как повторное определение. Если списки параметров двух функций абсолютно одинаковы, но отличаются только типы возврата, то второе объявление - ошибка:

```
int  Search(int Num);
long Search(int Num);  //ошибка!
```

ООП. Полиморфизм. Перегрузка функций

Как компилятор решает, какую именно функцию надо использовать для данного вызова (запроса).

Компилятор находит соответствие автоматически, сравнивая фактические параметры, используемые в запросе, с параметрами, предлагаемыми каждой функцией из набора перегруженных функций. Рассмотрим набор функций и один вызов функции:

```
void calc();
void calc(int);
void calc(int, int);
void calc(double, double=1.2345);
 calc(5.4321);  //вызвана будет функция void calc(double, double)
```

Как в этом случае компилятор ведет поиск соответствия функции?

Сначала определяются функции-кандидаты на проверку соответствия. Функция-кандидат должна иметь то же имя, что и вызываемая функция, и ее объявление должно быть видимым в точке запроса.

Затем определяются "жизнеспособные" функции. Они должны или иметь то же самое количество параметров что и в запросе, или тип каждого параметра должен соответствовать параметру в запросе (или быть конвертируемым типом).

Для нашего запроса calc(5.4321), мы можем сразу выкинуть функции-кандидаты calc() и calc(int, int). Запрос имеет только один параметр, а эти функции имеют нуль и два параметра, соответственно. calc(int) - вполне "жизнеспособная" функция, потому что можно конвертировать тип параметра double к типу int. Функция calc(double, double) тоже "жизнеспособная", потому что заданный по умолчанию параметр обеспечивает "якобы недостающий" второй параметр функции, а первый параметр имеет тип double, который точно соответствует типу параметра в вызове.

Потом компилятор определяет, какая из найденных "жизнеспособных" функций имеет "явно лучшее" соответствие фактическим параметрам в запросе. Что понимать под "явно лучшим"? Идея состоит в том, что чем ближе типы параметров друг к другу, тем лучше соответствие. То есть, точное соответствие типа лучше чем соответствие, которое требует преобразования типа.

В нашем запросе calc(5.4321) - только один явный параметр, и он имеет тип double. Чтобы вызвать функцию calc(int), параметр должен быть преобразован в int. Функция calc(double, double) является более точным соответствием для этого параметра. И именно эту функцию использует компилятор.

ООП. Полиморфизм. Перегрузка функций

Примеры

```
// Различные типы данных параметров:
//   выводится на экран значение параметра

#include <iostream.h>
// прототипы функций
  int refunc(int i);
  double refunc(double i);

int main()
{
  cout  <<  refunc(10)  <<  " ";
  cout  << refunc(5.4);
return 0;
}

// Функция для целых типов данных
int  refunc(int i)
{  return i; }
// Функция для данных двойной точности
double  refunc(double  i)
{  return i; }
```

```
// Различное количество параметров:
// выводится при одном параметре  значение
// параметра, а при 2-х – произведение их

#include <iostream.h>
// прототипы функций
  int refunc(int i);
  int refunc(int i, int j);

int main()
{
  cout  <<  refunc(10)  <<  " ";
  cout  << refunc(5, 4);
return 0;
}

// Функция для одного параметра
int  refunc(int i)
{  return i; }
// Функция для двух паарметров
int  refunc(int i, int j)
{  return i*j; }
```

ООП. Полиморфизм.
Перегрузка операторов

Перегрузка операторов (operator overloading) – это изменение смысла оператора, при котором возможно одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Например, оператора плюс (+), который обычно используется для сложения, при использовании его с определенным классом меняет смысл.

Для объектов класса string оператор плюс (+) будет добавлять указанные символы к текущему содержимому строки, а оператор минус (-) будет удалять каждое вхождение указанного символа из строки.

Перегрузка операторов осуществляется с помощью операторных функций (operator function), которые определяют действия перегруженных операторов применительно к соответствующему классу. Операторные функции создаются с помощью ключевого слова **operator**. Операторные функции могут быть как членами класса, так и обычными функциями. Однако обычные операторные функции, как правило, объявляют дружественными по отношению к классу, для которого они перегружают оператор.

ООП. Полиморфизм.
Перегрузка операторов

Операторная функция-член имеет следующий вид:

<тип_возвращаемого_значения> <имя_класса> :: operator# (список-аргументов)
{
... // Операции
}

Обычно операторная функция возвращает объект класса, с которым она работает, однако тип возвращаемого значения может быть любым. Символ # заменяется перегружаемым оператором.

Например, если в классе перегружается оператор деления "/", операторная функция-член называется operator/.

При перегрузке унарного оператора список аргументов остается пустым.

При перегрузке бинарного оператора список аргументов содержит один параметр.

ООП. Полиморфизм. Перегрузка операторов

ПРИМЕР

Программа создает класс loc, в котором хранятся географические координаты: широта и долгота. В ней перегружается оператор "+".

```
#include <iostream>
using namespace std;
  class loc
   {
    int longitude, latitude;
      public:
       loc() {}
       loc(int lg,  int lt)
          {
           longitude = lg;
           latitude = lt;
          }
        void show()
         {
           cout << longitude << " ";
           cout << latitude << "\n";
         }
       loc operator+(loc op2);
};
```

см. продолжение

Продолжение

```
// Overload + for loc.
loc loc :: operator+(loc op2)
  {
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
   }

int main()
 {
  loc ob1(10,  20),  ob2( 5, 30);
  ob1.show(); // Выводит на экран числа 10 20
   ob2.show();  // Выводит на экран числа 5  30
   ob1 = ob1 + ob2;
   ob1.show();  // Выводит на экран числа 15 50
   return 0;
  }
```

ООП. Полиморфизм.
Перегрузка операторов

Ограничения на перегруженные операторы

Во-первых, нельзя изменить приоритет оператора.
Во-вторых, невозможно изменить количество операндов оператора. (Однако операнд можно игнорировать.)
В-третьих, операторную функцию нельзя вызывать с аргументами, значения которых заданы по умолчанию.

И, в заключение, нельзя перегружать следующие операторы:

- **оператор выбора члена класса .**
- **оператор селектора члена класса .***
- **оператор разрешения области видимости ::**
- **тернарный оператор – условная операция ?:**
- **статический оператор вычисления длины оператора в байтах sizeof**

Кроме перечисленных операторов, относящихся к синтаксису языка C++, не разрешается переопределять следующие операторы препроцессора:

- **оператор превращения в строку #**
- **оператор конкатенации ##**

Примечание
Операторы # и ## используются в директиве препроцессора #define, которую современный стандарт C++ применять не рекомендует. Однако следует помнить о невозможности их перегрузки для классов.

За исключением оператора "=", операторные функции наследуются производными классами. Однако в производном классе каждый из этих операторов снова можно перегрузить.

Шаблоны в C++

С помощью шаблонов (templates) можно создавать родовые (обобщённые) функции (generic functions) и родовые (обобщённые) классы (generic classes). В родовой функции или классе тип данных, с которыми функция или класс работают, задается в качестве параметра.

Шаблон представляет собой функцию (template function) или класс (template class), реализованные для одного или нескольких типов данных, которые не известны в момент написания кода. При использовании шаблона в качестве аргументов ему явно или неявно передаются конкретные типы данных. Поскольку шаблоны являются средствами языка, для них обеспечивается полная поддержка проверки типов и областей видимости.

Таким образом, шаблоны дают возможность создавать многократно используемые программы.

Поскольку шаблоны являются логическим развитием механизма макроподстановок, то данные о типах, например длина операндов, подставляется на этапе компиляции (точнее, даже еще раньше). На момент выполнения все длины операндов, размеры элементов массивов и прочие величины уже вычислены, так что процессор работает с хорошо оптимизированным прямолинейным кодом.

Шаблоны функций

Обобщенная функция (шаблон функции – template function) определяет универсальную совокупность операций, применимых к различным типам данных. Тип данных, с которыми работает функция, передается в качестве параметра.

Многие алгоритмы носят универсальный характер и не зависят от типа данных, которыми они оперируют. Например, для массивов целых и действительных чисел используется один алгоритм быстрой сортировки. С помощью обобщенной функции можно определить алгоритм независимо от типа данных. После этого компилятор автоматически генерирует правильный код, соответствующий конкретному типу. По существу, обобщенная функция автоматически перегружает саму себя.

Обобщенная функция объявляется с помощью ключевого слова template. Создается шаблон, который описывает действия функции и позволяет компилятору самому уточнять необходимые детали.

Определение шаблонной функции выглядит следующим образом:

```
template <class Ттип> <тип_возвращаемого_значения>
<имя_функции> (<список_параметров>)
{
// Тело функции
}
```

Здесь параметр Ттип – имя типа – задает тип данных, с которым работает функция. Этот параметр можно использовать и внутри функции, однако при создании конкретной версии обобщенной функции компилятор автоматически подставит вместо него фактический тип. Традиционно обобщенный тип задается с помощью ключевого слова class, хотя вместо него можно применять ключевое слово typename.

Шаблоны функций

Обобщенная функция, меняющая местами две переменные.

Поскольку процесс перестановки не зависит от типа переменных, его можно описать с помощью обобщенной функции.

```
// Пример шаблонной функции
#include <iostream>
using namespace std;
// Шаблон функции
template <class X>
        void swapargs(X &a, X &b)
{
 X temp;
 temp = a;   a = b;   b = temp;
}
```

см. продолжение

Строка
template <class X> void swapargs(X &a, X &b)
сообщает компилятору, что: во-первых, создается шаблон, во-вторых, начинается описание обобщенной функции.
Здесь параметр X задает обобщенный тип, который впоследствии будет заменен фактическим типом. После этой строки объявляется функция swapargs(), в которой переменные, подлежащие перестановке, имеют обобщенный тип X. В функции main() функция swapargs() вызывается для трех разных типов данных: int, double и char. Поскольку функция swapargs() является обобщенной, компилятор автоматически создает три ее версии: для перестановки целых и действительных чисел, а также символов.

```
int main()                    продолжение
{ int i=10, j=20;
   double x=10.1, y=23.3;
   char  a='x';  b='z';
cout << "Исходные значения i, j: " << i << '  '
                                  << j << '\n';
cout << "Исходные значения x, y: " << x <<
                          '  ' << y << '\n';
cout << "Исходные значения a, b:   " << a <<
                              '   '  << b << '\n';
swapargs  (i,  j);  // Перестановка целых чисел
swapargs(x, y);    // Перестановка действительных
                          // чисел
swapargsfa, b);   // Перестановка символов.
cout << "Переставленные значения i,  j:   "
             << i <<'  ' << j <<'\n';
cout << "Переставленные значения x, y:  "
             << x << ' ' << y <<'\n';
cout << "Переставленные значения a, b:   "
<< a <<  '   '  << b <<  ' \n';
return 0;
}
```

Шаблоны функций

Уточним понятия связанные с шаблонами.

Обобщенная функция (т.е. функция, объявленная с помощью ключевого слова **template**) называется также **шаблонной функцией** (**template function**). **Эти термины являются синонимами.**

Конкретная версия обобщенной функции, создаваемая компилятором, называется **специализацией** (**specialization**) или **генерируемой функцией** (**generated function**). **Процесс генерации конкретной функции** называется **конкретизацией** (**instantiation**). **Генерируемая функция является конкретным экземпляром обобщенной функции.**

Поскольку язык C++ не считает конец строки символом конца оператора, раздел template в определении обобщенной функции не обязан находиться в одной строке с именем функции.
Такую особенность иллюстрирует следующий пример:

```
template <class X>
    void swapargs(X &a,  X &b)
      {
       X temp;
       temp = a; a = b; b = temp;
      }
```

НО, в этом случае следует помнить, что между оператором template и началом определения обобщенной функции не должно быть других операторов. Например, следующий фрагмент ошибочен.

```
// Этот фрагмент содержит ошибку:
template <class X>
   int i;  // Ошибка
   void swapargs(X &a, X &b)
{
X temp;
temp = a; a = b; b = temp;
}
```

Т.е., спецификация template должна непосредственно предшествовать определению функции.

Шаблоны функций

Функция с двумя обобщенными типами

Используя список, элементы которого разделены запятыми, можно определить несколько обобщенных типов данных в операторе template.

Например, в следующей программе создается шаблонная функция, имеющая два обобщенных типа.

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

template <class type1,  class type2>
 void myfunc(type1  x,  type2  y)
{
cout  <<  x <<'   ' <<  y  <<  '\n';
}

int main()
 {
   myfunc(10, "Я люблю C++");
   myfunc(98.6,   19L);
  return 0;
}
```

Шаблоны функций

Ограничения на обобщенные функции

Обобщенные функции напоминают перегруженные, но на них налагаются еще более жесткие ограничения. При перегрузке внутри тела каждой функции можно выполнять разные операции. В то же время обобщенная функция должна выполнять одну и ту же универсальную операцию для всех версий, различаться могут лишь типы данных.

РЕЗЮМЕ

Шаблонная функция (template function) – это функция, полностью контролирующая соответствие типов данных, которые задаются ей как параметры.

Объявление шаблона функции повторяет определение обычной функции, но первой строкой в объявлении обязательно должна быть строка следующего вида:

template <class имя_типа1, …, class имя_типаN >

В угловых скобках < > после ключевого слова template указывается список формальных параметров шаблона. Имя_типа называют параметром типа, который представляет собой идентификатор типа и используется для определения типа параметра в заголовке шаблонной функции, типа возвращаемого функцией значения и типа переменных, объявляемых в теле функции. Каждый параметр типа должен иметь уникальный идентификатор, который может быть использован в разных шаблонах. Каждому параметру типа должно предшествовать ключевое слово class.

За строкой template следует строка со стандартным объявлением функции, где в списке параметров используются как параметры типа, так и любые другие допустимые базовые или производные типы данных.

Шаблонная функция может быть перегружена другим шаблоном или не шаблонной функцией с таким же именем, но с другим набором параметров.

Компилятор выбирает вариант функции, соответствующий вызову. Сначала подбирается функция, которая полностью соответствует по имени и типам параметров вызываемой функции. Если попытка заканчивается неудачей, подбирается шаблон, с помощью которого можно сгенерировать функцию с точным соответствием типов всех параметров и имени. Если же и эта попытка неудачна, выполняется подбор перегруженной функции.

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

Сортировка представляет собой типичный пример универсального алгоритма. Как правило, алгоритм сортировки совершенно не зависит от типа сортируемых данных.

Функция bubble (), предназначенная для сортировки данных методом пузырька, упорядочивает массив любого типа. Ей передается указатель на первый элемент массива и количество элементов в массиве. Несмотря на то что этот алгоритм сортировки является одним из самых медленных, он очень прост и нагляден.

```
#include <iostream>
using namespace std;
  template <class X>
                    void bubble (
                            X *items,  // Указатель на упорядочиваемый массив
                            int count)  // Количество элементов массива
   {
    register  int a, b;
    X t;
     for(a=1; a<count; a++)
       for(b=count-1; b>=a; b--)
          if(items[b-1]  > items[b])
             // Перестановка элементов
             {  t = items[b-1];  items[b-1] = items[b];  items[b] = t; }
    }
```

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

```
int main()
 {
   int iarray[7]  = {7,  5,  4,  3,  9,  8,  6} ;
   double darray[5]  = {4.3,  2.5,  -0.9,  100.2, 3.0};
   int i;
  cout << "Неупорядоченный массив целых чисел: ";
    for(i=0;  i<7; i++)
      cout << iarray[i] <<  '  ';
    cout << endl;
  cout << "Неупорядоченный массив действительных чисел: ";
    for(i=0;  i<5; i++)
      cout << darray[i] <<   ' ';
    cout << endl;
  bubble(iarray, 7);    bubble(darray,  5);
  cout « "Упорядоченный массив целых чисел: ";
    for(i=0;   i<7; i++)
      cout << iarray[i] << '   ';
    cout << endl;
   cout << "Упорядоченный массив действительных чисел: ";
     for(i=0;  i<5; i++)
       cout << darray[i]  <<  '  ';
     cout << endl;
 return 0;
 }
```

Шаблоны функций

Применение обобщенных функций. Обобщенная сортировка

Программа выводит на экран следующие результаты:

- **Неупорядоченный массив целых чисел: 7 5 4 3 9 8 6**
- **Неупорядоченный массив действительных чисел: 4.3 2.5 -0.9 100.2 3**
- **Упорядоченный массив целых чисел: 3 4 5 6 7 8 9**
- **Упорядоченный массив действительных чисел: -0.9 2.5 3 4.3 100.2**

Как видим, в программе создаются два массива, имеющие соответственно типы int и double.

Поскольку функция bubble() является шаблонной, она автоматически перегружается для каждого из этих типов.

Эту функцию можно применить для сортировки данных другого типа, даже объектов какого-нибудь класса. Компилятор автоматически создаст соответствующую версию функции для нового типа.

Виртуальные функции

При наследовании часто бывает необходимо, чтобы поведение некоторых методов базового класса и классов-наследников различалось. Можно переопределить соответствующие методы в производном классе. Однако тут возникает одна проблема, которую лучше рассмотреть на простом примере:

```
#include <iostream>
using namespace std;
class Base      // базовый класс
{ public:
int  f(const int &d) //метод базового класса
{ return 2*d; }
int CallFunction(const int &d)
{return f(d)+l;} // вызов метода базового класса
};
Class Derived: public Base // производный класс
{   public:  // CallFunction наследуется
int f(const int &d) // метод f переопределяется
{ return d*d; }
};
int main()
{     Base a;  // объект базового класса
cout<<a.CallFunction(5)<<endl; //получаем 11
Derived b;  // объект производного класса
cout << b.CallFunction(5)<< endl; // какой
        // метод f вызывается?
return 0;
```

В базовом классе определены два метода – f() и) CallFunction(), причем во втором методе вызывается первый. В классе-наследнике метод f() переопределен, а метод CallFunction() унаследован. Очевидно, метод f() переопределяется для того, чтобы объекты базового класса и класса-наследника вели себя по-разному.

Объявляя объект b типа Derived, программист, естественно, ожидает получить результат 5*5 + 1 = 26 – для этого и переопределялся метод f().

Однако на экран, как и для объекта a типа Base, выводится число 11, которое, очевидно, вычисляется как 2*5+1 = 11.

Несмотря на переопределение метода f() в классе-наследнике, в унаследованной функции CallFunction() вызывается «родная» функция f(), определенная в базовом классе!

При трансляции класса Base компилятор ничего не знает о классах-наследниках, поэтому он не может предполагать, что метод f() будет переопределен в классе Derived.

Его естественное поведение – «прочно» связать вызов f() с телом метода класса Base.

Виртуальные функции

Чтобы добиться разного поведения в зависимости от типа, необходимо объявить функцию-метод виртуальной; в C++ это делается с помощью ключевого слова virtual.

Виртуальная функция (virtual function) – это функция-член, объявленная в базовом классе и переопределенная в производном.

Ключевое слово virtual указывается до объявления функции в базовом классе.

Производный класс переопределяет эту функцию, приспосабливая ее для своих нужд. По существу, виртуальная функция реализует принцип "один интерфейс, несколько методов", лежащий в основе полиморфизма.

Виртуальная функция в базовом классе определяет вид интерфейса, т.е. способ вызова этой функции. Каждое переопределение виртуальной функции в производном классе реализует операции, присущие лишь данному классу. Иначе говоря, переопределение виртуальной функции создает конкретный метод (specific method).

При обычном вызове виртуальные функции ничем не отличаются от остальных функций-членов. Особые свойства виртуальных функций проявляются при их вызове с помощью указателей. Указатели на объекты базового класса можно использовать для ссылки на объекты производных классов. Если указатель на объект базового класса устанавливается на объект производного класса, содержащий виртуальную функцию, выбор требуемой функции основывается на типе объекта, на который ссылается указатель, причем этот выбор осуществляется в ходе выполнения программы. Таким образом, если указатель ссылается на объекты разных типов, то будут вызваны разные виртуальные функции. Это относится и к ссылкам на объекты базового класса.

Виртуальные функции

Таким образом, объявление метода f() в базовом и производном классах должно быть таким:

```
virtual  int  f(const int &d)       // в базовом классе
   { return 2*d; }
 virtual int f(const int &d)      // в производном классе
   { return d*d: }
```

После этого для объектов базового и производного классов мы получаем разные результаты: 11 и 26.

Аналогично в объявление метода print() тоже должно начинаться со слова virtual:

```
virtual      void  print() const          // в базовом классе
     { cout << "Clock!" << endl: }
virtual      void  print() const          // в производном классе
     { cout << "Alarm!" << endl; }
```

После этого вызов settime() с параметром базового класса обеспечит вывод на экран слова «Clock», а с параметром производного класса – слова «Alarm». И при вызове по указателю наблюдается та же картина.

Вообще-то ключевое слово virtual достаточно написать только один раз – в объявлении функции базового класса. Определение можно писать без слова virtual – все равно функция будет считаться виртуальной. Однако лучше всегда это делать явным образом, чтобы всегда по тексту было видно, что функция является виртуальной.

Для виртуальных функций обеспечивается не статическое, а динамическое (позднее, отложенное) связывание, которое реализуется во время выполнения программы.

В C++ реализованы два типа полиморфизма:

- **статический полиморфизм, или полиморфизм времени компиляции (compile-time polymorphism), осуществляется за счет перегрузки и шаблонов функций;**
- **динамический полиморфизм, или полиморфизм времени выполнения (runtime polymorphism), реализуется виртуальными функциями.**

Виртуальные функции

С перегрузкой функций "разбирается" компилятор, правильно подбирая вариант функции в той или иной ситуации. Полиморфизм шаблонных функций тоже реализуется на этапе компиляции. Естественно, выбор осуществляется статически.

Выбор же виртуальной функции происходит динамически – при выполнении программы. Класс, включающий в себя виртуальные функции, называется **полиморфным.**

Правила описания и использования виртуальных функций-методов:

1) Виртуальная функция может быть только методом класса,
2) Любую перегружаемую операцию-метод класса можно сделать виртуальной, например, операцию присваивания или операцию преобразования типа,
3) Виртуальная функция, как и сама виртуальность, наследуется,
4) Виртуальная функция может быть константной,
5) Если в базовом классе определена виртуальная функция, то метод производного класса с такими же именем и прототипом (включая тип возвращаемого значения и константность метода) автоматически является виртуальным (слово virtual указывать необязательно) и замещает функцию-метод базового класса,
6) Конструкторы не могут быть виртуальными,
7) Статические методы не могут быть виртуальными,
8) Деструкторы могут (чаще – должны) быть виртуальными – это гарантирует корректный возврат памяти через указатель базового класса.

Виртуальные функции

Объясним виртуальные функции ещё раз по-другому:

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса, например:

```
monstr *p:              // Описание указателя на базовый класс
p = new daemon;         // Указатель ссылается на объект производного класса
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается,

поэтому при выполнении оператора, например, `p->draw(l, 1. 1. 1);`

будет вызван метод класса monstr, а не класса daemon, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием.

Чтобы вызвать метод класса daemon, можно использовать явное преобразование типа указателя: `(daemon * p)->draw(l, 1, 1. 1);`

Но это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример – связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм позднего связывания, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов.

Для определения виртуального метода используется спецификатор virtual, например:

```
virtual  void  draw(int x. int y, int scale, int position);
```

Виртуальные функции

Правила описания и использования виртуальных методов:

- **Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров – обычным,**
- **Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя,**
- **Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости,**
- **Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный,**
- **Если в классе вводится описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.**

Чисто виртуальный метод содержит признак - 0 вместо тела, например:

```
virtual void f(int) = 0;
```

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Виртуальные функции

Сравнение раннего и позднего связывания

Раннее связывание (early binding) означает события, происходящие на этапе компиляции. Раннее связывание означает, что на этапе компиляции известна вся информация, позволяющая выбрать вызываемую функцию. (Иначе говоря, объект и вызов функции связываются друг с другом на этапе компиляции.) Примерами раннего связывания являются обычные вызовы функций (включая стандартные библиотечные функции), вызовы перегруженных функции и операторов. Основное преимущество раннего связывания – эффективность. Поскольку вся информация о вызываемой функции на этапе компиляции уже известна, сам вызов функции происходит очень быстро.

Позднее связывание (late binding) является антиподом раннего. В языке C++ позднее связывание означает, что фактический выбор вызываемой функции осуществляется только в ходе выполнения программы. Основным средством позднего связывания являются виртуальные функции. Выбор вызываемой виртуальной функции зависит от типа указателя или ссылки, с помощью которых она вызывается. Поскольку в большинстве случаев на этапе компиляции эта информация отсутствует, связывание объекта и вызова функции откладывается до выполнения программы. Основное преимущество позднего связывания – гибкость. В отличие от раннего, позднее связывание позволяет создавать программу, реагирующую на события, происходящие в ходе ее выполнения, без использования большого количества кода, предусматривающего всевозможные варианты. Однако позднее связывание может замедлить работу программы.

Виртуальные функции

Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным (abstract class). Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов – объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении (т.к. чисто виртуальные функции не имеют определения).

Несмотря на то что объекты абстрактного класса не существуют, можно создать указатели и ссылки на абстрактный класс. Это позволяет применять абстрактные классы для поддержки динамического полиморфизма и выбирать соответствующую виртуальную функцию в зависимости от типа указателя или ссылки.

При определении абстрактного класса необходимо иметь в виду следующее:

- абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
- допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
- если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого тина в пределах одной иерархии.

Виртуальные функции

Практическое занятие: Вызов виртуальной функции с помощью указателя на объект базового класса

```
#include <iostream>
using namespace std;
class base {   public:
  virtual void vfunc() {
    cout<<"Функция vfunc() из класса base.\n"; }  };
class derived1 : public base {    public:
  void vfunc() {
    cout << "Функция vfunc() из класса derived1.\n";
                    }    };
class derived2 : public base {    public:
  void vfunc() {
    cout << "Функция vfunc() из класса derived2.\n";
                    }    };
int main()
{   base *p, b;  derived1 d1;  derived2  d2;
  // Указатель на объект базового класса
  p = &b;
 p->vfunc(); // Вызов функции vfunc() из класса base
  // Указатель на объект класса derivedl
  p = &d1;
  p->vfunc(); // Вызов функции vfunc() из класса
                          derivedl
  // Указатель на объект класса derived2
  p = &d2;
  p->vfunc();  // Вызов функции vfunc() из класса
                          derived2
  return 0;  }
```

Эта программа выводит на экран следующие строки:

- Функция vfunc() из класса base.
- Функция vfunc() из класса derived1.
- Функция vfunc() из класса derived2.

Как показывает эта программа, внутри класса base объявлена виртуальная функция vfunc(). Обратите внимание на ключевое слово virtual в объявлении функции. При переопределении функции vfunc() в классах derived1 и derived2 ключевое слово virtual не требуется. (Однако его использование не является ошибкой, просто оно не обязательно.)

В данной программе классы derived1 и derived2 являются производными от класса base. Внутри каждого из этих классов функция vfunc() переопределя-ется заново в соответствии с новым предназначением. В программе main() объявлены четыре переменные:

Имя	Тип
p	Указатель на базовый класс
b	Объект базового класса
d1	Объект класса derivedl
d2	Объект класса derived2

Виртуальные функции

Практическое занятие:

Вызов виртуальной функции с помощью указателя на объект базового класса

Указателю p присваивается адрес объекта b, а функция vfunc() вызывается с помощью указателя p. Поскольку указатель p ссылается на объект класса base, выполняется вариант функции vfunc() из базового класса.
Затем указателю p присваивается адрес объекта d1, и функция vfunc() снова вызывается с его помощью. На этот раз указатель p ссылается на объект класса derived1. Следовательно, вызывается функция derived1::vfunc(). В результате указателю p присваивается адрес объекта d2, поэтому выражение p->vfunc() приводит к вызову функции vfunc() из класса derived2. Принципиально важно, что вариант вызываемой функции определяется типом объекта, на который ссылается указатель p. Кроме того, выбор происходит в ходе выполнения программы, что обеспечивает основу динамического полиморфизма.

Виртуальную функцию можно вызывать обычным способом, используя имя объекта и оператор однако полиморфизм достигается только при обращении к ней через указатель.
Например, следующий фрагмент программы является совершенно правильным:

```
d2.vfunc();  // Вызывается функция vfunc() из класса derived2.
```

Несмотря на то что такой вызов виртуальной функции ошибкой не является, никаких преимуществ он не предоставляет.

Виртуальные функции

На первый взгляд, переопределение виртуальной функции в производном классе мало отличается от обычной перегрузки функций. Однако это не так, и термин перегрузка неприменим к переопределению виртуальных функций по нескольким причинам.

Наиболее важное отличие заключается в том, что прототип переопределяемой виртуальной функции должен точно совпадать с прототипом, определенным в базовом классе.

Этим виртуальные функции отличаются от перегруженных, которые отличаются типами и количеством параметров. (Фактически при перегрузке функций типы и количество их параметров должны отличаться! Именно эти отличия позволяют компилятору выбирать правильный вариант перегруженной функции.)

При переопределении виртуальной функции все аспекты их прототипов должны быть одинаковыми. Если не соблюдать это правило, компилятор будет считать эти функции просто перегруженными, а их виртуальная природа будет потеряна.

Второе важное ограничение заключается в том, что виртуальные функции не могут быть статическими членами классов. Кроме того, они не могут быть дружественными функциями. И, наконец, конструкторы не могут быть виртуальными, хотя на деструкторы это ограничение не распространяется.

Из-за перечисленных ограничений для переопределения виртуальной функции в производном классе используется термин замещение (overriding).

Динамическая идентификация типа

В языке C++ для поддержки объектно-ориентированного программирования используется **динамическая идентификация типа (RTTI – Run-Time Type Identification)** и четыре дополнительных **оператора приведения типов** (плюс 5-й оператор приведения типов cast из C).

Система RTTI позволяет идентифицировать тип объекта при выполнении программы. Дополнительные операторы приведения типов обеспечивают более безопасный способ приведения. Поскольку один из этих операторов приведения (оператор dynamic_cast) непосредственно связан с системой динамической идентификации типов имеет смысл рассмотреть их вместе.

Механизм динамической идентификации типа состоит из трех составляющих:

- оператора динамического преобразования типа dynamic_cast<>;
- оператора идентификации точного типа объекта typeid();
- класса type_info.

Динамическая идентификация типов не характерна для таких неполиморфных языков, как язык C. В этих языках пет необходимости определять тип объекта при выполнении программы, поскольку тип каждого объекта известен еще на этапе компиляции. Однако в полиморфных языках, таких как C++, возникают ситуации, в которых тип объекта на этапе компиляции неизвестен, поскольку природа этого объекта уточняется только в ходе выполнения программы. Язык C++ реализует полиморфизм с помощью иерархий классов, виртуальных функций и указателей на объекты базового класса. Поскольку указатели на объекты базового класса могут ссылаться и на объекты производных классов, не всегда можно предсказать, на объект какого типа они будут ссылаться в тот или иной момент. Эту идентификацию приходится осуществлять в ходе выполнения программы.

Динамическая идентификация типа

Для идентификации типа объекта используется оператор typeid, определенный в заголовке #include <typeinfo>.

Его формат: typeid (object)

Здесь, параметр object является объектом, тип которого мы хотим идентифицировать. Он может иметь любой тип, в том числе встроенный или определенный пользователем.

Оператор typeid возвращает ссылку на объект типа type_info, описывающий тип объекта. Стандарт "ISO/IEC 14882:2003(E), Programming languages - C++" определяет класс type_info так:

```
class type_info
 {
  public:
    virtual ?type_info();
     bool operator == (const type_info& rhs) const;
     bool operator != (const type_info& rhs) const;
     bool before(const type_info& rhs) const;
    const char* name() const;
  private:
   type_info(const type_info& rhs);
   type_info& operator=(const type_info& rhs);
 };
```

Как видно, объекты типа type_info невозможно ни создать, ни скопировать – конструкторы и операция присваивания закрыты.

Объекты типа type_info можно сравнивать между собой, используя перегруженные операторы "==" и "!=", – и это основные операции, которые используются совместно с оператором typeid(); функция name() возвращает указатель на имя заданного типа.

Метод before() позволяет сортировать информацию о типе type_info. Нет никакой связи между отношениями упорядочения, определяемыми before(), и отношениями наследования. Если вызывающий объект предшествует объекту, использованному как параметр, функция before() возвращает значение true.

Динамическая идентификация типа

Пример применения typeid к иерархии полиморфных классов

Самое важное свойство оператора typeid проявляется, когда он применяется к указателю на объект базового класса. В этом случае он автоматически возвращает тип реального объекта, на который ссылается указатель, причем этот объект может быть экземпляром как базового, так и производного классов. (Напомним, что указатель на объект базового класса может ссылаться на объекты производного класса.)

Т.о, в ходе выполнения программы, используя оператор typeid, можно идентифицировать тип объекта, на который ссылается указатель базового класса.

Результаты работы этой программы приведены ниже.

- Указатель p ссылается на объект типа class Mammal,
- Указатель p ссылается на объект типа class Cat,
- Указатель p ссылается на объект типа class Platypus.

```
#include <iostream>
#include <typeinfo>
using namespace std;
 class Mammal {public: //класс Mammal – полиморфный
   virtual bool lays_eggs() { return false; }
   // ...   };
class Cat: public Mammal { public:   /* ... */  };
class Platypus: public Mammal { public:
 bool lays_eggs() { return true; } /* ... */  };

int main()
{   Mammal *p, AnyMammal;
    Cat cat;
    Platypus platypus;
  p = &AnyMammal;
  cout << "Указатель p ссылается на объект типа ";
  cout << typeid(*p).name() << endl;
  p = &cat;
   cout << "Указатель p ссылается на объект типа ";
   cout << typeid(*p).name() << endl;
  p = &platypus;
   cout << "Указатель p ссылается на объект типа ";
   cout << typeid(*p).name() << endl;
 return 0;
}
```

Динамическая идентификация типа

Разбор примера применения typeid к иерархии полиморфных классов

Если оператор typeid применяется к указателю на объект полиморфного базового класса, тип объекта, на который ссылается указатель, можно определить в ходе выполнения программы. В любом случае, если оператор typeid применяется к указателю на объект неполиморфного базового класса, возвращается базовый тип указателя. Иначе говоря, невозможно определить, на объект какого типа ссылается этот указатель на самом деле.

Закомментируем, например, ключевое слово virtual, стоящее перед именем функции lays_egg() в классе Mammal, перекомпилируем программу и запустим ее вновь:

Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal
Указатель p ссылается на объект типа class Mammal

Класс Mammal больше не является полиморфным, и каждый объект теперь имеет тип Mammal, т.е. тип указателя p.

Поскольку оператор typeid обычно применяется к разыменованным указателям, следует предусмотреть обработку исключительной ситуации bad_typeid, которая может возникнуть, если указатель будет нулевым.

Оператор typeid имеет вторую форму, получающую в качестве аргумента имя типа. Ее общий вид приведен ниже:

typeid(<имя_типа>)

Например, следующий оператор совершенно правилен: `cout << typeid(int).name();`

Данная форма оператора typeid позволяет получить объект класса type_info, описывающий заданный тип. Благодаря этому ее можно применять в операторах сравнения типов.

Например:

```
void WhatMammal(Mammal  &ob)
{  cout << "Параметр ob ссылается на объект типа ";
   cout << typeid(ob).name()  << endl;
   if(typeid(ob)  == typeid(Cat))
       cout << "Кошки боятся воды.\n";
}
```

Динамическая идентификация типа

Применение динамической идентификации типа

```
#include <iostream>
using namespace std;
 class Mammal {  public:
    virtual bool lays_eggs() {return false;}
    // ...   };
 class Cat: public Mammal { public: /* … /* };
 class Platypus: public Mammal { public:
     bool lays_eggs() { return true; }    /* ... */ };
     class Dog: public Mammal { public: /* … /* };
// Фабрика объектов, производных от класса Mammal
 Mammal *factory()
  { switch(rand() % 3 ) { case 0: return new Dog;
    case 1: return new Cat; case 2: return new Platypus; }
   return 0;      }
int main()
 {  Mammal *ptr;  // Указатель на базовый класс
       int i;     int c=0, d=0, p=0;
 // Создаем и подсчитываем объекты
 for(i=0; i<10; i++)   // Создаем и подсчитываем объекты
    {  ptr = factory(); // Создаем объект
       cout<<"Тип объекта: "<< typeid(*ptr).name();
       cout<< endl;
   // Подсчёт
   if(typeid(*ptr) == typeid(Dog))  d++;
   if(typeid(*ptr) == typeid(Cat))   c++;
   if(typeid(*ptr) == typeid(Platypus))  p++;    }
 cout << endl;
 cout << " Созданные животные:\n";
 cout << " Собаки: " << d << endl;
 cout << " Кошки: " << c << endl;
 cout << " Утконосы: " << p << endl;
return 0;
}
```

Функция factory() создает объекты различных классов, производных от класса Mammal. (Функции, создающие объекты, иногда называются фабриками объектов (object factory)). Конкретный тип создаваемого объекта задается функцией rand(), которая представляет собой генератор случайных чисел в языке C++. Т.е., тип создаваемого объекта заранее не известен. Программа создаст 10 объектов и подсчитывает количество объектов каждого типа. Поскольку все объекты генерируются функцией factory(), для идентификации фактического типа объекта используется оператор typeid.

Результаты работы этой программы приведены ниже.

```
Тип объекта:  class Platypus
Тип объекта:  class Platypus
Тип объекта:  class Cat
Тип объекта:  class Cat
Тип объекта:  class Platypus
Тип объекта:  class Cat
Тип объекта:  class Dog
Тип объекта:  class Dos
Тип объекта:  class Cat
Тип объекта:  class Platypus
 Созданные животные:
Собаки: 2
Кошки: 4
Утконосы: 4
```

Динамическая идентификация типа

Применение оператора typeid к шаблонным классам

```
#include <iostream>
using namespace std;
  template <class T> class myclass {
    T a;
     public:
       myclass(T i) { a = i; }      // ...
   };
int main()
{  myclass<int> o1(10), o2(9);
   myclass<double> o3(7.2);
  cout << "Тип объекта o1: ";
  cout << typeid(o1).name() << endl;
  cout << "Тип объекта o2: ";
  cout << typeid(o2).name() << endl;
  cout <<  "Тип объекта o3: ";
  cout << typeid(o3).name() << endl;
  cout << endl;
   if(typeid(o1) == typeid(o2))
          cout << "Объекты o1 и o2 имеют
одинаковый тип\n";
   if(typeid(o1) == typeid(o3))
    cout << "Ошибка\n";
  else
    cout << "Объекты o1 и o3 имеют разные
типы\n";
  return 0;
}
```

Тип объекта, являющегося объектом шаблонного класса, определяется, в частности, тем, какие данные используются в качестве обобщенных при создании конкретного объекта. Если при создании двух экземпляров шаблонного класса используются данные разных типов, считается, что эти объекты имеют разные типы.

Результаты работы этой программы приведены ниже.

```
Тип объекта o1: class myclass<int>
Тип объекта o2: class myclass<int>
Тип объекта o3: class myclass<double>

Объекты ol и o2 имеют одинаковый типы
Объекты ol и o3 имеют разные типы
```

Как видим, хотя два объекта представляют собой экземпляры одного и того же шаблонного класса, если параметры не совпадают, их типы считаются разными. В данной программе объект o1 имеет тип myclass<int>, а объект o3 –myclass<double>. Таким образом, их типы не совпадают.

Динамическая идентификация типов применяется не во всех программах. Однако при работе с полиморфными типами механизм RTTI позволяет распознавать типы объектов в любых ситуациях.

Операторы приведения типа

В языке C++ существуют пять операторов приведения типов. Первый оператор является вполне традиционным и унаследован от языка C.

Операция приведения типов в стиле C:

Она может записываться в двух формах:

- **(тип) выражение – для C и C++**
- **тип (выражение) – только для C++**

Результатом операции является значение заданного типа, например:

```
int a = 2;
float b = 6.8;
printf ("%lf %d", double (a), (int) b);
```

Величина a преобразуется к типу double, а переменная b – к типу int с отсечением дробной части, в обоих случаях внутренняя форма представления результата операции преобразования иная, чем форма исходного значения.

Остальные четыре оператора приведения типов были добавлены впоследствии, в C++.

К ним относятся операторы: dynamic_cast,
const_cast,
reinterpret_cast,
static_cast.

Эти операторы позволяют полнее контролировать процессы приведения типов.

Операторы приведения типа

Оператор dynamic_cast

Оператор dynamic_cast осуществляет динамическое приведение типа с последующей проверкой корректности приведения. Если приведение оказалось некорректным, оно не выполняется.

Общий вид оператора dynamic_cast:

dynamic_cast <target_type> (expr);

Здесь - параметр target_type задает результирующий тип,

- параметр expr – выражение, которое приводится к новому типу.

Результирующий тип должен быть указательным или ссылочным, а приводимое выражение – вычислять указатель или ссылку. Таким образом, оператор dynamic_cast можно применять для приведения типов указателей или ссылок.

Оператор dynamic_cast предназначен для приведения полиморфных типов. Dynamic_cast применяется для приведения типов в иерархической структуре наследования; он может приводить базовый тип к производному, производный к базовому или один производный тип – к другому производному типу.

Допустим, даны два полиморфных класса B и D, причем класс D является производным от класса B. Тогда оператор dynamic_cast может привести указатель типа D* к типу B*. Это возможно благодаря тому, что указатель на объект базового класса может ссылаться на объект производного класса. Однако обратное динамическое приведение указателя типа B* к типу D* возможно лишь в том случае, если указатель действительно ссылается на объект класса D.

Оператор dynamic_cast достигает цели, если указатель или ссылка, подлежавшие приведению, ссылаются на объект результирующего класса или объект класса, производного от результирующего. В противном случае приведение типов считается неудавшимся. В случае неудачи оператор dynamic_cast, примененный к указателям, возвращает нулевой указатель. Если оператор dynamic_cast применяется к ссылкам, в случае ошибки генерируется исключительная ситуация bad_cast.

Операторы приведения типа

Оператор dynamic_cast

Пример.

Класс Base является полиморфным, а Derived – производным от него:

```
Base *bp, b_ob;
Derived *dp, d_ob;
bp = &d_ob;  // указатель базового типа ссылается на объект производного класса
dp=dynamic_cast<Derived*>(bp);  //приведение указателя производного типа выполнено успешно
if(dp) cout << "Приведение выполнено успешно";
```

Приведение указателя bp, имеющего базовый тип, к указателю dp, имеющему производный тип, выполняется успешно, поскольку указатель bp на самом деле ссылается на объект класса Derived. Т.о, этот фрагмент выводит на экран сообщение "Приведение выполнено успешно".

Однако в следующем фрагменте приведение не выполняется, потому что указатель bp ссылается на объект класса Base, а приведение базового объекта к производному невозможно:

```
bp = &b_ob; // указатель базового типа ссылается на объект класса Base
dp = dynamic_cast<Derived *> (bp);  // Ошибка
if(!dp) cout << "Приведение не выполняется";
```

Поскольку приведение невозможно, фрагмент выводит на экран сообщение "Приведение не выполняется".

Операторы приведения типа

Применение dynamic_cast к шаблонным классам

```
#include <iostream>
using namespace std;
template <class T> class Num { protected: T val;
  public:    Num(T x) { val = x; }
    virtual T getval() { return val; }   /* ... */  };
template <class T> class SqrNum : public  Num<T>
  { public: SqrNum(T x) : Num<T>(x) { }
  T getval() { return val * val; }  };
int main()
{   Num<int> *bp, numInt_ob(2);
    SqrNum<int> *dp, sqrInt_ob(3);
    Num<double> numDouble_ob(3.3);
  bp = dynamic_cast<Num<int> *> (&sqrInt_ob);
  if(bp) { cout << "Приведение типа SqrNum<int>* к
        типу Num<int>*\n"<< " выполнено успешно";
 cout<<"Значение равно "<<bp->getval()<<endl; }
       else  cout << "Ошибка\n";   cout << endl;
 dp = dynamic_cast<SqrNum<int> *> (&numInt_ob);
   if(dp)   cout << "Ошибка\n";   else { cout<<"Приве-
дение типа Num<int>* к типу SqrNum<int>*
невозможно\n";
   cout<<"Приведение указателя на объект базового \n";
cout<<" класса к указателю на объект производного типа
невозможно\n";   }   cout << endl;
bp = dynamic_cast<Num<int> *> (&numDouble_ob);
   if(bp)   cout << "Ошибка\n";   else   cout<<"Приведе-
ние типа Num<double>* к типу Num<int>* невоз-
можно\n"; cout<<"Это два разных типа\n";
  return 0;
}
```

Результаты работы этой программы:

```
 Приведение типа SqrNum<int>* к типу
Num<int>* выполнено успешно
 Значение равно 9

Приведение типа Num<int>* к типу
SqrNum<int>* невозможно
Приведение указателя на объект базового
класса к указателю на объект производного
класса невозможно.

Приведение типа Num<double>* к типу
Num<int>* невозможно.
Это два разных типа.
```

Основной смысл этого фрагмента заключается в том, что с помощью оператора dynamic_cast нельзя привести указатель на объект одной шаблонной специализации к указателю на экземпляр другой шаблонной специализации.

Напоминание: точный тип объекта шаблонного класса определяется типом данных, которые используются при его создании. Таким образом, типы Num<double> и Num<int> различаются.

Операторы приведения типа

Оператор const_cast

Оператор const_cast используется для явного замещения модификаторов const и/или volatile. Операция служит для удаления модификатора const. Как правило, она используется при передаче в функцию константного указателя на место формального параметра, не имеющего модификатора const. Результирующий тип должен совпадать с исходным, за исключением атрибутов const и volatile.

Общий вид оператора const_cast:

const_cast <type> (expr);

Здесь - параметр type задает результирующий тип приведения,
- параметр expr – выражение, которое приводится к новому типу.

```
#include <iostream>
using namespace std;
  void sqrval(const int *val)  {  int *p;
  // Удаление модификатора const
  p = const_cast<int *> (val);
  *p = *val * *val;                    }
int main()
{ int x = 10;
    cout<<"Значение x перед вызовом:
"<<x<<endl;
  sqrval(&x);
    cout<<"Значение x после вызова:
"<<x<<endl;
  return 0;
}
```

Эта программа выводит на экран следующие результаты:

Значение x перед вызовом: 10
Значение x после вызова: 100

Как видим, функция sqrval() изменяет значение переменной x, даже если ее параметр является константным указателем.

Применение оператора const_cast для удаления атрибута const небезопасно. Его следует использовать осторожно.

Атрибут const можно удалить только с помощью оператора const_cast.

Операторы dynamic_cast, static_cast и reinterpret_cast на атрибут const не влияют.

Операторы приведения типа

Оператор static_cast

Оператор static_cast выполняет неполиморфное приведение. Его можно применять для любого стандартного преобразования типов. Проверка приведения в ходе выполнения программы не производится.

Оператор static_cast имеет следующий вид:

static_cast<type> (expr)

Здесь - параметр type задает результирующий тип приведения,

- параметр expr – выражение, которое приводится к новому типу.

Оператор static_cast, по существу, заменяет исходный оператор приведения. Просто он выполняет неполиморфное приведение.

Например, следующая программа приводит переменную типа int к типу double:

```
#include <iostream>
using namespace std;
int main()
{
  int i;
  for(i=0; i<10; i++)
    cout << static_cast<double> (i) / 3 << " ";
  return 0;
}
```

Операторы приведения типа

Оператор reinterpret_cast

Оператор reinterpret_cast преобразует один тип в совершенно другой. Например, он может преобразовать указатель на целое число в целое число, а целое число — в указатель. Кроме того, его можно использовать для приведения несовместимых типов указателей.

Оператор reinterpret_cast имеет следующий вид:

reinterpret_cast<type> (expr)

Здесь - параметр type задает результирующий тип приведения,
- параметр expr – выражение, которое приводится к новому типу.

Например, эта программа иллюстрирует применение оператора reinterpret_cast:

```
#include <iostream>
using namespace std;
int main()
{
  int i;
  char *p = "This is a string";
  i = reinterpret_cast<int> (p); // Приведение указателя к целому числу
  cout << i;
  return 0;
}
```

Здесь оператор reinterpret_cast преобразует указатель p в целое число, т.е. один основной тип – в другой. Такое преобразование является типичным для оператора reinterpret_cast.