**Приведение типов в языке Си.**

char \*a = malloc (sizeof(char)\*100);

char \*a = (char\*) malloc (sizeof(char)\*100);

Malloc возвращает указатель на void\*, поэтому в 1 варианте неявное преобразование типа, а во 2 варианте явное. Программист говорит компилятору, что он в курсе того, что происходит и всё нормально. Компилятор С++ более жестко следит за таким.

int i = 2; char \*c = i; // Это тоже неявное приведение типов, в С будет warning, в С++ будет ошибка.

char \*c = (char\*) i; // Так норм

В каких случаях компилятор применяет неявное приведение типа ?

1)Инициализация: int c=3.5;

2)Присваивание (assign): double a=3.5; int c = 5; c=a;

3)Вызов функции (call): func (int a) {} main: double c=3.5; func (c);

Пусть существует класс Complex с конструктором Complex (int re=0, int im=0);

Complex N=5; // начнётся приведение типа, должен быть сгенерирован примерно следующий код:

Complex (Complex(5)); // то есть сначала создастся неименованный объект типа Complex конструктором с 1 параметром (либо конструктор с многими параметрами, но которые заданы по умолчанию =0 у нас, в нашей реализации класса Complex три конструктора (без параметров, с 1, с 2) заменились одним таким ), потом создастся объект N и у него вызовется либо оператор= либо конструктор копирования, в зависимости от того, что у вас реализовано, а что нет. Вообще компиляторы это всё оптимизируют конечно и скорее всего хороший компилятор сделает вот так: Complex(5); для создания объекта N.

Пусть реализован класс Matrix с конструктором Matrix (int size). Тогда при выполнении:

Matrix M = 3; будет создана матрица 3х3, чтобы такого поведения не было, есть слово explicit.

В объявлении конструктора пишем explicit Matrix(int size); тогда неявного вызова конструктора не будет никогда. Правда все равно можно сделать Matrix M = (Matrix) 3; будет вызван конструктор копирования или оператор=. Получается, что с помощью конструктора копирования можно сделать приведение между любыми двумя классами. Если есть классы A и B и нужно приводить объекты одного класса к другому, то можно добавить в класс А конструктор копирования, строящий по объекту класса B объект класса А.

A (const B& obj); и в B тоже можно сделать конструктор копирования B (const A& obj);

B b (5); A a(3); b=a; // Будет вызван метод B (const A& obj) и будет построен временный объект типа B, после из этого неименованного временного объекта в объект b с помощью оператора= (или конструктора копирования) будет скопировано содержимое. Если добавить в класс B operator= (const A& obj); то будет выполнен один шаг.

Ещё пример про Complex. Complex N = 3; (вызов конструктора с 2 параметрами), но если я захочу писать int x = N; применённый только что подход не подойдёт, т.к. доступа к встроенному типу int у нас нет, и мы не можем там создать какой-то конструктор. Но мы можем реализовать оператор приведения типа в классе Complex.

Complex :: operator int() { return Re; } например. Тогда компилятор вызовет оператор приведения типа в классе Complex и потом с помощью оператора= уже запишет int в int.

Примеры со строками. Пусть реализована некоторая функция showText (const string & obj); Очень удобно делать вот так showText (“Hello!”); то есть вызывать не на объекте string, а на объекте/типе, который с помощью конструкторов string приводится к string, а потом уже передается этот временный неименованный string по ссылке в метод и там не меняется уже. Это используется очень часто.

Константные объекты + mutable.

Объект, у которого нельзя менять поля, нельзя вызывать “не const” методы и нельзя передавать его по “не const” ссылке – константный. Примеры: const Complex pi(3.14); // мнимая часть пусть будет = 0, любые попытки вызывать pi.set() или другие не const методы не скомпилируются.

Допустим есть класс Matrix с методом int determinant() const; // чтобы вычислить определитель не нужно менять поля матрицы, поэтому const.

Matrix m(...); m.determinant(); // вообще метод вычислительно сложный, если матрица не меняется, хорошо бы результат вычисления закешировать где-то ?

class Matrix {

void set (int i, int j, int value) { isDeterminantChanged=true; }

int determinantValue;

bool isDeterminantChanged;

int determinant() const { if (isDeterminantChanged==true) calculate(); ... return determinantValue; }

};

Как это будет работать? Если матрица меняется, то флаг устанавливается в true, тогда метод determinant будет знать, что нужно заново вычислять. Довольно обычная схема кеширования. Но это не скомпилируется, в методе determinant стоит const. Такой метод не может менять поля класса. Можно убрать const. К сожалению теперь вызвать метод determinant у константной матрицы не получится. foo (const Matrix& obj) { obj.determinant(); } не скомпилируется т.к. const нету у метода теперь. Очень плохо, что же делать?:) Именно из-за таких извращенных случаев и нужен mutable :) Оставляем у метода const, а переменным навешиваем mutable. Т.е. эти поля const методы смогут изменять.

Указатели на функцию внутри класса.

class Person {

string getName() const;

string getSurname() const;

};

Person\* p = new Person[200];

sort(p,p+100,&Person::getName); // здесь сортируются первые 100 объектов с использованием в качестве сравнения метода getName, то есть по имени сравниваются. Опишем прототип функции:

void sort (Person \*begin, Person \*end, std::string (Person::\*)() const );

Указатель на метод класса Person возвращающий string, при этом const метод. Вообще писать такой длинный параметр неудобно, можно применять typedef.