C++ второй семестр

Купоросов Василий, Недиков Константин, Казначеев Дмитрий
Под редакцией Егора Суворова

9 июня 2017 г.

Содержание

1. Шаблоны **1**
1.1 Шаблонные классы . . . 1
1.1.1 C-style . . . 1
1.1.2 C++ style – шаблоны . . . 2
1.2 Классы с несколькими шаблонными параметрами . . . 3
1.3 Шаблонные функции . . . 3
1.4 Другие виды параметров шаблона . . . 5
1.5 Неинстанциированный шаблон . . . 6
1.6 Значение по умолчанию . . . 6
1.7 Специализация шаблона . . . 7
1.8 Частичная специализация . . . 7
1.9 Хитрый пример на понимание . . . 8
1.10 Экзотические случаи шаблонов . . . 9

2. Исключения **10**
2.1 Виды ошибок . . . 10
2.2 Обработка ошибок . . . 10
2.3 Обработка ошибок в C style . . . 11
2.3.1 Через возвращаемое значение . . . 11
2.3.2 Через глобальную переменную . . . 11
2.4 Недостатки C style . . . 12
2.5 C++ style, исключения . . . 12
2.6 Stack unwinding . . . 13
2.7 Типы исключений . . . 14
2.8 Исключения любого типа . . . 14
2.9 Подводные камни исключений . . . 15
2.9.1 Утечка памяти . . . 15
2.9.2 идиома RAII . . . 15

2.9.3 Исключения в конструкторе 15
2.9.4 Исключения в деструкторе 16
2.10 Гарантии при работе с исключениями 17
2.10.1 Basic guarantee 17
2.10.2 Strong guarantee 17
2.10.3 No throw 18

3. Стандартная библиотека шаблонов **19**
3.1 Контейнеры 19
3.2 Итераторы 20
3.3 Инвалидация итераторов 20
3.4 Исключения в STL 21
3.5 Несколько лирических отступлений 22
3.5.1 Почему обязательно закрывать файл после использования 22
3.5.2 Временные объекты 22
3.5.3 Операторы сравнения 23
3.6 Функторы 23
3.7 Алгоритмы 24
3.7.1 Микро-алгоритмы 24
3.7.2 Реализация алгоритмов 25
3.7.3 erase-remove idiom 25
3.8 Подробнее про итераторы 26
3.8.1 iterator_category 26
3.8.2 `distance` и `advance` 27
3.9 Ввод/Вывод 27
3.9.1 Иерархия классов 27
3.9.2 Методы для чтения/записи в консоль 28
3.9.3 Работа с файлами 28
3.9.4 Манипуляторы 29
3.9.5 Обработка ошибок 29
3.9.6 Ввод-вывод пользовательских типов 30
3.10 Приведение типов в `C++` 31

4. Стандарт C++11 **35**
4.1 Немного о стандартах 35
4.2 `default & delete` 35
4.3 `override & final` 36
4.4 Новые фишки конструкторов 37
4.5 `lvalue` и `rvalue` 38
4.5.1 `std::forward` и `universal reference` 39

4.6 auto & decltype . . . 40
4.7 nullptr . . . 41
4.8 Вычисления в compile time . . . 41
4.9 begin(), end(), for(:) . . . 41
4.10 lambda . . . 42
4.11 Variadic templates . . . 43
4.12 hash . . . 45
4.13 unique_ptr, array, tuple, regex . . . 45
4.14 std::function и std::bind . . . 46
4.15 std::thread . . . 47
4.16 Подробнее про многопоточность . . . 48
4.16.1 Краткое напоминание . . . 48
4.16.2 Потоки на однопроцессорном компютере . . . 48
4.16.3 Состояние гонки . . . 49
4.16.4 Атомарные типы и операции . . . 50
4.16.5 std::mutex . . . 51
4.16.6 std::lock_guard . . . 51
4.17 Множественное наследование . . . 51
4.18 Метапрограммирование . . . 53
4.18.1 SFINAE . . . 55
4.18.2 Отступление про typename . . . 57
4.18.3 enable_if . . . 57
4.18.4 type_traits . . . 58

1. Шаблоны

1.1. Шаблонные классы

Пусть мы написали свой класс, хранящий внутри себя объекты какого-то типа. Раньше эти типы фиксировались во время написания кода.

```
class MyArray {
private:
  int *array;
};

class Scoped_ptr {
  GaussNumber *ptr;
};
```

Но иногда нам хочется завести вектор или умный указатель другого типа. Приходится вручную переписывать весь класс. Как такую проблему решать в разных стилях написания кода?

1.1.1. C-style

Используем препроцессор. Обратите внимание на слэши в конце строк. Они нужны для многострочного дефайна, `##` – для склеивания `MyArray_` и типа, подставленного на место `TYPE`.

`MyArray.h`:

```
#define MyArray(TYPE) class MyArray_##TYPE { \
private:                                     \
  size_t size;                               \
  TYPE *array;                               \
public:                                      \
  TYPE get(size_t index) {                   \
    return array[index];                     \
  }                                          \
};
```

`main.c`:

```
#include "MyArray.h"
MyArray(int);     // подставляем определение класса с интами
MyArray(double);  // подставляем определение класса с даблами
int main() {
  MyArray_int a;
  MyArray_double b;
}
```

Если хочется посмотреть, как изменился код после препроцессора, можно запустить gcc с ключем -E.

Рассмотрим особенности данного подхода.

1) Все методы класса должны быть `inline`, потому что, если класс объявлен в разных файлах, то на стадии линковки будет `multiple definition`.

2) Препроцессор человек простой. Видит строку – заменяет её. Поэтому возникают спецэффекты. В приведенном ниже коде название функции `MyArray` заменится на какую-то чушь и будет очень много несуразных синтаксических ошибок. Нельзя использовать пробелы, типо `long long`.

```
#define MyArray(TYPE) . . .
MyArray(int);
MyArray_int arr;

int MyArray(int args) {return 42;}
```

3) Нельзя делать сложные объявления. Следующая строчка у препроцессора получится не лучшим образом.

```
MyArray(MyArray(int)) arr;
```

1.1.2. C++ style – шаблоны

Шаблоны позволяют делать всё то же самое, но на уровне компилятора. Этот стиль называется обобщенное (generic) программирование. Рассмотрим синтаксис.

`MyArray.h`:

```
template <typename T>
class MyArray {
private:
  size_t size;
  T *array;
public:
  T& get(size_t index) {
    return array[index];
  }
  T& operator[](size_t index);
  MyArray<T>& operator=(const MyArray<T>& obj);
};

template <typename T> // <class T> эквивалентно <typename T>
T& MyArray<T>::operator[](size_t index) {
  return array[index];
}
```

`main.cpp`:

```
#include "MyArray.h"
int main() {
  MyArray<int> a;
  MyArray<double> b;
  MyArray<MyArray<int> > arr; // работает всегда
  MyArray<MyArray<int>> arr2;  // только в 11 стандарте
}
```

5 строчка отличается от 6 наличием пробела между закрывающими угловыми скобками. компиляторы с++ до 11 стандарта путают ">>" с оператором сдвига, поэтому лучше всегда ставить

пробел.
T – тип шаблонных переменных класса, его принято обозначать именно так, но можно как угодно. Запись `template <typename T>` эквивалентна `template <class T>`.
Шаблон гораздо умнее препроцессорной подстановки: он разбирает то, что ему передаётся во время компиляции, поэтому проблем C-подхода не возникнет. Никаких multiple definition, все счастливы.
По сути происходит вот что: компилятор создает отдельный класс для каждого типа, который мы в программе поставляем в шаблон, получается такой "compile-time полиморфизм".
Важно, что здесь неприменима идея раздельной компиляции, так как неизвестно заранее, какой тип захочет пользователь. Поэтому реализацию не выносят в отдельный .cpp файл. Таким образом время компиляции и объём файлов увеличиваются.
Если взять стандартную библиотеку языка c++, то не будет .so (.dll) файлов (в отличие от библиотек языка си). Это потому что в STL всё написано с шаблонами и не компилируется заранее.

1.2. Классы с несколькими шаблонными параметрами

Класс может иметь несколько шаблонных параметров.

```
template <typename K, typename V>
class TreeItem {
  K key;
  V value;
}
```

Все методы в шаблонных классах по умолчанию `inline` независимо от того, где находится реализация методов. Без inline можно получить multiple definition.

1.3. Шаблонные функции

По традиции реализацию функций и методов класса всё же выносят в отдельный .h файл. Но еще раз напомню, что раздельная компиляция здесь неприменима.

`swap.h`:

```
template <typename T>
void swap(T& a, T& b);
#include "swap_ind.h"
```

`swap_ind.h`:

```
template <typename T>
void swap(T& a, T& b) {
  T t(a);
  a = b;
  b = t;
}
```

Если можно однозначно вывести типы аргументов, то тип для функции можно не указывать. Для классов нет автоматического выведения типов, а для функций порой это может не сработать.

```
int main() {
  int a = 2;
  int b = 3;
  swap<int> (a, b);
```

```
5    swap(a, b) // swap<int> выведется автоматически;
6  }
```

Примеры, когда компилятор не сможет вывести шаблонный параметр:

```
1  template <typename T>
2  class MyArray {
3  private:
4    size_t size;
5    T *array;
6  public:
7    MyArray(size_t s) {
8      array = new T[size];
9    }
10 };
11
12 MyArray arr(10); // Ошибка. Неизвестен тип хранимых объектов.
13 MyArray<MyArray<int> > arr(10); // Ошибка. Для внутреннего объекта отсутствует
14                                 // конструктор по умолчанию.
```

В целом с классами, как написано выше, не работает автоопределение типа
Ещё пример:

```
1  template <typename T>
2  bool less(const T& a, const T& b) {
3    return a < b;
4  }
5
6  int a, b;
7  double c;
8  less(a, b); // OK. Компилятор выведет тип int.
9  less(a, c); // Ошибка. Не понятно int или double.
```

Рассмотрим пример функции с несколькими шаблонными типами.

```
1  template <typename T, typename V>
2  void copy(MyArray<T>& a, MyArray<V>& b) { }
3
4  MyArray<int> a;
5  MyArray<double> b;
6  copy(a, b); // Типы однозначно выводятся.
7  copy<int, int>(a, b); // error: не найдёт по шаблону нужную для параметров функцию.
```

Характерный пример шаблонной функции – функция сортировки. Рассмотрим три реализации.

1) Процедурное программирование (язык си).

```
1  #include <algorithm>
2
3  int compare(const void *a, const void* b) {
4    int* aa = (int*)a;
5    int* bb = (int*)  b;
6    return *aa < *bb;
7  }
8
9  typedef int (*func_ptr)(const void *a, const void *b);
10 void sort(void *array, size_t n, size_t elem_size, func_ptr cmp) {
11 //тут должна быть сортировка
12 }
13
14 int main() {
15   int a[] = {1, 3, 2};
16   sort(a, 3, sizeof(int), compare);
```

```
17    return 0;
18 }
```

2) ООП.

```
1 class comparable { //базовый класс для всех сравниваемых типов
2   virtual int compare(const comparable *o) const = 0;
3 };
4 void sort(comparable **array, size_t n);
```

Чтобы объекты можно было отсортировать, они должны наследоваться от comparable и у них должна быть реализована виртуальная функция `compare()`.
По сути мы сортируем массив указателей, чтобы внутри сортировки независимо от типа можно было всегда смещаться на одно и то же число байтов. Отсюда берется двойной указатель (`comparable **array`).
Недостатки ООП подхода:

- Виртуальные функции. Динамическое связывание работает дольше.
- Нельзя на халяву отсортировать инты. Приходится городить новый класс.

3) generic programming (шаблоны):

```
1 template <typename T>
2 void sort(T *array, size_t n);
3
4 GaussNumber a[100];
5 sort(a, 100);//Шаблонный тип выведется автоматически
```

Здесь нет проблем со смещением, потому что код компилируется с уже известными типами. Каждый класс должен иметь оператор <.

Недостатки шаблонного подхода:

- увеличение времени компиляции и работы.
- увеличение размера исполняемого файла (компилятор генерирует функции и классы для каждого типа).
- нельзя заранее скомпилировать.

1.4. Другие виды параметров шаблона

В качестве параметра может быть не тип, а, например, число. Допустим я не хочу выделять массив в куче, поэтому я выделяю массив на стадии компиляции. Но на стадии компиляции размер массива должен быть фиксирован. Пример – битовое множество (Bitset).

```
1  template <size_t Size>
2  class Bitset{
3  private:
4    char m[((int)Size - 1) / 8 + 1];
5  public:
6    bool get(size_t index) { }
7  };
8
9  Bitset <128> b1;
10 Bitset <7> b2;
```

Когда ещё применяется такая штука? В языке си надо было помнить и передавать размер массива (например для функции сортировки). Чтобы этого избежать, можно создать обёртку над массивом с помощью шаблонов. **Нельзя предать float или не const в шаблоны**

1.5. Неинстанциированный шаблон

Можно передавать в качестве параметров шаблона другие шаблоны
Можно задавать, например, на чем реализовать стек – на векторе или на списке, передав ему в качестве шаблона имя соответствующего класса.

```
template <typename T, class Container>
class Stack{
private:
  Container c;
public:
  void push(const T& v) {}
};
```

При этом могут возникать проблемы.

```
Stack <int, List<int> > s1; //OK
Stack <double, Vector<int> > s2; //Будет работать, но будет потеря точности
```

Во втором случае мы якобы храним даблы, но на самом деле числа будут складываться в интовый вектор и обрезаться.
Чтобы этого избежать, можно не задавать тип вектора при объявлении, а сделать это внутри класса.

```
template <typename T, template <typename> class Container>
class Stack {
private:
  Container<T> c;
};
Stack <int, List> s1 ; // List<int>
Stack <double, Vector> s2 ; // Vector<double>
```

1.6. Значение по умолчанию

Можно задавать значения шаблона по умолчанию.

```
template <typename T, template <typename> class Container = Deque>
class Stack { ... };
Stack <int> s1; // Deque<int>
```

Пример кода, работающего с STL контейнерами:

```
template<class T,
                 template<class, class> class Container = std::vector,
                 template<class> class Alloc = std::allocator>
class Stack {
        Container<T, Alloc<T>> data;
};

int main() {
        Stack<int, std::vector> a;//std::vector указан для наглядности
}
```

1.7. Специализация шаблона

Допустим хотим обёртку над массивом любого типа. Но для хранения `bool` можно сделать более оптимальную по памяти реализацию. В этом нам поможет специализация.
В следующем примере массив динамический, поэтому размер в качестве шаблонного параметра не передаётся.
Код для общего случая:

```
template <typename T>
class Array {
private:
  T* a;
...
public:
  Array(size_t size) {
    a = new T[size];
  }
};
```

Код специально для `bool`:

```
template <>
class Array <bool> {
private:
  char* a ;
...
public:
  Array(size_t size) {
    a = new char[(size - 1) / 8 + 1];
  }
};
```

Обратите внимание, что в специальной реализации надо заново писать все методы (даже если некоторые их них совпадают с методами для общего случая).
Желательно чтобы сигнатуры методов совпадали в специализации с общим случаем, иначе получается очень неочевидный код, когда при разных параметрах шаблона функции вызываются по-разному и возвращают разные вещи, с этим почти невозможно работать.

Не все специализации хороши, `vector<bool>` – это не самая удачная специализация. Она работает медленнее, а колличество багов на квадратный сантиметр кода увеличивается из-за `Bit_reference` в первую очередь.

1.8. Частичная специализация

В специализации можно оставить часть параметров шаблонными.
Например, хотим, чтобы массив массивов хранился одним последовательным куском, а не массивом объектов типа `Array`:

```
template<class T>
class Array1 {
  T* a;
  ...
};
template <class T>
class Array1 <Array1<T>> {
  T** a;
};
```

Или мы хотим реализовать специализацию bool у массива с шаблонным размером.

```
template <class T, size_t N>
class Array2 {
  ...
};
template <size_t N>
class Array2<bool, N> {
  ...
};
```

1.9. Хитрый пример на понимание

Маленькая задачка для вас от Лизы.
Что выведет данная программа?

```
template<class T>
void f(T x) {              // шаблонная функция
  cout << "1" << endl;
}

template<>
void f<>(int* x) {         // специализация
  cout << "2" << endl;
}

template<class T>
void f(T* x) {             // перегрузка предыдущей функции
  cout << "3" << endl;
}

int main() {
  int *p;
  f(p);                   // вызов функции
  return 0;
}
```

Правильный ответ: выведется "3", потому что перегрузка функций работает только для базовых шаблонов, а не для специализаций. При вызове выбирается наиболее подходящий вариант из неспециализированных, а потом его лучшая специализация.
Теперь поменяем местами вторую и третью реализации.

```
template<class T>
void f(T x) {              // шаблонная функция
  cout << "1" << endl;
}
template<class T>
void f(T* x) {             // перегрузка
  cout << "3" << endl;
}
template<>
void f<>(int* x) {         // специализация
  cout << "2" << endl;
}
int main() {
  int *p;
  f(p);                   // вызов функции
  return 0;
}
```

Казалось бы, результат должен остаться прежним, ведь табор функций тот же. Тем не менее, такая программа выведет "2". Дело в том, что теперь третья функция – это специализация второй, а вторая – перегрузка первой. Сначала из первых двух перегрузок выберется вторая, а потом её специализация.

1.10. Экзотические случаи шаблонов

На досуге можете почитать про использование

- указателя в качестве шаблонной переменной,
- глобальной переменной в качестве шаблонной переменной.

2. Исключения

2.1. Виды ошибок

Виды ошибок:

1. Ошибки по вине программиста. Примеры:

```
char *s = NULL;
size_t l = strlen(s);
Array a(-1);
```

Что с ними делать:

- Лучше выявить на стадии тестирования (`assert`, `unit test`, etc).
- При выполнении идеальной программы их не происходит.
- Библиотека C подобные ошибки не обрабатывает.
- Библиотека C++ – по-разному в разных местах: `vector.at(i)` делает, а `vector[i]` не делает.
- Обрабатывать или нет – на усмотрение программиста.

2. Ошибки по вине окружения программы. Примеры:

- Файл не существует.
- Сервер разорвал сетевое соединение.
- Пользователь вместо числа ввел букву.

Что с ними делать:

- Могут произойти и при выполнении идеальной программы.
- Обязательно надо обрабатывать!

2.2. Обработка ошибок

Как же обрабатывать ошибки?

1. Проверить наличие ошибки в потенциально опасных местах (`if`).

2. Освободить ресурсы.

```
delete [] array;
fclose(file);
```

3. Сообщить пользователю и/или вызывающей функции.

```
FILE* f = fopen("a.txt", "r");
if (f == NULL) {
  printf("File a.txt not found\n");
} //или
if (f == NULL) {
  return -1;
}
```

4. Предпринять действия по устранению ошибки (например, не смогли соединиться – попробовали ещё пять раз).

Но как же сообщать пользователю об ошибках?
Рассмотрим наивную реализацию.

```
class GUI_VIew {
  load_config() {
    f = fopen();
    if (f == NULL)
      printf(”Error:...”);
      // или
      fprintf(stderr, ”Error:...”);
  }
}
```

Здесь есть проблема – функция `load_config` взяла в себя слишком много, ибо неизвестно, с каким интерфейсом она работает. Она может быть вызвана в консоли, может в браузере и т.д. Также возможно, если файл не открылся - не возможно продолжать работу и что либо напечатать.
Как сделать более грамотно?

2.3. Обработка ошибок в C style

2.3.1. Через возвращаемое значение

В случае ошибки функция может вернуть значение, зарезервированное под тип данной ошибки.

```
1  int load_config() {
2    f = fopen();
3    if (f == NULL)
4      return −1;
5  }
6
7  r = load_config();
8  if (r == −1) {
9    //обработка ошибки
10 }
```

Но таким способом мы узнаем слишком мало информации об ошибке.

2.3.2. Через глобальную переменную

Функция может записывать в глобальную переменную информацию о произошедшей ошибке.

```
1  #include <errno.h>
2  FILE∗ fopen(...) {
3    if (file not found) {
4      errno = 666;
5      return NULL;
6    }
7    if (permission denied) {
8      errno = 777;
9      return NULL;
10   }
11 }
```

```
12
13 file = fopen(''f.txt'');
14 if (errno == 666) {
15     //обработка ошибки file not found
16 }
17 else if (errno == 777) {
18     //обработка ошибки permission denied
19 }
```

2.4. Недостатки C style

- Не всегда хватает диапазона возвращаемых значений.
  Например, функция `strtol(''a'')` вернет ноль, как и `strtol(''0'')`.

- Код логики и обработка ошибок перемешаны.

```
r = fread(...);
if (r < ...) {
    //error
}
r = fseek(...);
if ( r != 0 ) {
    //error
}
```

2.5. C++ style, исключения

Суть такая – в любой непонятной ситуации бросай исключение и выходи из функции. Более формально. Если внутри функции произошла ситуация, в которой функция не знает, что ей делать, она бросает "исключение" – объект особого класса. При этом её выполнение прекращается, а исключение передаётся функции, которая её вызвала. Пример:

```
1  class MyException {
2  private:
3      char message[256];
4      // possible fields: filename, line, function name
5  public:
6      const char* get() {
7          return message;
8      }
9  };
10 double divide(int a, int b) {
11     if (b == 0) {
12         throw MyException(''Division by zero'');
13     }
14     return a / b;
15 }
```

Функция `divide` не знает, как делить на ноль. Когда её просят это сделать, она кидает исключение и со словами "ой всё!" прекращает выполняться. Как же поймать исключение?

```
1 try {
2     x = divide(c, d);
3 }
4 catch(MyException& e) {
5     std::cout << e.get();
6     //обработка ошибки
7 }
```

Если внутри блока `try` функция `divide` бросила исключение, оно сразу же ловится, ссылка на него попадает в переменную `e` и начинается выполнение блока `catch`. Если исключение не было брошено, блок `catch` выполняться не будет.
Бывает такое, что функция, поймавшая исключение может сделать только часть работы по обработке ошибки. В таком случае эта функция может бросить его дальше:

```
try {
  x = divide(c, d);
}
catch(MyException& e) {
  std::cout << e.get();
  throw e;
}
```

Мы решили проблему Си стиля, теперь есть отдельный механизм для возврата значений, а есть механизм для ошибок.

2.6. Stack unwinding

Пусть мы из функции `main` вызвали функцию `g`, а из неё – функцию `f`. Функция `f` бросила исключение. Проследим за его полётом.
Во время выполнения функции `f` стек выглядел так:

f()
g()
main()

Далее происходит следующее:

1. Нормальный процесс выполнения программы заканчивается, т.е. поток управления до `printf` внутри `f` не дойдет.

2. Начинается stack unwinding: последовательный просмотр стека до тех пор, пока не будет найден подходящий по типу исключения (в нашем примере тип `MyException`) блок `try-catch`. Функции без `try-catch` убиваются автоматически.

3. Если подходящий блок не был найден, и исключение вылетело за `main()` , то программа аварийно завершается.

2.7. Типы исключений

Если в программе несколько подсистем (GUI, Network, Model), то можно у каждой подсистемы сделать свой тип исключения (GuiException, NetworkException, ModelException) и обрабатывать их по-разному.

```
main() {
  try {
    doGame();
  }
  catch(GuiException& e) {
    showMessageBox (...);
  }
  catch(NetworkException& e) {
    showMessageBox(...);
    logger.log(...);
  }
  catch(ModelException& e) {
    logger.log(...);
  }
}
```

Можно наследовать исключения друг от друга, чтобы где-то ловить исключения сразу нескольких типов, а где-то каждого типа отдельно.

```
class MyException {};
class GuiException:public MyException {};
class NetworkException:public MyException {};
class ModelException:public MyException {};
```

Но надо быть аккуратным и помнить, что исключение поймается первым блоком catch, подходящий типу.
В следующем фрагменте кода будет всегда срабатывать только первый catch:

```
try { ... }
catch(MyException& e) {...}
catch(GuiException& e) {...}
```

Чтобы всё работало правильно, надо поменять порядок:

```
try { ... }
catch(GuiException& e) {...}
catch(MyException& e) {...}
```

В STL все исключения – наследники std::exception.

2.8. Исключения любого типа

Если я не знаю, какого типа исключение может броситься внутри блока try, я могу написать catch, который поймает исключение любого типа. При этом мы не будем знать структуры пойманного объекта и не сможем из него вытянуть информацию (даже если работаем в ФСБ). Зато мы, как и раньше, можем кинуть его дальше.

```
try {
  doMainWork();
}
catch (...) {
  throw;
}
```

Здесь `catch(...)` это именно синтаксис, там должно быть многоточие. `throw` в данном случае пишется без параметра.
На самом деле кидать можно переменные и объекты любого типа (`int, char*, ...`), но я вам этого не говорил! ;)

2.9. Подводные камни исключений

При использовании исключений увеличивается выделяемое место на стеке и увеличивается время работы. Ну тут понятно, за удобства надо платить.
Однако механизм исключений таит в себе более серьезные опасности, с которыми может столкнуться неопытный программист.

2.9.1. Утечка памяти

При Stack unwinding происходят вызовы деструкторов локальных переменных, которые снимаются со стека, однако если какие-то объекты внутри функции были выделены динамически, до их удаления может и не дойти. Тогда произойдет утечка ~~мозгов~~ памяти.

```
f () {
  int *buffer = new int[n];
  if ( ... ) throw MyException(...);
  delete[] buffer; // не выполнится.
}
```

2.9.2. идиома RAII

RAII – Resource Acquisition Is Initialization ("Взятие Ресурса Должно Происходить через Инициализацию", или как-то так). Взятие ресурса нужно инкапсулировать в класс, чтобы в случае исключения вызвался деструктор и освободил ресурс.

```
void f() {
  auto_ptr p(new Person("Jenya", 36, true)); // или другой умный указатель
  divide(c, e); // может бросить исключение
}
```

В коде выше, если `divide` бросит исключение, то у указателя вызовется деструктор и сделает с Женей всё что нужно.

2.9.3. Исключения в конструкторе

В следующем примере во время конструирования объекта функция divide может бросить исключение, и объект останется недостроенным.

```
class PhoneBookItem {
  PhoneBookItem(const char* audio, const char* pic) {
    af = fopen(audio, "r");
    pf = fopen(pic, "r");
    divide(c, e); // может бросить исключение
    f();
  }
  ~PhoneBookItem() {
    fclose(af);
    fclose(pf);
  }
};
```

У недостроенного объекта не будет вызывать деструктор, память останется неосвобождённой. А вот деструкторы полей класса вызовутся, если будет брошено исключение. Правильный подход - ловить исключения в конструкторе, удалять созданные ресурсы (например, если у нас два `new` подряд, то второй надо обернуть и в `catch` удалить результат первого) и пробрасывать исключение наверх. Ещё лучший подход - использовать везде `RAII`-поля, тогда не потребуется обрабатывать исключения; любое исключение в конструкторе вызовет деструкторы полей и утечек памяти не произойдёт.
В случае, когда у нас всё же идёт динамическое выделение памяти, нужно обрабатывать все исключения и высвобождать её.

```
class PhoneBookItem {
  PhoneBookItem(const char* audio, const char* pic) {
    try {
      af = fopen(audio, "r");
      pf = fopen(pic, "r");
      divide(c, e); // может бросить исключение
      f();
    }

    catch(MyException& e) {
      fclose(af);
      fclose(pf);
      throw e; // отправляем е лететь дальше
    }
  }
  ...
};
```

2.9.4. Исключения в деструкторе

Как уже говорилось, в процессе Stack unwinding вызываются деструторы. По этой причине деструкторы никогда ни за что **не должны кидать исключения!** Дело в том, что механизм исключений поддерживает не более одного одновременно летящего исключения. Как только появляется два одновременно летящих исключения, программа аварийно завершается.
Пример – В функции `f` бросилось исключение и вызвался деструктор базы данных. Этот деструктор хочет послать на сервер сообщение от том, что база закрыта. Сервер может оказаться недоступен, и бросится второе исключение, и всё упадет.

```
class PersonDatabase {
  ~PersonDatabase() {
    // бросает исключение, если сервер недоступен.
    networkLogger.log("Database is closed.");
    ...
  }
};
f() {
  PersonDatabase db;
  if(...) throw MyException("Error: disk is full.");
}
```

Чтобы этого не произошло:

```
PersonDatabase::~PersonDatabase() {
    try {
      networkLogger.log("Database is closed.");
    }
    catch(...) { } // поймать всё
  }
```

2.10. Гарантии при работе с исключениями

Гарантии – это своего рода документация для программиста, который работает с нашим методом/функцией. Мы гарантируем, что в случае ошибки функция обязана будет вести себя каким-то более-менее вменяемым образом.
Гарантии бывают трёх видов, разберемся с ними подробнее.

2.10.1. Basic guarantee

Базовая гарантия обещает, что при возникновении исключительной ситуации не будет утечек памяти. Также гарантируется, что все объекты в каком-то корректном состоянии, и их инварианты сохранены. Без этой гарантии у нас может получиться так, что при вылете `exception` нарушается какой-то внутренний инвариант, после которого случается `undefined behavior`. Так себе гарантия, если инварианты не соблюдаются.
При этом, если в ходе выполнения функции какие-то объекты поменяли свои значения, они, скорее всего, не вернутся в исходное состояние.
Например, здесь в цикле for может броситься исключение, у указателя на Person вызовется деструктор и утечки не произойдет. Однако какие-то элементы массива, которые успешно поменяли свои значения и вывелись в `stdout`, так и останутся с новыми значениями, и из потока вывода их будет уже не вынуть, тогда как остальные будут со старыми значениями и не выведутся.

```cpp
class PersonDatabase {
  MyVector<Person> array;
  void process() {
    auto_ptr <Person> p(new Person(...));
    for (int i = 0; i < array.length(); i++) {
      int a = divide(rand(), rand()); // может бросить исключение
      array[i]->setAge(a);
      std::cout << p << endl;
    }
  }
};
```

Функции, в которых применяется RAII, обеспечивают как минимум basic guarantee.

2.10.2. Strong guarantee

Гарантирует не только отсутствие утечек памяти, но и то, что в случае ошибки все переменные сохранят свои значения, которые они имели до начала выполнения функции.
Самый частый пример – это банковский перевод. Допустим, с одного счёта списалась круглая сумма, а во время начисления на второй счёт произошла ошибка. Вся сумма обязана вернуться на первый счёт.
Сильной гарантии можно добиться, например, используя идиому copy-and-swap. Когда мы хотим поменять значение объекта, мы создаём его копию, применяем к ней необходимые изменения и потом свопаем с нашим объектом (или просто присваиваем), если работа с копией прошла успешно.

```
class PersonDatabase {
  MyVector <Person> array;
  void process() {
    auto_ptr<Person> p (new Person (...));
    MyVector<Person> copy (array);
    for (int i = 0; i < array.length(); i++) {
      int a = divide(rand(), rand());
      copy[i] -> setAge(a);
    }
    array = copy;
  }
};
```

2.10.3. No throw

Такие функции гарантируют, что они не будут бросать исключений ни при каких обстоятельствах. Если такая функция вызовет другую, которая бросает исключения, то она должна его поймать и обработать внутри себя.

```
void f () {
  try {
    divide(a , b );
  }
  catch(...){ // ловим всё
  }
}
```

При написании кода рекомендуется по возможности стремиться к Strong guarantee. Однако это не всегда возможно, но хотя бы базовая гарантия должна быть.
Исключения – это довольно удобный способ отлавливать ошибки по вине окружения, однако не все их любят и признают. Например в Google ими пользоваться запрещено, однако это больше связано с тем, что так исторически сложилось, не захотелось переписывать весь код и требовать везде гарантий, а пользоваться исключениями без гарантий – это сплошные ошибки, которые невозможно нормально дебажить.

3. Стандартная библиотека шаблонов

В этом разделе мы немного рассмотрим основные аспекты стандартной библиотеки шаблонов (STL) языка C++ стандарта 2001 года. Стандарт C++11 будет рассмотрен отдельно позже.

3.1. Контейнеры

Подробнее про разные контейнеры можно почитать здесь и здесь.
Контейнеры бывают:

1. Последовательные
    - `array` (C++11) – обёртка над статическим массивом.
    - `vector` – динамический саморасширяющийся последовательный массив.
    - `deque` – двусторонний массив (можно добавлять в начало и конец). Поддерживает Random Access (быстрый доступ к любому элементу).
    - `forward_list` (C++11) – односвязный список.
    - `list` – двухсвязный список.
2. Ассоциативные
    - `set` – множество, хранит упорядоченные уникальные элементы в сбалансированном дереве поиска.
    - `map` – множество пар <ключ, значение>, упорядоченных по ключу. Ключи уникальны.
    - `multiset` – упорядоченное мультимножество.
    - `multimap` – упорядоченное мультимножество пар.
3. Неупорядоченные ассоциативные
    - `unordered_set` (C++11) – хеш таблица уникальных элементов.
    - `unordered_map` (C++11) – хеш таблица пар с уникальными ключами.
    - `unordered_multiset` (C++11) – хеш таблица повторяющихся элементов.
    - `unordered_multimap` (C++11) – хеш таблица пар с повторяющимися ключами.
4. Адаптеры последовательных контейнеров
    - `stack` – стек.
    - `queue` – очередь.
    - `priority_queue` – очередь с приоритетами (бинарная куча).

Адаптеры – это обёртки над последовательными контейнерами (их можно построить на любом типе послед. контейнера, подсунув этот тип в качестве шаблонного параметра).

3.2. Итераторы

Мы хотим добиться полиморфизма работы с контейнерами. Хотим уметь перебирать/добавлять/удалять элементы контейнера любого типа, не задумываясь о том, как они хранятся и в каком порядке их перебирать. Для этого есть итераторы. Это что-то вроде указателей, только с более сложной структурой. Они поддерживают операции ++, *(разыменование) и ->.
Таким образом любая последовательность задаётся двумя итераторами – начало и конец (конец не включается в последовательность). Рассмотрим пример.

```
vector<int> arr;
set<int> st;

for (vector<int>::iterator it = arr.begin(); it != arr.end(); ++it) {...}
for (set<int>::iterator it = st.begin(); it != st.end(); ++it) {...}
```

Для итераторов принято использовать префиксный оператор ++, потому что он работает быстрее постфиксного в силу того что не сохраняет промежуточное значение.
А вот как оператор ++ может быть реализован у разных типов итераторов.
Для вектора:

```
template <class T>
class vector {
  T* array;
  class iterator {
    T *pos;
    iterator& operator++() {
      pos++;
      return *this;
    }
  }
}
```

Для списка:

```
template <class T>
class list {
  Node* head;
  class iterator {
    Node pos;
    iterator& operator++() {
      pos = pos->next;
      return *this;
    }
  }
}
```

3.3. Инвалидация итераторов

Итераторы внутри себя хранят указатель или ссылку на элемент контейнера. Иногда при работе с контейнерами может очищаться/перевыделяться память, элементы могут переезжать с места на место.
Невалидный итератор – тот, который указывает на элемент, который был перемещен/удалён. Самый простой пример инвалидации – когда создают итератор на элемент, а потом элемент удаляют. После этого итератор указывает непонятно куда. Более интересные случае рассмотрим на примере вектора.

`push_back()` Вектор периодически расширяется и перевыделяет память, после этого все итераторы инвалидируются, так как все элементы переезжают на новые места.

`insert()` При добавлении элемента в середину вектора все элементы справа от добавленного сдвигаются на одну ячейку памяти вправо, итераторы на них становятся неправильными.

`erase()` То же, что и при `insert()`, только элементы сдвигаются влево.

Обратите внимание, что в списке никакая из этих трёх проблем не встречается, так как добавление/удаление одних элементов никак не влияет на положение других.
Можно продолжить познавать итераторы здесь: subsection 3.8.

3.4. Исключения в STL

В STL есть свои классы для исключений.
Базовый:

```
1 std::exception {
2   const char* what(); // сообщение об ошибке
3 }
```

И унаследованные:

- `logic_error`:
  - `invalid_argument`;
  - `out_of_range` (`vector::at(-1)`)
- `runtime_error`:
  - `bad_alloc` (`new int[10000000000]`)
- `ios_base::failure`.

Мы можем унаследовать свой класс исключений от стандартного. Для этого он обязательно должен иметь конструктор, принимающий строчку с сообщением об ошибке.

```
1 matrix_exception : public std::logic_error {
2   matrix_exception(char* s) : logic_error(s) {}
3 };
```

При работе с потоками ввода/вывода можно настраивать, в каких случаях они будут кидать исключения. Подробнее об этом здесь.

3.5. Несколько лирических отступлений

3.5.1. Почему обязательно закрывать файл после использования

С файлами, открытыми для записи всё более-менее понятно. Если мы открыли файл, а потом его захотела открыть другая программа, то у неё или у нас будут проблемы. Файл либо второй раз не откроется, либо откроется и перезапишется. Поэтому открытый для записи файл лучше как можно быстрее закрыть.
Почему же так важно закрывать файл для чтения? На это есть ряд причин.

1. Когда мы открываем файл, в программе выделяется память под структуру для этого файла и буфер. Для экономии памяти её лучше освободить.

2. В различных операционных системах есть ограничения на количество одновременно открытых файлов.

3. Как и в случае с файлом для записи, если мы открываем файл для чтения, доступ к нему других приложений ограничивается (в винде, например, его нельзя удалить).

4. Лучше закрывать всё всегда и везде, чтобы выработалась привычка. Если задумываться "закрыть, или не закрыть? Вот в чём вопрос", то можно забыть закрыть что-то действительно важное.

3.5.2. Временные объекты

В `C++` есть анонимные переменные — это переменные без имени.
Пусть у нас есть класс `MyArray` и функция, которая принимает объект этого класса. Мы можем написать это так:

```
void f(MyVector& v) {...}

MyVector arr(10);
f(arr);
```

А что если мы не хотим отдельно заводить переменную, потом хранить её? Мы можем передать в функцию сразу то, что вернёт конструктор.

```
void f(MyVector v) {...}
//или
void f(const MyVector& v) {...}
//но не
void f(MyVector& v) {...}


f(MyVector(10));
```

Обратите внимание, что теперь функция не может принимать неконстантную ссылку на объект, потому что временный объект является `rvalue` (он может находиться только справа от знака равенства). Можно либо константную ссылку, либо копию.
Какие бывают применения в реальной жизни? Пусть мы в `vector<int> arr` сделали много раз `push_back()`, а потом много раз `pop_back()`. При этом `capacity` осталась большой. В C++11 появился метод `shrink_to_fit()`, который сжимает память до достаточного объёма. В более ранних стандартах это можно было сделать так:

```
vector<int>(arr).swap(arr);
```

Мы создали временный объект – копию `arr` (при этом `capacity` уменьшилась) и вызвали у него метод, который свопает все поля с полями `arr`. После этого у временного объекта вызовется деструктор, а `arr` продолжит существовать с уменьшенным объемом.

3.5.3. Операторы сравнения

Мы хотим уметь сравнивать объекты любых классов. Но вот незадача, нам подсунули объект, у которого определен только оператор <. На самом деле нам этого достаточно чтобы выразить все остальные.
Итак, мы точно знаем, что у класса `T` определен оператор <.

```
template<class T>
bool operator > (T& a, T& b) {
  return b < a;
}
template<class T>
bool operator >= (T& a, T& b) {
  return !(a < b);
}
template<class T>
bool operator <= (T& a, T& b) {
  return b >= a;
}
template<class T>
bool operator == (T& a, T& b) {
  return (a <= b) && (b <= a);
}
template<class T>
bool operator != (T& a, T& b) {
  return !(a == b);
}
```

Можно было выражать и в другом порядке. В качестве упражнения можете выразить каждый оператор только через <.

3.6. Функторы

Допустим у класса Person оператор < сравнивает объекты по именам, а мы хотим по возрасту. Мы не можем переопределить оператор < для `Person`, потому что он уже есть. Можно создать функтор.

```
class by_age {
  bool operator()(const Person &p1, const Person &p2) const {
    return p1.age < p2.age;
  }
};

set<Person, by_age>;
```

Функтор – это класс, для которого перегружен оператор (). Объект такого класса чем-то похож на функцию, но с некоторыми фичами. Например, он может хранить внутри себя какую-то информацию, а ещё его можно передать куда-нибудь в качестве параметра, например попросить, чтобы `set` сравнивал Person нашим функтором.
Функтор, возвращающий `bool`, принято называть предикатом.

Ещё пример.

```
struct accum {
  int acc;
  accum() : acc(0) { }
  void operator()(int a) {
    acc += a;
  }
};
accum f;
f(13);
f(16);
cout << f.acc; // 29
```

f аккумулирует (суммирует) в себе все значения, которые мы ему передаём.

3.7. Алгоритмы

В STL реализовано более 100 алгоритмов. Здесь мы разберем только некоторые из них. Больше информации о библиотеке алгоритмов языка C++ ищите здесь и здесь.

3.7.1. Микро-алгоритмы

Рассмотрим некоторые популярные:

1. `swap(T &a, T &b)`
2. `iter_swap(It p, It q)` – меняет местами значения элементов, на которые указывают итераторы.
3. `max(const T &a, const T &b)` и `min(const T &a, const T &b)`
   У них может быть по три параметра - первые два такие же, а третий - компоратор.
4. `count(It p, It q, const T &x)` – возвращает, сколько раз элемент со значением `x` входит в последовательность, заданную итераторами `p` и `q`.
5. `count_if(It p, It q, Pr pred)` – вовзращает, сколько элементов подходят под предикат, т.е. на них возвращается `true`.
6. `find(It p, It q, const T &x)` – возвращает итератор на первое вхождение `x`.
7. `find_if(It p, It q, Pr pred)` – аналогично возвращает итератор при истинности предиката `pred`.
8. `min_element(It p, It q) max_element(It p, It q)`.
9. `equal(It p, It q, Itr i)` – сравнивает на эквивалентность. от p до q и от i до i + q - p.
10. `pair<It, Itr> mismatch(It p, It q, Itr i)` – возвращает пару итераторов, указывающую на первое несовпадение.
11. `F for_each(It p, It q, F func)` – для каждого элемента последовательности применяет функтор `func` и вовзращает его после всех применений.
12. `fill(It p, It q, const T &x)` – заполняет последовательность элементом `x`.

13. `generate(It p, It q, F gen)` – заполняет последовательность тем, что сгенерировал функтор `gen`.

14. `copy(It p, It q, Itr out)` – копирует в `out`.

15. `reverse(It p, It q)` – переворачивает.

16. `sort(It p, It q)` – алгоритм Intro sort, есть вариант с третим параметром – компаратором.

17. `transform(It p, It q, Itr out, F func)` – к последовательности от `p` до `q` применяет `func` и записывает результат в последовательность `out`.

3.7.2. Реализация алгоритмов

```
template<class InputIt, class OutputIt, class UnaryPredicate>
OutputIt copy_if(InputIt first, InputIt last,
                 OutputIt d_first, UnaryPredicate pred) {
  while (first != last) {
    if (pred(*first))
      *d_first++ = *first;
    first++;
  }
  return d_first;
}
```

Как достать тип объекта из итератора? У каждого итератора есть `value_type`, можно достать его. Но есть проблема, когда нам передают не итератор, а обычный указатель, у него нет этого поля. Для этого есть `iterator_traits`, у которого уже можно взять тип, будто мы берём его от итератора, независимо от того, что мы в него передали.

```
template <class It, class P>
void foo(It first, It last, P p) {
    while (first != last && !p(*first)) ++first;
    if (first == last)
        return;
    typename std::iterator_traits<It>::value_type val = *first;

    while (first != last) {
        if (p(*first))
            *first = val;
        ++first;
    }
}
```

Эта функция находит первый элемент, который удовлетворяет предикату, сохраняет его значение, а потом записывает во все следующие элементы, удовлетворяющие предикату, это значение.

3.7.3. erase-remove idiom

Часто бывает такое, что нужно удалить несколько объектов из последовательности по какому-то принципу (например все нечётные числа). Это можно делать вручную, а можно в одну строчку при помощи алгоритмов.
Но есть проблема: алгоритмы ничего не знают про внутреннее устройство контейнера, с которым работают. Поэтому они не могут изменять его размер, а могут лишь менять порядок элементов.
Тем не менее, существует способ быстро и эффективно избавиться от ненужных элементов, и

называется он "erase-remove idiom".
Его суть заключается в том, что сначала какой-то алгоритм (например `std::remove()`) сдвигает все удаляемые элементы в конец контейнера и возвращает итератор на первый такой элемент. А потом специальный метод, принадлежащий нашему контейнеру (например `vector::erase()`), "отрезает" их, изменив размер контейнера.
Пример:

```
bool is_odd(int i) { // функция, проверяющая, что число нечетно
  return (i % 2) != 0;
}

int main() {
  // вектор содержит все числа от 0 до 9.
  std::vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  // удалили двойку:
  v.erase(std::remove(v.begin(), v.end(), 2), v.end());
  // удалили все нечетные числа:
  v.erase(std::remove_if(v.begin(), v.end(), is_odd), v.end());
  // теперь v == 0, 4, 6, 8
  return 0;
}
```

Своё название идиома получила как раз от функций `erase()` и `remove()`, которые в совокупности дают нужный результат.

3.8. Подробнее про итераторы

3.8.1. iterator_category

У итераторов есть разные категории, определяющие их возможности и гарантии. Рассмотрим самые распространённые из них. Тут есть более подробное описание.

1. `Forward iterator(Fwd)` – поддерживает только ++ и разыменование. Позволяет чтение и запись.

2. `Bidirectional iterator (BiDi)` – поддерживает ещё и −−.

3. `Random access iterator (RA)` – поддерживает сдвиг на произвольны шаг, т.е. арифметические операции типа + =.

Вектор и дек в `STL` реализованы на `RA` итераторах, остальне контейнеры на `BiDi`. Как понять, какой итератор лежит в классе? Какого типа значение он под собой прячет? В нём находится что-то такое:

```
template <class T>
class vector {
  T *array;
public:
  class iterator {
    typedef ra_teg iterator_category;
    typedef T value_type;
  }
}
typename vector<int>::iterator::value_type a;
```

Чтобы получить эти поля не только из итераторов, но и из указателей, существует `iterator_traits`.

Пример использования:

```
//template<It>
// It it
typename std::iterator_traits<It>::value_type val = *it;
```

3.8.2. distance и advance

Есть две шаблонные функции, которые помогают в работе с итераторами – `distance` и `advance`.
Первая `std::distance(it1, it2)` возвращает расстоение между итераторами, а `std::advance(it, n)` делает для итератора +=n. Использует += для `RA` и ++ для `BiDi`.
Рассмотрим их реализацию:

```
template <class Iterator>
Iterator advance(Iterator it, int amount) {
  typedef template Iterator::iterator_category tag;
  advance(it, amount, tag());
}
template <class RAIterator>
RAIterator advance(RAIterator it, int amount, ra_tag t) {
  return it + amount;
}
template <class BiDiIterator>
BiDiIterator advance(BiDiIterator it, int amoint, bidi_tag t) {
  for (; amount; --amount) ++it;
  return it;
}
```

3.9. Ввод/Вывод

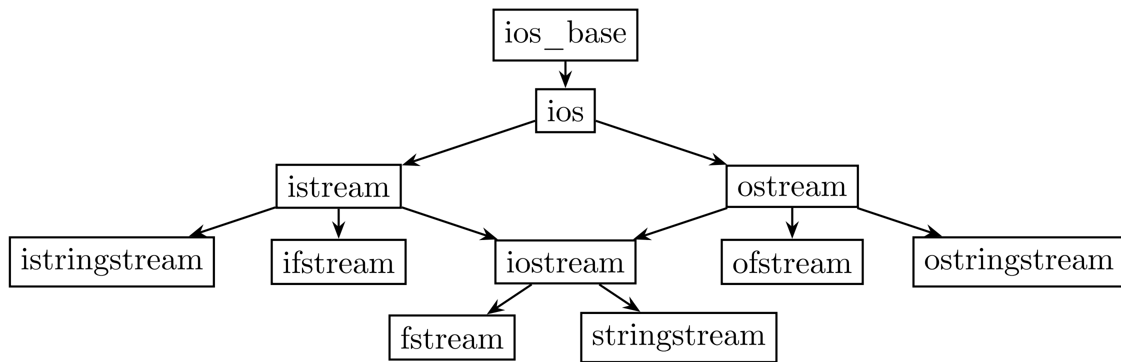
В языке C для чтения из файла и из консоли использовались разные функции.

```
FILE* f = fopen("infile.txt", "r");
int rf;
fscanf(f, "%d", &rf);
int rc;
scanf("%d", &rc);
```

В C++ мы хотим добавить уровень абстракции, чтобы ввод-вывод для всего был одинаковый. Эта абстракция – потоки (не thread, а stream!). Самый простой пример потока ввода-вывода – консольные `std::cin` и `std::cout`.

3.9.1. Иерархия классов

Все потоки - наследники класса `ios` (он сам наследуется от `ios_base`), от `ios` наследуются `istream` и `ostream` - потоки для ввода и вывода соответственно.
От `istream` наследуются `ifstream` и `istringstream` - потоки для чтения из файлов и строк.
От `ostream`, соответственно, `ofstream` и `ostringstream` - для записи в файл/строку.
Также есть `iostream`, который наследуется от `istream` и от `ostream`, он поддерживает как чтение, так и запись. Его наследники - `fstream` и `stringstream`, они делают, не поверите, чтение и запись в файл или строку. Внутри практически не отличается от родительских классов, но рекомендуется использовать `istream` или `ostream`, если нужно либо только читать, либо только писать - так меньше возможностей выстрелить себе в ногу.



3.9.2. Методы для чтения/записи в консоль

Самый простой пример потока ввода-вывода - консольные `std::cin` и `std::cout`.

```
int r1, r2;
std::cin >> r1 >> r2;
std::cout << "We have just read " << r1 << "and " << r2 << std::endl;
std::cerr << "Some console log output\n";
```

`cin` - объект класса `istream`. Он считывает из стандартного потока ввода (по умолчанию это консоль) значения нужного нам типа. Можно считывать несколько значений сразу, разделяя их ''»''.
`cout`, объект класса `ostream`, осуществляет запись в стандартный поток ввода. Можно выводить несколько объектов, разделяя их ''«''.
Вывод буферизованный, то есть записывается сначала в буфер, а потом уже куда надо. Если хотим насильно опустошить буфер, можем вызвать `cout.flush()`, можем вывести `std::endl` (это перевод строки с очисткой буфера). А поток `std::cerr` не буферизуется.

3.9.3. Работа с файлами

Чтобы работать с файлами, сначала нам надо открыть их на чтение или запись (или на оба). Мы делаем это, когда вызываем конструктор потока, либо командой `open()`. Закрываем файл командой `close()`.
Давайте считаем число из файла "input1.txt", ещё одно число из файла "input2.txt" и выведем результат в файл "output.txt":

```
int a, b;
std::ifstream fin("input1.txt");
fin >> a;
fin.close();
fin.open("input2.txt");
fin >> b;
std::ofstream fout("output.txt");
fout << a + b;
```

Файл сам закрывается в деструкторе, который вызовется, когда мы выйдем из нашего scope (функция, цикл, условие - все, что в фигурных скобочках), в котором создали поток. Не надо закрывать его руками без надобности! Хоть ничего и не сломается, это будет не по RAII.
Файл можно открывать в разных режимах:

1. `ios_base::in` - на ввод
2. `ios_base::out` - на вывод
3. `ios_base::ate` - при открытии переместить указатель в конец файла

4. `ios_base::app` - открыть файл для записи в конец файла
5. `ios_base::trunc` - удалить содержимое файла, если он существует
6. `ios_base::binary` - открыть файл в двоичном режиме

Эти флаги по сути являются битовыми масками, поэтому их можно комбинировать с помощью "|" (побитовое "или") вот так:

```
std::ifstream fin("input.bin", ios_base::in | ios_base::binary);
```

Когда мы открываем файл в двоичном режиме, для чтения нужно использовать метод `read()`, который есть у нашего потока. Ему потребуется буфер, куда он будет записывать считанные байты и количество байт, которые он будет считывать.

```
fin.read(buffer, 1);
cout << buffer[0];
```

Для записи то же самое:

```
std::ofstream fout("output.bin", ios_base::out | ios_base::binary);
fout.write(buffer, 1);
```

Забавный факт: `write()` возвращает `ostream&`, так что мы можем писать конструкции типа `fout.write(...).write(...).write(...)`. Аналогично с `read()`.
`tellg()` возвращает позицию в файле, `seekg()` перемещает на заданную позицию в файле. `eof()` возвращает, достигли ли мы конца файла или нет.

3.9.4. Манипуляторы

А еще есть манипуляторы. Их можно выводить в поток, и они будут что-то менять (подключите библиотеку iomanip). Например,

1. `std::hex` - вывод в шестнадцатеричной системе счисления
2. `std::flush` - выводит содержимое буфера
3. `std::endl` - переводит строку и делает flush
4. `std::setprecision(p)` - устанавливает число цифр после запятой

```
std::cout << std::hex << 15 << std::endl; // выведет 'f', перевод строки и сделает flush
```

3.9.5. Обработка ошибок

Потоки предоставляют нам множество возможностей выявления и обработки ошибок.
У потоков определён оператор `bool`, позволяющий, например, в форме `if (!fin)` узнать, смогли ли мы открыть файл. Точно так же `if (cin » x)` ответит про конкретную операцию считывания.

Больше деталей доступно при вызове методов:

1. `good()` - вернет `true`, если всё в порядке
2. `bad()` - если возникла фатальная ошибка
3. `fail()` - если произошла неудачная операция с потоком (например, попытались считать слово в `int`)
4. `eof()` - если достигли конца файла.

С теми же деталями мы можем ловить исключения. Для этого потоку необходимо задать, какие исключения он будет кидать (в формате битмасок).

```
1  std::ifstream fin;
2  file.exceptions(std::ifstream::failbit | std::ifstream::badbit);
3  int x;
4  try {
5    fin.open (''test.txt'');
6    fin >> x;
7  }
8  catch (std::ifstream::failure e) {
9    std::cerr << ''Exception opening/reading/closing file\n'';
10 }
```

3.9.6. Ввод-вывод пользовательских типов

Пусть у нас есть свой тип, и мы хотим, чтобы потоки могли его считывать и выводить. Для этого нам нужно будет перегрузить оператор ''»'' и/или ''«''. При этом если мы планируем выводить приватные поля класса или вызывать приватные методы, нужно будет также объявить оператор как `friend`.
Пример:

```
1  class MyClass {
2  private:
3    int value;
4  public:
5    friend ostream& operator<<(ostream& os, const MyClass& mc);
6    friend istream& operator>>(istream& is, MyClass& mc);
7  };
8
9  ostream& operator<<(ostream& os, const MyClass& mc) {
10   os << mc.value;
11   return os;
12 }
13
14 istream& operator>>(istream& is, MyClass& mc) {
15   is >> mc.value;
16   return is;
17 }
```

Заметим, что мы возвращаем тот же поток, чтобы поддерживать ввод/вывод многих объектов через несколько ''»''/''«''

3.10. Приведение типов в C++

В языке C компилятор позволял присваивать любые типы, опасные и не очень.

```
int a = 3;
char b = 0;
b = a;

const int c = 3;
int *p = &c;

double d = (double)1 / 5;// чтобы не получить инт.
```

Программа на Си успешно скомпилируется. Есть более тонкий момент с преобразованием `const`:

```
const char* s = "hello";// тут const будет уже в двоичном коде,
//если сделать так:
char *pc = s;
pc[0] = 0;// - аварийно завершится, ибо память в read only.
```

Ещё пример, когда ничего не сломается, но лучше так не делать:

```
f(const int *pi) {
  int *p = pi;
  p[0] = 1;
}

main() {
  char s[] = "hello";
  f(s);
}
```

Как итог - преобразование типов - это очень опасная вещь.
Теперь вернёмся к C++
Запретили неявные приведения типов(почти везде):

```
int *pi;
char *pc;
pc = pi; // ERROR!
pc = (char*)pi;//cool
void *pv;
pv = pi; //у void* нет адресной арифметики, любой указатель приводится неявно
// поэтому компилятор не будет ругаться
pi = (int*)pv; // без явного приведения будет плохо.
```

Также появились классы и наследники. Пусть `B` – наследник класса `A`. Функция `f` есть в обоих классах, а вот `g` появилась только в `B`.
Ситуация "кастуем ребёнка к родителю" называется upcast (и выполняется неявно).
Пример:

```
//upcast - кастуем ребёнка к родителю
B *pb = new B;
A *pa = pb; // не нужен (A*), т.к. B является и A, все поля и методы имеются
pa->g(); // ERROR: нет такого в
```

Обратная ситуаця называется downcast, для неё нужно явное преобразование.
Пример:

```
//downcast - кастуем родителя к ребёнку
B *pb = new B;
A *pa = pb;
B *pb1 = (B*)pa;
pb1->g() //OK
```

Плохой пример:

```
A *pa = new A;
B *pb1 = (B*)pa;
pb1->g(); //TRASH AND CRUSH!!
```

Для пользовательских классов порой хочется тоже деать преобразоания. Чтобы преобразовать класс к какому-то типу(например `double`, которого мы не можем изменить сам тип), нужно использовать специальный оператор, а чтобы преобразовать класс к другому классу, можно вместе этого у того класса создать конструктор от первого.

```
class fraction_array {
public:
  int array[2];

  fraction_array(int a, int b) {
    array[0] = a, array[1] = b;
  }

 int operator[](int i) const {
  return array[i];
 }
};

class fraction_pair {
public:
  int a;
  int b;
  fraction_pair(int x, int y) : a(x), b(y) {}

  operator double() const {
    return a * 1.0 / b;
  }

  fraction_pair(const fraction_array &arr) {
    a = arr[0];
    b = arr[1];
  }
};

int main() {
  fraction_pair p(1, 1);
  double d = p; //без явных приведений
  fraction_array a (2, 2);
  fraction_pair pa = a; // тоже без приведений.
}
```

Также в `C++` появились `explicit` конструкторы и операторы приведения типа. Они запрещают неявное приведение к типу. Например очень полезно всегда писать `explicit operator bool()`, потому что если у объекта есть каст к `bool`, то он будет использоваться где попало. Например, два объекта с кастом к булу отлично сравниваются `(a < b)`, когда на деле сравниваются их булы`(bool)a < (bool)b`. Explicit же запрещает их сравнивать без явного преобразования. Какие-то неявные очевидные касты остаются, например `if (a)` или `return a && (4 < b)`.

В C++ появилось 4 формы записи преобразования типа, которые чётко ограничивают область применения:

1. `static_cast<>` нужен для:
   (a) стандартные типы
   (b) наследование(справляется с тем, с чем может справиться во время компиляции)
   (c) пользовательские преобразования, типа `operator double`
   (d) преобразование в `void*`

   Если пользоваться неаккуратно, многие проблемы остаются. Если классы никак не связаны отношением наследования - оно не скомпилируется.
   Он применяется почти всегда, когда потребовалось явное приведение типов.

2. `const_cast<>` - преобразования в/из `const`
   Используется либо в реализации не-const-qualified `operator[]` через const-qualified, либо при работе со старыми сишными интерфейсами, которые почему-то принимают данные не по `const char*`, а по `char*`, хоть и не меняют

3. `reinterpret_cast<>` - преобразование указателей разных типов(насильно говорит "Ты теперь такого типа", без каких либо умных проверок или преобразований). Лучше им вообще не пользоваться, только в самых экстренных случаях, когда вы уверены, что ничего не сломается.

4. `dynamic_cast<>` - этот каст, в отличии от предыдущих, происходит на этапе выполнения программы, а не компиляции. Про него чуть позже.

Например:

```
1 int *pi;
2 char *pc;
3 pc = static_cast<char*>(pi); // шаблонный параметр - то, к чему преобразоывваем
```

Для чего нужны эти касты? Так теперь их легче искать в коде, программист думает, прежде чем сделать каст, ибо надо понять, какой каст уместнее. Важно то, что привычные конструкции преобразования типов из Си, вроде `(int)` переопределили, теперь он просто пытается применить все касты поочереди, пока у него что-то из этого не получится. Это не самый безопасный способ, особенно если учитывать то, что в конце концов он просто дойдёт до `reinterpret_cast<>`. Если у вас не кастуется что-то через `static_cast`, то компилятор скажет о проблеме, а `(type)` будет биться в конвульсиях, пока не скастует, попутно стреляя программисту в ноги.
С `reinterpret` надо всё равно быть аккуратным. Если `static` подвинет указатель, чтобы сошлись все поля, сделает корректное преобразование типов, позаботится о нас, то `reinterpret` просто скажет компилятору "считай, что это другой тип".

Теперь поговорим о `dynamic_cast`.

```
1  //A ← B
2  A* pA = new A;
3  B* pb = static_cast<B*> (pA);
4  pb->g(); // просто посмотрит, что всё это наследник
5  //он не поймёт, что у родителя нет такой функции, упадёт во время исполнения.
6  //---------
7  B *pb = dynamic_cast<B*> (pA); // он либо сделает каст, если это легально, либо вернёт NULL
8  if (pb == NULL) {
9    cout << "error";
10 }
```

Нам нужно что-то, что в онлайн может понимать настоящий тип класса.
Чтобы это всё работало, строго должен быть хотя бы один виртуальный метод у класса, но не обязательно должна быть таблица виртуальных функций(ибо с этими таблицами у компилятора больше свободы действий, но, в основном, они всё же создаются при создании виртуального метода), тогда можно будет по указателю на родительский класс понять его первоначальный тип, как наследника. Наглядный пример применения:
Пусть у нас есть программа с классом `IShape`, который имеет наследников `Square`, `Circle` и `Rectangle`.

```
1 std::vector<IShape*> shapes;
2 for (...) {
3   ISpahe *ps = shapes[i];
4   if (dynamic_cast<Rectangle*>(ps)){ //type_traits<> вернуло бы Ishape
5     //сохранить в бинарный файл.
6   } else {
7     //сохранить в текстовый файл.
8   }
9 }
```

это всё: RTTI - runtime type information.
Есть похожая вещь – `typeinfo`

```
1 intersect (IShape *p, std::vector<IShape> &v) {
2   type_info ti1 = typeid(*p);
3   for (...)
4     type_info ti2 = typeid(*v[i]);
5     if (strcmp(ti1.name, ''Square'') == 0 && strcmp(ti2.name, ''Circle'') == 0) { // во
        время работы узнает их тип
6     // и посмотрит имя.
7       // код пересечения фигур
8     }
9 }
```

При этом не стоит смотреть на имя класса через `typeinfo`, непонятно, что там действительно будет, здесь показано в качестве примера. Безопаснее сравнивать `typeinfo` оператором `==`. Также `typeid` работает только для `RTTI` классов, для других оно будет посчитано на этапе компиляции.

4. Стандарт C++11

4.1. Немного о стандартах

Какие есть стандарты языка C++:

Major – это основные стандарты, содержащие существенные изменения.

C++98 – стандарт языка в том виде, как мы его изучали с начала семестра.

C++11 – добавлено `auto`, `move`, `lambda`, `thread`, `unordered_map` и многое другое.

C++17 – находится в состоянии черновика. Должен выйти в финальной версии в конце года.

Minor – промежуточные стандарты. Содержат, в том числе, исправления ошибок основных стандартов. C++03, C++14.

`gcc` по умолчанию использует 98 стандарт. Чтобы включить поддержку 11 стандарта, необходимо компилировать с флагом `-std=c++11`.

4.2. default & delete

Раньше, чтобы запретить использовать какой-либо метод класса (например оператор копирования), его делали приватным. Такой трюк использовался, например, в `Singletone` (объект, существующий в единственном экземпляре) и в `scoped_ptr` (обёртка над указателем).
Кроме того, когда мы объявляли конструктор с параметрами, конструктор по умолчанию не генерировался, приходилось его тоже объявлять и определять. Выглядело это так:

```
1 class A {
2 public:
3   A(int);
4   A() {}; // т.к. есть конструктор от инта - не сгенерируется дефолтный
5           // он может пригодиться, например, для динамического массива (A *arr = new A[100])
6 private:
7   A(const A&); // теперь извне копировать нельзя
8   A& operator= (const A&); // и так тоже нельзя
9 }
```

Такой способ имеет недостаток – приватные методы можно вызывать внутри самого класса и в дружественных классах. В `C++03`, если объявить метод приватным, но не делать ему реализации - на этапе линковке будет ошибка. Есть один тонкий момент. Например мы записали `operator=`, как приватный и без реализации в классе `point`, также в самом классе сделали функцию `f`, которая его вызывает. Если нигде не вызвать в самом коде функцию `f`, то ошибку мы не найдём, если же `operator=` объявить, как удалённый метод в `C++11`, то ошибка будет независимо от того, есть ли вызов функции `f`. Это может сыграть роль, если мы, например, пишем библиотеку и не написали тестов для функции.
В `C++11` появилась возможность делать всё явно:

```
1 class A {
2 public:
3   A(int);
4   A() = default; // конструктор по умолчанию
5   A(const A&) = delete; // удаленный метод (использовать нельзя)
6   A& operator= (const A&) = delete; // удаленный метод (использовать нельзя)
7 }
```

4.3. override & final

Раньше писали так:

```
class Base {
public:
  virtual void f(int);
  virtual int g() const;
  void h(int);
};
```

```
class Derived : public Base {
public:
  void f(int);
  int g();       // не перекроется (забыли const)
  void h(int); // не перекроется (метод не виртуальный)
};
```

Чтобы понять, являются ли методы **f(), h()** виртуальными, надо посмотреть базовый класс. У метода **g()** забыли **const**, и он получилась перегрузка вместо перекрытия. (Дебажить очень сложно!)
В C++11 можно сказать - перекрывай мне функции явно!

```
class Derived : public Base {
public:
  void f(int) override;  // ok
  int g() override;      // compilation error
  void h(int) override;  // compilation error
};
```

Инструкция **final** связана, в основном, с оптимизацией. Вызовы виртуальных функций связаны с накладными расходами. **final** говорит, что с этого этапа наследования функция не виртуальна. Т.е. у всех потомков она не будет виртуальной. Тогда вызов будет быстрее потому что известно, что её перекрыть нельзя.

```
struct Base {
  virtual void f();
};
struct Derived : public Base {
  void f() final; // функция виртуальная, но у потомки её перекрыть не смогут
  void g() { f(); } // f() вызовется без просмотра виртуальной таблицы
};
struct DerivedDerived : public Derived {
  void f(); // ошибка: перекрывать нельзя
};
```

Классы тоже могут быть **final**. Такие классы не могут иметь детей :(

```
struct Base1 final { };
struct Derived1 : Base1 { }; // compilation error
```

4.4. Новые фишки конструкторов

Раньше из конструктора нельзя было вызвать другой конструктор этого же класса. Теперь можно.

```
class A {
  int avg = 1; // можно задать начальное значение
  A(int a1, a2) { avg = (a1 + a2) / 2; }
  // можно вызвать один конструктор из другого
  A (int *array) : A (array[0], array[1]) { }
};
```

Появились списки инициализации для векторов.

```
std::vector<std::string> v = {"AA", "AB", "AC"};
std::vector<std::string> v({"AA", "AB", "AC"});
std::vector<std::string> v{"AA", "AB", "AC"};
```

Пример использования здесь:

```
template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
         std::cout << "constructed with a " << l.size() << "-element list\n";
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};
```

Обратим внимание, что итератор в нём именно `const T*`, но это нам не сильно помешает, можно итерироваться и по ним, то есть если хочется достать какие-то значения - используем следующую конструкцию:

```
template<class T>
class vector {
public:
  vector(std::initializer_list<T> list) {
    data = new T[list.size()];
    T* dest = data;
    for (auto &val : list) {
      *dest++ = val;
    }
  }
private:
  T* data;
};
```

Небольшой вопрос на подумать — чем отличается `vector<int> v(4)` от `vector<int> v{4}`?

Правильный ответ — в первом случае создастся вектор на 4 элемента, а во втором один вектор с первым элементом 4.

4.5. lvalue и rvalue

Кто хочет получить перелом мозга - могут почитать формально в стандарте. Дадим менее формальное, но не менее корректное описание.

lvalue – может быть и в левой, и в правой части присваивания(переменные, включая константы)

- продолжает существовать за пределами места, где был использован(например аргумент для вызова функции)
- есть имя
- можно взять адрес

rvalue – выражение, которое может быть только в правой части присваивания

- не существует за пределами своего scope
- временное значение

Например:

```
int a = 42;
int b = 43;
int c = a * b; //это rvalue и всё окей
a * b = 42; //всё не окей, потому что это rvalue, ловите error
```

```
vector<Person> people;
people.push_back(Person("Antient evil", 9999)); //Person("Antient evil", 9999) - это rvalue
```

```
int square(int x) {return x * x;}
int a = square(5); // Это тоже rvalue
```

Логичный вопрос: зачем мне это всё?
Ответ: во имя оптимизации, конечно же. c++ - это быстрый язык для умных дядечек, которым не лень сделать отдельные конструкторы для разных видов значений, чтобы программа работала быстрее. Почему без разделения на подобные типы программа работает медленнее? Сейчас разберёмся.
Пусть `X` - это какой-то весомый класс, который хранит ресурсы: указатель на динамически выделенную память, файлы и т.п.

```
X foo() {/*реализация*/}
X a;
a = foo();
```

Что только что произошло? Очень много чего. Функция вернёт какое-то значение, его нужно будет склонировать во временную память, потом нужно отчистить то, что уже валялось в `a`, после присвоить в `a` всё из временной памяти и освободить ресурсы во временной памяти. А если foo возвращает 80 вёдер текста? Не порядок, нам нет дела до временной памяти, она уходит и приходит, поэтому мы хотим сразу всё скопировать в переменную без лишних затрат. Добавим классу ещё один конструктор, так называемый move constructor

```
X(X&& x) {std::swap(this->data, x.data);}
```

`&&` – это rvalue reference. Заметим вот ещё что, у нас был реализован `operator=`, который тоже занимался нудным копированием, нам хочется и его сделать быстрее, но есть грязные хаки, а именно - `swap idiom`. Оператор присваивания можно не менять, если в него передаётся не ссылка, а копия объекта. Почему? У нас передаётся в оператор копия объекта, он в нём копируется

с помощью `rvalue`, потому что мы написали такой конструктор для этой ситуации, а потом внутри данные этой копии подменяются. Т.е. по сути он работает так же быстро... ну может самую малость медленнее, зато точно не ошибётся.
Теперь давайте разберём побольше краеугольных случаев. Напишем свой своп.

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

Внимание, вопрос. Если реализован `move` конструктор у класса `T`, будет ли это работать быстрее? Нет. Так как `a` и `b` - это `lvalue`, у них есть имена. Если программист с++ может выстрелить себе в ногу – нужно ни в коем случае не лишать его этой возможности, поэтому добавлен специальный каст `lvalue reference` к `rvalue`, а имя ему – `std::move`.

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

Теперь всё быстренько.
В этом примере без `swap` нам не обойтись, но стоит использовать его аккуратно, ибо он не делает ничего особое умнее, кроме как приводит `lvalue` к `rvalue`

4.5.1. std::forward и universal reference

В C++ есть ещё одна превосходная вещь – `perfect forwarding`.
Начиная с 14ого стандарта появилась функция `make_unique`, которая создает уникальный указатель. Разберём её, как пример.
Мы хотим сделать такую функцию, чтобы она конструировала класс `unique_ptr` от аргумента, типы A и T шаблонные

```
template<A, T>
unique_ptr<A> make_unique(T a) {
  return unique_ptr(new A(a));
}
```

У нас шаблоанный `T`, поэтому возникают проблемы, когда мы начинаем хотеть передавать туда что душе угодно: ссылки, константные ссылки, rvalue, lvalue.
Было перелопачено куча правил шаблонов и прочее, лишь бы это работало.
В 11ом стандарте появилось правило схлопывания ссылок:
`A& &` → `A&`
`A&& &` → `A&`
`A& &&` → `A&`
`A&& &&` → `A&&`
Можно не запоминать все эти правила, просто запомним, что они есть.
Пришла пора поговорить про `universal reference - make_unique(T&& a)`
С учётом того, что `T` – это шаблонный тип, `T&&` – это `universal reference`, может у вас дежавю и вы такое уже видили, но это не `rvalue`, такое выражение принимает всё, константные, неконстантные ссылки, rvalue, литералы и любую чушь.
Оно работает по правилам схлопывания ссылок.
Но как мы помним, если у переменной есть имя - значит оно будет `lvalue`, а мы хотим дальше

работать с ней, как с `rvalue`
Есть замечательная шаблонная магия, которая сделанно ровно только для этого: `std::forward`

```
template<A, T>
unique_ptr<A> make_unique(T&& a) {
  return unique_ptr(new A(std::forward<T>(a)));
}
```

`std::forward` работает только с универсальной ссылкой, после того, как мы его применяем - то, что он вернёт, будет ровно того же типа, что был передан в функцию через универсальную ссылку.

```
make_unique<A>(f()) // как будто бы сделает new A(f());
make_unique<A>(b); // как будто new A(b)
```

4.6. auto & decltype

`auto` - удобный способ для самых ленивых людей, даже слишком удобный.

```
auto x = expression;
//выведет сам по expression.

decltype(expression) y = x;
//тип будет такой же, как у expression.
//т.е. он сделает y с типом, как у expression, но запишет туда x;
```

Немного более наглядных примеров, чтобы почувствовать разницу

```
const std::vector<int> v(1);
auto a = v[0];          // a - int
decltype(v[0]) b = 1; // b - const int&

//auto можно уточнить для констант, ссылок и указателей
std::string s = "hello";
auto &s1 = s;           // s1 - std::string&
const auto &c = v[0]; // c - const int&

//Также можно использовать decltype в typedef
typedef decltype(v[0]) new_type;
```

Увлекаться авто не надо, если ставите авто везде - идея плохая, если у вас есть собственный класс, другим будет непонятно, что выведется, в отличие от STL, который все более менее знают.

Обычно используют всё-таки `auto` - оно работает в том же смысле, что и автоматическое выведение типов при вызове шаблонных функций. Там семантика старается отображать именно смысл "T - ну это либо `int`, либо `string`, либо `int*`" (т.е. не захватывая const и ссылки). `decltype` - довольно специфичная штука, когда нужно именно получить то, как была объявлена некая переменная или получить тип выражения, когда нельзя применить `auto` (например, чтобы запихнуть его в шаблоннный параметр).

4.7. nullptr

Два слова о nullptr
обычно в C++ пишут что-то вроде

```
1 #define NULL 0
```

А это не есть хорошо, например такой код:

```
1 void f(int *ptr) {...}
2 void f(int n) {...}
3
4 f(NULL); // не поймём
5 f(nullptr); // поймём
```

Порой некоторые компиляторы определяют NULL немного сложнее:

```
1 #define NULL (void *)0
```

Почему так тоже не всё хорошо? Дело в преобразованиях типов.
Стоит везде вместо NULL использовать nullptr, так намного надёжнее.

4.8. Вычисления в compile time

1. static_assert

```
1 static_assert(sizeof(unsigned int) * 8 == 32, ''16bit CPU is not supported'');
```

   По сути то же самое, что и обычный assert, но работает во время компиляции.

2. constexpr

```
1 constexpr unsigned fibonacci(unsigned i) {
2   return (i <= 1u) ? i : (fibonacci(i-1) + fibonacci(i-2));
3 }
```

   Этот монстр будет считаться во время компиляции, если на вход будут подаваться константы.
   Подобные вещи могут пригодиться, когда нужно посчитать функцию во время компиляции, например, для константного массива или параметра шаблона. Если вы делаете какой-то класс S_group и хотите через объявление S_group<7> s получить группу перестановок размера 7!. constexpr подойдёт как раз, чтобы выделить массив нужного размера.

4.9. begin(), end(), for(:)

Появились шаблонные функции begin() и end(), они не делают ничего умнее, кроме как подставляют .begin() и .end() соответственно, если они есть, а если там массив - работают как должны работать с массивом.
Ещё появился for по контейнерам и массивам.

```
1 for (auto& x : v) {
2   ++x;
3 }
4 //Он просто раскроет это во что-то такое:
5 auto &__range = v;
6 for (auto __begin = begin(__range), __end = end(__range); __begin != __end; ++
     __begin) {
7   x = *__begin;
8   ++x;
9 }
```

Здесь важно, что перед x имеет тип auto&, то есть ссылку. В противном случае в x будут копироваться значения из v, и ++x будет изменять только его, а содержимое контейнера останется неизменными. Более того, версия без ссылки будет работать медленнее, так как будет тратиться время на копирование.
Такая конструкция цикла for не будет работать для массивов, память под которые была выделена динамически, ведь неизвестно на этапе компиляции, где у их конец. Если же размер задан на этапе компиляции, всё норм.
Стоит также по возможности всегда использовать ссылку(есть случае, когда нельзя так делать, чтобы не портить контейнер), ибо так работает быстрее и не нужно всегда копировать элементы.

4.10. lambda

Что-то полезное и интересное.
Раньше мы писали компараторы, чтобы, например, передать их в функцию сортировки. Сейчас появился более короткий способ записать функции, которым не нужно какое-то особое имя.
Лямбда функции имеют три секции:

1. capture list
   Это переменные в квадратных скобочках, которые используются в теле функции, но не являются её аргументами. Они "захватываются" из той области видимости, где лямбда создается.

2. аргументы
   В круглых скобках записаны аргументы, как и у обычной функции

3. тело функции
   Тут тоже всё как и у обычной функции.

Примеры:

```
//Отсортировать по абсолютным значениям
std::sort(v.begin(), v.end(), [](int a, int b){return abs(a) < abs(b);} );

//Изменяем total, прибавив к нему все значения из вектора
for_each(v.begin(), v.end(), [&total](int x) { total += x; });

//Не изменяем total, передав его без ссылки.
for_each(v.begin(), v.end(), [total](int x) { x -= total; });
```

Можно в capture list передавать особенные символы: [=] – взять всё по значению, [&] – взять всё по ссылке, [] – ничего не брать. Можно также комбинировать это, например: [=, &x], [&x, y], [&, y].
Также можно явно указать возвращаемое значение lambda функции, как в примере ниже, а также можно дать ей имя.

```
auto func = [](int a, int b)->bool {return a > b;};
bool res = func(4, 5);
sort(v.begin(), v.end(), func);
```

Пара вопросов на засыпку. Что выведет этот код?

```
int y = 0;
auto x = [y](){ std::cout << ++y << "\n"; };
x();
x();
```

Правильный ответ: сообщение об ошибке. Он считает y как read-only переменную, поэтому не может делать ей инкремент.
Какой же тип у лямда функции? Каждая лямбда генерирует свой тип.

4.11. Variadic templates

Многоточия в этом разделе - это синтаксис, а не лень автора.
Сначала немного в целом про шаблоны в C++11.
Исправлена проблема с двойными треугольными скобками, они больше не воспринимаются, как оператор.
Появилась чудесная возможность делать `typedef` для шаблонов, используя `using`:

```
1 template<typename T>
2 using 2dvector = std::vector<std::vector<T>>;
```

Теперь ближе к сути:

```
1 template <typename T>
2 T sum(T n) {return n;}
3
4 template<typename T, typename ...Args>
5 T sum(T n, Args... rest) {return n + sum(rest...);}
6 // это рекурсия, которая развенётся в момент компиляции
7
8 double d = sum(3, (double)4.3, 5);
```

Компилятор начинает работать. Я передаю четыре параметра. Он генерирует нужную мне функцию, теперь ему нужна функция от трёх параметров, он рекурсивно генерирует и её. Но для рекурсии нужна база – функция с одним или нулём аргументов.

"..." - отщипывает один аргумент, затем тот же шаблон применяется ещё раз, пока не дойдём до базы "Шаблонной рекурсии" с одним аргументом `T sum(T n)`
Для этого примера компилятор сгененрирует три функции:

```
1 T sum(T, Args ...) { } //[with T = int; Args = {double, int}]
2 T sum(T, Args ...) { } //[with T = double; Args = {int}]
3 T sum(T) { }           //[with T = int]
```

Как сделаны `printf, scanf` в языке си?
Когда мы фиксируем число и тип аргументов, компилятор сам генерирует код.
Здесь нам передают строку, а всё остальное просто кладется на стек, и мы сами в рантайме должны распарсить, сколько и каких там аргументов. Стек не знает, сколько на нём переменных и каких они типов, это должны контролировать мы.
Труднообнoружимые – это те ошибки, которые компилятор не найдет. Никакой возможности проверить аргументы нет.
Рассмотрим, как примерно реализована функция `printf` в языке Си.

```
// va_arg, va_list, va_start - это макросы
void simple_printf(const char* fmt, ...) {
// Он не будет во время компиляции понимать, какое там кол-во параметров,
// он просто засунет на стек всё, что было передано
  va_list args;
  va_start(args, fmt);
  while(*fmt != '\0') {
    if (*fmt == 'd') {
      int i = va_arg(args, int) // Если 'd' - начиная с этого адреса копируй переменную
      // размера инт по байтам, а потом сдвинь args на такое же количество байтов.

      // далее разбор кучи случаев
    }
  fmt++;
  }
  va_end(args);
}

//ошибки не будут замечены компилятором:
printf("%s", 5);
printf("%d %d", 4);
printf("%d", 5, 4);
```

Сделаем то же самое, но с контролем типов или хотя бы проверкой в рантайме. Нам поможет товарищ C++:

```
void printf(const char *s) {
  while (*s) {
    if (*s == '%' && *(++s) != '%')
      throw std::runtime_error("invalid format");
    std::cout << *s++;
  }
}

template<typename T, typename ...Args> // такой синтаксис
void printf(const char *s, T value, Args... rest) { // такой синтаксис
  while (*s) {
    if (*s == '%' && *(++s) != '%') {
      std::cout << value;
      printf(++s, rest...); // такой синтаксис
      return;
    }
    std::cout << *s++;
  }
  throw std::logic_error("extra arguments provided to printf");
}
```

Вопрос к экзамену. Какие ошибки эта версия лечит, а какие нет? (если здесь всё написано правильно, должны вылечиться бесконечные хождения по памяти и выход за границы выделенной памяти).
Важно помнить, что компилятор сгенерирует кучу функций (для всех промежуточных состояний рекурсии).

4.12. hash

Для `unordered_map` и `unordered_set` нужен хэш, давайте его сделаем:

```
class Point {
  private:
    int x, y;
  public:
    bool operator ==(const Point& rhs) const {
      return x == rhs.x && y == rhs.y;
    }
}
namespace std {
  template<>
  struct hash<Point> {
    size_t operator()(Point const & p) const {
      return (std::hash<int>()(p.getX()) * 51 + std::hash<int>()(p.getY()));
    }
  };
}

std::unordered_set<Point> // можно было отедельно сделать свой хэш и компаратор
std::unordered_set<Point, Point_comparator, Point_hasher>
```

4.13. unique_ptr, array, tuple, regex

`unique_ptr` – это то же самое, что и `auto_ptr`, в него тоже можно копировать только один раз, но в отличие от своего предшественника, у него не реализован оператор копирования и присваивания для `lvalue`, поэтому его можно передать только с помощью `std::move`, что обязывает программиста подумать, прежде чем скопировать куда-то уникальный указатель. Чуть подробнее есть на гисте с практики.

1. Появился шаблонный массив с итераторами и парой удобных фишек. Мы реализовывали что-то подобное на практике:

```
std::array<int, 3> arr = {1, 2, 3};
```

2. также теперь есть `tuple` - это обобщание `pair` на несколько полей(сделан на variadic templates).

```
auto t = std::make_tuple("String", 5.2, 1);
std::cout << std::get<0>(t) << ' '
          << std::get<1>(t) << ' '
          << std::get<2>(t) << '\n';
```

3. `regex`, `regex_search` и `regex_match` тоже появились в С++11, путь они и помедленнее, чем свой парсер, зато удобнее.

```
regex reg("[a-zA-Z_][a-zA-Z_0-9]*\\.[a-zA-Z0-9]+");
//Здесь два слэша, один чтобы экранировать точку, другой, чтобы экранировать слэш
regex_search("Print readme.txt", reg);
```

4.14. std::function и std::bind

Нам пригодится хранить где-то функции для потоков
На C++ понятие "то что можем запустить" расширилось, у нас есть как минимум три сущности:

1. указатель на функцию
2. функтор
3. лямбда выражения

Не хочется писать три варианта кода (для всех вариантов функций). В STL есть `std::function`, которая уже перегружена для всех трёх вариантов.
Вот у меня есть функция `execute()`, она получает вектор таких сущностей. Можно пройти по вектору и всё вызвать.

```
1  void execute(const vector<function<void ()>> &fs) {
2    for (auto &f : fs)
3      f();
4  }
5
6  void plain_old_func() {
7    cout << "old plain function" << endl;
8  }
9
10 struct functor {
11   void operator()() const {
12     cout << "functor" << endl;
13   }
14 };
15
16 int main() {
17   vector <function<void ()>> x;
18   x.push_back(plain_old_func);
19
20   functor functor_instance;
21   x.push_back(functor_instance);
22
23   x.push_back([](){cout << "lambda" << endl; });
24   execute(x);
25 }
```

Также появился `std::bind`, который позволяет создать обёртку над функцией и тем самым уменьшить кол-во её параметров, а всем неиспользуемым присвоить константные значения.

```
1  void show_text(const string &t) {
2    cout << "TEXT: " << t << endl;
3  }
4
5  int main() {
6    std::vector<function<void ()>> x;
7    function<void ()> f = bind(show_text, "Bound function");
8  //Хочу функцию чтобы у неё как бы было ноль параметров и я мог положить это внутрь
9
10   x.push_back(f);
11   execute(x);
12 }
```

Теперь захотели сделать функцию, которая будет один параметр держать константным, а другой переменным.

```
using namespace std::placeholders;
int multiply(int a, int b) {
  return a * b;
}

int main() {
  auto f = bind(multiply, 5, _1); //_1 - placeholder
  //первый параметр функции f подставится на второе место функции multiply
  // int f(int _1) return multiply(5, _1);
  cout << "out: " << f(6);
}
```

У `std::function` фиксированный размер, GNU g++ говорит, что это 32 байта. Вопросы для продвинутых(напишите в беседу, если ответите): Что произойдёт, если положить в него лямбду, в которой по значению захвачена тонна переменных, и которая сильно больше std::function? Каковы последствия производительности от применения std::function?

4.15. std::thread

```
void f1(int n) {
  std::cout << "f1: " << n << std::endl;
}

void f2(int &n) {
  n++;
}

int main() { // три потока - один мэин, потом ещё два.
  int m = 45;
  std::thread t1(f1, m); // в первом потоке вызовется f1 и будет передано m по значению
  std::thread t2(f2, std::ref(m));
  // тут будет вызвано по ссылке f2;
  // зачем ref? чтобы std::function понял, что нужно сгенирировать функцию.
  // Внутри этого треда как бы вызваетсяя std::function.
  // Нужно явно указать, что это ссылка, ибо это всё шаблоны,
  // и компилятор должен понять, какую функцию генерировать
  t1.join();
  t2.join();// дождаться первого и второго потока
  cout << m;
}
```

```
std::vector<std::thread> threads;

for(int i = 0; i < 5; i++) {
  threads.push_back(std::thread(
    []() { std::cout << std::this_thread::get_id() << std::endl; }
  ));
}

for (auto &thread : threads) {
  thread.join();
}
```

4.16. Подробнее про многопоточность

4.16.1. Краткое напоминание

Для начала вспомним, чем отличаются процессы и потоки.

- Процессы (программы):
  - Несколько программ запущено одновременно
  - Они имеют независимые адресные пространства
  - Каждая программа думает, то она живет в системе одна
  - Операционные системы поддерживают динамическую адрессацию, то есть если обе программы хотят обратиться к памяти по адресу 42, на самом деле они обращаются к разной памяти.
- Потоки (функции в программе):
  - В одной программе можно несколько функций запустить в параллельных потоках
  - У них будет одна область памяти (адресное пространство)
  - Общение между потоками происходит быстро

Теперь рассмотрим пример. У нас есть вектор. Хотим просуммировать его элементы. Можем это сделать для двух поливинок независимо – в разных потоках:

```
typedef vector<int> ivec;
// функция, выполнающая работу:
void sum_vec(const ivec& v1, size_t start, size_t end, const ivec& v2, ivec& res) {
  for (int i = start; i < end; i++) {
    res[i] = v1[i] + v2[i];
  }
}
void parallel_sum_vec(const ivec& v1, const ivec& v2, ivec& res) {
  // поток для первой половины:
  std::thread t1(sum_vec, (size_t)0, v1.size() / 2, cref(v1), cref(v2), ref(res));
  // поток для второй половины:
  std::thread t2(sum_vec, v1.size() / 2, v1.size(), cref(v1), cref(v2), ref(res));
  // std::cref() делает константную ссылку
  t1.join(); // Обязательно надо дождаться завершения работы потока
  t2.join(); // до того, как переменная thread умрет.
}
parallel_sum_vec(v1, v2, res);
```

Компилируем это счастье с ключом `-pthread` (но это только в GCC. В других компиляторах надо смотреть на их требования).

Важно помнить, что не все алгоритмы можно распараллелить. Например в алгоритме Евклида каждая следующая операция должна быть выполнена после предыдущей.

4.16.2. Потоки на однопроцессорном компютере

Теперь предположим, что у нас однопроцессорный компютер. Как ни странно, на нем тоже можно выполнять многопоточные программы. Как это работает?
В ОС есть планировщик. На материнской плате компа есть таймер. Каждому потоку планировщик даёт потоку на выполнение определенное фиксированное количество миллисекунд. По

прошествии отведенного времени он переносит все регистры этого потока в память, переходит к новому потоку, загружает его память в регистры и возобновляет его. Это называется переключение контекста.
Переключение контекста работает не мгновенно, то есть связано с накладными расходами. В следстве этого многопоточные программы на однопроцессорных машинах работают ещё медленнее, чем их однопоточные товарищи.
Тем не менее многопоточные программы пишут и для однопроцессорных компов. Например, есть поток, который работает с переферийным устройством. Он ждёт сигнала от сетевой карты (или ввода с клавиатуры). В таком случае распараллеливание может быть эффективно, ведь в то время, когда один поток ждёт, остальные работают.

4.16.3. Состояние гонки

Самое главное, что надо знать о многопоточности, и с чем всегда возникает больше всего проблем – это ”Состояние гонки” (Race condition).
Дело в том, что переключение может произойти абсолютно в любой момент, даже во время выполнения операции `x += 5` (как мы помним, одной такой операции соответствует длинная последовательность процессорных инструкций). И программист не знает, когда произойдет переключение. А когда процессоров (или ядер) несколько, то вообще невозможно определить, в каком порядке несколько процессов будут выполнять свои команды.
Рассмотрим пример:

```
1  void hello() {
2    std::cout << ”Hello from thread” << std::this_thread::get_id() << std::endl;
3  }
4
5  int main() {
6    std::vector<std::thread> threads;
7    for (int i = 0; i < 5; ++i) {
8      threads.push_back(std::thread(hello));
9    }
10   for (auto& thread : threads) {
11     thread.join();
12   }
13   return 0;
14 }
```

Создатель этой программы рассчитывал, что создадутся пять потоков, каждый из них выведет строчку со своим id, и эти строчки в каком-то порядке появятся на экране. То есть вот так:

```
Hello from thread 140276650997504
Hello from thread 140276667782912
Hello from thread 140276659390208
Hello from thread 140276642604800
Hello from thread 140276676175616
```

Однако из-за переключения контекста и одновременной работы нескольких процессоров строчки могли вклиниться друг в друга и превратиться в бессмысленный набор слов:

```
1  Hello from thread Hello from thread Hello from
2  thread 139810974787328Hello
3  from thread 139810983180032Hello from thread
4  139810966394624
5  139810991572736
6  139810958001920
```

Когда мы запускаем программу в отладчике, скорость выполнения будет низкой, поэтому операции будут успевать выполняться полностью, и таких проблем мы почти не увидим. Поэтому

отлаживать многопоточные приложения лучше `printf`-ами. Однако `printf`-ы тоже замедляют работу программы, поэтому при их использовании ошибки будут тоже проявляться реже. Отсюда вывод – многопоточный код надо писать сразу правильно!

4.16.4. Атомарные типы и операции

Атомарные операции – это те, которые не могут быть прерваны переключением контекста. Атомарные типы – это те, все операции с которыми атомарные. В языке C++ все примитивные типы не являются атомарными. Если переменная влезает в регистр процессора, то операция чтения/записи, скорее всего, не прервется. Переменная типа `int` укладывается в один регистр, поэтому запись будет атомарной, а если тип побольше, то операция может и прерваться.

Рассмотрим пример со связным списком. Первый поток вставляет элементы в середину списка и переприсваивает указатели. В это время второй поток выводит элементы списка на экран. Если переключение произойдет в то время, когда первыый поток ещё не закончил присваивать указатели, второй поток получит список в некорректном состоянии.

Ещё один классический пример. Один поток постоянно увеличивает значение переменной на 1, а другой поток проверяет её на четность, и если она четная, выводит её на экран. Тем не менее, на экране могут появиться нечетные числа, потому что между проверкой и выводом на экран первый поток мог изменить значение переменной на 1.

И последний пример. Создаем пять потоков, каждый из которых захватывает счетчик по ссылке и сто раз увеличивает его значение.

```
1  struct Counter {
2    int value;
3    Counter() : value(0) { }
4    void increment() {
5      ++value;
6    }
7  };
8
9  int main() {
10   Counter counter;
11   vector<thread> threads;
12   for(int i = 0; i < 5; ++i)  {
13     threads.push_back(thread(
14       [&counter]() {
15         for(int i = 0; i < 100; ++i)
16           counter.increment();
17       }
18     ));
19   }
20   for (auto& thread : threads) { thread.join(); }
21   cout << counter.value << endl;
22 }
```

Мы ожидаем, что выведется 500, однако, запустив это много раз, мы видим, что результат меняется (причем кардинально) от запуска к запуску. Почему так происходит?
Допустим первый поток прочитал значение счетчика, оно равно 0.
Сразу после этого второй поток тоже прочитал и тоже получил 0.
Затем первый поток увеличил значение и записал туда 1.
Второй поток сделал то же самое и тоже записал туда 1.
В результате счетчик увеличислся не на 2, как ожидалось, а только на 1.

4.16.5. std::mutex

Мьютекс (или Мутекс, как ближе вашему сердцу) – англ. mutual exclusion – "взаимное исключение" – объект, позволяющий одному потоку "закрыть доступ" к какому-то участку кода для других потоков. Это можно предсавить, как комнату, у которой дверь закрывается и открывается только изнутри. Поток заходит в эту комнату, закрывает дверь, и другие потоки не могут войти, пока тот не откроет и не выйдет.
Изменим код класса `Counter`, чтобы инкремент работал атомарно:

```
struct Counter {
  mutex mtx; // создали объект 'мьютекс'
  int value;
  Counter() : value(0) { }
  void increment() {
    mtx.lock(); // вошли и закрыли дверь
    ++value;
    mtx.unlock(); // открыли дверь и вышли
  }
};
```

Теперь конкуррентного доступа к памяти происходить не будет, и `value` будет каждый раз увеличиваться на единицу.

4.16.6. std::lock_guard

`lock_guard` – это объект-оболочка над мьютексом, который держит мьютекс закрым, пока существует. В констукторе `lock_guard` вызывает `mtx.lock()`, а в деструкторе вызывает `mtx.unlock()`. Перепишем тот же счетчик с помощью `lock_guard`:

```
struct Counter {
  mutex mtx; // создали объект 'мьютекс'
  int value;
  Counter() : value(0) { }
  void increment() {
    lock_guard<mutex> lg(mtx); // создали гард, он заблокировал мьютекс
    ++value;
  } // вызвался деструктор гарда и освободил мьютекс
};
```

Можно заметить, что `std::lock_guard` использует идиому RAII (Resource Acquisition Is Initialization). Эта идиома хороша в том числе и тем, что обеспечивает безопасность относитально исключений.

4.17. Множественное наследование

Множественное наследование (multiple inheritance) – возможность наследоваться сразу от нескольких классов.

```
struct Student {
  string name() const { return name_; }
  string university() const { return university_; }
private:
  string name_, university_;
};



```

```
struct FullTimeEmployee {
  string name() const { return name_; }
  string company() const { return company_; }
private:
  string name_, company_;
};

struct RichStudent : Student, FullTimeEmployee {
  string name() const { return Student::name(); }
};
```

В примере выше класс 'богатый студент' наследуется от классов 'студент' и 'работник'. При этом возникает проблема. У богатого студента два имени – `Student::name_` и `FullTimeEmployee::name_` Почему так происходит? Рассмотрим, как могут быть расположены объекты базовых классов при множественном наследовании:

Student
FullTimeEmployee
поля RichStudent

Это ведет за собой ряд других интересностей. Например в следующем коде `st_p` и `re_p` будут указывать на разные адреса в памяти, потому что `reinterpret_cast` тупо поменяет тип, а `static_cast` поймет, что надо сдвинуться назад, чтобы поместился `Student`.

```
FullTimeEmployee e;
RichStudent *st_p = static_cast<RichStudent *>(&e);
RichStudent *re_p = reinterpret_cast<RichStudent *>(&e);
```

Проблему с дублированием полей можно решить, заведя ещё более общий базовый класс и сделав студента и работника интерфейсами – классами, у которых все методы чисто виртуальные и нет полей:

```
struct Person {
  string name() const { return name_; }
  string name_;
};

struct IStudent {
  virtual string name() const = 0;
  virtual string university() const = 0;
  virtual ~IStudent(){ }
};

struct IFullTimeEmployee {
  virtual string name() const = 0;
  virtual string company() const = 0;
  virtual ~IFullTimeEmployee(){ }
};

struct RichStudent : Person, IStudent, IFullTimeEmployee {
  string name() const override { return Person::name(); }
  string university() const override { return university_; }
  string company() const { return company_; }
  string university_, company_;
};
```

Пару слов о том, что такое интерфейс. Это же не просто класс, у него немного другие цели. Например, раньше у нас можно было создавать отдельно `Student`, отдельно `FullTimeEmployee` и какие-то функции принимали эти переменные по значению. Что изменилось, если мы переходим к интерфейсам? Ну, довольно много чего – теперь по значению нельзя, надо создавать отдельно класс `Student` (реализация интерфейса `IStudent`) и `FullTimeEmployee`, итого получается два

интерфейса, один базовый класс (для всех людей) и три реализации. Тогда у нас в наследовании получается всегда не более одного "обычного класса с данными" и >= 0 интерфейсов – прямо как в Java (если кто-то её знает).

Если интерфейсы сделать не получается, можно сделать, чтобы `Student` и `FullTimeEmployee` наследовались от `Person` виртуально. Тогда в их общих наследниках все поля `Person` будут встречаться по одному разу:

```
struct Person { };
struct Student : virtual Person { };
struct Employee : virtual Person { };
struct RichStudent : Student, Employee { };
```

Заметим, что `virtual` надо писать именно при наследовании кого угодно от класса `Person`, а не ниже по иерархии как раз для того чтобы `Person` не складывался в память дважды. Теперь допустим, что и `Student`, и `Employee` вызывают конструктор `Person`:

```
struct Person {
  explicit Person(string const& name): name_(name) { }
  string name_;
};
struct Student : virtual Person {
  explicit Student(string const& name): Person(name) { }
};
struct Employee : virtual Person {
  explicit Employee(string const& name): Person(name) { }
};
```

Но при конструировании `RichStudent` мы хотим вызвать его только один раз. Для этого надо его явно вызвать:

```
struct RichStudent : Student, Employee {
  explicit RichStudent(string const& name): Person(name), Student(name),
  Employee(name) { }
};
```

Теперь конструктор `Person` вызовется только один раз.

4.18. Метапрограммирование

Метапрограммирование – это когда программа генерирует/модифицирует другую или сама себя.
Метапрограммирование в C++ – это когда какие-то операции (вычисления) производятся во время компиляции.
Метафункция во время компиляции:

- вычисляет значение/константу (`constexpr`) или тип
- выбирает тот или иной вариант алгоритма (`#ifdef`)
- генерирует алгоритм (`variadic templates`)

Рассмотрим следующий код:

```
template <int N>
struct Factorial {
  static const int value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0> {
  static const int value = 1;
};

std::cout << Factorial<10>::value << std::endl;
```

При компиляции последней строчки создастся класс `Factorial<10>`, для которого создастся класс `Factorial<9>`, и.т.д.
То есть компилятор будет рекурсивно генерировать классы, пока не дойдет до `Factorial<0>`, который является базой рекурсии.
После этого в runtime факториалы чисел от 0 до 10 будут вычисляться за $\mathcal{O}(1)$ без предподсчета!
Интересный факт. Шаблоны в C++ обладают Тьюринг-полнотой. То есть пользуясь только ими можно абсолютно любую программу, которую можно написать на C++ (подробнее в статье). Причем все вычисления будут происходить во время компиляции.
Для вычислений в compile time при помощи шаблонов полезно сначала "написать" решение в полностью функциональном стиле (скажем, на Haskell), после чего оно практически один-в-один переводится в шаблоны. А то могут возникать вопросы: "как же без циклов", "как же без переменных"... Так же, как в функциональных языках – рекурсия и дополнительные функции/параметры.

Рассмотрим другой пример. Функция, вычисляющая, евклидово расстояние между двумя точками "втупую":

```
float euclidean_baseline(int n, float* x, float* y) {
  for (int i = 0; i < n; ++i) {
    float num = x[i] - y[i];
    result += num * num;
  }
  . . .
}
```

Компиляторы имеют разные расширения, позволяющие заставить процессор что-то делать эффективнее. Например, следующий код решает ту же задачу, но с помощью супер-крутых эффективных команд, встроенных в процессор (`intrinsic functions`):

```
float euclidean_intrinsic(int n, float* x, float* y) {
  __m128 euclidean = _mm_setzero_ps();
  for(; n > 3; n -= 4) {
    __m128 a = _mm_loadu_ps(x);
    __m128 b = _mm_loadu_ps(y);
    __m128 a_minus_b = _mm_sub_ps(a, b);
    __m128 a_minus_b_sq = _mm_mul_ps(a_minus_b, a_minus_b);
    euclidean = _mm_add_ps(euclidean, a_minus_b_sq);
    x += 4;
    y += 4;
  }
  . . .
}
```

Не будем разбираться, как это работает, главное что это работает быстро, но не на всех процессорах. Поэтому иногда при компиляции нужно выбирать "тупую" реализацию. Здесь нам на помощь снова придет метапрограммирование:

```
float euclidean(int dim, float* x ,float* y) {
#ifdef __SSE__
  return euclidean_intrinsic(dim, x, y);
#else
  return euclidean_baseline(dim, x, y);
#endif
}
```

Функция `euclidean` по-умолчанию имеет внутри себя указатель на обычную реализацию, однако, если процессор поддерживает расширение команд SSE (в таком случае в стандартных заголовках есть дефайн `__SSE__`), то указатель становится на продвинутую реализацию.

4.18.1. SFINAE

SFINAE расшифровывается "Substitution Failure Is Not An Error" – неудачная подстановка – не ошибка.
Что же это означает?
Пусть есть **шаблонная** функция, имеющая несколько перегрузок. Некоторые перегрузки вызывают синтаксическую при попытке подставить туда наши шаблонные параметры. Но при этом не происходит ошибка компиляции, а перегрузка просто исключается из списка кандидатов, и компилятор ищет более подходящую версию.
Рассмотрим пример.

```
template <typename T>
void show (typename T::iterator x, typename T::iterator y) {
  for(; x != y; ++x)
    cout<< *x << " ";
  cout << endl;
}

show<int>(16, 18);
```

Здесь будет ошибка компиляции, потому что внутри `int`'а нет класса `iterator`.
Компилятор скажет:

```
error: no matching function for call to 'show(int, int)'
note: candidate  is:
note: template void show(typename T::iterator, typename T::iterator)
```

А теперь добавим ещё одну реализацию этой же функции

```
template <typename T>
void show (typename T::iterator x, typename T::iterator y) {
  for(; x != y; ++x)
    cout << *x << " ";
  cout << endl;
}

template <typename T>
void show(T a, T b) {
  cout << a << "; " << b << endl;
}

show<int>(16, 18);
vector<int> arr = {16, 17, 18};
show<vector<int>>(arr.begin(), arr.end());
```

Конкретно в таком примере не указывать шаблонный параметр нельзя – компилятор не может сам догадаться, что если ему передали нечто, то надо попробовать T = vector<int>.
Теперь всё успешно скомпилируется, причем при первом вызове выведется

$$16; 18$$

а при втором

$$16\ 17\ 18$$

То есть если для какой-то шаблонной подстановки подходят несколько вариантов, выбирается textbfнаиболее подходящий. В этом примере всегда подходит несколько версий `show`, но выбирается всегда "более конкретная в некотором смысле".

Как мы убедились, SFINAE не проверяет **всю** функцию на компилируемость, а проверяет только её сигнатуру: типы аргументов и возвращаемый тип. Если там всё сошлось и функцию выбрали, то любая ошибка **в теле функции** уже будет ошибкой компиляции. Поэтому нельзя просто в теле функции объявить `iterator` и надеяться на лучшее.
Но выход есть. Давайте напишем класс, в который можно передать какой-то тип и во время компиляции понять, есть ли у него итератор. После чего на основании этой проверки можно будет выбирать тот или иной алгоритм, который внутри будет хорошо приспособлен к тому, что в него передают.

```
1  template <typename T>
2  struct has_iterator {
3    template <typename U>
4    static char test(typename U::iterator* x);
5    template <typename U>
6    static pair<char, char> test (U* x); // пара из двух чаров,чтобы
7    \\ размерточнобылотличнымот     sizeof(char).
8    static const bool value = sizeof(test<T>(nullptr)) == 1;
9  };
10
11 bool first = has_iterator<int>::value;
12 bool second = has_iterator<std::vector<int>>::value;
```

ШТА здесь происходит? Давайте разбираться.
В качестве шаблонного параметра в класс передается какой-то тип. В первом случае (строка 10) это `int`.
Сразу же вычисляется статическая переменная `has_iterator<T>::value`. Она равна `true` если размер значения, возвращаемого функцией `has_iterator<T>::test<T>` равен 1 байту и `false` – иначе.
Функция `has_iterator<T>::test<T>` получает указатель на `T`. Мы передаем в неё ноль, который неявно приводится к нужному типу (нулевой указатель – тоже указатель).
При подстановке `int*` в первую реализацию возникает ошибка, так как нет итератора. Поэтому выбирается вторая. Тип её возвращаемого значения `'long'` занимает 4 байта. `4 != 1`, поэтому в `value` присваивается `false`.
Если же мы передадим классу тип `std::vector<int>`, у него будет итератор. Значит для подстановки подходят оба варианта функции `test<T>`, из них выберется первая. Она возвращет `char`, который имеет размер 1 байт. `value` будет равно `true`.
Таким образом в переменной `first` будет `false` (действительно, `int` не имеет итератора), а в переменной `second` будет `true`. И всё это посчиталось в `compile time`.

4.18.2. Отступление про typename

В нашем коде была строчка

```
static char test(typename U::iterator* x);
```

Зачем здесь ключевое слово typename?
Если его не будет, компилятор запутается и выдаст ошибку, потому что когда он видит выражение U::iterator* x, он не знает, что представляет из себя U::iterator. Это может быть тип, тогда выражение задает указатель на этот тип. А может быть статической переменной, тогда '*' в этом выражении будет оператором умножения. Чтобы явно сказать компилятору, что это тип, надо написать ключевое слово typename.

4.18.3. enable_if

Нестрого говоря, enable_if – это метакласс в STL, который позволяет включать в исходник функцию только если выполнено условие.
На самом деле то, что он включает-выключает функцию – побочный эффект его использования вместе со SFINAE: если первый параметр true, то всё ок, иначе получаем кривое определение и функция выбрасывается по SFINAE. Рассмотрим ещё одну пару функций show.

```
1 template <typename T>
2 typename enable_if<!has_iterator<T>::value, void>::type show(const T& x) {
3   cout << x << endl ;
4 }
5 template <typename T>
6 typename enable_if<has_iterator<T>::value, void>::type show(const T& x) {
7   for (auto& i : x)
8     cout << i << endl;
9 }
```

Эта пара функций отличается от предыдущей тем, что у них одинаковая сигнатура, поэтому компилятор не может понять, какую реализацию надо включать. enable_if делает так, что первая функция компилируется только для тех типов, у которых нет итератора (для которых has_iterator<T>::value == false), а вторая только для тех, у которых есть. Далее в игру вступает SFINAE и выбирает то что нужно.
У enable_if есть два шаблонных параметра. Первый – это условие, туда можно передавать любое выражение типа bool. Второй параметр – это тип, который подставится вместо этой страшной строчки в случае успеха.
Как реализован enable_if:

```
1 template <bool B, class T>
2 struct enable_if {};
3
4 template <class T>
5 struct enable_if<true, T> {
6  typedef T type;
7 };
```

Всё очень просто. У этого класса есть общая реализация, у которой нет никакого enable_if::type. И есть специализация enable_if<true, T> для случая, когда условие истинно. У этой специализации внутри есть тайпдеф, который делает enable_if::type синонимом T, то есть того типа, который мы передали вторым параметром.
Когда мы в коде пишем enable_if<EXPR, TYPE>::type, то в случае когда EXPR == false, ничего не скомпилируется, а в случае когда EXPR == true, скомпилируется, и на это место подставится TYPE.

Ещё примерчик применения enable_if. Все мы знаем, что процессоры и системы имеют разные битности. В зависимости от этого бывает эффективно делать операции с тем или иным количеством битов одновременно. В следующем примере в зависимости от битности системы происходит копирование памяти по 32 или 64 бита:

```
1 struct is_64_bit {
2   static const bool value = sizeof(void*) == 8; // true если указатель занимает 64 бита
3 };
4
5 template <typename T = void>
6 typename enable_if<is_64_bit::value, T>::type
7           my_memcpy(void* target, const void* source, size_t n) {
8   cout << ”64 bit mem cpy” << endl;
9 }
10
11 template <typename T = void>
12 typename enable_if<!is_64_bit::value, T>::type
13           my_memcpy(void* target, const void* source, size_t n) {
14   cout << ”32 bit mem cpy” << endl;
15 }
```

Внимательный читатель удивится, зачем мы искусственно вводим шаблонный параметр T = void, если функция ничего возвращать не может? Дело в том, что SFINAE работает только с шаблонными функциями, поэтому приходится вставлять костыль.

4.18.4. type_traits

В заголовочном файле <type_traits> реализовано множество классов в стиле нашего has_iterator
Они называются трейты (traits). Их можно использовать в enable_if. Например это is_integral, is_pointer, is_copy_assignable, ...
Без них пришлось бы писать реализацию функции для каждого примитивного типа или один общий случай и несколько специальных реализация при помощи шаблонов.
enable_if и type_traits можно сделать реализацию для целой пачки типов, удовлетворяющих определенному условию (например отдельно для целых типов и отдельно для вещественных).

Дополнительную информацию можно посмотреть в
презентациях с лекций;
Костиных черновиках;
Васиных черновиках;
Практике про rvalue, move, forward и unique_ptr.