Содержание

1 Общие сведения

C++ (произносится “си плас плас”) это язык общего назначения с сильной статической типизацией и ручным управлением ресурсами. Дефакто в современном программировании C++ является индустриальным стандартом. Язык был создан Бьярном Строструпом в 1980-м году как объектно-ориентированное расширение языка C, известного с 70-х. В 1985-м появилась первая коммерческая версия компилятора Cfront и по языку была опубликована первая книга.

По прошествии времени, C++ был принят сообществом, широко распространился на различных архитектурах и операционных системах. В 1998-м году язык C++ был стандартизован (работа велась с 1990-го года). Сейчас язык C++ имеет развитую инфраструктуру средств компиляции, отладки, библиотек, поддержку в IDE. Много новых возможностей было введено в язык новым стандартом, принятым в 2011-м году.

1.1 Стандартизация C++

Вопрос к студентам: что такое ISO, какие стандарты ISO вы знаете (читали).

Первый стандарт языка C++ под кодовым номером ISO/IEC 14882-1998 [1] был написан и утверждён ISO в 1998 году. Он включает в себя по нормативной ссылке стандарт языка C, ISO/IEC 9899-1990 [2], что означает следующее:

- Возможности C90 (кроме явно оговоренных исключений) также являются возможностями C++98, также и стандартная библиотека языка C в ревизии C90 является стандартной библиотекой C++98
- Новые возможности C99 [2] не являются возможностями C++98 (стоит отметить, что стандарт C99 включён по ссылке в C++11 [5], но стандарт C11 [4] снова разошёлся с текущим нижележащим подмножеством C в C++)

Стандарт языка C++ традиционно отличается от стандарта языка C тем, что стандарт C определяет корректность программы, написанной на C, стандарт же C++ определяет корректность компилятора. Это различие с первого раза кажется несущественным и бюрократическим, но во многом из-за него, например, стандарт C99 никогда не был целиком реализован ни в одном компиляторе C, поскольку конформные программы

можно писать и с подмножеством возможностей. В то же время гораздо более сложный C++98 был реализован в некоторых компиляторах даже в своих наиболее эзотерических частях, таких как экспорт шаблонов.

Любой код, подаваемый на вход удовлетворяющего стандарту компилятора, стандарт классифицирует на следующие типы:

- **Syntax violation** – нечто, вообще не являющееся кодом на языке C++. Это наиболее распространённый вид кода, производимого программистами.
- **Implementation defined** – зависящий от разработчика и документации конкретного компилятора. Пример – как ведёт себя знаковое целое при сдвиге вправо.
- **Unspecified** – поведение корректного кода, не регламентированное стандартом. Пример – порядок выполнения аргументов у функции.
- **Undefined behavior** – поведение некорректного кода, не запрещённое стандартом. Пример – поведение целого числа при переполнении.
- **Strictly conforming** – хороший, полностью удовлетворяющий стандарту код, не выходящий за пределы стандарта. "In anima vili" редко встречается за пределами hello world у профессионалов и никогда не встречается у новичков.
- **Conforming** – код удовлетворяющий стандарту + использующий implementation defined features.

У разных компиляторов есть своя стратегия выдачи диагностических сообщений. Для компиляции большинства примеров из этих лекций, будет использован компилятор g++ версии 4.7.1 или выше, входящий в GNU Compiler Collection с опциями:

```
g++ -pedantic-errors -Wall -Werror -g3 -O0 -std=c++98
```

```
g++ -pedantic-errors -Wall -Werror -g3 -O0 -std=c++11
```

Эти опции заставляют репортить о не-conforming коде и воспринимать все диагностические сообщения как ошибки, а также включать отладочную информацию включая отладку по макросам и отключать все оптимизации (с моей точки зрения это единственный приемлимый способ компиляции учебных примеров).

Пример conforming кода на C-подмножестве C++:

```
#include <climits>
#include <cstdio>

using namespace std;

int main(void)
{
  (void) printf("biggest int is %d\n", INT_MAX);
  return 0;
}
```

Приведённый выше код является conforming но не strictly conforming из-за использования INT_MAX, являющегося implementation-defined. Тем не менее, нет способа заставить GCC выдать такое диагностическое предупреждение и этот код проходит компиляцию без ошибок, так что можно считать уровень conforming достаточным.

1.2 Важность стандартизации

Стандарты и даже ошибки в стандартах отлиты в граните. Например, в стандарте IEEE POSIX 1003.1-1988 была допущена ошибка: в одной главе было указано, что PATH_MAX включает terminating null, в то время, как в другой главе было указано что не включает. Ошибка была обнаружена после публикации стандарта, люди обратились в комиссию стандартизации и оттуда пришло объяснение, что если написано и так и так, значит может быть и так и так. Хотя изначально идея была как раз в том, чтобы определённо установить включает ли PATH_MAX нулевой символ.

Язык как соглашение между программистами и разработчиками компиляторов утверждается своим стандартом и не существует вне его. Все стандарты ISO доступны бесплатно в версии final draft – последнего черновика перед утверждением. Часто пиратски доступны и окончательные версии.

Читать и понимать стандарт языка – то, чем занимается каждый программист при решении любых спорных вопросов, возникающих у него относительно тех или иных языковых средств. Это последняя и окончательная инстанция. Все эти лекции можно считать приглашением к самостоятельной проработке стандарта C++ совместно с решением практических задач. Собственно чтобы научится писать программы, надо знать

язык и писать программы, иных вариантов нет.

1.3 C++ это конгломерат языковых возможностей

При изучении C++ необходимо понимать, что C++ не является монолитным языком (как C), а является сложным, исторически сформировавшимся конгломератом возможностей, зачастую порождающих взаимоисключающие стили решения конкретных задач. Кроме того, язык имеет много "почти стандартных" библиотек и популярные расширения на конкретных компиляторах (самые известные это GNU расширения, позволяющие, например, использовать в языке C вложенные функции и возвращать результат выполнения блока из фигурных скобок.

Поэтому всегда, когда вы пишете на C++, вы на самом деле пользуетесь неким его подмножеством или надмножеством, логическим или же техническим. Эти лекции охватывают (в указанном порядке) следующие основные подмножества C++:

- Old plain C – подмножество C в языке C++, основные сложности, сходства и отличия
- Object-oriented C++ – объектно-ориентированные возможности в C++
- Template meta-programming – шаблоны и метапрограммирование
- STL – стандартную библиотеку, её основные идеи и возможности
- Exceptional C++ – исключения, гарантии исключений, проектирование с учётом исключений
- Функционал нового стандарта C++11 – rvalue references и lambda-expressions
- Расширения GNU, популярные библиотеки, в частности Boost и "несовершенные" решения в языке

Для успешной разработки и поддержки кода на C++, каждую из его частей необходимо знать (хотя бы поверхностно). Сложность изучения языка C++ делает абсурдной задачу уложить его в 14 лекций (да даже и в 140 лекций, если уж на то пошло), но при достаточной работе дома, это должно дать хороший старт.

1.4 Домашняя наработка по первой части

1. Найти, скачать и бегло посмотреть стандарты о которых шла речь на этой лекции. В дальнейшем необходимость консультироваться с этими документами будет периодически возникать.

2. Пусть дан код на C:

```
1 foo(const char **p) { }
2
3 main(int argc, char **argv)
4 {
5   foo(argv);
6 }
```

   Задача: охарактеризовать этот код с точки зрения стандарта C90, указать причины, по которым могу возникнуть проблемы. То же самое для C99 То же самое для C++89 То же самое для C11 То же самое для C++11

Ключ к разгадке загадочных сообщений об ошибках здесь – указатели на неквалифицированные типы (но там ещё много нюансов).

В основном в этих лекциях рассматривается стандарт C++98, с небольшими заходами в C++11 особенно в конце.

2 Корни, кровавые корни

Язык C++ невозможно ни понять, ни оценить, если начинать осваивать его с его высокоуровневых возможностей (как иногда рекомендуют делать). Так или иначе, но C++ это не Java и программирование на нем без понимания того, что происходит под капотом, чревато крайне неприятными сюрпризами. Этот раздел будет посвящён C-подмножеству языка C++.

2.1 Простые задачи для языка C

Предполагается, что человек, слушающий этот курс, обладает хотя бы минимальным знанием языка C в объёме стандарта 2011-го года.

Предлагается решить следующие задачи (у доски каждому студенту, в случайном порядке):

- Перевернуть текстовую строчку
- Найти подстроку в строке
- Найти старший установленный бит в длинном слове
- Перевернуть односвязный список в памяти
- Написать быструю сортировку массива

Несмотря на то, что существенная часть этой лекции посвящена повторению языка C, я во-первых призываю всех читать и перечитывать Кернигана и Ричи, а во-вторых практиковаться в C, не забывая о нём при изучении C++. Лично я предпочитаю C и решения в стиле C за их простоту и близость аппаратуре.

Начать следует, как и полагается, с азов.

2.2 Дьяволы деталей синтаксиса C

Пусть стоит задача прочитать простое объявление на языке C

```
1 char *j[20];
```

Что это? Ответ: да, это массив указателей. Вопрос: кто запишет указатель на массив? Ответ:

```
char (*j)[20];
```

Вопрос: есть ли разница между следующими объявлениями:

```
const int *x;
int const *x;
int * const x;
```

Ответ: между первым и вторым нет, между вторым и третьим очень существенная разница. Во втором случае (как и в первом) речь идёт о константном указателе на не константные данные. В третьем случае речь идёт о константном указателе на не константные данные.

Вопрос: кто запишет константный указатель на константные данные? Сделать это надо двумя способами, не меньше.

Вопрос: кто понимает, что значит ключевое слово volatile.

Вопрос: Запишите теперь константый указатель на волатильные данные. Имеет ли он смысл? Можете ли вы представить ситуацию, когда вам может понадобиться волатильный указатель на константные данные?

Теперь самостоятельно прочитайте объявление:

```
char* const *(*next)();
```

Алгоритм чтения таких объявлений следует из стандарта и в данном случае довольно прост:

- Идём к имени переменной “next”
- Группируем её с содержимым её скобок: “next is a pointer to”
- За пределами скобок объявление функции имеет больший приоритет, поэтому “next is a pointer to a function, returning”
- Далее обрабатываем слева звёздочку, имеем: “next is a pointer to a function, returning pointer to”
- И далее парсим char * const: “next is a pointer to a function, returning pointer to constant pointer to character”

Его можно обобщить и изобразить на рисунке:

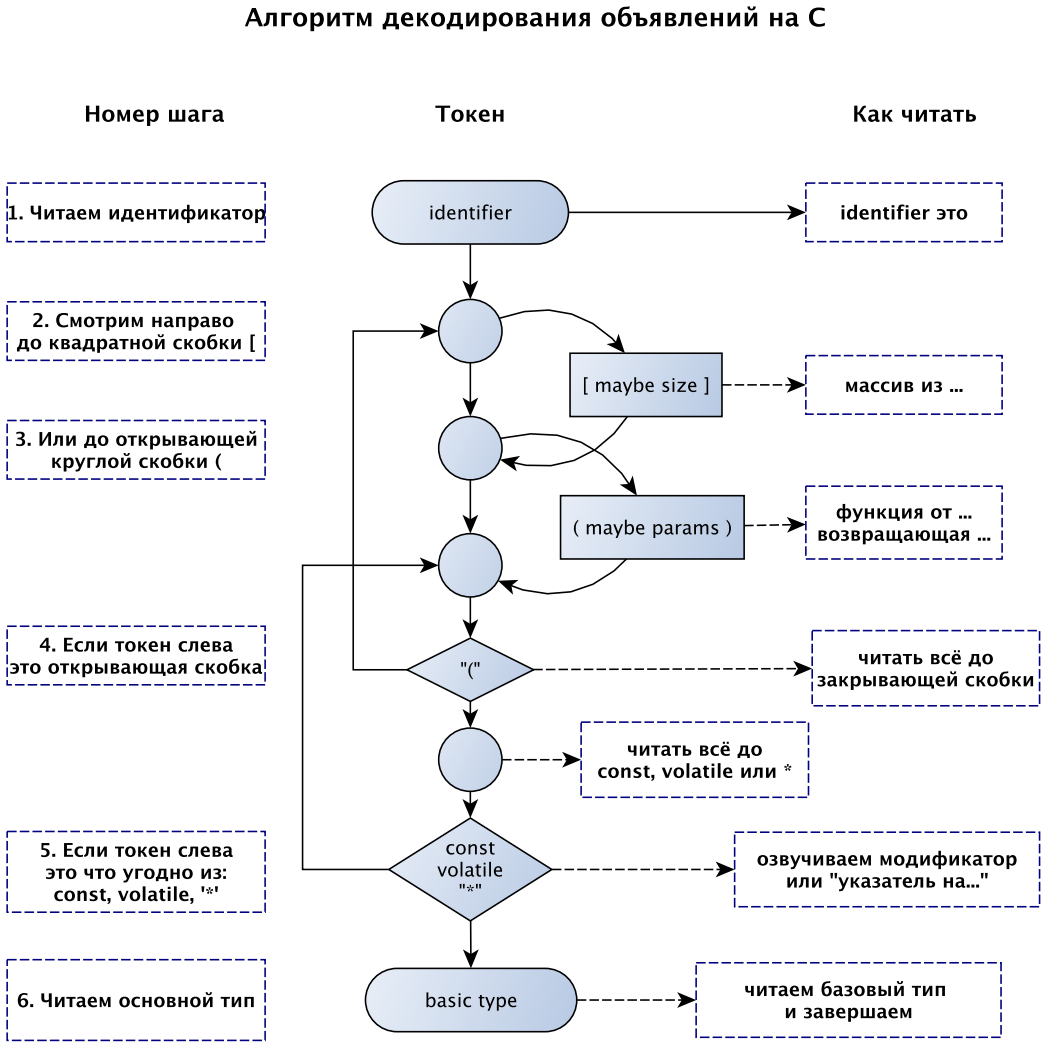

Рис. 1: Алгоритм разбора объявлений на C

Домашняя наработка: по этому алгоритму написать программу, которая парсила бы произвольное объявление на C. Дополнительно: учесть возможность enum, struct, union.

С учётом новых знаний попробуйте прочитать самостоятельно:

```
1 char *(*c[10])(int **p);
```

Ответ: c is array of 10 pointers to functions, accepting pointer to pointer to int and returning pointer to char.

2.3 Ваш друг typedef

Рассмотрим прекрасное объявление типа:

```
void (*signal(int sig, void (*func)(int)) ) (int);
```

С помощью typedef его можно переписать, доставив гораздо меньше боли глазам читающего (и без риска посадить случайную опечатку).

```
typedef void (*ptr_to_func) (int);
ptr_to_func signal(int, ptr_to_func);
```

Умение читать запутанные объявление не означает необходимости их писать (если вы не участвуете в международном конкурсе по обфускации программ). Где возможно, пользуйтесь typedef, это хороший стиль. Но ваш личный хороший стиль (с другой стороны) это не основание не иметь навыка чтения запутанных объявлений. Каждый программист в жизни имеет дело с тоннами унаследованного кода, где встречается всякое.

У typedef есть некоторые опасности, которые часто ускользают от новичков. Так например typedef может объявить сразу несколько синонимов типов:

```
typedef int *ptr, (*fun)(), arr[5];
```

Это считается дурным тоном, старайтесь этого избегать. Ещё более дурным тоном считается зарыть typedef вглубь объявления, то тоже позволяется синтаксисом.

```
unsigned const long typedef int volatile *kumquat;
```

Опытный программист на языке C часто чувствует себя свободнее с препроцессором, скажем вместо

```
typedef int * int_ptr;
```

Есть соблазн написать

```
#define INT_PTR int *
```

Но здесь есть тонкая ловушка:

```
typedef int * int_ptr;
#define INT_PTR int *
INT_PTR x, y; /* x is pointer, y is int */
int_ptr w, v; /* w, v are pointers */
```

Здесь переменные x и y будут разных типов, а w и v – одного типа. Предпочитайте const, enum и inline при программировании на C++ Пример с typedef показывает нам, что использование возможностей языка лучше, чем использование препроцессора. Есть такие возможности C++, которые кардинально отличают программирование на C-подмножестве C++ и использование которых является хорошим тоном. Например всегда следует предпочитать введение символьных констант препроцессорным объявлениям.

```
#define ASPECT_RATIO 1.653
const double AspectRatio = 1.653;
```

В чём разница? В первом случае препроцессор сделает текстовую подстановку 1.653 везде, где вы использовали ASPECT_RATIO, во втором случае компилятор объявит (и поместит в таблицу символов для отладочной информации) символьную константу. Ещё в большей степени это касается работы с перечислениями (строго говоря, она и в C была введена в C99).

```
/* C-style */
#define ASM 1
#define AUTO 2
#define BREAK 3
/* C++-style */
enum keyword {ASM = 1, AUTO, BREAK};
```

Здесь кроме очевидной экономии места, вводится тип keyword, который может принимать только определённые в enum значения, что также является преимуществом.

Ещё сильнее C++ выигрывает для определения небольших функций, где в C были выгодны макросы:

```
/* C - style */
#define MAX(a, b) ((a) > (b) ? (a) : (b))
/* C++ - style */
template <typename T> inline const T
max (const T a, const T b)
{
  return (a > b) ? a : b;
}
```

Давайте рассмотрим контекст использования

```
int a = 5, b = 0;
```

```
2 int c = MAX(++a, b);
3 int d = MAX(++a, b+10);
```

Вопрос: что будет содержаться в c и в d? А что если подставить вызов функции max? Ещё лучше объявление функции max будет себя вести при использовании ссылок (мы познакомимся с ними далее).

Кстати, поскольку шаблон раскроется на этапе компиляции, а вызов max почти всегда будет проинлайнен, эффективность C++ метода как минимум не страдает.

Более того, почти всегда, когда в C использовался void* для передачи параметров неопределённого типа, в C++ можно написать более эффективный код, используя шаблоны. Это обязательно будет предметом обсуждения следующих лекциях.

2.4 Различие объявлений и определений

До сих пор действия с объявлениями и определениями выполнялись интуитивно, без полного понимания того, что это такое. Давайте теперь посмотрим на детали.

Вопрос: охарактеризуйте эти две строчки, что из них является определением, что объявлением.

```
1 extern int *x;
2 extern int y[];
```

Ответ: первое это определение переменной x как указателя на целое. Второе это объявление массива, определённого где-то ещё.

В языке C это довольно просто и это надо просто знать.

Определение чего-то это фактическое выделение для него места в памяти. Оно встречается лишь один раз во всех единицах трансляции на каждую определяемую переменную.

Объявление это просто описание типа переменной, определённой где-то ещё. Объявления нужны, потому что имя в C++ может быть использовано только тогда, когда оно было объявлено.

Можно запомнить мнемоническое правило чтобы не путать объявление с определением, а в англоязычной литературе declaration и definition, нужно смотреть на словарное упорядочение. Буква “б” расположена раньше, чем “п”. Значит в словаре слово “объявление” будет раньше, чем

“определение” и так же declaration в английском словаре идёт раньше, чем definition. И так же в программе – объявление всегда должно идти раньше определения.

Важные термины здесь – полный и неполный тип. Если переменная (структура, класс, массив) только объявлена, но не определена то её тип считается неполным. Полным тип переменной становится только в точке, в которой встречается её определение. Рассмотрим на примере структур.

```
1 struct t; /* incomplete type t */
2
3 int foo(t *p);
4
5 /* definition */
6 struct t {
7   int x;
8   int y;
9 };
```

2.5 Lvalue и rvalue

Разговор о C++ невозможен без введения таких фундаментальных понятий, как lvalue и rvalue. Для начала рассмотрим пример, который многим из вас может показаться простым.

```
1 x = y;
```

Что здесь написано? Здесь написано – взять адрес переменной x и записать по этому адресу значение переменной y. В этом выражении присваивания, переменная x находится слева, а y справа и стандарт C++98 вводит специальные термины rvalue (right-hand-side value) и lvalue (left-hand-side value) интуитивно понимаемые как “нечто, что может быть справа (слева) в выражении присваивания”.

Итак, какие ограничения этот пример накладывает на x? Похоже, что x должен быть полного типа, не константым, и иметь определённое местоположение в памяти (быть адресуемым). Есть ли ограничения на y? Да есть. Он должен быть полного типа и этот тип должен быть совместим с типом x по присваиванию.

Это типичный пример того, как компилятор может решить из контекста (в данном случае из положения справа или слева от присваивания)

будет ли он использовать адрес переменной или её значение в своих фактических вычислениях. Такая возможность у компилятора есть, потому что адрес переменной полного типа всегда известен во время компиляции и нет необходимости заставлять программиста его специально получать.

```
1 *(&x) = y;
```

Возможно, кстати, писать так было бы честнее. Интересные вещи начинаются, когда дело доходит до массивов и указателей.

2.6 Массивы и указатели

Многие программисты заучивают правило для новичка: массивы в C это указатели и наоборот. Это правило, пожалуй, действительно полезно для новичка. В конце концов, любая ссылка на элемент массива в тексте программы действительно может быть переписана как указатель и наоборот. Есть ещё много контекстов, в которых массивы и указатели взаимозаменяемы.

Но для человека, входящего в детали языка и читающего стандарт, очевидно, что в целом это "правило" не работает. Массивы и указатели – очень разные вещи и в C и в C-подмножестве C++. Разработчики по-разному с ними работают и по-разному с ними отдыхают.

Представьте запись:

```
1 char b[] = "abcdefgh", a, c;
2 int i, j;
3 a = b[i];
4 c = j[b];
```

Что здесь написано на строчке 3? Здесь написано – взять значение b, прибавить к нему значение i и прочитать по получившемуся адресу значение для записи в x. Поскольку у компилятора всегда есть контекст, запись на строчке 4 так же легальна, хотя и считается дурным тоном.

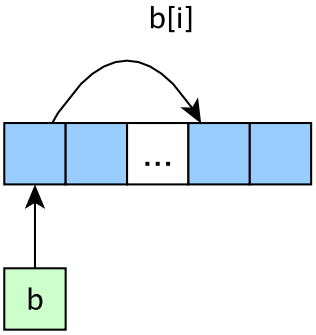

Рис. 2: Визуальное представление массивов

Теперь как это происходит в случае указателей:

```
1 int *p;
2 int c;
3 c = *p;
```

Что здесь написано на строчке 3? Здесь написано – взять значение p и используя его как адрес, взять по этому адресу значение для записи в c.

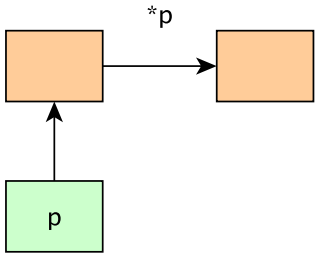

Рис. 3: Визуальное представление указателей

Важно понять: указатели (неконстантные) являются lvalue. В принципе вы можете записать:

```
1 int p = 3;
2 int *p = (int *) p;
```

Этот код является дурным тоном, он будет плохо переносим и вы совершенно точно используете тут указатель не по назначению, но важно, что вы можете это сделать. Указатель это честная ячейка памяти. С массивом этот номер не пройдёт. Указатели хранят адреса данных. Массивы хранят сами данные.

Теперь представьте два варианта записи:

```
1 char a[] = "abcdefgh";
2 char *b = "abcdefgh";
3 assert (a[3] == b[3]);
```

Вопрос: есть ли разница между этими двумя записями и какую вы предпочтёте? Почему?

Верный ответ: строчка 1 предпочтительней, чем (устаревшая, с Wall + Werror выдаст “error: deprecated conversion from string constant to `char *`”) строчка 2 и они имеют разную семантику. Память под массивы выделяется автоматически (и строчка 1 подразумевает неявный memset) но память никогда автоматически не выделяется под указатели, поэтому для построения динамических структур данных (например, связных списков) используются указатели, а не массивы.

Между прочим, строковые литералы в инициализации указателей это счастливое исключение. Строчка 2 в примере ниже:

```
float a[] = {1.0, 2.0, 3.0}; /* ok */
float *b = {1.0, 2.0, 3.0}; /* ka-boom! */
```

Не будет скомпилирована.

Для одномерных массивов есть золотое правило – они могут быть переписаны с помощью указателей везде, где они выступают как rvalue, то есть в правых частях присваиваний и аргументах функций, но не могут быть переписаны указателями в иных случаях, например в объявлениях и определениях.

```
assert(a[3] == *(a+3)); /* ok */
int f (int *x) {}
int a[9];
f(a); /* ok again */
```

Во всех этих случаях массивы взаимозаменяемы с указателями только потому, что компилятор неявно трактует (и по стандарту должен трактовать) имя массива a как адрес его нулевого элемента `&a[0]`.

Но это всё были одномерные случаи. Что же с многомерными массивами?

2.7 Многомерные массивы

Язык C и C-подмножество языка C++ не поддерживают семантику "настоящего" многомерного массива на уровне языка. То, что поддерживается, является по факту "массивом массивов ... массивов".

К ним применяются те же правила, что к обычным массивам, например

```
assert(a[3][4] == *(*(a+3) + 4)); /* ok */
```

Причём последние индексы идут последними. Так же и с инициализацией:

```
float a[][3] = {{1.0, 2.0, 3.0}, {4.0, 5.0}}; /* ok */
```

Причём наиболее вложенные скобки относятся к последним индексам. Первый индекс ясен из контекста и его можно опускать в инициализаторе.

Если часть вложенных инициализаторов пропущена, они считаются нулевыми, поэтому, например двумерные массивы символов могут быть проинициализированы последовательностями строковых литералов.

При передаче многомерных массивов в функции может быть опущен только первый индекс в прототипе (и он может быть любым). Остальные индексы у формального и фактического аргументов должны совпадать.

2.8 От указателей к ссылкам

Очень важной особенностью C++ является введение довольно низкоуровневой, но принципиально новой конструкции – ссылок. Ссылка это альтернативное имя переменной.

```
1 int x = 5;
2 int &xref = x;
3 xref += 1;
4 assert (x == 6);
```

Каждая ссылка обязательно должна быть инициализирована в точке определения. "Ссылка сама по себе" так же как "нулевая ссылка" не могут существовать. Указатель сам по себе является местом для хранения данных, ссылка – это просто имя. Поэтому ссылки не нуждаются в явном разыменовании – каждое обращение к ним это уже разыменование. Поэтому различия существенны:

```
1 int x[2] = {10, 20};
2 int &xref = x[0];
3 int *xptr = &x[0];
4 xref += 1;
5 xptr += 1;
6 assert ((xref == 11) && (*xptr == 20))
```

В целом при программировании на C++ следует повсюду предпочитать использование ссылок использованию указателей.

2.9 От malloc и free к new и delete

Для управления динамической памятью в языке C использовались библиотечные функции malloc и free. Они остались в C++, но их использование не рекомендовано и считается дурным тоном, поскольку C++ предоставляет гораздо более гибкие возможности с помощью ключевых слов new, delete и delete[]

```
1 struct S *t = new S; /* t = (struct S*) malloc (sizeof(struct S
    )); */
```

```
delete t; /* free(t); */
struct S *a = new S[5]; /* t = (struct S*) malloc (sizeof(
    struct S) * 5); */
delete[] a; /* free(a); */
```

Следует говориться, что для C-подмножества эти возможности пожалуй менее гибкие и более ограничивающие. Нужно помнить о парности new/delete и new a[]/delete[] a и нет возможности сделать realloc. Вся сила новых ключевых слов раскроется позже, когда будут рассматриваться элементы ООП. Пока что вы можете просто запомнить их и начинать применять.

2.10 От приведения в стиле C к приведению в стиле C++

Язык C, поскольку он разрабатывался с интенцией отображения один на один в машинные типы имеет слабую типизацию. Кусок памяти который хранит int или указатель на int или механическую структуру из чего-нибудь, легко приводится к другому такому же куску памяти.

```
int *b, *c;
char d;
const int *e;
int a = (int)(b + c);
d = (char) a;
b = (int *) e;
```

Язык C++ содержит гораздо более развитую систему типов и позволяет определять типы, обладающие состоянием и поведением, о которых пойдёт речь в следующих лекциях. Поэтому, несмотря на то, что формально C++ унаследовал от C способ непосредственного приведения C-style cast, применять его считается крайне плохим тоном. Вместо этого следует применять `static_cast`, `reinterpret_cast` или `const_cast`. Чаще всего вы будете применять `static_cast`. Он записывается так:

```
char a;
int b = static_cast<int>(a);
```

В принципе это ничем не отличается от C-стиля, рассмотренного выше. Запись чуть более уродлива, зато чуть лучше бросается в глаза в коде программы. Разница в том к чему можно применять `static_cast` , а к чему C-cast. Последний применяется к чему угодно. `static_cast`

применяется только к переменным, совместимым по статическим типам. Его можно использовать в преобразовании int* к float* (и то и другое – указатели).

Гораздо более редкий `const_cast` нужен для снятия константности и волатильности. С его помощью можно привести const int * к int *.

```
1 const int *a;
2 int *b = const_cast<int*>(a);
```

Вы конечно понимаете опасность этих игр – в памяти, выделенной для `const int*`, компилятор размещает данные, изменения которых не ожидает. В тот момент, когда вы принудительно снимаете константность и изменяете (пытаетесь изменить) нечто объявленное ранее константным, вы стреляете себе в ногу.

И самый редкий `reinterpret_cast` существует для непереносимых низкоуровневых приведений, которые, тем не менее, иногда нужны.

```
1 int a;
2 int *b = reinterpret_cast<int>(a);
```

Любое использование `reinterpret_cast` (как и C-cast) компрометирует вашу программу. Но случаи использования `reinterpret_cast` читающему ваш код будет куда проще найти и вычистить.

Выучить и использовать эти три оператора не намного сложнее, чем использовать обычные преобразования в стиле C, но в дальнейшем они сослужат вам отличную службу, упрощая поддержку кода и не давая посадить тяжело обнаружимых ошибок приведения.

2.11 Перегрузка функций, аргументы по умолчанию и искажение имён

Вопрос к слушателям сколько функций вычисления квадратного корня вы можете назвать из стандартной библиотеки языка C?

Правильный ответ: три (7.12.7.5) `sqrtf`, `sqrt` и `sqrtl`. Три функции с разными именами понадобилось вводить потому, что они принимают аргументы разных типов, а язык C предоставляет достаточно сильную гарантию того, что любое имя, использованное в вашей программе будет отображено в ассемблер вашей целевой машины.

Язык C++ такой гарантии не даёт. Вместо этого он согласен сделать

эту (и многую другую) работу за вас посредством встроенного искажения (манглирования) имён. Используя C++ вы можете написать три функции с одинаковыми именами, но различными типами:

```
char foo (char x) { return x; }
int foo (int y) { return y; }
long long foo (long long z) { return z; }
```

Посмотрим во что они были откомпилированы в ассемблер:

```
_Z3fooc:
...
_Z3fooi:
...
_Z3foox:
...
```

Конвенции манглирования не документированы и являются implementation-defined, закладываться на них не надо. Но грамотно использовать механизм перегрузки функций в C++ бывает очень выгодно для облегчения читаемости вашей программы.

Домашняя наработка: посмотрите как работает манглирование в вашем любимом компиляторе. Можете ли вы установить некие закономерности?

Также удобная концепция (и снова проистекающая от отсутствия обязательств C++ быть близким к машине) это аргументы по умолчанию. Посмотрим как они могут быть заданы:

```
char foo (char x = 0) { return x; }
int foo (int y = 7) { return y; }
long long foo (long long z = 0xafff) { return z; }
```

Функция не может быть перегружена по значению аргумента по умолчанию. Для перегрузки вы можете использовать только сигнатуру – возвращаемый тип функции и типы её аргументов.

Перегрузка функций и аргументы по умолчанию сильно упрощают работу с именами функций в C++, перенося всю её тяжесть на плечи компилятора и вам следует научиться использовать эти возможности правильно.

2.12 Пространства имён, using и поиск Кёнига

Каждое объявление в C++ принадлежит некоторому пространству имён. Глобальное пространство имён это префикс через два двоеточия, а все стандартные функции принадлежат пространству имён std. Напишем hello world с явным указанием пространств имён.

```
1 #include <cstdio>
2 const char * const helloworld = "Hello, world";
3 int main(void)
4 {
5   std::printf("%s\n", ::helloworld);
6   return 0;
7 }
```

Обратите внимание на включение `<cstdio>` вместо привычного `<stdio.h>` хедера. Все заголовочные файлы к которым вы привыкли в C, сохранены в C++. Но хорошим тоном считается писать унаследованные заголовочные файлы в C++ conforming виде, то есть `<cXXX>` вместо `<XXX.h>`.

Засорять собственными именами, такими как `::helloworld`, глобальное пространство имён это на самом деле крайне плохая идея. Несколько лучше объявить своё пространство имён, включив туда всё, что специфично именно для вашей программы.

```
1 #include <cstdio>
2 namespace hellowapp {
3 const char * const helloworld = "Hello, world";
4 }
5 int main(void)
6 {
7   std::printf("%s\n", hellowapp::helloworld);
8   return 0;
9 }
```

Допустимо объявлять вложенные пространства имён с произвольным количеством уровней вложенности. При помещении в пространство имён функции с большим телом, вполне достаточно поместить в пространство имён явно только объявление (например в заголовочном файле), указав пространство имён при определении (например в файле реализации).

```
1 /* foo.h */
2 namespace foo {
```

```
3  int bar(void);
4  }
5  /* foo.cpp */
6  int foo::bar(void) { /* ... */ }
```

Пространства имён являются областями видимости (как блоки из фигурных скобок) и подчиняются тем же правилам – если имя указано в охватывающем пространстве имён оно может быть использовано без квалификации. Но в отличии от блоков скобок они могут быть поименованы и тогда переменная или функция с квалификацией может быть использована где угодно

```
1  namespace foo {
2  int y;
3  }
4  namespace {
5  int y;
6  void bar(int x) { y = x; } /* ok */
7  }
8  namespace buz {
9  void bar(int x) { foo::y = x; } /* ok */
10 }
```

Здесь приведён пример неименованного пространства имён (по русски это звучит странно). В пространстве имён buz у нас нет доступа к y, объявленному в анонимном пространстве имён, зато есть доступ к `foo::y`. Также можно избежать необходимости постоянно ставить некий префикс (скажем std) если включить это пространство имён в текущее с помощью директивы using. Этой директивой можно включить и одно имя и целое пространство.

Когда компилятор видит, например, вызов функции, имя этой функции он будет искать:

сначала в области видимости вызова и текущем пространстве имён

далее в пространствах имён аргументов, включая их классы и все базовые классы

Этот трюк называется “поиск Кёнига” по имени человека, который его придумал и ввёл в стандарт C++98.

```
1  namespace ns {
2    struct S {};
3    void f (S x) {}
```

```
}

void g(void)
{
  ns::S x;
  f(x); /* ok, f is ns::f */
}
```

Впрочем, если компилятор встречает вызов фукции из функции-члена некоего класса, то иные члены этого класса и его родительских имеют приоритет над функциями, найденными на основании информации о типах аргументов.

Любая стандартная функция в C++ принадлежит пространству имён std. Каждый раз писать `std::printf` или `std::memcpy` бывает накладно. Чтобы этого избежать, добавьте строчку using namespace std после включения хедеров. Директива using name также может применяться чтобы внести из произвольного пространства имён только одно имя.

```
#include <cstdlib>
#include <cstdio>
using namespace std;

int
main(void)
{
  printf("%s\n", "Hello, world");
  return 0;
}
```

Все правила для пространств имён нужно хорошо знать и ещё раз освежить в стандарте по мере того, как вы забудете то, о чём я говорил здесь.

2.13 Мелкие отличия C-подмножества C++ от ANSI C

- Функция `main()` в C++ не может быть вызвана из пользовательского кода. В языке C это разрешено, хотя и несколько необычно.
- Прототипы функций обязательны в C++, но опциональны в C.

- Имена, определяемые через `typedef` не могут совпадать с существующими именами структур в C++, но могут в C (последний требует явной квалификации `struct`).
- При присвоении к `void *` указателю указателя на иной тип, C++ требует приведения (C не требует, но оно считается хорошим тоном).
- C++ вводит более десяти новых ключевых слов. Они могут быть использованы ка к идентификаторы в программе на C, но компилятор C++ выдаст ошибку.
- В языке C++ объявление переменной может появится везде, где может быть выражение; в C, объявления должны быть в начале блока.
- Имя struct во внутренней области видимости скроет такое же имя любой переменной во внешней области видимости в C++, но не в C.
- У символьных литералов тип char в C++, но тип int в C. То есть `sizeof('a')` даёт 1 в C++, но может дать большее значение в C.

2.14 Домашняя наработка по второй части

Определить петлю в односвязном списке

Выкинуть комментарии из программы на C

Написать простой калькулятор на C++ в стиле K&R C

3 Объектно-ориентированное счастье

Как известно самой целью создания C++ Бьёрном Строструпом было добавление ОО-возможностей к C, поэтому изначально язык назывался “C с классами”. Известно что Строструп вдохновлялся языком Simula, но сейчас уже нельзя оценить насколько это была удачная идея, поскольку этот язык канул в лету. С другой стороны, известному гуру ООП, Кенту Беку, приписывается высказывание: “Это я придумал термин *объектно-ориентированный* и я не имел в виду C++”. Так или иначе, но C++ действительно обладает уникальной среди современных ОО-языков моделью объявления и инстанциирования классов. Многим нравится богатство и гибкость её возможностей, многие в ужасе отползают. Впрочем, давайте начнём с основ.

3.1 Структуры в C и в C++, POD и NPOD

В языке C структура являлась способом ввести пользовательский тип, являющийся механическим объединением разнородных данных:

```
typedef struct pair
{
  int x;
  int y;
} pair_t;

pair_t
transpose_pair(pair_t pa)
{
  pair_t pr = {pa.y, pa.x};
  return pr;
}
```

Такие типы возможны и в C++ и они называются POD-типами (от английского Plain Old Data). Но в C++ была также добавлена принципиально новая возможность группировать данные с методами их обработки внутри структуры:

```
struct pair_t
{
  int x;
  int y;
```

```
5   pair_t transpose_pair(void);
6 };
7
8 pair_t
9 pair_t::transpose_pair(void)
10 {
11   pair_t pr = {this->y, this->x};
12   return pr;
13 }
```

Получившийся тип несколько удобнее в работе. Но при этом теряются гарантии по расположению в памяти и размерам (которые с учётом выравнивания и в C в общем-то были довольно призрачными). Обратите внимание что в C++ была исключена необходимость добавлять struct к символьному имени структуры, что делает ненужным оставшийся в C-style коде typedef тэга структуры на её имя.

Кстати имя метода `transpose_pair` структуры `pair_t` в C++ также будет манглировано (вспоминаем прошлую лекцию) и в ассемблере встретится, например в виде:

```
1 _Z14transpose_pair4pair:
```

Определённая так функция называется методом. Объявление метода происходит внутри определения структуры, определение метода имеет явную квалификацию того к чему метод относится. Обратите внимание, что символ сдвоенных двоеточий такой же, как и в случае пространств имён. Он дословно означает пространство имён, задаваемое структурой.

Использование this внутри метода позволяет получить данные той структуры, для которой метод был вызван, так что нам не надо специфицировать исходную структуру как параметр. Его использование можно опустить.

3.2 Инкапсуляция и игра в мячик

Представим, что вас попросили разработать тип данных, который будет использован для моделирования полёта материальной точки в двумерном мире (высота, длина). Мяч может лететь свободно, для чего у него вызывается метод fly(double t) или его можно толкнуть, придав ему определённую скорость под определённым углом (после чего например опять отправить в полёт и так далее). Вы напишете нечто вроде:

```
struct ball
{
  double t;
  double vx;
  double vy;
  double x;
  double y;
  void push(double v, double alpha);
  void fly(double time);
}
```

Представляет ли эта структура данных абстракцию мяча, о которой вас просили? Нет, не представляет. Каждый пользователь вашего "мяча" может произвольно менять его координаты. Это означает, что в плохо отлаженной программе симуляции, ваш мяч сможет свободно "телепортироваться", а это явно не то, чего ждут от законченной модели.

```
int main(void)
{
  ball a_ball;
  a_ball.vx = 5.0; /* sad, but ok */
  a_ball.push(5.0, 0.75); /* ok */
  a_ball.x = -1.0; /* hmm... still ok */
}
```

Хуже того, время может быть свободно переставлено вперёд или назад. Но пусть даже никто не ошибся и всё написано правильно. А потом... возникла необходимость "запустить" ваш мяч в многопользовательской среде. Всё пропало – для того, чтобы вставить синхронизацию, переписывать придётся каждый участок кода где ссылались на эти поля.

3.2.1 Конкретные классы

Для разграничения состояния модели от её поведения и более гибкого управления поведением, в C++ были введены классы. Простые классы, без использования полиморфизма и без иерархий наследования, называются "конкретными классами". Конкретные классы – мощный и полезный инструмент для поддержания консистентности абстракции. Перепишем модель мяча, создав его класс

```
class ball_t
{
```

```
3  private:
4    double m_x;
5    double m_y;
6    double m_vx;
7    double m_vy;
8    double m_t;
9  public:
10   void push(double a_v, double a_alpha);
11   void fly(double a_time);
12 };
```

Модификатор public означает, что любой пользователь типа ball имеет доступ к этим методам или данным.

Модификатор private означает, что доступ к соответствующим методам и данным имеют только методы этого класса.

```
1  void
2  ball_t::push(double a_v, double a_alpha)
3  {
4    assert(a_v > 0);
5    m_vx += a_v * cosf(alpha); /* ok */
6    m_vy += a_v * sinf(alpha);
7  }
8
9  int main(void)
10 {
11   ball_t ball;
12   ball.vx = 5.0; /* fail */
13   ball.push(5.0, 0.75); /* ok */
14   return 0;
15 }
```

Хорошим тоном считается закрывать данные, составляющие состояние объекта и открывать функции, составляющие его поведение.

Обратите внимание на опущенный `this` в коде `ball_t::push`, это допустимо.

3.2.2 Инициализация и уничтожение

Отсутствие доступа к состоянию означает, что при создании объекта он должен уметь сам установить своё состояние, а при уничтожении

– освободить свои ресурсы. Для этого в класс вводятся конструктор и деструктор – специальные функции, вызывающиеся при создании и уничтожении объекта. Например у класса мяча, конструктор может устанавливать начальное положение, деструктор же может быть тривиальным.

```
public:
  /* Construction & destruction */
  ball(double a_x = 0.0, double a_y = 0.0):
    m_x(a_x), m_y(a_y), m_vx(0.0), m_vy(0.0), m_t(0.0) {}
  ~ball(void) {}
```

Обратите внимание на список инициализации у конструктора в этом примере кода. Можно, конечно, написать инициализацию в теле конструктора, но использование списков инициализации является лучшей идеей. По умолчанию, удовлетворяющий стандарту языка C++ компилятор позаботится о вас, сгенерировав вам конструктор и деструктор по умолчанию (позднее мы поговорим о степени этой заботы и её оборотных сторонах). Конструктор по умолчанию вызывает конструкторы всех членов класса, у которых они есть, деструктор – их деструкторы.

Важная тема в конструкторах это их использование для задания неявного преобразования типа. Неявные преобразования есть и в C, там они называются type promotions, некоторые из них (действуют и в C++) сведены в таблицу ниже (здесь anytype это любой встроенный тип, совместимый по операции но не перечисленный выше):

```
anytype `op` long double => long double `op` long double
anytype `op` double => double `op` double
anytype `op` float => float `op` float
anytype `op` unsigned long long => unsigned long long `op`
   unsigned long long
anytype `op` long long => long long `op` long long
anytype `op` unsigned long => unsigned long `op` unsigned long
anytype `op` long => long `op` long
anytype `op` unsigned int => unsigned int `op` unsigned int
anytype `op` int => int `op` int
```

Но в C++ неявные преобразования также пробуются компилятором для пользовательских типов. Определение в пользовательском типе конструктора, который может быть истрактован как конструктор с одним аргументом (считая аргументы по умолчанию частично подставленными всюду кроме первого аргумента), считается определением неявного преобразование из аргумента конструктора к этому типу. Это может иметь

неприятные последствия:

```
int emulateBall(ball b);

...

int foo(void)
{
  emulateBall(1.0); /* ??? but legal */
}
```

Верный способ определить конструктор, чтобы явно заявить компилятору, что он не поддерживает неявного преобразования это определить его с ключевым словом explicit

```
  explicit ball_t(double a_x = 0.0, double a_y = 0.0):
    m_x(a_x), m_y(a_y), m_vx(0.0), m_vy(0.0), m_t(0.0) {}
```

Это важное решение при проектировании и его надо принимать осознанно. Лепить explicit куда ни попадя – дурной тон (скажем это ключевое слово возможно но совершенно не нужно на конструкторе более чем с одним аргументом и без аргументов по умолчанию). Но иногда он очень нужен.

3.2.3 Селекторы

Отражает ли созданная до сих пор абстракция физический мяч? Всё ещё нет. Мяч в физическом мире обычно виден пользователю, у которого есть возможность считать его координаты. То есть нам нужны некоторые методы, которые будут, сохраняя состояние мяча, давать возможность прочитать его. Такие методы традиционно называются селекторами.

```
  /* Selectors */
  double get_x(void) const { return m_x; }
  double get_y(void) const { return m_y; }
```

Обратите внимание на const в их объявлении. Внутри объявленного таким образом метода изменить любое поле класса это ошибка компиляции. Исключение составляют поля, объявленные с mutable.

Хорошим тоном является делать селектором любой метод, который теоретически может быть селектором и по логике не должен менять внутреннего состояния объекта.

3.2.4 Статические константы в классе

Задумаемся над реализацией метода fly. Очевидно, нам понадобиться ускорение свободного падения. Логично сделать его константой общей для каждого экземпляра класса `ball_t`, но при этом закрытой, так как это его деталь реализации.

Пишем объявление:

```
1 private:
2 static const double g;
```

Определяем вне определения класса:

```
1 const double ball_t::g = 9.81;
```

И теперь можно написать код полёта:

```
1 void
2 ball_t::fly(double time)
3 {
4   m_x += m_vx * time;
5   m_y += m_vy * time - g * time * time;
6   m_vy -= g * time;
7   m_t += time;
8 }
```

И мяч полетит.

Вообще спецификатор static в классах используется для объявления статических членов, то есть таких атрибутов и методов, которые являются методами и атрибутами класса, а не объекта. Подобно константе **g**, все статические члены которые нуждаются в инициализации должны быть определены вне объявления класса.

3.2.5 Объявления и определения классов

Это важный момент, перекликающийся с затронутой на прошлой лекции темой объявлений и определений. Объявление класса как неполного типа выглядит так:

```
1 class ball_t;
```

С этого момента тип `ball_t` можно использовать по правилам, прописанным в стандарте для неполных типов. Определение класса это объявление всех его методов и полей.

Но внутри определения класса, каждое объявление поля, статического поля или метода это его объявление. Определением нестатического члена считается конструктор класса (поэтому если членом класса является ссылка она должна быть инициализирована в списке инициализации конструктора). Определение метода может как встречаться внутри класса, так и быть вынесено вне его. Определение статического объекта всегда должно быть вне класса.

Давайте сведём всё воедино.

3.2.6 Игра в мячик (код)

```
1  #include <iostream>
2  #include <cmath>
3
4  class ball_t
5  {
6  private:
7    /* internal state */
8    double m_x, m_y, m_vx, m_vy, m_t;
9    static const double g;
10 public:
11   /* Construction & destruction */
12   explicit ball_t(double a_x = 0.0, double a_y = 0.0):
13     m_x(a_x), m_y(a_y), m_vx(0.0), m_vy(0.0), m_t(0.0) {}
14   ~ball_t(void) {}
15
16   /* Selectors */
17   double get_x(void) const { return m_x; }
18   double get_y(void) const { return m_y; }
19
20   /* Behavior */
21   void push(double v, double alpha);
22   void fly(double time);
23 };
24
25 /* Static constant definition */
```

```
26 const double ball_t::g = 9.81;
27
28 void
29 ball_t::push(double v, double alpha)
30 {
31   m_vx += v * cosf(alpha);
32   m_vy += v * sinf(alpha);
33 }
34
35 void
36 ball_t::fly(double time)
37 {
38   m_x += m_vx * time;
39   m_y += m_vy * time - g * time * time;
40   m_vy -= g * time;
41 }
42
43 int
44 main(void)
45 {
46   ball_t ball(0.0, 600.0);
47   ball.push(30.0, 0.75);
48   for(int i = 0; i != 10; ++i)
49     {
50       ball.fly(1.0);
51       std::cout << "x = " << ball.get_x()
52           << ", y = " << ball.get_y() << std::endl;
53     }
54   return 0;
55 }
```

3.3 Классы для управления ресурсами, идиома RAII

При программировании на языке C часто возникает проблема освобождения занятых ресурсов. Ресурс это нечто, что ваша программа может использовать, но при этом должна вернуть системе, иначе могут возникнуть неприятности. Самым частым примером ресурса является динамическая память.

Вопрос студентам: какие ещё вы знаете ресурсы.

Ожидаемые ответы: файловые дескрипторы, мьютексы, шрифты и

кисти, объекты гуя, соединения с бд, сокеты.

Рассмотрим пример файлового дескриптора.

```
int foo_1 (const char * fname, int x)
{
  FILE *f = fopen(fname, "r");
  if (x < 5)
    {
       usef(f);
       fclose(f);
       return somemore(x);
     }
  else if (x > 7)
    {
       int i;
       for (i = 0; i < x; ++i)
         {
           if (another(f, i) != x)
             {
                fclose(f);
                return -1;
             }
           x += i;
         }
      fclose(f);
      return x;
    }

  remainder(f, x);
  fclose(f);
  return 0;
}
```

Этот код существенно подвержен ошибкам и вызов fclose() неприятное количество раз дублируется. Конечно, его можно переписать с использованием goto.

```
int foo_2 (const char * fname, int x)
{
  FILE *f = fopen(fname, "r");
  int retval = 0;
  if (x < 5)
```

```
6     {
7       usef(f);
8       retval = somemore(x);
9       goto cleanup;
10     }
11   else if (x > 7)
12     {
13       int i;
14       for (i = 0; i < x; ++i)
15         {
16           if (another(f, i) != x)
17             {
18               retval = -1;
19               goto cleanup;
20             }
21           x += i;
22         }
23       retval = x;
24     }
25   remainder(f, x);
26
27 cleanup:
28   fclose(f);
29   return retval;
30 }
```

Но я придерживаюсь мнения, что goto всё таки следует считать вредным.

Ещё один метод это использовать обёрточную функцию:

```
1  static int foo_3_1 (FILE *f, int x)
2  {
3    if (x < 5)
4      {
5        usef(f);
6        return somemore(x);
7      }
8    else if (x > 7)
9      {
10       int i;
11       for (i = 0; i < x; ++i)
```

```
12          {
13            if (another(f, i) != x)
14              return -1;
15            x += i;
16          }
17        return x;
18      }
19
20    remainder(f, x);
21    return 0;
22  }
23
24  int foo_3 (const char * fname, int x)
25  {
26    FILE *f = fopen(fname, "r");
27    int retval = foo_3_1(f, x);
28    fclose(f);
29    return retval;
30  }
```

Но этот метод имеет свои недостатки – он как минимум создаёт лишний вызов функции и запутывает код. “Всего лишь ещё одна” обёрточная функция прощённая себе десять раз это плюс десять уровней косвенности при отладке. Поэтому в C++ при программировании (а в C++ исключения добавляют огня, но об этом на одной из следующих лекций) используют важную идиому – идиому обёрточного класса **Resource Aquistion Is Initialization** сокращённо RAII (выделение ресурса это инициализация), создавая для каждого ресурса обёрточный объект, в котором ресурс будет захвачен в конструкторе и освобождён в деструкторе.

```
1  class CFile
2  {
3  public:
4    CFile(const char * fname, const char *acc) { m_file = fopen(
       fname, acc); }
5    ~CFile() { fclose(m_file); m_file = 0; }
6    FILE *fl(void) const { return m_file; }
7  private:
8    FILE *m_file;
9  };
10
```

```
11 int foo (const char * fname, int x)
12 {
13   CFile f(fname, "r");
14   if (x < 5)
15     {
16       usef(f.fl());
17       return somemore(x);
18      }
19   else if (x > 7)
20     {
21       int i;
22       for (i = 0; i < x; ++i)
23         {
24           if (another(f.fl(), i) != x)
25             return -1;
26           x += i;
27         }
28       return x;
29     }
30
31   remainder(f.fl(), x);
32   return 0;
33 }
```

Использованный здесь CFile имеет ряд собственных недостатков и его можно заменить на более безопасную обёртку, но об этом будет идти речь на следующих лекциях. Часто класс для управления ресурсами программист решает написать сам, с квадратными колёсами (как это и было сейчас сделано в целях обучения). В промышленном коде это почти всегда неверно. Дважды подумайте, прежде чем садиться это делать, часто такой класс уже написан. Домашняя наработка: поищите замену для нашего CFile.

3.4 Наследование и сбор урожая

Предположим, вы пишете программу, в которой важной абстракцией является абстракция фрукта. Каждый из ваших конкретных объектов должен обладать своим состоянием и поведением, но все их без исключения можно взвесить (узнать массу) и каждый из них должен обладать массой как частью своего состояния. Вы конечно можете реализовать

метод у каждого:

```
class Plum {
public:
  Plum():m_mass(3){}
  int get_mass (void) const {return m_mass;}
private:
  int m_mass;
};

class Apple {
public:
  Apple():m_mass(5){}
  int get_mass (void) const {return m_mass;}
private:
  int m_mass;
};

class Pear {
public:
  Pear():m_mass(6){}
  int get_mass (void) const {return m_mass;}
private:
  int m_mass;
};
```

Но как тогда написать цикл по всем фруктам в вашей программе или сохранить их все в массив? Гораздо лучше использовать механизм наследования. Для этого нужно сделать базовый класс и наследуем прочие от него.

```
#include <cstdlib>
#include <cstdio>
#include <cassert>

using namespace std;

class Fruit {
public:
  Fruit(int a_mass):m_mass(a_mass){}
  int get_mass (void) const { return m_mass; }
protected:
```

```
12    int m_mass;
13  };
14
15  class Apple: public Fruit {
16  /* Apple-specific code */
17  public:
18    Apple():Fruit(5){}
19  };
20
21  class Pear: public Fruit {
22  /* Pear-specific code */
23  public:
24    Pear():Fruit(6){}
25  };
26
27  class Plum: public Fruit {
28  /* Plum-specific code */
29  public:
30    Plum():Fruit(3){}
31  };
32
33  int main(void)
34  {
35    Apple a; Pear b; Plum c;
36    assert (a.get_mass() < b.get_mass());
37    assert (a.get_mass() > c.get_mass());
38    return 0;
39  }
```

Обратите внимание на модификатор protected. Отмеченные таким образом данные доступны только методам класса и его производным классам. Тут, кстати, есть вопрос терминологии. В некоторой литературе для обозначения базового класса используется “суперкласс”, а для производного “подкласс”. Мне это не нравится, потому что реально в подклассе обычно содержится больше кода, чем в суперклассе и их названия вводят в заблуждение, тогда как “базовый” и “производный” – вполне логичны.

Модификаторы доступа могут применятся при определении наследования.

3.4.1 Открытое наследование

Открытое наследование моделирует отношение “является”.

```
1 class Apple: public Fruit {};
```

Яблоко является фруктом. Это означает, что в любой контекст, где был использован фрукт может быть подставлено яблоко (принцип подстановки Лисков).

Проверим? Попросить студентов поискать такие истинные высказывания о фруктах, куда нельзя подставить яблоко. Очевидно они найдут нечто типа “Некоторые фрукты являются грушами”. Объяснить что это не общее, а частное высказывание. Спросить видят ли разницу между общеутвердительными и частноутвердительными высказываниями. Попросить поискать общеутвердительное.

Свойства открытого наследования:

- компилятор автоматически преобразует объект производного класса к его открытой базе
- открытые члены остаются открытыми

Чаще всего вам придётся использовать именно открытое наследование, но это конечно не повод не знать иные его виды.

3.4.2 Закрытое наследование

Моделирует отношение “является частью реализации”.

```
1 class Car: private Wheel {};
```

Колесо есть часть реализации автомобиля. Есть ряд отличий между закрытым и открытым наследованием:

- компилятор не преобразует по умолчанию значение к закрытой базе
- все члены базы становятся в производном классе закрытыми

Отношение “являться частью реализации” можно также смоделировать композицией, когда один класс включается в закрытую часть другого

```
class Car {
private:
  Wheel a_wheel;
public:
/* */
};
```

Выбор между закрытым наследование и композицией – предмет бесконечного холивара в статьях и книгах по ООП. С моей точки зрения разница не так велика, как считают все эти люди и всё определяется личным опытом и сиюминутным удобством, но решать вам. Защищённое наследование

```
class Apple: protected Fruit
```

Домашняя наработка: посмотреть отличия защищённого наследования от закрытого, составить пример когда использование одного вместо другого может привести к проблемам.

3.4.3 Конструкторы базовых классов

При конструировании каждого класса будут неявно вызваны конструкторы его базовых классов по умолчанию, если они не вызваны явно в списке инициализации. Но в последнем случае, необходимо понимать, что время жизни объекта начинается после того, как отработал его конструктор. Это означает, что следующий код (хотя и корректен с точки зреняи языка) не верен:

```
class A {
  public:
    A (const char *s) {}
    const char *f(void) { return "hello, world"; }
};

class B : public A {
  public:
    B() : A ( s = f() ) {}
  private
    const char *s;
};
```

Здесь сразу две ошибки. Во-первых попытка вызвать метод `f()` базового подобъекта, который ещё не сконструирован в этой точке (поскольку только вызывается его конструктор). Это может прокатить, может нет, но это в любом случае некорректно. Во-вторых попытка инициализировать и потом использовать ещё не существующий член производного класса (до конструирования которого там пока вообще далеко).

Не следует допускать таких ошибок в своём коде.

3.4.4 Наследование интерфейса и наследование реализации

Обратим внимание что теперь код, вычисляющий сумму масс двух фруктов у нас не зависит более от их природы:

```
1  int add_mass(const Fruit &x, const Fruit &y)
2  {
3    return x.get_mass() + y.get_mass();
4  }
5
6  int main(void)
7  {
8    Apple a;
9    Pear b;
10   int total_mass = add_mass(a, b);
11 }
```

Такая запись возможна потому что и в груше и в яблоке есть общая базовая часть “фрукта” и ссылка или указатель на яблоко или грушу могут быть автоматически преобразованы в ссылку или указатель на фрукт.

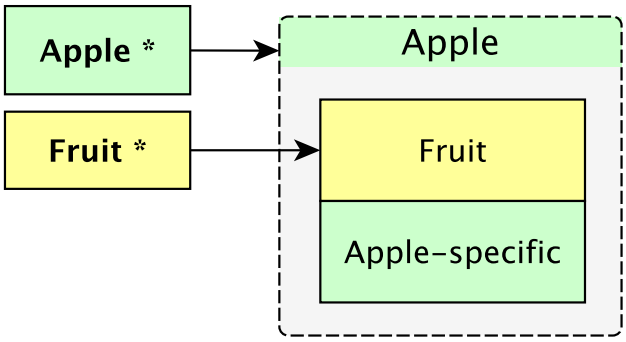

Рис. 4: Преобразование указателя к базе

Мы использовали наследование в двух разных его ипостасях – унаследовав интерфейс базового класса и его код. В принципе наследование кода это почти всегда очень плохая идея. Старайтесь по мере возможности избегать наделять абстрактные классы состоянием и поведением. Для того, чтобы разработать хороший интерфейс фрукта, необходимо сделать его абстрактным, то есть оставить все методы без реализации, предоставив реализацию в производных классах. Давайте поговорим об этом подробнее.

3.5 Полиморфизм

При наследовании интерфейса `get_mass`, наследовалась так же его реализация в общем предке. Но есть вещи, которые предок не знает как сделать за потомка. В этом случае логично предоставить потомку интерфейс, который во время выполнения вызовет нужный метод динамического типа. Простой и расхожий пример – абстрактный класс фигуры совершенно точно не знает как рисуется "фигура вообще", зато это прекрасно знают его конкретные потомки – квадрат, круг, овал – каждый разумеется только о себе.

Рассмотрим как можно реализовать такой механизм без поддержки в языке, чтобы было понятно что делает для нас компилятор, обеспечивая нам поддержку. Изменим метод

```
1  class Fruit {
2    public:
3      typedef int (get_mass_t *)(void);
4      explicit Fruit(get_mass_t *a_entry) : m_entry(a_entry) {}
5      int get_mass(void)
6        {
7          return m_entry();
8        }
9    private:
10     get_mass_t *m_entry;
11 }
```

Разумеется в C++ нет необходимости ударяться в такой хардкор и язык предоставляет достаточно сахара чтобы спрятать под капотом практически все эти детали. Но пример поучителен. Во-первых для каждого метода, вызываемого через потомка необходимо иметь указатель на него (как `m_entry` в примере выше). Вся совокупность таких указателей

называется таблицей виртуальных методов. Важно понимать что таблица виртуальных методов это оверхед по памяти на каждый объект класса. Во-вторых, обратите внимание на то, что метод `get_mass` очень прост и по сути сводится лишь к делегированию вызова по таблице виртуальных методов.

3.5.1 Как подружить Дарта Вейдера с покемоном

Первое, что предоставляет язык C++ это возможность иметь в классе абстрактный, или чисто виртуальный метод, который обязан быть реализован по своему в каждом объекте (и компилятор проверит, что это действительно так). Таким образом мы как бы говорим компилятору что этот метод предназначен только для диспетчеризации вызова к методам произвольных классов. Второе – это убранная под капот таблица виртуальных методов, от которой осталось только слово `virtual` которое мы должны предпослать названию метода в базовом классе, не заботясь ни о заполнении элемента таблицы ни о конструкторе. Простой пример показывает мощь и гибкость этих средств.

```
1  class NamedObject {
2  public:
3    virtual void whoareyou(void) const = 0;
4  };
5
6  class Pokemon : public NamedObject {
7  public:
8    void whoareyou(void) const { printf("Pokemon"); }
9  };
10
11 class DartVeider : public NamedObject {
12 public:
13   void whoareyou(void) const { printf("Dart Veider"); }
14 };
```

Теперь можно писать довольно абстрактный код, полагающийся только на общий интерфейс объектов (способность сказать имя) и отдающий им на откуп детали реализации:

```
1  void
2  friendship(const NamedObject& n1, const NamedObject& n2)
3  {
4    printf("Hello, I am "); n1.whoareyou(); printf("\n");
```

```
5   printf("Hello, I am "); n2.whoareyou(); printf("\n");
6   printf("Lets be friends!\n");
7 }
8
9 int
10 main(void)
11 {
12   Pokemon p;
13   DartVeider d;
14   friendship(p, d);
15 }
```

Это прекрасное свойство называется полиморфизм. Часто говорят, что полиморфизм является единственным оправданием для существования наследования, и что если в вашем классе нет ни одной виртуальной функции, наследование от него – дурной тон.

3.5.2 Статический и динамический тип

Пока мы не стёрли с доски, ещё раз поглядим на функцию `friendship` . Что можно сказать об аргументе `n1`? Всего лишь, что его тип, объявленный в списке аргументов (иначе говоря статический тип) это константная ссылка на `NamedObject`. Это всё, что компилятор может сказать об этом типе статически, без контекста исполнения.

Но динамически при вызове из `main`, функция `friendship` вызывается с двумя параметрами, один из которых имеет тип `Pokemon`, другой – тип `DartVeider`, оба являются производными классами от `namedObject` , то есть, согласно принципу подстановки Лисков действительно могут быть использованы вместо него в любых контекстах.

То, что любая переменная может иметь разные статический и динамический типы в конкретной точке исполнения, это крайне важно, потому что при вызове виртуальной функции (как в примере выше) вызывается виртуальная функция, определённая в классе, соответствующем динамическому типу объекта, а при вызове невиртуальной функции – функция его статического типа.

Вызов функции статического типа также называется ранним или статическим связыванием, а вызов функции динамического типа называется поздним или динамическим связыванием (имеется в виду связывание функции с именем типа).

3.5.3 Проблема срезки

Мы всё ещё не стираем с доски и глядим на `friendship`. Аргументы `n1` и `n2` переданы в неё по ссылке. В принципе, они могли бы быть переданы и по указателю:

```
void friendship(const NamedObject *n1, const NamedObject *n2)
   /* still ok */
```

Но крайне дурным тоном является передача аргументов по значению.

```
void friendship(const NamedObject n1, const NamedObject n2) /*
   error! */
```

Это происходит от того, что формальные аргументы конструируются при входе в функцию. Для указателей и ссылок всё нормально. Но попытка сконструировать объект абстрактного класса (то есть класса, содержащего хотя бы один чисто виртуальный метод) сама по себе ошибочна.

Хорошо, пусть даже это удалось. В этом случае новосконструированные объекты `n1` и `n2` уже не будут содержать частей от Дарта Вейдера и Покемона, а станут безликими "Именованными объектами". Этот срез информации при передаче по значению называется "проблемой срезки". Срезка часто возникает в практических контекстах и важно о ней помнить и избегать передавать параметры по значению.

3.5.4 Параметры по умолчанию и виртуальные функции

Тонкий вопрос, который многие программисты на C++ часто упускают из виду это то, как ведут себя виртуальные функции с параметрами по умолчанию. Представим следующий код:

```
class Shape {
public:
  enum ShapeColor {Red, Green, Blue };
  virtual void draw( ShapeColor cl = Red) const = 0;
};

class Rectangle : public Shape {
public:
  virtual void draw( ShapeColor cl = Green) const;
};
```

```
11
12 int drawShape(const Shape &shape)
13 {
14   shape.draw();
15 }
16
17 int main(void)
18 {
19   Rectangle rc;
20   drawShape(rc);
21 }
```

Программист, читающий этот код, возможно, ожидает, что в этом случае будет нарисован зелёный прямоугольник. Но это не так. Прямоугольник будет нарисован красным цветом. Дело в том, что функции в C++ связываются динамически, а аргументы по умолчанию – статически. Таким образом для вызова draw компилятором будет сгенерирован вызов виртуальной функции с аргументом по умолчанию, объявленном в классе `Shape`. Именно поэтому виртуальные функции с аргументами по умолчанию считаются крайне плохим тоном

Домашняя наработка: исследуйте самостоятельно поведение виртуальных функций с переменным числом аргументов.

Домашняя наработка: также самостоятельно исследуйте случай вызова виртуальной функции из конструктора или деструктора и чем это чревато.

3.6 Перегрузка операторов

Очень важным частным случаем полиморфизма в C++ и частой причиной ненависти разработчиков к чтению кода на этом языке является возможность перегружать для пользовательских типов такие операторы как “+”, “-”, “*” и т. п.

Рассмотрим классический пример – реализацию некоего пользовательского класса, оперирующего комплексными числами.

```
1 class Complex
2 {
3 public:
4   Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
5   Complex operator+ (Complex x) { re += x.re; im += x.im; }
```

```
6   Complex operator* (Complex x) { re = re * x.re - im * x.im;
        im = re * x.im + im * x.re; }
7
8   double get_real(void) const { return re; }
9   double get_imag(void) const { return im; }
10
11 private:
12   double re, im;
13 };
```

Благодаря такому определению, арифметику с комплексными выражениями можно записывать в форме, близкой к общепринятой.

```
1  void test_simple_cases(void)
2  {
3    Complex a = Complex(1.0, 3.1);
4    Complex b = Complex(1.2, 2.0);
5    Complex c = b;
6    a = b + c;
7    b = b + c * a;
8    c = a * b + Complex(1, 2);
9
10   printf("%lf + %lf * i\n", c.get_real(), c.get_imag());
11 }
```

Главная проблема здесь в том, что вы действительно вольны написать любой код в определении оператора сложения. Ваше сложение комплексных чисел не обязано удовлетворять даже каким-то базовым инвариантам сложения:

```
1  assert (a + b == b + a); /* ORLY? */
```

Вместо этого ваша операция сложения может осуществлять вычитание, лезть в базу данных или форматировать диск. Почти всегда, когда вы читаете код на C++, вы обязаны иметь в виду, что не можете быть уверены что значит операция “+” этим утром. Ночью кто-нибудь мог вкоммитить в неё чудовищные изменения – из лучших побуждений, разумеется.

Всё усугубляется тем, что общий список операторов, которые можно переопределить, крайне внушающ. Его всегда можно посмотреть в стандарте, но кроме основных арифметических и логических операций, определение которых довольно таки прямолинейно, перегрузке могут подвер-

гаться совершенно экзотические вещи – сравнение, присваивание, приведение к типу (любому, включая встроенные), обращение по указателю и даже выделение памяти с помощью new и её освобождение. Давайте побеседуем об нескольких специальных и проблематичных случаях переопределённых операторов.

Домашняя наработка: кстати а как бы вы определили оператор == для комплексных чисел? Только помните, что определять сравнение чисел типа double через == это вообще не лучшая идея...

3.6.1 Переопределение симметричных бинарных операций

Для того, чтобы абстракция комплексных чисел была завершённой, должна быть возможность неявного преобразования double в complex, ведь по сути число `2.0` это `2.0 + 0.0 * i`. Эта возможность заложена в конструкторе класса (вспоминаем, что конструктор с одним аргументом это неявное преобразование типа).

```
1  Complex a = Complex(1.0, 3.1), b;
2  b = a + 2.0; /* ok */
```

Но здесь таится и опасность, потому что сложение перестаёт быть коммутативным и простая запись, вида:

```
1  Complex a = Complex(1.0, 3.1), b;
2  b = 2.0 + a; /* error! */
```

Выдаст ошибку компиляции. Ошибка связана с тем, что неявные преобразования применяются только к параметрам, перечисленным в списке параметров. Поэтому `a.operator+(2.0)` преобразует `double` к `Complex`, но для `(2.0).operator+(a)` неявный параметр `this` – указатель на объект для которого вызывается метод, не подвергается, согласно стандарту, никаким неявным преобразованиям.

Правильный метод: определить операторы сложения и умножения вне класса как отдельные функции:

```
1  class Complex
2  {
3  public:
4    Complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
5
6    double get_real(void) const { return re; }
7    double get_imag(void) const { return im; }
```

```
8
9  private:
10   double re, im;
11 };
12
13 Complex operator+ (const Complex &x, const Complex &y)
14 {
15   return Complex(x.get_real() + y.get_real(), x.get_imag() + y.
       get_imag());
16 }
17
18 Complex operator* (const Complex &x, const Complex &y)
19 {
20   return Complex(x.get_real() * y.get_real() - x.get_imag() * y
       .get_imag(),
21                  x.get_real() * y.get_imag() + x.get_imag() * y.
                      get_real());
22 }
```

Точно так же необходимо поступать со всеми функциями, которые по определению должны быть коммутативными. Тогда коммутативность будет сохранена.

3.6.2 Переопределение выделения и освобождения памяти

Переопределение `new` и `delete` в классах это всегда отдельная и крайне скользкая тема. Они уже были коротко рассмотрены выше, но теперь настало время уточнить подробности. Основным отличием `new` от `malloc` является то, что malloc выделяет просто кусок памяти, в то время как new конструирует объект – в частности вызывает его конструктор, но не только. Например конструируется и заполняется таблица виртуальных методов. И опять-таки не только это. Во многом из-за всех этих "не только", детали которых следует пока оставить за кадром и может быть позже вернуться, напрямую вызвать конструктор в C++ нельзя, так что обойтись без new для этого не удастся. Но, использовав `malloc()` для выделения буфера, вместе с так называемым размещающим new (placement new) можно обойтись без `delete` (поскольку деструктор может быть явно вызван).

```
1 /* new and delete */
2 #include <cstdlib>
```

```
3  #include <cstdio>
4  #include <cassert>
5  #include <new>
6
7  using namespace std;
8
9  class Widget
10 {
11   private:
12     int m_i;
13   public:
14     Widget(int a_i): m_i(a_i * a_i) {}
15     int get_state() const { return m_i; }
16     ~Widget() {}
17 };
18
19 Widget *
20 test_malloced(int initializer)
21 {
22   Widget *wbuf = (Widget *) malloc(sizeof(Widget));
23   Widget *w = new (wbuf) Widget(initializer);
24   return w;
25 }
26
27 int
28 main(void)
29 {
30   Widget *w = test_malloced(5);
31   assert(w->get_state() == 25);
32   w->~Widget();
33   free(w);
34   return 0;
35 }
```

В этом листинге наверное сейчас всё непонятно. Сделаем шаг назад и рассмотрим типичное выделение памяти:

```
1 class Widget {};
2 ...
3 Widget *w = new Widget;
```

Что здесь происходит? Неожиданно много всего.

1. Сначала, по форме оператора new, компилятор определяет какой именно new имеется в виду. Существуют три нормальные формы new:

   - new в куче с возбуждением исключений при исчерпании памяти. Именно он использован в рассматриваемом примере.
   - new в куче с возвратом нуля при исчерпании памяти
   - размещающий new (с размещением в заданном месте). Именно он и был использован в листинге выше. Его синтаксис похож на выделяющий new из этого примера, но буфер, в котором размещается сконструированный объект, не выделяется, а передаётся как параметр в скобках.

   Кроме нормальных существуют ещё и специальные формы new, когда пользователь некоего типа переопределяет размещающий new этого типа передавая ему не область памяти, а нечто иное – скажем область памяти и поток для записи лога и т.п. В следующем листинге показано использование разных форм new.

```
1 class Widget {};
2 ...
3 Widget *w = new Widget;
4 Widget *w = new (std::nothrow) Widget;
5 Widget *w = new (WidgetPool) Widget;
6 Widget *w = new (WidgetPool, WidgetLogStream) Widget;
```

   Здесь первая строчка это `malloc`+вызов контруктора с возбуждением исключения при ошибке выделения, вторая строчка это `malloc`+вызов контруктора без возбуждения исключения при ошибки выделения (тогда в `w` просто запишется `NULL`), третья строчка это просто вызов конструктора на уже выделенную память (при этом неудачи выделения быть, разумеется, уже не может). И четвёртая строчка иллюстрирует специальную пользовательскую форму оператора `new` и не будет у вас скомпилирована, без специальных объявлений внутри класса `Widget`.

2. Далее компилятор выделяет память в куче (строго при необходимости, размещающие и специальные формы пропускают этот пункт) и в случае нехватки памяти вызывает пользовательский обработчик, который можно поставить и на стандартное выделение с помощью `set_new_handler`.

3. Далее в случае если вызвана нормальная форма неразмещающего new, компилятор вызывает конструктор размещённого в куче объекта или все конструкторы, если был размещён массив.

4. Разрушение происходит в обратном порядке.

Давайте напишем класс для которого объявлены переопределения всех нормальных форм new/delete.

```
1 class StandardNewDeleteForms {
2 public:
3 static void *operator new(std::size_t size);
4 static void operator delete(void * memory);
5 static void *operator new(std::size_t size, const std::
    nothrow_t &nt);
6 static void operator delete(void * memory, const std::nothrow_t
     &nt);
7 static void *operator new(std::size_t size, void *ptr);
8 static void operator delete(void * memory, void *ptr);
9 };
```

Домашняя наработка: Напишите и протестируйте код для каждого из переопределений.

C++ предоставляет достаточно тонкий инструментарий для работы с памятью, который необходимо использовать с осторожностью. Переопределяя каждый оператор не забывайте о возможном пользовательском вызове `set_new_handler`. Переопределяя `new` не забывайте переопределять соответсвующий `delete`. Работайте с памятью осторожно и внимательно. И главное – трижды подумайте нужно ли вам это вообще.

3.6.3 Переопределение копирования и присваивания

Представим пустой, сферический класс в вакууме.

```
1 class Empty {};
```

Так ли он пуст, как это кажется на первый взгляд? Совершенно очевидно, что даже объект такого совершенно пустого класса в программе на C++ может быть создан, создан по образцу, скопирован и разрушен. Всю эту обвязку, если её не предоставляете вы, предоставляет компилятор C++. То есть написанное определение эквивалентно следующему:

```
/* There is nothing empty in C++ */
class Empty {
public:
  Empty() { /* default implementation */ }
  Empty(const Empty &rhs) { /* default implementation */ }
  ~Empty() { /* default implementation */ }
  Empty& operator=(const Empty &rhs) { /* default
     implementation */ }
};
```

Конструктор и деструктор должны быть к этому моменту уже понятны. Но мы видим, что C++ генерирует ещё две функции специального вида – копирующий конструктор, отвечающий за создание объекта по образцу такого же и оператор присваивания, который будет вызван при присваивании объекта в выражениях вида `a=b`.

```
Empty e1;
Empty e2(e1); /* copy constructor called */
e1 = e2; /* assignment called */
```

По умолчанию сгенерированные компилятором конструктор копирования и оператор присваивания просто побитово копируют код аргумента в целевой объект. Разумеется, это не пройдёт если в классе есть член-ссылка, тогда вы должны писать оператор присваивания самостоятельно.

Домашняя наработка: объяснить почему присваивание по умолчанию не годится при членах класса, являющихся ссылками.

Иногда такое умолчательное поведение приводит к мрачным проблемам, тяжёлым в отладке. Предположим, вы написали некий класс, управляющий внутренним буфером, который выделяется в конструкторе и освобождается в деструкторе.

```
class CBuffer {
public:
  CBuffer(int size) { m_size = size; m_buffer = new char[size];
      }
  ~CBuffer() { delete[] m_buffer; }
  char &get(int x) { assert(x >= 0 && x < m_size); return
     m_buffer[x]; }
private:
  char *m_buffer;
  int m_size;
```

```
};
```

А потом кто-то по незнанию скопировал его внутри какой-то функции.

```
void wrong(void)
{
  CBuffer b1;
  CBuffer b2 = b1;
} /* Segfault here */
```

Что при этом произойдёт? Выделенный вами буфер доступен по указателю. Указатель будет побитово скопирован. Это значит что теперь на буфер есть два указателя. При выходе за границы блока оба будут освобождены. Это крайне неприятная ошибка двойного освобождения и проблемы.

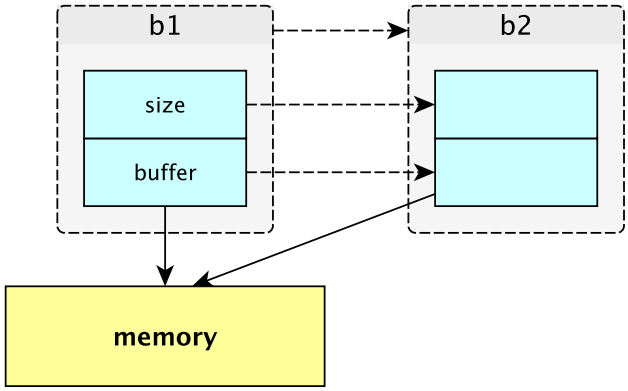

Рис. 5: Ошибка двойного освобождения

Говорят, что по умолчанию C++ реализует поверхностное копирование (shallow copy). Копирование с выделением нового буфера и копированием в него содержимого старого (ещё и с проверкой на присваивание самому себе) это глубокое копирование (deep copy) которое пользователь всегда должен реализовать самостоятельно.

```
class CCopyableBuffer {
public:
  CBuffer(int size) { m_size = size; m_buffer = new char[size];
      }
  ~CBuffer() { delete[] m_buffer; }
  CBuffer(const CBuffer& rhs) {
    m_size = rhs.m_size; m_buffer = new char[m_size];
```

```
7     memcpy(m_buffer, rhs.m_buffer, m_size);
8   }
9   CBuffer& operator= (const CBuffer& rhs)
10  {
11    if (rhs == this) return *this;
12    delete [] m_buffer;
13    m_size = rhs.m_size; m_buffer = new char[m_size];
14    memcpy(m_buffer, rhs.m_buffer, m_size);
15  }
16  char &get(int x) { assert(x >= 0 && x < m_size); return
       m_buffer[x]; }
17 private:
18  char *m_buffer;
19  int m_size;
20 };
```

Так всё будет работать. Но иногда копирование не нужно и его проще запретить, объявив собственные конструктор копирования и оператор присваивания в закрытой части класса.

```
1  class CBuffer {
2  public:
3    CBuffer(int size) { m_size = size; m_buffer = new char[size];
         }
4    ~CBuffer() { delete[] m_buffer; }
5    char &get(int x) { assert(x >= 0 && x < m_size); return
       m_buffer[x]; }
6  private:
7    char *m_buffer;
8    int m_size;
9    CBuffer(const CBuffer& rhs);
10   CBuffer& operator= (const CBuffer& rhs);
11 };
```

Теперь компилятор не сгенерирует за вас неправильные варианты и копирование просто не будет скомпилировано.

3.7 Динамическое приведение и RTTI

Кроме всех операторов приведения, рассмотренных в предыдущей лекции, существует ещё один, разработаный специально, чтобы приво-

дить статический тип к динамическому типу. Он называется dynamic_cast . Его интересной особенностью является то, что он по-разному работает для указателей и для ссылок. Сначала разберём dynamic_cast для указателей

```
Derived* temp = dynamic_cast<Derived*>(base);
```

Пытается привести p, типа Base* к типу Derived*, где Base и Derived принадлежат одной и той же иерархии. Если Derived является базовым классом для Base, то это ничем не отличается от static_cast. Но dynamic_cast работает также если Base является полиморфным базовым классом для Derived, то есть является базовым классом для Derived и содержит виртуальные функции. Звучит запутано? Давайте посмотрим пример.

```
#include <cstdio>

using namespace std;

class NamedObject {
public:
  virtual const char *whoareyou(void) const = 0;
};

class Pokemon : public NamedObject {
public:
  const char *whoareyou(void) const { return "Pokemon"; }
};

class DartVeider : public NamedObject {
public:
  const char *whoareyou(void) const { return "Dart Veider"; }
};

int check_DartVeider(NamedObject *p)
{
  if (dynamic_cast<DartVeider *>(p))
    {
      printf("You are Dart Veider!\n");
    }
  else
    {
```

```
28        printf("You are not Dart Veider, you are %s\n", p->
              whoareyou());
29      }
30    return 0;
31  }
32
33  int main(void)
34  {
35    DartVeider dv;
36    Pokemon pm;
37    check_DartVeider(&dv);
38    check_DartVeider(&pm);
39    return 0;
40  }
```

Что будет на выдаче? Ответ:

```
1  You are Dart Veider!
2  You are not Dart Veider, you are Pokemon
```

Коротко говоря, `dynamic_cast`, использованый для указателя даёт возможность "спросить" такой ли у этой переменной динамический тип, как он полагает. Он приводит к этому типу если ответ "да" или возвращает `NULL`, если ответ "нет".

При использовании `dynamic_cast` для ссылок, вопрос превращается в утверждение. Если это утверждение нарушается, кидается исключение, но о них мы поговорим в другой раз.

Для `dynamic_cast` очень важно, чтобы в базовом классе была хотя бы одна виртуальная функция. Очень часто отличным кандидатом на роль такой функции является виртуальный деструктор, о котором мы сейчас поговорим.

3.8 Виртуальные деструкторы

Для программиста на C++ важно знать один очень специальный случай полиморфизма, относящийся к семантике освобождения, а именно виртуальные деструкторы. Рассмотрим пример – пусть у нас есть некий класс измерителя времени и фабричная функция, которая в зависимости от точности, с которой пользователю необходимо измерять время, возвращает ему какие-нибудь часы от солнечных до атомных.

```
class TimeKeeper {
public:
  TimeKeeper() {};
  virtual ~TimeKeeper() {};
};

class SolarWatch : public TimeKeeper {/* ... some details ...
    */};
class SwissWatch : public TimeKeeper {/* ... some details ...
    */};
class AtomicClock : public TimeKeeper {/* ... some details ...
    */};

TimeKeeper * getTimeKeeper (double reqprec)
{
  if (reqprec < 1.0)
    return new SolarWatch;
  if (reqprec < 10.0)
    return new SwissWatch;
  return new AtomicClock;
}

int use_time(TimeKeeper *clock)
{
  /* ... some details ... */
  return 0;
}

int main(void)
{
  TimeKeeper *ptk = getTimeKeeper(0.1);
  use_time(ptk);
  delete ptk;
}
```

Что произойдёт при уничтожении, если деструктор будет не виртуальным? Будет освобождена базовая часть, но не будет освобождена производная часть, что неминуемо приведёт к утечкам памяти и проблемам. Если же сделать деструктор виртуальным, то будет автоматически правильно вызван деструктор производного класса по указателю на базовый.

Наличие в классе невиртуального деструктора является в мире промышленного программирования достаточным основанием никогда ничего не наследовать от этого класса. Обратное – плохой тон.

3.9 Проблемы, возникающие при проектировании открытого наследования

Принцип подстановки Лисков требует навыка при работе с ним. Предположим, вы проектируете иерархию геометрических объектов и столкнулись с необходимостью расположить в ней такие абстракции как "квадрат" и "прямоугольник". Первой мыслью может быть унаследовать прямоугольник от квадрата – в конце концов прямоугольник вводит ещё одно поле и может быть какие-нибудь дополнительные методы

```
1  class Square {
2  public:
3    Square(double x) : m_x(x) {}
4    virtual ~Square() {}
5    double a(void) { return m_x; }
6    virtual double get_sq(void) { return a() * a(); }
7  protected:
8    double m_x;
9  };
10
11 class Rect: public Square {
12 public:
13   Rect(double x, double y) : Square(x), m_y(y) {}
14   int b(void) { return m_y; }
15   virtual double get_sq(void) { return a() * b(); }
16 private:
17   double m_y;
18 }
```

Но такой подход создаёт проблемы, связанные с тем, что прямоугольник не является частным случаем квадрата и здесь нарушается принцип подстановки.

Вопрос к студентам: опишите проблемы которые может вызвать такое проектирование. Например рассмотрите реализованную в Square (и даже виртуальную) функцию `void increase(int times)`, в times раз увеличивающую площадь квадрата. Как вы реализуете её для прямо-

угольника?

Домашняя наработка: рассмотрите вариант наследования квадрата от прямоугольника. К чему это приводит?

Подсказка: совсем ничего хорошего, как вы догадываетесь.

3.10 Иерархии

Продолжим разговор о наследовании. До сих пор рассматривалось только одиночное одноуровневое наследование, но C++ даёт в этом отношении гораздо больше свободы. Один класс может наследовать многим классам, которые сами кому-то наследуют и так далее. Для этого базовые классы с их модификаторами доступа перечисляются через запятую

```
1 class Man: public AnimalsWithTwoLegs, public OnesWithoutWings
    {};
```

Это позволяет строить иерархии взаимодействующих классов и объектов при проектировании сложных программных систем.

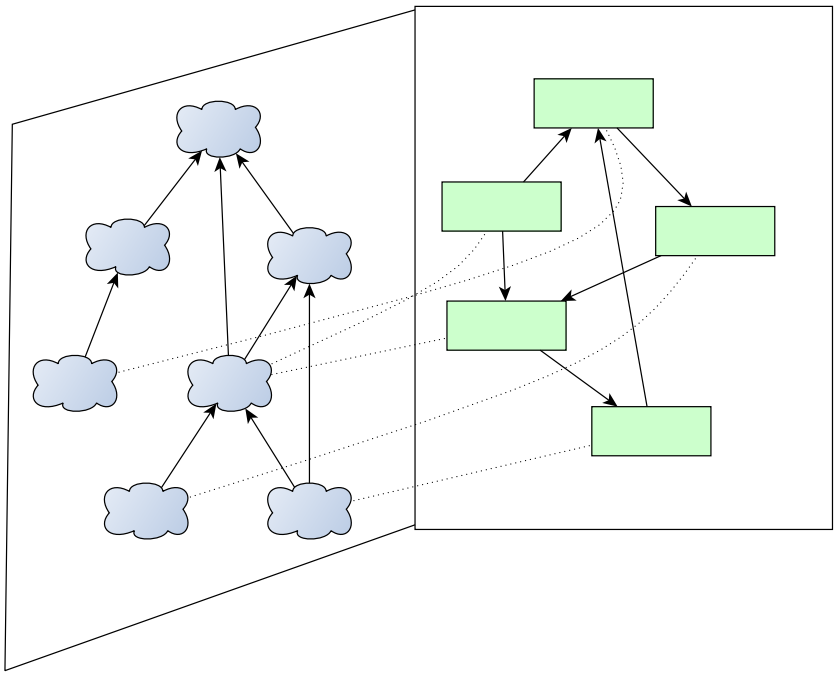

Рис. 6: Иерархии классов и объектов

Проектирование хорошей иерархии это всегда сложный инженерный процесс, выходящий за рамки этого курса. Зато можно на игрушечных примерах рассмотреть основные проблемы, ожидающие разработчика на этом пути.

3.10.1 Ромбовидные схемы и виртуальные базовые классы

Предположим, вы проектируете систему, поддерживающую абстракции файлов ввода и вывода. Рано или поздно вы пришли к чему-то вроде такого

```
1 class File {};
2 class InputFile : public File {};
3 class OutputFile : public File {};
4 class IOFile : public InputFile, public OutputFile {};
```

Графически это может быть выражено ромбовидной схемой

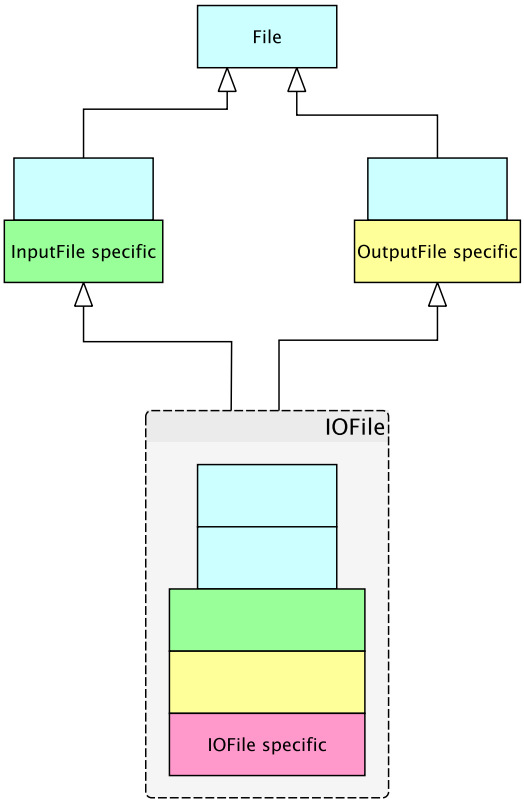

Рис. 7: Ромбовидная схема

Предположим, что в классе `File` есть поле `File::filename`. По умолчанию, в классе `IOFile` у вас получится два члена: `InputFile::File::filename` и `OutputFile::File::filename`, но файл у которого два имени это абсурд. Чтобы избежать такой ситуации, базовый класс в наследовании может быть объявлен виртуальным (ещё можно вообще никогда ничего не писать на C++, а использовать C, это предпочтительный вариант).

```
1 class File {};
2 class InputFile : virtual public File {};
3 class OutputFile : virtual public File {};
4 class IOFile : public InputFile, public OutputFile {};
```

Теперь всё хорошо и в ромбовидной схеме у самого нижнего производного класса есть только одна копия базового.

3.10.2 Порядок инициализации в сложных диаграммах

Важно очень хорошо представлять себе порядок конструирования сложных диаграмм классов. Представим несколько синтетический (но на самом деле гораздо менее сложный, чем многие практически важные иерархии) пример.

```
1  class B1 { };
2  class V1 : public B1 { };
3  class D1 : virtual public V1 { };
4  class B2 { };
5  class B3 { };
6  class V2 : public B1, public B2 { };
7  class D2 : virtual public V2, public B3 { };
8  class M1 { };
9  class M2 { };
10 class X : public D1, public D2 {
11   M1 m1_;
12   M2 m2_;
13 };
```

Для простоты можно изобразить эту иерархию на листочке:

Порядок инициализации объекта класса `X` для изображённой иерархии следующий:

- Сначала конструируются виртуальные базовые классы

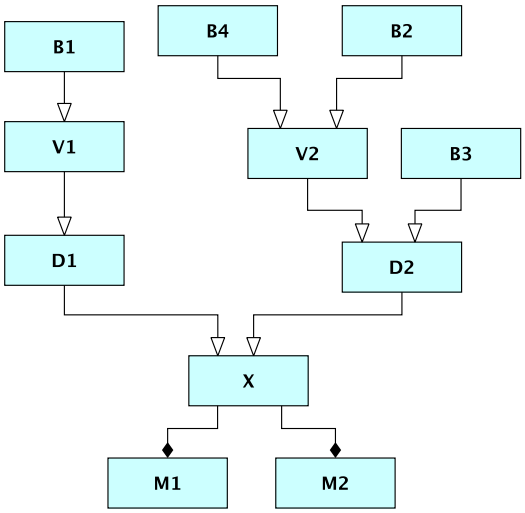

Рис. 8: Сложная иерархия

1. Конструирование `V1`: `B1::B1()`, `V1::V1()`
2. Конструирование `V2`: `B1::B1()`, `B2::B2()`, `V2::V2()`

- Затем конструируются невиртуальные базовые классы
  1. Конструирование `D1`: `D1::D1()`
  2. Конструирование `D2`: `B3::B3()`, `D2::D2()`
- Затем конструируются члены `M1::M1()`, `M2::M2()`
- И в последнюю очередь выполняется конструктор `X::X()`

Вид наследования (открытое, закрытое или защищённое) не влияет на порядок инициализации.

3.10.3 Вложенные классы и снова о пространствах имён

Вложенные функции в C++ невозможны так же как и в C (что кстати не вполне логично, так как вложенные лямбда-функции в новом стандарте возможны и будут рассмотрены на следующих лекциях, так что казалось бы гулять так гулять). Зато C++ позволяет вкладывать классы. Синтаксис очевиден, но, если необходимо, чтобы вложенный класс "знал" о своём окружении, об этом надо позаботиться отдельно:

```
#include <cstdio>

class DeathStar
{
public:
  DeathStar(const char *name) : m_name(name)
    {
       m_dv.starptr = this;
    }

  class DartVeider
  {
    public:
      void whoareyou() const
        {
          printf("I am Dart Veider, flying on %s\n", starptr ?
              starptr->m_name : "hmm... I don't know where");
        }
      DeathStar *starptr;
  };

  void ask_dart_veider() const { m_dv.whoareyou(); }

private:
  DartVeider m_dv;
  const char *m_name;
};

int main(void)
{
  DeathStar d("Old Good Death Star");
  d.ask_dart_veider();
  return 0;
}
```

Каждый вложенный класс определяет область видимости своих имён. Таким образом, можно создать отдельный объект типа `DeathStar::DartVeider`, но можно и запретить это, поместив его в private. Собственно обычные пространства имён имеют всего одно отличие от классов со всеми открытыми статическими членами. В случае обычных пространств имён вы можете писать много их объявлений и все они будут объединены

компилятором. Таким образом, даже зная как можно сэмулировать их классами, есть смысл всё же использовать пространства имён.

3.11 Скажи мне кто твой друг

Модель инкапсуляции C++ логична и стройна (общий смех). В общем случае закрытые члены класса формируют его состояние и недоступны извне, иначе, чем через его методы, отражающие поведение – это хорошо и правильно. Но в некоторых случаях, разработчик хотел бы дать некоему классу эксклюзивный доступ к закрытым членам другого класса. Например для того, чтобы не писать кучу геттеров и в то же время сохранить консистентность абстракции. Для этого язык C++ предусматривает механизм друзей (friends) класса.

```
1  class Node
2  {
3  private:
4    int data;
5    int key;
6    /* ... */
7
8    friend class BinaryTree; /* class BinaryTree can now access
       data directly */
9  };
```

Класс Node, назначив себе друга BinaryTree открыл ему всё свое закрытое состояние, но при этом сам не получил никакого доступа к BinaryTree. Теперь внутри BinaryTree можно написать код вида:

```
1  class BinaryTree
2  {
3    private:
4    Node *root;
5
6    int find(int key);
7  };
8
9  int BinaryTree::find(int key)
10 {
11   /* check root */
12   if(root->key == key)
```

```
13      {
14        /* no need to go through an accessor function */
15        return root->data;
16      }
17    /* rest of find */
18  }
```

Класс может открывать своё состояние не только классам, но и функциям. Скажем, не открывая доступ всему BinaryTree, можно открыть его только методу find:

```
1  class Node
2  {
3      private:
4      int data;
5      int key;
6      /* ... */
7
8      friend int BinaryTree::find(); /* Only BinaryTree's find
           function has access */
9  };
```

Дружба не наследуется, но друг статического родительского типа достаточен, чтобы делать дружественные вызовы полиморфных наследников.

```
1  class Parent
2  {
3      friend class Family;
4      protected:
5          virtual void Answer() = 0;
6  };
7
8  class Child : public Parent
9  {
10   private:
11     void Answer() { printf("Child!\n"); }
12 };
13
14 class Family
15 {
16 friend class Parent;
17 public:
```

```
18    ParentAnswer(Parent *p) { p->Answer(); } /* ok */
19    ChildAnswer(Child *c) { c->Answer(); } /* error! */
20  };
21
22  int test(void)
23  {
24    Child c;
25    Family(&c); /* ok */
26  }
```

В общем случае дружба статических типов вредна для инкапсуляции. Но дружба, позволяющая полиморфные вызовы по большой и закрытой иерархии это полезно и иногда необходимо. Также дружба может быть полезна во многих неожиданных контекстах при работе с шаблонами, которые будут рассмотрены ниже.

3.12 Домашняя наработка по части 3

1. Используя возможности вашей любимой операционки (Windows, Linux, ...) добавьте поддержку объектов синхронизации к игре в мяч, используя RAII. После этого запустите два потока которые будут играть в физически реалистичный волейбол и выводить лог своей игры на экран.

2. Добавьте рандомизацию силы ударов, сетку, уровень земли, границы площадки.

3. Расширьте игру в мяч до трёхмерной.

4. Предложите вашим двум потокам поиграть сразу десятком мячей. Сотней.

4 Особая шаблонная магия

Шаблоны, введённые в C++ были в своё время введены туда экспериментально. Их не было ни в одном другом языке и никто по настоящему не знал, что из этого получится. Поэтому большинство интересных и важных свойств шаблонов не были разработаны. Они были открыты. На этой лекции нас тоже ожидают открытия.

4.1 Шаблоны функций

Вернемся на секунду к тому примеру, который я уже приводил когда говорил о преимуществах C++ в объявлении обобщённых функций без использования макросов и постараемся достичь в нём понимания сути происходящего.

```
1 template <typename T> inline const T
2 max (const T a, const T b)
3 {
4   return (a > b) ? a : b;
5 }
```

Говорят, что здесь объявлен шаблон функции. Синтаксис шаблона ясен из определения. После слова template в треугольных скобках через запятую перечисляются параметры шаблона.

```
1 template <typename T, class U, int x>
```

Здесь слово `typename` означает параметр шаблона, задающий имя типа. По историческим причинам можно использовать `class`, но новый стандарт поощряет использование `typename`. Кроме того вы уже должны видеть в функции max существенный недостаток – параметры, передающиеся по значению, в случае увесистого `T` будут неэффективны, лучше заменить их константными ссылками.

Расширим и исправим наш пример и посмотрим как шаблоны функций ведут себя при перегрузке.

```
1 /* 1. max of two ints */
2 inline int const& max (int const& a, int const& b)
3 {
4   return a < b ? b : a;
5 }
6
```

```
7  /* 2. maximum of two values of any type */
8  template <typename T>
9  inline T const& max (T const& a, T const& b)
10 {
11   return a < b ? b : a;
12 }
13
14 /* 3. maximum of three values of any type */
15 template <typename T>
16 inline T const& max (T const& a, T const& b, T const& c)
17 {
18   return max (max(a,b), c);
19 }
20
21 int main(void)
22 {
23   ::max(7, 42, 68);   /* calls [3]<int>  */
24   ::max(7.0, 42.0);   /* calls [2]<double> */
25   ::max('a', 'b');    /* calls [2]<char> */
26   ::max(7, 42);       /* calls [1]       */
27   ::max<>(7, 42);     /* calls [2]<int>  */
28   ::max<double>(7, 42); /* calls [2]<double> */
29   ::max('a', 42.7);   /* calls [1]       */
30   return 0;
31 }
```

Если это кажется вам сложным, то вы ошибаетесь. Это всё довольно просто по сравнению с тем, что нас ждёт дальше.

4.2 Простые шаблоны классов

Представьте, что перед вами стоит задача спроектировать стек – класс объектов любого типа (но однородных) которые будут туда помещаться и вытаскиваться. Как обычно сделаем синтаксис понятным через пример (детали вы всегда сможете прочитать в стандарте).

```
1  template <typename T>
2  class Stack {
3  public:
4    Stack() : m_top(NULL) {}
5    void push(const T& elem);
```

```
6   void pop();
7   T top() const { return m_top->elem; }
8 private:
9   struct StackElem {
10    T elem;
11    StackElem *next;
12  } *m_top;
13 };
```

Обратите внимание на то, что T, помещённый в шаблон, используется внутри класса как совершенно обычный тип. Мы можем вернуть его из метода, передать в метод и даже использовать как деталь реализации скрытой подструктуры.

```
1  template <typename T>
2  void Stack<T>::push(const T& elem)
3  {
4    StackElem *newelem = new StackElem;
5    newelem->elem = elem;
6    newelem->next = m_top;
7    m_top = newelem;
8  }
9
10 template <typename T>
11 void Stack<T>::pop(void)
12 {
13   if (NULL == m_top)
14     return;
15   StackElem *topelem = m_top;
16   m_top = m_top->next;
17   delete topelem;
18 }
```

Вопрос к студентам: что мы забыли?

Ожидается хоровой ответ: деструктор, копирующий конструктор и оператор присваивания. Позвать кого-нибудь к доске разобрать как их писать.

Давайте посмотрим как применять разработанный шаблонный класс:

```
1  int main (void)
2  {
3    Stack <int> intstack;
```

```
4   Stack <double> dblstack;
5   intstack.push(2);
6   dblstack.push(2.0);
7   printf("top1 = %d, top2 = %lf\n", intstack.top(), dblstack.
        top());
8   intstack.push(3);
9   dblstack.push(3.0);
10  printf("top1 = %d, top2 = %lf\n", intstack.top(), dblstack.
        top());
11  intstack.pop();
12  dblstack.pop();
13  printf("top1 = %d, top2 = %lf\n", intstack.top(), dblstack.
        top());
14  return 0;
15 }
```

Из примера ясно, что с использованием шаблонов, поддержка двух разнородных контейнеров (трёх, десяти) это крайне легко. Скорость компиляции, правда, страдает, вместе с объёмом кода, но по современным реалиям это невысокая цена.

4.2.1 Специализация

В случае с функциями был использован механизм перегрузки функций чтобы получить специальное поведение для конкретного типа аргументов. Но нельзя “перегрузить” класс. Зато вы можете специализировать шаблон класса для конкретных шаблонных параметров. Синтаксис, опять же, прост:

```
1 template <>
2 class Stack<int> { ... };
```

Такая запись создаёт отдельное поведение для стеков, инстанцированных целыми числами. Реализация и поведение такого шаблона могут не иметь вообще ничего общего с исходным. Это создаёт почти такую же радость при чтении кода на C++ как перегрузка операторов и неявные приведения.

4.2.2 Частичная специализация

Суть частичной специализации в том, что вы пишете отдельный код для определённых обстоятельств инстанциирования, но часть параметров этого кода всё ещё задаёт шаблон. При исходном шаблоне

```
template <typename T, typename U> class MyClass { ... };
```

Мы можем написать например такие специализации

```
/* partial specialization: both template parameters have same
   type */
template <typename T> class MyClass<T,T> { ... };
/* partial specialization: second type is int template <
   typename T> */
class MyClass<T,int> { ... };
/* partial specialization: both template parameters are pointer
    types */
template <typename T1, typename T2> class MyClass<T1*,T2*> {
   ... };
```

И далее посмотрим как конкретные объявления будут раскрыты.

```
MyClass<int,float> mif; /* uses MyClass<T1,T2> */
MyClass<float,float> mff; /* uses MyClass<T,T> */
MyClass<float,int> mfi; /* uses MyClass<T,int> */
MyClass<int*,float*> mp; /* uses MyClass<T1*,T2*> */
```

Учтите, что если под некое объявление подходят сразу два варианта частичной специализации, то это ошибка

```
MyClass<int,int> m;     // ERROR: matches MyClass<T,T>
                        //       and MyClass<T,int>
MyClass<int*,int*> m;   // ERROR: matches MyClass<T,T>
                        //       and MyClass<T1*,T2*>
```

Можно исправить ошибку сделав уникально подходящий шаблон

```
template <typename T>
class MyClass<T*,T*> { ... };
```

Правила частичной специализации также необходимо усвоить путём внимательного чтения стандарта и применять уместно.

4.2.3 Разное о параметрах шаблонов

Параметры шаблонов могут идти по умолчанию, в этом они похожи на параметры по умолчанию функций. Точно так же как параметры функций по умолчанию, параметры шаблонов по умолчанию связываются статически.

```
template <typename T, typename U = int> class MyClass { ... };
MyClass<int> foo; /* MyClass <int, int> */
```

Применять их следует с осторожностью.

Очень часто возникает идея параметризовать некий шаблон неким шаблоном. Рассмотрим простой пример инстанцирования стека стеков, чтобы понять синтаксис:

```
Stack < Stack<int> > stack;
```

Вложение таким образом можно осуществлять теоретически бесконечно. Реально больше трёх уровней уже крайне плохой тон (и чревато комбинаторными взрывами).

4.2.4 Шаблоны членов и инкапсуляция

До сих пор шаблоны классов и шаблоны функций рассматривались раздельно, но в принципе шаблонная функция тоже может быть членом класса. Это не вносит почти никаких изменений в рассмотренный материал – для неё верно всё то же, что и для других шаблонных функций. Но использование шаблонных членов вносит серьёзные коррективы в инкапсуляцию. Коротко говоря, использование шаблонных членов отменяет инкапсуляцию. Рассмотрим класс:

```
class X {
public:
  X() : private_(1) {}
  tempalte <typename T> void f(const T& t) { /* ... */ }
  int Value(void) const { return private_; }
private:
  int private_;
};
```

За счёт наличия в этом классе шаблонного члена, человек, использующий его, может написать код:

```
namespace {
  struct Y {};
}
template<>
void X::f(const Y&) {
  private_ = 2;
}
```

Этот код эксплуатирует оговорённую в стандарте возможность специализировать шаблон-член для любого типа. Это выглядит забавно – специализацию шаблонов-функций заменяет перегрузка, но шаблон-член вы можете как перегрузить, так и специализировать. Эта ортогональность прекрасна, если ей не злоупотреблять.

То, что шаблоны-члены неявно нарушают инкапсуляцию – последствие взаимодействия объектно-ориентированной и шаблонной подсистем языка, которые проектировались во многом ортогонально и имеют нюансы совместного использования. Ещё о некоторых нюансах пойдёт речь далее, но прежде мы остановимся на альтернативах объектно-ориентированной модели, которые предлагаю шаблоны. И главная из них это шаблонный полиморфизм как альтернатива ОО-полиморфизму.

4.3 Два полиморфизма

Что же, к этому моменту вы знаете о шаблонах уже достаточно, чтобы вернуться к серьёзным вещам, таким как Дарт Вейдер и покемоны. Идея наследовать их от общего предка действительно хороша, если вы программист на Java, где вообще всё наследуется от общего предка. Но на C++ вы можете написать два отдельных класса:

```
class Pokemon {
public:
  void whoareyou(void) const { printf("Pokemon"); }
};

class DartVeider {
public:
  void whoareyou(void) const { printf("Dart Veider"); }
};
```

И тем не менее сохранить возможность писать довольно абстрактный код, полагающийся только на общий интерфейс объектов (способность

сказать имя) и отдающий им на откуп детали реализации:

```
template <typename T, typename U>
void friendship(const T& n1, const U& n2)
{
  printf("Hello, I am "); n1.whoareyou(); printf("\n");
  printf("Hello, I am "); n2.whoareyou(); printf("\n");
  printf("Lets be friends!\n");
}

int main(void)
{
  Pokemon p;
  DartVeider d;
  friendship(p, d);
  return 0;
}
```

Это прекрасное свойство называется полиморфизм времени компиляции, или статический полиморфизм. Заметьте, никаких больше виртуальных функций, никаких таблиц виртуальных методов, никакого оверхеда времени выполнения (за счёт раздутия кода помещением туда 100500 инстанций функции friendship, разумеется).

Разница в том, что класс NamedObject создавал нам явный интерфейс, в случае же статического полиморфизма, шаблонная функция накладывает неявные требования к обоим своим инстанциирующим типам, сформулированное точно так же "поддерживать функцию-член `whoareyou(void)`". В стандарт вводили-вводили "концепты", то есть способ задать неявный интерфейс явно, да так и не ввели.

Перед каждым взрослым программистом на C++ регулярно встаёт выбор что использовать – динамический полиморфизм и виртуальные функции или же статический полиморфизм и шаблонные функции. Обычно используется и то и другое, ведь мы же должны делать нашу жизнь интересной. Чуткое сердце разработчика и кровавый топор в руках тимлида будут вам здесь лучшими советчиками, чем этот курс лекций.

4.4 Ваш друг typename

Это ваш второй воображаемый друг, первый был, как вы помните, typedef. Изначально typename был введен в C++ чтобы явно указывать компилятору на вложенные шаблонные типы. Рассмотрим код:

```
1 template <typename T>
2 int foo (const T& x)
3 {
4   T::subtype *y;
5   /* ... */
6 }
```

Что здесь написано? Думаете объявление указателя на вложенный тип? Вы не поверите, но дословно здесь написано следующее: “В классе `T` должен быть статический член с именем `T::subtype`. Необходимо **умножить** его на некую (очевидно глобальную) переменную `y` и проигнорировать результат”. Печально, да? Давайте исправим код:

```
1 template <typename T>
2 int foo (const T& x)
3 {
4   typename T::subtype *y;
5   /* ... */
6 }
```

Теперь всё хорошо, мы явно указали компилятору, что `T::subtype` это тип. Именно так `typename` и был запланирован. По английски устранение такой неоднозначности называется “**disambiguation**” и именно так этот термин устоялся в англоязычной литературе (не встречал его в русскоязычной). То, что я перевожу disambiguation как “устранение неоднозначности” это некая вольность, скорее передающая смысл термина, чем его перевод.

В использовании `typename` для вложенных шаблонных классов может встретится один небольшой трюк – дополнительное использование слова `template` для устранения неоднозначности при разборе шаблона. Представим у нас есть код:

```
1 template <typename Type>
2 class Outer
3 {
4 public:
5   template <typename InType>
```

```
  class Inner
  {
  public:
    template <typename X> void nested(void)
      {
        printf("victory\n");
      }
  };
};
```

И надо написать вызов для шаблонной функции вложенного класса (например из другого шаблонного класса, с достаточно простым определением)

```
template <typename T1, typename T2>
class Usage
{
public:
  void caller( /* parameter type */ &obj )
    {
      /* obj.nested<int>(); */
    }
};
```

Можно написать простую проверочную программу, где использовать этот вызов как-то так:

```
int main(void)
{
  Usage<int, int> u;
  Outer<int>::Inner<int> obj;
  u.caller(obj);
  return 0;
}
```

Сам по себе этот пример представляет собой лишь изящную головоломку, но полезен для иллюстрации концепции. В реальных проектах эти проблемы могут всплыть в гораздо более сложном коде и нужно вдоволь натренироваться на простых примерах, чтобы уметь их угадывать. Правильный ответ выглядит несколько необычно:

```
template <typename T1, typename T2>
class Usage
```

```
{
public:
  void caller(typename Outer<T1>::template Inner<T2> &obj)
    {
      obj.template nested<int>();
    }
};
```

Но что произойдёт если мы не упомянем ключевого слова `template`? Примерно та же проблема, что и с `typename` – компилятор не будет в точности знать как ему трактовать треугольную скобку, открывающуюся после имени типа `Inner` или после имени функции `nested`. И в том и в другом случае, без явного указания ключевого слова template, он будет трактовать эту скобку как переопределённый оператор “меньше” (снова нечто невероятное). Это может привести вас к массе неявных но интересных ошибок.

4.5 Наследование от шаблона и CRTP

Шаблонный класс со всеми определёнными парамтерами это уже настоящий класс и значит от него можно наследоваться. На этой технике основана интересная и часто используемая идиома CRTP.

CRTP – curiously reccuring template parameter означает параметризацию шаблона, являющегося базовым классом, шаблонным параметром, являющимся самим классом-потомком. Предположим, вы проектируете класс, реализующий абстракцию лазерного меча. При попытке ударить лазерным мечом, он должен попросить своего владельца использовать Силу. Тогда ваш код может выглядеть как-то так:

```
#include <cstdio>

using namespace std;

template <typename Owner>
class LaserSword {
public:
  void hit(void)
    {
       static_cast<Owner*>(this)->use_power();
    }
```

```
12 };
13
14 class DartVeider: public LaserSword<DartVeider> {
15 public:
16   void use_power(void)
17   {
18     printf("Dart Veider uses power\n");
19   }
20 };
21
22 int main(void)
23 {
24   DartVeider d;
25   d.hit();
26   return 0;
27 }
```

По сути, здесь CRTP это способ для класса DartVeider попросить себе лазерный меч, который понимает, что он лазерный меч Дарта Вейдера. Подобным образом стек может попросить себе аллокатор, который понимает что он аллокатор для стека и т.п.

Вы должны хорошо освоиться с этой идиомой, она нам пригодится при разговоре про STL.

4.6 Правила инстанцирования и SFINAE

Выше уже несколько раз было употреблено слово “инстанцирование”. Интуитивно оно понятно, но оно имеет разный смысл для обычных классов (там инстанцирование класса это создание его объекта) и для шаблонов. Настало время дать этому слову более точное определение.

Шаблонным инстанцированием называется процесс порождения типов и функций из обобщённых шаблонных определений.

В книгах по шаблонам, таких как [8] традиционно много места отводится тому, что называется “Separation model”. Это модель инстанцирования, когда код шаблона находится в одной единице трансляции, а инстанцирование происходит в другой. Начиная с C++11 [5] экспорт шаблонов был запрещён, а Separation model объявлена устаревшей. Поэтому мы далее будем полагать, что код шаблона и инстанцирование происходят в одной единице трансляции.

Самое важное, что нужно знать про инстанцирование – оно по стандарту обязано работать лениво. Это означает, что проверены на корректность будут только те части шаблона, которые действительно были использованы. Давайте рассмотрим пример:

```
template <int N>
class Danger {
public:
  typedef char block[N]; // would fail for N<=0
};

template <typename T, int N>
class Tricky {
public:
  void test_lazyness() {
      Danger<N> no_boom_yet;
  }
};

int main(void)
{
  Tricky<int, -2> ok;
  return 0;
}
```

Несмотря на то, что инстанцирование определяет typedef с некорректным параметром размера массива, это не является ошибкой, поскольку реально этот зависимый тип нигде не был использован. Ленивость вынуждает компилятор откладывать инстанцирование любой части шаблона так долго, как это возможно.

Ещё более разрешающей идиомой в правилах инстанцирования является SFINAE, что означает **substitution failure is not an error**. Она означает, что некорректная подстановка шаблонных параметров сама по себе не является ошибкой. SFINAE является очень мощной техникой, позволяющей делать интересные вещи. Например определить на этапе компиляции имеет ли некий тип вложенный зависимый тип с определённым именем.

```
#include <cstdio>

using namespace std;

```

```
5  template <typename T>
6  struct has_typedef_foobar {
7    typedef char yes[1];
8    typedef char no[2];
9
10   template <typename C>
11   static yes& test(typename C::foobar*);
12
13   template <typename>
14   static no& test(...);
15
16   static const bool value = sizeof(test<T>(0)) == sizeof(yes);
17 };
18
19 struct foo {
20   typedef float foobar;
21 };
22
23 struct bar {
24 };
25
26 int main(void)
27 {
28   printf("foo: %d, bar: %d\n", has_typedef_foobar<foo>::value,
       has_typedef_foobar<bar>::value);
29   return 0;
30 }
```

Этот код выведет результат `foo: 1, bar: 0`, верно определив наличие зависимого типа `foo::foobar` в `foo` и его отсутствие в `bar`. Эта техника может казаться взрывающей мозг, но я рекомендую изучить идею, пока она не станет ясна.

4.7 Разрешение имён с учётом шаблонов

Вернёмся к рассмотренной на прошлой лекции теме разрешения имён. Если бы всегда при работе с шаблонами возникала необходимость явно указывать шаблонные аргументы, код довольно скоро разрастался бы, обрастая ненужными подробностями. Скажем так ли нужно писать `concat<std::string, int>(s, 3)` если компилятор достаточно умён, что-

бы понять и более простую форму `concat(s, 3)` из контекста.

При выводе параметров, компилятор строит вывод для каждого параметра и выдаёт ошибку, если вывод расходится. Пусть снова дана функция `max` и написан неоднозначный вызов для неё:

```
template <typename T> inline const T
max (const T a, const T b)
{
  return (a > b) ? a : b;
}

...

int g = max(1, 1.0);
```

Что произойдёт? Вывод типов для данного шаблона будет неоднозначен и подстановка провалится. Заметьте – подстановка провалена не означает, что программа некорректна (снова SFINAE). Возможно где-то определён более подходящий шаблон, скажем

```
template <typename T, typename U> inline const T
max (const T a, const U b) { ... }
```

и подстановка в него завершится успехом.

Нужно быть крайне осторожным с символьными литералами. На первый взгляд вызов `max("Apple", "Pear")` будет приведен к `const char*`, но это неверно. Статическими типами для этих двух строк будут `char const[5]` и `char const[6]`, и C++ посчитает их разными типами.

Подстановка также может провалиться, если результатом подстановки служит код, некорректный по стандарту C++. Например:

```
template<typename T>
typename T::ElementT at (T const& a, int i)
{
    return a[i];
}

void f (int* p)
{
    int x = at(p, 7);
}
```

Здесь для p будет выведен тип `int*`, который создаёт неверное присваивание в точке использования и такая подстановка также будет провалена.

4.8 Немного истории (трюк Бартона-Накмана)

В долгой истории шаблонов C++ встречаются примеры интересных обходных решений, изучение которых полезно не с практичекой а с общеразвивающей точки зрения. Одно из них, которое сейчас уже не актуально, решало интересную и важную проблему. Итак, перенесёмся в 1994-й и представим что у нас ещё нет перегрузки шаблонных функций и пространств имён.

Зато у нас есть некий шаблон Array, для которого необходимо переопределить оператор сравнения на равенство ==. Первый вариант – определить его как член класса, но это действительно плохая идея, поскольку первый аргумент (указатель на `this`) будет преобразоваться иначе чем второй (см. выше про переопределние бинарных операций, где разбирается такой пример). Поэтому конечно его хочется определить вне класса:

```
1  template<typename T>
2  class Array {
3    public:
4    ...
5  };
6
7  template<typename T>
8  bool operator == (Array<T> const& a, Array<T> const& b)
9  {
10   ...
11 }
```

Однако, если у нас нет перегрузки шаблонных функций, это создаёт проблему: ни одна больше функция с названием `operator ==` не может быть объявлена в этом же пространстве имён. Тем не менее, вполне логично, что оператор сравнения может быть нужен в том же пространстве имён для других классов. Бартон и Накман предложили определить этот оператор в классе как функцию-друг.

```
1  template<typename T>
2  class Array {
3    public:
```

```
4       ...
5       friend bool operator == (Array<T> const& a,
6                                Array<T> const& b) {
7           return ArraysAreEqual(a, b);
8       }
9   };
```

Пусть эта версия класса `Array` инстанцирована для типа `float`. Тогда оператор сравнения объявлен как результат этого инстанцирования, но сам по себе не является шаблоном функции и, таким образом, может быть перегружен как обычная функция.

В связи с тем, что `operator == (Array<T> const&, Array<T> const &)` определён внутри определения класса, он неявно считается встраиваемой функцией. Поэтому там и заложена делегация к не-инлайновой ArraysAreEqual, которая уже вряд ли будет конфликтовать с другими именами.

4.9 Шаблонные шаблонные параметры

Представьте, что вы используете CRTP для предоставления “интерфейса” к набору дочерних шаблонов, при этом как родитель, так и потомки параметризованы:

```
1   template <typename DERIVED, typename VALUE> class interface {
2       void do_something(VALUE v) {
3           static_cast<DERIVED*>(this)->do_something(v);
4       }
5   };
6
7   template <typename VALUE> class derived : public interface<
        DERIVED, VALUE> {
8       void do_something(VALUE v) { ... }
9   };
10
11  typedef interface<derived<int>, int> derived_t;
```

Строчка, определяющая `derived_t` содержит неприятное дублирование типа `int`, который на самом деле один и тот же и для потомка и для предка. Мало того это даёт возможность ошибки при опечатке. Чтобы явно отобразить, что шаблон параметризуется зависимым параметризованным типом:

```
template <template <typename> class DERIVED, typename VALUE>
    class interface {
    void do_something(VALUE v) {
        static_cast<DERIVED<VALUE>*>(this)->do_something(v);
    }
};

template <typename VALUE> class derived : public interface<
    DERIVED, VALUE> {
    void do_something(VALUE v) { ... }
};

typedef interface<derived, int> derived_t;
```

Обратите внимание на `template <typename> class DERIVED` – именно такой синтаксис внутри списка аргументов показывает, что DERIVED это шаблонный тип, специфицированный неким зависимым. Александреску использует шаблонные шаблонные параметры для реализации policy-классов

```
template <template <typename> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
};
```

Считается, что `WidgetManager` "знает" о том, что ему нужна `CreationPolicy` именно для `Widget`. При этом нужно понимать, что если шаблонный шаблонный класс параметризован более чем одним зависимым аргументом, это должно быть явно указано:

```
template <template <typename, typename> class CreationPolicyEx>
class WidgetManager : public CreationPolicyEx<Widget,
    WidgetPattern>
{
   ...
};
```

Разумеется, явно может быть указано сколько угодно зависимых типов.

При использовании CRTP с закрытой базой, бывает полезно объявить закрытую базу другом, чтобы дать ей доступ к своей закрытой части.

```
/* Transport<service> is implementation detail */
template<template<typename> class Transport>
class service : private Transport<service> {

    /* since we derive privately, make the transport layer a
       friend of us,
      so that it can cast its this pointer down to us */
    friend class Transport<service>;

public:
    typedef Transport<service> transport_type;

    /* common code */
    void do_something() {
        this->send(....);
    }
};

template<typename Service>
class tcp {
public:
    void send(....) {

    }
};

template<typename Service>
class udp {
public:
    void send(....) {

    }
};

typedef service<tcp> service_tcp;
typedef service<udp> service_udp;
```

И вот теперь, когда разминка с шаблонами закончена, можно переходить к серьёзным вещам.

4.10 Метапрограммы

Шаблонное метапрограммирование было открыто в 1994-м году. Эрвин Анрух (Erwin Unruh) на заседании комитета по стандартизации сделал доклад, из которого следовало, что шаблоны могут быть использованы для вычисления на этапе компиляции и продемонстрировал генератор простых чисел, который на этапе компиляции выводил в виде сообщений об ошибках простые числа от 2 до заранее заданного настраиваемого предела. Но этот генератор довольно сложен. На простом примере можно показать как работает метапрограммирование в целом:

```
1  /* primary template to compute 3 to the Nth */
2  template<int N>
3  class Pow3 {
4    public:
5      enum { result=3*Pow3<N-1>::result };
6  };
7
8  /* full specialization to end the recursion */
9  template<>
10 class Pow3<0> {
11   public:
12     enum { result = 1 };
13 };
14
15 int main()
16 {
17     std::cout << "Pow3<7>::result = " << Pow3<7>::result
18               << '\n';
19 }
```

Здесь значение `Pow3<7>::result` будет вычислено до выполнения программы на этапе компиляции и впечатано в генерированный ассемблер как число. Давайте посмотрим по пунктам как это работает.

1. TODO: расписать

В метапрограммах можно делать также ветвления if-then-else. Напишем простой шаблон:

```
1  // primary template: yield second or third argument depending
       on first argument
```

```
template<bool C, typename Ta, typename Tb>
class IfThenElse;

// partial specialization: true yields second argument
template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb> {

  public:
    typedef Ta ResultT;
};

// partial specialization: false yields third argument
template<typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb> {
  public:
    typedef Tb ResultT;
};
```

С помощью этого шаблона теперь можно легко написать, например, вычисление целочисленного квадратного корня во время компиляции:

```
// primary template for main recursive step
template<int N, int LO=1, int HI=N>
class Sqrt {
  public:
    // compute the midpoint, rounded up
    enum { mid = (LO+HI+1)/2 };

    // search a not too large value in a halved interval
    typedef typename IfThenElse<(N<mid*mid),
                                Sqrt<N,LO,mid-1>,
                                Sqrt<N,mid,HI> >::ResultT
            SubT;
    enum { result = SubT::result };
};

// partial specialization for end of recursion criterion
template<int N, int S>
class Sqrt<N, S, S> {
  public:
    enum { result = S };
};
```

Рассмотренные примеры показывают, что мы можем делать в метапрограммах ветвления и циклы, хранить промежуточные данные и нам доступна целочисленная арифметика. Всего этого достаточно для того, чтобы доказать Тьюринг-полноту шаблонов C++.

4.11 Домашняя наработка по шаблонам

Доработать обёрточный класс CFile сделав его шаблонным. Какие выгоды это принесло?

Написать на шаблонах код, генерирующий числа Каталана, числа Бернулли и числа Фибоначчи.

5 Правила для исключений

Управляющие конструкции в программе делятся на local flow – обычные управляющие конструкции, такие как `if`, `for`, etc и non-local flow. Последние пока не рассматривались, поскольку использование их в C-подмножестве C++ или даже при чистом объектно-ориентированном программировании, в том числе и с шаблонами, обычно не нужно и ведёт к проблемам. Но на этой лекции будет идти речь в основном о нелокальных техниках, таких как исключения C++. Давайте прежде перечислим некоторые типы нелокальной передачи управления:

- Нелокальный GOTO (для C и для C++ это `setjmp` и `longjmp`)
- Continuations (более структурная форма GOTO, в C и C++ можно имитировать передачей указателя на функцию-continuation),
- Исключения (для C++ это C++ exceptions)
- Coroutines, сопрограммы (популярны в Lua, но и в C++ после выхода нового стандарта есть экспериментальная библиотека Boost.Coroutine о которой мы возможно в будущем поговорим)
- Генераторы в стиле Python (являющиеся на самом деле подвидом coroutines, в C++ реализуемы через input iterators, о них мы тоже поговорим позднее)
- Экзотические вещи, такие как нотификации (обобщение исключений, реализованное в Lisp, не встречается в C++, но в принципе реализуемо через rethrow exceptions)
- Возможности операционной системы, такие как fibers (кооперативные легковесные нити исполнения)

Среди всех нелокальных управляющих конструкций, исключения выделяются своей особой ролью при обработке исключительных ситуаций, возникающих во время работы программы. Собственно от этого они и получили своё имя. Важно понимать, что является и что не является исключительной ситуацией чтобы не использовать исключения там где их не нужно использовать и наоборот не отказываться от использования исключений там, где они упрощают жизнь.

- Исключительной ситуацией не является ошибка пользователя, такая как ввод неверно форматированного числа в ячейку электронной таблицы или попытка открыть файл неверного формата. Это нормальная ошибочная ситуация, которая может быть обработана локально.

- Исключительной ситуацией не является ошибка времени выполнения, такая как деление на ноль или assertion failure в программе. Лучшее что можно сделать с такой ошибкой это завершить выполнение и сбросить трассу.

- Исключительные ситуации характеризуются тем, что после них консистентное состояние программы восстановимо. Всегда лучше завершить программу, чем оставить её и её данные неконсистентными, но если у нас произошло переполнение стека, исчерпание памяти, не хватает прав на создание временного файла – всё это обрабатываемые и потенциально устранимые проблемы.

- Исключительную ситуацию как правило нельзя обработать на том уровне, где она возникла. Глупо ожидать, что программа сортировки массива, столкнувшаяся с нехваткой памяти, обработает эту нехватку где-то между разбиением массива и рекурсивным вызовом на каждую из получившихся частей. Скорее всего она должна будет вернуть (возможно через неопределённое число уровней рекурсивного возврата) управление программе-драйверу, которая знает что делать (может быть захочет отсортировать тот же массив менее оптимальным алгоритмом но без использования дополнительной памяти).

Верный навык отличать где и как использовать исключения требует времени и внимания к деталям.

5.1 Исключения под капотом

Для того, чтобы лучше понимать исключения, бывает полезно попробовать реализовать их руками (как это было сделано с виртуальными функциями ранее). Пример заменяет тысячу слов. Допустим функция `allocate` вызывает `malloc` чтобы выделить n байт, и возвращает указатель, который вернул `malloc`. Если же `malloc` возвращает нулевой указатель, что индицирует нехватку памяти, `allocate` делает нечто похожее на возбуждение исключения `Allocate_Failed` (без размотки стека

и вызова деструкторов пока что). Исключение само по себе имеет тип `jmp_buf`, который определён в стандартном хедере `setjmp.h`:

```
#include <setjmp.h>

int Allocation_handled = 0;
jmp_buf Allocate_Failed;
```

`Allocation_handled` ноль, пока обработчик не инстанциирован и `allocate` проверяет `Allocation_handled` прежде чем возбуждать исключение:

```
void *allocate(unsigned n) {
  void *new = malloc(n);
  if (new)
    return new;
  if (Allocation_handled)
    longjmp(Allocate_Failed, 1);
  assert(0);
}
```

И теперь можно сделать практически `try` и `catch`:

```
/* try { */
Allocation_handled = 1;
if (setjmp(Allocate_Failed)) {
  goto alloc_except_label;
}

/* ... here some code, calling somewhere allocate() */

/* } */
/* catch (...) { */
alloc_except_label:
/* ... here processing exception */
Allocation_handled = 0;
/* } */
```

Язык C++ в отличие от C предоставляет достаточное количество синтаксического сахара для организации исключений.

5.2 Исключения в C++

Поддержка исключений в C++ средствами языка проста и интуитивна. Код, обнаруживший исключение, генерирует объект исключения инструкцией `throw`. Объект исключения может быть объектом некоего класса или любой другой переменной, или даже константой или литералом – язык ничем здесь не ограничивает пользователя (и это уникально, поскольку большинство других языков требуют чтобы объект исключения входил в некую иерархию). Разумеется хороший тон это генерировать потомка `std::exception` или одного из его потомков в иерархии стандартной библиотеки, но это совершенно не обязательно.

Фрагмент кода выражает своё желание обрабатывать исключение с помощью конструкции `catch`. Результатом `throw` является раскрутка стека до тех пор, пока не будет найден подходящий `catch`, которому и передаётся управление.

```
1  class MathErr {};
2  class Overflow : public MathErr {};
3
4  void foo()
5  {
6    try {
7      /* ... dangerous code here ... */
8    }
9    catch (Overflow) {
10     /* ... dealing with overflow ... */
11   }
12   catch (MathErr) {
13     /* ... dealing with other guys ... */
14   }
15 }
```

В приведённом примере исключения перехватываются по значению. Это очень плохо (вспоминаем проблему срезки, рассмотренную выше). Хорошо написанный код перехватывает исключения по константной ссылке или по указателю (если это конечно не исключения POD-типов, которые вы также имеете право возбуждать, их можно перехватывать по значению).

Важно понимать, что обработчики проверяются по порядку перечисления. Поэтому ставить обработчик `MathErr` выше чем `Overflow`, означает сделать последний чуть более чем бесполезным.

Опасной и неприятной конструкцией является перехват всех возможных исключений:

```
void foo()
{
  try {
    /* ... dangerous code here ... */
  }
  catch (...) {
    /* ... dealing with everything??? ... */
  }
}
```

Если в вашем коде вы вынуждены использовать такую конструкцию, значит у вас проблемы с проектированием. Единственное исключение – если вы используете внутри `catch (...)` конструкцию перевыброса исключения

```
void foo()
{
  try {
    /* ... critical resource usage here ... */
  }
  catch (...) {
    /* ... cleaning up critical resource ... */
    throw; /* and rethrow */
  }
}
```

Перевыброшенное исключение продолжит размотку стека пока не будет поймано кем-то, кто знает как его обработать. Если выброшенное исключение не нашло своего обработчика, оно инициирует вызов функции `terminate` и аварийный выход из программы. Кроме перевыброса того же исключения (`throw` без параметра, которое может встретиться только внутри `catch`-блока) вы можете в любом месте использовать `throw` с параметром, для того чтобы бросить произвольное исключение.

Вообще освобождение ресурсов с помощью деструкторов предпочтительней использования для этой цели `try`-блоков с перехватом всех исключений.

5.3 Гарантии безопасности исключений

Исключения, являясь частным случаем нелокальной управляющей конструкции, добавляют строчки в неявный контракт на каждый метод и добавляют неожиданные дуги возможного выхода в каждом месте вызова небезопасной с точки зрения генерации исключений функции. Поэтому при проектировании кода, серьёзно использующего исключения, программист несёт дополнительную интеллектуальную нагрузку. Он должен следить за тем, что называется безопасностью кода относительно исключений.

Существует три гарантии безопасности, которые может предоставлять код:

- **Базовая гарантия** заключается в том, что сбой при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты в согласованном (но не обязательно предсказуемом) состоянии, т.е. пригодными к дальнейшему использованию.
- **Строгая гарантия** обеспечивает транзакционную семантику: при сбое операции гарантируется неизменность состояния программы относительно задействованных в операции объектов.
- **Гарантия бессбойности** означает, что функция не генерирует исключений.

При написании кода на C++ следует придерживаться хотя бы одной из этих гарантий и предпочитать RAII. Также исключения не должны генерироваться деструкторами, функциями, освобождающими ресурсы и функциями обмена.

5.4 Безопасное копирование и присваивание

Вернемся к шаблонному классу стека и теперь попытаемся сделать его безопасным относительно исключений. На первый взгляд интерфейс класса стек может выглядеть как-то вот таким образом:

```
1 template <typename T>
2 class Stack
3 {
```

```
public:
  Stack(): m_v(new T[10]), m_vsize(10), m_vused(10) {}
  ~Stack() { delete [] v; }
  Stack(const Stack&);
  Stack& operator= (const Stack&);
  void Push(const T&);
  T Pop();
private:
  T* m_v;
  size_t m_vsize;
  size_t m_vused;
};
```

Конструктор и деструктор уже тривиально безопасны (если исключения не выходят за `delete[]`, но они и не должны за него выходить). Сложнее с копированием и копирующим присваиванием. Проще всего сделать их безопасными, определив шаблон общей вспомогательной функции `safe_copy`, копирующей буфер в заданный с перехватом всех возможных исключений и освобождением ресурсов.

```
template <typename T>
T *safe_copy(const T* src, size_t srcsize, size_t destsize)
{
  size_t idx;
  T * dest;
  assert (destsize >= srcsize);
  dest = new T[destsize];
  try
    {
       for (idx = 0; idx != srcsize, ++idx)
         dest[idx] = src[idx];
    }
  catch (...)
    {
       delete [] dest;
       throw;
    }
  return dest;
}
```

Внутри этой функции исключения могут быть сгенерированы:

- В функции `new`, переопределённым для `T` оператором `operator new` или конструктором `T` – в этом случае никакой памяти выделено не будет и утечки не произойдёт, а исключения покинут `safe_copy`
- В функции `operator=` или `operator[]` типа `T` при копировании. В этом случае исключения будут перехвачены, ресурсы освобождены и исключения отпущены дальше.

Имея эту функцию, мы уже можем разработать симпатичный конструктор копирования

```
template <typename T>
Stack<T>::Stack(const Stack& rhs) : m_v(safe_copy(rhs.m_v, rhs.
    m_vsize, rhs.m_vsize)),
                                    m_vsize(rhs.m_vsize),
                                    m_vused(rhs.m_vused)
{
}
```

Единственный источник исключений здесь – `safe_copy`, в которой никакой утечки быть не может. Теперь присваивание:

```
template <typename T>
Stack<T>& Stack<T>::Stack(const Stack<T>& rhs)
{
  T *v_new;

  if (this == &rhs)
    return *this;

  v_new = safe_copy(rhs.m_v, rhs.m_vsize, rhs.m_vsize);
  delete [] m_v;
  m_v = v_new;
  m_vsize = rhs.m_vsize);
  m_vused = rhs.m_vused);
  return *this;
}
```

Таким образом состояние остаётся неизменным и даже при генерации исключения, оно выходит наружу не нарушая консистентности объекта.

Подобные же средства обеспечения безопасности можно предложить для помещения элемента в стек:

```
template <typename T>
void Stack<T>::Push(const T& t)
{
  size_t vsize_new;
  T *v_new;

  if (m_vused < m_vsize)
    {
      m_vused = t;
      m_vused += 1;
      return;
    }

  vsize_new = m_vsize*2 + 1;
  v_new = safe_copy(m_v, m_vsize, vsize_new);
  delete [] v;
  m_v = v_new;
  m_vsize = vsize_new;
}
```

Общий урок раздела заключается в том, что все места, где может быть сгенерировано исключение полезно собрать в одну вспомогательную функцию для использования безопасным образом.

5.5 Неявное копирование и безопасность исключений

Но проблемы как обычно подстерегают там, где их не ждали. Давайте рассмотрим метод Pop, который пока что остался не реализованным. К сожалению, его реализация только кажется простой.

```
template <typename T>
T Stack<T>::Pop(void)
{
  T result;

  assert(m_vused > 0);
  result = m_v[m_vused - 1];
  m_vused -= 1;
  return result;
}
```

Вся проблема с этим кодом заключается в его возможном использовании:

```
Stack<Sometype> s;
SomeType s2;
/* .. some code .. */
s2 = s.Pop();
```

В последней строчке проихсодит копирование возвращаемого SomeType в место назначения. Если в этот момент возникнет исключение, то все побочные эффекты со стеком уже окажутся учтены – элемент будет снят с верхушки, но он пропадёт безвозвратно. Именно поэтому в коде стандартной библиотеки и в любом безопасном относительно исключений коде, эти операции разделены:

```
template <typename T>
class Stack
{
public:
  Stack(): m_v(new T[10]), m_vsize(10), m_vused(10) {}
  ~Stack() { delete [] v; }
  Stack(const Stack&);
  Stack& operator= (const Stack&);
  void Push(const T&);
  T& Top();
  void Pop();
private:
  T* m_v;
  size_t m_vsize;
  size_t m_vused;
};
```

Метод Top() даёт верхний элемент но не модифицирует стек, метод Pop() модифицирует стек но не возвращает никакого значения. Основной урок здесь состоит в том, что безопасность исключений действительно влияет на проектирование класса и думать о ней необходимо уже на этапе проектирования.

5.6 Идиома PImpl для безопасного кода

5.7 Что нельзя деструкторам и можно нам

6 Два лица стандартной библиотеки

6.1 Контейнеры

6.2 От обработки строк в стиле C к std::string

6.3 От массивов к std::vector

6.4 Ассоциативные массивы

6.5 Итераторы

6.6 Алгоритмы

6.7 Блеск и нищета обобщения

6.8 Потоки ввода/вывода

7 Новые горизонты

Новый стандарт C++ предлагает много нововведений. Часть из них относится к тому, чего все давно ждали и успели освоить, что перекочевало из библиотеки boost или из technical reports к C++98. Но есть и такие, при работе с которыми требуется оперировать совершенно новыми понятиями и заставлять себя мыслить иначе, мыслить в духе действительно новых возможностей.

7.1 Rvalue references

Ссылки (references) это то, что наиболее сложно воспринимается при переходе с C на C++. Человеку с опытом программирования на C, может быть непонятно, зачем они нужны, когда уже есть такие удобные и привычные указатели, которые позволяют делать всё то же самое только лучше... Подлинное осознание того, каким именно инструментом являются ссылки и как ими оперируют, приходит с опытом C++. При переходе на новый стандарт C++11, идея о том, что теперь есть ещё один вид ссылок – ссылки на rvalue (rvalue references) кажется новаторской и неочевидной (особенно неочевидна её полезность). Скорее всего, многим придётся преодолеть внутреннее сопротивление, прежде чем rvalue references станут удобным и привычным инструментом. Но прежде, чем переходить к их описанию, есть смысл немного вспомнить о том, что такое rvalue и lvalue.

7.1.1 Ещё раз rvalue и lvalue

Термин lvalue был, прежде чем перейти в C++, документально зафиксирован в стандарте языка C, поэтому логично начать изложение с него. Для языка C, lvalue (от left hand side value) – то, что может появиться слева в выражении присваивания. Формально (6.3.2.1 стандарта C99), “An lvalue is an expression with an object type or an incomplete type other than void” (под object type в стандарте понимаются все типы, не являющиеся function type или incomplete type).

```
1 int c = a * b;
2 a * b = 42;
3 foo() = 42;
```

Собственно, в стандарте языка C90 и последовавшем за ним C99 нет термина rvalue. В C99 есть оговорка “What is sometimes called “rvalue” is in this International Standard described as the “value of an expression””, позволяющая предположить, что к этому времени этот термин существовал и использовался, но фиксировать его строго в случае языка C было не нужно.

Всё изменилось с приходом языка C++, который принёс с собой references. Выражение из прошлого примера:

```
foo() = 42;
```

может быть ошибкой если foo возвращает значение, но совершенно корректно, если foo возвращает ссылку. Чтобы разрешить эту неоднозначность, стандарт C++ 98 объявляет (3.10.1) два термина – lvalue и rvalue, причём каждое выражение языка C++98 является либо тем, либо другим. При этом определение в (3.10.2) гласило “An lvalue refers to an object or function”.

То есть lvalue это не первоклассный объект как в C, а просто некое выражение ссылающееся на область памяти. Таким образом в качестве главного различия выступает возможность взять адрес, а не отношение к операции равенства.

```
int& foo();
foo() = 42;
int* p1 = &foo();
int foobar();
int* p2 = &foobar();
```

В некотором смысле, lvalue в C++98 содержательно становится locator value (а не left hand side value). Так, например если есть указатель val, то выражение val+1 можно разыменовать, но нельзя взять его адрес.

```
int *val;
int correct = *(val + 1);
int **wrong = &(val + 1);
```

Стандарт C++11 объявляет (3.10) пять терминов: lvalue, rvalue, xvalue, а также glvalue (обобщение lvalue и xvalue) и prvalue (более узкая специализация rvalue). Базовыми терминами являются знакомые нам lvalue, rvalue и новый xvalue (от expiring value). Забегая вперёд – например xvalue является результат функции возвращающей rvalue ссылку. То, что понималось под rvalue в C++98, в C++11 стало prvalue (в то время как

rvalue обобщает xvalue и prvalue), а новый термин glvalue обобщает lvalue и xvalue. Можно (неточно, но образно) сказать, что prvalue это то, брать адрес от чего нельзя, lvalue – от чего можно, а xvalue – от чего бесполезно.

Если всё это не вызывает вопросов, настало время перейти к объяснению ссылок.

7.1.2 Отличаем rvalue refernces от lvalue references

В C++11, если X это тип, то `X&` это lvalue reference, а `X&&` это rvalue reference для этого типа (8.3.2.2). При этом важно понимать – `X&, X&&` – это разные типы, но семантически они эквивалентны и о них можно вместе говорить как о "ссылках вообще". Тем не менее, поскольку это разные типы, для них работает перегрузка:

```
int a, b;
foo(int &x);
foo(int &&x);
foo(a * b);
foo(a);
```

При этом сама по себе rvalue reference может быть lvalue (и является им, если она именована).

```
int &&x = a * b;
foo(x);
```

Это очень важное правило, потому что представьте ситуацию, в которой объявлен класс Base и в нём есть необходимость определить конструкторы:

```
Base(Base const & rhs);
Base(Base&& rhs);
```

Теперь в наследуемом от него классе Derived, есть необходимость эти конструкторы переопределить:

```
Derived(Derived const & rhs) : Base(rhs) {};
Derived(Derived&& rhs) : Base(rhs) {};
```

Перемещающий конструктор переопределён неверно – поскольку rhs имеет имя, оно есть lvalue, а значит `Base(rhs)` в данном случае вызовет снова `Base(Base const & rhs)`, что, вероятно, не желательно. Вместо этого следует писать

```
Derived(Derived&& rhs) : Base(std::move(rhs)) {};
```

Сюрпризы подобные этому ожидают, казалось бы, в очевидных вещах. Ссылка на ссылку – невозможна. Это все знают и тысячу раз отвечали на собеседованиях. Что же, этот ответ всё ещё верен и стандарт C++11 (8.3.2.5) запрещает ссылки на ссылки, указатели на ссылки и массивы ссылок. Но как быть, когда в шаблоне или при typedef получается смесь правых и левых ссылочных типов? Это способно поставить в тупик. Давайте разберёмся с тем, как работают ссылки в C++11.

7.1.3 Свёртка ссылочных типов

Стандарт (8.3.2.6) определяет следующие правила свёртки ссылок, применимые для определений typedef и decltype, а также параметров шаблонов:

`A& &` становится `A&` `A& &&` становится `A&` `A&& &` становится `A&` `A&& &&` становится `A&&`

Пусть определён шаблон

```
template<typename T>
void foo(T&& t);
```

Пусть теперь foo вызвана с аргументом x типа X, причём x является lvalue. Тогда T разрешается в `X&`, а реальным типом аргумента t будет `X& &&`, то есть (см. выше правила свёртки) `X&`. Если же x является rvalue, то T разрешается в X и реальным типом аргумента t будет `X&&`.

Стандарт также определяет функтор remove_reference (20.9.7.2), который позволяет получить не-ссылочный тип из ссылочного. Он работает похоже на static_cast и прочие привычные вещи. Аналогично (но наоборот) работает пара `add_lvalue_reference`/`add_rvalue_reference`.

```
int *p = (std::remove_reference<int&>::type *)0;
```

(Я надеюсь, никто никогда не напишет такую строчку в реальном коде).

Теперь настала пора посмотреть зачем же комитетом по стандартизации были введены все эти сложности.

7.1.4 Применение rvalue ссылок

В условной реализации C++11, удовлетворяющая стандарту реализация библиотечной функции std::swap (20.2.2) может выглядеть, например, так:

```
template<class T>
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

В этой реализации нет операций копирования (без которых не обойтись в C++98). Функция std::move позволяет здесь реализовать семантику перемещения, поскольку она работает так: из любого аргумента, std::move делает rvalue. Эта функция в свою очередь, в удовлетворящем стандарту (20.2.3) виде может быть определена так:

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
  typedef typename remove_reference<T>::type&& RvalRef;
  return static_cast<RvalRef>(a);
}
```

Методологически полезно проследить, что будет если вызвать std::move с аргументом, имеющим тип X и являющимся lvalue.

1) T будет разрешено в `X&` 2) тип аргумента `X& &&` a будет свёрнут в `X&` a 3) `remove_reference<X&>::type&&` будет означать `X&&`, который и будет значением RvalRef

Итак, получим эквивалент:

```
X&& std::move(X& a) noexcept
{
  return static_cast<X&&>(a);
}
```

Что и требовалось.

Сама идея, что теперь обмен значениями может быть семантически выражен как обмен значениями, а не копирование (а значит и соптимизирован соответствующим образом) после её осознания, приводит людей в лёгкую эйфорию. В конце концов стремление к максимальной эффективности – нормальное стремление. Тем не менее, с функцией std::move следует быть осторожным. Часто идея вернуть std::move(x) вместо x, когда реально нужно rvalue и хочется избежать лишних копирований, приводит к отключению оптимизаций возвращаемого значения в компиляторе и результат ухудшается. Большинство таких трансформаций на современных компиляторах требует замеров и не должны выполняться preliminary:

```
1 X foo()
2 {
3   X x;
4   return std::move(x);
5 }
```

Итак, rvalue references позволяют (совместно с `std::move`) по новому взглянуть на ссылки и открывают простор к более тонкому различию таких терминов естественного языка как “копирование”, “присваивание значения”, “присваивание результата” на языке C++, что открывает как простор к оптимизациям, так и возможности более аккуратного выражения старых идиом (хороший пример блестящего использования правых ссылок это `std::unique_ptr` который в новом стандарте пришёл на смену печально известному `auto_ptr`). Их освоение – важная черта отличающая программиста на C++11.

За кадром этого изложения осталось использование `std::forward` (20.2.3), предлагающееся на самостоятельное изучение, как упражнение на практическое применение rvalue refernces.

7.2 Lambda expressions

Ещё одной непросто воспринимаемой для программиста с опытом C++ концепцией являются lambda expressions. С одной стороны понять их кажется чуть проще, в конце концов все когда-нибудь писали функторы. С другой стороны, настоящее осознание мощи и гибкости этого инструмента также приходит только с опытом.

Следующий код с первого взгляда немного шокирует:

```
1 int main()
```

```
{
  auto func = [] () -> int { std::cout << "Hello world"; return
      0; };
  return func();
}
```

Здесь, согласно стандарту (5.1.2): `[]` – capture specificator, определяет lambda-выражение, внутри этих скобок также можно задать захватываемый контекст. `()` – argument list, здесь можно задать список параметров `-> int` – означает что возвращаемое значение имеет тип int. Внутри `{}` – тело lambda-выражения.

Отдельно следует остановиться на спецификаторе auto (7.1.6.4). Его употребление вместо типа переменной означает, что тип переменной должен быть выведен из инициализирующего её выражения (есть также специальное назначение auto, когда он употребляется как тип функции, который должен быть в этом случае выведен из выражения, стоящего под return).

```
auto x = new auto('a');
```

Каким будет выведен тип func, пока не так существенно. Рассмотренный пример использования lambda-выражения, может быть переписан несколько проще.

```
int main()
{
  auto func = [] { std::cout << "Hello world"; return 0; };
  return func();
}
```

По стандарту, если список аргументов не указан, то он пуст (5.1.2.4) и возвращаемый тип выражения выводится из return в его теле, если он указан и единственный (там же).

Лямбда-выражения в их простейшем виде хороши для объявления на месте несложных функторов. Стандарт (5.1.2.1) приводит простой пример:

```
void abssort(float *x, unsigned N)
{
  std::sort(x,
            x + N,
            [] (float a, float b) { return std::abs(a) < std::
                abs(b); });
```

```
}
```

Тем не менее, в лямбда-выражениях есть и более интересные детали для рассмотрения, одна из которых – возможность захвата контекста.

7.2.1 Захват контекста

Список для захвата контекста пишется в квадратных скобках. Правила формирования списка для захвата контекста регулируются стандартом (5.1.2.8). Существует два специальных символа – "=" и "&". Переменные захватываемые по значению, входят в список без модификаторов, захватываемые по lvalue reference – с модификатором "&"

```
[foo, &bar]
```

Употребление "&" означает, что "может быть захвачена по ссылке любая переменная из контекста", тогда захватываемая переменная не должна предваряться & и всё равно будет захвачена по ссылке (по ссылке значит по lvalue ссылке, говоря точно).

```
[&, foo]
```

Употребление "=" означает, что если имя захватываемой переменной не предварено символом "&", то она будет захвачена по значению, а не по ссылке (5.1.2.14).

Кроме того, употребление любого из спецсимволов отдельно, означает, что "захвачен this", тогда захватываемая переменная из контекста класса или структуры не должна предваряться `this->`. В случае "=", упоминание this – ошибка (5.1.2.8) Такая неоднозначность может несколько запутывать.

Важно помнить, что захват по ссылке это по умолчанию захват по константной ссылке и нужно ключевое слово mutable чтобы это изменить (5.1.2.5). При этом круглые скобки списка параметров если используется mutable опускать нельзя:

```
class Foo
{
    int m_x;
  public:
    Foo () : m_x( 3 ) {}
    void func ()
      {
```

```
8           /* ok: */
9           [=] { std::cout << m_x; } ();
10          [&, m_x] () mutable { m_x = 5; } ();
11
12          /* errors: */
13          [&, m_x] { m_x = 5; } ();
14          [&, &m_x] () mutable { m_x = 5; } ();
15          [=, this]{ std::cout << m_x; } ();
16          [m_x, m_x]{ std::cout << m_x; } ();
17      }
18  };
```

Кроме простых правил захвата контекста, которые следует знать, есть и несколько более сложные, знание которых также полезно.

7.2.2 Элементы высшего пилотажа

Внутри lambda-выражения может быть использован тип переменной из контекста, для чего нет необходимости делать отдельно захват (5.2.1.18):

```
1  void f(float x)
2  {
3    [=] { decltype(x) y1 = x; std::cout << y1; } ();
4  }
```

Также lambda-выражения могут быть вложенными. Это довольно странно, учитывая, что в C++ до сих пор нет вложенных функций, но тут можно выдвинуть такой аргумент, что lambda-выражение представляется скорее классом, чем функцией (см. связанный контекст), а вложенные классы в C++ есть.

```
1  int main()
2  {
3    auto nested = [] (int x)
4                  {
5                    return [](int y) { return y * 2; }(x) + 3;
6                  };
7
8    std::cout << m << endl;
9  }
```

Поскольку lambda-выражения типизированы, их можно использовать вместе с шаблонами C++

```
template <typename T>
void negate_all(std::vector<T>& v)
{
  for_each(v.begin(), v.end(), [] (T& n) { n = -n; } );
}
```

Отдельной и интересной темой является обработка исключений внутри lambda-выражений. Мне не удалось набрать достаточно материала чтобы делать выводы, но это место выглядит достаточно error-prone и мне кажется здесь надо соблюдать существенную осторожность (пока Герб Саттер не порадует нас книгой на эту тему, я надеюсь).

Теперь пришло время разобраться с тем, как же типизированы lambda-выражения.

7.3 Осваиваем std::function

lambda-выражение с пустой спецификацией считается обычной функцией и может присваиваться указателям на функции в стиле C:

```
int test(void)
{
  typedef int (*fptr_t)();
  fptr_t fptr = [] { return 2; };
  return fptr();
}
```

Пусть теперь есть необходимость создать lambda-выражение с захватом контекста. Можно создать его использовав auto переменную:

```
int one_more_test(int x, int y)
{
  auto f1 = [&x, &y] { return x + y; };
  return f1();
}
```

Но что если есть необходимость вернуть не результат вычисления lamda-выражения, а его само, чтобы использовать его (со связанным контекстом) где-то ещё? Это можно сделать использовав auto на результат

функции, но можно выразить эту же идею более явно через std::function (20.8)

```
std::function<int ()> harder_test(int x, int y)
{
  return [&x, &y] { return x + y; };
}
```

С помощью std::function, есть возможность даже реализовать на C++ функции высшего порядка. Функция высшего порядка это функция, которая берёт на вход lambda-выражение и возвращает lambda-выражение.

```
int main()
{
  auto g = [](int x) -> function<int (int)>
     { return [=](int y) { return x + y; }; };

  auto h = [](const function<int (int)>& f, int z)
     { return f(z) + 1; };

  auto a = h(g(7), 8);

  std::cout << a << endl;
}
```

Итак, lambda-expressions позволяют (совместно с std::function) привнести в C++ некоторые полезные особенности функциональных языков, предоставляют удобный синтаксис для объявление функторов и использования алгоритмов стандартной билиотеки, и даже частично совместимы со старыми указателями на функции. Освоение lambda-выражений позволяет сделать код короче, выразительней и проще для поддержки.

8 За пределами беспредельного

8.1 Что такое technical report

8.2 Обзор boost

8.3 Расширения GNU

Список иллюстраций

Список литературы

[1] ISO/IEC, *«International Standard, Programming Languages»*, ISO/IEC 14882:1998, 1998

[2] ISO/IEC, *«International Standard, Programming Languages»*, ISO/IEC 9899:1990, 1990

[3] ISO/IEC, *«International Standard, Programming Languages»*, ISO/IEC 9899:1999, 1999

[4] ISO/IEC, *«International Standard, Programming Languages»*, ISO/IEC 9899:2011, 2011

[5] ISO/IEC, *«International Standard, Programming Languages»*, ISO/IEC 14882:2011, 2011

[6] Bjarne Stroustrup, *«The C++ Programming Language, The Special Edition»*. Reading, MA: Addison-Wesley, 2001.

[7] Scott Meyers. *«Effective C++ : 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)»*, MA: Addison-Wesley, 2005

[8] Davide Vandevoorde, Nicolai M. Josuttis. *«C++ Templates. The Complete Guide»*, Boston, Pearson Education, Inc., 2003.

[9] Grady Booch. *«Object-Oriented Analysis and Design with Applications (2nd Edition)»*

[10] Herb Sutter. *«Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions»*. Reading, MA: Addison-Wesley, 2000