Подготовительная программа по программированию на C/C++

Занятие №3

Валентина Глазкова

Объектная модель языка C++
Класс как область видимости

- Основные принципы объектно-ориентированной парадигмы.
- Определение и состав класса.
- Работа с членами класса и указателями на них.
- Дружественные классы и функции.
- Вложенные типы.
- Классы-объединения

Инкапсуляция — базовый принцип ООП

Инкапсуляция, или сокрытие реализации, является фундаментом объектного подхода к разработке ПО.

- Следуя данному подходу, программист рассматривает задачу в терминах предметной области, а создаваемый им продукт видит как **совокупность абстрактных сущностей — классов** (в свою очередь формально являющихся пользовательскими типами).
- Инкапсуляция **предотвращает прямой доступ** к внутреннему представлению класса из других классов и функций программы.
- Без нее теряют смысл остальные **основополагающие принципы объектно-ориентированного программирования** (ООП): наследование и полиморфизм. Сущность инкапсуляции можно отразить формулой:

Открытый интерфейс + скрытая реализация

Класс: в узком или широком смысле?

Принцип инкапсуляции распространяется не только **на классы** (class), но и **на структуры** (struct), а также **объединения** (union). Это связано с расширенным толкованием понятия «класс» в языке C++, трактуемом как в узком, так и широком смысле:

- **класс в узком смысле** — *одноименный* составной пользовательский тип данных, являющийся контейнером для данных и алгоритмов их обработки. Вводится в текст программы определением типа со спецификатором class;
- **класс в широком смысле** — *любой* составной пользовательский тип данных, агрегирующий данные и алгоритмы их обработки. Вводится в текст программы определением типа с одним из спецификаторов struct, union или class.

Определение класса: синтаксис

Простейшее определение класса в языке C++ имеет вид:

```
//заголовок класса
<спецификатор класса> <имя класса>
{
        //тело класса
        [<члены класса>]
};
```

где <*спецификатор класса*> - ключевое слово class, union или struct, <*имя класса*> - правильный идентификатор, а <*члены класса*> определены в соответствие с требуемым уровнем доступа (открытые - public, закрытые - private, защищённые - protected)

Каждое определение класса вводит **новый тип данных**

Тело класса определяет **полный перечень его членов**, который не может быть расширен после закрытия тела

Определение класса: пример

```
// пустой класс
class Document { };

//класс
class Book {
public:
        Book();
private:
        string _author, _title;
};
```

Описание класса

Описание класса вида

<спецификатор класса> <имя класса>;

вводит в программу имя класса и указывает его природу, не определяя состав атрибутов, методов и иных частей класса.

В случае если класс описан, но не определен, допускается:

- определять ссылки и указатели на объект класса;
- определять член другого класса как ссылку или указатель на данный класс.

В случае если класс описан, но не определен, запрещается:

- определять объект класса;
- определять член другого класса как принадлежащий данному классу;
- разыменовывать указатели на объект класса;
- использовать ссылки и указатели для доступа к членам класса.

Описание класса: пример

```
// описание класса
class Account;                  // допустимо

// определения объектов:
Account *pAcc = NULL;           // допустимо
void foo(const Account *pA); // допустимо

class Account acc;              // недопустимо
```

Объект класса

Выделение памяти под объект (экземпляр) класса происходит **при определении** такого **объекта**, **ссылки** на объект или **указателя** на него. Объект класса:

- имеет **размер**, достаточный для размещения в нем всех (нестатических) атрибутов, собственную копию каждого из которых имеет каждый индивидуальный объект;
- характеризуется **областью видимости** и обладает **временем жизни**.

Объекты одного класса **могут присваиваться** друг другу. В отсутствие в определении класса конструктора копирования копирование объектов эквивалентно копированию атрибутов по правилам копирования значений их типов. Будучи параметром или возвращаемым значением функции, объект класса передается через стек **по значению**.

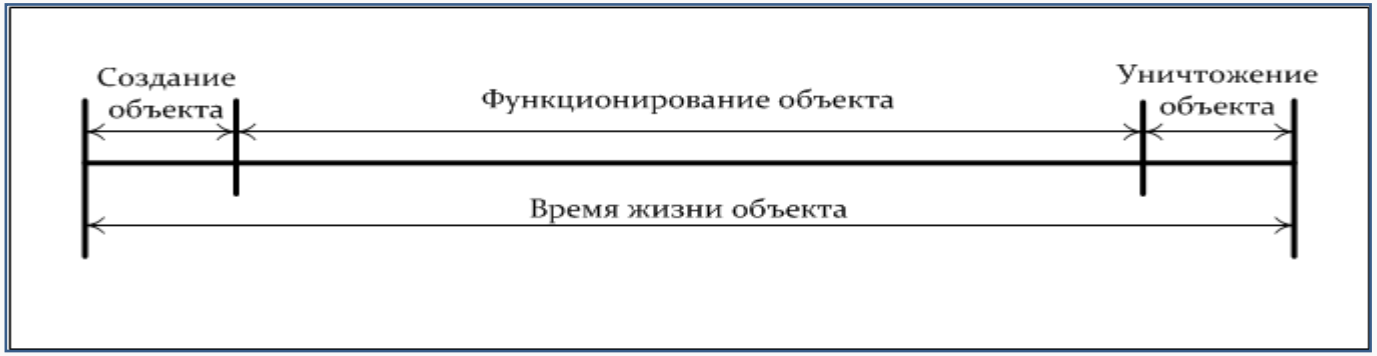
Объект класса: пример

```
// определение класса
// (и объекта – допустимо)
class Account { /* … */ };
class Deposit { /* … */ } deposit;
// определения объектов: допустимо
class Account acc;
Account *pAcc = new Account(100);
Account &rAcc = acc;
class Account *&prAcc = pAcc;
// определения объектов: недопустимо
Account &rAcc2 = &acc;
```

Константный объект класса (1 / 2)

Объект класса (как и объект базового типа) может быть объявлен **константным**. Константный объект не допускает изменения значения его атрибутов на протяжении всего времени жизни объекта, за исключением:

- времени работы конструктора (объект создается);
- времени работы деструктора (объект уничтожается).



Константный объект класса (2 / 2)

Неизменность членов-данных класса, являющихся указателями, распространяется **только на их значение** (адрес) и не распространяется на содержимое областей памяти, которые они адресуют.

Примечание: Модификация адресуемых таким образом областей памяти демонстрирует «плохой стиль» программирования

Пример объявления константного объекта класса:

```
const Document myDocument;
```

Состав класса: атрибуты

За содержательную сторону класса на языке C++ отвечают входящие в него **атрибуты** (члены данных):

- статические (со спецификатором static):
  - в том числе константные (со спецификатором const);
- неустойчивые (со спецификатором volatile);
- изменчивые (со спецификатором mutable);
- прочие (без формальных спецификаторов).

Состав класса: методы

За алгоритмическую сторону класса на языке C++ отвечают входящие в него **методы** (функции-члены), в том числе специальные (**конструкторы и деструкторы**):

- встроенные (со спецификатором inline);
- константные (со спецификатором const);
- статические (со спецификатором static);
- неустойчивые (со спецификатором volatile);
- прочие (без формальных спецификаторов).

Состав класса: прочие элементы

В состав класса на языке C++, помимо атрибутов и методов, могут входить **следующие прочие элементы**:

- **описания дружественных объектов**:
  - прототипы дружественных функций;
  - описания дружественных классов;
- **определения типов**;
- **битовые поля**;
- **вложенные классы**.

Нестатические члены данных

За содержательную сторону класса ответственны его **атрибуты**. Чаще всего каждый экземпляр (объект) класса имеет свой набор **нестатических** атрибутов.

Нестатические члены данных **сопоставляются с конкретным экземпляром** класса и тиражируются на все объекты. Время жизни такого члена данных равно времени жизни объекта. Нестатические члены данных нельзя инициализировать в теле класса.

Для обращения к нестатическим членам данных служат операции доступа (`.` или `->`), левым операндом которых выступает леводопустимое выражение: идентификатор объекта, ссылка или указатель на объект

Нестатические члены данных: пример

```
class Book
{
        /* … */
 public:
        int             pages;
        string          title;
        vector<string> chapTitles;
        /* … */
} book;
Book *pBook = &book;
/* … */
book.pages = 100;
// эквивалентно: pBook->pages = 100;
// эквивалентно: (*pBook).pages = 100;
```

Статические члены данных (1 / 2)

Статический член данных класса — это **глобальный объект**, совместно используемый всеми объектами своего класса.

Статические объекты **не тиражируются** и **существуют** даже **при отсутствии экземпляров**, поскольку связаны не с переменной (объектом), а типом (классом).

Преимущества статических членов данных перед глобальными объектами состоят в том, что:

- статические члены находятся в области видимости класса, а не в глобальном пространстве имен;
- на статические члены распространяется действие спецификаторов доступа.

Статические члены данных (2 / 2)

Обычно статический член инициализируется вне определения класса. Определение статического члена данных в программе может быть лишь одно. Для обращения к статическому члену класса могут использоваться:

- операция доступа с леводопустимым выражением;
- операция разрешения области видимости с именем класса в качестве левого операнда.

Примечание: В качестве исключения константный статический член целого типа может быть инициализирован в теле класса. В этом случае он трактуется как именованная константа.

Статический член данных может иметь тот же тип класса, членом которого он является, а также быть аргументом по умолчанию для его метода.

Статические члены данных: пример

```
class BinaryTree
{       /* … */
 public:
        static char delimiter;
        static const short base = 10;
        static const char *format;
};
char BinaryTree::delimiter = ',';
const char * BinaryTree::format = "(%d) %d,_";
BinaryTree tree; // ...
tree.delimiter = ' ';
BinaryTree::delimiter = ';';
```

Указатель this

Указатель this — **неявно определяемый константный указатель** на объект класса, через который происходит вызов соответствующего нестатического метода.

Для неконстантных методов класса T имеет тип T *const, для константных — имеет тип const T *const, для неустойчивых — volatile T *const. Указатель this допускает разыменование (*this).

Применение this внутри методов допустимо, но чаще всего излишне. Исключение составляют две ситуации:

- сравнение адресов объектов:

```
if (this != someObj) /* ... */
```

- оператор return:

```
return *this;
```

Нестатические методы класса

За поведение реализованной в виде класса абстракции отвечают функции-члены (**методы**). В отличие от атрибутов, методы класса существуют в единственном экземпляре, причем даже тогда, когда ни один объект класса не существует.

Подавляющее большинство методов класса оперирует нестатическими атрибутами и в этом смысле может условно именоваться **нестатическими** методами.

Для методов произвольного класса справедливо следующее:

- методы класса имеют доступ ко всем атрибутам класса;
- методы класса могут перегружать другие методы того же класса;
- нестатические методы класса получают в свое распоряжение указатель this.

Нестатические методы класса: пример

```
class Book
{       /* … */
        // конструктор по умолчанию
        Book();
        // конструктор копирования
        Book(const Book &other);
        // деструктор
        ~Book();
        /* … */
        void printInfo();
        string getISBN(char *format = NULL);
};
```

Встроенные методы класса

Функция-член, определенная **внутри класса**, **по умолчанию** является встроенной, то есть подставляемой (на уровне объектного кода) в точку своего вызова.

Чтобы интерпретироваться как встроенная, функция-член, определенная **вне класса**, в теле класса **должна** явно **сопровождаться спецификатором** inline.

Конструктор класса может быть объявлен как встроенный.

Деструктор класса также может быть встроенным.

Встроенные методы класса: пример

```
class Account
{       /* … */
        double getAmount() { return _amount; }

        inline string getCurrCode()
        { return _currCode; }

        inline string getAccInfo(char *format);
};
inline string Account::getAccInfo(char *format)
{ /* … */ }
```

Константные методы класса

Методы класса могут модифицировать атрибуты соответствующего объекта, а могут не делать этого. «Безопасные» с точки зрения работы с константными объектами методы могут помечаться программистом как **константные**.

Константный метод не может модифицировать атрибуты класса (за исключением изменчивых). Применительно к константному объекту могут быть вызваны **только константные методы**. Константные методы могут перегружаться неконстантными методами с идентичной сигнатурой и типом возвращаемого значения.

Примечание: Применение константных методов с неконстантными объектами обеспечивает дополнительный уровень безопасности кода при разработке.

Константные методы класса: пример

```
class Book
{       /* … */

        void printInfo() const;

        // перегруженные методы
        string getChapTitle(int number);
        string getChapTitle(int number) const;
};
```

Статические методы класса

Методы класса, обращающиеся **только к статическим членам данных**, могут объявляться как **статические**.

Статическим методам класса не передается указатель this. Они не могут быть константными или неустойчивыми.

Для вызова статического метода класса может использоваться операция доступа с леводопустимым выражением или операция разрешения области видимости с именем класса в качестве левого операнда.

```
class BinaryTree
{       /* … */
public: static char *getFmt() { /* … */ }
private:        static char *_format;
};
```

Класс как область видимости

Класс – наряду с блоком, функцией и пространством имён – **является конструкцией** C++, которая **вводит** в состав программы одноимённую **область видимости**. (Строго говоря, область видимости в программу вводит определение класса, а именно его тело.)

Все члены класса видны в нём самом с момента своего объявления.

Порядок объявления членов класса важен: нельзя ссылаться на члены, которые предстоит объявить позднее. Исключение составляет разрешение имён в определении встроенных методов, а также имён (статических членов), используемых как аргументы по умолчанию.

В области видимости класса находится не только его тело, но и внешние определения его членов: методов и статических атрибутов

Конструкторы и деструкторы (1 / 2)

Конструктор – метод класса, автоматически применяемый к каждому экземпляру (объекту) класса перед первым использованием (в случае динамического выделения памяти – после успешного выполнения операции new)

Освобождение ресурсов, захваченных в конструкторе класса либо на протяжении времени жизни соответствующего экземпляра, осуществляет **деструктор**.

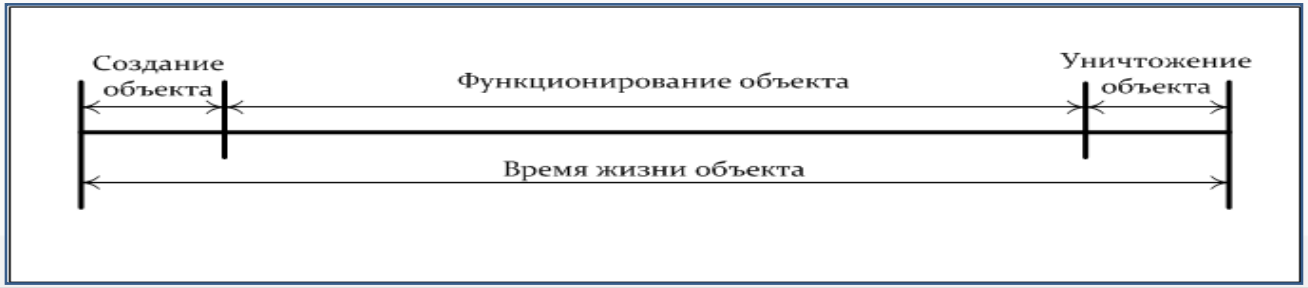
В связи с принятым по умолчанию почленным порядком инициализации и копирования объектов класса в большинстве случаев возникает необходимость в реализации, - наряду с конструктором по умолчанию, - *конструктора копирования* и перегруженной *операции*-функции *присваивания* operator=

Конструкторы и деструкторы (2 / 2)

Выполнение любого конструктора состоит из двух фаз:

- фаза явной инициализации (обработка списка инициализации);
- фаза вычислений (исполнение тела конструктора)

Конструктор **не может определяться** со спецификатором const и volatile. Константность и неустойчивость объекта устанавливается по завершении работы конструктора и снимается перед первым вызовом деструктора



Инициализация без конструктора (1 / 2)

Класс, все члены которого открыты, может задействовать механизм **явной позиционной инициализации**, ассоциирующий значения в списке инициализации с членами данных в соответствии с их порядком.

```
class Test
{
public:
        int    int_prm;
        double dbl_prm;
        string str_prm;
};
// …
Test test = { 1, -3.14, "dictum factum" };
```

Инициализация без конструктора (2 / 2)

Преимуществами такой техники выступают:

- скорость и эффективность, особо значимые при выполнении во время запуска программы (для глобальных объектов).

Недостатками инициализации без конструктора являются:

- пригодность только для классов, члены которых открыты;
- отсутствие поддержки инкапсуляции и абстрактных типов;
- требование предельной точности и аккуратности в применении

Конструкторы по умолчанию (1 / 2)

Конструктор по умолчанию **не требует задания значений** его параметров, хотя таковые могут присутствовать в сигнатуре.

```
class Test
{
public:
        Test(int ipr = 0, double dpr = 0.0);
        /* … */
};
```

Наличие формальных параметров в конструкторе по умолчанию позволяет **сократить общее число конструкторов** и объем исходного кода.

Конструкторы по умолчанию (2 / 2)

Если в классе определен хотя бы один конструктор с параметрами, то при использовании класса с динамическими массивами экземпляров конструктор по умолчанию **обязателен**.

```
Test *tests = new Test[TEST_PLAN_SIZE];
```

Если конструктор по умолчанию **не определен**, но существует хотя бы один конструктор с параметрами,
в определении объектов должны присутствовать аргументы. Если ни одного конструктора не определено, объект класса не инициализируется (память под статическими объектами по общим правилам обнуляется).

Конструкторы с параметрами: пример

```
class Test
{
public:
        Test(int prm) : _prm (prm) {}
private:
        int     _prm;
};

// все вызовы конструктора
//допустимы и эквивалентны
Test   test1(10),
test2 = Test(10),
test3 = 10;     // для одного аргумента
```

Массивы объектов: пример

```
// массивы объектов класса определяются
// аналогично массивам объектов
// базовых типов
// для конструктора с одним аргументом
Test testplan1[] = { 10, -5, 0, 127 };
// для конструктора с
//несколькими аргументами
Test testplan2[5] = {
        Test(10, 0.1),
        Test(-5, -3.6),
        Test(0, 0.0),
        // если есть конструктор по умолчанию
        Test()
};
```

Закрытые и защищённые конструкторы

Объявление конструктора класса **защищённым** или **закрытым** даёт возможность ограничить или полностью запретить отдельные способы создания объектов класса

В большинстве случаев закрытые и защищённые конструкторы используются для:

- **предотвращения копирования** одного объекта в другой;
- указания на то, что конструктор **должен вызываться** только для **создания подобъектов** базового класса в объекте производного класса, а не создания объектов, доступных в коде программы непосредственно

Почленная инициализация и присваивание (1 / 2)

Почленная инициализация по умолчанию — механизм инициализации одного объекта класса другим объектом того же класса, который активизируется независимо от наличия в определении класса явного конструктора.

Почленная инициализация по умолчанию происходит в следующих ситуациях:

- явная инициализация одного объекта другим;
- передача объекта класса в качестве аргумента функции;
- передача объекта класса в качестве возвращаемого функцией значения

Почленная инициализация и присваивание (2 / 2)

Почленная инициализация по умолчанию **подавляется** при наличии в определении класса конструктора копирования.

Запрет почленной инициализации по умолчанию осуществляется одним из следующих способов:

- описание закрытого конструктора копирования (не действует для методов класса и дружественных объектов);
- описание конструктора копирования без его определения (действует всюду).

Почленное присваивание по умолчанию — механизм присваивания одному объекту класса значения другого объекта того же класса, отличный от почленной инициализации по умолчанию использованием копирующей операции-функции присваивания вместо конструктора копирования.

Конструкторы копирования

Конструктор копирования принимает в качестве единственного параметра **константную ссылку** на существующий объект класса.

В случае отсутствия явного конструктора копирования в определении класса производится почленная инициализация объекта по умолчанию.

```
class Test
{
        /* … */
        Test(const Test &other);
        /* … */
};
```

Конструкторы и операции преобразования

Конструкторы преобразования служат для построения объектов класса по одному или нескольким значениям иных типов.

Операции преобразования позволяют преобразовывать содержимое объектов класса к требуемым типам данных.

```
class Test
{
        // конструкторы преобразования
        Test(const char *);
        Test(const string &);
        // операции преобразования
        operator int    () { return int_prm; }
        operator double () { return dbl_prm; }
        /* … */
};
```

Деструкторы (1 / 2)

Деструктор — не принимающий параметров и не возвращающий результат метод класса, автоматически вызываемый при выходе объекта из области видимости, а также в случае применения к указателю на объект класса операции delete.

Типичные задачи деструктора:

- сброс содержимого программных буферов в долговременные хранилища;
- освобождение (возврат) системных ресурсов, главным образом — оперативной памяти;
- закрытие файлов или устройств;
- снятие блокировок, останов таймеров и т.д.

Деструкторы (2 / 2)

Для обеспечения корректного освобождения ресурсов объектами производных классов деструкторы в иерархиях, как правило, определяют как **виртуальные**.

```
class Test
{
        /* … */
        virtual ~Test();
};
```

Примечание: деструктор не вызывается при выходе из области видимости ссылки или указателя на объект.

Список инициализации в конструкторе

Выполнение любого конструктора состоит из двух фаз:

- фаза явной (неявной) инициализации (обработка списка инициализации) — предполагает **начальную инициализацию** членов данных;
- фаза вычислений (исполнение тела конструктора) — предполагает **присваивание значений** (в предварительно инициализированных областях памяти).

Присваивание значений членам данных – объектам классов в теле конструктора **неэффективно** ввиду ранее произведенной инициализации по умолчанию. Присваивание значений членам данных, представляющих базовые типы, **по эффективности равнозначно** инициализации.

К началу выполнения тела конструктора все **константные члены и члены-ссылки** должны быть инициализированы.

Дружественные классы и функции

Реализованный в объектной модели C++ **механизм дружественных отношений позволяет** классу **разрешать доступ** к своим неоткрытым (закрытым и защищенным) членам. Дружественные объекты не являются членами класса, поэтому на них не распространяется действие спецификаторов доступа.

Отношения дружественности могут устанавливаться:

- между классом и функцией из пространства имен;
- между классом и методом другого класса;
- между двумя классами.

Синтаксис отношений дружественности прекрасно иллюстрирует перегрузка операций `>>` и `<<` для организации потокового консольного и файлового ввода-вывода.

Потоковый ввод-вывод (1 / 2)

Консольный и файловый потоковый ввод-вывод экземпляра класса организуется путем перегрузки операций >> и <<, которая производится следующим образом:

```
#include <iostream>
class Account
{
        friend istream & operator >>(istream &, Account &);
        friend ostream & operator <<(ostream &, Account &);
/* … */
};
```

Потоковый ввод-вывод (2 / 2)

Реализация каждого из дружественных методов тривиальна:

```
#include <iostream>
ostream &operator << (ostream &ostr, Account &acc)
{
        ostr << "Name: " << acc._name << "; "
        ostr << "Acc. No.: " << acc._number << "; " << endl;
        return ostr;
};
```

Для обеспечения файлового ввода-вывода используются потоки классов ifstream (входной поток) и ofstream (выходной поток).
Дополнительная перегрузка операций >> и << не требуется.

Указатели на атрибуты класса

Наряду с «обычными» указателями на данные и глобальные функции выделяют **указатели на нестатические методы и атрибуты** классов.

Примечание: Указатели на статические члены класса оформляются и используются так же, как указатели на объекты, не являющиеся членами класса.

Полный тип указателя на атрибут класса содержит имя класса и тип его атрибута. Например:

```
class Screen {
public:
        short _height; /* … */
};
short Screen::*psh = &Screen::_height;
```

Указатели на методы класса (1 / 2)

Полный тип указателя на метод класса содержит имя класса, список типов параметров (сигнатуру) метода и тип возвращаемого значения.

Например:

```
class Screen {
public:
        int height() { return _height; }
        int width()  { return _width; }
        /* … */
};
int (Screen::*pmeth1)() = NULL;
int (Screen::*pmeth2)() = &Screen::width;
typedef Screen& (Screen::*Action)();
```

Указатели на методы класса (2 / 2)

Адреса методов **нельзя** присваивать указателям на глобальные функции, даже если их сигнатуры и типы возвращаемых значений полностью совпадают. Причина: методы класса находятся в области видимости класса-владельца.

Адреса методов **можно** использовать для объявления формальных параметров функций, типов возвращаемого значения и задания значений параметров функций по умолчанию.

Для доступа к атрибутам и методам класса по указателям предназначены операции `.*` и `->*`. Например:

```
Screen *tmpScreen  = new Screen();
short Screen::*psh = &Screen::_height;
tmpScreen->*psh = 80;
```

Вложенные классы

Класс, объявленный внутри другого класса, называется **вложенным** (в **объемлющий** класс). При этом:

- определение вложенного класса может находиться в любой секции объемлющего, а его имя известно в области видимости объемлющего класса, но нигде более;
- объемлющий класс имеет право доступа только к открытым членам вложенного, и обратно;
- как правило, вложенный класс объявляют закрытым в объемлющем, а все члены вложенного класса объявляют открытыми;
- невстроенные методы вложенных классов определяются вне самого внешнего из объемлющих классов;
- вложенный класс может быть объявлен в теле объемлющего, но не определен в нем (принцип сокрытия реализации).

Вложенные классы: пример

```
class List
{
public:
//...
private:
        //ListItem – закрытый вложенный тип
        class ListItem {
        //а его члены открыты
        public:
                ListItem(int val=0);
                ListItem *next;
                int value;
        };
        ListItem *list;
};
```

Неустойчивые объекты (1 / 2)

Неустойчивые, или **асинхронно изменяемые** (volatile), объекты могут изменяться незаметно для компилятора.

Пример: переменная, обновляемая значением системных часов (например, в обработчике события, сигнала).

Целью определения объекта как неустойчивого является информирование компилятора о том, что тот не может определить, каким образом может изменяться значение данного объекта.

Спецификатор volatile сообщает компилятору о том, что при работе с данным объектом **не следует выполнять оптимизацию кода**.

Неустойчивые объекты (2 / 2)

Допустимы неустойчивые объекты скалярных и составных типов, указатели на неустойчивые объекты, неустойчивые массивы:

- в неустойчивом массиве неустойчивым считается каждый элемент;
- в неустойчивом экземпляре (объекте) класса неустойчивым считается каждый член данных. **Объекты классов неустойчивы целиком.**

Например:

```
volatile unsigned long timer;// неустойчивый скаляр
volatile short ports[size]; // неустойчивый массив
// указатель на неустойчивый объект класса
volatile Timer *tmr;
```

Для преобразования неустойчивого типа в устойчивый используется const_cast.

Неустойчивые методы класса

Методы класса могут объявляться как **неустойчивые**.

Неустойчивые методы класса являются **единственной категорией методов**, которые (наряду с конструкторами и деструкторами) могут вызываться применительно к неустойчивым объектам класса (объектам, значение которых изменяется способом, не обнаруживаемым компилятором).

Например:

```
class Timer
{       /* … */
        void getCurTime() volatile;
        /* … */
};
```

Изменчивые члены данных

Атрибуты класса, допускающие модификацию при любом использовании объекта, должны определяться как **изменчивые**.

Изменчивые атрибуты **не являются константными**, даже будучи членами константного объекта, что позволяет модифицировать их значения, в том числе константными методами.

Например:

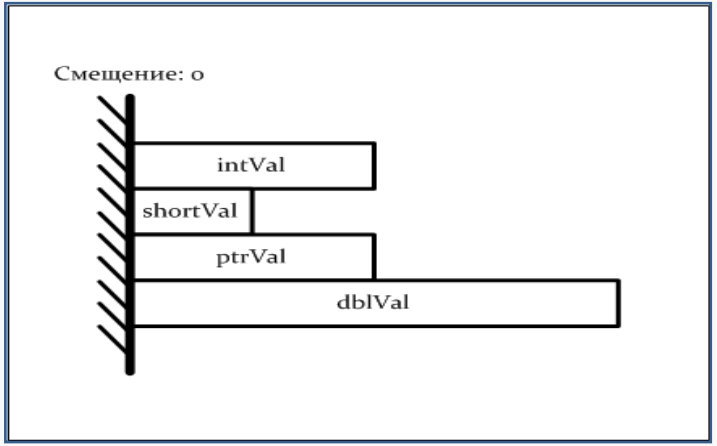
```
class Book
{       /* ... */
        mutable int _currentPage;
        /* ... */
        void locate(const int &value) const
        { /* ... */ _currentPage = value; /* ... */ }
};
```

Классы-объединения

Объединение в C++ — специальная категория класса, в котором члены данных физически располагаются, начиная с одного машинного адреса.

Размер класса-объединения определяется размерами его атрибутов и в целом **равен максимальному** среди них.

В любой момент времени значение может быть присвоено **только одному атрибуту**.



Объединение может включать как члены-данные, так и члены-функции

Классы-объединения

Все члены объединения **открыты** по умолчанию.

Объединение **не может наследовать** какие-либо другие классы.

Объединение **не может** использоваться **в качестве базового класса**.

Объединение **не может** иметь **виртуальные члены-функции**.

Никакие **статические** переменные **не могут** быть членами объединения.

Никакой объект **не может** быть членом объединения, если этот объект имеет **конструктор или деструктор**.

Классы-объединения: пример

```
union DataChunk
{
        int       intVal;
        short     shortVal;
        char*     ptrVal;
        double    dblVal;
};
DataChunk dc;
DataChunk *pdc = &dc;

dc.intVal = 0xFFAA;
pdc->shortVal = 077;
```

Безымянные объединения

```
// имя типа объединения
// может быть опущено
class IOPort
{       /* … */
        union
        {
                int         intVal;
                short       shortVal;
                char*       ptrVal;
                double dblVal;
        } _value;
} port;
port._value.intVal = 0xABCD;
```

Анонимные объединения

Объединение без имени, за которым не следует определение объекта, называется **анонимным**. К его членам можно обращаться непосредственно из той области видимости, в которой оно определено.

Анонимные объединения позволяют устранить один уровень доступа, у них не может быть каких бы то ни было методов.

Анонимные объединения: пример

```
class IOPort
{
        /* … */
        union {
                int     intVal;
                short   shortVal;
                char*   ptrVal;
                double dblVal;
        };
} port;
port.ptrVal = NULL;
```

Битовые поля в определении классов

Для хранения заданного числа двоичных разрядов может быть определен член класса, называемый **битовым полем**. Его тип должен быть знаковым или беззнаковым целым.

Определенные друг за другом битовые поля по возможности «упаковываются» компилятором. Например:

```
class IOPort
{
        unsigned int _ioMode  : 2;
        unsigned int _enabled : 1;
        /* … */
};
```

К битовому полю запрещено применять оператор взятия адреса.

Битовые поля не могут быть статическими членами класса.

Валентина Глазкова

Спасибо за внимание!