Const-References-Casts-Static-Inline

Правила разбора const:

1.То, после чего стоит const является константным.

2.Разбирать инструкцию надо справа налево, примеры:

const int \* m; <- указатель на целую константу;

int \* const n; <- константный указатель на целое;

const int\* const p; <- константный указатель на целую константу.

Const:

1.const int size=10;

Локальная переменная, нельзя в дальнейшем изменять её значение. const делает неизменяемым то, после чего стоит.

2.char const \*s;

Теперь нельзя изменять массив или строку на которую указывает s, нельзя написать s[1]=’a’; Зато можно написать s=s2; Теперь s указывает на другую строку. Это называется указатель на константу (const стоит после char, поэтому неизменяемым здесь являются как раз char\_ы то есть символы строки).

3.char \* const s; теперь нельзя менять указатель, s[1]=’a’ можно писать, а вот s=s2 нельзя. Это называется константный указатель (const стоит после char\*, поэтому неизменяемым здесь является именно указатель на char).

4.можно написать const char \*s; это аналог первой записи, нельзя изменять содержимое строки.

5.const char\* const s; здесь нельзя изменять ни сам указатель ни значения в строке. Это называется константный указатель на константу.

6.const у членов-данных классов (у полей).

private:

const int size;

Присвоить что-то константе нельзя, как задать значение? Есть специальный синтаксис в конструкторах – список инициализации

Safearray :: Safearray (int \_size) : size(\_size) {}

Safearray :: Safearray (Safearray &object) : size\_(object.size\_) {}

Т.к. поле константным сделали, в операторе присваивания мы не сможем менять размер. Поэтому оператор присваивания нельзя будет реализовать, т.к. списка инициализации у обычных методов не может быть (только у конструкторов).

7.const у параметра функции. Safearray (const Safearray& object) – ссылка на константу (аналог константного указателя на константу). Иногда полезно передавать объекты в функции всегда как const Nameclass & object – если конечно их не надо изменять внутри функции. То, что передаём по ссылке – экономит память, то, что const – предостерегает от изменений объекта. Если то же самое написать через указатель, то получаем const Safearray\* const object; - константный указатель на константу.

8.const у возвращаемого значения.

const Array& operator= (const Array& object);

9.const у метода класса. Это означает, что такой метод не изменяет поля класса. Примеры – вывод значения полей, возвращение размера и т.п.

int getSize () const;

int get () const;

Ссылки.

T& в каком-то смысле близко к T\* const, то есть ссылка аналогична в некотором смысле константному указателю на объект, за исключением того, что ссылка не может быть null, т.е. всегда должна быть инициализирована. Указатель может равняться null.

Приведение типов (без dynamic\_cast).

Вообще говоря, приведение типов – это место в программе, где могут происходить ошибки. При использовании простых типов данных ошибки могут происходить из-за потери точности. Если это указатели, особенно на объекты, проблемы могут быть посерьёзнее, если по указателю вызвать метод, которого на самом деле нету, программа может аварийно завершиться.

У введения нового синтаксиса в С++ для приведения типов было несколько идей.

Первое – чтобы было проще идентифицировать места в коде, где происходит приведение типов (поиск инструкций). В стиле Си это в принципе возможно, но очень трудоёмко, т.к. круглые скобки везде используются. В стиле С++ это всего лишь одна команда grep \*\_cast и всё.

Второе – безопасность. С использованием некоторых конструкций компилятор может лучше проверить, не приведёт ли ваше преобразование типов к чему-то плохому. Например, при dynamic\_cast можно проверить, есть ли отношение наследования у объектов.

Мораль: если у вас используется только примитивное приведение типов, и вас не сильно беспокоит будущий поиск мест применения этого приведения типов, можно ограничиться синтаксисом Си. Однако во всех остальных случаях рекомендуется применять стиль С++ приведения.

1.const\_cast из названия ясно – либо добавить, либо убрать const. Пример:

int i = 5;

const int \*pi = &i;

int \*j = const\_cast<int\*> (pi);

\*j=6;

Была переменная, потом сделали указатель на константу, потом обратно вернули. Const\_cast обеспечивает только находимость, безопасности никакой нету. Чуда не происходит:

const char \* str = “hello”;

char \* s1 = const\_cast <char\*> (str);

s1[0] = ‘H’;

Поведение не определено.

2.static\_cast нужен для:

а) приведения примитивных типов данных: i = static\_cast <int>(f);

б) для пользовательских преобразований (операторы приведения, которые позволяют Complex к Int привести например): static\_cast<string>(”Hello”); Пока что это ничем не лучше (string)”Hello” кроме поиска

в)Указатель на void\* - то есть к нему и от него:

C \*c = new C(); void\* p = static\_cast<void\*>(c); c = static\_cast<C\*>(p);

3.reinterpret\_cast - аналог приведения типов в Си. Не контролируй компилятор ничего, пожалуйста, я сам знаю, что делаю. Частое применение – преобразование между указателями, которые ссылаются на разные типы. Потенциально опасная операция. Пример:

C \*c = new C;

char \*p = reinterpret\_cast<char\*>(c);

Есть у нас в памяти объект с, а мы хотим по нему ходить. Если вам такое пришлось делать, что-то неправильно спроектировано в проекте.

static

static:

1.Глобальные static переменные. Один из способов решения проблемы глобальных переменных в рамках всего проекта – локализация глобальных переменных. 1.cpp : static int count=0; - переменная count будет глобальной на уровне одного файла 1.cpp. Теперь в 2.cpp : можно писать int count=0; - это будет глобальная переменная для всего проекта, она не будет пересекаться с count в 1.cpp.

Технически на этапе линковки происходит два вида линковки: внутренняя (internal) и внешняя (external).

2.Локальные static переменные. Подразумевается, например, локальная переменная некоторой функции. void f () { int c=0; c++; return c; } – неудачная попытка посчитать, сколько раз вызвана функция f. Если объявить переменную как static, то она создастся и инициализируется один раз. int f() { static int c=0; c++; return c; } – на втором вызове вернётся уже 2.

3.static-функции. Когда вы объявляете функцию, она автоматически объявляется с ключевым словом extern. То есть все функции по умолчанию имеют внешнюю линковку. void f(){} == extern void f(){}. Если мы хотим запретить использование функции в других модулях, сделать её доступной только в пределах одного файла исходного кода, то делаем её static и получаем внутреннюю линковку: static void f () {}. Зачем это нужно? Например функция test() может появиться в проекте в десятках файлов разных отделов разработки. Проблема во внешней линковке как раз, делая функцию static мы используем внутреннюю линковку.

4.static поле класса. class Process {}; Хотим знать сколько объектов будет присутствовать в системе (в запущенной программе). В конструкторе Process::Process() { ++count; } , в деструкторе Process::~Process() {--count; }. Как объявить такую переменную? По идее можно как глобальную, но ведь глобальные переменные для всего проекта – это зло, так мы делать не будем. Внутри класса объявим поле private: static int count; Такая переменная не будет находиться в объектах класса, она будет одна для всех объектов. Чего мы достигли таким образом? Глобальная переменная count - не понятно что считала бы: количество открытых файлов, процессов или чего-то ещё? А так мы и синтаксически и по смыслу определили, что static int count считает количество объектов нашего класса. Использование: public: int getCounter(){}. Проблема в том, что объектов может и не быть, что вернёт функция? Нужно её тоже делать статической.

5.static метод класса. public: static int getCount(){}. Для вызова Process::getCount(); даже не имея ни одного объекта. Ограничения таких методов: мы не можем обращаться к нестатическим полям данных. Вообще static в терминах с++ подразумевает, что нам не нужны объекты, чтобы обращаться к полю или методу static. Зачем может понадобится static метод?

static bool cmp (Process &a, Process &b) { return a.id < b.id; } – если бы это была внешняя по отношению к классу функция, то она бы не имела доступа к private полям этих объектов и не смогла бы их сравнить.

static color fromRGB(byte r, byte g, byte b) {}

Инициализация статического поля класса. В конструкторе нельзя, внутри класса нельзя. Нужно явно инициализировать в одном из .cpp файлов: int Process :: counter=0; Даже если поле private все равно получится. Именно на этом этапе в глобальной / статической области памяти выделяется под неё место.

inline

В заголовочном файле объявим и определим функцию a.h : void f() {}. В три .cpp файла подключим этот заголовочный файл, на этапе внешней линковки получим ошибку multiply definition. Вполне логично, три функции с одним именем, параметрами. Теперь включим эту функцию в класс, теперь f это метод класса A. Пробуем подключить заголовочный файл в три .cpp и не получаем такую ошибку. Почему? Все равно ведь три одинаковых метода получается определены.

Те методы, которые определены внутри класса автоматически становятся inline функциями. Вообще inline означает, что мы советуем компилятору встроить функцию (заменять её вызов через стек и все прелести просто подстановкой её кода в месте её вызова).

Нюансы: на этапе компиляции, чтобы сделать функцию встраиваемой, нужно точно знать её тело, поэтому разделённые на .h (объявление) и .cpp (определение) файлы функции никак не удастся сделать встраиваемыми. Теперь, если линковщик видит такую в .h файле Inline функцию, и в нескольких .cpp она подключается #include\_ом, то линковщик просто выкидывает лишние копии и в таблицу общую заносит один раз (правда не стандартизованно какую именно). Суть в том, что таким образом можно реализовать небольшой math.h типа без math.cpp, без лишней нагрузки, и во все свои файлы исходного кода подключать такой заголовочный файл.

Все маленькие методы класса принято объявлять и определять в .h файле а большие и сложные методы определять в .cpp отдельно. То, что в .h функции окажутся по умолчанию inline нас не особо волнует, ну пусть будут встраиваемыми. Если функция отдельно определена в .cpp она перестаёт быть inline.