Информатика: семестр 3

Программирование на C++

Конспекты лекций

Лектор: Хирьянов Т.Ф.

21 ноября 2017 г.

Оглавление

Лекция 1

Введение в C++

1.1 Этапы компиляции

C++ — компилируемый язык. Перевод с человеческого языка на машинный код происходит не во время исполнения, а заранее. Сборка программы при этом проходит в несколько этапов:

1. Препроцессинг: my.cpp → my.cpp′
   Здесь происходит подстановка всех макросов, а акже удаление комментариев:

   Код 1.1: Пример макроса

   ```
   1 #include "mylib"
   2 #define A 123456
   ```

2. Компиляция: my.cpp′ → main.o
   Он содержит переведенную на машинный код инструкции, но все еще содержит сигнатуру функций. Это означает, что объектный файл может содержать названия функций, их входные данные, но не саму реализацию.

3. Компоновка (линковка): my.o → my.exe
   Здесь уже все файлы связываются и функции, не реализованные в п.3, получают реализацию. Программа может быть разделена на модули так, что разные файлы используют функции других файлов. При этом, если один файл ссылается на ресурсы других файлов, эти ресурсы необходимо объявить в отдельном файле: header–файле (хидер на жаргоне). Хидер содержит лишь сигнатуры функций и переменных, однако не входит в не конечный результат компиляции. Во время компановки эти хидеры будут показывать, где искать функции.
   Подобное деление лежит в основе принципа раздельной компиляции: программы делятся на компоненты, которые собираются по отдельности. Это уменьшает время компиляции.

1.2 Синтаксис

Код 1.2: Демонстрация синтаксиса

```
1 #include<iostream>
2 int main()
3 {
4   int x, y;
5   std::cin >> x;
6   std::cout << "x=" << x << "\n"; // "\n" === std::endl
7   << "y=" << y << std::endl;
8   return 0;
9 }
```

В языке C++ есть особенности, отличающие его от Python. Так, при создании переменной необходимо явно указать ее тип (строка 4).
В первой строке — подключение библиотеки для работы ввода/вывода.
В строке 5 стоит оператор ввода из стандартного ввода (это консоль) и спец. оператор >>. Результат считывания уходит в переменную x. std — это пространство имен, о них будет сказано позже.
В строке 6 — вывод, по аналогии с вводом. При этом строки можно выводить конвейером (как видно ‘cout << "x="<< x‘ — эти строки будут сшиты при выводе).
В C++ табуляция не имеет значения, перевод строки после (6) и дальнейшие команды показывают это.
Строка 8 содержит код возврата — он будет возвращен по окончании программы (0 означает успех).

1.3 Арифметические операции и сравнение

x = y — арифметическая операция (не присваивание, как в питоне). Можно писать такой код

Код 1.3: Неясное поведение

```
1 y=(x=3)+(z=5)
```

В коде выше все работает, однако такое поведение программы будет неочевидным. В C++ все еще работают x += 1. Помимо этого имеются два оператора: x++ ≈ ++x ≈ x+=1. Разница следующая: y = (x++) запишет в y **старый** x, в то время как в y=(++x) будет записан **новый** x. Операции ** (возведение в степень) *нет*. Примеры некоторых арифметических действий:

1. x*y — при применении к целым числам результат будет целым числом.

2. x/y — при применении к целым числам будет применено целочисленное деление. Проблему можно решить, если добавить в конце ".". Так, 2/3 станет 2/3. — точка превратит число в вещественное.

3. x > y, x >= y, x == y. Это тоже арифметические операции, возвращают, правда, значение типа *bool*: **true** или **false**.

1.4 Циклы, условия и конструкции

В C++ три вида циклов, их синтаксис отличается от Python:

Код 1.4: Виды циклов

```
1 // 1 type
2 while(condition)
3 {
4 some code;
5 }
```

Условие надо обязательно кидать в скобки, иначе C++ не сможет отличить его от тела.

Код 1.5: Виды циклов

```
1 // 2 type
2 while(condition)
3 {
4 if(condition)
5 {
6 std::cout << "Exiting..\n";
7 break;
8 }
```

Здесь мы выходим из цикла по команде **break**.

Код 1.6: Виды циклов

```
1 // 3 type
2 do {
3 ...
4 } while(condition);
```

Как обычный while, но сначала выполняет тело цикла, а затем проверяет условие.

Код 1.7: Виды циклов

```
// 4 type
for(op1;condition;op3)
{
some code;
}
// EQUAL:
op1;
while(condition)
{
some code;
op3;
}
// Example:
for(i=0; i < 10; i++)
{
some code;
}
```

Это очень похоже на то, что в питоне было под видом

Код 1.8: Аналог цикла for в Python

```
 for i in range(0, 10, 1):
     some code
```

Переменные существуют локально, поэтому можно переменную i сделать только для цикла. Более того, переменная "живет"до тех пор, пока не будет завершен соответствующий блок:

Код 1.9: Виды циклов

```
for(int i=0; i < 10; i++)   // "i" won't be seen outside
{
some code;
}

// else === elif
if(condition)
{
..
} else if(condition2)
{
..
}
else {
..
}
```

1.5 Тернарный оператор

Тернарный оператор заменяет условия (иногда полезно):

Код 1.10: Пояснение к тернарному оператору

```
1  z = (a > 0) ? x : y;
2  // EQUALS TO:
3  if(a > 0)
4  {
5  z = x;
6  }
7  else
8  {
9  z = y;
10 }
```

Такой подход здорово уменьшает код, однако при неправильном использовании может сделать его нечитаемым.

1.6 Оператор switch

Код 1.11: Оператор switch

```
1  switch(x)
2  {
3    case 0: op1; // if x=0
4         op2;
5        break;
6    case 1: op1; // if x=1
7         op2;
8         break;
9    default: op1; // By default
10 }
```

break — обязательно в конце, так как **case 0** — это метка (такое поведение берет корни из Assembler) для перехода. Как только значение x будет подходящим, программа прыгнет в соответствующую метку, но что будет после выполнения основного кода? Дальше она просто пойдет по остальным, ведь метка не накладывает никаких условий для входа (она просто обозначает кусок программы). Именно поэтому необходим **break** — он завершает выполнение **switch**.

Лекция 2

Типы в C++

2.1 Строгая статическая типизация в C++

Проверка типов происходит в момент компиляции.

Код 2.1: Неправильное присваивание

```
1 int x;
2 x = "Hello"; // won't work
```

Помимо этого, присваивание является арифметической операцией (делает дело и возвращает число).

2.2 Представление целых чисел в C++

Существует три способа хранить целые отрицательные числа в памяти компьютера:

1. Прямой ход. Сделаем первый бит знаковым. 0 — знак минус, 1 — знак плюс. Недостатком будет то, что данное число можно интерпретировать как беззнаковое. Пример: $10010011_2 = -19_{10}$. Есть еще одна проблема. 0 трактуется дважды: $-0 = 0$

2. Обратный ход. Мы инвертируем исходное число: 10010011 $\rightarrow$ 11101100. Просто поменяли 0 на 1 и 1 на 0 (это называется побитовое отрицание).

3. Дополнительный код. Обратный код, к которому прибавлена единица. В чем profit: 11101101 $= -19$. Попробуем к нему прибавить число 31:

$_{+}$31	00011111
-19	11101101
12	00001100

Обратите внимание: здесь ушла единица, которая возникла слева при сложении

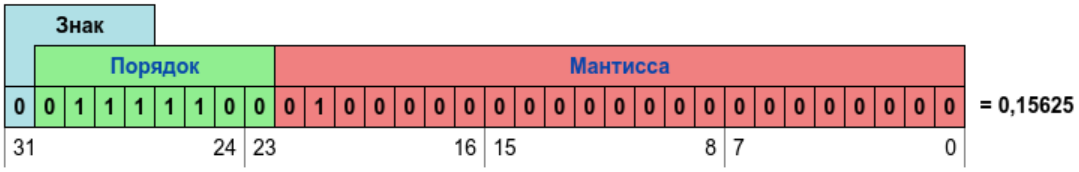
(от увеличения старшего разряда). В этом и заключается вся прелесть дополнительного кода: можно использовать тот же сумматор для двух чисел.

Помимо этого есть *тупоконечное* и *остроконечное* представление. Когда число сохраняется в память, его биты уходят в память в определенном порядке. В тупоконечном порядке сначала в памяти идут старшие разряды, т.е., если мы будем "читать"содержимое ОЗУ слева направо сверху вниз, то все будет идти в правильном порядке. В остроконечном порядке - наоборот. В компьютерах Intel, например, используется остроконечная система.

2.3 Хранение чисел с плавающей точкой

Мы можем условиться: первые биты брать как целую часть, дальше - дробную. Поскольку любое число можно представить бесконечной периодической двоичной дробью, мы сможем представить все числа.

IEEE 754



Стандарт для хранения чисел с плавающей точкой. Можно почитать здесь: вики.
Числа одинарной точности с плавающей запятой обеспечивают относительную точность 7-8 десятичных цифр в диапазоне от 10^{-38} до примерно 10^{38}. Для вычисления показателя степени из восьмиразрядного поля порядка вычитается смещение порядка равное $127_{10} = 7F_{16} = 01111111_2$ (то есть, $01111100_2 - 01111111_2 = 124_{10} - 127_{10} = -3_{10}$). Так как в нормализованной двоичной мантиссе целая часть всегда равна единице, то в поле мантиссы записывается только её дробная часть. Для вычисления мантиссы к единице добавляется дробная часть мантиссы из 23-х разрядного поля дробной части мантиссы $1,01000000000000000000000_2$. Число равно произведению мантиссы со знаком на двойку в степени порядка $= 1,01_2 * 2_{10}^{-3_{10}} = 101_2 * 2_{10}^{-5_{10}} = 5_{10} * 2_{10} - 5_{10} = 0,15625_{10}$.
(Взято из другой статьи)

2.4 Явное и неявное преобразование типов

Код 2.2: Неявное преобразование

```
1 double x;
2 int a = 2, b = 5;
```

```
3 x = b/a; // NOT DOUBLE without casting
4
5 x = double(b)/double(a); // explicit casting
6
7 x = b; // auto convert b to double, then will write to x
```

В коде выше используется неявное преобразование: дает возможность писать код гибче, однако требует осторожнее (так, в строке 3 используется *целочисленное* деление).

Код 2.3: Не всегда удается преобразовать типы

```
1 std::string s;
2 s = std::string(x); //won't work
```

2.5 Типы целых чисел

В C++ есть несколько типов целых чисел:

1. **char** - код символа (по умолчанию 1 байт)
2. **int** - целое число (сколько байт - не известно).

Не стоит быть уверенным, что **char** - 1 байт, на разных процессорах по-разному. В C++ так устроено: язык заточен под процессор, на котором он будет выполняться. Для обоих типов можно указать **signed** и **unsigned** - знаковый тип или нет.

Код 2.4: Пример доп. указателей к типам

```
1  signed int x;
2  unsigned int y;
3  long int z;
4  // size
5  sizeof(int); // about 4
6  long int // 4 bytes, too
7  unsigned long long int // 8 bytes now
8  short int // 2 bytes, wow
9
10 int8_t x8;
11 int16_t x16;
12 int32_t x32;
13 int64_t x64;
14 // They all has fixed size, dude
```

. Стоит быть внимательным: сравнение для знаковых и беззнаковых может пойти не так, как хотелось бы. Пример: $-1 > 50$ - верное равенство, если -1 имеет тип **signed**, а 50 - **unsigned**. Проблема заключается в том, что -1 соответствует максимальному числу из **unsigned**.

2.6 Логические операции

Работают над переменными типа **bool**.

не	$!A$
и	$A\&\&B$
или	$A\|\|B$

Здесь почти то же самое, что в Python, лишь синтаксис другой.

2.7 Битовые операции C++

Как можно поменять две переменные без третьей? Для этого могут помочь логические операции. В C++ их несколько:

1. Инверсия – x – работает над *любым* числом, не обязательно **bool**. Заменяет все ноли и единички в представлении числа на противоположные.
2. Битовое **"и"** – $x\&y$ – умножает биты в двоичном представлении двух чисел. Иногда битовое "и"называют *наложением маски*.
3. Битовое **"или"** – $x|y$ – складывает биты в двоичном представлении двух чисел.
4. xor - x∧y. С его помощью можно как раз осуществить обмен переменных:
   x = x ∧ y
   y = x ∧ y
   x = x ∧ y

2.8 Автоматические типы переменных

Код 2.5: Автоматический тип переменной

```
#include<iostream>

int get_x(); // 'auto x' will be int

double get_x(); // now 'auto x' will be double
// and so on...

int main()
{
  auto x = get_x();
  return 0;
}
```

Код будет присваивать переменной x различные типы, однако этот трюк будет работать, только если компилятору сказать использовать современный стандарт (например, C++11)!

2.9 Простейшие массивы

В C++, как и во многих языках, существуют массивы.

Код 2.6: Пример объявления простейшего массива

```
1  const int N_max = 1000;
2
3  int main()
4  {
5    int A[N_max];
6    int B[N_max] = {1, 2, 3};
7
8    // Let's iterate!
9    for(int i = 0; i < N_max; i++)
10     A[i] = i*i;
11
12 }
```

При объявлении массива надо указать его размер. По массиву можно итерироваться через обращение к элементу (точно так же, как в Питоне): A[1] – второй элемент и так далее; нумерация идет с *нуля*.
Подробнее о массивах будет рассказано на следующей лекции.

Исходные коды всех программ можно найти здесь

Лекция 3

Адреса и указатели

3.1 Доступ к памяти

Память, выделяемая программе, неоднородна – она делится на **сегменты**. Первый сегмент — коды программы (режим "только чтение"). Второй – сегмент DATA – содержит данные программы. Более того, элементы DATA (к примеру, начальные значения переменных) записывается в .exe файл. Эти данные уходят в оперативку при запуске программы. Переменные, записанные в DATA, называются *статическими*. Помимо этого, для функционирования программы необходима область *стека* — такой сегмент памяти, который заполнятся снизу вверх и изначально пуст. Когда ОС только загрузилась, в стек выделяется некоторое количество автоматической памяти – для *локальных переменных*.
Память программе выделяется на время ее работы; после ее завершения память высвобождается.

Код 3.1: Пример многократных вызовов функции

```
1  #include<iostream>
2
3  using namespace std;
4
5  void count(int n)
6  {
7   int k = 1;
8   while(k <= n)
9   {
10   cout << k << " ";
11   k++;
12  }
13  cout << "\n";
14 }
15
16 int main()
```

```
17 {
18  count(5);
19  count(10);
20 }
```

Рассмотрим, как происходит работа с памятью в примере 12.1. Перед вызовом функции в стек программы будет выделено место для хранения "5"(оно в программе будет интерпретировано как n), stack pointer поднимается. stack pointer - это указатель, показывающий на текущее положение стека. Затем, после вызова функции, ее адрес возврата будет записан в стек - stack pointer двинется выше. Потом в стек уйдет "1"(мы же объявили ее).
Попробуем написать функцию, считающую количество рекурсивных вызовов.

Код 3.2: Функция для подсчета количества собственных вызовов

```
1  #include<iostream>
2
3  using namespace std;
4
5  int k2 = 0;
6
7  void count2()
8  {
9   k2++;
10  cout << k2 << " ";
11 }
12
13 void reducer()
14 {
15  k2--;
16 }
17
18 int main()
19 {
20  count2();
21  reducer();
22  // any your staff ..
23 }
```

Здесь k2 — глобальная переменная. Ее присваивание происходит в *момент компиляции* программы (т.е. она запишется в .exe и будет переноситься в память в момент загрузки). Функция **reducer** уменьшает значение k2.
Однако глобальные переменные могут быть опасны! Считается, что нормальная функция не заглядывает в глобальные переменные (в таких случаях функцию называют "чистой в противном - "грязной").

3.2 Модификатор static

Код 3.3: Та же программа через static

```
#include<iostream>

using namespace std;

//int k2 = 0;

void count2()
{
 static int k3 = 0;
 k3 ++ ;
 cout << k3 << endl;
}

void reducer()
{
 k3--; // won't work, k3 is not defined in this block
}

int main()
{
 count2();
 reducer();
 // any your staff ..
}

// Recursive func sample
void count(int n)
{
 int k = 1;
 if(n > 0)
  count(n-1);
}
```

Мы переделали программу 12.2, теперь переменная k3 не видна в других блоках, но по-прежнему она записывается в сам .exe . Если бы функция была рекурсивной, то была бы такая ситуация, когда переменных k было бы много - на каждом слое рекурсии она создается. Они просто будут записаны в разные ячейки памяти и уйдут после того, как рекурсивный вызов завершится.

Код 3.4: Подсчет рекурсии

```
#include<iostream>

using namespace std;
```

```
4
5  //int k2 = 0;
6
7  void count2()
8  {
9   static int k3 = 0;
10  k3++ ;
11  cout << k3 << endl;
12  if(n > 0)
13   count2(n-1)
14  k3--;
15 }
16
17 int main()
18 {
19  count2();
20  // any your staff ..
21 }
```

В коде выше как раз **static** делает k одним для всех вызовов рекурсии — оно при каждом вызове не будет присваиваться в ноль. Вместо этого переменная будет увеличиваться на 1, таким образом, мы сможем увидеть глубину рекурсии.

3.3 Указатель

Указатель - переменная адресного типа.

Код 3.5: Адрес x можно указать

```
1 int x;
2 &x; // pointer to x
3     // it has type 'int*'
4 int* p = &x;
5 double* pd = &x; // won't work
```

Указатели тоже работают только со своими типами: в коде выше нельзя в **double*** записать число **int**. Дело в том, что возникнет ложная интерпретация – ссылка на double превратися в ссылку на int, а поскольку в памяти double и int имеют разные размеры, то можно даже вылезти за пределы своей памяти. Компилятор это запретит, однако все можно обойти:

Код 3.6: Пример обхода заперта

```
1 int x;
2 double* pd = (double*) &x;
```

```
3
4 int main()
5 {
6 return 0;
7 }
```

У указателей могут тоже быть указатели:

Код 3.7: Указатель на указатель

```
1 int x;
2 int* p = &x;
3 int** pp = &p;
4 int*** ppp = &pp;
5 // ...
```

Обратиться к переменной можно несколько способами:

Код 3.8: Разница

```
1 int x;
2 int* p = &x;
3 cout << x; // print x
4 cout << p; // print ADDRESS of x
5 cout << *p; // print x
```

В последней строке происходит *разыменование* адреса - грубо говоря, считывание данных из того адреса, куда указывает указатель. Разыменовывать надо последовательно:

Код 3.9: Двойное разыменование

```
1 cout << x;
2 cout << *p;
3 cout << **pp;
4 cout << ***ppp;
```

3.4 Несколько слов о массивах

Массивов в C++ нет. Массив - это константный указатель на область памяти, представляющую содержимое!

Код 3.10: Работа массива

```
1 int A[5] = {0, 1, 2, 3, 4};
2 int* p = A; // works!
3 *p = 7
4 p + 1 // magic, read in text
5
6 // Furthermore:
7 A[i] === *(A+i)
```

Команда объявления массива по факту означает следующее: создать n переменных типа int.

С указателями можно прийти к *адресной арифметике*: так, указатель $p+1$ выше означает, что мы сдвигаемся по памяти от места, куда указывает p, на **sizeof(1)**. Это иллюстрирует строка 7: A[i] абсолютно равносильна указателю *(A+i). Проверки на i *нет*, то есть, в теории, мы можем убежать вообще в произвольную область памяти.

3.5 Динамическая память

Код 3.11: Пример выделения из дин. памяти

```
1  int get()
2  {
3   int* p = new int[1000]
4   return p;
5  }
6
7  int main()
8  {
9   int* q;
10  q = get();
11  // ...
12  // after your code
13  q = get(); // takes another amount of mem
14  delete[] q;
15 }
```

Динамическая память не является автоматической, т.е. ей надо явным образом указывать, когда ее выделять, а когда удалять. Если без конца писать вызов get(), то можно постоянно выбивать у системы память и это все приведет к *memory leak* - утечке памяти.

3.6 Передача параметров по указателю

Передача по параметрам бывает полезна, когда хочется поменять значение переменной внутри самой функции. Функции передается указатель и она, имея ссылку непо-

средственно на область памяти, может ее менять:

Код 3.12: Передача указателей функции

```
1  void changer(int* x)
2  {
3   (*x)++;
4  }
5
6  // You can use another construction in C++
7
8  void changer1(int& y)
9  {
10     y++;
11 }
12
13
14 int main()
15 {
16  int x;
17  changer(&x);
18  changer1(x); // no need for &
19  cout << x;
20 }
```

Подробнее – в следующей лекции.

Лекция 4

Динамическая память. Продолжение.

4.1 Динамические массивы

Как было сказано, массивов в C++ нет. Повторим объявление указателей:

Код 4.1: Пример объявления указателей

```
1 double A[100]; // has type `const double*`
2 double* p = A;
3 //or
4 double* p1 = &A[5]
5 //equals to
6 double* p1 = A + 5
7
8 double* p = new double(5.);
```

В строке **4** мы получаем указатель на **A[5]**. В строке **8** мы создаем число, да и еще присваиваем ему значение.

Код 4.2: Объявление массива через **new**

```
1 double* p = new double[50];
2
3 for(int k = 0; k < 50; k++)
4   p[k] = k*k;
5
6 delete[] p;
```

При объявлении массива через **new** нужно обязательно его удалять через **delete[]** – оператор удаления.

4.2 Структуры в C++

В C++ существует такая вещь, как структура – набор переменных под одной крышей.

Код 4.3: Простейшая структура

```
1  #include<iostream>
2
3  struct Student {
4      int8_t age;
5      int16_t group;
6      std::string name;
7  };
8
9  void student_print(Student x) {
10     std::cout << x.name << " " << x.age
11     << " " << x.group << "\n";
12 }
13
14 int main()
15 {
16     Student vasya = {16, 649, "Vasilii Ivanov"};
17     Student masha = {18, 649, "Maria Vasilyeva"};
18
19     Student *p = nullptr;
20
21     p = &vasya;
22
23     std::cout << (*p).name << "\n";
24     // This code is equal to
25     std::cout << p -> name << "\n";
26     student_print(masha);
27
28     return 0;
29 }
30
31 void studentPrint1(Student *x) {
32     std::cout << x->name << " " << x->age
33     << " " << x->group << "\n";
34 }
35
36 void studentPrint2(const Student &x) {
37     std::cout << x.name << " " << x.age
38     << " " << x.group << "\n";
39 }
```

Обратите внимание на строки **23** и **25**: такое сокращение еще часто будет встречаться (для указателей вместо "."используется "− >"). Метод для распечатки мы вынесли

отдельно для удобства.
Подход к распечатке в самой первой функции имеет минус: при вызове функции мы создаем локальную копию Student. Метод **studentPrint1** лишен этого – там передается по ссылке. Но возникает проблема, ведь мы внутри функции случайно можем изменить переменную x и этот результат будет виден извне. Решение дает передача по константной ссылке, что сделано в **studentPrint2**.
Существует еще одно решение задачи, в котором мы сначала создаем объект Student в памяти, получаем на него указатель, а затем через него меняем свойства объекта:

Код 4.4: Еще одно решение со структурой

```
#include<iostream>

struct Student {
    int8_t age;
    int16_t group;
    std::string name;
};

void studentPrint(const Student &x);

int main()
{
    Student *p_student = new Student;
    p_student->age = 17;
    p_student->group = 649;
    p_student->name = "Vasya Ivanov";

    studentPrint(*p_student);
    return 0;
}

void studentPrint(const Student &x) {
    std::cout << x.name << " " << x.age
    << " " << x.group << "\n";
}
```

Из структур можно составлять массивы, что неудивительно, учитывая, что массив - это просто несколько переменных. В качестве примера можно привести следующий код:

Код 4.5: Массив из структур

```
#include<iostream>

struct Student {
    int8_t age;
    int16_t group;
    std::string name;
};

void studentPrint(const Student &x);

int group_max_size = 10;
int main()
{
    Student *my_group = new Student[group_max_size];
    for(int k = 0; k < group_max_size; k++)
    {
        my_group[k].age = 17 + k;
        my_group[k].group = 600 + k;
        my_group[k].name = "Vasya Ivanov";
    }

    for(int k = 0; k < group_max_size; k++)
    {
        studentPrint(my_group[k]);
    }
    return 0;
}

void studentPrint(const Student &x) {
    std::cout << x.name << " " << x.age
    << " " << x.group << "\n";
}
```

4.3 Массивы и функции

В функциях можно менять массивы, но их нельзя возвращать. Код внизу – пример изменения массива внутри функции.

Код 4.6: Изменение массива внутри функции

```
#include<iostream>

void squarer(int* B, const int& B_size)
{
    for(int k = 0; k < B_size; k++) {
        B[k] = B[k] * B[k];
    }
}

int main()
{
    int A[5];
    squarer(A, 5);
    return 0;
}
```

4.4 Двумерный массив и передача его функции

Двумерные массивы работают так же, как в Питоне. Тем не менее, у них есть некоторые особенности, связанные с их способом представления в памяти (помним, что массив по факту лишь набор переменных). Одна из этих особенностей видна в коде ниже, где мы передаем двумерный массив в функцию:

Код 4.7: Передача массива в функцию

```
#include<iostream>

//void squarer(A);

void printer(double A[][4])
{
    for(int k = 0;k < 3; k++)
    {
        for(int j = 0; j < 4; j++)
            std::cout << A[k][j] << '\t';
        std::cout << '\n';
    }
}

int main()
{
    double A[3][4];
    for(int k = 0;k < 3; k++)
        for(int j = 0; j < 4; j++)
            A[k][j] = 1 + k*4 + j;
```

```
21
22      printer(A);
23
24      return 0;
25  }
```

Обратите внимание: в объявлении функции **printer** мы обязательно указываем длину (вторая квадратная скобка).

4.5 Статические массивы

Код 4.8: Указание на элемент массива

```
1  A[i][j] == *(A+i*y+j)
```

В коде выше видно, что для прохождения по элементам массива в функции необходимо заранее знать его длину. Если посмотреть на выражение 4.8, то станет видно, что эта длина уходит в инструкции прохода по массиву еще на этапе компиляции, поэтому ее и нужно было передавать заранее, как это сделано в 4.7. Но можно ли сделать функцию **printer** без явного указания длины массива? Ответ дают динамические массивы.

4.6 Динамические массивы

Динамические массивы — это массивы, хранящиеся в так называемой динамической памяти (*heap*). Динамические массивы объявлять несколько сложнее: так, для двумерного случая необходимо делать указатель на область памяти с указателями, каждый указатель в которой, в свою очередь, будет указывать на область памяти с нашими объектами (к примеру, **int**). Звучит не очень, однако пример 4.10 может немного прояснить этот момент.

Код 4.9: Объявление динамического массива

```
1  #include<iostream>
2
3  //void squarer(A);
4
5  // TODO: func must take widh and height, fix it
6  void printer(double A[][4])
7  {
8      for(int k = 0;k < 3; k++)
9      {
10         for(int j = 0; j < 4; j++)
11             std::cout << A[k][j] << '\t';
```

```
12          std::cout << '\n';
13      }
14  }
15
16  int main()
17  {
18      int width = 4, height = 3;
19      double **A = nullptr;
20      A = new double*[height];
21      for(int k = 0; k < height; k++)
22          A[k] = new double[width];
23
24      for(int k = 0; k < height; k++)
25          for(int j = 0; j < width; j++)
26              A[k][j] = 1 + k*width + j;
27
28      printer(A);
29
30      for(int k = 0; k < height; k++)
31          delete[] A[k];
32      delete A;
33
34      return 0;
35  }
```

A – это массив из указателей типа **double***. Каждый элемент этого массива - массив размера **width**. Таким образом, **A** есть двумерный массив. Обращение к его элементам происходит так же, как и при обычном объявлении массива, что видно в строке **26**. Мы сделали массив, распечатали его, а затем удалили, чтобы освободить память. При этом следует помнить такой трюк:

Код 4.10: Порядок разыменования

```
1  A[k][j] == *(*(A + k) + j)
```

Исходные коды всех программ можно найти здесь

Лекция 5

Строки и файлы в C++

5.1 Хранение строк

Строки делятся на *C-строки* и *ANSI-строки*. В разных языках строки реализованы по-разному. Так, в Pascal они хранятся как *ANSI-строки*: в первой ячейке памяти записана длина строки, а затем последовательно идут символы строки. Главный минус такого подхода — ограниченность длины строки (255 символов). Этот недостаток отсутствует в *C-строках*.

В *C-строках* используется следующая схема хранения: в памяти просто идет последовательность символов и завершается **нулем** (типа **int**, не "0"). Так, строка "Hello" будет записана как последовательность 'H', 'e', 'l', 'l', 'o', 0x0 (строки такого типа называются *zero-ending string*). НО! Если мы сделаем обращение и изменим какой-либо символ на 0 (типа целочисленное), то строка поделится на две, ведь теперь у нее два символа окончания. Это — один из минусов такого подхода.

5.2 Операции со строками

Конкатенация строки можно делать с помощью стандартной библиотеки **cstring**.

Код 5.1: Работа со строками

```
1  #include <cstring>
2  #include <iostream>
3
4  using namespace std;
5
6  int main()
7  {
8   char s1[10000], s2[1000];
9   s1[0] = 0;
10  std::cin.getline(s2, 1000);
11  strcat(s1, s2); // concatanate
```

```
12 }
```

В этом коде мы считываем строки, а затем их конкатенируем. Но Т.Ф. не любит эту библиотеку. Дело в том, что **strcat** сначала проходит и за линейное время находит ноль в конце строки, дабы узнать, где ее конец, а затем уже объединяет обе. Если такой код засунуть в цикл, то (из свойств арифметической прогрессии) мы подарим нашему коду квадратичную асимптотику.

В MacOS сделано несколько интереснее: используется как число, равное длине строки, в ее начале, так и символ окончания в конце (комбинированный подход).

При работе со строками следует быть осторожными: при операции `+=` можно уйти за аллоцированную (выделенную) память для данной строки и стереть ее содержимое. Кстати, в C++ строка — это объект. Для нас сейчас это значит, что операции, которые нам кажутся простыми (наподобие `+=`), на деле такими не являются — они реализованы по уму. К тому же, у строк есть свои методы и поля. К примеру, метод **s.size()** вернет длину строки за $O(1)$. У строк автоматическая работа с памяти — язык автоматически выделяет столько памяти, сколько нужно.

Т.Ф. рекомендует пользоваться C++ – строками вместо C–строк.

5.3 C++ – строки

В C++ строки входят в окружение **std**.

Код 5.2: Объявление строки в C++

```
1 #include <string>
2 #include <iostream>
3
4 int main()
5 {
6  std::string s("Hello");
7  for(int k=0; k < s.size(); k++)
8   std::cout << s[k] << std::endl;
9 }
```

В Python строки кодируются стандартом UTF-8. В таком подходе память используется в разных количествах для разных символов, поэтому при вставке символа в середину строки возникают проблемы переноса (одна из причин, почему в Python строки неизменяемы). C++ лишен такого недостатка и здесь строку можно менять посимвольно.

C++ не разделяет типы символа и *кода* символа. Символ = код символа. Вообще, подробности можно изучить здесь.

5.4 Считывание строки

Существует несколько способов читать строки:

Код 5.3: Считывание строки

```
1  while(cin >> s) // (cin >> s) returns true or false
2  {
3   cont << s << '*';
4  }
5
6  // Another way
7  getline(cin, s)
```

Обратите внимание: **cin** ≫ **s** возвращает **true** или **false**, в зависимости от того, закончился ли ввод.

Другой способ заключался в методе **getline**, которому на вход поступает поток ввода (см. ниже) и строка s, куда будет записано. Различие следующее: **cin** читает до символа пробела, тогда как **getline** читает до символа переноса строки.

5.5 Файловые потоки

Стандартная библиотека для работы с потоками — это **fstream**. Она дает три потока для работы: **ifstream** для ввода и **ofstream** для вывода и **fstrim** для обоих вариантов.

Код 5.4: Пример работы с потоками

```
1  #include <fstream>
2  #include <iostream>
3
4  using namespace std;
5
6  void read_array(int *A, int A_size, string f_name)
7  {
8   ifstream fin(f_name); // input stream based on file
9   for(int k=0; k < A_size; k++)
10   fin >> A[k];
11 }
```

В коде выше было бы неплохо закрыть файл после окончания работы с ним. Более того, операционная система может дать отказ, если мы затребуем открытие слишком большого количества файлов. Однако C++ умный, он закроет файл после окончания существование объекта (в ходе выполнения деструктора, о них будет сказано в следующих лекциях).

Во время открытия потока можно указать для него *режим*: только для чтения и т.п.

Код 5.5: Режимы потока

```
fstream f("1.txt", "rwt")
```

Подробнее о режимах можно почитать в документации.

5.6 Строковые потоки

Работа со строковыми потоками тоже проходит через стандартную библиотеку:

Код 5.6: Строковые потоки

```
#include <sstream>

using namespace std;

int main()
{
 std::stringstream t;
 t << "x = " << 7 << "." << true << std::endl; // any type is compatable
 string s;
 s = sout.str(); // cast stream to string
 return 0;
}
```

5.7 Односвязный список

Попробуем сделать односвязный список в C++ по аналогии с тем, что мы делали в Python.

Код 5.7: Реализация односвязного списка

```
#include <iostream>

using namespace std;

typedef std::string DataType;

struct tNode
{
    DataType data;
    tNode* next = nullptr;
};

tNode * insert_node(tNode *p_begin, DataType data);
```

```
14
15 int main()
16 {
17
18     tNode* p = new tNode;
19     p->data = "Hello";
20     // Sequently add nodes
21     p = insert_node(p, "World");
22     p = insert_node(p, "Reverse order");
23
24     while(p) {
25         std::cout << p->data << std::endl;
26         p = p->next;
27     }
28
29     return 0;
30 }
31
32 tNode * insert_node(tNode *p_begin, DataType data)
33 {
34     tNode *p = new tNode;
35
36     p->data = data;
37     p->next = p_begin;
38     return p;
39 }
```

1. **typedef** — это создание синонима типа **string**. Это сделано только для того, чтобы не писать каждый раз **std::string**, другого назначения у такой конструкции для нас нет.

2. **struct tNode** — объявление структуры ноды. В ней всего две переменные: свое значение и ссылка на следующую ноду.

3. **tNode * ...** — сигнатура функции *insert_node*. Ее мы опишем позже.

4. В методе **main** мы создаем ноду через **new**, затем сохраняем ссылку на созданный объект. После этого мы записываем туда какую-то информацию ("Hello"в примере), далее прикрепляем к ней другие ноды.

5. В цикле мы прогуливаемся по всем нодам и выводим их содержимое. Обратите внимание на переход к следующей ноде в последней строке цикла.

6. Функция **insert_node** — изюминка нашего кода. Ей передается указатель на нашу ноду. Внутри функции мы создаем новую ноду, заполняем в нее данные из **data**, а в качестве следующей ноды записываем ту, что мы передали функции.

Из этого получается, что мы добавляем ноды в обратном порядке, так как только что созданная нода встает впереди старой.

Исходные коды всех программ можно найти здесь

Лекция 6

Продвинутая работа с функциями

6.1 Перегрузка (overload) функций

Полиморфизм — много разных функций с одним именем. При вызове будет выбираться только одна: та, которая лучше всего подходит по входным параметрам. Это означает, что у функции один *интерфейс* и много *реализаций*. Рассмотрим пример:

Код 6.1: Пример полиморфизма

```
1  #include <iostream>
2
3  int absolute(int x) // for int
4  {
5      std::cout << "Call for int" << "\n";
6      if(x < 0)
7          return -x;
8      else
9          return x;
10 }
11
12 double absolute(double x)   // for double
13 {
14     std::cout << "Call for double" << "\n";
15     return ((x > 0) ? x : -x);
16 }
17
18 long absolute(long x)    // for long
19 {
20     std::cout << "Call for long" << "\n";
21     return (x > 0) ? x : -x;
22 }
23
24 int main()
25 {
26     int a = -5;
```

```
27      double b = -5.1;
28      long c = -1L;
29
30      std::cout << absolute(a) << std::endl;
31      std::cout << absolute(b) << std::endl;  // different types
32      std::cout << absolute(c) << std::endl;
33  }
```

В примере 12.1 мы пишем одну функцию **absolute** три раза: по одному на каждый тип входного (и, соответственно, возвращаемого) объекта. В случае, когда для заданного типа данных нет подходящей функции, C++ отбирает несколько функций как кандидатов на вызов, а затем выбирает их в следующем порядке:

1. Точное совпадение типов, если оно есть.
2. Приведение путем повышения разрядности числа и вызов кандидата с полученным типом (например, **int8** → **int16**)
3. Стандартное приведение типов (например, **float** → **int**). Оно уже менее надежно, поэтому и находится ниже по приоритету.
4. Пользовательское приведение типов, т.е. приведение к типу, объявленному пользователем (программистом, написавшим тип).
5. Эллипсис. В C++ существуют функции, у которых на вход поступает неопределенное число параметров – они называются эллипсис.

Код 6.2: Пример эллипсиса

```
1  void printf(char *s, ...)
```

Если на какой-то стадии проверки будет несколько кандидатов, равноправно удовлетворяющих условиям, то компилятор выдаст ошибку (он не поймет, какую вызывать).

6.2 Шаблоны

Хотелось бы, чтобы можно было делать функцию с неопределенным типом, как здесь:

Код 6.3: Неопределенный тип

```
#include <iostream>

T absolute(T x)     // Will not work
{
    return (x > 0) ? x : -x;
}

int main()
{
    int a = -5;
    double b = -5.1;
    long c = -1L;

    std::cout << absolute(a) << std::endl;
    std::cout << absolute(b) << std::endl;  // different types
    std::cout << absolute(c) << std::endl;
}
```

Однако C++ не позволит этого сделать. Дело в том, что при компиляции подобного кода необходимо под каждый тип переменной, для которой вызывается функция, иметь функцию в соответствии с *шаблоном*. Исправим код:

Код 6.4: Неопределенный тип с шаблоном

```
#include <iostream>

template<typename T>

T absolute(T x)     // That is fine
{
    return (x > 0) ? x : -x;
}

int main()
{
    int a = -5;
    double b = -5.1;
    long c = -1L;

    std::cout << absolute(a) << std::endl;
    std::cout << absolute(b) << std::endl;  // different types
    std::cout << absolute(c) << std::endl;
}
```

Тут может возникнуть проблема. Если бы мы вызвали **absolute** от строки, то во время компиляции, при *инстанцировании* функции для типа **string**, возникнет ошибка

– у **string** нет оператора унарного минуса (т.е. нельзя для строки s делать операцию ‘-s‘). Эту проблему можно решить, если явно указать, каким типом инстанцировать функцию:

Код 6.5: Явное инстанцирование функции

```
#include <iostream>

template<typename T>

T absolute(T x)     // That is fine
{
    return (x > 0) ? x : -x;
}

int main()
{
    std::cout << absolute<int32_t>(-8.0) << std::endl;
}
```

Здесь мы изменили вызов на **absolute<int32_t>**, тем самым указав тип, которым будет инстанцирована функция при компиляции.

Посмотрим, как ведет себя шаблон:

Код 6.6: Сложное инстанцирование

```
#include <iostream>

template<int number>
void self_counter()
{
    static int32_t counter = 0;
    counter++;
    std::cout << "Was called: " << counter << " times";
}

int main()
{
    //self_counter(); // won't work
    self_counter<1>();
    self_counter<1>();
    self_counter<1>();
    self_counter<2>();
    self_counter<2>();
    return 0;
}
```

В коде функция **self_counter<1>** будет вызвана 3 раза, а **self_counter<2>** – только 2. Дело в том, что мы функцию инстанцируем по-разному: сначала с 1, потом с 2 (как будто это разные функции). Простой код **self_counter()** в этом случае уже не работает, так как для инстанцирования есть несколько подходящих кандидатов (см. 6.1).

Код 6.7: Инстанцирование с параметром по умолчанию

```
#include <iostream>

template<int number>
void self_counter()
{
    static int32_t counter = 0;
    counter++;
    std::cout << "Was called: " << counter << " times";
}

template<>
void self_counter<0>;

int main()
{
    //self_counter(); // won't work
    self_counter<1>();
    self_counter<1>();
    self_counter<1>();
    self_counter<2>();
    self_counter<2>();
    self_counter(); // Now without instansiate
    return 0;
}
```

. Пример выше может не работать, Т.Ф. обещает рассказать об этом через несколько лекций.

6.3 Аргументы функции по умолчанию

В C++ можно создавать функции, у которых будут значения по умолчанию:

Код 6.8: Функция с параметрами по умолчанию

```
void foo(int x = 0, int y = 0, int z = 0)
{
    std::cout << x << y << z << std::endl;
}
// Can be called in these ways
foo()                 // 000
```

```
foo(1)              //100
foo(1, 2)          // 120
foo(1, 2, 3)     // 123
```

Единственная тонкость: значение по умолчанию пишется в *реализации* функции, а не в ее объявлении.

6.4 Односвязный список для любого типа данных

Теперь мы можем переписать **LinkedList** из прошлой лекции:

Код 6.9: LinkedList

```
#include <iostream>
#include <string>

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next = nullptr;
};

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data)
{
    tNode<DataType> *p = new tNode<DataType>;
    p->data = data;
    p->next = p_begin;
    return p;
}

int main()
{
    tNode<std::string> *p_begin = nullptr;

    p_begin = insert_node(p_begin, std::string("Hello"));
    p_begin = insert_node(p_begin, std::string("World"));
    return 0;
}
```

Эту реализацию LinkedList можно несколько усложнить, разделив на файлы: один будет хранить лишь LinkedList, а другой будем пользоваться готовым кодом из первого. Получится что-то вроде библиотеки для Linked List. Часть ее реализации представлена ниже. Код далее достаточно сложен, и многие его аспекты будут объяснены в следующих лекциях.

Код 6.10: Основная часть

```cpp
#include <iostream>
#include <string>
#include "linkedList.hpp"

int main()
{
    LinkedList<std::string> str_list;
   // LinkedList<int32_t> int_list; // Not used now
    list_init(str_list);

    list_insert(str_list, std::string("World"));
    list_insert(str_list, std::string("Hello"));
    list_insert(str_list, std::string("First"));
    list_print(str_list);

    list_destroy(str_list);

    return 0;
}
```

Код 6.11: Хидер

```cpp
// Added module

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next;
};

void list_init(List<T>);

void list_insert(List<T>, const T &value);

void list_print(List<T>);

void list_destroy(List<T>);

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data);
```

Исходные коды всех программ можно найти здесь

Лекция 7

Контейнеры в C++

7.1 LinkedList

Продолжим работу над кодом из предыдущей лекции:

Код 7.1: LinkedList

```cpp
#include <iostream>
#include <string>
#include "linkedList.hpp"

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next = nullptr;
};

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data)
{
    tNode<DataType> *p = new tNode<DataType>;
    p->data = data;
    p->next = p_begin;
    return p;
}

int main()
{
    tNode<std::string> *p_begin = nullptr;

    p_begin = insert_node(p_begin, std::string("Hello"));
    p_begin = insert_node(p_begin, std::string("World"));
    return 0;
}
```

Код 7.2: Header-файл для нашей программы

```
// Added module

template<typename DataType>
struct tNode
{
    DataType data;
    tNode* next;
};

template<typename DataType>
struct List{
    tNode<DataType> *begin;
}

template<typename DataType>
void list_init(List<T>);

template<typename DataType>
void list_insert(List<T>, const T &value);

template<typename DataType>
void list_print(List<T>);

template<typename DataType>
void list_destroy(List<T>);

template<typename DataType>
tNode<DataType> * insert_node(tNode<DataType> *p_begin, DataType data);
```

Код заполнен не до конца, я не смог поспевать за Т.Ф. (если у кого будут исправления, прошу написать мне).

ОБНОВЛЕНИЕ. Код всего проекта можно смотреть здесь.

7.2 Библиотека стандартных шаблонов C++

Библиотека стандартных шаблонов C++ называется **STL**. Она состоит из нескольких частей и типов, но выдерживает следующую структуру:

1. Контейнеры – некоторый объект, содержащий данные.
2. Итераторы – позволяют *итерироваться по элементам контейнера.*
3. Алгоритмы – название само говорит о предназначении. Примечательно, что они не имеют привязки к конкретному типу контейнера.

Таблица (для наглядности)

Контейнеры	Итераторы	Алгоритмы
Типы контейнеров: 1. **vector<T>** (похож на **list** в Питоне) 2. **list<T>** (это **LinkedList** из Питона) 3. **set<T>** – красно-черное двоичное дерево поиска ($O(\log N)$ - добавление/удаление/поиск. 4. **unordered_set<T>** – хэш-таблица $O(1)$ – добавление/удаление/поиск 5. **map<T, V>** – ассоциативный массив, состоящий из пар *<key, value>*. При этом *key* типа **T**, *value* типа **V**	Типы итераторов: 1. input output – доступ к элементу содержимого контейнера по итератору 2. bidirectional −−i – если контейнер 'знает', что он двунаправленный 3. random access i = i + 100 – итератор произвольного доступа (а не только по +1) 4. forward ++i – обычный шаг на один вперед	1. см. ниже

Итерирование

Попробуем проитерироваться по контейнеру и применить к каждому элементу функцию **foobar**. Это можно сделать несколькими способами:

1.

Код 7.3: Пример итерирования

```
vector<int> A = {10, 3, 17, 28};

for(vector<int>::iterator i = A.begin(); i != A.end(); ++i)
{
    foobar(*i);
}
```

Здесь итератор **i** пробегает по всем значениям массива. Как он ведет учет текущей позиции, нас не волнует – в этом и есть прелесть подобной структуры. Начало итерирования – начало массива, конец итерирования – конец массива. Есть и защита: если попытаться выйти за A.end() (например, на единицу), то при попытке разыменовать такой адрес выйдет ошибка.

2. То же самое итерирование можно было провести с помощью **foreach** (из библиотеки **<algorithm>**)

Код 7.4: Применение алгоритмов для итерирования

```
#include <algorithm>

int main()
{
    vector<int> A = {10, 9, 15, 7, 20};
    for_each(A.begin(), A.end(), foobar);
}
```

Обратите внимание: функции **for_each** уходит *указатель* на функцию **foobar** (она ведь тоже хранится в памяти, как и переменные).

В стандарте C++ 11 использование контейнеров упрощено:

Код 7.5: Контейнеры в C++11

```
#include <algorithm>

int main()
{
    vector<int> A = {10, 9, 15, 7, 20};
    //for_each(A.begin(), A.end(), foobar);
    for(auto x : A) {
        x = x*2
        std::cout << x;
    }
}
```

Единственный минус: чтобы понять, что происходит в итераторе, приходится копать код вверх, до описания функции **foobar**.
В примере 12.5 используется **auto** – автоматический выбор типа. При таком подходе компилятор сам подбирает подходящий тип для переменной.

Лекция 8

Парадигмы программирования

8.1 Структурное программирование

Структурное программирование строится на следующей идее: разбивать сложную задачу на простые подзадачи. Достигается это путем решения подзадач в функциях. Таким подходом мы уже часто пользовались, но у него есть минус: со временем функций накапливается много и код становится менее читабельным.

8.2 Модульное программирование

В качестве решения указанной проблемы можно функции собрать в отдельные *модули*. Полученная парадигма носит название **модульного программирования**. Разумеется, при таком подход будет главный модуль – к примеру, main.cpp. Главный модуль имеет одну особенность: в main.cpp есть функция **main**, которая содержит вход в программу. Таким образом достигается иерархия модулей.

При построении взаимоотношений модулей очень важно правильно выстроить их зависимости. Нельзя допускать т.н. "межмодульную рекурсию": когда модуль **module1** инклюдит модуль **module2**, а модуль **module2** в то же время инклюдит модуль **module1**. Использование модулей заставляет думать об их организации.

Стоит понимать, что все парадигмы нужны лишь для удобной воспринимаемости кода, но сами по себе они никак не ускоряют работы программ (а иногда даже могут и замедлить).

8.3 Раздельная компиляция

Начнем с примера. Напишем структуру для дроби с возможностью использовать комплексные числа. Будем придерживаться следующей структуры:

1. **main.cpp** – единица трансляции (то есть, компилируется). Подключает все хидеры. Собирается в **main.o**
2. **fractions.hpp** – хидер для **fractions.cpp**
3. **fractions.cpp** – единица трансляции (то есть, компилируется). Вместе со своим хидером собирается в **fractions.o**
4. **frac_complex.hpp** – хидер для **frac_complex.cpp**
5. **frac_complex.cpp** – единица трансляции (то есть, компилируется). Подключает свой хидер **и** хидер **fractions.hpp** (потому что мы наверняка будем использовать его функции). При сборке соберется в **frac_complex.o**.

Код всего проекта можно смотреть здесь.

Помимо прочего, наш проект может иметь несколько *целей*: например, мы захотим собрать готовую программу или же тестовую; подключить файл-тестировщик или нет. Для этого применяется Makefile и система сборки **make**, однако она не будет обсуждаться в этой лекции. Если вас это заинтересовало, можете поискать в Гугле либо посмотреть исходники лекции, которую в данный момент читаете: она собрана с использованием **make**.

Разделяемые (динамические) библиотеки

Так называются библиотеки, которые содержатся в ОС. Распространяются в .dll в ОС Windows или же .so в Linux.

Отступление. Полезный материал на Фоксфорде

Подробности по C++ можно найти здесь (материал достаточно ознакомительный, но из него можно вынести много полезного).

8.4 Условная компиляция

Условная компиляция – компиляция с условиями. Условия компилятору сообщают *директивы* – команды, обрабатываемые на этапе *препроцессинга*. Часто используемая директива – это **#IFDEF**. Приведем пример ее использования:

Код 8.1: Директива IFDEF

```
1 #ifdef SOMETHING
2 some code
3 #else
4 other code
5 #endif
```

Код в someCode будет выполнен только тогда, когда был определен SOMETHING. Существует похожая директива: **#IFNDEF** – только он срабатывает, если *не* был определен аргументы.

Еще один т.н. "страж включения": **#define SOMETHING** – объявляет **SOMETHING**. После него **#ifdef** уже будет думать, что **SOMETHING** объявлено.

Все эти штуки используются для того, чтобы не инклюдить один и тот же файл по несколько раз, ведь в таких ситуациях компилятор будет кидать ошибки.

8.5 Макросы

Код 8.2: Пример использования макросов

```
1 #define max(a, b) (((a) > (b)) ? (a) : (b))
2
3 int main()
4 {
5  std::cout << max(5, 3) + 1;
6 }
```

Макросы не рекомендуется использовать – они создают обманчивую привлекательность, однако порождают непредвиденное поведение, поскольку **не** являются вызовами функции, а всего лишь подставляются препроцессором при текстовой обработке.

8.6 extern-переменные

Представим себе, что мы хотим создать переменную, которая будет доступна и в том, и другом модуле. Можно ее описать в **main.cpp**, однако тогда нельзя будет ей пользоваться в тех файлах, которые не подключают **main.cpp**.

У нас есть **fractions.cpp**, которого все подключают. В нее запихнем эту переменную, а в **fraction.hpp** запишем **extern int g;**. Это скажет компилятору, что переменная **g** будет, но стоит ее дожидаться ТОЛЬКО во время компоновки (а не создавать каждый раз при **include**). То есть, все файлы, которые будут инклюдить **fractions.hpp**, будут иметь возможность обращаться к переменной **g**, но при этом сама переменная **g** будет объявлена только в **fraction.cpp** (и никак не в файлах, которые индклюдят **fraction.hpp**). Это делает переменную **g** как глобальной, так и единой для всех файлов — то, чего мы и добиваемся.

Лекция 9

Объектно-ориентированное программирование

9.1 Класс

Класс — это пользовательский тип, который объединяет данные и код, их обрабатывающий.

Раньше мы ‘клали‘ данные структуры в объект **struct**, а рядом дописывали функции для нее. Теперь же мы хотим, чтобы с нашей структурой можно было работать **только** через функции, объявленные внутри нее. Этот принцип носит название *принципа сокрытия данных*, а такая организация кода – *инкапсуляция.*

9.2 Пример класса

Самый простой способ сделать подобие класса – это использовать **struct**. Построим структуру Студент, имеющую вид:

1. Поле **age** – возраст студента
2. Поле **name** – имя
3. Остальные методы, а именно:
   (a) **init** – создание экземпляра
   (b) **aging** – увеличение возраста на 1
   (c) **print** – печать информации о студенте.

Код 9.1: **struct** как пример класса

```
1  #include<iostream>
2
3  struct Student
4  {
5    int age;
6    std::string name;
7
8    void init(int _age, const std::string& _name)
9    {
10       this->name = _name;
11       // Alternatively, we can write
12       name = _name;
13       this->age = _age;
14   }
15
16   void aging()
17   {
18       age++;
19       std::cout << name << ": I'm now " << age << " years old" << "\n";
20   }
21
22   void print() const    // We cannot change the object itself
23   {
24       std::cout << name << "-" << age << "\n";
25   }
26 };
27
28 int main()
29 {
30     Student a, b;
31     //a.init(17, "Vasya");    // Won't work
32     // "Vasya" is char*[], but not std::string
33     a.init(17, std::string("Vasya"));
34     b = a;  // Dangerous!
35     a.print();
36     b.print();
37 }
```

В коде выше есть некоторые особенности:

1. Строка 20 — **const** после объявления функции означает, что внутри функции мы не можем менять сам объект класса.

2. Строка 31 – не будет работать, так как "Vasya"есть **char[]***, а метод класса требует **std::string**.

3. Строка 34 – опасная операция (побитовое копирование может вести себя непредсказуемо, прим. ТФ).

4. Мы можем изменять снаружи переменные *внутри* объекта, то есть, имеет силу код **a.age += 10**. Это плохо, и дальше мы рассмотрим, как такую проблему решить.

9.3 Конструкторы и деструкторы

Конструктор

Когда мы создавали объект ‘студент‘ в коде выше, нам каждый раз надо было инициализировать его через функцию **init()**. Однако мы можем это забыть сделать.

Для защиты от этого существуют конструкторы класса: особые методы, вызываемые при создании экземпляра класса. Перепишем код:

Код 9.2: Наш класс с конструктором

```
#include<iostream>

struct Student
{
  int age;
  std::string name;

  Student(int _age, const std::string& _name)
  {
      name = _name;
      age = _age;
  }

  void aging()
  {
      age++;
      std::cout << name << ": I'm now " << age << " years old" << "\n";
  }

  void print() const    // We cannot change the object itself
  {
      std::cout << name << "-" << age << "\n";
  }
};

int main()
{
    Student c(17, "Petya");
    c.aging();
    c.print();
}
```

Конструктор ничего не возвращает. Помимо этого, объект класса **не** считается созданным до тех пор, пока не будет выполнен его конструктор.

Деструктор

По аналогии с конструктором существует *деструктор*.

Деструктор — особый метод, который ничего не возвращает и вызывается при удалении объекта.

Конструктор по умолчанию

Если мы не создадим конструктор, то компилятор это сделает за нас, создав *конструктор по умолчанию*. То же самое с деструктором.

Однако, если мы создадим свой конструктор, то экземпляры класса уже надо будет клепать по его правилам, то есть, нам надо будет передавать объекту при создании те аргументы, которые принимает конструктор (в нашем случае **age** и **name**).

Код 9.3: Особенности работы конструктора

```
1 ps = new Student;    // Won't work; we don't have empty constuctor!
2 ps = new Student(18, "Misha");  // What is OK
3
4 m = new Student[10]; // Won't work; no default constuctor!
```

В примере выше на строке 4 требуется вызов конструктора по умолчанию (при создании массива), а его нет – поэтому она и помечена как неработающая. То же самое справедливо для строки 1.

9.4 Примеры работы конструктора и деструктора

Код 9.4: Пример вызова деструктора

```
1  #include<iostream>
2
3  struct Student
4  {
5    int age;
6    std::string name;
7
8    Student(int _age, const std::string& _name)
9    {
10       name = _name;
11       age = _age;
12       std::cout << "I AM CREATED: " << name << "-" << age << std::endl;
13   }
14
15   void aging()
16   {
17       age++;
18       std::cout << name << ": I'm now " << age << " years old" << "\n";
19   }
20
21   void print() const    // We cannot change the object itself
22   {
23       std::cout << name << "-" << age << "\n";
24   }
25
26   ~Student()
27   {
28       std::cout << "I AM DESTROYED. RIP: ";
29       print();
30   }
31 };
32
33 int main()
34 {
35     Student c(17, "Petya"); // Create Petya
36     c.aging();
37     {
38         Student b(18, "Vasya"); // Create Vasya
39         b.aging();
40     }   // Destroy Vasya
41 }   // Destroy Petya
```

В коде выше на строке 12 мы делаем запись о том, что объект создан. На строке 26 объявлен дестуктор – код внутри будет выполнен, когда объект перестанет существовать.

Код 12.4 компилируется: попробуйте его запустить на своей машине и увидите, как работает деструктор.

9.5 Пример. Структура Stack

Код 9.5: Структура Stack

```
1  struct Stack
2  {
3      double *s;
4      int allocated; // bytes allocated to stack
5      int top;
6
7      Stack(int max_size=100) // It is default and custom constructor
8      {
9          s = new double[max_size];
10         allocated = max_size;
11         top = 0;
12     }
13
14     void push(double x)
15     {
16         if(top < allocated)
17             s[top++] = x;
18     }
19
20     double pop()
21     {
22         if(top == 0) return -1;
23         return s[--top];
24     }
25
26     ~Stack()
27     {
28         delete[] s;
29     }
30 };
```

Обратите внимание на конструктор (строка 7) – он может использоваться и как конструктор по-умолчанию (если не передавать ему параметры, так можно, когда есть значение параметра по умолчанию), и как свой (не дефолтный) конструктор (если передать параметр **max_size**). В конце мы освобождаем ту память, которую занял объект **Stack**.

Код всего проекта можно смотреть здесь.

Лекция 10

Инкапсуляция

10.1 Понятие инкапсуляции

В Питоне инкапсуляции не было. Когда мы делаем структуру, мы полагаем, что пользователь будет взаимодействовать с ней через те методы, что мы пропишем. Однако это может нарушиться. Для того, чтобы предотвратить такую ситуацию, существует *инкапсуляция*. Инкапсуляция создает ограничения для программиста, тем самым организуя *логику* нашего кода.

Код 10.1: Инкапсуляция на примере класса Student

```cpp
#include<iostream>

class Student
{
private:
    std::string name;
    int age;
public:
    Student(std::string x, int a)
    {
        age = a;
        name = x;
    }
    // Setter
    void set_age(int a)
    {
        if(a > 0 && a > age) age = a;
    }

    void aging()
    {
        age++;
        std::cout << "BIRTHDAY!" << std::endl;
```

```
24      }
25
26      // Getter
27      int get_age() const // See description
28      {
29          return age;
30      }
31  };
32
33  int main()
34  {
35      Student a("Vasya", 17);
36      // a.age = 17   // Won't work, in private section
37      const Student& c = a;
38      std::cout << c.get_age();    // Call const method
39  }
```

В коде выше класс делится на две секции: **private** и **public**. То, что находится в секции **private**, не будет видно снаружи класса. Эти секции могут следовать друг за другом в любом порядке и любом количестве(т.е., мы можем сначала описать публичную, потом приватную, затем опять публичную), но обычно их пишут один раз, объявляя сразу все, что нужно.

Общий принцип инкапсуляции заключается в следующем: все атрибуты класса надо "прятать".

При принятии решения, в какую из секций пихнуть свой метод, следует помнить, что *конструкторы* класса **должны** быть публичными.

Доп. инфа от ТФ. Иногда может возникнуть необходимость разрешить создавать экземпляры класса, но не копировать их. В таких случаях конструктор класса следует сделать публичным, а конструктор клонирования положить в приватную секцию.

10.2 Геттеры и сеттеры

При сокрытии полей стандартом обращения к ним служат *геттеры* (getters) и *сеттеры* (setters) – методы, соответственно возвращающие и изменяющие поле. В нашем примере 12.1 они объявлены на **13** и **18** строках.

Геттеры и сеттеры имеют интересную фишку: они не обязаны работать с существующим полем класса. Вполне возможно, что они будут задавать/возвращать другие величины, но пользователь будет думать, что они содержатся в классе. Так можно создавать удобные абстракции.

10.3 Модификатор const после функции

В примере 12.1 на строчке **25** мы видим модификатор **const** *после* объявления метода.

Это говорит компилятору, что метод не будет менять никаких полей класса в ходе своей работы. Стоит учесть, что если мы будем пользоваться константной ссылкой на экземпляр класса, то в ней мы сможем вызывать только константные методы (чтоб компилятор был уверен, что мы не изменим константный объект).

10.4 Модификатор static

Статические поля

Иногда бывает необходимым сделать поле класса универсальным (т.е. единым) для всех его экземпляров. В таком случае используется модификатор **static**. Рассмотрим пример:

Код 10.2: Простой пример статического поля

```
#include<iostream>

class SelfCounter
{
private:
    static int counter;
public:
    SelfCounter()
    { counter++;}

    ~SelfCounter()
    { counter--;}

    static int get_counter()
    {
        return counter;
    }
};

void foo()
{
    SelfCounter a;
    for(int i = 0; i < 3; i++)
    {
        SelfCounter b;
    }
    std::cout << b.get_counter();
}

int main()
{
    std::cout << SelfCounter::get_counter();     // See description
```

```
33      SelfCounter x;
34      foo();
35  }
```

Здесь мы при создании нового экземпляра класса увеличиваем **count** на 1, а при удалении – уменьшаем. Важно отметить, что переменная **count** существует всегда, даже если не было создано ни одного объекта данного класса.

Статические методы

Однако у такого подхода есть проблема: чтобы получить это поле через какой-либо метод класса, нам необходимо будет создать объект. Решает это, как можно догадаться, *статический метод* (строка **12**). Такой метод привязан к классу, но никак не к объекту класса. Это несет свои минусы: статический метод не может в своем коде обращаться к нестатическим методам и полям (они ведь уже зависят от объекта, а статический метод от этого избавляется). Более того, в статическом методе не определен **this** – указатель на экземпляр класса.

Пространство имен

В коде мы использовали (строка **32**) **SelfCounter** как *пространство имен*. Каждый класс создает свое пространство имен, в котором находятся его поля и методы (разумеется, те, что в публичной секции). Таким способом удобно обращаться к статическим полям и методам.

10.5 Дружественные функции

Бывает, что определенному классу **2** требуется добраться до атрибутов другого класса (класса **1**) (как это могут делать его экземпляры), но при этом сами классы никак не связаны. Для этого придется немного ослабить *инкапсуляцию* и объявить в классе **1** (откуда хотим брать) требуемые дружественные функции или даже классы. Затем дружественные функции или классы достаточно объявить где-нибудь снаружи – и они уже будут иметь доступ внутрь **1**. Более того, можно сделать функцию дружественной даже в том случае, если она не была еще объявлена (предполагая, что это будет сделано позднее). При этом дружественные функции **не** являются методами класса **1** (того, куда мы открываем доступ).

Иллюстрацией этому может служить следующий пример:

Код 10.3: Пример дружественных функций

```
1  class CarMaster;     // For 8 line
2  class Car
3  {
4  private:
5      int engine;
6      friend class CarMaster; // Give class all access
7      // But how to give less permssions?
8      friend void CarMaster::repair(Car& x);  // Told full signature
9      // Won't work, CarMaster is not defined!
10     // We define it in 1st line, yeah!
11 };
12
13 class CarMaster
14 {
15 public:
16     void repair(Car& x)
17     {
18         x.engine++; // Can access private field. CarMaster is a friend!
19     }
20 }
```

В строке **1** мы объявляем класс **CarMaster**, но описываем его позже. Зачем?

Если не объявить заранее CarMaster, то строка **8** будет вызывать ошибку, так как мы попытаемся обратиться к необъявленному классу. А объявить класс **CarMaster** перед **Car** мы не можем, так как в этом случае он будет обращаться к полю еще не объявленного класса **Car**. Решение этого – описать сигнатуру класса **CarMaster** перед классом **Car**, но *описать* ее позднее. Такой трюк мы будем часто использовать в будущем.

Лекция 11

Наследование

11.1 Пример интерфейса – UnivMan

Представим, что у нас есть некий шаблон, описывающий общую схему кода. Пусть это будет, к примеру, *University Man* – для краткости будет звать его **UnivMan**. Им попробуем описать человека из МФТИ. Для нашего интерфейса сделаем некоторые методы: **getName()**, **getPhoto()**, **getDepartment()**. Их тело мы писать не будем; эта работа ляжет на конкретную *реализацию* нашего шаблона.

Составим затем реализацию нашего шаблона: пусть, к примеру, мы захотим описать класс учителя вуза — это класс **Teacher**. Создав этот класс, мы возьмем на себя обязанность реализовать все методы, которые мы прописали в нашем 'шаблоне' **UnivMan**. При этом мы, помимо прочего, можем и дополнить наш 'дочерний' класс своими методами.

11.2 Определение наследования

Такая иерархия классов подводит нас к третьей парадигме ООП: **наследованию**. Наследование — это явление, когда один класс (потомок) перенимает все (строго говоря, почти все; о подробностях см. ниже) поля и методы другого класса (родителя).

Замечание. В нашем примере мы затребовали, чтобы класс **UnivMan** не содержал реализаций никаких функций. Такие классы называются *интерфейсами* – сами по себе эти штуки бесполезны, но они задают некий шаблон, по которому должны строиться его потомки (в примере это были **Teacher** и **Student**).

11.3 Иерархия наследования

Вопрос о том, какой класс сделать родителем, а какой – потомком, требует хорошей продуманности.

Если бы мы хотели в нашем примере добавить класс *старосты*, то его следовало бы сделать наследником класса *студентов* и просто добавить специфичные методы (например, функцию раздачи задавальников). НЕ стоит делать обычного *студента* наследником *старосты*, равно как и не стоит делать *старосту* прямым наследником **UnivMan** и потом по-новой реализовывать те же самые методы, что *студент* уже реализовал.

Хорошим правилом для построения схемы наследования дает критерий Барбары Лисков: "Дочерний класс должен удовлетворять всем интерфейсам родительского класса и не требовать ничего дополнительного."

11.4 Модификаторы доступа

Для полей и методов можно разграничить права доступа: кто сможет их видеть, а кто нет. Для этого существует 3 модификатора доступа:

1. **public** – доступно любым другим функциям (внутри класса, для потомков и остальных)
2. **protected** – доступно любым наследникам и самому классу.
3. **private** – доступно только самому классу.

Код 11.1: Демонстрация модификаторов доступа

```
1  class Base
2  {
3  public:
4      void b_pub() {}
5  private:
6      void b_priv() {}
7  protected:
8      void b_prot() {}
9  };
10
11 class Derivative: public Base // 'public' -- see descr.
12 {
13 public:
14     void d_pub() {
15         // We now have all methods from Base
16         b_pub();    // We are calling OWN b_pub(). It exists
17         b_prot();   // Child has access to protected section
18     }
19 };
20 int main() {
21     Base b;
22     b.b_pub();  // What's okay
```

```
23      Derivative d;
24      d.b_pub();   // OK
25  }
```

Обратите внимание на строку **11** – здесь слово **public** означает, что класс *наследуется* публично. Другие куски кода будут знать, что **Derivative** наследует **Base** и мы сможем через объекты класса **Derivative** доступаться до всего, что есть в **Base**. В таком случае тоже можно делать разные модификаторы:

1. **public** – все, кто имеет доступ к **Child** и **Parent**, будут знать о наследовании.
2. **protected** – только потомок и потомок потомка будут знать о наследовании. Поясним это. Путь наследование происходит по следующей схеме: **Parent** → **Child** → **Child-of-child**. При использовании **protected** в **Child** класс **Child-of-child** и только он будет знать о том, что **Child** унаследовал **Parent**. Потомки более высшего порядка и другие классы не будут знать о наследовании.
3. **private** – никто не будет знать о наследовании, кроме самого **Child**.

Подробнее об этом можно прочитать здесь (осторожно, английский!).

11.5 Override – переопределение методов

В дочернем классе мы можем не просто пользоваться методами родителя, а переопределять их. Это называется *переопределением* метода.

При этом есть и свои тонкости. Рассмотрим их на примере:

Код 11.2: Тонкости вызова переопределенных функций

```
1  class Base
2  {
3  public:
4    void b_pub() {}
5  private:
6    void b_priv() {}
7  protected:
8    void b_prot() {}
9  };
10
11 class Derivative: public Base
12 {
13 public:
14     void d_pub() {
15         b_pub();
16         b_prot();
17     }
```

```
18 };
19
20 void foo(Base& x) {
21     x.b_pub(); // x is considered to be from Base
22     // Equal to:
23     x.Base::b_pub();
24 }
25
26 int main()
27 {
28     Derivative d;
29     foo(d);  // Will call b_pub from Base!
30 }
```

Метод **foo** (строка **20**) 'видит' **x** как объект класса **Base**. И метод **b_pub** он будет тоже вызывать из этого класса, даже если мы **foo** на вход подадим экземпляр класса-наследника.

Заметьте, что функцию мы могли бы и вызывать через обращение к классу как к пространству имен (строка **23**).

11.6 Списки инициализации

С конструкторами бывают проблемы. Рассмотрим пример:

Код 11.3: Тонкости конструкторов

```
1  #include<iostream>
2
3  class Student
4  {
5  public:
6      Student(int x, std::string s)
7      {
8          age = x;
9          name = s;
10     }
11
12     void aging()
13     {
14         age++;
15         std::cout << "I am " << age << " years old\n";
16     }
17 protected:
18     int age;
19     std::string name;
20 };
21
```

```
22 class ElderStudent: public Student
23 {
24 private:
25     int groupID;
26 public:
27     ElderStudent(int x, std::string s, int group_id) : Student(x, s)
28     {
29         groupID = group_id;
30     }
31 };
32
33 int main() {
34     return 0;
35 }
```

На строке **27** конструктор старосты. Ему нужно вызвать конструктор родителя. Почему же?

При создании объекта типа **ElderStudent** мы по факту берем класс **Student** и обертываем его тем, что мы написали в наследнике. Инициализация класса **Student** происходит через *конструктор* класса **Student**. Таким образом, за счет наследования этого класса в **ElderStudent** мы заставляем программу выполнить конструктор родительского класса (**Student**). Но вызывать конструктор напрямую нельзя. Конечно, можно понадеяться, что компилятор сделает все тихо и в неявном виде, но что будет, если наш *староста* в конструкторе будет получать не те переменные, что получал в конструкторе обычный *студент*? А если у *студента* нет конструктора по умолчанию, какой же тогда выбрать? Компилятор в этом случае не разберется и единственный способ разрулить ситуацию – использовать *списки инициализации*. Это сделано после символа ':'.

Магия списка инициализации заключается в том, что значение у нас будет инициализирован **Student** с параметрами **x** и **s** (их мы передали *старосте* в конструктор) еще до входа в само тело конструктора. Теперь у нас все хорошо: мы имеем инициализированного студента и уже готовы его отнаследовать и переопределять (что и сделано уже в *теле* конструктора).

Из вышесказанного вытекает *важный вывод*: при создании объекта будут вызваны конструкторы **всех** родительских классов, начиная от самого старшего в иерархии.

О порядке в списке инициализации

(Т.Ф. не успел этого рассказать на лекции) При использовании списка инициализации можно инициализировать несколько переменных (а не только **Student(x, s)**, как у нас в примере). Важно то, что порядок инициализации не будет зависеть от порядка перечисления через запятую в нашем конструкторе, а будет целиком определяться порядком в исходном классе. На нашем примере этого нельзя увидеть, потому

что мы инициализируем лишь одну переменную, однако это будет заметно в следующем коде:

Код 11.4: Маленький пример к порядку инициализации

```
1 class A {
2 public:
3     int age;
4     int mark;
5     A: age(5), mark(age) {}
6     // Absolutely the same as:
7     A: mark(age), age(5) {}
8 };
```

Порядок инициализации будет одинаков. Сначала в **age** уйдет 5, а затем в **mark** уйдет **age** (то есть 5). Кстати, в строке **5** видим еще один способ присвоить переменную **age** – **age(5)**. Для **age** вызывается конструктор с параметром 5, от этого в **age** уйдет значение 5 (что логично). (Для тех, кто шарит: на примитивах этот трюк тоже работает. В C++ им для таких целей специально добавили конструктор)

Лекция 12

Виртуальные методы

12.1 Вызовы методов при работе с классами-наследниками

Простейшие классы – структуры. Напишем на них пример:

Код 12.1: Пример виртуальных функций

```
#include<iostream>

 struct A
 {
      char x;
      void bar() {
          std::cout << "A";
     )
};

struct B : A    // struct inheritance is always public
{
     int y;
     void bar() {
         std::cout << "B";
     }
};

struct C: A
{
     double z;
     void bar() {
         std::cout << "C";
     }
}
```

Заметьте, что у структур наследование всегда публичное.

Распределение памяти

Экземпляр класса A будет занимать память, соответствующую размеру одного **char**'а, B – уже одного **char** и одного **int** (для шарящих: пока пренебрежем выравниванием), C – **int** и **double**. При этом занимаемые размеры расположатся в таком порядке: $a < b < c$.

12.2 Виртуальные методы

Рассмотрим пример:

Код 12.2: Вызов наследника вместо родителя

```
#include "1.cpp"

void foo(A& p)
{
    p.bar();
}

int main() {
    A a;    // Create class objects
    B b;
    C c;
    foo(b); // ???
    foo(c); // Will this stuff work??
}
```

Какая функция будет вызвана в **foo()**? Ответ прост: всегда из класса **A**. Дело в том, что функция **foo** *компилируема*: при сборке проекта **p** будет 'увиден' компилятором как экземпляр класса **A** и вызов **p.bar()** будет привязан к области памяти **A**.

Как бороться с этим? Есть несколько способов:

1. Поле типа (не рекомендуется). Добавим специальный атрибут **int m_type** в базовый класс. В конструкторах производных классов заполним **m_type** своими ID типа (например, у **A** будет 0, у **B** будет 1). При этом будет происходить *коммутация* вызовов:

   p.bar() превратится вот в такое:

Код 12.3: Условия на поле типа

```
if(p.m_type == 1) { // is A
    p.bar();
} else if(p.m_type == 2) {  // is B
    static_cast<B&>(p).bar();
```

```
5       // Cast p to B&, after that call bar() -- it will be from B
6   } else {    // is C
7       static_cast<C&>(p).C::bar();
8       // Call bar() from C namespace, the same idea
9   }
```

Это кошмар. Нам надо знать все классы, которые наследуют базовый, да и к тому же вести их учет через ID. Есть альтернатива.

2. Сохраним указатель на функцию в классе. Она уйдет всем наследникам. В конструкторе каждого наследника мы будем переопределять ссылку на этот метод, вот так:

Код 12.4: Переопределение указателя на функцию

```
1   *p.p_bar();
```

Остаются проблемы. Вдруг дочерний класс будет использовать родительский метод? Вдруг таких методов будет много? Приведем еще один способ.

3. Можно хранить не одну ссылку на метод, а указатель на таблицы, содержащей указатели на нужные методы. Те же костыли, те же проблемы. Запутаться еще легче, ведь для каждого класса эта таблица будет своя.

Но есть и хороший способ. Все это уже встроено в язык. Все обрабатывается автоматически. Для этого нужно только одно – указать перед методом слово **virtual**. Для всех потомков будет создана таблица виртуальных методов (подобно той, что указана в п.3 выше), которая будет автоматически заполняться у потомков.

Итак, *виртуальные* методы – те методы, которые предполагаются быть переопределенными в дочерних классах.

12.3 Множественное наследование

Абстрактный класс

Класс, у которого есть хотя бы один абстрактный метод с нулевой реализацией, является **абстрактным**. Его экземпляры нельзя создавать, но при этом абстрактные классы обладают мощной штукой – множественное наследование.

Код 12.5: Абстрактный метод с нулевой реализацией

```
1   class Test {
2       virtual foo() = 0;  // Zero realization
3   }
```

Множественное наследование

Множественное наследование — это явление, когда один класс наследует несколько других. При этом класс, как это говорят, будет удовлетворять нескольким *интерфейсам*. У такой штуки есть масса проблем. Что, если оба класса имеют два одинаковых метода с одинаковой сигнатурой? А атрибуты? С ними могут возникнуть большие проблемы при пересечении. Но если все методы виртуальны, а атрибутов нет, то все славно: даже если классы содержат пересекающиеся методы, мы сможем выбрать для переопределения любой из них (компилятору будет все равно, ведь они все равно не содержат реализации). Именно поэтому рекомендуется делать множественное наследование только в абстрактных классах и наследовать при этом только абстрактные классы.

Применение абстрактных классов

Итак, в чем польза абстрактных методов и классов? Они позволяют грамотнее и логичнее организовывать структуру кода, объединяя схожие классы. Наглядный пример их использования мы видели в лабе 12 'Арифметические драконы', когда реализовывали класс **Dragon**, а впоследствии и класс **Enemy** – каждый из них не обозначал какой-либо реальный объект сам по себе, однако служил шаблоном для клепания других классов, уже вполне связанных с реальными объектами (например, **Dragon** служил образом для класса **GreenDragon**).