

Тема 1. Введение в C++ (продолжение)

Ранее обсуждались основные новшества в C++ по сравнению с C. Теперь рассмотрим их чуть подробнее.

Ссылочный тип данных

Рассматривалось на 1 курсе, но тут повторим.

```
int x;
int &rx = x;      rx - синоним для x

void fval(int a)      void fptr(int *p)      void fref(int &r)
{
    a = 1;
}
int s = 0;
fval(s);             // s есть 0
fptr(&s);            // s есть 2
fref(s);             // s есть 3

void fceref(const int &r) // защита от изменения параметра
{
    int b = r; // можно
    r = 3;     // нельзя
}
```

Перегрузка операций и указатель this

operator... — специальное имя функции (внешней или члена класса).

Есть правила, что можно перегружать, что нельзя. Сохраняется приоритет и ассоциативность операции. Для функции члена класса сам класс — это первый аргумент операции. Для внешней функции ее аргументы и есть аргументы операции (один аргумент обязательно данный класс (или ссылка)).

Перегрузка операций может реализовываться либо дружественными функциями, либо в некоторых случаях как методы класса. В последнем случае первым аргументом операции является класс, от имени которого вызывается данный метод. “Присваивающие операции” перегружаются как члены класса, другие обычно как внешние функции. Некоторые операции перегружаются только как члены класса (см. справочники).

Операция может быть вызвана не только записью ее знака вместе с аргументами, но так же как и функция класса (например `a+b` или `a.operator+(b)`).

Аргументы и возвращаемые значения целесообразно задавать как (константные) ссылки с тем, чтобы лишний раз не задействовать конструктор копирования. При этом надо понимать, что возвращать ссылку можно только на объект, существующий вне контекста данной функции, поскольку локальные объекты (переменные) функции перестают существовать после завершения функции, и ссылка на них уже не будет иметь смысла.

Есть тонкость с перегрузкой `++` и `--`, поскольку они имеют префиксную и постфиксную формы. Чтобы их различать, используется фиктивный `int` аргумент.

Более подробную информацию о перегрузке операций и прочих рассмотренных вопросах можно получить из справочной литературы или поиском в сети.

`this` — указатель на класс, от которого вызывается данный метод.

```

struct A
{
    int x;
    A(int xx) { x = xx; }
    // A(int xx) { this->x = xx;}
    // A(int x) { this->x = x;}
    operator int() { return x;}
    const A & operator++() { ++x; return *this; }
    const A * ObjectPtr() { return this; }
};

```

```

A a(1), b(2), c(3);

++a;
int r = ++b;
int r = (b.operator++()).int();
int r = int(++b);
c.ObjectPtr() - это &c
b.ObjectPtr() - это &b

```

Для других примеров см. класс Number.

const методы и вообще о константных объектах

```

double a;
int i, j;
char str[20];

#define PI 3.14159265358979
const double pi = 3.14159265358979;
const double ccc = f(.....);
    a = pi; // можно
    pi = 2; // нельзя!
    a = PI;

const int *p; // или int const *p;
    p = &i; // можно
    j = *p; // можно
    *p = j; // нельзя!

int * const p = &i;
    p = &j; // нельзя!
    j = *p; // можно
    *p = j; // можно

const char * s = "the text";
    printf("%s\n", s); // можно
    s[4] = '_'; // нельзя!
    s = str; // можно

const char * const s = "the text"; // только читать!

struct A
{
    int x;
    A(int xx) { x = xx; }
    int Value() const { return x; }
    int & lValue() { return x; }
    const int & lValue() const { return x; }
};

```

```

void fun(const A & r)
{
    int z = r.x;
    r.x = 3; // !!!
    z = r.Value();
    z = r.lValue();
    r.lValue() = 25; // !!!
}

```

Полиморфизм функций и параметры по умолчанию

```
fun(3.0);
```

```

int fun();
int fun(int x);
int fun(double x);
int fun(int x, double y);           fun(1.0, 2);   ???
int fun(double x, double y);       ambiguous call

double fun(int x);   !!!!!!!!!!!

int fun(double x = 1, double y = 2);

    fun(x, y);
    fun(x);           fun(x, 2.0);
    fun();           fun(1.0, 2.0);

void print(FILE * f = stdout);
print();
print(my_file_ptr);

```

Записывается при объявлении (прототипе) функции. При отдельном определении функции писать уже не надо.

Строгая типизация указателей, различные способы преобразования указателей

C-style — (тип)объект или в C++ можно тип(объект).

static_cast, dynamic_cast, const_cast, reinterpret_cast — преобразования типов в разных ситуациях — обсудим чуть позже отдельно.

Механизм шаблонов template

```

template<typename T>   или   template<class T>

template<class T>
T abs(T x)
{
    return (x>=0) ? x : -x;
}

double r = abs(-2.4);
int s = abs<int>(3.5);

                                az = abs<double>(-3);

```

Это просто шаблон кода, у которого тип является параметром. Шаблон превращается компилятором в реальный код, когда встречается использование шаблона с каким-то фактическим типом данных. Тогда этот тип подставляется в шаблон (по сути как тестовая подстановка соответствующих слов) и получается код, который компилируется. В этот момент может проявиться ошибка компиляции, если используемый тип не согласуется с операциями, записанными в шаблоне.

Точно также можно параметризовать описания и определения классов

конец лекции 2

Статические члены класса

static

static int w; — устанавливает зону видимости в рамках файла
{ static int v; int d; ... } — сохранение и первичная инициализация в блоке
для классов — общий элемент для всех экземпляров класса

```

class A
{
public:
    int x;
    static int y;
    static void fun() { y = 5; }
};

```

```

A a, b, c;
a.x = 1;
b.x = 2;
c.x = 3;
A::y = 6;
b.y = 7;
a.y = 7;

```

```

a.fun();
A::fun();

```

int A::y; - по сути обычная переменная, но только "привязанная" к классу, должна быть определена вне объявления класса.

Пространства имен namespace

```

namespace ABC
{
    int r;
    ... прочий код ...
}

```

```

ABC::r = 1;
using namespace ABC;
r = 1;

```

```

double r;
r = 1;
ABC::r = 1;

```

```

using namespace std;

```

Управление памятью new delete

(“вызовы” конструктора и деструктора)

```

void * malloc(size_t num_of_bytes);
void free(void * ptr);

```

```

new <конструктор>
new <конструктор без параметров> [количество]
delete указатель
delete [] указатель

```

```

Number *p = new Number(1);
Number *q = new Number [128];
delete p;
delete [] q;

```

при отказе - исключение std::bad_alloc

Есть специальные операторы так называемые placement new, которые работают практически как malloc, т.е. память выделяют, но конструкторы не вызывают (дают нам возможность эти конструкторы вызывать в другой момент). Но об этом тоже не сейчас.

Исключения, try-catch блоки

Сигнал, “распространяющийся” вверх по последовательности вызовов функций и, возможно, несущий с собой некоторое значение (экземпляр класса).

Исключение вызывается (throw) явно в коде (своем или библиотечном) или операционной системой в процессе выполнения (run time exception).

Этот сигнал можно перехватить try-catch блоком.

```

if (возникла ошибка) throw 25;      try {                               int main()
                                     .....                               {
if (другая ошибка) throw 26;      } catch (int x) {                 try
                                     if (x==25) ....                     {
                                     if (x==26) ....                     .....
                                     } catch (double a) {                 } catch(...) {
                                     }                                       {
                                     }                                       }

class MyException {...};           try {
if (....) throw new MyException(....);   .....
                                         } catch (MyException * p) {
                                         cout << p->Code() << std::endl;
                                         cout << p->Message();
                                         delete p;
                                         exit(-1);
                                         }

int F(.....);
errcode = F(.....);
switch(errcode) {                   main -> f1 -> f2 -> f3 -> g -> h
    case ....                         catch                               throw
}

```

Теперь все это надо аккуратно уложить в голове и осмыслить ...

Что будет потом:

- потоковый ввод вывод;
- понятие инициализаторов класса;
- понятие наследования классов;
- понятие умных указателей;
- семантика перемещения;
- понятие универсальных функций (функторов, лямбда функций и пр.);
- библиотека STL и все, что к ней относится (контейнеры, итераторы, алгоритмы)