Федеральное государственное автономное образовательное учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет: Московский институт электроники и математики

Вопросник к экзамену по курсу «Алгоритмизация и программирование»

БПМ204, 2 курс, 1 модуль

Москва, 2021 год

Содержание

1 Встраиваемые функции. Примеры.

Встраиваемые функции обозначаются как `inline` (рекомендацию компилятору сделать функцию встраиваемой).

Копилятор понимает, что необходимо будет заменять вызов функции фактическим кодом из функции.

Зачем: помогает избежать накладные расходы, связанные с вызовом функции: помещение параметра в стек или регистр, передача управления функции, возврат из функции, освобождение стека. Макросы в C приводят к куче проблем. Спасают встроенные функции.

(При вызове обычной функции осуществляется переход к определенному адресу (адресу функции) с последующим возвратом после завершения. Переходы и запоминание соответствующих адресов влекут затраты времени, связанные с использованием функций)

Встроенные функции обладают всеми **достоинствами** функций: они приводят к типу, они знают об области видимости, они могут содержать условные операторы, они могут быть перегруженными или функциями- членами классов, или членами пространств имён, или функциями-друзьями, или задаваться шаблонами, или быть параметрами шаблонов. Макросам всё это недоступно.

Следует отметить, что `inline` — **лишь рекомендация** (а не команда) компилятору заменять вызов функции её телом. Компилятор может посчитать встраивание нецелесообразным, так как оно ведёт к разбуханию кода, а это, в свою очередь, к более частым промахам кэша команд, что в конечном итоге может даже снизить производительность. В некоторых случаях компилятор попросту не может встроить функцию (например, если в текущей единице трансляции отсутствует её тело).

Также **желательно понимать**, что встроенные функции лучше всего использовать только для коротких функций (не более нескольких строк), которые обычно вызываются внутри циклов и не имеют ветвлений.

Наконец, современные компиляторы **автоматически конвертируют** соответствующие функции во встроенные — этот процесс автоматизирован настолько, что даже лучше ручной выборочной конвертации, проведенной программистом. Даже если вы не пометите функцию как встроенную, компилятор автоматически выполнит её как таковую, если посчитает, что это способствует улучшению производительности. Таким образом, в большинстве случаев нет особой необходимости использовать ключевое слово `inline`. Компилятор всё сделает сам.

Пример:

```
inline double Sqr(double x) { return x*x; }
```

`Sqr(x)` будет (почти во всех реализациях) вычисляться так же быстро, как $x * x$, но x предварительно приводится к типу `double`.

Еще один пример, который если считаете лишним, то можете удалить.

```
#include <iostream>

inline int max(int a, int b)
{
    return a < b ? b : a;
}

int main()
{
    std::cout << max(7, 8) << '\n';
    std::cout << max(5, 4) << '\n';
    return 0;
}
```

Теперь, при компиляции функции `main()`, ЦП будет читать код следующим образом:

```
int main()
{
    std::cout << (7 < 8 ? 8 : 7) << '\n';
    std::cout << (5 < 4 ? 4 : 5) << '\n';
    return 0;
}
```

2 Аргументы функций по умолчанию. Примеры.

Параметр по умолчанию (или «необязательный параметр») — это параметр функции, который имеет определенное (по умолчанию) значение. Если пользователь не передает в функцию значение для параметра, то используется значение по умолчанию. Если же пользователь передает значение, то это значение используется вместо значения по умолчанию.

Один или больше последних аргументов функции могут задаваться по умолчанию.

```
void f(int x, int y = 5, int z = 10) /*аналогично*/ f(1), f(1,5), f(1,5,10)
void g(int x=5, int y) //неправильно, по умолчанию могут задавать только
↪  последние аргументы
f(1); //будет вызвано f(1, 5, 10)
f(1, 2); //будет вызвано f(1, 2, 10)
f(1, 2, 3); //будет вызвано f(1, 2, 3)
```

Для каждого параметра значение по умолчанию можно указать не более одного раза, но каждое последующее объявление функции, а также определение функции может назначать параметрам значения по умолчанию. При этом, после каждого объявления/определения значения по умолчанию должны иметь лишь последние параметры.

```
void foo(int a, int b=3); //правильно, параметр b имеет значение по умолчанию
void foo2(int a=3, int b); //ошибка: значения по умолчанию могут иметь лишь
↪  последние параметры
void foo3(int a, int b, int c=4); //правильно, параметр c имеет значение по
↪  умолчанию
void foo4(int x, int y, float z, double t=3, double u=4); //правильно,
↪  параметры t и u имеют значения по умолчанию
```

Умолчания параметров строго равносильны конструкциям с `inline` функциями:

```
void foo(int x, int y, double z);
inline void foo(int x, int y) { foo(x, y, 10); }
inline void foo(int x) { foo(x, 5); }
```

Выражение для умолчания одного параметра не может использовать имена других параметров. Это связано с тем, что порядок вычисления фактических параметров вызова не определен языком и зависит от реализации.

Если же такое очень хочется, то можно написать явно:

```
void f(int x, int y);
inline void f(int x){ f(x, x); }
```

Рекомендации:

1. Все параметры по умолчанию в прототипе или в определении функции должны находиться справа.

2. Если имеется более одного параметра по умолчанию, то самым левым параметром по умолчанию должен быть тот, который с наибольшей вероятностью (среди всех остальных параметров) будет явно переопределен пользователем.

3. Объявляйте параметры по умолчанию в предварительном объявлении функции, в противном случае (если функция не имеет предварительного объявления) — объявляйте в определении функции.

Стоит ли писать про перегрузку функций?

3 Неизвестные аргументы функций. Примеры.

Если количество или типы некоторых аргументов неизвестны, надо заменить их многоточием. При этом хотя бы один аргумент должен быть известен.

!Известные аргументы должны быть в начале списка:

```
void f(); // функция без аргументов
void g(void); // тоже функция без аргументов
int printf(const char* fmt, ...); // функция с неизвестными аргументами
int sprintf(char* s, const char* fmt, ...); // ещё одна
int bad1(...); // Неправильно! Нужен хотя бы один известный аргумент.
int bad2(..., char* fmt); // Неправильно! Известные аргументы должны быть в
↪  начале списка
```

Аргументы, соответствующие многоточию и имеющие тип float, перед передачей автоматически приводятся к типу double.

Доступ к дополнительным аргументам такой функции в её теле требует средств из библиотеки stdarg.h.

Вот тут нормально расписано про stdarg.h: https://metanit.com/cpp/c/5.13.php

```
#include <iostream>
#include <stdarg.h>
using namespace std;

int sum(int n, ...) {
    int sum = 0;
    va_list p;          //указатель va_list
    va_start(p, n);     // устанавливаем указатель
    for (int i = 0; i < n; i++)
        sum += va_arg(p, int);  // получаем значение текущего параметра типа
        ↪  int
    va_end(p); // завершаем обработку параметров
    return sum;
}

int main(void) {
    cout << sum(4, 1, 2, 3, 4) << endl;
    cout << sum(5, 12, 21, 13, 4, 5) << endl;
    return 0;
}
```

4 Ссылки. Примеры использования ссылок.

Ссылка — это имя, которое является псевдонимом или альтернативным именем для ранее объявленной переменной. Однако основное их предназначение – использование в качестве формальных аргументов функций. Принимая ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями. Ссылки представляют собой альтернативу указателям при обработке крупных структур посредством функций.

Виды ссылок:

1. Ссылки на неконстантные значения (обычно их называют просто «ссылки» или «неконстантные ссылки»).
2. Ссылки на константные значения (обычно их называют «константные ссылки»).
3. В `C++11` добавлены ссылки `r-value`.

Пример использования ссылок

```
int value = 7; // обычная переменная
int &ref = value; // ссылка на переменную value
```

Ссылки в качестве псевдонимов

```
#include <iostream>

int main()
{
        int value = 7; // обычная переменная
        int &ref = value; // ссылка на переменную value

        value = 8; // value теперь 8
        ref = 9; // value теперь 9

        std::cout << value << std::endl; // выведется 9
        ++ref;
        std::cout << value << std::endl; // выведется 10

        return 0;
}
```

В примере, приведенном выше, объекты `ref` и `value` обрабатываются как одно целое. Использование оператора адреса с ссылкой приведет к возврату адреса значения, на которое ссылается ссылка:

```
std::cout << &value; // выведется 0035FE58
std::cout << &ref; // выведется 0035FE58
```

!В ссылках **разыменовывание** предполагается неявно.

!Одно из отличий ссылок от указателей в том, что ссылку **нельзя объявить без инициализации**.

```
int value = 7;
int &ref = value; // корректная ссылка: инициализирована переменной value

int &invalidRef; // некорректная ссылка: ссылка должна ссылаться на что-либо
```

!Ссылки на неконстантные значения могут быть инициализированы только неконстантными `l-values`. Они не могут быть инициализированы константными `l-values` или `r-values`:

```
int a = 7;
int &ref1 = a; // ок: a - это неконстантное l-value
```

```
const int b = 8;
int &ref2 = b; // не ок: b - это константное l-value
```

```
int &ref3 = 4; // не ок: 4 - это r-value
```

!Ссылка пожизненно указывает на один и тот же адрес. После инициализации изменить объект, на который указывает ссылка — нельзя. Рассмотрим следующий фрагмент кода:

```
int value1 = 7;
int value2 = 8;

int &ref = value1; // ок: ref - теперь псевдоним для value1
ref = value2; // присваиваем 8 (значение переменной value2) переменной
↪  value1.
```

Здесь НЕ изменяется объект, на который ссылается ссылка!

Чаще всего ссылки используют в качестве параметров функций. Такой метод называется **передачей аргументов по ссылке**.

Когда целесообразно:

1. Когда нужно позволить изменять объект данных в вызывающей функции

2. Когда нужно ускорить работу программы за счет передачи ссылки вместо полной копии объекта данных.

Важно избегать ситуаций, когда возвращаемая ссылка указывает на область памяти, которая прекращает существование после завершения работы функции. (Чтобы избежать, лучше возвращать ссылку, которая была передана в качестве аргумента. Второй способ – использование операции new для нового хранилища).

Пример:

```
void foo (int &x) { x = 17; }
int main ()
{
    int z = 5;
    foo (z); // z=17
}
```

Другой пример:

```
#include <iostream>

// ref - это ссылка на переданный аргумент, а не копия аргумента
void changeN(int &ref)
{
        ref = 8;
}

int main()
{
        int x = 7;

        std::cout << x << '\n';

        changeN(x); // обратите внимание, этот аргумент не обязательно должен
        ↪  быть ссылкой

        std::cout << x << '\n';
        return 0;
}
```

Совет: Передавайте аргументы в функцию через неконстантные ссылки-параметры, если они должны быть изменены функцией в дальнейшем.

Сслыки на константные значения.

```
const int value = 7;
const int &ref = value; // ref - это ссылка на константную переменную value
```

Инициализация ссылок на константы

В отличие от ссылок на неконстантные значения, которые могут быть инициализированы только неконстантными `l-values`, ссылки на константные значения могут быть инициализированы неконстантными `l-values`, константными `l-values` и `r-values`:

```
int a = 7;
const int &ref1 = a; // ок: a - это неконстантное l-value
```

```
const int b = 9;
const int &ref2 = b; // ок: b - это константное l-value
```

```
const int &ref3 = 5; // ок: 5 - это r-value
```

Как и в случае с указателями, константные ссылки также могут ссылаться и на неконстантные переменные. При доступе к значению через константную ссылку, это значение автоматически считается const, даже если исходная переменная таковой не является:

```
value = 7;
const int &ref = value; // создаем константную ссылку на переменную value

value = 8; // ок: value - это не константа
ref = 9; // нельзя: ref - это константа
```

Обычно `r-values` имеют область видимости выражения, что означает, что они уничтожаются в конце выражения, в котором созданы

Однако, когда константная ссылка инициализируется значением `r-value`, время жизни `r-value` продлевается в соответствии со временем жизни ссылки:

```
int somefcn()
{
    const int &ref = 3 + 4; // обычно результат 3 + 4 имеет область видимости
    ↪ выражения и уничтожился бы в конце этого стейтмента, но, поскольку
    ↪ результат выражения сейчас привязан к ссылке на константное значение,
    std::cout << ref; // мы можем использовать его здесь и время жизни
    ↪ r-value продлевается до этой точки, когда константная ссылка
    ↪ уничтожается
```

```
}
```

Константные ссылки в качестве параметров функции

Ссылки, используемые в качестве параметров функции, также могут быть константными. Это позволяет получить доступ к аргументу без его копирования, гарантируя, что функция не изменит значение, на которое ссылается ссылка:

```
// ref - это константная ссылка на переданный аргумент, а не копия аргумента
void changeN(const int &ref)
{
    ref = 8; // нельзя: ref - это константа
}
```

Ссылки на константные значения особенно полезны в качестве параметров функции из-за их универсальности. Константная ссылка в качестве параметра позволяет передавать неконстантный аргумент `l-value`, константный аргумент `l-value`, литерал или результат выражения:

```
#include <iostream>

void printIt(const int &a)
{
    std::cout << a;
}

int main()
{
    int x = 3;
    printIt(x); // неконстантное l-value

    const int y = 4;
    printIt(y); // константное l-value

    printIt(5); // литерал в качестве r-value

    printIt(3+y); // выражение в качестве r-value

    return 0;
}
```

Во избежание ненужного, слишком затратного копирования аргументов, переменные,

которые не являются фундаментальных типов данных (типов `int`, `double` и т.д.) или указателями, — должны передаваться по (константной) ссылке в функцию. Фундаментальные типы данных должны передаваться по значению в случае, если функция не будет изменять их значений.

Правило: Переменные нефундаментальных типов данных и не являющиеся указателями, передавайте в функцию по (константной) ссылке.

5 Динамическая память. Примеры.

В C++ для динамической памяти используются операторы `new` — для выделения памяти и `delete` — для освобождения памяти.

Важное отличие оператора new от функции `malloc()` заключается в том, что он возвращает значение типа «указатель-на-объект» (то есть `MyClass *`),

Существует и две формы применения delete:

- `delete` указатель; — для одного элемента
- `delete[ ]` указатель; — для массивов

Пример:

```
int main()
{
    MyClass *obj= new MyClass(5); //инициализация объекта MyClass пятеркой
    MyClass *arr= new MyClass[15]; //массив из 15 элеметов
    MyClass *arr= new MyClass("something")[15]; //ошибка (нельзя
     ↪  комбинировать два типа скобок)
}
```

6 Константы. Константы и указатели. Константы и ссылки. Примеры.

Константа — это переменная, которую необходимо обязательно инициализировать и которая после этого не меняет своего значения.

Константы обязательно инициализировать, например:

```
const int foo = 10; //можно
const int bar; //нельзя
```

Константы с указателями:

Указатель на константное значение — это неконстантный указатель, который указывает на неизменное значение. Для объявления указателя на константное значение, используется ключевое слово const перед типом данных:

```
const int *foo;
```

Значение указателя изменить можно (так, чтобы он указывал на что- нибудь другое), а вот значение переменной, на которую он указывает, менять нельзя.

Константный указатель — это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется ключевое слово const между звёздочкой и именем указателя:

```
int *const foo = &x;
```

Значение указателя менять нельзя. Значение того, на что он указывает, менять можно. Константный указатель обязательно инициализировать, как и любую другую константу.

Можно объявить **константный указатель на константное значение**, используя ключевое слово const как перед типом данных, так и перед именем указателя:

```
const int *const foo = &x;
```

Ничего нельзя изменить: ни значение указателя, ни значение того, на что он указывает. Опять же, инициализация обязательна.

Константы с ссылками:

У ссылок разнообразия значительно меньше, ибо «указательная» часть ссылки и так всегда константа, то есть ссылка всегда указывает на одну и ту же область памяти. Значит, бывает только:

```
const int &foo = x;
```

В данном случае значение переменной по ссылке изменить нельзя.

7 Пространства имен. Области видимости. Примеры.

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен(когда два идентификатора находятся в одной области видимости, и компилятор не может понять, какой из них двух следует использовать в конкретной ситуации).

Пример:

`boo.h`: и `doo.h`: — заголовочные файлы с функциями, которые могут выполнять разные вещи, но при этом имеют одинаковые имена, параметры и тип возвращаемого значения.

`boo.h`:

```
// Функция doOperation() выполняет операцию сложения своих параметров
int doOperation(int a, int b)
{
    return a + b;
}
```

`doo.h`:

```
// Функция doOperation() выполняет операцию вычитания своих параметров
int doOperation(int a, int b)
{
    return a - b;
}
```

Если подключить оба заголовочных файла и попробовать вызвать функцию `doOperation()`, возникнет конфликт имен. Для его разрешения необходимо указать пространство имен, которое может иметь определенную область видимости.

Области видимости:

- область видимости файла;
- локальная область видимости (внутри блока-внутри `{}`);
- область видимости класса.

По умолчанию, **глобальные переменные** и обычные функции определены в глобальном пространстве имён. Например:

```
int gz = 4;
int boo(int z)
{
    return -z;
}
```

C++ позволяет объявлять собственные пространства имён через ключевое слово namespace. Всё, что объявлено внутри пользовательского пространства имён, принадлежит только этому пространству имён (а не глобальному).

Перепишем заголовочные файлы с примера выше, но уже с использованием namespace:

boo.h:

```
namespace Boo
{
    // Эта версия doOperation() принадлежит пространству имен Boo
    int doOperation(int a, int b)
    {
        return a + b;
    }
}
```

doo.h:

```
namespace Doo
{
    // Эта версия doOperation() принадлежит пространству имен Doo
    int doOperation(int a, int b)
    {
        return a - b;
    }
}
```

Доступ к пространству имён через оператор разрешения области видимости (::)

Один из вариантов — использовать название необходимого пространства имён + оператор разрешения области видимости (::) + требуемый идентификатор.

```
int main()
{
    Boo::doOperation(5, 4);
    Doo::doOperation(5, 4);
```

```
    return 0;
}
```

Оператор разрешения области видимости хорош, так как позволяет выбрать конкретное пространство имён. Также этот оператор можно использовать без какого-либо префикса (например, `::doOperation`). В таком случае мы ссылаемся на глобальное пространство имён.

Псевдонимы и вложенные пространства имён

Одни пространства имён могут быть вложены в другие пространства имён.

```
#include <iostream>
namespace Boo
{
    namespace Doo
    {
        const int gx = 7;
    }
}
namespace Foo = Boo::Doo; // Foo теперь считается как Boo::Doo

int main()
{
    std::cout << Foo::gx; // это, на самом деле, Boo::Doo::gx
    return 0;
}
```

Вложенность пространств имён не рекомендуется использовать, так как это подвержено ошибкам при неумелом использовании и дополнительно усложняет логику программы.

Использование «объявления `using`»

Одной из альтернатив является использование «директивы using». Лучше — намеренно ограничить область применения стейтментов using с самого начала, используя правила локальной области видимости:

```
int main()
{
    {
        using namespace Boo;
        doOperation(5, 4);
    } // Действие using namespace Boo заканчивается здесь
    {
```

```
        using namespace Foo;
        doOperation(5, 4);
    } // Действие using namespace Foo заканчивается здесь
    return 0;
}
```

8 Потоковый ввод-вывод. Примеры.

Для работы подключается библиотека ввода-вывода `iostream`.

Библиотека `iostream` определяет три стандартных потока:

- `cin` стандартный входной поток
- `cout` стандартный выходной поток
- `cerr` стандартный поток вывода сообщений об ошибках

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

Вывод информации:

```
cout << "значение";
```

Ввод информации:

```
cin >> "идентификатор";
```

Возможно многократное назначение потоков:

```
cout << "значение1" << "значение2" << ... << "значение n";
```

Ввод/вывод в файлах осуществляется с помощью библиотеки `fstream`. Нужно создать переменную `ofstream` (для вывода), `ifstream` (для ввода), `fstream` (двунаправленный поток) и потом связать ее с определенным файлом функцией `open()`. После работы закрыть поток `close()`.

Для ввода/вывода можно так использовать операции `<<` и `>>`:

```
file << "значение1" << "значение2" << ... << "значение n";
```

Так же для считывания данных из файлов можно использовать

```
getline("поток","переменная, куда считываем")
```

9 Парадигма ООП. Преимущества и недостатки.

Парадигма программирования — совокупность идей и понятий, определяющая стиль написания программ.

Процедурное программирование

Процедура — синтаксически обособленная часть программы для решения подзадачи.

Объектно-ориентированное программирование

Объект — синтаксически обособленная часть программы для имитации сущности реального мира.

Парадигма ООП:

Объектно-ориентированное программирование (ООП) – это методология программирования, основанная на следующих концепциях:

- инкапсуляция
- наследование
- полиморфизм

Объект – упрощенное, идеализированное описание реальной сущности предметной области.

Характеристики объекта:

- состояние — заданные значения атрибутов(свойств)
- поведение — набор операций над атрибутами(методы)

Свойства ООП:

- **Инкапсуляция** — скрытие деталей реализации; объединение данных и действий над ними.
- **Наследование** позволяет создавать иерархию объектов, в которой объекты-потомки наследуют все свойства своих предков. Свойства при наследовании повторно не описываются. Кроме унаследованных, потомок обладает собственными свойствами. Объект в `C++` может иметь сколько угодно потомков и предков.
- **Полиморфизм** — возможность определения единого по имени действия, применимого ко всем объектам иерархии, причем каждый объект реализует это действие собственным способом.

Преимущества и недостатки ООП:

Преимущества (при создании больших программ):

- использование при программировании понятий, более близких к предметной области;

- локализация свойств и поведения объекта в одном месте, позволяющая лучше структурировать и, следовательно, отлаживать программу;
- возможность создания библиотеки объектов и создания программы из готовых частей;
- исключение избыточного кода за счет того, что можно многократно не описывать повторяющиеся действия;
- сравнительно простая возможность внесения изменений в программу без изменения уже написанных частей, а в ряде случаев и без их перекомпиляции.

Недостатки:

- некоторое снижение быстродействия программы, связанное с использованием виртуальных методов;
- идеи ООП не просты для понимания и в особенности для практического использования;
- для эффективного использования существующих ОО систем требуется большой объем первоначальных знаний.

Технология разработки ОО программ

1. В предметной области выделяются понятия, которые можно использовать как классы. Кроме классов из прикладной области, обязательно появляются классы, связанные с аппаратной частью и реализацией.
2. Определяются операции над классами, которые впоследствии станут методами класса. Их можно разбить на группы:
    - связанные с конструированем и копированем объектов;
    - для поддержки связей между классами, которые существуют в прикладной области;
    - позволяющие представить работу с объектами в удобном виде.
3. Определяются функции, которые будут виртуальными.
4. Определяются зависимости между классами. Процесс создания иерархии классов — итерационный. Например, можно в двух классах выделить общую часть в базовый класс и сделать их производными.

10 Классы: поля и методы. Константные объекты и методы. Примеры.

Классы в C++ — это абстракция, описывающая методы и свойства ещё не существующих объектов.

Описание класса:

```
class <имя>
{
[ private: ]
    <описание скрытых элементов>
[ protected: ]
    <описание защищенных элементов>
public:
    <описание доступных элементов>
}
```

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);
- могут быть описаны с модификатором `const`;
- могут быть описаны с модификатором `static`;
- инициализация полей при описании не допускается.

Методы класса — функции, определенные внутри класса, называются методами. Методы могут быть реализованы как внутри, так и вне класса.

Константный объект:

Объекты классов можно сделать константными (используя ключевое слово `const`). Инициализация выполняется через конструкторы классов:

```
const monstr Dead (0,0);
```

Любая попытка изменить переменные-члены объекта запрещена

Константный метод — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект). Чтобы сдалать метод константным, достаточно добавить ключевое слово `const` к прототипу функции после списка параметров, но перед телом функции:

```
int gethealth() const { return health; }
```

`int gethealth() const;` - объявление константного метода, который определен вне тела класса

Метод класса:

- объявляется с ключевым словом `const` после списка параметров;
- не может изменять значения полей класса;
- может вызывать только константные методы;
- может вызываться для любых (не только константных) объектов.

Пример

```
class Anything
{
public:
    int m_value;

    Anything(): m_value(0) { }
    // неконстантный метод - изменяет поле класса
    void setValue(int value) { m_value = value; }
    // константный метод
    int getValue()  const { return m_value ; }
};

int main()
{
    // объявляем константный объект класса
    const Anything any_const; // вызываем конструктор по умолчанию с const в
    ↪  начале

    any_const.m_value = 7; // ошибка компиляции: нарушение const
    any_const.setValue(7); // ошибка компиляции: нарушение const
    any_const.getValue(); // ok, к константному объекту применяем константный
    ↪  метод

    // объявляем обычный объект класса
    Anything any; // все методы применимы, ошибок не будет

    return 0;
}
```

11 Классы и объекты. Модификаторы доступа. Методы. Примеры.

Классы в C++ — это абстракция, описывающая методы и свойства ещё не существующих объектов. (Класс - это пользовательский тип данных, тогда объект класса - это переменная данного пользовательского типа)

Объекты — конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса.

Спецификация класса определяет, что является классом и как он себя ведет. Спецификация класса для объекта — это определение его свойств и методов.

Имеются также другие элементы спецификации класса, включая параметры класса и модификаторы класса.

Спецификация класса состоит из двух частей:

1. Объявление класса — это описание компонентов данных в терминах членов класса, а также открытый интерфейс в терминах функций-членов, называемых методами.

2. Определение методов класса, которые описывают функции-члены.

Интерфейсом класса (на данный момент) будем считать открытые члены и методы класса, которые доступны пользователю (программе, использующей классы).

Спецификатор (модификатор) доступа определяет, кто имеет доступ к членам этого спецификатора. Каждый из членов «приобретает» уровень доступа в соответствие со спецификатором доступа (или, если он не указан, в соответствие со спецификатором доступа по умолчанию (`private`)).

Уровни доступа:

- `public` — доступ любой программе, которая использует объект класса
- `private` — доступ только через открытые (public) функции-члены класса или через дружественные функции. Такая изоляция называется сокрытием данных.
- `protected` — доступ только членам дочерних классов.

Так как элементы данных объявлены в `private`, то и доступ к ним могут получить только методы класса, внешний доступ к элементам данных запрещён.

Поэтому принято объявлять в классах специальные методы — так называемые `set` и `get` функции, с помощью которых можно манипулировать элементами данных.

`Set`-функции инициализируют элементы данных, а `get`-функции позволяют просмотреть значения элементов данных.

Пример

```
class some {
// дружественная функция сеттер
  friend void set(int);
public:
  int a_
protected:
  int b_;
private:
  int c_;
};
void set(int number) {
  this->a_ = 0; // OK
  this->b_ = 0; // OK
  this->c_ = 0; // OK
}
// внешняя функция имеет доступ только к public полям класса
void func(some& obj) {
  obj.a_ = 0; // OK
  obj.b_ = 0; // compile time error
  obj.c_ = 0; // compile time error
}
```

12 Конструкторы. Конструктор по умолчанию. Примеры.

Конструктор — функция, предназначенная для инициализации объектов класса. Это особый тип метода класса, который автоматически вызывается при создании объекта этого же класса. (Если переменные-члены класса открыты, то можно обойтись без конструктора и инициализировать их напрямую).

- При создании объектов класса, все переменные-члены класса должны быть проинициализированы (иначе, если вызвать `get`-функцию сразу после создания объекта, мы получим мусор).
- Инициализацией элементов данных класса занимается конструктор.
- Имя конструктора обязательно должно совпадать с именем класса.
- Конструктор не возвращает никаких значений! Вообще никаких, даже `void`.
- В любом классе должен быть конструктор. Если он не объявлен, компилятор создаст его автоматически (будет создан конструктор без параметров).
- Но! Если хоть один конструктор объявлен (допустим, с параметрами), то компилятор уже не будет создавать никаких конструкторов.
- Конструкторы не наследуются.
- Их нельзя описать как `const`, `virtual` или `static`.
- Конструкторы глобальных объектов вызываются до вызова функции `main`.
- Локальные объекты создаются, как только становится активной область их действия.
- Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

Пусть в классе объявлен **конструктор с параметрами по умолчанию** вида

`Triangle ( float a=10, float b = 20, float c = 30) {}`

Тогда **вызов конструктора** выполняется, если в программе встретилась одна из конструкций:

`имя-класса имя-объекта [(список параметров)]`

```
//прямая инициализация
Triangle obj(4, 5, 6)
Triangle obj(4) // с использованием параметров по умолчанию
Triangle obj /*аналогично*/ Triangle obj()
```

```
//uniform-инициализация
Triangle obj{4, 5, 6}
Triangle obj{4}
```

имя-класса (список параметров)

```
Triangle obj;
obj = Triangle(4, 5, 6)
```

имя-класса имя-объекта = выражение

```
Triangle obj;
obj = (4, 5, 6)
```

использование оператора new

```
Triangle *obj_ptr = new Triangle(4, 5, 6)
Triangle *obj_ptr = new Triangle(4)
Triangle *obj_ptr = new Triangle
```

13 Конструкторы с параметрами. Ограничения. Список инициализаторов. Примеры.

Конструктор с параметрами:

```
Triangle (float a, float b, float c)
{
    ...
}
```

```
class Vector
{
private:
    float x_, y_, z_;
public:
    //конструкторы
    Vector(float x, float y, float z)//конструктор с параметрами
    {
        x_ = x;
        y_ = y;
        z_ = z;
    }

    Vector() //конструктор без параметров
    {
        x_ = 0;
        y_ = 0;
        z_ = 0;
    }

    //деструктор
    ~Vector() {}
};
```

?Ограничения:

- Конструктор не возвращает значение, даже типа `void`. Нельзя получить указатель на конструктор.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).

- Конструктор, вызываемый без параметров, называется конструктором по умолчанию.
- Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию
- Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию
- Если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов.
- Конструкторы не наследуются.
- Конструкторы нельзя описывать как `const`, `virtual` и `static`.
- Конструкторы глобальных объектов вызываются до вызова функции main.
- Локальные объекты создаются, как только становится активной область их действия.
- Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

Список инициализаторов:

Помимо рассмотренной ранее инициализации присваиванием в конструкторе, в C++ есть метод инициализации переменных-членов класса через список инициализации членов, вместо присваивания им значений после объявления. Данный метод позволяет работать с константными свойствами класса, так как их нельзя создать без инициализации.

Пример с помощью оператора присваивания:

```
class Values
{
private:
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Values()
    {
        // Это всё операции присваивания, а не инициализация
        m_value1 = 3;
```

```
        m_value2 = 4.5;
        m_value3 = 'd';
    }
    // при создании объекта класса, сначала создаются переменные-члены
     ↪  класса, потом вызывается конструктор - в нашем случае
     ↪  конструктор-присваивания
};
```

Пример с помощью списка инициализации:

```
class Values
{
private:
// это не объявление переменных, а лишь информация о формате объектов данного
 ↪  класса
    int m_value1;
    double m_value2;
    char m_value3;

public:
    Values() : m_value1(3), m_value2(4.5), m_value3('d') // напрямую
     ↪  объявляем и сразу инициализируем переменные-члены класса
    {
    // Нет необходимости использовать присваивание
    }
};
```

14 Конструктор копирования. Примеры.

Конструктор копирования — это особый тип конструктора, который используется для создания нового объекта через копирование существующего объекта.

И, как в случае с конструктором по умолчанию, если не предоставлен конструктор копирования, то язык C++ создаст public-конструктор копирования автоматически. По умолчанию созданный конструктор копирования будет использовать почленную инициализацию (т.е. каждый член объекта-копии инициализируется непосредственно из члена объекта-оригинала).

Конструктор копирования вызывается:

- при описании нового объекта с инициализацией другим объектом;
- при передаче объекта в функцию по значению;
- при возврате объекта из функции;
- при обработке исключений.

Пример явного определения конструктора копирования:

```
#include <iostream>

// класс дробь
class Drob
{
private:
    int m_numerator; // числитель
    int m_denominator; // знаменатель
public:
    // Конструктор копирования
    Drob(const Drob &drob) : m_numerator(drob.m_numerator),
    ↪ m_denominator(drob.m_denominator) {};

    // Конструктор по умолчанию
    Drob(int numerator = 0, int denominator = 1) : m_numerator(numerator),
    ↪ m_denominator(denominator) {};

    friend std::ostream& operator<<(std::ostream& out, const Drob &d1);
};

std::ostream& operator << (std::ostream& out, const Drob &d1)
{
```

```
21      out << d1.m_numerator << "/" << d1.m_denominator;
22      return out;
23  }
24
25  int main()
26  {
27      Drob sixSeven(6, 7); // прямая инициализация объекта класса Drob,
        ↪  вызывается конструктор Drob(int, int)
28      Drob dCopy(sixSeven); // конструктор копирования
29      std::cout << dCopy << '\n';
30  }
```

15 Деструкторы. Примеры.

Деструктор — это специальный тип метода класса, который выполняется при удалении объекта класса. В то время как конструкторы предназначены для инициализации класса, деструкторы предназначены для очистки памяти после него.

Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) деструктор не нужен, так как C++ автоматически выполнит очистку самостоятельно. Деструктор вызывается автоматически, когда:

- объект выходит из области видимости (для локальных объектов — при выходе из блока {}, в котором они объявлены)
- для глобальных — как часть процедуры выхода из main)
- для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции delete.

Если для переменных-членов класса была выделена динамически память при помощи оператора new, то автоматически вызванный деструктор эту мпамять не освободит. Необходимо явно прописать деструктор для класса.

Пример деструктора:

```
class Massiv
{
private:
        int *m_array;
        int m_length;
public:
        Massiv(int length) // конструктор
        {
            // выделяет динамически память для массива заданной длины
                m_array = new int[length];
                m_length = length;
        }
        ~Massiv() // деструктор
        {
                // Динамически удаляем массив, который выделили ранее
                delete[] m_array ;
        }
};

int main()
```

```
{
    // вызываем конструктор, передаем целое значение-длину массива
        Massiv arr(15);

        /* какие-то действия с массивом */

        return 0;
}
// объект arr класса Massiv был создан глобально, таким образом данный объект
↪  удаляется при завершении функции main
// таким образом в данном месте вызывается деструктор ~Massiv()
// если не прописывать в явном виде деструктор в классе, то вызовется
↪  деструктор по умолчанию, котрый не освободит выделенную при помощи new
↪  память
```

Особенности:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как `const` или `static`;
- не наследуется;
- может быть виртуальным
- Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

16 Классы: статические методы и поля. Примеры.

Если вы объявляете переменную статической, то может существовать только одна копия этой переменной — независимо от того,сколько объектов данного класса создается. Каждый объект просто использует (совместно с другими) эту одну переменную.

Статические данные относятся ко всем объектам класса.

Такие данные используются, если:

- требуется контроль общего количества объектов класса;
- требуется одновременный доступ ко всем объектам или части их;
- требуется разделение объектами общих ресурсов.

Статические члены описываются с помощью ключевого слова `static`.
Обращение к статическому элементу осуществляется как `ИмяКласса :: ИмяЭлемента` (`cl::x` Если `x` — статическое член-данное класса `cl`, то к нему можно обращаться)
Статические поля:

- Память под статическое поле выделяется один раз

```
class A
{
public:
    static int count; /* Объявление */
};
int A::count; // Определение (или int A::count = 10;)
```

- поля доступны через имя класса и через имя объекта:

```
cout << A::count << a->count << b.count; (A *a, b;)
```

- На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как `private`, можно изменить только с помощью статических методов.
- Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции `sizeof`.

Статические методы:

```
class A
{
    static int count;
public:
    static void inc_count()
    {
        count++;
    }
};

A::int count;
void f()
{
    A a;
    // a.count++ - нельзя
    a.inc_count(); // или A::inc_count();
}
```

17 Дружественные функции и дружественные классы. Примеры.

В классах, напомню, есть 3 уровня доступа к полям и методам класса. Так как поля и методы из `private` и недоступны вне класса для обращения. Но часто бывают ситуации когда есть какие-то внешние фукнции/другие классы, которые должны иметь доступ к каким-то закрытым полям. Что делать? Можно, конечно, сделать сеттеры и геттеры. Но такая реализация плоха тем, что мы всем дадим возможность получать/изменять закрытые поля или методы класса. Чтобы этого не делать, а дать выборочный доступ к непубличным частям класса, есть механизм дружественных функций.

Мы можем внутри класса объявить любую как внешнюю обычную функцию, так и метод другого класса *дружественной* к данному классу.

Это значит следующее: мы говорим что данная функция имеет доступ ко всем полям и методам класса, а не только к тем, которые `public`. Аналогично с целым классом — можно объявить целый класс внутри другого класса дружественным к другому. Это значит что этот класс является другом и имеет доступ ко всем полям и методам класса.

Одна функция может быть дружественной сразу нескольким классами.

```
class monstr;
class mistress;

class hero{
public:
    void kill(monstr &);
};

class monstr{
private:
    int health, ammo;
public:
    monstr(int he = 100, int am = 10){
        health = he;
        ammo = am;
    }
    friend int steal_ammo(monstr &); //внешняя функция
    friend void hero::kill(monstr &); //дружественная функция из класса hero
    friend class mistress; //дружественный класс
    // теперь у mistress есть доступ к членам monstr
};

class mistress{
public:
```

```
    void func(monstr& M) {
        std::cout << M.ammo;
    }
};

//внешняя функция
int steal_ammo(monstr &M) {return --M.ammo;}

//дружественная функция из класса hero
void hero::kill(monstr &M) {
    M.health = 0;
    M.ammo = 0;
}
```

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса.

18 Пример перегрузки префиксного и постфиксного инкремента (декремента).

Инкремент — увеличение переменной на 1.

Декремент — уменьшение переменной на 1.

Префиксный — до переменной (`++x`, `--y`). В этом случае переменная будет изменена до того, как будет использована.

Постфиксный — после переменной (`x++`, `y--`). В этом случае переменная будет изменена уже после ее использования программой.

Перегрузка операторов позволяет определить действия, которые будет выполнять оператор. Перегрузка подразумевает создание функции, название которой содержит слово `operator` и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса. Перегрузить можно только те операторы, которые уже определены в `C++`. Создать новые операторы нельзя.

Ограничения перегрузки:

- Перегруженные операции должны иметь как минимум один операнд (аргумент) типа, определяемого пользователем (чтобы предотвратить перегрузку операций, работающих со стандартными типами).
- Нельзя использовать операцию в такой манере, которая нарушает правила синтаксиса исходной операции (то есть нельзя `%` заставить работать с одним операндом)
- Не допускается изменение приоритетов операций.
- Нельзя определять новые символы операций.

Пример перегрузки постфиксного инкримента:

Заметим, что функция перегрузки постфиксного операнда использует фиктивную переменную (или «фиктивный параметр»). Этот фиктивный целочисленный параметр используется только с одной целью: отличить версию постфикс операторов инкремента/декремента от версии префикс.

```
1  #include <iostream>
2  #include <stdarg.h>
3  using namespace std;
4
5  class Test {
6          int a_;
7  public:
8          Test(int a = 0) : a_(a) {};
9
10         void print() const {
```

```
                cout << this->a_ << endl;
        }

        Test& operator++() { // перегрузка ++x
                this->a_++;
                return *this;
        }

        Test& operator++(int) { // перегрузка x++
                Test temp(this->a_);
                ++(*this); // увеличиваем *this
                return temp; // возвращем то, что было до вызова ++(*this)
        }
};

int main(void) {
        Test x;
        x.print();
        ++x;
        x.print();
        x++;
        x.print();
}
```

ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАЦИЙ

ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАЦИЙ

То же самое можно сделать не внутри класса, а внутри дружественных функций.
(Посмотреть в лекции)
bool operator != (const monstr a, const monstr b) {
return (a.health != b.health);
}
Тип возвращаемого знаечения у каждого оператора свой

19 Классы: простое наследование. Примеры.

Наследование — это механизм создания нового класса на основе уже существующего.

Класс, от которого наследуют, называется родительским (или «базовым», «суперклассом»), а класс, который наследует, называется дочерним (или «производным», «подклассом»).

Наследование позволяет:

- добавлять новые возможности в существующий класс (например, в базовый класс массива можно добавить арифметические операции);
- добавлять данные, которые представляет класс (например, взять за основу базовый класс строки, и породить класс, в котором добавлен член данных, представляющий цвет, который будет использоваться при выводе строки на экран);
- изменять поведение методов класса (от класса Passenger обслуживания пассажиров породить класс VIP обслуживания)

Для **порождения** нового класса на основе существующего используется следующая общая форма:

```
class Имя : МодификаторДоступа  ИмяБазовогоКласса { ОбъявлениеЧленов; };
```

(`МодификаторДоступа` может принимать значения `public`, `private`, `protected` либо отсутствовать, по умолчанию используется значение `private`).

Варианты наследования:

- открытое;
- защищенное;
- закрытое.

При защищенном наследовании публичные и защищенные поля родительского класса являются защищенными и доступны его «внукам» — классам, унаследованным от производного класса.

При закрытом наследовании – они доступны только самому производному классу.

Хорошая таблица-иллюстрация:

Модификатор наследования	Модификатор доступа		
	public	private	protected
public-наследование	public	private	protected
private-наследование	private	private	private
protected-наследование	protected	private	protected

Открытое

Открытое наследование является наиболее общей формой и оно моделирует отношения является (`is-a`).

При общем наследовании порожденный класс имеет доступ к наследуемым членам базового класса с видимостью `public` и `protected`. Члены базового класса с видимостью `private` — недоступны.

Общее наследование означает, что порожденный класс — это подтип базового класса. Таким образом, порожденный класс представляет собой модификацию базового класса, которая наследует общие и защищенные члены базового класса.

Пример:

```
#include <iostream>
#include <stdarg.h>
using namespace std;

class Test {
        int a_;
public:
        Test(int a = 0) : a_(a) {};

        virtual void print() const {
                cout << this->a_ << endl;
        }

        Test& operator++() { // перегрузка ++x
                this->a_++;
                return *this;
        }

        Test& operator++(int) { // перегрузка x++
                Test temp(this->a_);
                ++(*this); // увеличиваем *this
                return temp; // возвращем то, что было до вызова ++(*this)
        }

        int get_a() { return this->a_; }
};

class Lol : public Test {
        int b_;
```

```
public:
        Lol(int a = 0, int b = 0) : Test(a), b_(b) {};
        void print() const {
                this->Test::print();
                cout << this->b_ << endl;
        }
};


int main(void) {
        Test x(1);
        x.print();
        Lol y(2, 3);
        y.print();
}
```

Пример:

```
#include <iostream>
using namespace std;

class student
{
protected:
    char fac[20];
    char spec[30];
    char name[15];
public:
    student(char *f, char *s, char *n);
    void print();
};

student::student(char *f, char *s, char *n)
{
    strcpy_s(fac, strlen(f) + 1, f);
    strcpy_s(spec, strlen(s) + 1, s);
    strcpy_s(spec, strlen(n) + 1, n);
}

void student::print()
```

```
{
    cout << endl << "fac: " << fac << "spec : " << spec << " name : " <<
    ↪  name;
}

class grad_student : public student
{
protected:
    int year;
    char work[20];
public:
    grad_student (char *f, char *s, char *n, char *w, int y);
    void print();
};

grad_student::grad_student(char *f, char *s, char *n, char *w, int y) :
↪  student(f, s, n)
{
    year = y;
    strcpy_s(work, strlen(w) + 1, w);
}

void grad_student::print()
{
    student::print();
    cout << " work : " << work << " year : " << year;
}

int main()
{
    student s("ДПМ","Математика","Сидоров Иван");
    grad_student stud("ДКИ", "Сети","Иванов Петр", "Курсовая",2020);
    student *p = &s;
    p->print();
    grad_student *gs = &stud;
    student *m;
    gs->print();
    m = gs;
    m->print();
```

```
    return 0;
}
```

Функция main создаёт объект класса `student` и объект класса `grad_student`. Указателю `p` присваивается адрес объекта класса `student`, а указателю `s` объект класса `grad_student`. При этом функции вывода корректно отрабатываются для своего класса. Но что происходит, когда мы присваиваем указателю класса `student` ссылку на объект класса `grad_student`? В этом случае происходит преобразование указателей, и в строке вызывается уже функция `print()` класса `student`.

Неявные преобразования между порожденным и базовым классами называются предопределенными **стандартными преобразованиями**:

- объект порожденного класса неявно преобразуется к объекту базового класса;
- ссылка на порожденный класс неявно преобразуется к ссылке на базовый класс;
- указатель на порожденный класс неявно преобразуется к указателю на базовый класс.

Конструкторы и деструкторы:

В `C++` конструкторы и деструкторы не наследуются. Однако они вызываются, когда дочерний класс инициализирует свой объект. Если и у базового и у производного классов есть конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы — в обратном порядке. То есть если `A` — базовый класс, `B` — производный из `A`, а `C` — производный из `B` (`A-B-C`), то при создании объекта класса `C` вызов конструкторов будет иметь следующий порядок:

1. конструктор класса A;
2. конструктор класса B;
3. конструктор класса C.

Вызов деструкторов при удалении этого объекта произойдет в обратном порядке:

1. деструктор класса C;
2. деструктор класса B;
3. деструктор класса A.

Поскольку базовый класс «не знает» о существовании производного класса, любая инициализация выполняется в нем независимо от производного класса, и, возможно, становится основой для инициализации, выполняемой в производном классе. Поскольку базовый класс лежит в основе производного, вызов деструктора базового класса раньше деструктора производного класса привел бы к преждевременному разрушению производного класса.

20 Классы: множественное наследование. Порядок вызова конструкторов. Примеры.

Множественное наследование позволяет порожденному классу наследовать элементы более, чем от одного базового класса. Синтаксис заголовков классов расширяется так, чтобы разрешить создание списка базовых классов и обозначения их уровня доступа:

```
class X
{...};
class Y
{...};
class Z
{...};

class A : public X, public Y, public Z
{...};
```

Для доступа к членам порожденного класса, унаследованного от нескольких базовых классов, используются те же правила, что и при порождении из одного базового класса. Проблемы могут возникнуть в следующих случаях:

- если в порожденном классе используется член с таким же именем, как в одном из базовых классов;
- когда в нескольких базовых классах определены члены с одинаковыми именами.

В этих случаях необходимо использовать оператор разрешения контекста для уточнения элемента, к которому осуществляется доступ, именем базового класса.

Порядок выполнения конструкторов при порождении из нескольких классов следующий:

- конструкторы базовых классов в порядке их задания;
- конструкторы членов, являющихся объектами класса;
- конструктор порожденного класса.

Деструкторы вызываются в порядке обратном вызову конструкторов.

Так как объект порожденного класса состоит из нескольких частей, унаследованных от базовых классов, то конструктор порожденного класса должен обеспечивать инициализацию всех наследуемых частей. В списке инициализации в заголовке конструктора порожденного класса должны быть перечислены конструкторы всех базовых классов.

```
class Human
{
private:
    std::string m_name;
    int m_age;
public:
    Human(std::string name, int age) : m_name(name), m_age(age) {}
    std::string getName() { return m_name; }
    int getAge() { return m_age; }
};

class Employee
{
private:
    std::string m_employer;
    double m_wage;
public:
    Employee(std::string employer, double wage) : m_employer(employer),
    ↪ m_wage(wage) {}
    std::string getEmployer() { return m_employer; }
    double getWage() { return m_wage; }
};

// Класс Teacher открыто наследует свойства классов Human и Employee
class Teacher: public Human, public Employee
{
private:
    int m_teachesGrade;
private:
    int m_teachesGrade;
public:
    Teacher(std::string name, int age, std::string employer, double wage, int
    ↪ teachesGrade) : Human(name, age), Employee(employer, wage),
    ↪ m_teachesGrade(teachesGrade) {}
};
```

21 Классы: виртуальные функции. Примеры.

Виртуальные функции — специальныи вид функции-членов класса.

Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функции это происходит во время выполнения программы.

Для объявления виртуальнои функции используется ключевое слово `virtual`.

Функция-член класса может быть объявлена как виртуальная, если

1. класс, содержащий виртуальную функцию, базовый в иерархии порождения;
2. реализация функции зависит от класса и будет различной в каждом порожденном классе.

Виртуальная функция — это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через **указатель или ссылку на базовыи класс**.

Конструктор и деструктор:

Если деструктор в БК не виртуальный, то вызовется только деструктор, соответствующий типу указателя (хотя он фактически может указывать на другой тип!)

При наличии виртуальных деструкторов, если указатель указывает на объект производного типа, то вызовется деструктор производного класса. Применение виртуальных деструкторов гарантирует корректный вызов деструкторов.

Конструкторы: не могут быть виртуальными. Всегда вызывается конструктор ПК, который, в свою очередь, вызывает конструктор БК.

Затраты:

- Размер каждого объекта увеличивается на значение, необходимое для хранения адреса.
- Для каждого класса компилятор создает таблицу (массив) адресов визуальных функций.
- При каждом вызов функции выполняется дополнительный шаг поиска адреса в таблице.
- Несмотря на то, что невиртуальные функции эффективнее виртуальных, они не обеспечивают динамического связывания.

Пояснение:

Связывание — это интерпретация вызова функции в исходном коде в виде выполнения определенной части кода.

Связывание, выполняемое на этапе компиляции, называется статическим (ранним) связыванием.

Когда компилятор не знает, с объектом какого тип собирается работать пользователь, он генерирует код, который позволяет выбирать нужный виртуальный метод во время работы программы. Этот процесс называется динамическим (поздним) связыванием.

```
#include <iostream>

using namespace std;

class X
{
protected:
    int i;
public:
    void seti(int c) { i = c; }
    virtual void print() { cout << endl << "class X : " << i; }
};
class Y : public X // наследование
{
public:
    void print() { cout << endl << "class Y : " << i; } // переопределение
    ↪ базовой функции
};

int main()
{
    X x;
    X *px = &x; // указатель на базовый класс
    Y y;
    x.seti(10);
    y.seti(15);
    px -> print(); // класс X: 10
    px = &y;
    px -> print(); // класс Y: 15
    cin.get();
    return 0;
}
```

Восходящие и нисходящие человеческими словами?

22 Чисто виртуальные функции и абстрактные классы. Примеры.

Чистые виртуальные функции:

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придается лишь в порожденных классах.

Чистая виртуальная функция — это метод класса, тело которого не определено.

В базовом классе такая функция записывается следующим образом:

```
virtual void func() = 0;
```

Чистая виртуальные функции используются для того, чтобы отложить решение задачи о реализации функции на более поздний срок. В терминологии ООП это называется **отсроченным методом.**

Класс, имеющий по крайней мере одну чистую виртуальную функцию, называется **абстрактным** базовым классом. Для иерархии типа полезно иметь абстрактный базовый класс. Он содержит общие свойства иерархии типа, но каждый порожденный класс реализует эти свойства по-своему.

Абстрактные классы могут быть только наследуемыми, но указатели абстрактного класса создавать можно.

Функцию вычисления площади целесообразно задать чистой виртуальной функцией, которую переопределяет каждый наследуемый класс.

```
virtual double area() = 0;
```